London Metropolitan University

Resilience of an embedded architecture using hardware redundancy

by

Victor Castano

A THESIS SUBMITTED

IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Prof. Hassan Kazemian, Chair
FLSC, London Metropolitan University

Dr. Kai Goebel, External Examiner
Amherst, NASA

Dr Vassil Vassilev, Internal Examiner
FLSC, London Metropolitan University

London, UK
December 2014

Abstract

In the last decade the dominance of the general computing systems market has being replaced by embedded systems with billions of units manufactured every year. Embedded systems appear in contexts where continuous operation is of utmost importance and failure can be profound.

Nowadays, radiation poses a serious threat to the reliable operation of safety-critical systems. Fault avoidance techniques, such as radiation hardening, have been commonly used in space applications. However, these components are expensive, lag behind commercial components with regards to performance and do not provide 100% fault elimination. Without fault tolerant mechanisms, many of these faults can become errors at the application or system level, which in turn, can result in catastrophic failures.

In this work we study the concepts of fault tolerance and dependability and extend these concepts providing our own definition of resilience. We analyse the physics of radiation-induced faults, the damage mechanisms of particles and the process that leads to computing failures. We provide extensive taxonomies of 1) existing fault tolerant techniques and of 2) the effects of radiation in state-of-the-art electronics, analysing and comparing their characteristics. We propose a detailed model of faults and provide a classification of the different types of faults at various levels. We introduce an algorithm of fault tolerance and define the system states and actions necessary to implement it. We introduce novel hardware and system software techniques that provide a more efficient combination of reliability, performance and power consumption than existing techniques. We propose a new element of the system called *syndrome* that is the core of a resilient architecture whose software and hardware can adapt to reliable and unreliable environments. We implement a software simulator and disassembler and introduce a testing framework in combination with ERA's assembler and commercial hardware simulators.

Dedicated to my parents

Acknowledgments

Special thanks to my advisor and mentor Prof. Igor Schagaev for helping me define my area of research and, more importantly, for his guidance and persistence during this process. This work would not have been possible without our endless discussions. Without doubt, this journey has been the greatest professional challenge of my career. Likewise, I consider myself very fortunate to be able to thank:

- My second advisor Dr. Nicholas Ioannides for his support and helpful suggestions;
- My third advisor Dr. Eugene Zouev for his helpful comments on System Software and for his work in the development of an assembler for ERA's architecture;
- Dr. Kai Goebel, deputy area lead of the Discovery and Systems Health Technology Area at NASA Ames Research Center and Dr. Vassil Vassilev, Senior lecturer and researcher at London Metropolitan University for investing their valuable time reading this work and for their valuable feedback and support during the viva process;
- The reviewers that approved my progression and gave me valuable feedback during the research meetings including: Dr. Boris Cogan, Dr. Anoosh Nabijou's, Dr. Shahram Salekzamankhani and Prof. Mike Brinson;
- Dr. Thomas Kaeghi for his previous research work in ERA's system software;
- Alex Petuhov, for his active participations and views in those early brainstorming sessions with Prof. Schagaev;
- Harry Benetatos, for his trust and support and for taking me on board to teach some of his modules;
- Robert and Sidney Dourmashkin for their constant caring advice and encouraging support;

- Dr. Anne Delextrat, Dr. Daniel Cohen and Dr. Jo Gills for their friendship and for the necessary coffee breaks;
- Catherine Lee, for her honest professional advice and flexibility when I most needed it.
- My parents, Carolina and Fernando for their unconditional love and for being always there.
- My partner Alicia, who has been exceptionally supportive and has given me the freedom to work without objection;
- My good friends Miguel Suarez, Francisco Hechavarría, Freddy de la Cruz, and Kasra Motamedi, who suffered my mood in so many occasions
- Alex and Josephine Mitrides for showing me what hard work really means

This research was supported in part with a Vice-chancellor's Research Scholarship awarded by the Faculty of Life Sciences and Computing at London Metropolitan University.

Contents

List of Figures

List of Tables

List of Equations

Nomenclature

ARQ	Automatic repeat request
ASIC	Application-specific integrated circuit
ATPG	Test pattern generation tools
ASW	Application software
BCH	Bose-Chaudhuri-Hocquenghem
BEC	Backward error correction
BICMOS	Bipolar complementary metal oxide semiconductor
BIST	Built-In-Self-Test
BPSG	Boron-phosphor-silicate-glass
CCD	Changed-coupled device
CED	Concurrent error detection
CM	Corrective maintenance
CMF	Common-mode failure
CMOS	Complementary metal oxide semiconductor
COTS	Commercial off-the-shelf
CUT	Circuit under test
CSP	Cold standby spare
DDD	Displacement damage dose
DEC/TED	Double bit error correction and triple bit error detecting
DFT	Design for testability
DMR	Dual-modular redundancy
DRAM	Dynamic random-access memory
DRE	Detected recoverable error
DUE	Detected unrecoverable error
DUT	Device under test
DW	Data word
ECC	Error correcting codes
EDAC	Error detection and correction codes

EDC	Error detecting codes
EEPROM	Electrically erasable programmable read-only memory
EPI	Epitaxial substrate doping
FCR	Fault containment region
FEC	Forward error correction
FIT	Failures in time
FM	Fault model
FPGA	Field-programmable gate array
FT	Fault tolerant
GAFT	Generalized algorithm of fault tolerance
GCR	Galactic cosmic ray
GDS	Gracefully degrading system
HARQ	Hybrid automatic repeat request
HW	Hardware
HSP	Hot standby spare
ICV	I_{DDQ} checkable voter
IDDQ	Quiescent power supply currents
IDE	Integrated Development Environment
Iff	If and only if
IR	Information redundancy
LET	Linear energy transfer
MBU	Multiple-bit upset
MCU	Multiple-cell upset
MTBF	Mean time between failures
MTTD	Mean time to detection
MTTF	Mean time to failure
MTTR	Mean time to repair/restore
MOS	Metal oxide semiconductor
MOSFET	Metal oxide silicon field effect transistor
MSB	Most significant bit
NIEL	Non-ionising energy loss
nMOS	N-channel metal oxide semiconductor

NMR	N-modular redundancy
ORA	Output response analyser
PCSE	Power cycle soft errors
PDF	Probability density function
PI	Primary input
PKA	Primary knock-on atom
PM	Preventive maintenance
pMOS	P-channel metal oxide semiconductor
PSF	Pattern-sensitive fault
REDWC	Recomputing with comparison
RERO	Recomputing with rotated operands
RESO	Recomputing with shifted operands
RESWO	REDWC Recomputing with swapped operands
ROM	Read-only memory
RF	Register file
RS	Reed-Solomon
RT	Real-time
RTS	Real-time systems
SAF	Stuck-at fault
SBU	Single bit upset
SDC	Silent data corruption
SEC-DED	Single error correction and double error detection
SEBO	Single event burnout
SEDR	Single event dielectric rupture
SEE	Single event effect
SEFI	Single event functional interrupt
SEFLU	Single event fuse latch upset
SEGR	Single event gate rupture
SEHE	Single event hard error
SEL	Single event latchup
SEMU	Single event multiple upset

SESB	Single event snapback
SER	Single event rate
SET	Single event transient
SEU	Single event upset
SOC	System on a Chip
SOI	Silicon on insulator
SOS	Silicon on sapphire
SPF	Single point of failure
SR	Structural redundancy
SRAM	Static random-access memory
SSW	System software
SW	Software
TID	Total ionizing dose
TBF	Time between failures
TTF	Time to failure
TTR	Time to repair
TMR	Triple-modular redundancy
TMRV	TMR system with non-perfect single voting
TR	Time redundancy
UART	Universal asynchronous receiver/transmitter
WSP	Warm standby spares

Chapter 1

Introduction

1.1. Motivation

Embedded systems are ubiquitous nowadays, built into homes, offices, bridges, medical instruments, cars, aeroplanes, and satellites and even into clothes. The market size of such systems is already larger than the one for general purpose computing. The majority of embedded systems are real-time systems (RTSs) and most RTSs are embedded in a product.

For decades, embedded RTSs are being used in fields where their correct operation is vital to ensure the safety and security of the public and the environment: from automotive systems and avionics to intensive health care and industrial control as well as military operations and defence systems. These systems are subject to time constraints and must guarantee a response within specified timing bounds. The safety critical nature of RT embedded systems employed in those fields demands the highest possible availability and reliability of system operation.

The exponential growth of clock frequency and memory size has lead to important achievements in the technological development of microprocessors.

Manufacturers of advanced silicon electronics have been able to create more complex designs by periodically scaling down the technology, increasing the transistor density. This growth is supported by the progressive miniaturization of electronic components predicted by Moore's law in 1965.

This phenomenon has also produced undesirable consequences that introduce physical limitations to the law. Due to the area reduction of electronic components to nanometre scales and due to the increase in clock frequencies (ITRS, 2011), supply voltages have been reduced to keep power dissipation manageable while thermal noise voltages have increased (Asanovic et al., 2006; Kish, 2002).

For a long time, radiation effects have been a serious concern in aviation and spacecraft electronics. As the dimensions and voltages of embedded systems are reduced, their sensitivity to ionizing particles has considerably increased. Energizing particles can produce a number of faults at the hardware level, not only in contexts with harsh environmental conditions such as outer space but also at sea level with regular conditions. Components with lower power and noise margins are less reliable and therefore recent systems are more prone to transient faults induced primarily by radiation (Baumann, 2002; R. C. Baumann, 2005; Seifert et al., 2002; Shivakumar et al., 2002). Transient faults do not cause permanent damage in circuits but can affect system behaviour by corrupting stored information or signal communication (Karnik and Hazucha, 2004; Mavis and Eaton, 2002; "JEDEC JESD89-3A," 2007).

Besides the typical stress experiments in laboratories based on particle bombarding, there is a considerable amount of evidence of radiation induced malfunctions and catastrophic failures during operation in real life environments. Radiation induced faults are frequent in space environments (Adams and Gelman, 1984; Adams et al., 1982; Binder et al., 1975; Blake and Mandel, 1986; Waskiewicz et al., 1986). The Saturn's Cassini (Swift and Guertin, 2000), Deep Space 1 (Caldwell, 1998), Mars Odyssey (Eckert, 2001) and Jupiter's Galileo (Fieseler et al., 2002) are examples of missions that presented

malfunctions as a result of cosmic rays. The satellites X-ray Timing Explorer (Poivey et al., 2004), Gravity Probe B (Owens et al., 2006), TOPEX/Poseidon (Swift and John, 1997) and GRACE (Pritchard et al., 2002) have also reported anomalies during operation.

Radiation Induced faults are also present to a lesser extent in atmospheric (Taber and Normand, 1993) and terrestrial environments (Hauge et al., 1996; Normand et al., 2010; Ziegler, 1996).

Due to the reasons stated above there is an increasing need to deal with faults. There are two classes of mechanisms to deal with them: fault avoidance and fault tolerance (FT) (Avizienis et al., 2004). Fault avoidance means developing components/systems that are less likely to present faults while fault tolerance techniques focus on the system's ability to tolerate the effects of these faults. Fault-tolerance is defined as *the ability to provide uninterrupted service, conforming to the desired levels of reliability even in the presence of faults* (Avizienis et al., 2004). Applications of modern electronic systems require more and more mechanisms to mitigate the effect of these faults (Nicolaidis, 2010).

Complete avoidance of faults in a system is practically impossible and hence a balance of the two approaches is currently applied.

The research community mainly focuses on a) identifying all possible mechanisms leading to accidents and b) on providing pre-planned defence techniques against them. However, too little research effort has been employed towards systems that can respond to deviations from desirable states.

The research is driven by observations of limitations from the evolution of computer architectures, which have been motivated by technological and market choices as well as physical limitations. These observations refer to performance decrease, increase in power consumption/dissipation, reliability aspects, parallelization challenges, design complexities, and hardware and software inefficiencies. A brief explanation for each follows:

Performance deceleration: Transistor density and frequency have increased to satisfy the immediate market demands. Therefore, more raw materials in the form of transistors is available for system design. However, unjustified complexity has been introduced in the current computer architectures. In recent years, clock rates of commodity microprocessors have flattened and performance of processor cores has slowed down (Asanovic et al., 2006; Hill, 2010).

Power consumption / dissipation: Scaling processor clock speed increases power consumption (and consequently power dissipation) while the die size remains the same. Therefore, the power/density ratio will keep increasing to the point where no practical technique can dissipate the generated heat.

Reliability: Performance, heat and power consumption are not the only concerns. Reliability of intra-chip communication is also affected by physical constraints. Transistor scaling shortens wire distances, which improves performance but also implies thinning of those wires. As wires become narrower, in order to reduce the resistance per unit length they also become taller. Media resistance limits the speed of electrons within. Tall wires within close distance vary dependent timing characteristics at best and produce data corruption at worst. In short, thinner wires increase delays and harm reliability. Furthermore, as explained earlier, the same radiation fluxes that in the past had no effect on electronics are now able to induce faults that affect the logic value of current transistors with lower critical charge.

For these reasons, it is commonly believed by the research community that the classic Hardware/Software uniprocessor model has reached the *power/performance wall* (Asanovic et al., 2006; Hill, 2010).

Parallelization: The microprocessor industry approach is based on using the billions of transistors (now available on a die) to a) replicate the off-the-shelf core design multiple times and to b) increase the size of caches. Nevertheless, effective programming of multi-core is not trivial and introduces multiple

challenges (Geer, 2007; Goth, 2009; Pankratius et al., 2009). As an attempt to overcome the power wall, the computer science research community has reincarnated parallel computing. Parallel computing and parallel programming are not new; they have been a mainstay in high-performance since the early 50s (Hill and Rajwar, 2001).

Complexity: The semiconductor industry, driven by economic reasons and time-to-market needs, has introduced unjustified complexity in microprocessor designs.

Software and Hardware Inefficiency: In terms of software, modular programming (Turski and Wasserman, 1978; Wirth, 1983) and later object oriented programming (Wirth, 1992, 1988) were introduced to maximize performance and effectiveness of the human agent in the programming process. To maximize performance of HW/SSW/ASW, several approaches of parallelism using distributed, dataflow and cluster architectures were introduced in the late 50s. The Flynn diagram (Flynn, 1972) is still in use: SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data) and MISD (Multiple Instruction Single Data) are very well known architectures, each with their own benefits and drawbacks. In the early 80s the VLIW (Very Long Instruction Word) (Fisher, 1983) approach was also introduced. But since then, no significant new architecture has been introduced.

To make the next step in the design of special systems for safety critical applications we should analyse what is applicable from the well-developed theory and design of fault tolerant systems since early 70's, in particular their reliability and resilience to electromagnetic impulses. In turn, the success of future computer systems for safety critical applications will depend on trading-off performance, reliability and power consumption.

The combination of the following two statements forms a framework for this research. At first, we should analyse the technological achievements of modern electronics in terms of performance. Finally, we should find ways to improve the

efficiency of current embedded systems in terms of performance, reliability and power consumption.

1.2. Scope and Contribution

This work relates to techniques that improve the reliability of embedded systems with regards to permanent and transient hardware faults induced by radiation. However, these techniques are also efficient to mitigate the effect of faults induced by other means. Note that software faults as the source of errors are out of the scope of this thesis. This section briefly explains these contributions.

The main goal of this research was to find efficient techniques and original mechanisms to improve the reliability, performance and energy use of real time systems in safety-critical applications. This includes the design, development and analysis of a fault tolerant reconfigurable architecture in presence of radiation-induced faults. Such architecture will be further used as a core element for reconfigurable computers with key requirements for reliability, power awareness, performance and scalability.

This research is an attempt to overcome known drawbacks of modern RTS. The outcomes of such work can be summarized as follows:

- The traditional Reliability, Fault Tolerance and Dependability concepts and definitions do not take into account the transient nature of some of the faults induced by radiation. The result is a new concept of resilience that takes into account the changing nature of environment and the different FT contexts.
- We provide a systematic examination of the physical mechanisms that lead to faults induced by radiation and the error process. The result is an comprehensive taxonomy of radiation-induced effects in modern microprocessor technologies;

- We develop a fault model that contains an extensive taxonomy of faults that can assist in the serviceability and coverage attributes of fault tolerant and resilient system designs.
- We introduce a novel combination of structural hardware elements at the active, passive and interfacing zones. In combination with system software, these hardware elements can improve the resilience of a system with a better compromise in silicon area, reliability, power and performance that known fault tolerant systems. We design and implement a hardware prototype as a proof-of-concept.
- We develop a framework and testing scheme for the testing and debugging of the hardware prototype. As part of the framework, we implement an assembler for the hardware prototype together with a disassembler and simulator tool.

The research is part of a joint research effort performed internationally (so-called Evolving Reconfigurable Architecture (Schagaev et al., 2010). Theoretical development and hardware testing of RA will provide the hardware prototype platform for testing hardware reconfigurability.

1.3. Structure

This thesis is divided in seven chapters configured as:

- Chapter 1. *Introduction*: this first chapter summarizes the approach of this doctoral thesis, describes its contribution to science and defines its general structure.
- Chapter 2. *Resilience*: in this chapter we provide part of the theoretical framework of reliability. We analyse the properties of classic dependability and we describe our own view of the concept of resilience.
- Chapter 3. *Dealing with faults - redundancy*: this chapter provides a complete review of state-of-the art techniques employed to deal with faults and explores the different types of redundancy and fault tolerant techniques.
- Chapter 4. *Impact of radiation in electronics of embedded systems*: This chapter studies the physical mechanisms of radiation as the primary phenomenon that causes faults in current computing systems. We also analyse their effect on semiconductors at low, circuit and system levels.
- Chapter 5. *Fault tolerance models*: We analyse a model of hardware faults. We introduce GAFT and define the different states and actions required to implement fault tolerance.
- Chapter 6. *Hardware support and System Software Support for Resilience*: This chapter details the hardware and system software elements of a novel resilient architecture that can achieve various levels of performance, reliability and energy consumption.
- Chapter 7. *Implementation: Hardware Prototype, Simulation and Testing*: This chapter focuses on the development and testing of the hardware prototype. Details of the design and development of a software simulator of the hardware architecture are also provided.
- Chapter 8 summarizes and concludes this work.

Chapter 2

Chapter 2

Resilience

This chapter provides a background of necessary concepts in the field of fault tolerance and resilience. First, we introduce the system failure lifecycle and describe the main threats to resilience. Then, the concept of resilience and its attributes and measures are reviewed. We explain our own view of the performance and reliability problems that the microprocessor industry is currently facing. The classic theory of reliability is presented and an explanation is given on how the hardware components of an embedded system can be made more resilient to hardware faults. We review the classic mathematical definition of reliability and show how to calculate the reliability of a system depending on the topology of its components. Other attributes of resilience including safety, performability, integrity, maintainability and availability are also reviewed. Finally, we extend the definition of resilience and apply it to the field of safety critical computing.

2.1. System failure lifecycle

Correct service (Laprie, 1995) also named *proper service* (Laprie and Avizienis, 1986) is delivered by a system when the service implements the function as specified. The fundamental threats to the correct service and to the resilience of safety critical systems are faults, errors and failures that, in turn, can cause catastrophic failures. Among these four terms there is a causal effect relationship.

A *failure*, service failure or system failure is an event that takes place when the delivered service deviates from proper service. Hence, a service failure implies a transition of the system from proper service to an *improper service*, not implementing the functions as specified by the functional specification of the system. The downtime or period of delivery of improper service is also referred to as *service outage*. The transition from improper service to proper service is called *service restoration*, service *recovery* or *repair*.

Since a service is a sequence of the external states of a system, a service failure takes place when one or more of its external states deviate(s) from the correct service state. These deviations are *errors*. An error is a part of the system state that is liable to lead to a subsequent failure. The hypothesized or adjudged cause of such error is a fault.

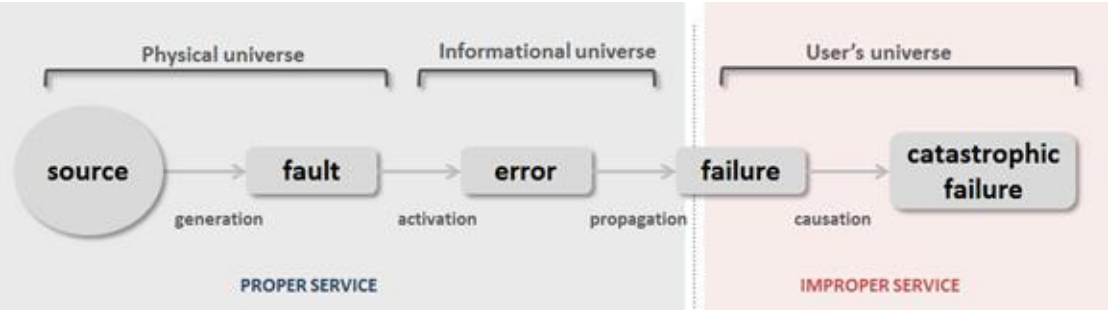


Figure 2-1. System failure lifecycle within a three universe model

A *fault* is a weakness, blemish or shortcoming of a particular hardware component or unit. An error is the manifestation of a fault, a deviation from

accuracy or correctness. Finally, if the error leads to one of the system's functions being performed incorrectly then a failure has occurred.

Figure 2-1 graphically describes the well-known lifecycle of system failure within a three universe model (Johnson, 1989) adapted from the four universal model originally developed in (Avizienis, 1982). In the first universe, the physical one, faults are generated due to various sources. Faults can activate errors within the second universe, the informational one. Errors take place when some information units become incorrect. In turn, errors could propagate the user universe and lead to a failure. It is in this final universe, where the user can witness the effects of faults and errors in the form of failures. One or more failures could potentially cause a catastrophic failure in the case of safety critical systems.

The arrows between the entities in Figure 2-1 correspond to latencies. *Fault latency* (activation latency in Figure 2-1) is the time length between the occurrence of a physical fault and the appearance of an error. Likewise, *error latency* is the length of the propagation time that takes place between the activation of the error and the manifestation of the failure.

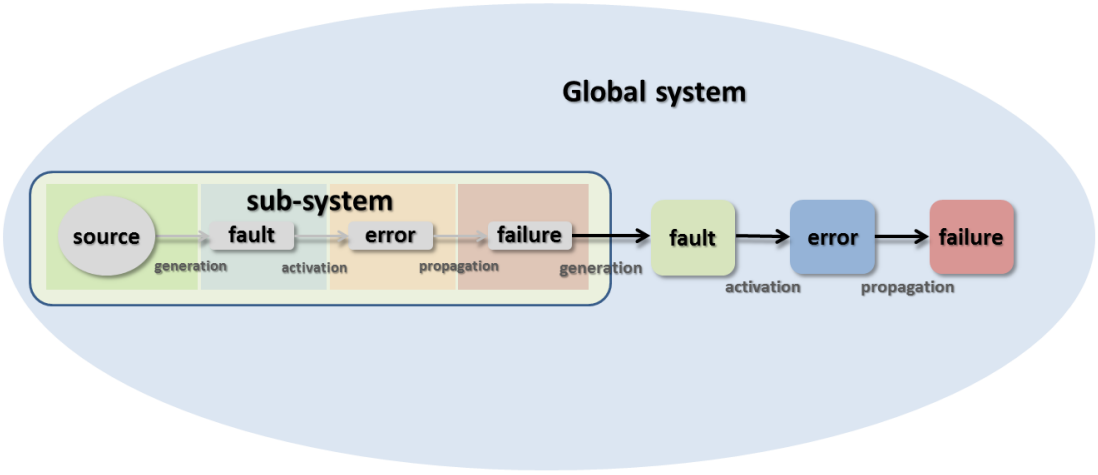


Figure 2-2. Failure-fault transition between different levels of a system

The term fault and failure is sometimes unclear in reliability literature. In this thesis, the term *fault* is sometimes equivalent to failure. For instance, a *system fault* can be the same as a *component failure*. Figure 2-2 shows the fault-failure transition between a subsystem and a global subsystem. The fault-failure cycle can be applied at different levels of abstraction within a system; consider a transistor as a subsystem that is part of a more global system (e.g. memory cell): the occurrence of incorrect functionality of the transistor during normal operation (e.g. the effects of aging and stress) is a subsystem failure of such component but may lead to, for instance, a logic fault (global system fault). This logic fault will remain dormant unless is activated, producing an error, which is likely to propagate and create other errors. If the correct service of that global system is affected, a global system failure occurs. The same subsystem-system transition can take place between the memory cell, the memory circuit that the cell is part of, the microprocessor system that can be part of a multiprocessor, etc.

2.2. Resilience: Attributes and measures

The word *resilience* (from the Latin origin *resilire*, to jump back, or to rebound) is literally the tendency, ability, act or action of springing back, and thus the ability of a body to recover its normal shape and size after being pushed or pulled out of shape. That is, the ability to recover to normality after a disturbance, shock or deviation from the intended state and go back to a pre-existing or acceptable or desirable, state.

The meaning of resilience is different between authors. Hollnagel defines resilience as (Hollnagel et al., 2012):

> *"The intrinsic ability of a system to adjust its functioning prior to, during, or following changes and disturbances, so that it can sustain required operations under both expected and unexpected conditions"*

The US Department of Defense (DoD) defines a resilient system as (Neches, 2012):

> *"A resilient system is trusted and effective out of the box in a wide range of contexts, easily adapted to many others through reconfiguration or replacement, with graceful and detectable degradation of function"*

The Keck Institute for Space Studies has also made a big effort studying the attributes of Resilience. During its study (Murray et al., 2013) numerous definitions were proposed and discussed.

The term *Resilient* has been traditionally used essentially as a **synonym** of fault-tolerant (Laprie, 2008). Before we discuss fault tolerance as a concept and review the resilience concept, several other terms need to be defined.

One of them is *Dependability*, which is an integrative concept that encompasses many other quantitative and qualitative attributes. Laprie (Laprie et al., 1992) defines dependability as the "trustworthiness of a computer system such that reliance can be justifiably placed on the service that it delivers".

Dependability is the ability to deliver a service that can be trusted justifiably. Laprie defines the service delivered by the system, as its behaviour as it is perceptible by is user(s); a user is another system (physical or human) which interacts with the former. Such service is classified as "proper" or "correct" if it is delivered as specified; otherwise it is considered as "improper" or "incorrect" (Laprie and Avizienis, 1986). Again, the "properness" or "correctness" of the system service depends on the viewpoint of the user.

The terms covered by dependability have been re-defined over the years (Avizienis et al., 2004). We merge and organize the attributes or measures of dependability and adapt them to the field of safety-critical applications. The attributes of dependability are: *reliability*, *safety*, *performability*, and *security*. The

later encloses a subset of attributes including *integrity*, *maintainability* and *availability*.

2.3. Reliability

The reliability measure is most often used to characterize systems in which failures are unacceptable; therefore, it is suitable to the field of safety critical systems.

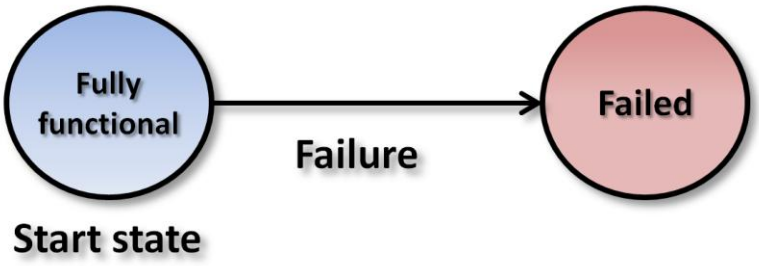


Figure 2-3. A non-repairable system with two states

Figure 2-3 shows a non-repairable system with two possible states: a fully functional start state (up) and a failed state (down), involving loss of functionality, which can be reached after a transition due to failure.

There is no disagreement about the need for reliable systems but some vague notion of reliability is not enough in safety-critical engineering. Reliability can be defined as follows: R*eliability R(t) is the probability that a system or component will perform its intended function without failure over the entire interval [0,t] under specified environmental and operating conditions*. R(t) is a probability in the sense of being a recurring event. The intended function, period of time and stated conditions are all defined as system requirements when designing a real-time system. Note that the following mathematical equations regarding reliability are based on the classical theory of reliability of (Birolini, 2007) and are not our original work.

2.3.1. Performance and Reliability

2.3.1.1. Power-reliability wall

Since the invention of the integrated circuit in 1958 each generation of semiconductor technology has exponentially decreased the transistor price and exponentially increased the transistor density per chip (Hutcheson, 2009).

This technological shrink model has led to the impressive level of technology and hardware element density recently achieved (Nair, 2002) with processor frequencies reaching up to 4.7 GHz ("Power 6 Specs: IBM Power6 Microprocessor and IBM System p 570," 2007). The higher number of transistors and the kilometres of wire operating at higher frequencies lead to several Watt/cm^2 on modern chips leading the peak energy consumption well over 140W. Most of that energy becomes heat, rising operating temperatures.

The cost of the manufacturing process of smaller feature technologies is increasing exponentially. Such cost is doubling every four years, which makes smaller nanometre technologies and the continuation of this law no only a technical challenge but an economic one.

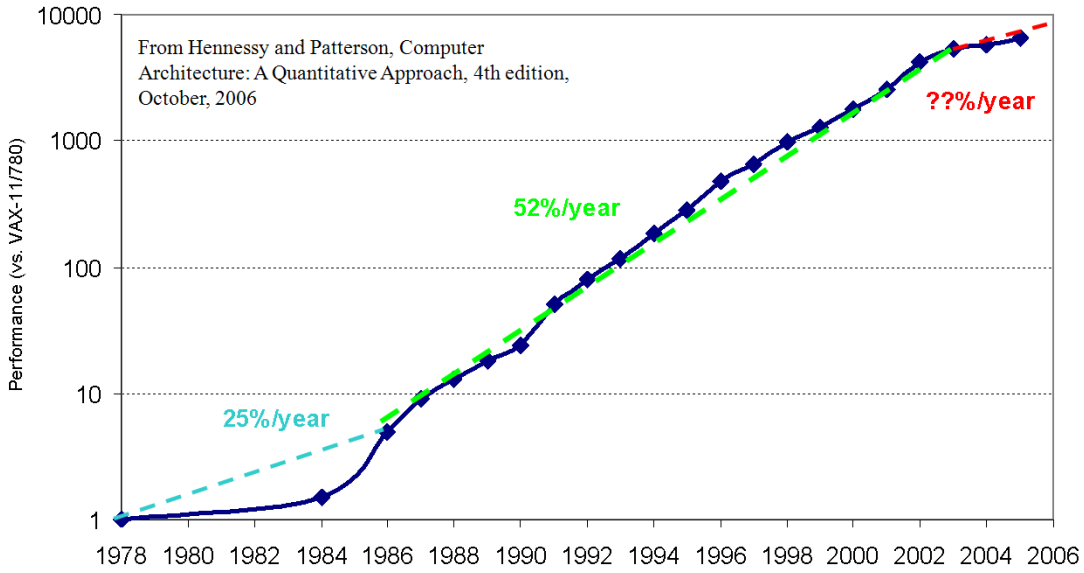


Figure 2-4. Growth in performance since the mid-1980's (Hennessy and Patterson, 2006)

Evidence of this phenomenon is the chart shown in Figure 2-4 that plots performance gap between the processor and memory of the VAX 11/780 (measured by the *SPECint benchmarks*[1]). Subsequent to the mid-1980s, processor performance growth averaged about 52% per year. Since 2002, uniprocessor performance has slowed down to about 20% per year reaching the power-reliability wall in 2006. On the other hand, memory has averaged a constant performance increase of 9%.

Using Moore's law as a measure of progress has become misleading, as improvements in transistor density no longer translate into performance and energy efficiency. Starting around the 65nm technologies, transistor scaling no longer delivers the performance and energy gains that drove the semiconductor growth during the past decades (Dreslinski et al., 2010).

The Research community and the Industry believe that parallelism is the answer to overcome the performance wall, however with different implementation approaches. The industry has attempted to react by escalating the number of processors introducing multi-core architectures and parallelism. Multiplying the number of big, complex and power demanding existing cores, (which are part of the problem) does not adequately solve any of the performance, reliability and power awareness concerns (Asanovic et al., 2006).

[1] SPECint benchmarks are a set of benchmarks design to test the integer processing performance of modern CPU (http://www.spec.org/)

2.3.1.2. Reliability within the vicious cycle

What follows is an attempt to interpret the current performance and reliability issues of the microprocessor industry. In this context, we use the term *vicious cycles*: *cycles* as chains of events that reinforce themselves in a feedback loop. The term *vicious* is used, as the results of such chains are detrimental.

The semiconductor industry driven by economic reasons and time-to-market needs has introduced too much complexity in microprocessor designs. Figure 2-5 shows our interpretation of the reliability problem in current computing.

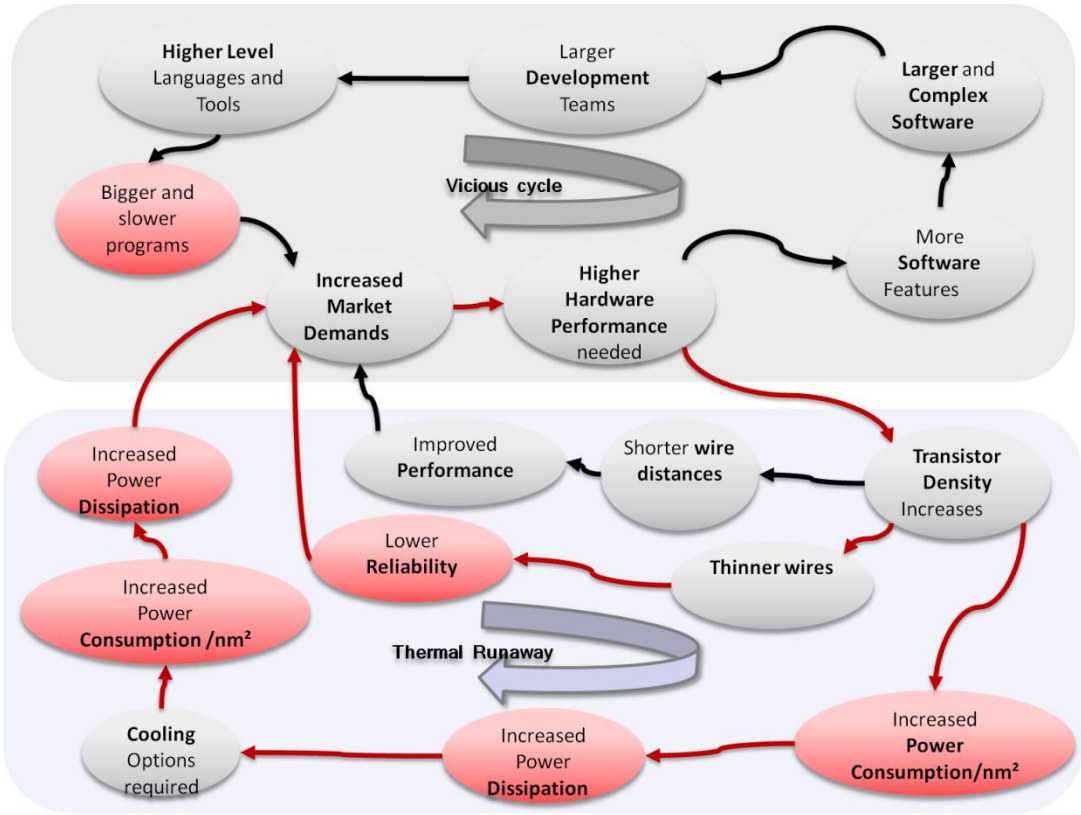


Figure 2-5. The Vicious Cycle and the evolution of computing systems. 1950-2005

An efficient and logical design could have achieved better results in the long-term. Instead, a brute force approach, increasing frequency, deeper pipelines and cache levels (pipelines and cache levels provide slight performance at a high cost of chip floor plan) has been employed (Asanovic et al., 2006).

Increased processor performance allows software companies to develop larger and feature-richer software, which involves larger development teams. Consequently, developers need higher-level languages and abstractions, which are less efficient and generate slower programs. As a result faster processors are needed, reinforcing this vicious cycle (Figure 2-5) and generating detrimental results. Under this cycle, existing programs would run faster on the latest generation of microprocessors.

Since 2005-2006, no considerable increase in functional hardware performance has occurred. Existing programs need to be redeveloped to take advantage of the new multi-core. Consequently, the vicious cycle does not apply anymore. The power wall has dramatically slowed down the evolution of microprocessors in terms of performance.

Clearly, technological developments have not been supported by a logical evolution. There is an increasing need for unified hardware and software technologies. Development of a new computing paradigm and its implementation through the whole cycle of hardware, software and application design, development and prototyping is required.

2.3.2. Reliability and unreliability functions

Let's suppose we have a system with N identical components. We define $S(t)$ as the number of surviving components at time t and $Q(t)$ as the number of failed components up to time t. Therefore:

$$S(t) + Q(t) = N$$

Equation 2.1. Surviving and failed components at time t

The *reliability R(t)* is the proportion of components that continue to perform without failure after being used for a period of time *t*. That is the probability of survival of the components, given by:

$$R(t) = \frac{S(t)}{N}$$

Equation 2.2 – Probability of survival of components at time t

Unreliability or *Cumulative failure distribution* function is generally referred to as the probability of failure. More specifically, *unreliability F(t)* is the conditional probability that the system begins to perform incorrectly during the interval $[t_0, t]$ given that the system was performing correctly at time t_0:

$$F(t) = \frac{Q(t)}{N}$$

Equation 2.3. Probability of failure of components at time t

Based on Equation 2.1:

$$R(t) + F(t) = 1$$

$$F(t) = 1 - R(t)$$

Equation 2.4 – Reliability and Probability of failure of components at time t

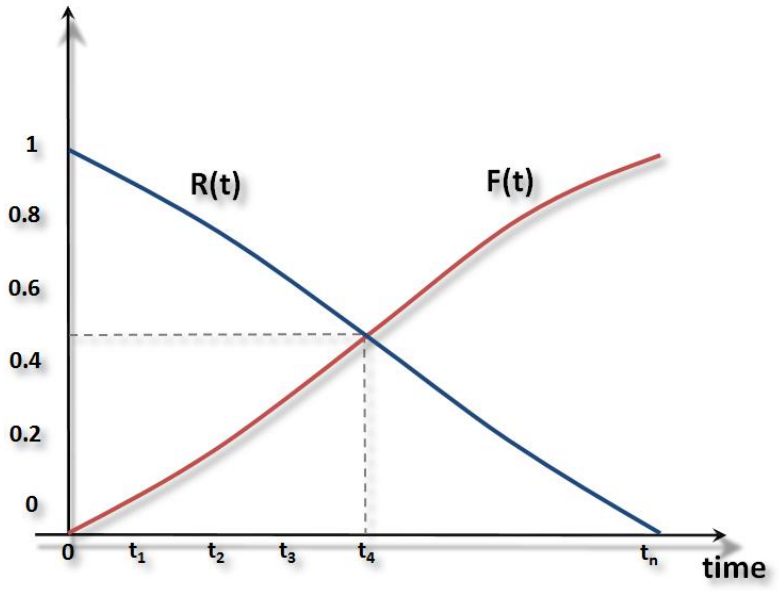


Figure 2-6. Reliability R(t) and Failure probability F(t) functions over time t

Figure 2-6 shows a graph of the reliability and failure probabilities over time with a constant failure rate. *R(t)* is a monotonically decreasing function that has an initial value 1 whereas *F(t)*, starting at 0, increases monotonically. The sum of *F(t)* and *R(t)* at any given time is 1.

2.3.3. Probability density function

The derivative of *F(t)* is a probability distribution function (PDF) that defines the probability of failures per unit time f(t) of a particular component that has been used for a period of time t (Birolini, 2007). Based on this definition, the *probability density function* is described as:

$$f(t) = \frac{dF(t)}{dt}$$

Equation 2.5. Probability density function as a function of Unreliability

Using Equation 2.4:

$$f(t) = \frac{d[1 - R(t)]}{dt} = -\frac{dR(t)}{dt}$$

Equation 2.6. Probability density function as a function of Reliability

Thus, the probability of a failure during the time range [0,t] is:

$$F(t) = \int_0^t f(t)dt$$

Equation 2.7. Probability of failure during the time range [0,t]

Using Equation 2.4:

$$R(t) = 1 - F(t) = 1 - \int_0^t f(t)dt \ = \int_t^\infty f(t)dt$$

Equation 2.8 – Reliability during the time range [0,t]

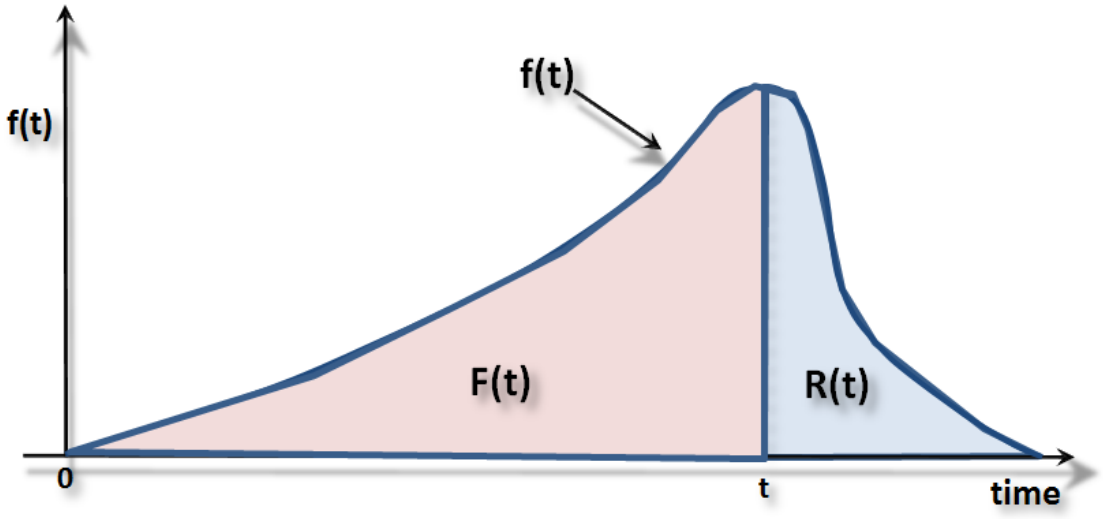


Figure 2-7. Representation of Reliability, Unreliability and the probability density function

Figure 2-7 is a schematic that illustrates the relationship between the unreliability or probability of failure (area in red), and probability of success (area in blue) and the PDF. In this schematic only two mutually exclusive states can occur: failure or success. F(t) and R(t) are the probability of these two states and the sum of these two is always equal to 1.

2.3.4. Failure rate function

The failure rate function $\lambda(t)$ (also known as *momentary failure rate* or *hazard function*) describes the number of failures per unit of time versus the number of components still operating at a time (surviving components) (Birolini, 2007):

$$\lambda(t) = \frac{1}{S(t)} \frac{-dQ(t)}{dt}$$

Equation 2.9. Failure rate as failures vs components at time t

Using Equation 2.3 and Equation 2.2:

$$\lambda(t) = \frac{1}{N\,R(t)} \frac{N\,dF(t)}{dt}$$

$$\lambda(t) = \frac{1}{R(t)} \frac{dF(t)}{dt} = \frac{f(t)}{R(t)}$$

Equation 2.10. Failure rate as a function of reliability and probability density

The failure rate function is very accurate to express the reliability of semiconductor components for long periods. However, calculating the failure rate at a specific point of time within a short period is impractical. Consequently, *average failure rate*, with longer time periods, is preferred:

$$Average\ failure\ rate = \frac{Total\ failures\ during\ a\ period}{total\ operating\ time\ within\ a\ period}$$

Equation 2.11. Average failure rate

The values of average failure rate can be expressed by *%* or *ppm*[2]. However, FIT[3] it is more widely used as a unit for reliability.

2.3.5. Cumulative hazard function

Using Equation 2.4:

$$\lambda(t) = \frac{-1}{R(t)} \frac{dR(t)}{dt}$$

Equation 2.12. Failure rate function as a function of reliability

[2] ppm is the abbreviation of "parts per million". One ppm means 1 faulty component out of 1000000 components. Hence, an average failure rate of 10 ppm means that there are 100 faulty components out of 1000000, or 1 component out of 100000.

[3] FIT is a unit widely used to express failure rate. One FIT equals to one failure per billion (10^{9}) hours (one failure in about 114,155 years), or 1ppm/1000h

This expression can be integrated from time 0 to time t giving the *cumulative hazard function H(t)*:

$$H(t) = \int_0^t \lambda(t)dt \ = -\int_1^{R(t)} \frac{dR(t)}{R(t)}$$

Equation 2.13. Integration of the failure rate from time 0 to t.

The limits of the integration are obtained as follows:

- at time *t=0*, *R(t)=1*
- at time t by definition the reliability is *R(t)*

Given the assumption of a constant failure rate λ of a component (typically in per million hours or *FIT*):

$$\lambda t = -logR(t)$$

$$-\lambda t = -logR(t)$$

$$R(t) = e^{-\lambda t}$$

Equation 2.14. Reliability at time t with constant failure rate λ .

2.3.6. Bathtub curve of failure rates

The following section describes the classic Bathtub Curve used in reliability engineering. In the 1950's the Advisory Group for the Reliability of Electronic Equipment discovered this typical curve, which defines the failure rate of electronic equipment.

A value can be assigned to the reliability of a system. For instance, a system may have 97% reliability over a two-year mission, subject to a maximum vibration V_{max}, a humidity range $[H_{min}, H_{max}]$ and temperature range $[15^oC, 30^oC]$. Although the above definition is generally accepted, it is not a complete definition from the starting to the end time of a safety-critical system's life. System reliability will be different for different time periods. Therefore, more factors need to be

considered. For a correct service delivery in a specific period, the system must be operating properly at the beginning of the observation period.

The operational age of the system is one of the factors that should be taken into account. The above definition does not differentiate between:

- a new system,
- a system that has been operational for a substantial amount of time and whose faults have already been corrected, and
- an old system with a long operational history and wear out issues

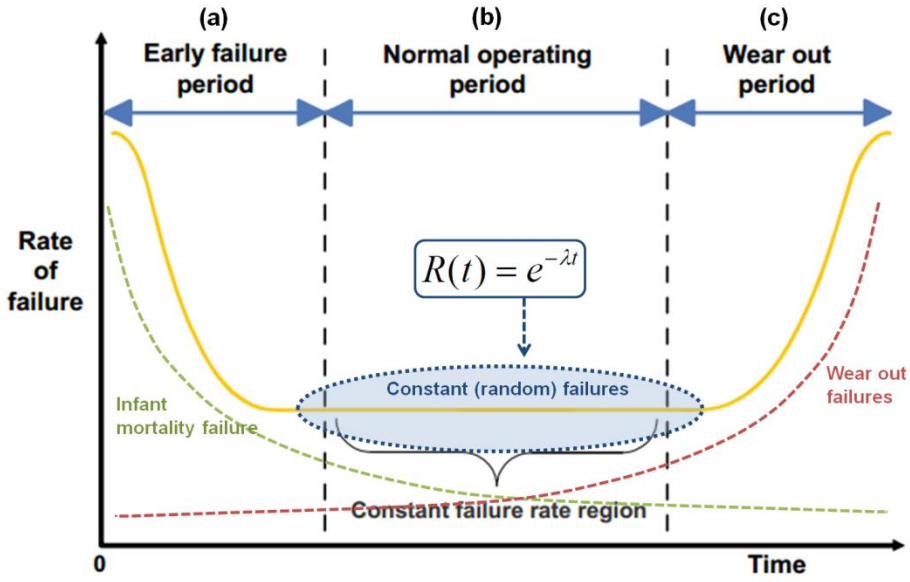


Figure 2-8. A bathtub curve of failure rates. During normal operation period the failure rate λ is constant and faults are independent

Reliability distributions with decreasing, constant and increasing failure rate as a function of time are illustrated in Figure 2-8 during period (a), (b) and (c) respectively. The assumption made is that faults are independent and that the failure rate (λ) is constant. The system failure rate is dependent on the system's lifetime constituting a function with a bathtub shape and three distinctive areas or periods: an early failure period (a), a normal operating period (b) and a wear

out period (c). For failure rates higher than the constant failure rate (λ), the chance of system failure becomes higher.

For a new system (case a) there is an *early failure* or *infant period* with a decreasing but high failure rate due to latent manufacturing defects that escape the initial testing of the product. As the products get into operation, these defects surface quickly when the devices are stressed. Once the infant failures are eliminated, this high failure rate rapidly decreases to an almost constant value during the *normal operating* or *grace period* (case b). This long period represents the useful life of the system where failures occasionally occur due to the sporadic breakdown of weak components. It is highly desirable that this period of low failure rate and high reliability dominates the product's lifetime.

During the *wear out* or *breakdown period* (case c) the reverse situation takes place. As the system gets older, the failure rate increases sharply due to age-related wear out. Note that many devices that form part of the same system will initiate this phase roughly at the same time. This could create an avalanche effect that could critically decrease the overall reliability of the system.

After analysing the bathtub curve and the three periods of operation involved, it is clear that the previous equations of reliability only suit the normal operating period with a constant failure rate. This curve represents very well hardware reliability due to aging and degradation but it is not suitable to software, especially in the case of versioning and upgrades. The silicon failure mechanisms will be further studied in Chapter 4.

2.3.7. Mean time between failures (MTBF)

Instead of a monotonic function of time reliability can also be expressed as a numeric index. Mean time between failures (MTBF) is *the average time that the system will run between failures (Garland and Stainer, 2013)*. This measure is convenient to compare the reliability of different repairable systems. MTBF can be estimated by averaging the time between failures, including any additional

time required to repair the system and place it back to a functional state. The equations in this section are obtained from (Birolini, 2007).

Being f(t) the probability of failure per unit time, MTBF can be described by:

$$MTBF = \int_0^\infty tf(t)dt$$

Equation 2.15. Probability of failure per unit time (MTBF).

Using Equation 2.6:

$$MTBF = -\int_0^\infty t\frac{dR(t)}{dt}$$

Equation 2.16

Integrating the above equations by parts we obtain:

$$MTBF = -[tR(t)]_0^\infty \int_0^\infty R(t)dt$$

Equation 2.17

For $t = 0$, $R(t) = 0$, hence $t \times R(t)=0$. As t increases from 0, $R(t)$ decreases. As t tends to ∞, $t \times R(t)$ tends to zero. Therefore, the first term of the previous equation is zero. For any kind of failure distribution with a failure rate λ as a function of time, the general expression for MTBF can be described as:

$$MTBF = \int_0^\infty R(t)dt$$

Equation 2.18. General expression of MTBF

The higher the MTBF is, the higher is the reliability of the system or component. Moreover, for failure distributions independent of time with a constant rate, MTBF is given by:

$$MTBF = \int_0^{\infty} e^{-\lambda t} dt$$

Equation 2.19. MTBF for time independent failure distributions with constant rate of failures

$$MTBF = \frac{1}{\lambda} \left[e^{-\lambda t} dt \right]_0^{\infty} = \frac{1}{b}$$

Equation 2.20. MTBF for time independent failure distributions with constant rate of failures

Hence, MTBF of a system is reciprocal to its failure rate (given a constant failure rate). MTBF will be expressed in hours if the constant rate is also expressed in hours.

2.3.8. Mean time to failure (MTTF)

As described above, MTBF is a good measure of reliability for systems that can be repaired. A similar single-parameter indicator of reliability for components that cannot be repaired is the mean time to failure (MTTF). MTTF is the average time until the first system's failure. Results of life testing can be used to calculate MTTF by testing a set of N identical units until all of them have failed with the time to the first failure of the individual units identified as *t1, t2, t3, ...,* tn. It can be observed that MTTF is given by:

$$MTTF = \frac{1}{\lambda} \sum_{i=1}^{n} t_i$$

Equation 2.21. Mean time to failure (MTTF)

As before, the failure rate, if independent of time, can be calculated by:

$$\lambda = \frac{1}{MTTF}$$

Equation 2.22. Failure rate as the inverse of MTTF

MTBF and MTTF are sometime used interchangeably. Although the numerical difference is small in many cases, both measures represent different concepts. MTTF is related to MTBF but does not include the repair time (MTTR or mean time to repair/restore) nor the detection time (MTTD or mean time to detection):

$$MTBF = MTTF + MTTD + MTTR$$

Equation 2.23. MTBF as a function of MTTF, MTTD and MTTR

MTTR is the average time required to repair a system whereas MTTD is the average time required to detect a failure. In most applications, MTTR and MTTD are just a small fraction of the total MTTF. Therefore, the approximation that MTBF and MTTF are almost equal is sometimes fair. MTTR and MTTD are difficult to estimate and can be determined by injecting faults into a system, measuring the time required to repair it. Both measures will be further discussed in the availability section.

2.3.9. Reliability prediction

In the case of design of hardware systems, there are two different known theoretical methods to meet the above mentioned reliability requirements and specifications:

- Fault avoidance: makes use of substantially higher reliability components and substantially higher than expected lifetime. Birolini (Birolini, 2007) introduced a comprehensive theoretical approach based on the application of reliability engineering throughout the system to reach this goal.
- Fault tolerance: deliberately introduces redundancy in the system to achieve continuous operation.

During the last 50 years there have been several attempts (Gnedenko et al., 1999; Koren and Krishna, 2007; Kovalenko et al., 1997) to connect probability and

reliability. A brief review of the probabilistic theory of reliability for the analysis of real objects and their features (fault tolerance) is presented below.

Reliability of systems can be estimated by partitioning those systems into more elemental entities (e.g. subsystems or components) and then by assessing the individual probability theory of these individual entities. The entities can be interconnected in serial, parallel or both. Therefore reliability models are needed to illustrate the functional relationship among the entities of the system and the way in which a failure of each component would affect the overall reliability of the system.

2.3.9.1. Serial Reliability

The mathematical equations in this section are based on the classic reliability of (Birolini, 2007). In this model, the entities are connected in series. When minimum design and costs are specified in the design requirements of a system, a series system is the usual choice for designers. For the system to be operational, all of the components or subsystems should be operational and work correctly. Serial systems are inherently unreliable since the failure in one of the elements would cause a stoppage of the overall system.

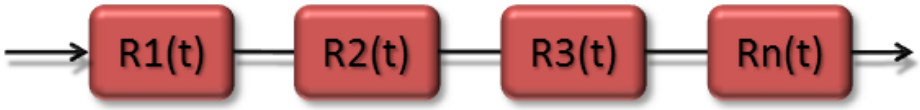


Figure 2-9. Logic diagram of Serial reliability

The reliability of a system without redundancy may be described with a sequential reliability block diagram (see Figure 2-9). In this arrangement the system reliability is the product of its individual component reliabilities, assuming they are organized in serial (cumulative) structure. Note that for this structure, if the reliability of each component is R_i, the total system reliability R_s is given by:

$$R_s(t) = \prod_{i=1}^{n} R_i(t) = exp\left(-\left(\sum_{j=1}^{n} \lambda_j\right)t\right)$$

Equation 2.24. System reliability of a serial system

And the failure rate of the system λ_s is given by:

$$\lambda_s = \lambda_1 + \lambda_2 + \lambda_3 + \ \ldots + \lambda_n$$

Equation 2.25. Failure rate of a serial system

Furthermore, the Mission Time Function $M_T(r)$ gives the time at which system reliability falls below the given threshold level r. The relationship between reliability $R(t)$ and mission time $M_T(r)$ is given by the definitions:

$$R[M_T(r)] = r$$

Equation 2.26. Mission time function M_T with threshold level r

$$M_T[R(t)] = t$$

Equation 2.27. Mission time function M_T at a given time

If λ is constant then, using Equation 2.14:

$$t = \frac{-ln(r)}{\lambda}$$

$$M_T(r) = \frac{-ln(r)}{\lambda}$$

Equation 2.28. Mission time function M_T with constant failure rate

So for a non-redundant system with n components

$$M_T(r) = \frac{-ln(r)}{\sum_{i=1}^{n} \lambda_i}$$

Equation 2.29. Mission time function M_T for non-redundant systems with n components

The failure rate of a sequential independent element system is equal to the sum of the failure rates of its elements. In the case of a constant failure rate across all elements, the MTTF of the whole system ($MTTF_S$) can be calculated as follows:

$$MTTF_S = 1/\lambda_s$$

Equation 2.30. Mission time to failure of a system with constant failure rate

Note that this equation highlights the fact that the reliability of a system is directly impacted (in practice often dominated but not solely determined) by the reliability of its least reliable component.

2.3.9.2. Parallel reliability: Redundancy and fault tolerance

In the previous model, no redundancy was taken into account to calculate the system reliability. A second approach to achieve a required level of reliability is the deliberate introduction of extra components into the system. The sole purpose of introducing this redundancy artificially is to increase reliability. However, there is a price to pay for such improvement in the system's reliability.

This approach assumes a deliberate introduction of redundancy in the system and has been applied since the original work of Von Neumann (von Neumann, 1956) and Pierce (Pierce, 1965). Note that introducing redundancy involves some additional components and complexity and it is therefore imperative that the reliability benefit accruing from the redundancy scheme must far exceed the decrease in reliability due to the actual implementation of the redundancy mechanism itself.

The classic parallel generalization of the redundancy model (Birolini, 2007) describes a system of n statistically identical elements in active redundancy, where k element(s) is/are required to perform a function and the remaining n-k are in reserve.

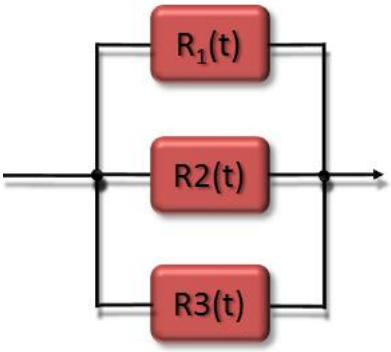


Figure 2-10. Parallel reliability

A function of the system is considered successful if during scheduled time k element(s) of the system was/were available. As an example, in the case of a *1-out-of-3* system (Figure 2-10), its function would be complete if at least one of the elements was known to be working correctly. The second and third elements are redundant and introduced only for reliability purposes when the first unit is known to be faulty.

For the system of Figure 2-10 the reliability function is as follows:

$$R(t) = R_1(t) + R_2(t) + R_3(t) - R_1(t)R_2(t)R_3(t)$$

Equation 2.31. General reliability of a 1-out-of-3 parallel system

Assuming that the elements are identical, work or fail independently of each other and have constant failure rate $R_1(t) = R_2(t) = e^{-\lambda t}$, then by substitution:

$$R(t) = 3e^{-\lambda t} - e^{-3\lambda t}$$

Equation 2.32. Reliability of a 1-out-of-3 parallel system with constant failure rate

$$MTTF_s = \frac{3}{\lambda} - \left(\frac{1}{3}\right)\lambda = \frac{8}{3\lambda}$$

Equation 2.33. Mean time to failure of a 1-out-of-3 parallel system with constant failure rate

Therefore the apparent working time of the redundant system is increased.

In the general case where n redundant elements are introduced as spares to provide successful completion of an element's function with the same assumptions as above, the overall system reliability is given by:

$$R(t) = 1 - \left(1 - e^{-\lambda t}\right)^n$$

Equation 2.34. General reliability of a 1-out of n parallel system with constant failure rate

In the above equation n is the number of modules, $e^{-\lambda t}$ is the reliability of the original system and it is assumed that:

- There is a fault-free mechanism to detect and report failure of the active module,
- There is a fault free switching mechanism to replace the active module in case of detected failure, and
- All modules have equal reliability

Thus, there is no doubt that redundancy even for this classic case could improve reliability of the system considerably. Note that the redundant components do not necessarily need to be identical, but could also correspond to additional hardware with different reliabilities used to detect and treat transient faults.

2.3.9.3. Mixed reliability: Serial and Parallel

In practice, systems are usually made of a combination of serial and parallel components. More complex math applies to the reliability of these mixed arrangements. This type of arrangement is frequently used in systems where a specific part is particularly prone to failure. Figure 2-11 depicts an example of M of N system, whose elements may or may not have constant rates, and has a voter that counts for the serial reliability element.

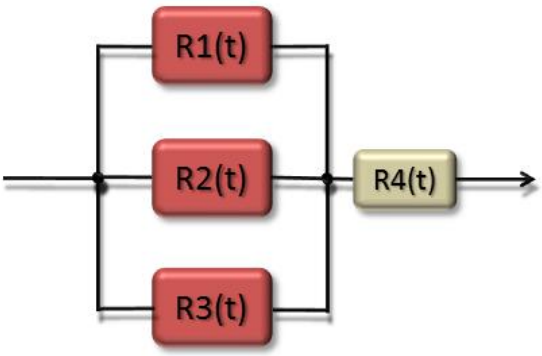


Figure 2-11. Reliability of a combination of serial/parallel components with a voter

Assuming that only 1 out of N parallel components needs to operate, the reliability of the parallel section of the system is defined by:

$$R(t) = 1 - \left[\left(1 - R_1(t)\right)\left(1 - R_2(t)\right)\left(1 - R_3(t)\right)\right]$$

Equation 2.35. General reliability of a parallel section of a specific mixed serial/parallel system

$$R_{1-3}(t) = 1 - \prod_{i=1}^{n}(1 - R_i(t))$$

Equation 2.36. General reliability of a parallel section of a specific mixed serial/parallel system

The total reliability of the mixed serial/parallel system shown in Figure 2-11 is specified by:

$$R(t) = R_{1-3}(t)R_4(t)$$

Equation 2.37. General reliability of a parallel section of a specific mixed serial/parallel system

Therefore, a relatively reliable voter would dominate the reliability of a redundant system.

2.4. Safety

In safety-critical systems, safety describes the absence of catastrophic failures for users and the environment when a failure takes place. A system that can be repaired after failure presents a minimum of two states: functional and failed. Some other systems are able to have extra states even under faulty conditions. An example of such system, depicted in Figure 2-12, has the possibility of transiting to a safe state, in a manner that does not cause any harm.

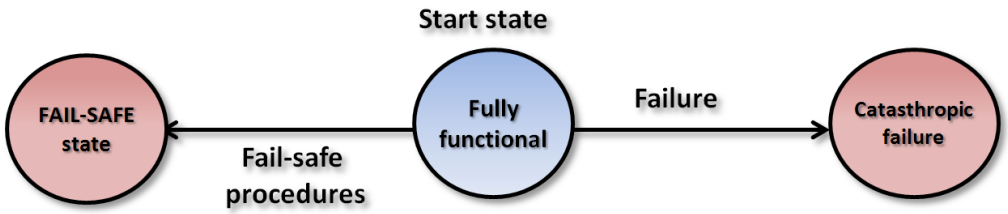


Figure 2-12. A basic fail-safe system with three states

Safety is a measure of the fail-safe capability of a system and it is defined as the probability that a system will either perform its function correctly or will discontinue its operation in a safe way (Laplante and Ovaska, 2011). Quantitatively, safety is the probability that the system will not fail in the interval *[0,t]* in such a manner as to cause unacceptable damage to other systems or compromise the safety of any people associated with the system.

The safety function can be described by:

$$S(t) = P_{functional}(t) + P_{safe-mode}(t)$$

Equation 2.38. Availability as reliability and recoverability

Safety is directly dependent on *"risk"*, as the probability of loss associated to a particular failure. In turn, risk is a function of the probability of failures and their severity on the system. A system can be unreliable, have low availability and yet be safe. A system is safe if it functions correctly or if in case of failure it can remain in a safe state.

2.5. Security

Contrary to (Avizienis et al., 2004) that adds confidentiality as one of the attributes of security**,** we consider Security as a property that can be defined by three attributes: integrity, maintainability and availability. For a resilient system in the field of RTS's, confidentiality is not an essential attribute.

2.5.1. Integrity

The attribute of Integrity is inward-looking and is related to the capability of a system to protect computational resources and data under severe circumstances. Integrity can be defined as the absence of improper system state alterations. As suggested by (Storey, 1996) two types of integrity can be defined:

- System integrity: the ability of a system to detect faults during operation and to inform to a human operator.
- Data integrity: the ability of a system to prevent damage in data and possibly to correct errors that occur as a consequence of faults.

2.5.2. Maintainability

Based on the definition of (*McGraw-Hill concise encyclopedia of engineering*, 2005), Qualitatively, we define *maintainability* as *the ease and rapidity in which, following a failure, a repairable system can be restored to a specified operational condition*. Quantitatively, we define *maintainability* as the *probability M(t) that a failed system will restore to a normal operable state specified within a given time frame t.*

The restoration process involves the location of the problem, the reparation\recovery of the system bringing it back to a normal operational condition. Maintainability has two main components, serviceability and recoverability that should by carefully analysed in the implementation of self-repairing systems:

$$M(t) = f\big(S(t), RC(t)\big)$$

Equation 2.39. Maintainability of a system

Maintainability characteristics are determined by the system design of maintenance procedures, such *preventive* (*PM*) and *corrective maintenance* (*CM*) procedures. These two procedures apply to the serviceability and recoverability components and determine the length of repair times (Bodsberg and Hokstad, 1995; Dhillon, 2006). PM is the set of activities performed on a system before the occurrence of failure in order to prevent any degradation in its operating condition. PM aim to reduce the probability of failure at predetermined intervals or along with prescribed criteria. CM is the remedial set of activities performed on a system in order to recover an item to its fully functional condition. CM is usually unplanned that requires urgent attention

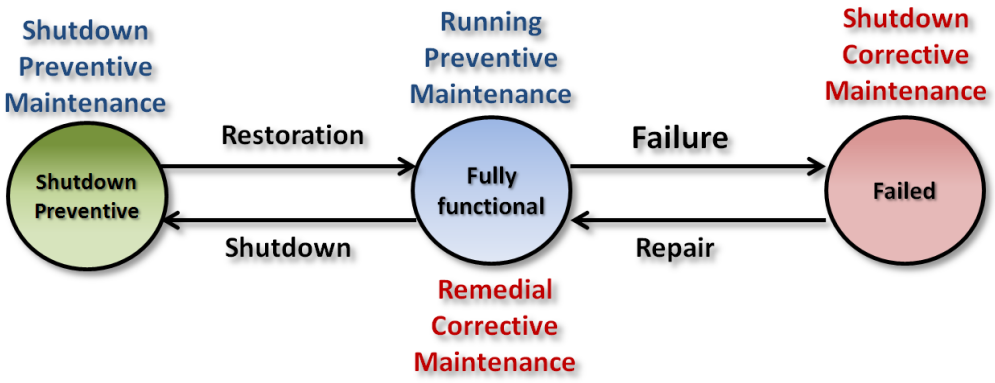


Figure 2-13. Preventive and corrective maintenance on a three state repairable system

Figure 2-13 shows PM and CM mechanisms on a three-state repairable system. Note that not all maintenance leads to downtime of the three-state system. Whilst *running* PM and *remedial* CM prevent and correct failures during normal operation, *shutdown* PC and CM take place during non-functional states.

2.5.2.1. Recoverability

Once the problem has been identified and located by the testing mechanisms, CM can be carried out to complete the necessary repairs. Consider a repairable

system with two states: a fully functional and a failed one (as in Figure 2-3); however, in this case the failed state can be abandoned after successful CM, transiting back to a fully functional state (as in Figure 2-14).

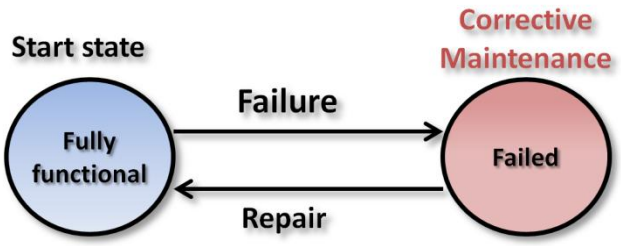


Figure 2-14. A repairable system with two states and corrective maintenance

Recoverability RC(t) may be defined as *the ease of restoring the service after failure*. It can be modelled as:

$$RC(t) = 1 - e^{-\mu t}$$

Equation 2.40. Recoverability of a system

where μ is the repair rate or average number of repairs that can be performed per time unit, the key aspects of recoverability, MTTR and MTTD, are given by:

$$MTTD + MTTR = \frac{1}{\mu}$$

Equation 2.41. Mean time to repair (MTTR) and Mean time to detection (MTTD) of a system

Note that good testing would affect recoverability to a degree.

MTTR will be further discussed below in the availability section.

2.5.2.2. Serviceability or Testability, T(t)

Testability T(t) is the ease in which servicing and inspections can be conducted in order to identify the characteristics of a system; it is the ability to check certain attributes within a system. Measures of testability allow the system to assess the

ease of performing tests. Ideally, in order to improve testability the tests can be automated and implemented as an integral part of the system. These techniques can be used for error detection and error correction within the system. Since most of the time, testability is often used to determine the source of the problem, one way to improve the maintainability of the system significantly is the use of automatic diagnosis.

Testability relates to reliability since it allows detection and correction of errors that would, otherwise become failures, thus improving the overall reliability of the system. Testability is clearly connected with recoverability due to the importance of minimizing the time to locate and identify specific problems.

Two properties/measures closely associated with testability, *controllability* and *observability* (Franklin and Saluja, 1995, p. 199; Goldstein, 1979). Observability relates to the probability of *"observing"*, via output measurements, the state of a system. Controllability instead is associated to the ease of forcing parts of the system into desired states by using appropriate control signals. Design for testability techniques (DFT) (Alanen and Ungar, 2011; Karimi and Lombardi, 2002; Landis, 1989; Mathew and Saab, 1993), can be used in order to increase observability and controllability of systems.

2.5.2.3. Coverage

Mathematically, *fault coverage* C is the conditional probability that, given de existence of a fault in the operational system, the system is able to recover, and continue information processing with no permanent loss of essential information (Bouricius et al., 1969) i.e.:

$$C = \Pr\ [system\ recovers \mid system\ fails]$$

Equation 2.42. Mathematical definition of coverage

Fault coverage is a good measure of maintainability and, specifically of the system's ability to detect, locate, diagnose, contain and recover from the

presence of a fault. Several types of fault coverage can be distinguished, depending on whether the designer is concerned with fault detection, diagnosis, containment or recovery (Kaufman and Johnson, 2001). In Figure 2-15, we extend the phases of fault handling by (Dugan and Trivedi, 1989), showing the relationship among the steps of recovery and their coverage.

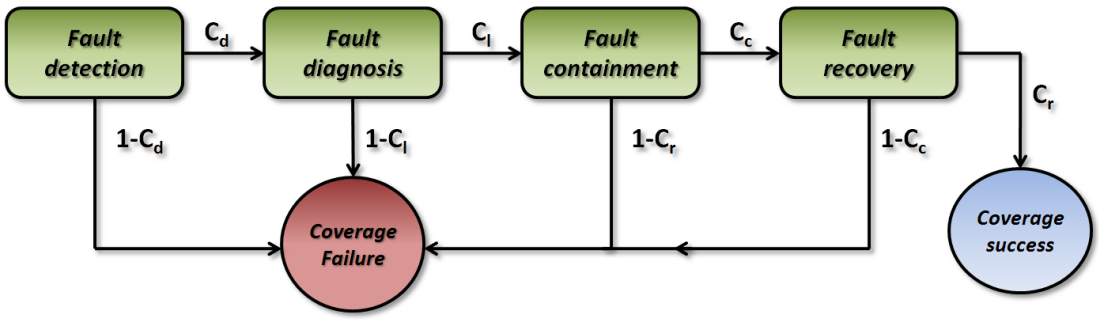


Figure 2-15. Four phases of fault handling and their coverage

Fault detection coverage C_d measures the system's ability to detect fault. *Fault diagnosis coverage* [4]C_l is a measure of the system's ability to locate and determine the type of fault. *Fault containment/isolation coverage* C_c is a measure of the system's ability to contain faults within a predefined boundary (*fault containment region* or *FCR*). For instance, fault that occurs in a subsystem can be detected, located, and its effects can be prevented from propagating to other subsystems.

Finally, the general term "*coverage*" or "*fault coverage*" is often used to refer to *fault recovery coverage*, which measures the system's ability to recover from faults and maintain correct operation. Recovery may involve modifying the structure to remove the faulty component (reconfiguration) including graceful degradation. The fault coverage *C* for the system is given by:

[4] Fault diagnosis involves both the location (*fault location*) and determination of the fault type (*fault determination*)

$$C = C_d \times C_l \times C_c \times C_r$$

Equation 2.43. Coverage as a function of fault detection, diagnosis, containment and recovery coverages

Clearly, high fault recovery coverage requires high fault detection, diagnosis and containment coverage.

2.5.3. Availability

A simple definition for *availability* of a repairable[5] system is *"Readiness for correct service"* (Avizienis et al., 2004). This measure is suitable for applications in which continuous performance is not essential but where it would be costly to have long downtimes. Availability is strongly dependent on how frequently the system becomes non-operational (reliability) and how quickly it can be repaired (maintainability) (see Figure 2-14).

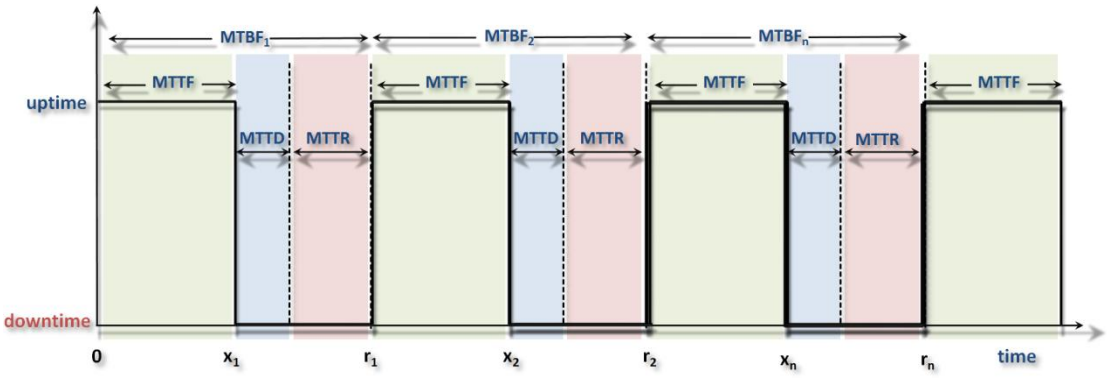


Figure 2-16. Failure and repair cycle of a system

[5] The concept of availability is applicable to repairable systems. Availability of a non-repairable system would be the equivalent to reliability.

As defined in the MTBF equation (Equation 2.23) the mean time between failures of a system can be defined as a combination of MTTF, MTTR and MTTD. Figure 2-16 illustrates the variations of the state (functional-failed) of a repairable system. The time of operation of such systems is discontinuous. From time *0* to time X_1 the system is continuously available and therefore has an internal availability of *1*. After the first failure at time x_1 internal availability keeps decreasing until the detection and recovery mechanisms complete the repair at time r_1, returning to the original functional state. The system will fail again at time x_2 after a certain time of operation $[r_1 - x_2]$, get repaired at time r_2, and this process will reiterate. Assuming that X_i is an average of system failure and i an average of system repair, for $i>1$:

$$MTBF = \sum_{k=1}^{n} (X_i - X(i-1))$$

Equation 2.44. Mean time between failures with average failure and system repair

The relation between time to failure, time between failures and time to repair is displayed in Figure 2-17 above.

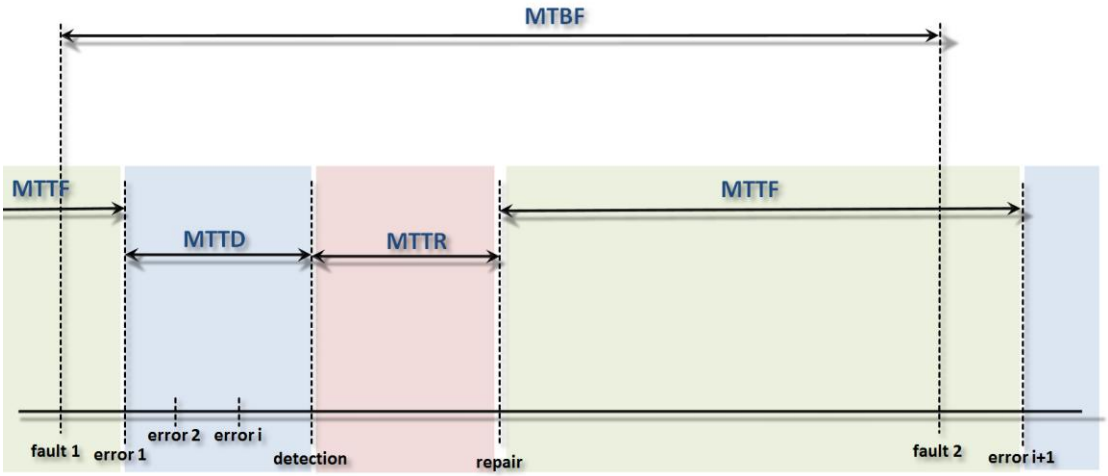


Figure 2-17. Relation between Time to failure (TTF), time between failures (TBF) and time to repair (TTR)

There are various availability measures that can be classified differently depending on the time interval preferred or the downtimes used.

2.5.3.1. Instantaneous or point availability, A(t)

Instantaneous or *point availability A(t)* is the probability that the system will be operational at a random time *t* (Barlow and Proschan, 1975). It describes the on-demand probability of proper service. It is equivalent to reliability when there is no repair. While internal availability is based on an interval time, instantaneous availability is based on a specific instant of time. At any given time t, the system will be functional if one of the following conditions is met (Elsayed, 1996):

- The system was functional from *0* to *t* (it never failed by time *t*). The probability of this happening is *R(t)* (Equation 2.14).
- The system has been functional since the last repair time r_i (see Figure 2-16) when $0 < r_i < t$. This has a probability of:

$$\int_0^t R(t - r_i)m(r_i)dr_i$$

Equation 2.45. Probability that the system has been functional since last repair time for $0 < r_i < t$.

- With $m(r_i)$ being the renewal density function of the system.

The instantaneous availability of the system is the sum of these two probabilities:

$$A(t) = R(t) + \int_0^t R(t - r_i)m(r_i)dr_i$$

Equation 2.46. Instantaneous or point availability of a repairable system

2.5.3.2. Average uptime availability (or mean availability), $\overline{A(t)}$

The *average uptime availability* or *mean availability* $\overline{\boldsymbol{A(t)}}$ (Lie et al., 1977) is the proportion of time during a time period [0-t] that the system is functional and is given by:

$$\overline{A(t)} = \frac{1}{t}\int_0^t A(r_i)dr_i$$

Equation 2.47. Average uptime availability of a repairable system

This type of measure is suitable to systems with periodical downtime for maintenance/repairing.

2.5.3.3. Limiting or Steady-state availability, A(∞)

The *limiting* or *steady state availability* (Applebaum, 1965) of the system *A(∞)* is the limit of the availability function as time t tends to infinity:

$$A(\infty) = \lim_{t\to\infty} A(t)$$

Equation 2.48. Average uptime availability of a repairable system

2.5.3.4. Inherent availability, A_I

In its simplest form, availability A can be mathematically generalised as:

$$A = \frac{Uptime}{Uptime + Downtime}$$

Equation 2.49. General availability as a function of uptime and downtime

During the design phase of a FT system, *Inherent availability* A_I is a useful measure (Valstar, 1965). A_I defines the availability of a system only in regard to effective functional time (uptime) and downtime due to *corrective maintenance (CM)*. It can be calculated using estimated parameters (MTTF, MTTD and MTTR) as:

$$A_I = \frac{MTTF}{MTTF + MTTD + MTTR} = \frac{MTTF}{MTBF}$$

Equation 2.50. Inherent availability as MTTF and MTBF

Hence, if MTTF or MTBF are long compared to MTTR and MTTD then the system's availability will be high. Likewise, if MTTR and MTTD are short then the

system's availability will also be high. As reliability decreases (e.g. low MTTF), better recoverability will be needed (lower MTTR/MTTD) to achieve the same availability.

2.5.3.5. Achieved availability, A_A

A_I is a good parameter to measure systems under ideal conditions where downtime due to *preventive maintenance* (*PM*) is overlooked. Achieved availability A_A is similar to inherent availability with the exception that downtimes due to PM tasks are also included (Conlon et al., 1982). In can be defined as:

$$A_A = \frac{OT}{OT + TCM + TPM}$$

Equation 2.51. Achieved availability according to USA department of defence

Where *OT* is the *total operating time*, TCM is the *total corrective maintenance time* and *TPM* the *total time spent during preventive maintenance actions*.

2.5.3.6. Availability-recoverability-reliability relationship

At first glance, it might seem that a highly available system would also have high reliability. Nonetheless this in not always the case, a system can be highly available yet suffer from frequent periods of non-operation as long as the length of the downtime is extremely short. Let's explore further the relationship between availability and reliability. Reliability represents the probability of systems and components to perform its intended function for a desired period of time *[0,t]* under specified environmental and operating conditions. However, reliability in itself does not take into account any repair actions. Reliability does not reflect how long the recovery of a component/system will need in order to take it back to a working condition. Availability reflects not only how often a system fails but how often it can be repaired (it accounts for repair actions). Thus, it is a function of reliability, recoverability and thus testability.

$$A(t) = f(R(t), M(t))$$

Equation 2.52. Availability as reliability and recoverability

Table 2-1. Reliability-Recoverability-Availability relationship

Reliability	Recoverability	Availability
Constant	Constant	**Constant**
Constant	Decreases	**Decreases**
Constant	Increases	**Increases**
Decreases	Constant	**Decreases**
Increases	Constant	**Increases**

Table 2-1 above, presents the relationship between reliability, recoverability and availability. As shown by the table, once again, high reliability does not necessary imply high availability. Availability decreases as time to repair increases. Even an unreliable system could present high availability if MTTR is low.

2.6. Performability

The all-or-none nature of operation implicit in classic reliability and availability models does not measure in detail systems that can operate with different capability levels (e.g. multiprocessor systems). Consequently, another key attribute of resilience, performability and its measure, *mean computation before failure* (*MCBF*) can be employed. MCBF is described as the expected amount of computation available on the system before its first failure, given an initial state (Beaudry, 1978).

In qualitative terms, we define *performability* as *the ability of a system or component to accomplish its designated functions within specified constraints such as speed, accuracy or memory usage*. It is the measure of the likelihood that some subset of the functions of the system or component is performed correctly during a certain time interval. Quantitatively, Performability *P(L,t)* has been

defined as *"the probability that the component's or system's performance will be at or above some level L at the instant of time t"* (Fortes and Raghavendra, 1985).

After the occurrence of faults and errors, certain systems have the ability to continue to perform correctly, however with a diminished level of performance. This ability or feature is called *Graceful degradation*, or *fail-soft operation* (Gountanis and Viss, 1966), and it is the ability of a system (*gracefully degrading system* or *GDS*), upon failure of one or more of its component units, to continue the processing of tasks at the expense of decreasing its performance level. The performability of a GDS *P(L,t)* at time *t* depends on the amount of available resources and their computational capability provided.

Note that performability differs from reliability in that reliability measures the likelihood that all functions are performed properly, whereas performability measures the likelihood that some subset of the functions is performed properly. Nevertheless, these two concepts are related since a GDS with a low rate of failure (high reliability) will have most of its resources computational capability available and therefore performability of the system will be close to its ideal value.

2.7. Resilience

Historically, the term resilience has had multiple meanings in various fields. As a property it has different connotations. In social psychology resilience is about elasticity, spirit, resource and good mood. On the other hand, in material science resilience involves not only elasticity but robustness. In computer science it has been identified as a synonym for fault tolerance. In this thesis we extend the concept of resilience for safety critical applications. First we start by selecting the material science connotations. Hence, our definition of resilience includes both attributes: *robustness* and *elasticity*.

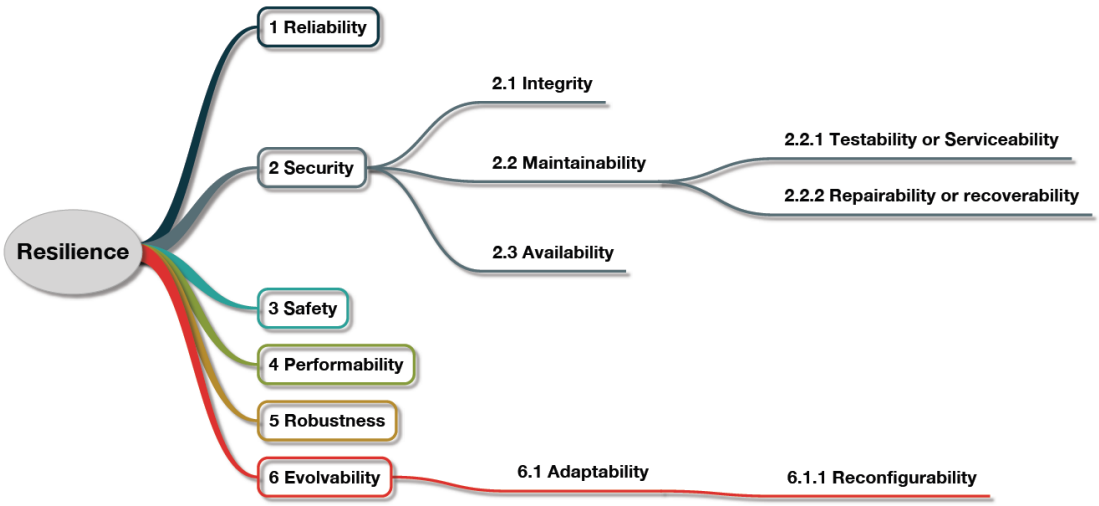


Figure 2-18. Attributes and measures of resilience

Figure 2-18 illustrates the different attributes and measures of resilience. The term robustness involves the use of static techniques such the use of very reliable materials or the use of rigid and pre-design approaches of fault tolerance. A robust system can deliver correct service in conditions beyond the normal domain of operation without fundamental changes to the original system. This is more an aim that an objective. Total reliability to unforeseen faults other than the normal domain of operation is not feasible.

On the other hand, we interpret elasticity as the ability to spring back without losing the intrinsic properties of the material. Applied to resilience, we

understand elasticity as the ability to evolve, to successfully accommodate changes (*evolvability*). An evolvable system may perform changes to the system, decreasing its level of performance or reliability for a specific time range 1) to compensate for faults or 2) during exceptional circumstances (graceful degradation).

More specifically, we consider that a resilient system must have the ability to be adaptable, understanding *adaptability* as the ability to evolve while executing. Therefore, adaptability is a subset of evolvability and requires the ability to anticipate to changes prior to the occurrence of the resulting damage.

Therefore a resilient architecture must include different mechanisms to acquire both attributes: a) static pre-design fault tolerant techniques (robust) and b) dynamic techniques (elastic) that may be achieved with the ability to reconfigure elements of the system (reconfiguration).

2.7.1. Requirements

The main aim required to implement a resilient architecture for safety critical applications is the *"ability to deliver correct service adapting to disturbance, disruption and change within specified time constraints"*.

The above aim can be subdivided into more specific objectives, as follows:

- Continuity of service (reliability);
- Readiness for usage (availability);
- Non-occurrence of catastrophic consequences (safety);
- Non-occurrence of incorrect system alterations (integrity);
- Ability to undergo corrective maintenance and recovery with maximum coverage of faults (testability and recoverability)
- Ability to perform in the presence of faults (performability)
- Ability to decrease the level of performance for a specific time range in order to compensate for hardware faults (graceful degradation)
- Ability to regain operational status via reconfiguration in the presence of faults (recoverability via reconfiguration)
- Ability to accommodate changes (evolvability)
- Ability to anticipate to changes (adaptability)

2.7.2. Effectiveness of resilience

We consider the following attributes: reliability $R(t)$, security $SC(t)$, integrity $I(t)$, maintainability $M(t)$, testability $T(t)$, recoverability $RC(t)$, availability $A(t)$, safety $S(t)$, performability $P(L,t)$, robustness $RB(t)$, evolvability $E(t)$, adaptability $AD(t)$ and reconfigurability $RC()$.

Maintainability is a function of serviceability and repairability:

$$M(t) = f(T(t), RP(t))$$

Equation 2.53. Maintainability as a function of serviceability and repairability

Security is a function of integrity, availability and maintainability:

$$SC(t) = f\big(I(t), A(t), M(T(t), RP(t))\big)$$

Equation 2.54. Security as a function of Integrity, availability and maintainability

Evolvability is a function of adaptability and reconfigurability:

$$E(t) = f(AD(t), RC(t))$$

Equation 2.55. Evolvability as a function of adaptability and reconfigurability

Therefore resilience *RES(t)* will be a function of all these attributes:

$$RES(t) = f\begin{pmatrix} reliability, integrity, testability, recoverability, \\ availability, safety, performability, \\ robustness, adaptability, reconfiguration \end{pmatrix}$$

Equation 2.56. Availability as reliability and recoverability

With all these attributes the following systems would benefit from the implementation of effective resilience.

- Safety-life critical: e.g. aircraft and nuclear reactor control. life support systems
- Business critical
- Reliable critical: e.g. telephone switching-, traffic light control-, automotive control (ABS, fuel injection) systems
- Mission critical and long life systems: e.g. manned and unmanned space borne, satellites and other systems in inaccessible locations
- Non-stop systems that demand high availability

Resilience is not a simple and single concept, rather, it possesses different components or key attributes. Taking into consideration all these attributes, our definition of resilience is as follows:

"A resilient system is a system that over a specified time interval, under specified environmental and operating conditions, is ready to perform its intended function, guaranteeing the absence of improper system alterations, having the ability to anticipate and accommodate changes while executing, and the ability to conduct servicing and inspections so that in case of failure quick restoration to a specified working condition must be achieved, or otherwise discontinue of the operation in a safe way is provided"

2.8. Conclusion

This chapter explains the concept of resilience that encompasses important attributes and measures that will be used during the thesis. Such concepts have been reviewed and combined to define our concept of resilience.

Safety critical systems must provide correct service at all times by trying to avoid the occurrence of any catastrophic failure. Different techniques can be employed to increase reliability by avoiding/preventing hardware faults from becoming errors that may lead to failures and catastrophic failures.

We introduce the concept of *vicious cycle* that explains the reasons behind the performance and reliability problems that the microprocessor industry is currently facing. The increase of transistor density, operating frequencies and architectural complexity is drastically decreasing the reliability of newer systems. There is, therefore, a need for implementing mechanisms that can deal with the upcoming fault rates.

The mathematical background for classical reliability has been reviewed together with the basic definitions for reliability evaluation. For constant failure rate, independent of time, the exponential distribution is the most suitable for

the reliability analysis of the useful time of systems. The age of a system should be taken into account when analysing reliability. Three different periods with different reliability distributions have been explained by reviewing the bathtub curve, which represents very well the effect that aging and degradation have on HW reliability.

In addition, it is also shown how to estimate the reliability of serial, parallel and mixed components. The failure rate of a serial system is equal to the sum of the failure rates of its individual elements. Therefore the more components a serial system has the higher the probability of system failure. The reliability of a system is often dominated by the reliability of its least reliable component. By deliberately and carefully introducing extra components into a system, overall reliability can be increased as long as the reliability benefit accruing from the redundancy scheme exceeds the decrease in reliability due to the actual implementation of the redundancy mechanisms itself.

We extend the classical definition of resilience and apply it to the field of safety critical computing. Moreover, we quantify the key attributes that a resilient system must have, exploring the relationships among these quantitative measures. The attributes of safety and performability are explained. The concept of security is described, including its attributes: integrity, availability, testability and recoverability. The mathematical background and the basic definitions for system availability are also developed. We show how the availability of FT systems can be estimated using different methods and measures.

Finally, the main aim and objectives required to implement a resilient architecture for safety critical applications are defined. A resilient system, over a specified time interval, under specified environmental and operating conditions (performability), “must be ready” (in terms of availability) to perform its intended function (reliability), guaranteeing the absence of improper system alterations (integrity). It must have the ability to conduct servicing and inspections (testability) so that in case of failure quick restoration to a specified working condition must be achieved (maintainability) can be provided or can

discontinue its operation in a safe way (safety). Furthermore, a resilient system must have the ability to anticipate changes and evolve (evolvability) while executing (adaptability), successfully accommodating changes by reconfiguring elements of the system if necessary (reconfiguration).

Chapter 3

Chapter 3

Dealing with faults: redundancy

This chapter provides a survey of state-of-the art design strategies to handle faults with special stress on redundancy-based techniques. Section one presents an overview of fault avoidance design strategies. Section two provides a survey of fault tolerance techniques. The notion of redundancy and its different types are presented and a notation, which may be used to describe the different types of redundancy, is introduced. The concepts, capabilities and applications of the different techniques based on structural (SR), information (IR) and time redundancy (TR) are compared and discussed.

3.1. Handling faults: design strategies

In order to increase the reliability of safety-critical systems so that correct service can be delivered, techniques need to be developed to prevent or reduce the appearance of faults that could cause catastrophic failures. Depending on the phase of the development cycle and the level of abstraction at which the faults are tackled, two different design strategies can be adopted: fault avoidance and fault tolerance.

Fault avoidance strategies can be used at device level during design time. Typical in mainstream applications, in order to reduce the number of failures, this approach focuses on preventing the occurrence of faults. Since a failure is the consequence of an error propagating, and an error is the consequence of a fault, eliminating faults would maximize reliability. Examples of this are silicon on insulator (SOI) and hardened memory cells. These techniques have drawbacks in terms of cost, speed of operation and chip area.

At execution or run time and at different levels of abstraction, fault tolerant strategies can be implemented.

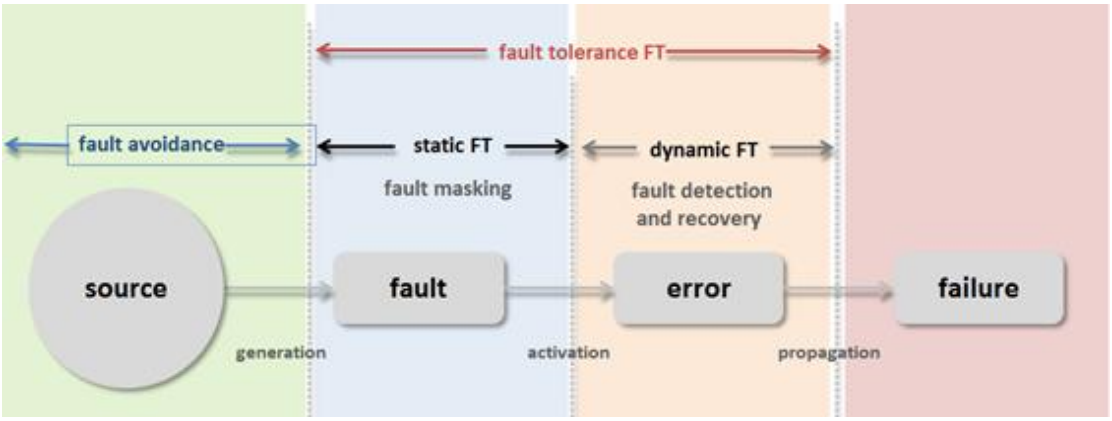


Figure 3-1. Mechanisms to deal with faults within the fault-failure lifecycle

Following the failure lifecycle and its different phases already described in Section 2.1, Figure 3-1 adds the different mechanisms to deal with faults within

the fault generation, error activation and failure propagation phases. Additionally, Figure 3-1 serves as a summary of the chapter introducing the fault avoidance and fault tolerance techniques and their phase of interaction within the failure lifecycle.

Focusing on the source of faults, fault avoidance mechanisms attempt to prevent faults from occurring in the first place. Once a fault has been generated it can be prevented from activating an error using static fault tolerant techniques such as masking. Alternately, errors can be detected and recovered using dynamic fault tolerance techniques. Therefore, either we prevent the faults from taking place (fault avoidance) or we deal with them using fault tolerance techniques.

3.2. Fault avoidance

Nowadays, mainstream systems employ fault avoidance design strategies in order to achieve their projected failure rates. Manufacturing companies perform assessments of sources and weaknesses that could lead to potential failures. Based on the assessments, preventive measures are taken to ensure that the overall reliability target is not compromised. Additionally, fault avoidance strategies may include technology and design mitigation techniques that implicate modifications of conventional manufacturing processes. These techniques involve the use of specific materials, the modification of the doping profiles of devices and substrates and the optimization of deposition processes for insulators.

Technology mitigation techniques consist of IC process variations by either improving the manufacturing process or by improving the materials used.

Improving materials: implicates the selection of specific materials with better characteristics, e.g.:

- Boron has been used extensively as a p-type dopant in silicon and has also been used in Boron-Phosphor-Silicate-Glass (*BPSG*) dielectric layers. For

BPSG-based semiconductor processes, *BPSG* can, in fact, be the predominant source of transient errors (Baumann, 2001). The removal of B-10 Boron isotopes in BPSG has been proven effective in the reduction of transient errors (Baumann et al., 1995).

- *Lead-free* materials can reduce the effect of alpha particles (May, 1979) (extensive information on alpha particle effects is provided in Chapter 4).
- Implanting of elements such as Al, As, Fl, P, and Si into oxides improves the resilience to Total Ionizing dose effects (*TID*). (Kato et al., 1989; Mrstik et al., 2000; Nishioka et al., 1989).

Improving the manufacturing process is based on changing the charge collection and charge sharing capabilities of the devices:

- *Substrate techniques*: e.g. using epitaxial substrate doping (EPI layer charge reduction)(Puchner et al., 2006), wells (single well, twin well and triple well processes) (Pellish et al., 2006; Puchner et al., 2006; Roche and Gasiot, 2005), buried layers (Roche and Gasiot, 2005) and dry thermal oxidation (Hughes and Benedetto, 2003)
- *Non-capacitance techniques*: e.g. increasing the node coupling capacitance between storage nodes and memory, or using a DRAM capacitor on top of the memory cell (Geppert, 2004)
- Using *alternative insulating substrates*; e.g. the use of Silicon on Insulator (SOI) or Silicon on Sapphire (SOS) (J. R. Schwank, 2003) would mitigate significantly the transient faults due to radiation (described in Chapter 4).

Whilst technology mitigation techniques are based at the process level, **design mitigation techniques** operate on the layout level. An example of this type of technique is the use of enclosed layout transistors. Furthermore, to prevent the effects of radiation memory cells can be hardened with the use of contact and guard rings.

The effect of silicon failure mechanisms, such as radiation induced transient faults and wear-out defects, is proportional to the clock speed, supply voltage, temperature, etc. Therefore, to ensure system reliability safety margins are inserted into clock speed, operating temperature and supply voltage margins. If the system failure rates resulting from the use of fault avoidance strategies fall within the specified reliability targets, the use of redundancy techniques is not justified. However, this is not the case for safety-critical systems.

Despite all the testing, verification techniques and technology improvement, hardware components will eventually fail. The fault avoidance approach will not be panacea and will be insufficient if:

- Failure rate and *MTTR* are unacceptable or
- the system is inaccessible for repair and maintenance actions

Therefore, fault avoidance techniques are only part of the solution for real time safety critical domains. Complete removal of faults via fault avoidance is not possible; above all, it has drawbacks in terms of cost of manufacturing the elements required, speed of operation and increased chip area.

3.3. Fault tolerance: using redundancy

The key ingredient of fault tolerance is redundancy. Redundancy is defined as the addition of information, resources or time beyond what is needed for correct system operation (Latchoumy et al., 2011). Fault tolerant techniques rely on redundancy that may include a combination of additional elements of hardware and/or software to detect and/or recover from faults. These components are

called redundant since they are not required in a *perfect system*[6]. *Artificially built-in* or *protective redundancy* is a system property that we define as the incorporation of extra components (transistors at a low level) in the design of a system so that its function is not impaired in the event of a failure. Redundancy may arise by design (artificially built-in redundancy) or as a natural by-product of design (*natural redundancy*). Natural redundancy is usually unexploited whilst artificially built-in redundancy has been deliberately introduced. In this thesis, when the term redundancy (or redundant) is used it is meant to have the artificial connotation instead of the natural one.

When a system does not provide the minimum reliability required, extra redundancy, not strictly necessary for the normal functioning of the system can be added in order to increase the probability of normal functioning. Notice that the term redundant does not mean identical functionality; it just denotes that it performs the same job. In this sense, heterogeneous hardware performing the same work can also provide redundancy.

Fault tolerance assumes actions such as fault detection, location of the faulty component, recovery, and if necessary, reconfiguration of the system. Fault detection is the process of determining the presence of faults and the time of occurrence. Fault location is to exactly locate the reason/origin of the fault. The system must be dynamically restored as though it is *'as good as new'* in operational terms, except for the fact that some of the redundancy has been used up and this may limit the possibilities for future repairs.

[6] A perfect system is a system with a theoretical 100% reliability. A perfect system is usually assumed to model extra reliable systems.

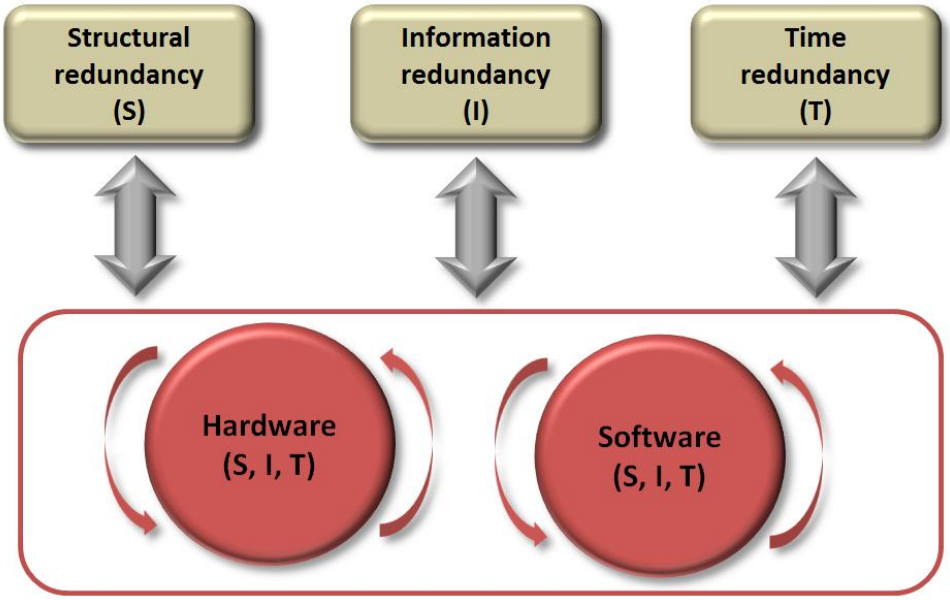


Figure 3-2. Redundancy types and their implementation (Schagaev, 2001)

Many different attempts to classify redundancy have been made (Avizienis, 1971; Carter and Bouricius, 1971; Schagaev, 1989, 2001). This thesis follows the approach proposed by (Schagaev, 2001) to classify redundancy. Figure 3-2 shows the different types of redundancy (at the top of the figure) and the way it can be implemented (at the bottom of the figure). In general, three types of redundancy exist: structural (*S*), involving multiplication of components, information (*I*), involving multiplication of information, and time redundancy (*T*), involving multiplication of functions in time. These can be implemented in hardware and software. This thesis focuses on the hardware aspect of redundancy and fault tolerance.

Redundancy comes with a cost. Information and structural redundancies require additional hardware components, extra power and perhaps extra area and shielding. Time redundancy requires faster processing to achieve the same performance, which in turn requires extra hardware and power.

3.3.1. Redundancy notation

Existing implementations of system redundancy use at least one of these three redundancy types, usually more than one and can be implemented in hardware (*HW()*), software (*SW()*) or a combination of both (*HW()*; *SW()*). As an example, hardware based information redundancy is abbreviated as *HW(I)*. Additional quantifiers are used together with the redundancy type to further specify the used redundancy as shown below in Table 3-1:

Table 3-1. Redundancy classifiers (Schagaev, 2001)

Quantifier	Example	Description
	SW(I)	No quantifier means general, not further specified redundancy. SW(I) for instance just indicates general software information redundancy
δ	*SW(δI)*	Additional used software based redundancy
Number	*HW(2S)*	The number indicates duplication (2), triplication (3), etc. of a system if used as a prefix for the redundancy type. The original system and the copies are identical. n instead of a discreet number is used to mark repetition until success in case of time redundancy
indices	*HW(S_1, S_2)*	Indices are used to mark a duplicated system implementation/hardware components

Note that the current notation does not include the implementation level. *HW(2S)* only indicates duplication, but not whether the whole system is duplicated or it is just parts of that system, such as, for example, duplicated memory.

Table 3-2 and Table 3-3 present some concrete examples of notation of hardware and software based redundancy. Any type of redundancy (hardware and software) needs additional structural redundancy for its implementation.

Table 3-2. Examples of notation of HW based redudancy

Redundancy type	Description
HW(2S)	Structural (material) redundancy of hardware such as duplicated memory system
HW(S_1, S_2)	A duplicated FT computer system with principally non identical parts
HW(δI)	Redundant information bit, for example an additional parity bit per data word in HW memory for error detection
HW(nT)	Special HW to delay execution (like in a timing diagram) to avoid transient faults
HW(δT)	Special HW to delay execution to avoid transient faults

Table 3-3. Examples of notation of SW based redudancy

Redundancy type	Description
SW(2S)	Structural (material) redundancy of hardware such as duplicated memory system
SW(S_1, S_2)	A duplicated FT computer system with principally non identical parts
SW(δI)	Redundant information bit, for example an additional parity bit per data word in HW memory for error detection

For instance, instruction repetition *HW(nT)* needs additional hardware registers to store the internal state to be able to perform instruction rollback. We refer to this as *supportive redundancy* and we define it as the redundancy needed for the implementation of the main redundancy technique. For the sake of simplicity, we usually omit this supportive redundancy. In cases where it is not clear whether an applied redundancy type is supportive or not, more than one redundancy type can be used. An example of this is the case of software based and hardware checks that are performed during idle time of the system: *SW(δS,δT)*.

3.3.2. Prognostics: Health management

An important contribution to the increase of system reliability is the use of *Prognostics*, defined by the International Organisation for Standardization as (ISO, 2004):

> *"... the estimation of time to failure and risk for one or more existing and future failure modes"*

This concept can be applied to real-time critical systems with active redundancy and fault reporting. Prognostics Health Management, also referred to as Condition-Based Maintenance (CBM), are strategies that recommend maintenance decisions based on information collected via condition monitoring (Jardine et al., 2006). These capabilities have been integrated in many safety-critical systems from unmanned vehicles, to aircraft, to power generation plants, etc. (DeCastro et al., 2011; Zhang et al., 2011)

In contrast with Planned Scheduled Maintenance (PM) where maintenance is carried out upon pre-defined schedule, CBM is performed only when it is triggered upon specific asset conditions.

CBM strategy consists of three major steps: data acquisition, data processing and maintenance decision-making. During the data acquisition phase the condition of the equipment is monitored to detect developing issues. The data processing phase involves diagnosis; this phase attempts to isolate the root of the cause. Finally, the maintenance decision-making phase where a corrective plan is developed and applied based on the data obtained from the previous step.

3.4. Structural redundancy: *HW(S)*

Structural hardware redundancy involves the multiplication of independent hardware components and execution of the same computation over such

components at the same time. Errors are exposed by checking/comparing the results of the independent executions.

In terms of granularity, redundancy in general, and not only hardware redundancy, can be applied on different abstraction levels. From bottom up we can distinguish between finer-grained: transistor level, gate or logic level, and between coarser-grained designs: circuit level, function level, system level, microcode level and chip level of abstraction. Therefore, redundant components can be as simple as transistors or logic gates but also as complex as processors or even larger entities.

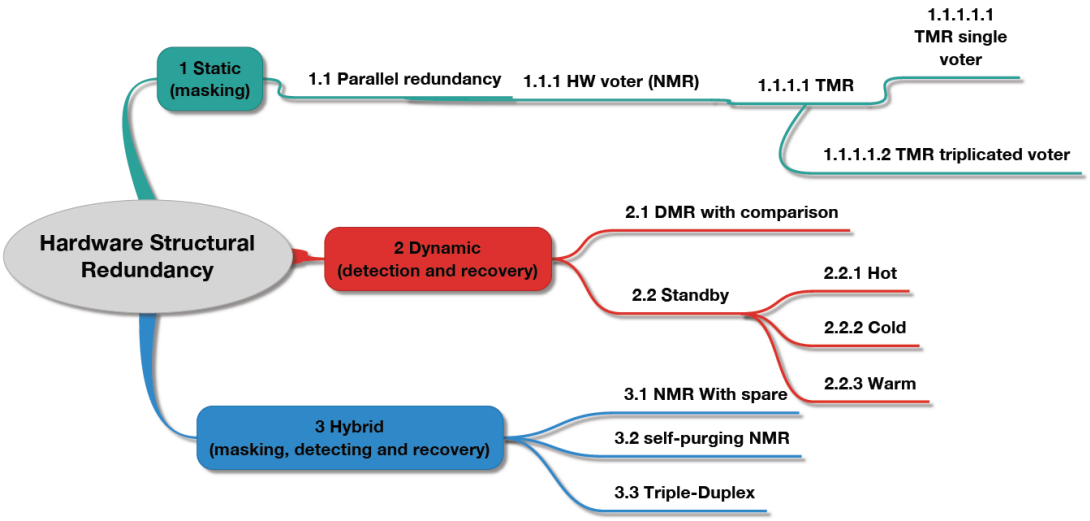


Figure 3-3. Taxonomy of structural HW redundancy

Figure 3-3 displays a taxonomy of the different hardware techniques based on structural redundancy. Two different architectures of redundancy can be distinguished: Parallel redundancy with redundant components running concurrently and Standby redundancy with a spare component being activated upon failure of an active component.

Furthermore, these extra resources can be used passively (*passive redundancy*), actively (*active redundancy*) or combined. In systems with active redundancy, all redundant components are in operation, sharing the load with the normal components. This implies that both, regular and redundant components, age

together. Passive components are not fully energized and start normal operation only when normal components fail. Passive components can be further broken down into two types: warm and cold standby. *Warm standby* components remain partially energized until becoming active and tend to deteriorate with time, hence, having lower failure rate than the regular components. *Cold standby* components are kept in reserve and they only become energized when put into use. These types of components have a zero failure rate, meaning they do not fail when they are in standby mode. Whilst passive components are switched off completely, standby components are partially activated. Standby redundancy is usually applied when the start time of the component is unacceptably long.

3.4.1. Static redundancy

Static redundancy, also called masking redundancy, implements error mitigation. The term static relates to the fact that redundancy is built into the system structure. Fault tolerant techniques based on this type of redundancy (*static fault tolerance*) transparently remove errors on detection. The most common form of hardware redundancy is *Triple modular redundancy* (*TMR*) (von Neumann, 1956) and its generalization *N-modular redundancy* (*NMR*). Note that *Dual modular redundancy* (*DMR*) (*DMR* is further explained in Section 3.4.2.1) is not considered static redundancy since the mismatch can take place but recovery is not possible.

3.4.1.1. Triple modular redundancy: HW(3S)+HW(δS)

A basic *TMR* system (*two-out-of-three*) is a fault tolerant form of NMR that consists of three fully redundant and active components or modules working in parallel with equivalent functionality (Johnson, 1989; von Neumann, 1956).

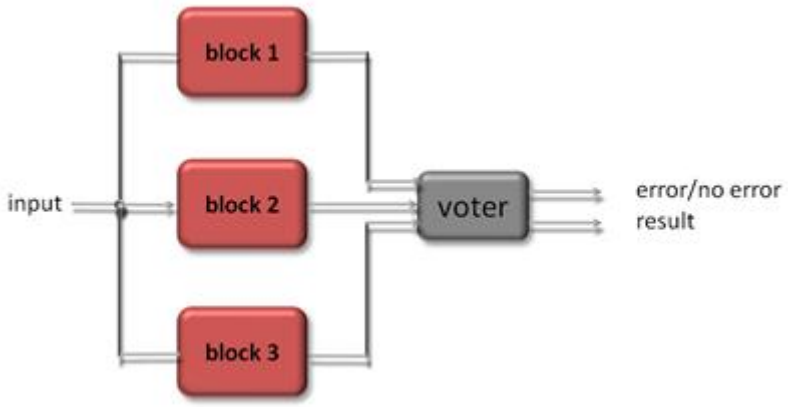


Figure 3-4. Triple modular redundancy (TMR) with a voter

Figure 3-4 above presents an example of a *TMR* system with a voter. The three components perform a process based on individual inputs whose results are in turn processed by a voting system to produce a single output. The voting is based on majority; if any of the three components has a fault, the other two systems can mask the fault. It is assumed that two out of three modules must deliver correct results. Therefore, *TMR* is capable of masking a single error.

Generally, a majority voting mechanism should:

- Guarantee a majority vote on the input data to the voter
- Determine the faulty block

In order to guarantee the majority vote, loosely synchronized systems require synchronization of the inputs to the voter.

A specific example of this technique is the *Boeing TMR 777* primary flight computer (Yeh, 1996), which has triple redundancy for all hardware including computing system, communication paths, electrical and hydraulic power.

3.4.1.2. Comparing the Reliability of Simplex and *TMR* with perfect voter[7] systems

A simplex system is a system with a single component. The reliability of a simplex system is given by:

$$R_{simplex} = e^{-\lambda t}$$

Equation 3.1. Reliability of a simplex system

where λ is the failure rate for the single component; the *MTTF* of a simplex system can be expressed as:

$$MTTF_{simplex} = \int e^{-\lambda t} = \frac{1}{\lambda}$$

Equation 3.2. MTTF of a simplex system

A *TMR* system such the one in Figure 3 includes three blocks, two of which are required for the system to provide correct service. The reliability of a *TMR* system with a perfect voter is given by:

$$R_{TMR} = R_m^3 + \binom{3}{2} R_m^2 (1 - R_m)$$

$$R_{TMR} = e^{-3\lambda t} + \binom{3}{2} e^{-2\lambda t} (1 - e^{-\lambda t})$$

$$R_{TMR} = 3e^{-2\lambda t}$$

Equation 3.3. Reliability of a TMR system with a perfect voter

[7] A perfect voter is a voter with a theoretical 100% reliability. A perfect voter is usually assumed to model extra reliable voters.

and therefore:

$$MTTF_{TMR} = \frac{3}{2\lambda} - \frac{2}{3\lambda} = \frac{5}{6\lambda}$$

$$MTTF_{simplex} > MTTF_{TMR}$$

Equation 3.4. MTTF of a TMR with a perfect voter

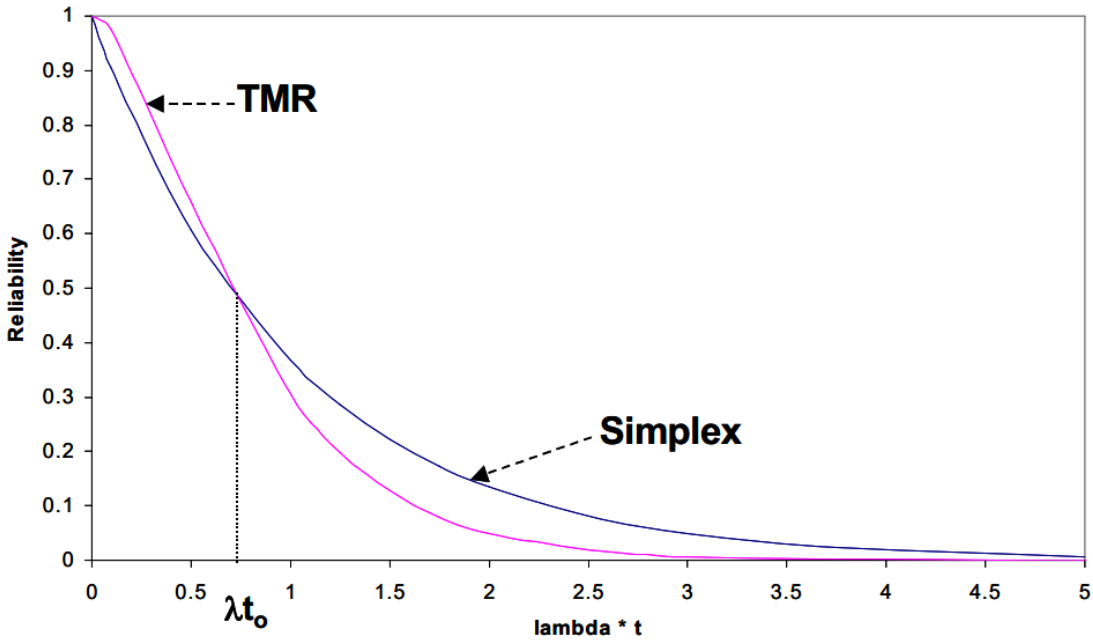


Figure 3-5. Comparative reliability of TMR and Simplex systems (Ravishankar K. Iyer, 2003).

Figure 3-5 shows how *TMR* has higher reliability than Simplex for short missions ($t<t_0$). Note that:

$$R_{TMR}(t) \geq R(t) \quad 0 \leq t \leq t_0$$

$$R_{TMR}(t) \leq R(t) \quad t_0 \leq t \leq \infty$$

Equation 3.5. Comparative reliability of *TMR* and Simplex systems (Ravishankar K. Iyer, 2003)

Where:

$$t_0 = \frac{ln\,2}{\lambda} \approx \frac{0.7}{\lambda}$$

TMR is very useful in aircraft applications offering high reliability for short missions. (Ravishankar K. Iyer, 2003) shows that TMR is not suitable for long safety-critical missions ($t>t_o$) because paradoxically, after the first failure, the two remaining components compete to fail. Higher reliability can be achieved extending *TMR* to *N-Modular Redundancy*. Therefore, a blind use of redundancy can lead to seemingly paradoxical results.

3.4.1.2.1. Reliability of *TMR* with voting

The previous expression of reliability of *TMR* assumes that the voter is perfect, i.e. the voter is 100% reliable.

The reliability of a generic TMR system with non-perfect single voting (*TMRV*) and identical blocks is given by:

$$R_{TMRV} = R_V \left(R_m^3 + \binom{3}{2} R_m^3 (1 - R_m) \right)$$

Equation 3.6. Reliability of a *TMR* system with a non-perfect voter and identical blocks

where R_v is the reliability of the voter mechanism and R_m is the reliability of the block. In terms of reliability, the voter becomes the weak part of this configuration. The voter is a *single point of failure* (*SPF*); if the voter fails then the complete system will potentially fail. This can be tackled following different alternatives:

- By increasing the reliability of the voter using fault avoidance techniques
- By triplicating the voter and connecting the module outputs to all three voters (Johnson, 1989) so that individual voting failures can be corrected by the extra voting process
- By implementing online self-testing for the voting circuitry (Cazeaux et al., 2004; Metra et al., 1997)

- Using an I_{DDQ} *checkable voters* (*ICVs*) (Bogliolo et al., 2000): under fault-free conditions, *ICVs* work as traditional *CMOS* voters; however, they cause *quiescent supply currents* (*IDDQs*)[8] in the presence of maskable stuck-at faults (see Section 5.3.1). Faults can be detected using I_{DDQ} *testing,* by monitoring *IDDQs* (Williams et al., 1996)

A basic TMR system does not support common-mode failures (*CMFs*[9]) (Lala and Harper, 1994). *CMFs* are the result of failures affecting more than one component, usually due to a common cause, which may be due to design-faults or operational ones resulting from external (such as radiation or electromigration) or internal causes. For instance, a radiation source causing multiple event upsets (Reed et al., 1997) can potentially lead to the failure of more than one component in a *TMR* system.

3.4.1.3. N-modular redundancy: *HW(nS)+HW(δS)*

The generalized version of *TMR* is *NMR* where *N* stands for the number of redundant modules. The main advantage of using *N* modules as opposed to only three is that often more faults can be tolerated. For instance, a *5MR system* contains 5 replicated modules including a majority voting arrangement. The voter allows the system to deliver correct service in case of as many as two module faults.

---

[8] Quiescent current is the current consumed by a circuit when no load is present. Fault-free *CMOS* devices have very quiescent currents when they are in a quiescent state. Faults that cause high quiescent currents can be detected if the quiescent current is significantly higher that that of a fault-free circuit (Williams et al., 1996)

[9] *Multiple faults* can be either *independent* (attributed to different causes) or *related* (attributed to a common cause. Both can lead to *similar errors* (e.g. errors that cannot be distinguished by the detection mechanisms being used) (Avizienis and Kelly, 1984). The failures triggered by similar errors are called *CMF*

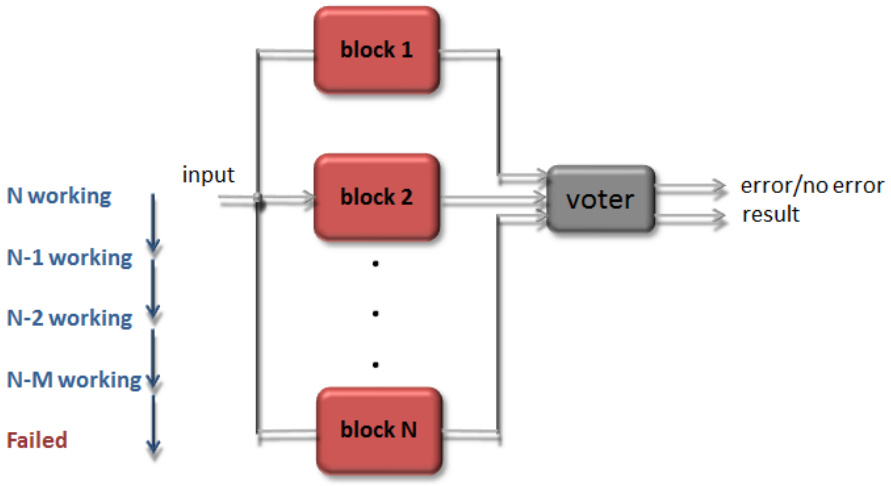


Figure 3-6. N-modular redundancy with a voter: M-out-of-N system

Figure 3-6 depicts a generic *N-modular* redundant system with a voter. The redundancy of this system can be defined as *HW(nS)+HW(δS)* using the previous notation. *NMR* works similarly to *TMR* but this type of structure is able to detect $[(N-1)\,]\,/\,2$ errors in different processing modules. Besides *TMR*, *5-* and *7-modular redundancies* are the most common structures and are capable of detecting two and three errors respectively.

M-out-of-N systems are a type of *NMR*. The reliability of a generic *M-out-of-N* system assuming that it has a perfect voter and *M* out of *N* modules need to function is expressed by:

$$R_{MN} = \sum_{i=0}^{N-M} \binom{N}{i} R_m^{N-i} \, (1 - R_m)^i$$

Equation 3.7. Reliability of an M-out-of-N system with perfect voter

Note than *NMR* systems offer higher reliability than *TMR* but at a much higher cost. Undoubtedly, for practical applications there must be some limit on the amount of redundancy that can be employed.

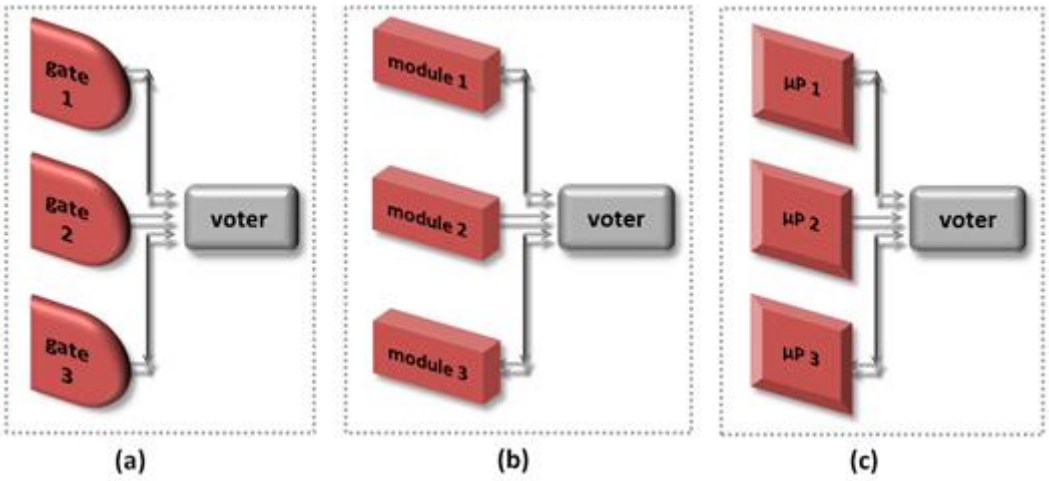


Figure 3-7. Redundancy applied on different levels of abstraction: (a) Three logic gates in a TMR at the logic or gate level of abstraction; (b) Three memory modules in a TMR configuration at the circuit abstraction level; (c) Three microprocessors in a TMR configuration at the chip level

TMR and *NMR* could be applied on different levels of abstraction by triplicating logic gates, single memory cells, memory modules or complete microprocessors. Figure 3-7 displays how *TMR* can be applied on logic (a), circuit (b) and chip level (c).

TMR and *NMR* are typically employed in aerospace applications where the cost of failure is particularly high. However, the higher reliability of these systems involves more than 200% increase in redundancy. Such an example is the *NASA* Space Shuttle onboard system, which is based on four computers with a majority voter (Sklaroff, 1976).

3.4.2. Dynamic redundancy

To reduce the extensive space, energy and performance requirements of *TMR* and *NMR* systems, numerous approaches have been developed. These approaches are usually based in dynamic redundancy, which implements error processing. This type of redundancy is similar to static redundancy with the main difference being the voter logic is replaced with a switch that is controlled by an error detection block. At least one of the modules is working as the main module, whereas the rest of the modules or replicas can either be working in

parallel (e.g. *DMR* with comparison) or can be turned off and used as spares (stand-by redundancy).

To avoid failures, after a fault has been detected, the system must be reconfigured. Detection, reconfiguration and recovery are required in order to prevent error propagation. Some examples of this type of redundancy are: pair and spare, duplex systems (*DMR* with comparison), backup sparing techniques etc.

3.4.2.1. Dual modular redundancy: *HW(2S)+HW(δS)*

By duplicating two components and adding a comparison structure, Dual modular redundancy (*DMR*), or duplex, are common systems to detect errors (von Neumann, 1956). *DMR* uses two fully redundant units working in parallel and has been widely used in low level circuit implementations where a signal is duplicated as an input to two redundant and independent logic gates and it is transparently checked for errors.

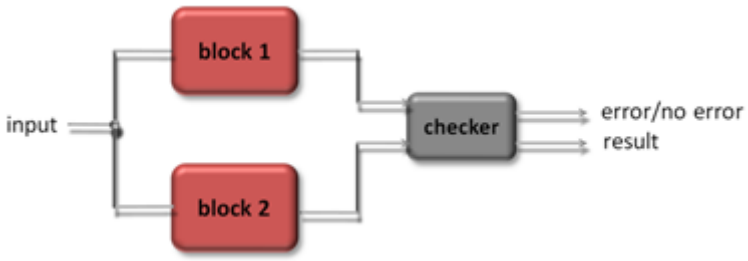


Figure 3-8. Dual modular redundant (*DMR*) structure

Figure 3-8 depicts a *DMR* structure with a checker component. The checker logic compares the output of block 1 and block 2. In the case of normal execution with no error, both blocks would produce the same output and a result would be delivered. On the other hand, in case of a mismatch between the two outputs of the blocks, the output of the checker would produce an error signal and no result will be given. Therefore, in its simplest version, as the checker logic is unable to identify the incorrect unit, DMR through output comparison will only provide error detection and will not provide error recovery capabilities on its own.

Additional mechanisms will be needed to provide error recovery so that if one of the units experiences an error, the surviving/correct unit can continue execution. Upon successful repair/recovery *DMR* is fully restored.

3.4.2.1.1. Redundant execution

A widespread and simple implementation of coarse-grained *DMR* is lock-stepping, or lock-step execution (Buckle and Highleyman, 2003; McEvoy, 1981; Sherman, 2003). Here, the processor pipeline is duplicated and the clock is shared, comparing each instruction result before committing the results. This type of error detection is considered to perform at the macro-level since it is applied at the microprocessor's scope.

Lock-stepping is widely used in a number of commercial processor designs and can both detect and correct certain errors; e.g. *IBM G5* (Slegel et al., 1999) and Compaq Himalaya (Wood, 1999). Redundant threads are executed in multiple processors and every instruction result is compared. No instruction can be committed until its identical pair has also been completed and verified, hence involving considerable overhead.

3.4.2.2. Standby redundancy

Standby redundancy, standby replacement, or standby sparing, is a well-known fault tolerant design technique used as a failover mechanism (Avizienis, 1976). In this case some units are online and operational and one or more backup units serve as standby units.

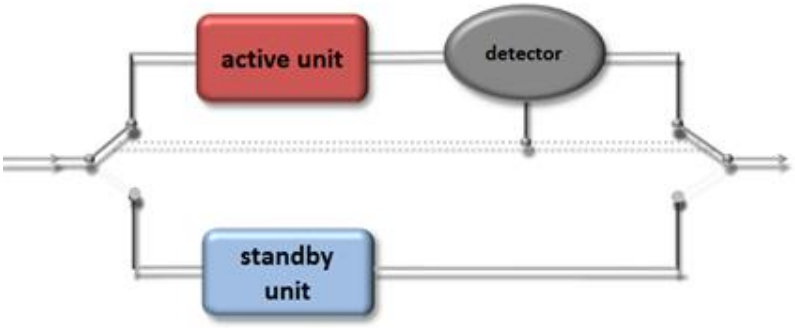


Figure 3-9. Simple standby sparing configuration

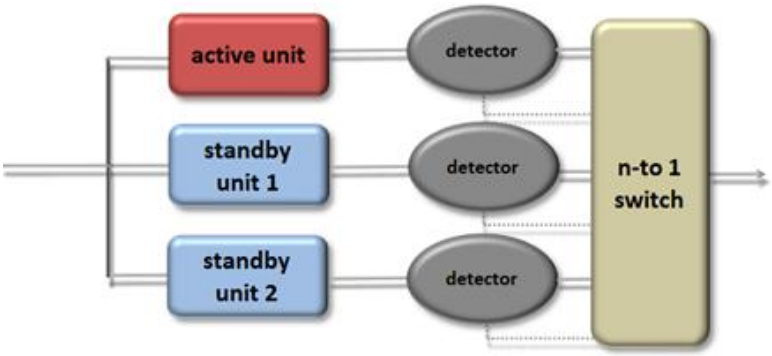


Figure 3-10. Multiple standby spares with n-to1 switch

Figure 3-9 and Figure 3-10 present simple and multiple standby configurations. When a fault is detected in an online/active unit, a standby unit replaces the affected unit by using the selector (Figure 3-9) or by using the 3-to-1-switch (Figure 3-10).

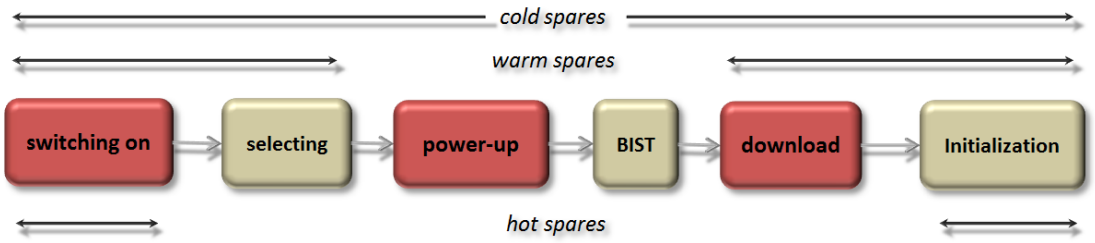


Figure 3-11. Typical reconfiguration steps for backup sparing

There are three common forms of standby redundancy: hot, warm and cold. The type of application plays a key role in selecting the type of standby spare units. Figure 3-11 graphically describes the typical reconfiguration steps for hot, cold

and warm backup spares. When a spare unit is to be switched over, the selected spare is powered up and gets ready to become active. The reconfiguration process whereby a standby spare unit becomes operational is composed of:

- Switching on the power and the bus connections
- Powering up of the unit
- Running the *Built-In-Self-Test*[10] (*BIST*): Extensive testing is usually done after powering up to avoid replacing a faulty module with a faulty module before starting normal operation, e.g. memory tests of a spare module
- Loading programs and data
- Initializing the software if needed

Hot standby spares (HSP) operate in synchrony with the operational units and are ready to take over whenever a fault is detected. *HSP* units reduce the mean time to recovery (*MTTR*) and therefore their use is suitable for applications that require short recovery time, that is, applications where the disruption of processing must be minimized.

Cold standby spares (CSP) remain unpowered and thus do not operate or consume any power until they need to replace an active unit. Since the restarting of the units is required, the use of *CSP* is best suited to remote operations where power is hard to come by, e.g. satellites and sensor systems. *CSP* units are also suitable for applications where short lapses in operation are acceptable and state data is not critical. In addition, *CSP* are likely to have a lower failure rate than operational modules. However, the startup delay required to switchover to a spare module is high since power up, *BIST* and initialization are needed. In

[10] Built-In-Self-Tests (*BISTs*) are one of the common methods of testing circuits. *BIST* is a *DFT* technique that takes place on the same substrate as the device under test (*DUT*) within the system allowing them to perform self-testing (Stroud, 2002).

particular, the time necessary for *BIST* depends upon the fault *coverage* and the complexity of the unit/module.

Warm standby spares (WSP) consist of a trade-off between the high power consumption of *CSP* and the long recovery time of *HSP*. *WSP* units have time dependent behaviour. Before and after replacing an operational unit, *WSP* present different failure distributions.

The advantage of standby sparing for a system with n identical units is that a certain level of fault tolerance can be provided with $k<n$ spare modules.

3.4.2.3. Pair and spare

The *pair and spare* configurations are a combination of *DMR* with comparison and extra spare techniques.

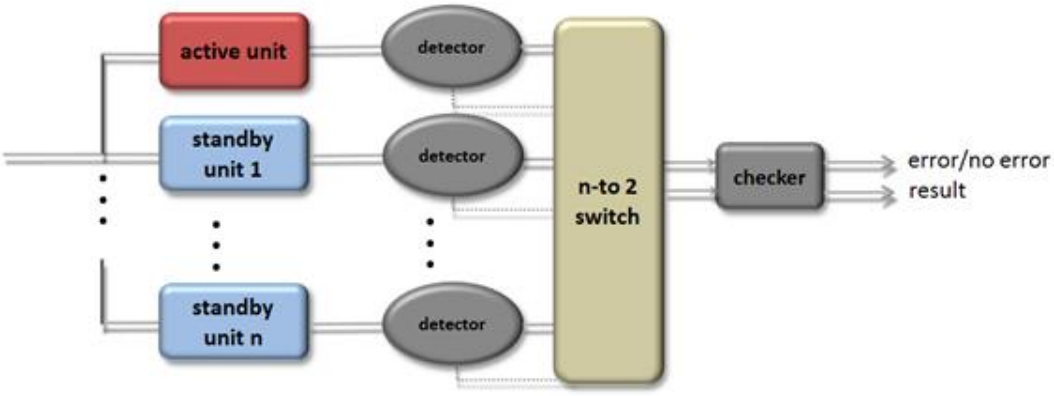


Figure 3-12. Pair and spare configuration

Figure 3-12 depicts a pair and spare configuration where two units are always online and compared to each other, with any of the n spares being able to replace either of the operational units.

3.4.3. Hybrid redundancy

By mixing fault masking, detection location and recovery, the advantages of static and dynamic redundancy can be combined (Johnson, 1989). Hybrid

approaches use Fault masking to prevent erroneous results from being activated. Fault detection, location and recovery are also employed in hybrid techniques to improve fault tolerance by removing errors.

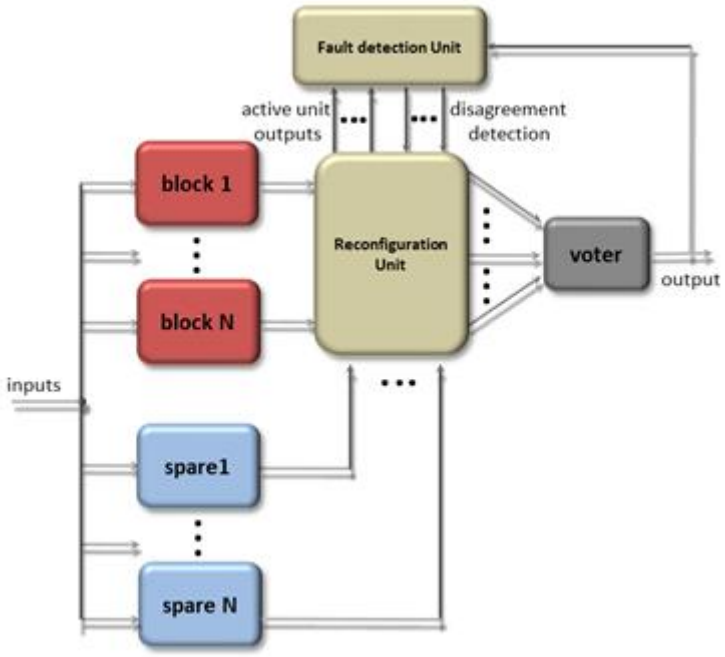


Figure 3-13. Hybrid approach using TMR with spaces (Johnson, 1989)

A general approach is to back up the replicated modules with spares, e.g. a *TMR* configuration with a fault/disagreement detector, a voter and a reconfiguration unit (see Figure 3-13). In such a system, the triplicated operational modules are backed up with an additional pool of spares that can replace faulty modules (*TMR with spares*). The system will work as a basic *TMR* configuration until the disagreement detector determines that a faulty module exists. One alternative approach towards fault detection is to feed the output of the majority voter back to the faulty detection unit whose job is to compare the output of the voter with the individual outputs of each operational module. Any disagreement with a specific module's output would indicate that the module should be labelled as faulty and therefore replaced by a spare unit. The reliability of the basic *TMR* system is retained as long as the pool of spares is not exhausted. Note that voting only occurs among the operational modules in the *TMR* core, masking faults and making sure that continuous correct service is delivered.

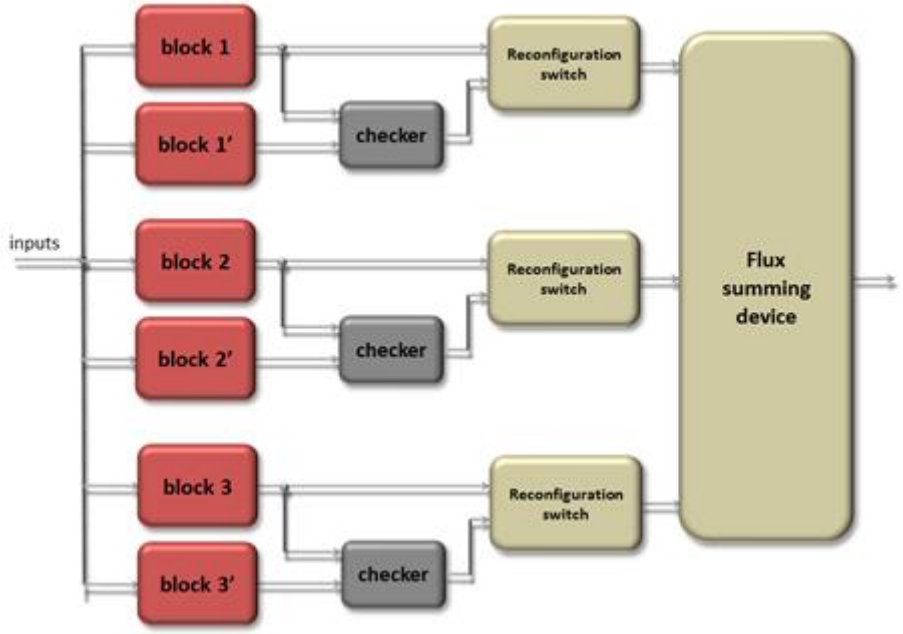


Figure 3-14. A triple-duplex approach

A variation to *NMR* with spares is the triple-duplex approach depicted in Figure 3-14 that combines duplication with comparison and *TMR*. The use of passive redundancy in the form of *TMR* allows potential faults to be masked and continuous correct service to be provided for a maximum of two faulty modules. The use of *DMR* with comparison allows faults to be detected and faulty modules to be removed from the voting process and replaced by spares.

These options are simple but far more expensive in terms or real estate of the chip than traditional static techniques. Besides, as seen in section 3.4.1.2.1, the reliability of *TMR* depends mostly on that of the voters. Hence, if a fault takes place within a voter, an incorrect majority vote may be given to the output and propagated throughout the system thus compromising the correctness of the system's service. In order to avoid such unreliability, voters can be designed to be capable of testing themselves online with regards to their own internal faults (Cazeaux et al., 2004; Metra et al., 1997, p. 97).

An extra form of hybrid scheme that allows error detection and correction thus improving the reliability of a memory system is depicted in Figure 3-15:

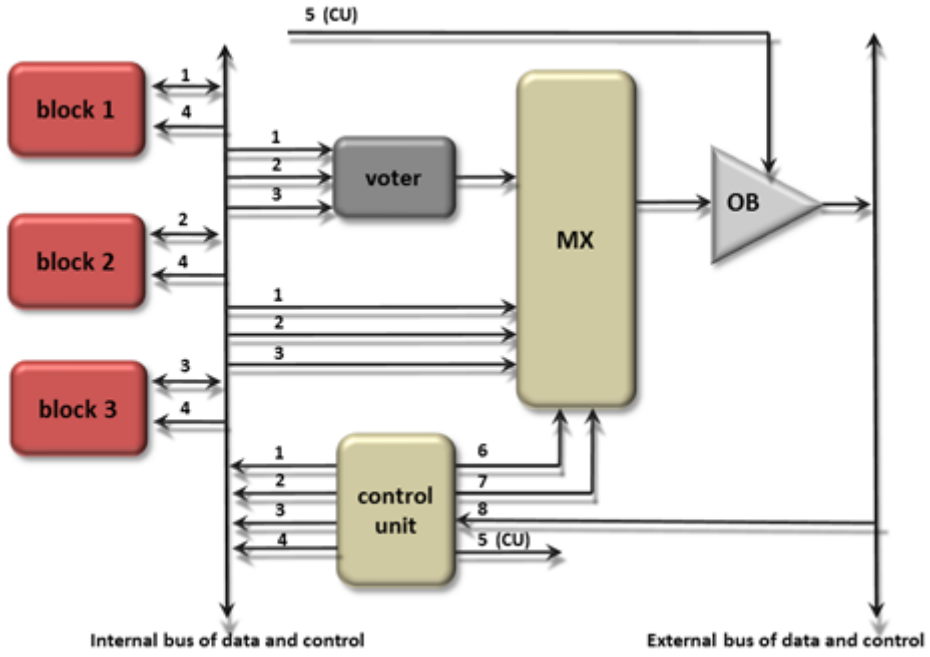


Figure 3-15.Transient faults tolerant TRAM (Schagaev and Buhanova, 2001)

Any reading and writing operation is followed by a content check of a specific address in all three blocks. In case of a mismatch among these a majority voting takes place whose result is then rewritten (via control unit) to the inputs of all elements using the same address.

3.5. Information redundancy

Information redundancy involves the addition of new information to existing information, often in compressed form i.e. using more bits than needed. The most common form of information redundancy is *coding* (see Figure 3-16). Coding theory in hardware and software fault tolerance goes back a very long way and was initially motivated by the need to mitigate errors in information transmission (Shannon, 1948).

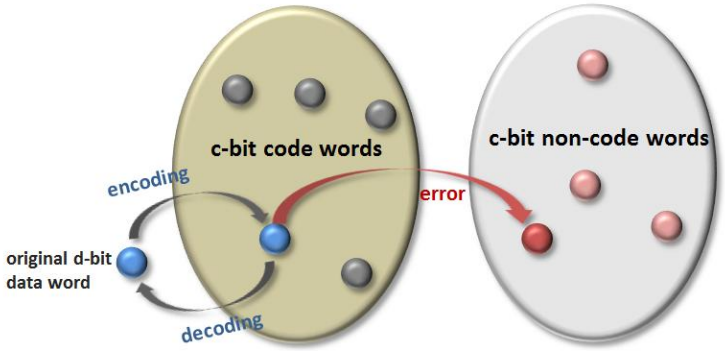


Figure 3-16. Coding-encoding process of a d-bit word into a c-bit word

Coding consists of adding check bits to the data allowing 1) the verification of data correctness and/or 2) the correction of erroneous data. Therefore, with coding an original piece of meaningful information, or *d-bit* data word, is encoded obtaining a *c-bit* code word, where $c>d$ (see Figure 3-16). Because of these extra bits not all 2^c possible binary combinations are valid code words. Therefore, a code should be selected so that any potential error would transform the codeword, after decoding, into an invalid code word (non-codeword).

An important property of coding is separability. Two main approaches are possible:

- Separable or systematic codes: the code word is formed by adding extra information (check bits) to the original data. A separable code has separable fields for data and check bits. Decoding this type of code is simple and consists of selecting the data bits and disregarding the check bits.
- Non-separable or non-systematic codes: data and check bits are integrated together requiring some extra processing and therefore incurring additional delays and overheads.

Important parameters for codes are:

- The number of erroneous bits that can be detected
- The number of those that can be corrected
- The number of additional bits that are required
- The time needed to encode
- The decoding time

Information redundancy techniques make use of detection-based codes (EDC) or correction based (*ECC*) codes. Figure 3-17 presents a taxonomy of coding techniques.

Examples of some of these techniques include:

- Error detection and correction codes for cross-checking the contents of main memory, register files and cache,
- Cross-checking of run-time control flow using signatures and
- Algorithm based checksums for cross checking of the data values generated

In general, information redundancy involves some space and computational overheads, thus requiring extra circuitry and is thus more commonly implemented in memory structures instead of in processor datapaths.

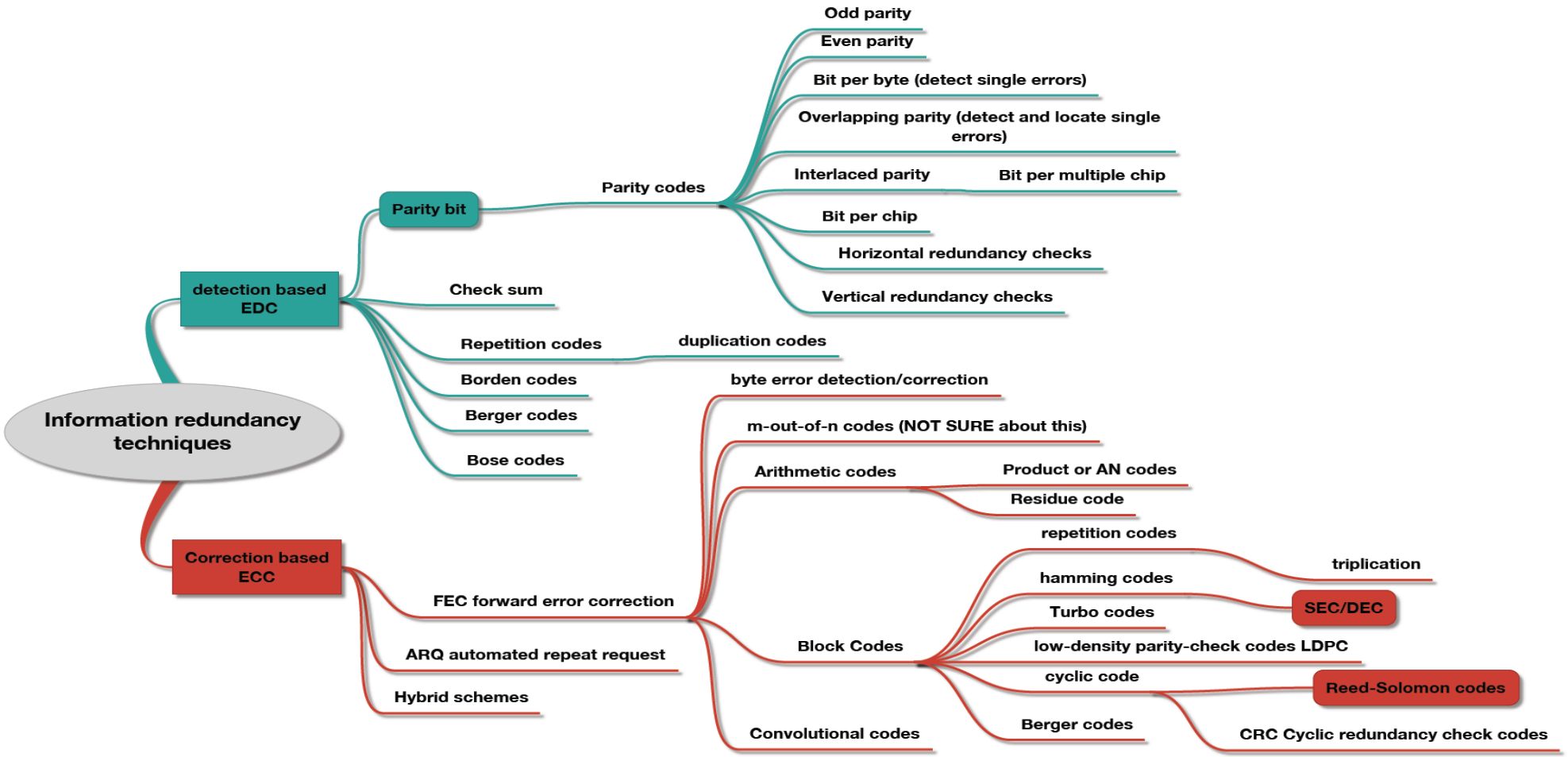


Figure 3-17. Taxonomy of information redundancy coding techniques

3.5.1. Error Detection Codes: EDC

Error detection codes have the ability to expose error(s) in a given data word based on the encoding-decoding principles discussed in Section 3.5, In general, error detecting codes (*EDC*) present less overhead than error correcting codes (*ECC*) since they do not have correction capabilities.

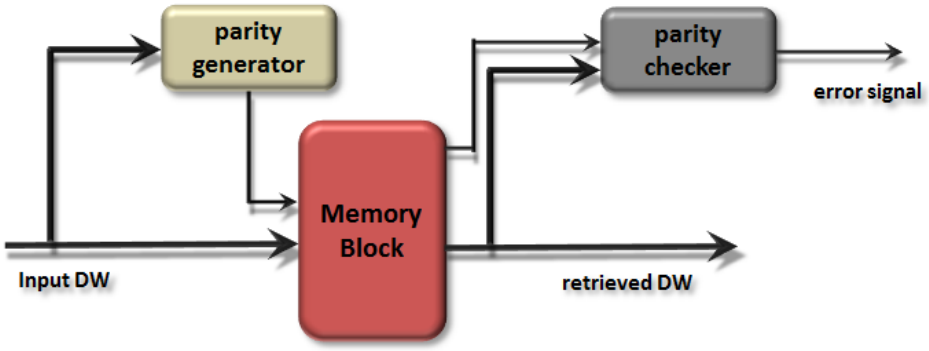


Figure 3-18. Coding-encoding in a memory block with parity checking

The simplest *EDC* are parity codes, which involve the addition of extra bits or parity bits. Figure 3-18 depicts a basic scheme of memory with parity checking. Before storing a word in the memory block a parity generator computes the parity bits based on the bits of the input data word (*DW*). A parity bit is an extra bit added to a group of source bits (*DWs*) in order to ensure that the outcome or coded word has an even (in the case of even parity) or odd (in case of odd parity) number of bits set to 1. When a memory block is read, the parity checker compares the computed and the stored parity bits, setting the error signal consequently. If both, computed and stored parity bits, match then the error signal would indicate a correct output; otherwise the error signal would indicate that the retrieved *DW* is incorrect. Note that for n bits of data there are 2^n possible *DWs*. Adding one parity bit would allow 2^{n+1} possible *DWs*. Among these possible *DWs* there are $\frac{2^{n+1}}{2}$ possible *DWs* with an odd number of 1s and $\frac{2^{n+1}}{2}$ possible *DWs* with an even number of 1s. In the case of odd parity, only the *DWs* with an odd number of 1s are valid code words (*CWs*). In the presence of a

single bit flip (error) an odd parity *CW* would change into an even parity *CW* and therefore the parity checker will detect the error. Nonetheless, it will not know which bit has been flipped. This simple configuration can be used to detect single or any odd number of errors in the retrieved *DW*. However, an even number of flipped bits would make the parity bit of the *CW* appear to be correct although the data is incorrect. With single parity, double errors and even number of errors would remain undetected.

3.5.2. Error Correction Codes: ECC

More powerful codes than parity codes can be created by adding more check bits to the original data. The size of the data to be protected will determine the number of check bits needed. Using this basic principle, error correction codes have the ability to detect errors and reconstruct the original error-free data. These can generally be realized in three different manners (see Figure 3-17):

- *Backward Error Correction (BEC)* sometimes referred to as *Automatic repeat request (ARQ)*: combines an error detection technique (error detection encoding prior to transmission) with retransmission request of erroneous data. BEC requires simpler decoding infrastructure than *FEC* but frequent retransmissions would significantly compromise performance in high data rate transmissions.
- *Forward Error Correction (FEC)* or *Channel Coding*: With this approach, errors are both detected and corrected at the receiver's end. Thus, it involves error-correcting encoding prior to transmission without retransmission of the original information. *FEC* requires more complex decoding infrastructure than *BEC* but it is suitable for high data rate applications.
- *Hybrid automatic repeat request (HARQ)*: *BEC* and *FEC* are combined. e.g.: a scheme where minor errors are corrected without retransmission (*FEC*) and major errors are corrected via retransmission (*BEC*).

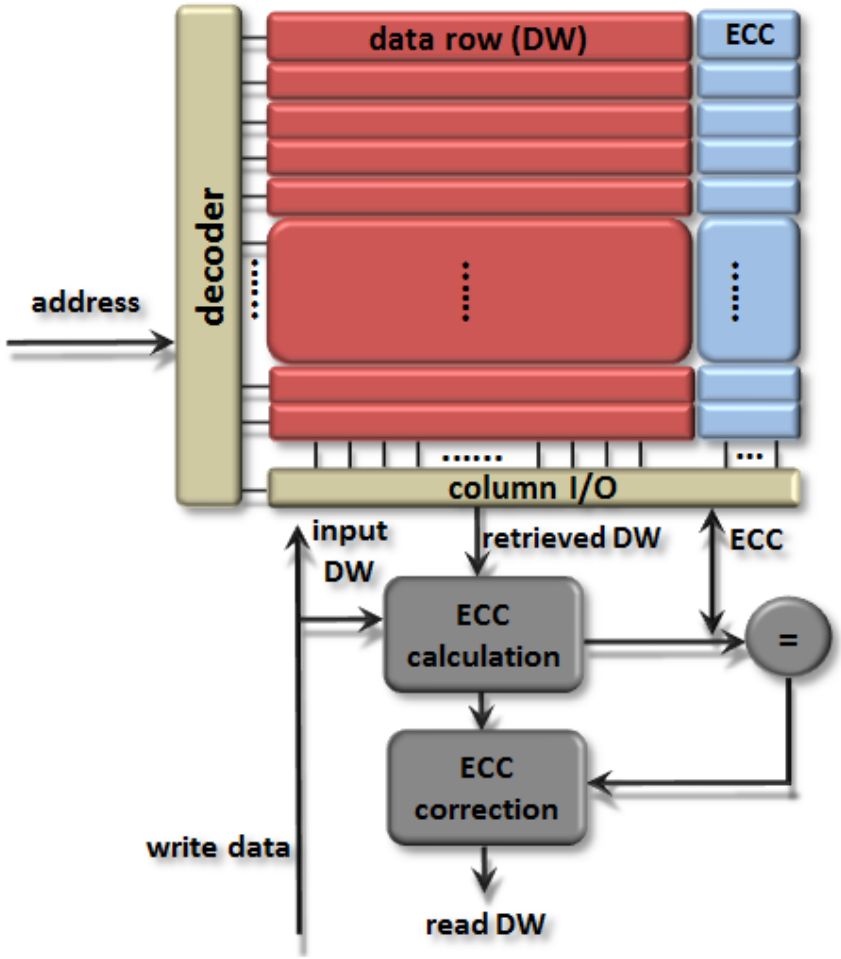


Figure 3-19. Basic ECC memory scheme including calculation, checking and correcting

An overview of popular *FEC* schemes employed in fault tolerant design of embedded systems follows. Figure 3-19 shows a basic *ECC* memory scheme that applies to any of the following codes including calculation, checking and correcting logic. When data is written into the data row specified by the address signals, the *ECC* encoding logic generates the parity checks (as specified by the code) and introduces them into the *ECC* part of the memory. When the DW is read from the memory the parity bits would allow missing data to be reconstructed in the case of an error being detected.

3.5.2.1. SEC-DED[11]: Hamming and Hsiao: HW(δI)

The most common *ECCs* are based on Hamming (Hamming, 1950) or Hsiao (Hsiao, 1970). These two separable code families introduce the concept of overlapping parity by which every data bit has a part in adjusting the value of several parity bits. These codes can correct single bit errors in a given word, can detect double bit errors, are relatively fast decoding and have moderate redundancy.

Hamming codes are a family of perfect codes[12] that generalize the original Hamming(7,4)-code (Hamming, 1950). A minimum distance d means that it takes d bit changes to move from one valid codeword to the other. Extended Hamming code sometimes generalized as *SEC-DED* (single error correction and double error detection), is an example of this type of code. In *SEC-DED*, an extra parity bit is added achieving a distance of four instead of the three (as in the original Hamming). The extra parity bit allows the decoder to distinguish between two possible situations:

- When at most one bit flip has occurred and
- When two bit flips have taken place

In contrast with *Hamming(7,4)*, *SEC-DED* provides single-bit-error correction and simultaneous double-bit-error detection.

Compared to Hamming codes, Hsiao codes (Hsiao, 1970) provide improvements in speed, reliability and calculation cost as well as checking and correcting logic.

---

[11] SEC-DED: Single error correction and double error detection

[12] A Hamming code is *perfect* in the sense that it can achieve the highest possible rate for codes with a given block length and minimum distance of three (Moon, 2005)

However, in situations that demand higher reliability requirements than those provided by *SEC-DED*, more complex codes are required.

3.5.2.1.1. SEC-DED limitations and alternative techniques

The main limitation of *SEC-DED* codes is that triple-bit errors may not only remain undetected but it may also be miscorrected as if they were single-bit-errors (Hsiao, 1970). The probability of this type of miscorrection for 32bit data words is around 60% or more.

Multiple errors are usually taking place in adjacent memory locations, therefore increasing the chances of having multiple bit errors in a given word (Bentoutou and Djaifri, 2008; Boatella et al., 2009). These are called *burst errors*[13], errors that are highly correlated. If a specific memory cell has an error, it's likely that adjacent cells may also be corrupted by the same event that triggered the error in the first place. Theses are sometimes referred to as *spatial multi-bit errors* (Mukherjee et al., 2004). In contrast, *temporal multi-bit errors* are errors that take place when two different cells of the same word are affected by different events (Mukherjee et al., 2004).

An important risk for *SEC-DED* schemes is that if a specific memory word is not accessed for a long period of time, the chance of accumulating errors increases (temporal multi-bit errors). One method to avoid these is the use of *memory scrubbing* (Mukherjee et al., 2004; Saleh et al., 1990; Weaver et al., 2004) , by which every memory location is read periodically. This may be implemented by having a hardware controller that, during idle periods, reads every memory location searching for errors and correcting any single error found during the

[13] Also called cluster of errors

process, thus reducing the chance of detected (*DRE*[14] and *DUE*[15]) and undetected errors (e.g. *SDC*[16]). Scrubbing does, however, impose additional *SW* and/or *HW* overheads depending on the implementation. In current architectures with high memory bandwidths, *HW* scrubbing is preferred due to its lower timing overhead. In combination with *SEC*, scrubbing is effective against single-bit- and temporal multi-bit errors but not against spatial multi-bit errors.

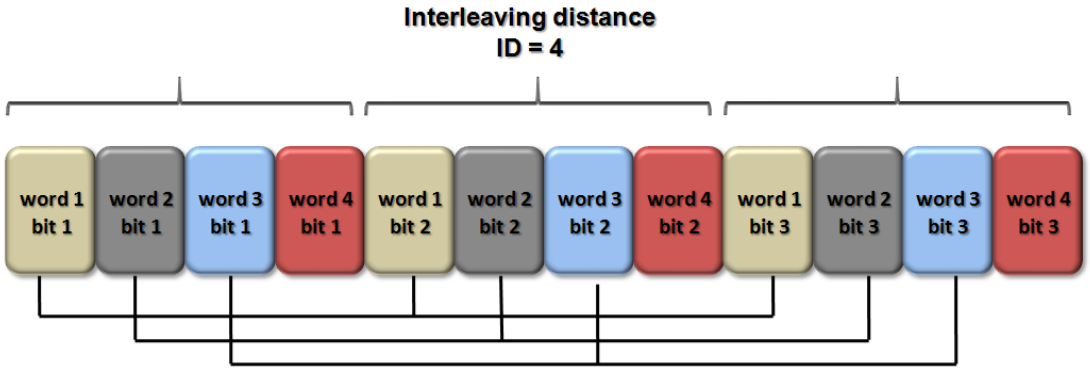


Figure 3-20. Memory interleaving of four 3-bit words with a 4 interleaving distance (ID)

To avoid this problem of spatial multi-bit errors, *memory interleaving* (Haraszti, 2000; Reviriego et al., 2010, 2007) is commonly used in conjunction with *ECC* ensuring that cells that are physically closely located belong to different logic

[14] A *DRE* is a detected recoverable error, a benign type of error since recovery of the normal operation by fault tolerant techniques is possible (Kadayif et al., 2010; Weaver et al., 2004)

[15] A *DUE* is a detected unrecoverable error. *DUE* take place when fault tolerant techniques are able to discover and/or report an error, from which recovery is not possible (Kadayif et al., 2010; Weaver et al., 2004)

[16] *SDC* stands for Silent data corruption. A *SDC* take place when an error is undetected and causes data corruption (*SDC*). In this case, the corrupted data could go unnoticed making this type of error benign, or could result in a visible error and/or catastrophic failure such as crashing a computer system (Constantinescu et al., 2008; Kadayif et al., 2010; Weaver et al., 2004)

words. That is, cells that belong to the same logical word are physically apart. Figure 3-20 illustrates an example of memory interleaving in a four *3-bit* memory word. This type of memory distributes logical data into a non-continuous arrangement. More columns than the number of bits of a single word are added, and the corresponding columns for each word are interleaved. In this way, *burst errors* are distributed over a number of words each suffering only one single bit error. Any *4-bit-upset* affecting adjacent memory cells would cause four single bit errors in separate words, which can be easily corrected by *SEC-DEC*.

A shortcoming of interleaving is that high interleaving distances (*ID*) involve more complex designs and thus higher area and latency overheads (Baeg et al., 2009; Reviriego et al., 2010). Ideally the ID should be selected as the maximum expected *MCU* size so that all upsets in a burst error would occur in different logical words.

Table 3-4. *ECC-TMR* comparison

Characteristic	***Hamming (SEC-DED)***	***TMR***	***RS (DEC-TED)***	***BCH (DEC-TED)***
Area	Small overhead to implement Varies depending on the number of bits (7-32%)	Extra 200% plus the voting and correcting logic Number of voters is proportional to the number of units	Varies depending on the number of bits (13-75%)	Varies depending on the number of bits (13-75%)
Performance	It can be affected by the coder-decoder functions Proportionally dependent on number of bits to be corrected	High performance. Voter is the only source of delay, hence almost constant delay	Lower performance than *BCH* and much lower compared to Hamming and *TMR*	Higher performance than RS but much lower than Hamming or *TMR*
Error Correction	Limited capabilities: it corrects only one single incorrect bit per word.	Corrects up to n errors in an n-bit word as long as the errors are located in a distinct position/unit.	Can handle multiple errors; Efficient for correlated errors (e.g. burst)	Can handle multiple errors; Efficient for uncorrelated errors (e.g. random errors)
Implementation	Binary based Simple to implement	Simple to implement	Symbol based Complex to decode and implement	Binary based Complex but simpler to decode and implement than *RS*

3.5.2.2. Complex codes

EDAC implementations based on Hamming codes are the easiest to implement but only provide single error correction (Hentschke et al., 2002). There are alternatives to *SEC-DED* like Bose-Chaudhuri-Hocquenghem (*BCH*) (Bose and Ray-Chaudhuri, 1960) and Reed-Solomon (*RS*) codes (Reed and Solomon, 1960) based on finite-field arithmetic that can correct multiple faults.

Table 3-4 shows a comparison of the main error correction techniques in memories. *BCH* codes are able to correct a given number of bits at any position whereas *RS* codes group the bits in blocks in order to correct them. *RS* based codes provide a more robust error correction capability but uses a large amount of system resources[17]. The *RS* decoding process has several stages to get the location of the error and correct it. Implementations of *RS* codes can be found in (Neuberger et al., 2005, 2003). Although the *RS* algorithm can be simplified (Neuberger et al., 2003) the main disadvantage of these two codes is having complex and iterative algorithms.

Table 3-5. *EDC-ECC* storage array overheads, based on (Slayman, 2005).

Data bits	Single Parity		SEC-DED		SNC-DND		DEC-TED	
	check bits	overhead	check bits	overhead	check bits	overhead	check bits	overhead
16	1	6%	6	32%	12	75%	11	69%
32	1	3%	7	22%	12	38%	13	41%
64	1	2%	8	13%	14	22%	15	23%
128	1	1%	9	7%	16	13%	17	13%

[17] *DEC-TED* implementations are expensive from both area-penalty and computational-complexity points of view

As with hamming based *SEC-DED*, more complex codes can be implemented based on RS and BCH algorithms. Some examples are *SNC-DND*[18] (Chen and Hsiao, 1984) and *DEC-TED*[19] codes (Lin and Costello, 1983). Table 3-5 is an overhead comparison of various *EDAC* schemes: Single parity *EDC*, Hamming SEC-DED, SNC-DND and DEC-TED. Note that the calculation of overheads is just the number of check bits divided by the number of data bits and does not include the extra overheads (e.g. I/O and checkers). Complex errors increase the overhead rapidly as correction capability is increased (Kim et al., 2007). For any given technique, as the data size increases, the relative overhead of a given scheme decreases (Table 3-5).

In addition to the area penalty, as the correction capability increases, timing overheads also increase. Results on 64kb *SRAM* developed in 90nm processes show that the implementation of a *DEC-TED* encoder involves a latency penalty of 80% to 85% as compared to *SEC-DED* (Naseer et al., 2006).

Schemes based on information redundancy can also be applied on different levels. For instance parity codes can be applied to registers, cache and internal memory whereas *SEC-DED* can be implemented in external memory, etc. As all these are more complex codes than *SEC-DED* let alone single parity codes they produce higher overheads as the correction capability increases (Kim et al., 2007) and are thus not suitable for areas of real-time systems that demand high possessing performance.

[18] *SNC-DND*: single nibble error correcting, double nibble error detecting

[19] *DEC-TED*: double bit error correcting, triple bit error detecting

3.6. Time redundancy: *HW(T)*

Figure 3-21 shows a list of the most relevant techniques based on time redundancy, which are fully described in Sections 3.6.2, 3.6.3, 3.6.4, 3.6.5 and 3.6.6.

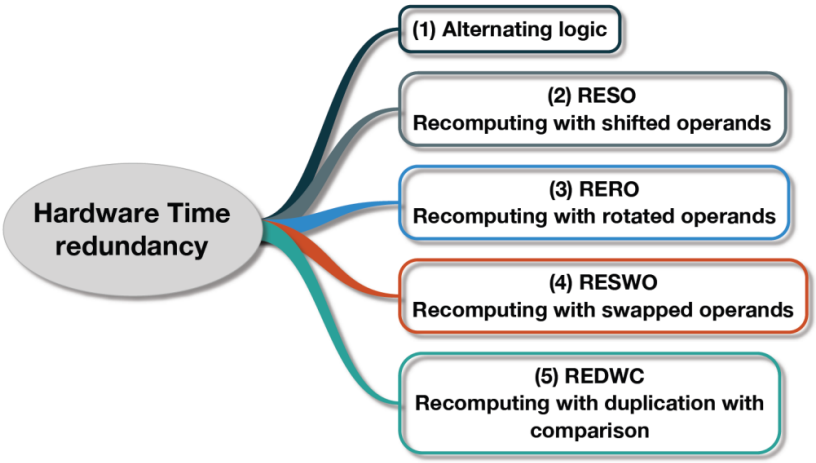


Figure 3-21. Taxonomy of time redundancy techniques

3.6.1. Concurrent error detection: Basics of time redundancy

The main problem with the space and information redundancy types reviewed is the penalty imposed in the form of extra hardware. At the expense of using additional time, *FT* techniques based on time redundancy (*TR*) aim to reduce the amount of hardware required for the implementation. Time redundancy techniques involve the re-execution of code using the same piece of hardware and comparing the two execution results to determine if a fault has occurred. This approach was commonly used in the past and is effective in detecting errors resulting from transient faults.

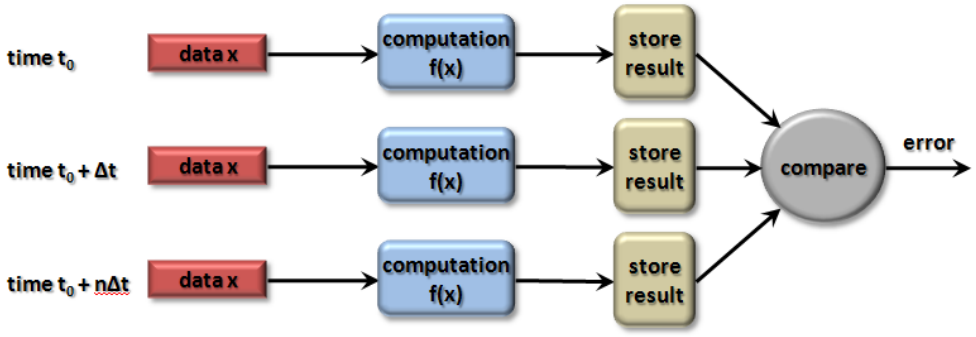


Figure 3-22. Transient fault detection mechanism based on redundant execution

Figure 3-22 shows the basic transient fault detection mechanism based on re-execution. With this technique two or more different computations are performed at different times t_0, $t_0+\Delta t$, and $t_0+n\Delta t$ given $n>1$. The result of a given computation is stored in the corresponding register and then compared to the results obtained from the previous computation(s). If the re-execution is performed twice and a disagreement exists, then transient errors can be detected. This type of technique was used in the past, but on its own, and did not provide protection against errors due to permanent faults. However, the executions can be performed again to check if the discrepancy remains or not. This is useful in order to distinguish between permanent and transient faults. If after re-execution the fault disappears, it is assumed to be transient. The hardware resource affected by a transient fault is still usable. On the other hand, if after re-execution the problem persists, the fault is assumed to be permanent and reconfiguration of the specific hardware resource is necessary.

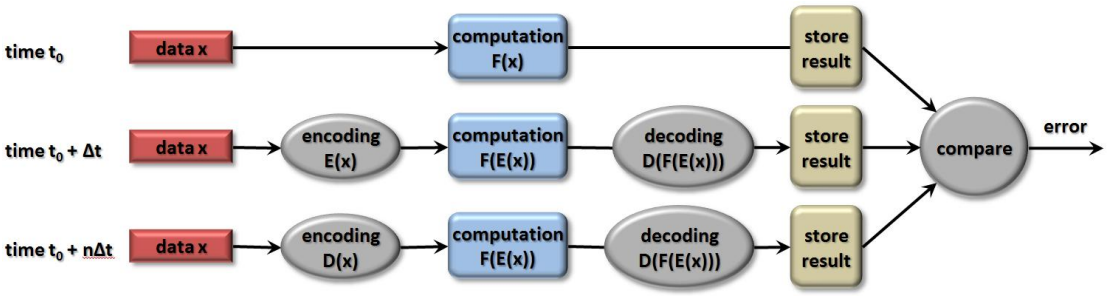


Figure 3-23. Transient and permanent fault detection mechanism based on redundant execution

Modern *FT* techniques based on time redundancy can detect permanent faults as shown in Figure 3-23. In this case, during the first computation, the results obtained are simply stored in a register. Then, prior to the next computation(s) a specific type of encoding is performed on the operands. After the relevant computation(s) take(s) place on the encoded operands, the results of all computations are then decoded and compared.

Given that x is the input data, *E(x)* is the data decoding, *F(x)* is the functional computation, *F(E(x))* is the functional computation of the decoding data and *D(E(f(x)))* is the decoding of the encoded data after computation, and assuming that the functional block is free of permanent faults and assuming that:

$$\forall\, x \quad D\big(E(x)\big) = x$$

Equation 3.8. Encoding-decoding relationship

Then, the following relation can be stated:

$$\forall\, x \quad D\left(F\big(E(x)\big)\right) = F(x)$$

Equation 3.9. Relationship among encoding, decoding and functional computation

If the decoder and the encoding process are carefully selected so that a failure in *x* would affect *F(x)* differently than it would affect *F(E(x)* then if $\Delta t>0$ the comparison mechanism would produce an error signal.

The main problem with time redundancy techniques is that if the system's data is corrupted by a transient or permanent fault, it will be difficult to repeat a given computation. The critical part of these techniques is assuring that the data is correct and identical before each one of the redundant computations take place. The leading concurrent error detection (*CED*) techniques based on time redundancy are *alternating logic* (Reynolds and Metze, 1978), *recomputing with shifted operands* (*RESO*) (Patel and Fung, 1982), *recomputing with rotated operands* (*RERO*) (Li and Swartzlander, 1992), *recomputing with swapped operands* (*RESWO*) (Hana and Johnson, 1986) and *recomputing with comparison*

(*REDWC*) (Johnson et al., 1988). All these techniques work as specified in Figure 3-23. The main difference among them is the type of encoding and decoding used.

3.6.1.1. Self-duality

Self-duality is a property required for certain circuit's functions in order to implement specific error detection techniques based on time redundancy. A function is said to be self-dual if it satisfies the property:

$$\forall\, x \qquad f\left(x_{1\_}, x_{2\_}, \dots, x_{3\_}\right) = f\_(x_1, x_2, \dots, x_3)$$

Equation 3.10. Property of self-duality

Where x_1, x_2, x_3, …, x_n is the set of inputs to the circuit, $x_{1\_}$, $x_{2\_}$,…,$x_{n\_}$ the set of complemented inputs, *f()* the output and *f_()* the complemented output.

By letting *C* be a function that complements each bit of a given vector:

$$\forall\, x \qquad C\left(x_{1\_}, x_{2\_}, \dots, x_{3\_}\right) = (x_1, x_2, \dots, x_3)$$

Equation 3.11. Property of self-duality

It becomes clear that:

$$C^{-1} = C$$

$$C^{-1}\left(f\left(c(x)\right)\right) = f(x)$$

Equation 3.12. Complementary function

Resulting in:

$$C\left(f(x)\right) = f\left(C(x)\right)$$

Equation 3.13. Complementary function and self-duality

There are several problems that must be considered when designing a fault tolerant technique using time redundancy. A function *C* that satisfies the

previous property must firstly be determined. Finding a *C* may not guarantee the desired level of error detection since different circuit implementations based on different *C* can have different coverage. Complexity is also an important issue. In the case where the hardware required to implement the coding and decoding functions based on *C* and *C-1* is similar to that of implementing *f(x)*, then structural redundancy becomes the more effective choice. In short, the aim of a cost effective design should be finding a function *C* that provides a good trade-off between high coverage and low complexity.

3.6.2. Alternating logic

An example of encoding/decoding function is the complementation operation used in *alternating logic* (Reynolds and Metze, 1978) and successfully applied to permanent fault detection of data transmission and digital systems.

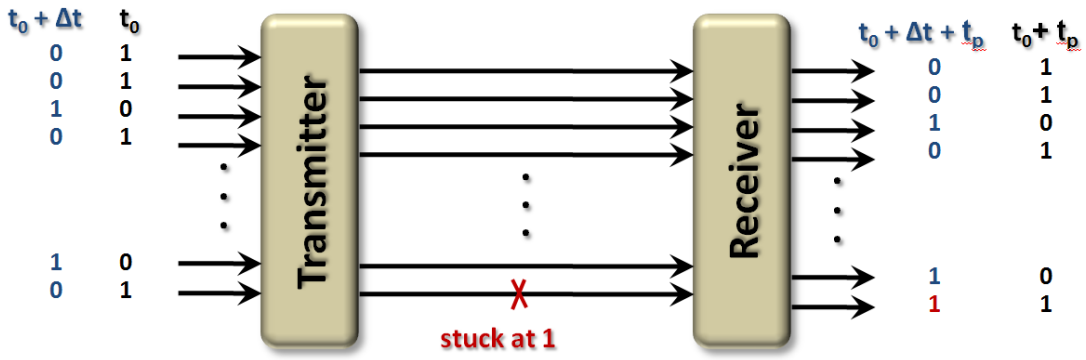


Figure 3-24. Time redundancy technique based on alternating logic

As shown in Figure 3-24, the data computed at time t_0 is then complemented and transmitted at time $t_0+\Delta t$. In the case of a stuck line (either at 0 or at 1) the two computations will generate data that are not complement of each other and therefore the error signal will become enabled after comparison. In the case of

Figure 3-24, the last communication line is *stuck at 1*[20] and therefore complement and data would both become 1, which is not an alternate output and therefore a fault is detected. In order to implement error detection in this coding the circuit function must have the property of *self-duality* otherwise extra input bits would be required. For certain circuits 100% area overhead may be required for certain error detection circuits in addition to time redundancy (Carter and Schneider, 1968; Johnson et al., 1988; Woodard and Metze, 1978).

The key for fault detection is to determine that at least one input vector exists for which the fault will not result in alternated outputs. Although any single *stuck-at fault* can be detected by this technique, extra redundancy and hardware modifications are required to create self-dual functions from non-self-dual ones. Any non-self-dual function of *x* variables can be converted into an *x +1* variable function that is self-dual and can thus be implemented with an alternating logic circuit.

3.6.3. Recomputing with shifted operands (*RESO*)

Recomputing with shifted operands is a logic level concurrent error detection technique based on time redundancy developed by Patel and Fung (Patel and Fung, 1982). *RESO* can be applied to certain problems in which shifting the inputs forms a complementing function that produces a known relationship in the outputs. It has been originally used for arithmetic and logic units. The error detection capability of *RESO* depends on the number of shift operations. The generalized version is *RESO-k* and it refers to shifting by *k* bits.

[20] A *stuck-at fault* is a particular fault model used to represent a manufacturing defect within an integrated circuit. Depending on the effect of the fault, a *suck-at fault* can be stuck either at a logical value of 0 (*stuck-at 0*) or 1 (*stuck-at 1*)

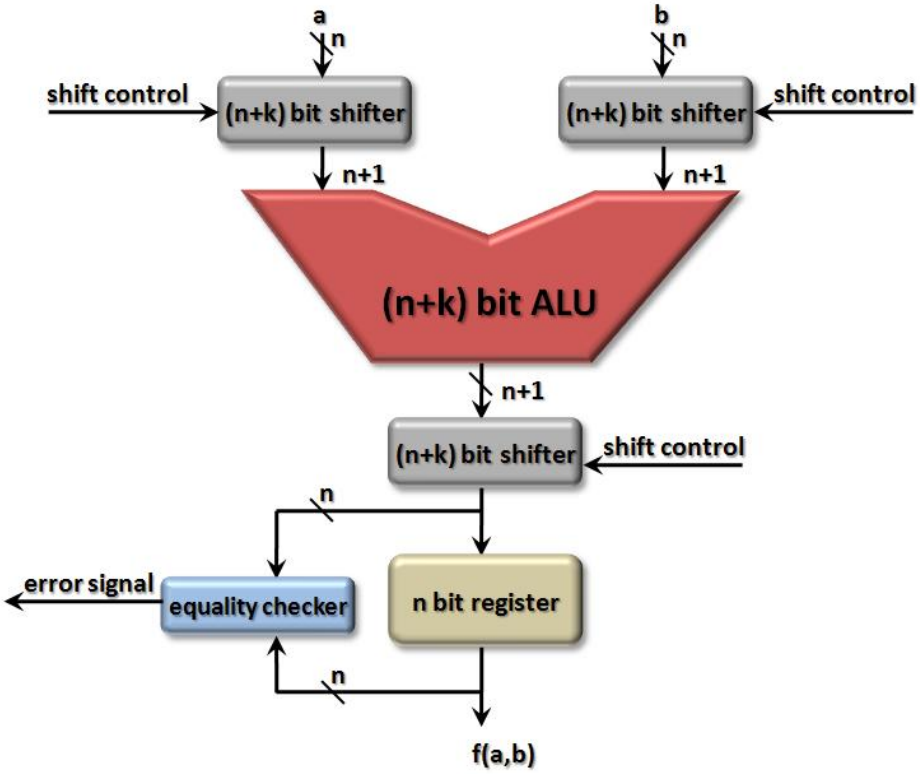


Figure 3-25. *ALU* concurrent error detection using recomputing with shifted operands (*RESO-k*)

Figure 3-25 shows a schematic of a concurrent error detection mechanism on an *ALU* using *RESO*. The operands a and b undergo a normal *ALU* operation $f(a,b)$ during the first computation at time t_0 and the result is stored in a register. During the second computation at time $t_0+\Delta t$, before entering the *ALU* the operands are shifted left by k bits and the result of the *ALU* operation is right shifted and finally compared to the ones previously stored in the register. In such operations, left and right shifting can also be denoted as $E(x)$ and $D(x)$ (or $E^1(x)$). Therefore, if the equivalent notation for the recomputation is $E^{-1}\left[f\left(E(a,b)\right)\right]$ it should be equal to the first computation $f(a,b)$. If the results are identical the output of the computation will be $f(a,b)$. However, if there is a discrepancy an error signal will be generated.

When an n-bit operand is shifted left by k-bit(s), its leftmost k bit(s) move out and the right most k-bit(s) become zero. This may lead to an incorrect result of $f(a,b)$ since k essential bit(s) are removed whenever shifted left. As with

alternating logic, extra redundancy is needed as an $(n + k)$ shifters need to be implemented. Furthermore, a bigger $(n + k)$ bits length *ALU* is needed and therefore the recomputation takes $(n + k)$ bit operations rather than the original n-bit ones. Furthermore, a totally self-checking equality checker is required for the comparison process and error signalling. Additionally, parity codes can also be used to detect error in the shifter logic.

Note that the fault coverage capability of *RESO* depends on the number of shifts. *RESO-1* can detect all single bit-slice errors in an *ALU* for all bitwise operations, including *AND, OR, NOT, NOR* and *XOR*. As k becomes larger, an increase of space and time complexity is entailed which in turn increase the probability of error. Consider an *ALU* with an n-bit shifter and a *RERO-2* implementation and an operand a equal to *11010*. After the *11010* is being shifted left by two bits, it will have the two *MSBs* shifted out, thus becoming *01000*. As a consequence, the result of the calculation $f(a, b)$ will probably be incorrect.

If the shifter is replaced by an $(n + k)$-*bit* shifter with *k=2* in this particular case (*RESO-2*), then the operand a after the shifting operation will be equal to *1101000*, thus keeping the *MSBs* and ensuring the correct result $f(a, b)$. Note that during the first computation *k-zero MSBs* are added to each of the operands.

This is one way of detecting errors using *RESO*. Alternatively, as before, during the first computation at time t_0, the operands a and b undergo a normal *ALU* operation $f(a, b)$ but the results are now left-shifted before being stored in the register. In the second computation at time $t_0+\Delta t$, the operands are also left-shifted by k bits, but in this new way, the results are directly compared with the ones in the register (there is not right-shifting performed on the operands).

The penalty paid for implementing *RESO* is that every component must be extended to accommodate the shifting. For instance, to implement *RESO-1* on a *32bit ALU* the main system and the shifters are required to be *33* bits whereas the storage registers and the equality checker must be *34* bits.

3.6.4. Recomputing with rotated operands (*RERO*)

Recomputing with rotated operands (Li and Swartzlander, 1992) is another technique designed to overcome the limitations of *RESO*. *RERO-k* has similar time redundancy characteristics to *RESO* but with different structural redundancy demands.

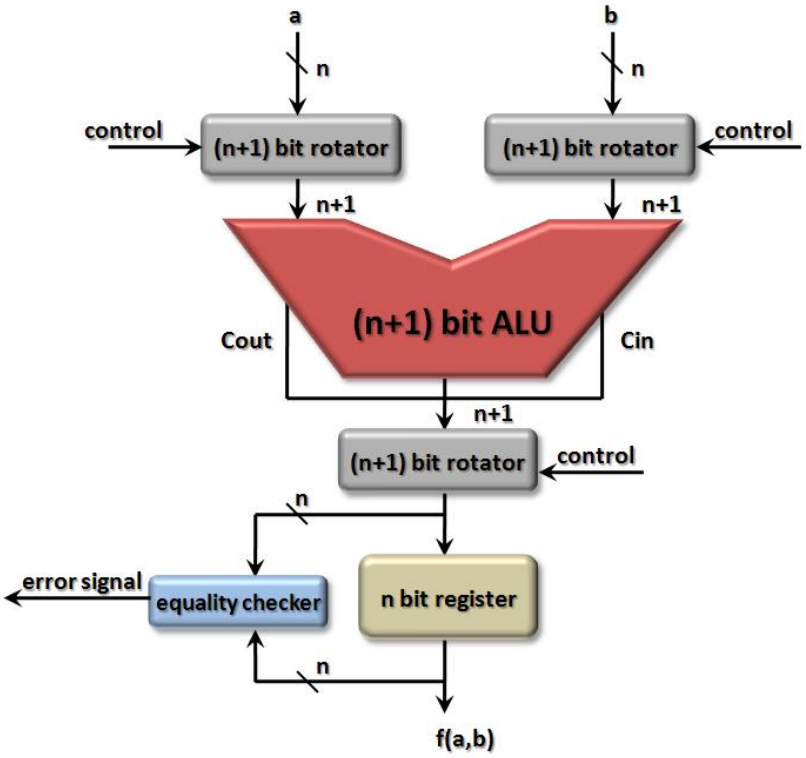


Figure 3-26. *ALU* concurrent error detection using recomputing with rotated operands

Figure 3-26 displays an *ALU* with *RERO-k* based concurrent error detection. RESO-k requires an $(n+k)$-bit rotator and an $(n+k)$-bit *ALU* whilst *RERO-k* only requires an $(n+1)$-bit rotator and an $(n+1)$-bit *ALU*.

During the first computation, the $(n+1)$ bit rotators do not rotate the operands, thus the input and output of the rotators is identical. Both operands undergo a regular ALU operation whose result is stored in a register. During the second computations, the first two rotators perform a k-bit(s) right-rotation of the input operands before they enter the *ALU*. Next, the result is rotated left and compared

to the previous result from the first computation. If the results are identical the output of the computation will be $f(a, b)$. However if there is a discrepancy an error signal will be generated.

With regards to error coverage, a *RERO-k* implementation with an *n*-bit ALU can detect:

- (*k mode n*) consecutive errors for bit-wise logical operations.
- (*k-1*) consecutive errors in a ripple carry adder for arithmetic operations.

3.6.5. Recomputing with swapped operands (*RESWO*)

Recomputing with swapped operands (Hana and Johnson, 1986) is an extension of the *RESO* technique that tries to detect errors by alternating the position of the operands. *RESWO* implementation is very intuitive but with limited applications such as addition, multiplication and Boolean functions but not division or subtraction operations.

The first computation at time t_0 is performed on unmodified operands. During recomputation, at time $t_0+\Delta t$, the operands are first split into two halves, upper and lower, then swapped before calculation and finally swapped back after it. The logic for implementing *RESWO* has been shown to be less complex and less expensive than in *RESO*, in particular when the complexity of individual modules is high (Shedletsky, 1978).

3.6.6. Recomputing with comparison (REDWC)

Recomputing with comparison (Johnson et al., 1988) uses a combination of both hardware and time redundancy. The operands *a* and *b* of an n-bit operation are split into two halves and computed by two virtually divided devices ($n/2$-*bit* size) twice. In a first time slot, the least significant $n/2$-bits (lower halves) of the operands and their duplicates are carried out and then their results compared. Upon completion, in a second time slot, the same operation is repeated for most

significant $n/2$-bits (the upper halves) of the operands. As long as the separate halves do not become faulty in the same way and at the same time, *REDWC* can detect all single faults.

3.7. Redundancy schemes comparison

In general, the addition of correction capabilities to a mechanism involves extra area and/or time overheads. Table 3-6 compares timing, area overheads and capabilities of structural- and time- based *FT* mechanisms.

Table 3-6. Comparison structural-, time- based *FT* mechanisms

Scheme	Space redundancy	Time Redundancy	Detection (D) and correction (C)
Time redundancy based			
Alternating logic	≈0%- 100%	>100%	*D*
RESO	≈0%- 93%%	>100%	*D*
RESO	≈0%- 93%	>200%	*DC*
RERO	≈0%- 93%	>100%	*D*
RESWO	≈0%- 77%	0-100%	*D*
REDWC	≈0%- 90%	0-100%	*D*
Structural redundancy based			
DWC	>100%	≈0%-17%	*D*
TMR	>202%	≈0%-17%	*DC*
TMR with triplicated voter	>208%	≈0%-17%	*DC*
Information redundancy based			
Single Parity	1-6%	≈0%-10%	*D*
SEC-DED	7-32%	10%-129%	*DC*
SNC-DND	13-75%	-	*DC*
DEC-TED	13-69%	22%-200%	*DC*

TR techniques involve low area overheads at the cost of extra timing. Note that some of these techniques, such as *RESO*, can provide correction capabilities by performing more than two computations. In contrast, at the cost of area penalties, *SR* techniques can provide detection and correction with very little timing overheads.

Error detection codes, such as parity coding involve low complexity and low overheads but have limited detection abilities and are not able to detect multi-bit errors. Although information redundancy schemes can be feasible to correct single and double errors in high-capacity memories (Paul et al., 2011), for n>2, n-bit correction circuitry demands considerable area, energy and timing overheads, especially in low capacity memories. For instance, in an *8-bit ECC* scheme integrated to a *64kb* SRAM the area overhead can be more than *80%* (Kim et al., 2007). Application of Hamming *SEC-DED* codes to 16M-bit *DRAM* chips has a 10% access time penalty on a to *16M*-bit *DRAM* (Arimoto et al., 1990; Furutani et al., 1989). For an experimental *1M*-bit *DRAM* cache, applying a *SEC-DED* code imposes up to *15%* access time overhead (Asakura et al., 1990)

The area penalty is even greater in register files (RFs); experimental results for *SEC* applied to a *64*-bit *32*-word RF using *90*nm standard cell *ASIC* technology (Naseer et al., 2006) incurs a *22%* area penalty and a *129%* increase in read access time. *TMR* applied to the same type of registers incurs a *204%* area penalty but increases the read access time by only *17%*. Therefore, for sensitive *ASIC* applications that demand low-latency, *TMR* is more suitable.

3.8. Conclusion

The use of fault avoidance techniques does not guarantee complete removal of faults, having many drawbacks in terms of cost, speed of operation and chip area. System testing and verification techniques can never be exhaustive enough to remove all potential faults and their causes.

Structural redundancy techniques, such as *DWC* for single error detection and *TMR* for single error correction, are very popular. However, both techniques entail high area and power overheads and may not be suitable in embedded applications where power consumption is an important issue.

CED techniques based on *EDC*, such as parity coding, involve lower area overheads than structural redundancy techniques but have limited detection abilities and cannot correct errors or efficiently detect multi-bit errors. *ECC* and physical interleaving incur large area overheads for multi-bit errors. Identifying the time interval for scrubbing can be tricky.

In terms of time redundancy, the aim of a cost effective design should be finding a function *C* that provides a good trade-off between high coverage and low complexity. If the hardware required to implement the coding and decoding functions is similar to that of implementing *f(x)* then structural redundancy techniques are more effective. Time redundancy cannot be used in every application due to the additional time required. For instance, certain long-life critical systems used in space applications can tolerate additional time much easier than additional space or power requirements whereas real-time safety critical systems used in avionics cannot afford any additional performance penalty. Apart from time, extra hardware in the form of shifters, registers, comparators and extra bits are needed in *ALUs*. Moreover, fault coverage is not provided for shifters, rotators and comparators unless they are implemented with self-checking capabilities. However, if time is available *TR* techniques do offer an opportunity to minimize the additional hardware required as compared to structural redundancy.

When it comes to implementing *FT*, the selection of particular types of redundancy greatly depends upon the application. Therefore to select a specific set of redundancy techniques for implementation we should examine a) the different requirements of the particular application and b) the techniques that are more suitable for such requirements. Likewise, not only the type of redundancy technique is important, but where and at which level it's applied; for

instance, applying *TMR* at the gate, register or circuit level would have a different fault coverage, time, structural and power consumption trade-off.

Chapter 4

Chapter 4

Impact of Radiation on electronics of embedded systems

Exposure to radiation of electronic devices can lead to catastrophic system failures in embedded systems, significantly affecting their reliability. Therefore, prior to the design of a resilient architecture, several factors shall be considered in the earliest phases of the system design. The physics of radiation-induced faults, the study of the error process and the sources of error are discussed. The phenomenon that causes faults at the physical level is reviewed. This chapter presents a review of unwanted effects in semiconductor devices caused by high-energy particles focusing on standard electronic materials: silicon and its oxide. We learn that the number of faults, and in particular the ones due to radiation are expected to increase significantly.

4.1. Introduction

To develop efficient fault tolerant systems, designers need to be aware of the impact of permanent and transient faults. Hardware faults are a major concern in silicon based electronic components such as SRAM, DRAM, microprocessors

and FPGA. These devices have a well-documented history of faults mainly caused by high-energy nuclear particles.

In the cases of safety-critical systems, aerospace and health monitoring systems, maximum reliability can be achieved assuming susceptibility of those systems to faults produced by various internal (e.g., interconnect coupling noise) and external reasons (e.g., cosmic and solar radiation). The traditional reliability analyses of these systems assume failure rates of permanent faults. A typical failure rate for permanent faults due to hard reliability mechanisms such as gate oxide breakdown or metal electromigration is generally between 1 and 50 FITS. So far, design and reliability engineers are discounting the effect of transient faults. Moreover, advances in semiconductor technology have been gradually increasing performance. Aggressive scaling of transistor sizes has driven these remarkable improvements in computational performance. However, the density of modern silicon chips makes them vulnerable to particles of lower energy causing transient faults and, as a consequence, catastrophic failures (Constantinescu, 2003; Hazucha and Svensson, 2000; Hazucha et al., 2003). Without mitigation mechanisms the error rates due to these transient faults can easily exceed 50,000 FITS per chip.

4.2. Radiation and its effects on electronics

The term “radiation” is commonly used to describe a process in which energy travels through a medium, or space, ultimately to be absorbed by another body. Radiation can generally be divided into ionising and non-ionising radiation depending on its ability to ionise matter. Non-ionizing radiation does not usually carry enough energy to produce changes to electronic circuitry. Non-ionising radiation can move atoms in a molecule around or cause them to vibrate but does not carry enough energy to ionise atoms or molecules, and as such is not a concern. Non-ionising radiation comes in the form of visible and infrared light, radio- and micro- waves and thermal.

Ionising radiation has enough energy to directly or indirectly remove electrons from atoms or molecules, thus causing the formation of ions. It includes highly energetic protons, alpha particles, heavy ions, galactic cosmic rays and others. Even though neutrons are not ionizing particles their collision with nuclei produces ionizing radiation and therefore are also included in this classification.

As manufacturing technologies evolve, the effects of ionising radiation are becoming a primary concern. Semiconductor devices are sensitive to ionising radiation in the space environment, high altitudes and sea levels. There are different radiation damage mechanisms affecting electronics including atomic "lattice displacements and ionisation damage. Such mechanisms induce different types of failures such Total Ionising Dose (TID), Single Event Effects (SEEs) and Displacement Damage Dose (DDD).

Resulting particles from distinct radiation sources affect diverse electronic technologies in a variety of ways. Due to the reduction in size of the transistors and the reduction in critical charge of logic circuits, the natural resilience of previous technologies to information corruption is decreasing (Baumann, 2002; R. C. Baumann, 2005; Seifert et al., 2002; Shivakumar et al., 2002). Collision of energetic particles with sensitive regions of the semiconductor can alter stored information, potentially leading to logic errors.

Transient faults (Breuer, 1973), the predominant faults in modern technologies, can be caused by environmental conditions like temperature, pressure, humidity, voltage, power supply, vibrations, fluctuations and electromagnetic interferences due to crosstalk between long parallel lines in a die. However, ionising particles are the major source of this type of fault. Transient errors in electronic devices due to ionising radiation in the space environment are well known (Adams and Gelman, 1984; Adams et al., 1982; Binder et al., 1975; Blake and Mandel, 1986; Waskiewicz et al., 1986) as is the impact of such radiation on application-specific electronics such as commercial (Dyer et al., 1990; Johansson et al., 1998; Olsen et al., 1993) and military (Taber and Normand, 1993) avionics, nuclear exposed environments (Mahout et al., 2000; Marshall, 1963), medical instrumentation

(Bradley and Normand, 1998), and other sea level domains (Hauge et al., 1996; Ziegler, 1996).

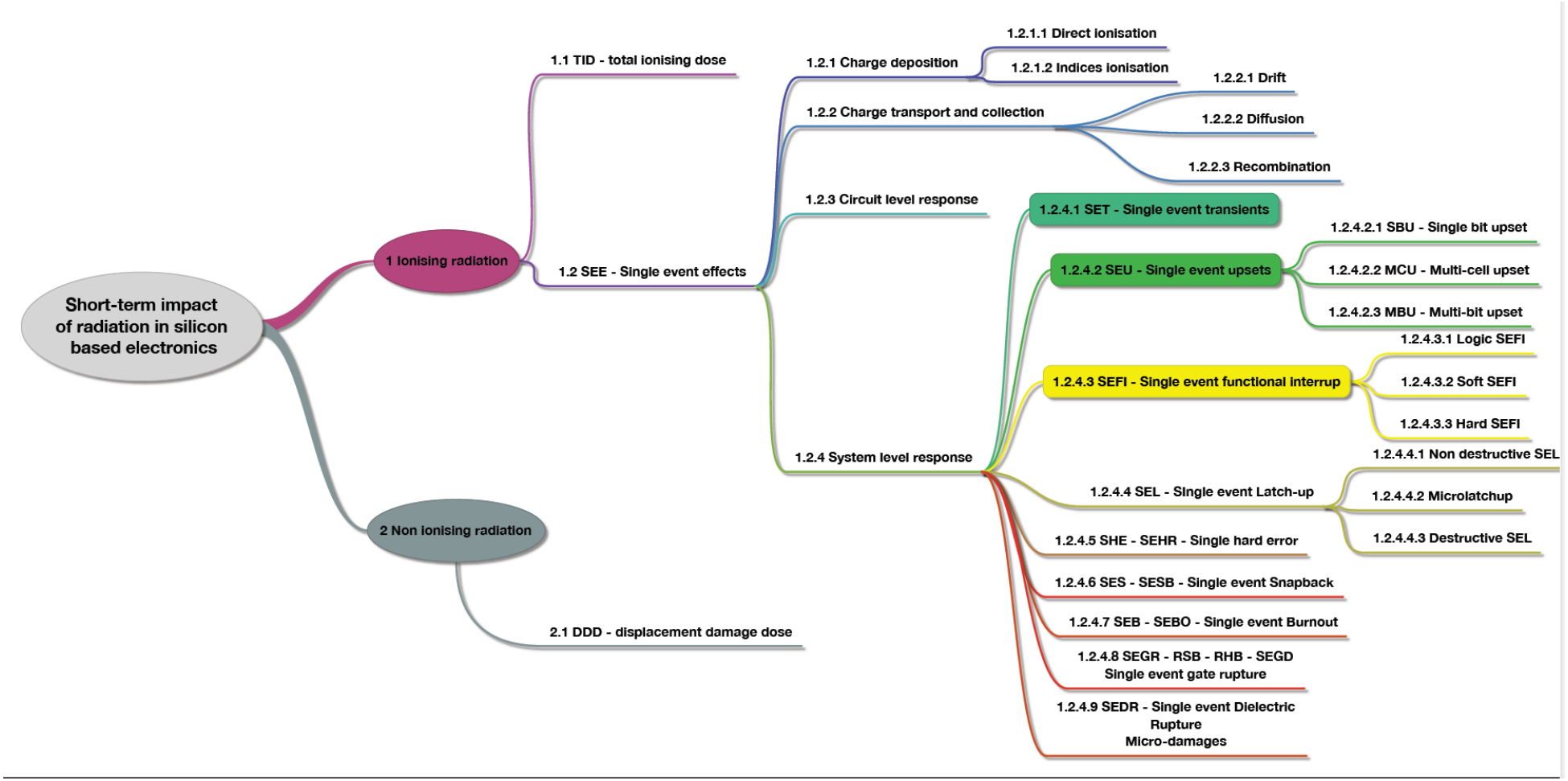


Figure 4-1.Taxonomy of radiation effects in silicon based electronics

4.3. Damage mechanisms

The two fundamental damage mechanisms (Claeys and Simoen, 2002) to electronic elements due to radiation are *atomic lattice displacements* and *ionisation damages*.

Atomic lattice displacement occurs when an energetic particle undergoes a nuclear collision with one or more atoms of the electronic device, changing its original position (see Figure 4-2 below) and thus the analog properties of the semiconductor junctions, potentially worsening in the long term the properties of the material and creating lasting damage. In silicon, an impacted atom can become displaced if it is part of the crystalline structure and the incident particle is capable of inducing a minimum energy (displacement threshold energy) of around 20eV (Miller et al., 1994).

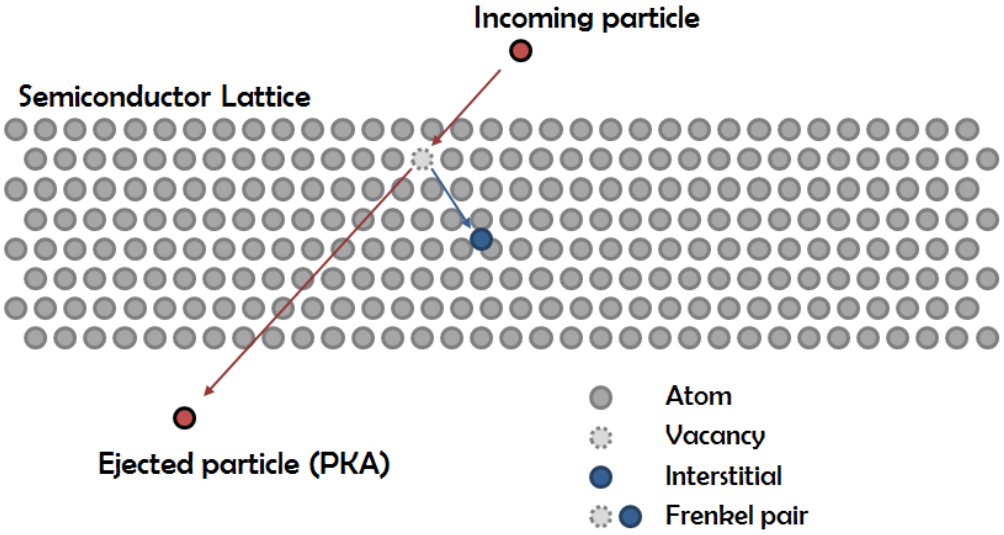


Figure 4-2. Atomic lattice displacement

The displaced atom is referred to as "*primary knock-on atom*" (PKA) and its new non-lattice position is called "*interstitial*" while its absence from its original lattice position is named "*vacancy*". Normally, the simplest configuration is a vacancy and an adjacent interstitial generated as a result of a low energy particle hitting the material, a combination referred to as "*Frenkel pair*". However, in most cases the displaced atom has enough energy to knock out a neighbouring

atom creating a more complex configuration called “cluster”, altering the properties of the bulk semiconductor material. In silicon, vacancies and clusters are of unstable nature and tend to be filled by near atoms leading to more stable defects. In general, this migration leads to the most typical process, called “*defect reordering*” or “*forward annealing*” reducing the amount of damage and its effectiveness. Yet, in some cases, depending on the time, temperature and nature of the device, “reverse annealing” can take place, resulting in more efficient defects.

Ionisation damage is primarily induced by charged particles usually leading to transient effects causing temporary variation of the functionality of the system. Since no permanent damage is induced in the electronic circuit, this type of error is called soft error. Ionisation damage may also lead to small degradation and permanent errors, also called hard errors. A key factor in the damage process is the critical charge, or Q_{crit}, which is the smallest amount of charge that can cause a change of value in a cell. The effects provoked by the above damage mechanisms can vary depending on the type or combination of types of radiation, radiation flux, total dose, critical charge of the device and manufacturing technology. These factors make modelling of faults difficult and time consuming.

4.4. Radiation macro effects

Three major macro effect categories may be used to classify the resultant effects: *Total Ionising Dose* (TID), *Displacement Damage Dose* (DDD) and *Single Event Effects* (SEE). As far as the type of degradation that these macro effects have, TID and DDD are considered as long term cumulative and SEE as short term. Table 4-1 summarizes the characteristics of these radiation macro effects.

Table 4-1. Characteristics of radiation macroeffects

Radiation Effect	Type of degradation	Source	Damage Mechanism	Microeffects	Counter measures - Mitigation Techniques	Sensitive Technologies	Temperature dependency
Total Ionizing Dose (TID)	Long-term cumulative	Trapped protons, trapped electrons and solar event protons	Ionizing damage	Small energy transfers deposited uniformly and delivered over a long time.	-Partial mitigation: Additional shielding is only effective in particular technologies and environments Robust electronic design. High drive currents. High noise immunity, large gain margins, etc. Cold redundancy using spares. Not suitable for all technologies.	Power MOS, CMOS, NMOS, PMOS, SOI, SOS, Bipolar, BiCMOS	YES
Displacement Damage Dose (DDD) - Bulk damage	Long-term cumulative	Trapped and solar protons and neutrons	Atomic Lattice Displacement damage	Accumulation of small energy transfers to atomic nuclei (Coulomb, nuclear interactions).	-Shielding is not only ineffective, but it is also the root of the problem	Bipolar, BiCMOS	NO
Single Event Effects (SEE)	Short-term	GCRs, particles from solar events, trapped protons, and secondary neutrons	Ionizing damage	Sudden large energy transfers at the 'wrong place and time'.	-Additional shielding is NOT effective. - Ensure systems are not sensitive to transient effects. - Fault tolerant design techniques. - Error Detection and Correction for critical circuits. - System Autonomous re-boot.	Power MOS, CMOS, NMOS, PMOS, Bipolar, SOI. SOS, BiCMOS	YES

Total Ionising Dose is a measure of the cumulative effects of the prolonged exposure to ionising radiation. In the context of silicon devices, it is also called surface damage. MOS and bipolar electronic technologies are affected by TID and once the material is damaged, it will not return to its original state (Felix et al., 2007). In today's devices, the formerly used bipolar transistors have been almost completely replaced by the MOSFETs (Metal Oxide Silicon Field Effect Transistors).

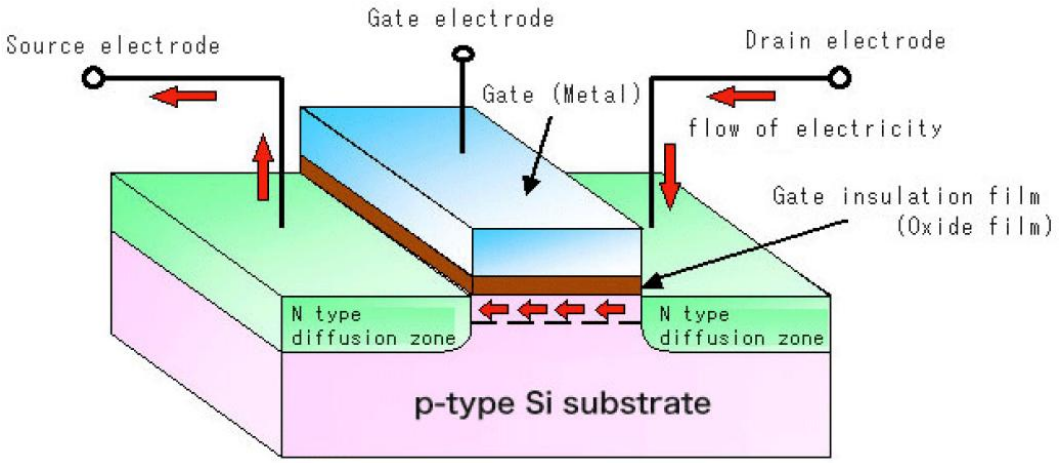


Figure 4-3. Schematic of a MOS transistor

The schematic of a typical MOS transistor is shown in Figure 4-3. Its basic architecture is based on an N-(P-) doped silicon substrate and two highly P-(N-) doped contacts, the source and the drain. The channel between the source and the drain is covered by the gate oxide. This thin silicon dioxide (SiO_2) insulating layer is situated under the gate electrode and can attract charge carriers into the channel region. If no voltage is applied at the gate electrode, no current can flow between drain and source. By regularly applying low voltages at the gate, the current between drain and source is regularly switched on and off.

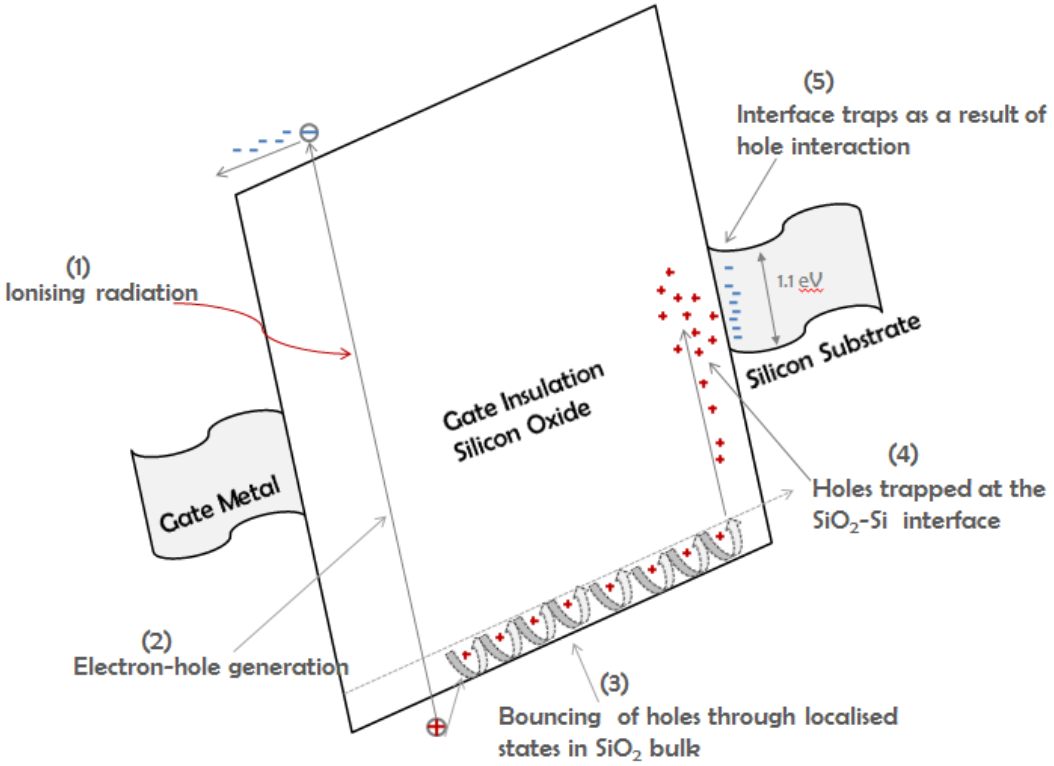


Figure 4-4. Schematic of the motion of electron holes in a silicon oxide

When a highly energetic particle strikes the semiconductor material, as shown in Figure 4-4, electron hole pairs are generated but disappear quickly due to the low resistance of the gate and the substrate. However, in the oxide, and due to their different mobility, electrons rapidly move either to the gate or to the channel whereas the holes slowly bounce from site to site until they become trapped[21] by defects near the silicon oxide interface. Some of these holes may be trapped for a long time resulting in a positive charge in the oxide that can affect the characteristics of the transistor and generate shifts in its operating threshold. These voltage shifts are the most common form of radiation damage in MOS technology and can persist from hours to years.

[21] In MOS structures *oxide traps* are defects in the SiO_2 layer, *interface traps* are defects at the Si/SiO_2 interface and *border traps* are defects near the interface (Fleetwood et al., 2008)

TID effects can lead to degradation within the electrical circuit (threshold shifts), decreased functionality, switching speed, device current, increased device leakage (higher power consumption) and even functional failures. The primary sources of TID are trapped protons and electrons, and solar protons (Barth et al., 2004).

Modern submicron electronics offer relative relief to these effects in the way of natural radiation hardening (Pouponnot, 2005; Velazco et al., 2007). Current gate oxides are around 100 times thinner than the approximately 100nm oxide layers employed in the early 1990s. Modern gate oxides are around 1nm thick, which allow electrons to tunnel through the potential barrier at the silicon oxide interface, neutralizing the trapped holes. Since there is not enough trapped charge, transistor threshold shifts cannot be generated.

Circuit level radiation hardening techniques, i.e. changes in the geometry of transistors, have been used to mitigate TID effects but such techniques are expensive. Furthermore, TID effects might be partially reduced with the use of shielding material that absorbs most electrons and low energy protons. However, the amount of shielding is inversely proportional to its effectiveness in stopping the protons with higher energy (Dyer et al., 1996). TID is considered a severe problem (Claeys and Simoen, 2002) during the lifetime of satellites.

Displacement Damage Dose (DDD) or “Bulk” damage (Barth et al., 2004; Yu Qingkui et al., 2005), occurs when high energy particles dislodge or displace atoms from the semiconductor lattice due to its long time exposure to non ionising energy loss (NIEL). DDD results in a similar long-term cumulative degradation to that caused by TID. The damage mechanism is the result of collisions with atoms, which become displaced from the lattice creating interstitials and vacancies. Consequently, DDD is an effect of concern for all semiconductor bulk based devices such as bipolar devices (BJT circuits and diodes), BiCMOS, electro optic sensors (CCDs, photo diodes, phototransistors), silicon detectors and solar cells, whereas CMOS is almost insensitive to it.

DDD accumulation primarily occurs when the semiconductor material is exposed to neutrons, trapped protons and solar protons over time. Likewise, secondary radiation produced in shielding materials can cause DDD effects. The overall effect of DDD in semiconductors is alteration in the minority carrier lifetimes, which results in lower currents between the collector and the emitter and therefore reduced transistor gain. An extended review of literature related to this type of damage can be found on (Srour et al., 2003).

4.5. Single event effects (SEE)

The term Single Event (SE) is used to lay emphasis on the fact that the effect is caused by an individual particle interacting with the material. In current semiconductor technologies single event effects represent a much larger problem than the combination of all long-term cumulative effects.

SEEs are induced by the strike of a single energetic particle (ion, proton, electron, neutron, etc.) in sensitive regions of the material. The particle travels through the semiconductor material leaving an ionised track behind depositing sufficient energy to cause an effect on a localized area of the electronic device. Both TID and SEE take place as a result of ionising radiation; however, whilst the former is a long term effect that changes the electrical properties of the device, SEEs are the result of an instantaneous perturbation.

Neutron and alpha (α) particles are the most common sources of SEEs in terrestrial environments whilst cosmic rays and heavy ions are most responsible for space applications. SEEs affect many different types of electronic devices and technologies resulting in data corruption, high current conditions and transient disturbances. If not handled well, unwanted functional interruptions and catastrophic failures could take place.

4.5.1. Physical mechanisms responsible for SEEs

In the "technological shrink model" of sensitivity to upsets (Baumann, 2002; Seifert et al., 2002; Shivakumar et al., 2002) the detailed physical mechanisms responsible for SEE are identified in four consecutive steps (Dodd and Massengill, 2003; Wirth et al., 2008) taking place before an SEE occurrence:

- Prior charge deposition by the incident particle striking the semiconductor,
- Transport of the released charge into the device,
- Charge collection by the different sensitive regions,
- Circuit response.

4.5.1.1. Charge deposition

Ionising radiation can release charge in the semiconductor in different ways. SEEs can occur through the impact of the incident particles themselves (e.g., direct ionisation from galactic cosmic rays (GCRs) or solar particles). SEEs can also occur as a result of secondary particles generated via inelastic or elastic nuclear reactions (Howe et al., 2005; Reed et al., 2006; Warren et al., 2005) and Coulombic (Rutherford or inelastic Coulomb) scattering (Wrobel et al., 2006) between the incident particles and the stationary targets in the struck material (indirect ionisation).

An incident particle can experience a number of interactions before its kinetic energy is expended. In every interaction the path of the particle can be altered and can lose some of the kinetic energy. To measure the energy transferred to the material the terms Linear Stopping Power and Linear Energy Transfer (LET) can be used. Equation 4.1 describes the rate at which a particle loses energy while moving through an absorber. The incremental energy (dE) may be expressed in units of MeV while the path length (dx) may be expressed in units of cm.

$$S(E) = -\frac{dE}{dx}$$

Equation 4.1. Linear stopping power

From these interactions, two types of stopping power can be distinguished (ECSS, 2007; Podgorsak, 2009):

- Nuclear stopping power (also called radiation stopping power) resulting from energy loss per unit path length due to inelastic Coulomb interactions between the charge particle and the nuclei of the absorber. Only light particles, such as electrons and positrons, experience significant energy loss via nuclear stopping power. For heavier charged particles, such as protons and α particles, this type of loss is insignificant.
- Electronic stopping power (also called ionisation or collision stopping power) resulting from inelastic Coulomb interactions between the charge particle and orbital electrons of the absorber. Electronic stopping power describes the energy lost due to direct ionisation. Unlike nuclear stopping power, heavy and light particles experience this type of interaction that results energy transfer from the incident particle to the orbital electrons via excitation and ionisation (ECSS, 2007).

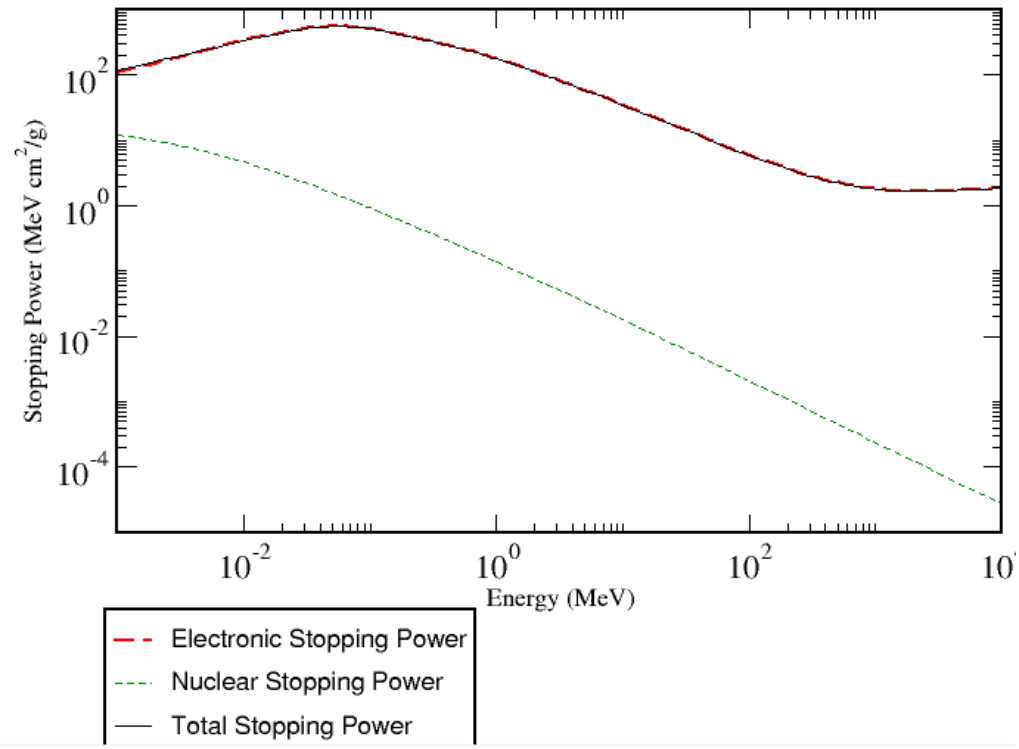


Figure 4-5. Electronic, nuclear and total stopping power of protons in silicon, computed with PSTAR from NIST laboratory (Berger et al., 2005)

The electronic, nuclear and total stopping energy of different particles are presented in Figure 4-5 (for protons) and Figure 4-6 (for electrons). Figure 4-5 shows that at all energies the electronic stopping power of protons dominates and that the nuclear stopping power is insignificant. Figure 4-6 shows that the nuclear stopping power of electrons dominates at higher energies.

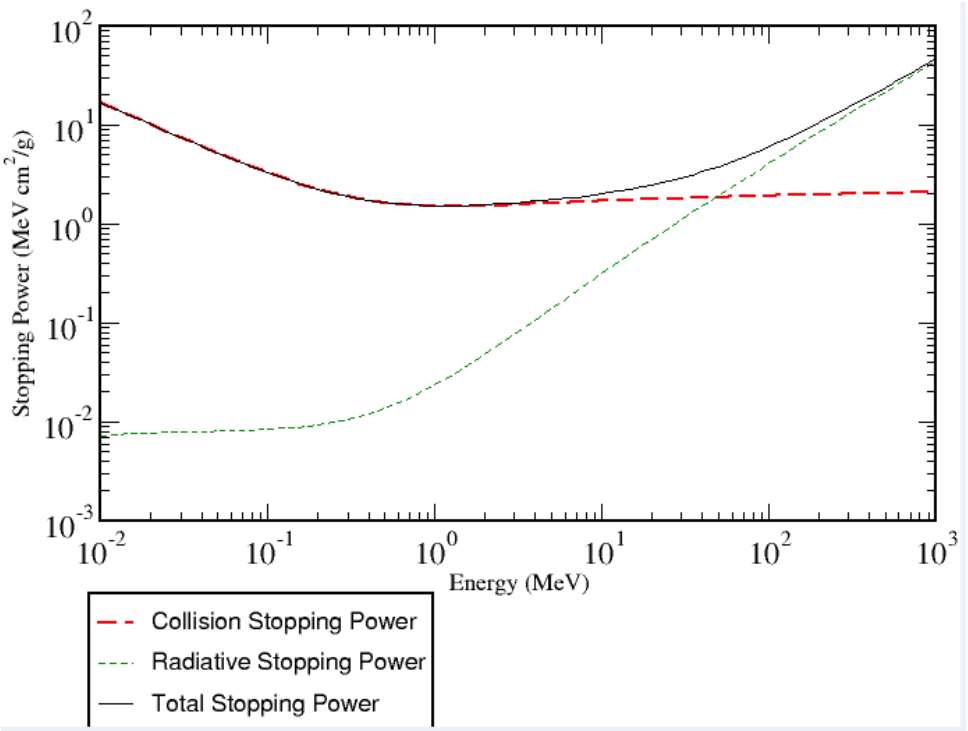


Figure 4-6. Electronic, nuclear and total stopping power of electrons in silicon computed with ESTAR from NIST laboratory (Berger et al., 2005)

The total stopping power ($S(E)_{tot}$) for a charged particle with E_k energy passing through an absorber of atomic number Z is in general the sum of nuclear stopping power and electronic stopping power as shown in Equation 4.2 (Podgorsak, 2009):

$$S(E)_{tot} = S(E)_{nuclear} + S(E)_{electronic}$$

Equation 4.2. Total stopping power for a charged particle

Charge deposition is often characterized by mass stopping power, instead of Linear stopping power. Mass stopping power is defined as the Linear Energy Transfer (LET) (not equal to Linear Stopping Power, but approximated) and can be obtained by dividing S(E) (expressed in MeV/cm) by the density of the material p (expressed in mg/cm3). Nearly independent of the density of the material, LET (Equation 4.3) describes the linear rate of energy transfer to the material as the energetic particle traverses the absorber.

$$LET = \frac{1}{p}\frac{dE}{dx}$$

Equation 4.3. Linear energy transfer

The LET of an incident ion and thus the density of ionisation, typically increase to a maximum immediately before the particle comes to rest. This peak, the Bragg peak, occurs due to the increasing cross section as the particle loses energy.

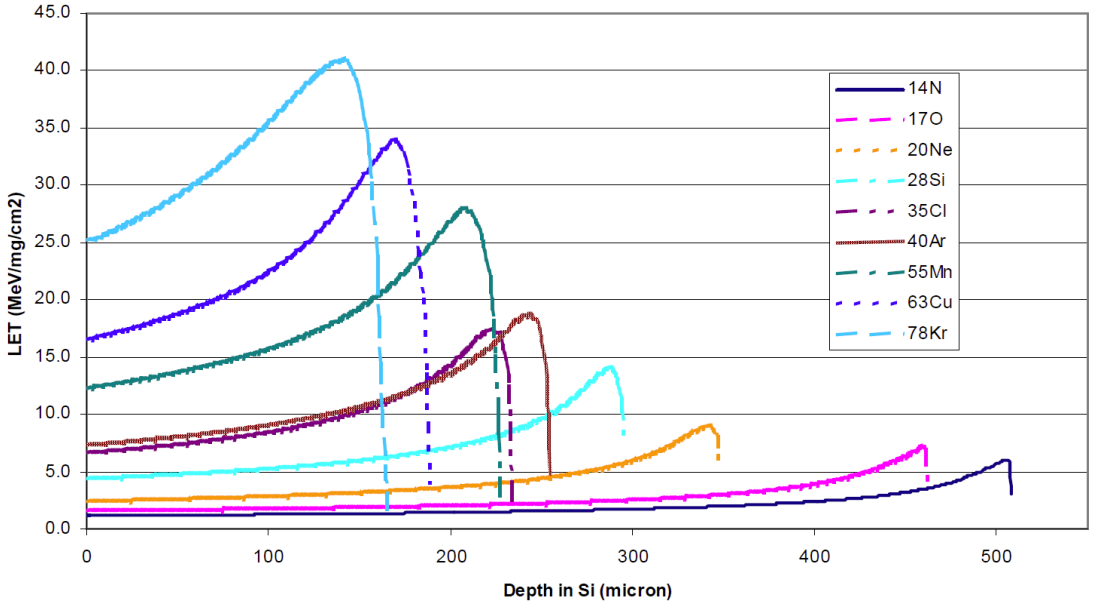


Figure 4-7. Bragg peaks: LET (MeV/cm2) of the standard components of a 16MeV/nucleon cocktail versus depth in silicon (µm) (McMahan et al., 2004)

Incident particles can cause different nuclear reactions depending not only on the striking energy but also on the target mat erial. Figure 4-7 shows a plot of the LET of the standard components of a 16MeV/nucleon cocktail as a function of depth in silicon. The LET of a given ion is dependent on its energy and the target material, and therefore is an important parameter to quantify the sensitivity of electronic devices. Theoretical and experimental values of LET for most ions in different materials have been published (Northcliffe and Schilling, 1970). In addition, stopping power for different particles can be calculated using the TRIM code (Ziegler et al., 2010), and the ESRAR, ASTAR and PSTAR programs (Berger et al., 2005).

The LET can be converted into charge per unit length (fC/µm or pC/µm). This is more suitable to situations that take into account the physical dimensions of the device and the charge stored at the critical nodes. For example, in silicon based technologies a particle with a LET of 97 MeV-cm2/mg corresponds to a charge deposition of approx. 1 pC/µm.

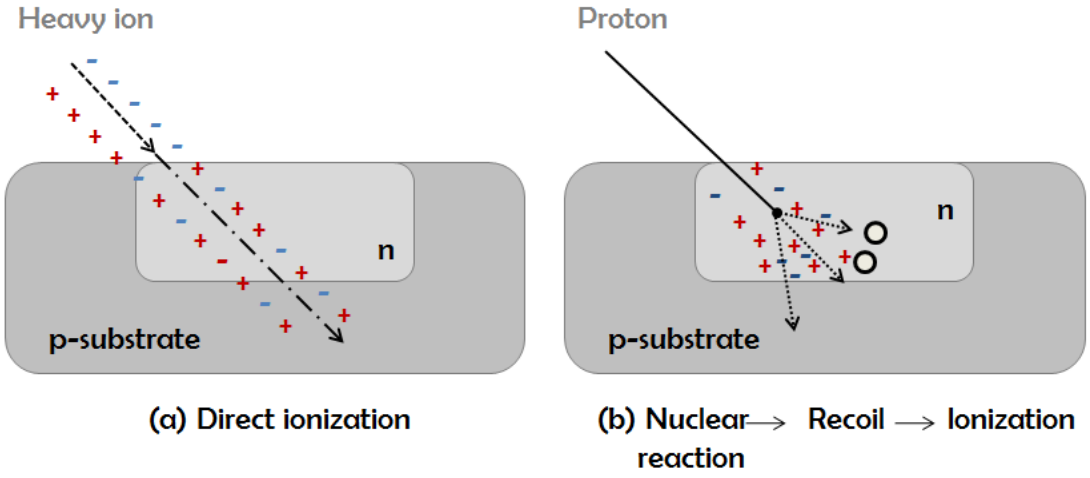


Figure 4-8. Energetic particle strike and generation of electron hole pairs: a) direct ionisation due to heavy strike; b) indirect ionisation due to proton strike

In passing through a semiconductor material, high energetic particles (direct ionisation Figure 4-8a) can deposit energy in the absorber through a one step process involving Coulomb interactions with the electrostatic field electrons in the target atom (Podgorsak, 2009). The energy introduced allows bound electrons to leave their atoms, releasing free electron hole pairs and converting their energy into charge (part b of Figure 4-8b). The particle rests in the semiconductor material once almost all its energy is lost. The formation of an electron hole requires an average energy of 3.6eV. The energy lost due to direct ionisation can be referred as the electronic stopping power.

The total path length or total distance travelled is referred as particle's range and is highly dependent on the type of particle, its initial energy and the properties of the semiconductor material.

At sea level, direct ionisation is the main charge deposition mechanism for upsets caused by heavy ions and alpha particles, emitted due to the contaminants in packaging materials. Traditionally, since protons and neutrons are lighter, the charge released by them is not enough to produce upsets via direct ionisation.

As suggested in 1997 the technological shrink model would soon be affected by direct ionisation of low energy particles (Duzellier et al., 1997). Recent experimental evidence (Heidel et al., 2008) of 65nm SOI SRAM sensitivity to direct ionisation from protons supported the latter suggestion with results that the low energy proton for the 65nm technology is different to those from previous generations.

However, the most significant upset rates due to light particles are caused via indirect ionisation mechanisms. In fact, in today's semiconductor technology, high-energy neutrons derived from cosmic rays are the primary contributor to soft error rates at sea level.

In those mechanisms, the highly energetic particles (protons or neutrons) do not directly interact with the material. The three indirect ionisation mechanisms are:

- Inelastic nuclear reactions that take place when the incident particle hits a target nucleus causing fragmentation and ejection of secondary particles;
- Elastic nuclear reactions that take place when the incident particle transfers some of its energy to a target nucleus that recoils (Figure 4-8) with extra energy transferred from the incident particle;
- Coulombic scattering, similar to elastic nuclear reactions, takes place when the incident particle gets close to a target nucleus that recoils due to Coulomb force with less momentum and smaller angle than with elastic nuclear reactions.

Among these three mechanisms, inelastic nuclear reactions have the higher probability of depositing larger amounts of charge, and hence are the most

significant indirect mechanism in the formation of SEE. If an inelastic nuclear reaction takes place, a collision with a target nucleus leads to the emission of reaction products that can, in turn, deposit energy via direct ionisation.

Those resulting particles are much heavier than the incident particle, which involves higher charge deposition that may result in a SEE. Since the incident particles do not directly interact with the semiconductor material, the number of counts or neutrons per cm^2 is used to measure the effect rather than the LET.

4.5.1.2. Charge transport and collection

Subsequent to the charge deposition, the released carriers are transported and collected by the semiconductor elementary structures. The transport of the charge is based on three main mechanisms (Dodd, 2005):

- Charge collection by drift: The charge can drift in regions with an electric field. Reverse biased semiconductor p-n junctions are usually the most sensitive regions. If the ionised track affects one of those junctions, the high electrical field present in the region can collect the incident charge, which can result in significant transient currents. This is a fast mechanism in the order of 100ps.
- Charge collection by diffusion: the charge may diffuse in neutral zones (bulk of the device), leading to considerable transient currents. This is a slow mechanism in the order of nanoseconds.
- Recombination: The charge can recombine with free carriers in the lattice.

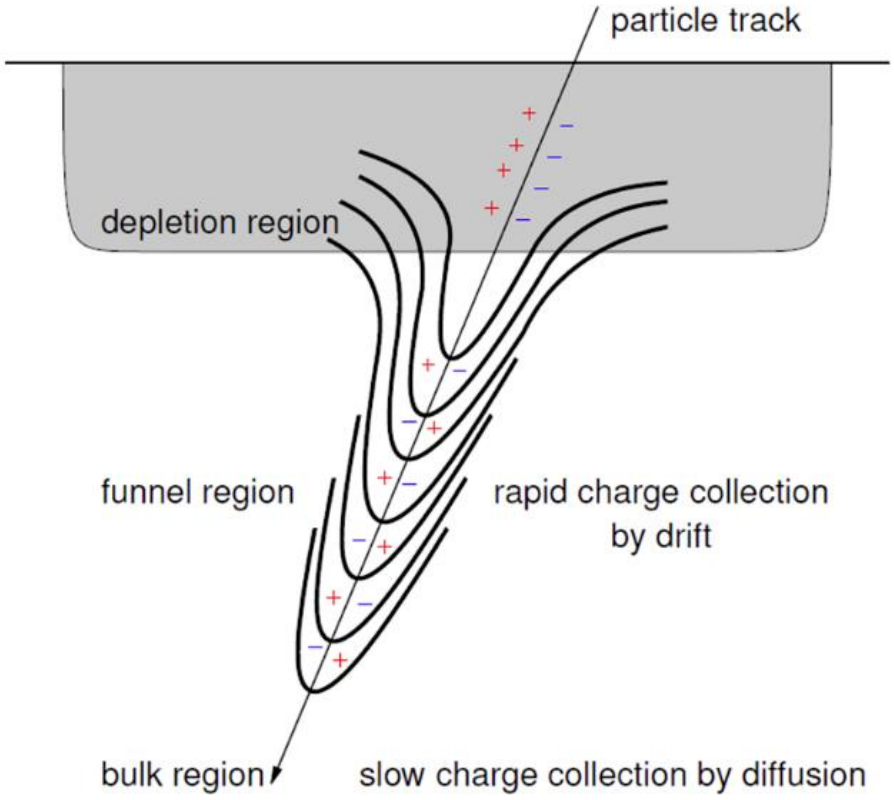


Figure 4-9. Funnelling effect and charge collection mechanisms (Messenger and Ash, 1992)

As Figure 4-9 illustrates the charge collection can be extended via "field funnelling" (Chang-Ming Hsieh et al., 1983; Hsieh et al., 1981). If a high field region, such as the depletion region of a p-n junction, is traversed by a column of electron holes, the associated electric field can be disturbed, spreading down along the particle's track deep into the substrate, consequently reducing the net charge in the depletion region.

Three different areas within the track can be distinguished: 1) the initial depletion region, 2) the funnel region, and 3) the bulk region.

Within the external depletion region, positive potential areas attract the electrons and negative potential areas attract the holes. Rapid collection by drift will take place in the funnel region whilst the diffusion mechanisms will slowly collect the charge of the residual carriers in the bulk region. The "funnelling effect" is effective in the range of a few nanoseconds. The generated carrier density in the vicinity of the junction becomes similar to the substrate doping

concentration, and the electrical field is then re-established back to its original position (Figure 4-10). Therefore, field funnelling and consequently charge collection are highly dependent on the substrate doping concentration.

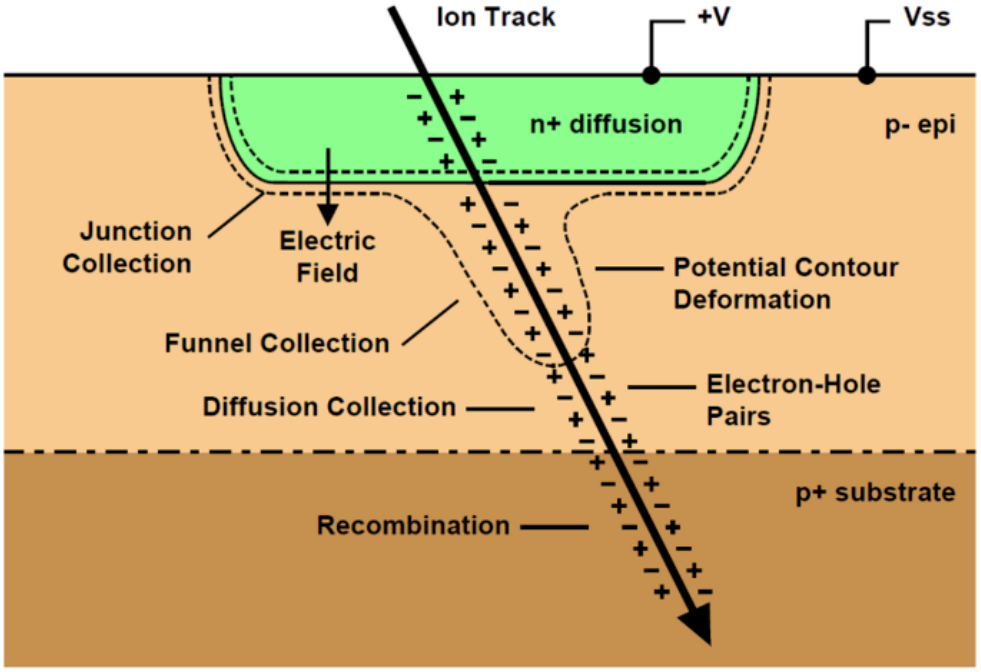


Figure 4-10. Funneling effect and charge collection mechanisms after a particle strike on a p-n junction (Mavis, 2002)

4.5.1.3. Circuit level response

The collected charge transported in the device induces parasitic transient currents, which turn could induce disturbances in the external circuits. Depending on a) the collected charge, b) the intensity of the resultant current transient, c) the details of the circuit application and d) the area affected, the excess of charge can be manifested as one of many types of SEE (or a combination of them).

Semiconductor devices experience SEEs in two major forms: in the form of destructive effects, which result in permanent degradation or even destruction of the device affecting functionality, and in the form of non-destructive effects, causing no permanent damage. Table 4-2 presents different type of errors, their nature, characteristics and a solution to eliminate their effect.

Table 4-2. Type of errors and how to fix them

Type of error		Characteristics	Nature	Fix
Soft	Transient soft	Functionality in place Incorrect logical value	Non-destructive	Reading or writing
	Firm, Static soft	Functionality in place Incorrect logical value Reading does not fix it	Non-destructive	Writing
Pseudo-hard		Functionality lost No permanent damage	Non-destructive	Power-off cycle or Reducing the power supply voltage below the holding voltage
Hard		Functionality lost Physical-permanent damage	Destructive	Replacement of HW

Soft errors are of temporal nature and imply that the physical functionality of the circuit is not affected even though its temporal integrity is. Soft errors have been defined ("JEDEC JESD89-3A," 2007) as *"an erroneous output signal from a latch or memory cell that can be corrected by performing one or more normal functions of the device containing the latch or memory cell"*.

Typical examples of this are undesired changes of logic value in sequential logic and undesired analog pulses that temporarily change the output of combinational logic. Soft errors can be further categorized into transient and static errors (Mavis and Eaton, 2002).

Transient soft errors are *"soft errors that can be corrected by repeated reading without rewriting and without the removal of power"* ("JEDEC JESD89-3A," 2007). On the other hand, static soft errors, or firm errors, are those that cannot be corrected by repeated reading but can be corrected by rewriting without the removal of power, resulting in a completely functional memory (Caywood and Prickett, 1983).

When a soft error has occurred, it could result in a detected recoverable error (DRE), detected unrecoverable error (DUE) or silent data corruption (SDC) (Kadayif et al., 2010; Weaver et al., 2004). If fault tolerant techniques are implemented a soft error could potentially be recovered, either by hardware or software. This is a DRE, a more benign type of error, since recovery of the normal

operation is possible. DUE take place when the same fault tolerant techniques are able to discover and/or report an error, from which recovery is not possible. A SDC take place when an error is undetected and causes data corruption (SDC) (Constantinescu et al., 2008). In this case, the corrupted data could go unnoticed making this type of error benign, or could result in a visible error and/or catastrophic failure such as crashing a computer system.

Hard errors, or Permanent errors, lead to loss of device functionality but, in contrast with transient soft and firm errors, the functionality of the device is permanently damaged. Repeated reading, writing or re-powering is not effective in recovering from this type of errors. In general, hard error effects can only be corrected via maintenance action, involving replacement of components.

A further categorization between hard and soft errors is pseudo hard errors, sometimes referred to as power cycle soft errors (PCSE) ("JEDEC JESD89-3A," 2007). These take place as a result of the ionising radiation from a particle strike, when the functionality of the device is lost but the device is not permanently damaged. Unlike soft errors, pseudo hard errors cannot be corrected by repetitive readings or writings. Instead, they can be corrected by removing the power from the device. Examples of this are non-destructive latchup and firm errors in FPGA where the area affected by the particle strike is the control path (Edwards et al., 2004). Although the data may not be corrupted, the device functionality is compromised. SRAM based FPGA devices are subject to this type of error if the "gate array" configuration in SRAM is corrupted. These systems contain the "gate array" configuration area within ROM, which is loaded into the SRAM during power up. Recovery can be achieved via repowering and reinitialization.

A classification of SEEs is presented in Table 4-3. The numerous types of SEE can be categorized depending on the type of degradation, recoverability and technologies susceptibility. Long- and short- term radiation effects on different manufacturing technologies are presented in Table 4-4.

Table 4-3. Classification of single event effects

Acronym	Name	Type of error	Affected technology
SET	Single event transients	Transient soft	Combinatorial logic, operational amplifiers, analogic and mixed signal circuits
SEU	Single event upset	Static soft	RAM, PLC - Sequential logic
SBU	Single bit upset	Static soft	RAM, PLC - Sequential logic
MCU	Multiple Cell Upset	Static soft	RAM, PLC - Sequential logic
MBU	Multiple Bit Upset	Static soft	RAM, PLC - Sequential logic
SEL	Single event latchup (microlatchups)	Pseudo-hard	CMOS, CPUs, PLC
SEFI	Single event functional interrupts	Pseudo-hard	Complex devices with built-in state or control sections
Logic SEFI	Address error, recoverable bust error, temporary block error	Pseudo-hard	Complex devices with built-in state or control sections
Soft SEFI	Resettable single event functional Interrupt	Static soft	Complex devices with built-in state or control sections
Hard SEFI	Reboot or Permanent single event functional interrupt	Pseudo-hard	Complex devices with built-in state or control sections
SEL	Single event latchup	Hard	CMOS, BiCMOS
Destructive SEL	Address error, recoverable bust error, temporary block error	Hard	CMOS, BiCMOS
Non-destructive SEL	Resettable single event functional Interrupt	Pseudo-hard	CMOS, BiCMOS
Micro-latchup	Reboot or Permanent single event functional interrupt	Pseudo-hard	CMOS, BiCMOS
SEHE or SHE or SEHR	Single event hard error	Hard	Memories and latches in logic devices
SESB or SES	Single event snapback	Pseudo-hard	Power MOS, SOI
SEBO or SEB	Single event burnout	Hard	Power MOS and bipolar
SEGR	Single event gate rupture	Hard	Power MOSFETS, Flash memory
SEDR	Single event dielectric rupture or micro-damages	Hard	Non-volatile nMOS structures, FPGA (antifuse), linear devices

Table 4-4. Long and short term radiation effects on different manufacturing technologies - X1 - Except SOI

Technology	Function	SET	SEU	SEFI	SEHE	SEL	SESB	SEBO	SEGR	SEDR	TID	DDD
CMOS, SOI	SRAM		x		x	x_1	x				x	
	DRAM/ SDRAM		x	x	x	x_1	x				x	
	EEPROM/ Flash EEPROM	x	x	x		x_1			x	x	x	
	Mcontroller /µP	x	x	x	x	x_1					x	
	FPGA	x	x	x		x_1	x		x	x	x	
Power MOS								x	x		x	
Bipolar		x	x					x	x		x	x

4.5.2. System level response

Many different acronyms are used to describe the numerous SEEs in digital integrated circuits. Also called “reversible errors”, non-destructive effects can be classified as SET, SEU, MBU, MCU, and SEFI.

4.5.2.1. Single event upsets (SEUs): conventional upset mechanisms

SEUs are a particular type of SEE that take place when a single energetic particle strike causes a charge disturbance, large enough to directly modify the logic state of a sequential element, such as a register, latch, flip-flop or a memory cell. It is by far the most common effect affecting all kinds of memory devices, including SRAM, DRAM, FLASH memories, microprocessor registers, DSPs, FPGAs, logic

programmable state machines and other similar. SEUs can be categorized as static soft errors since the device functionality is not permanently affected (soft), and cannot be corrected by repetitive reading (static) but only through the rewriting of new data (R. C. Baumann, 2005; "JEDEC JESD89-3A," 2007).

Between 1954 and 1957, there were reports of anomalies in electronic equipment during above ground nuclear bomb tests. Since these anomalies were random, and not related to any permanent hardware fault, these were attributed to electronic noise from the bomb's electromagnetic shock wave. Even though the actual term "Single event upset" was first adopted in 1979 (Guenzer et al., 1979), SEUs were, in fact, predicted in 1962 (Wallmark and Marcus, 1962) when it was forecasted that terrestrial cosmic rays would lead to the eventual occurrence of upsets in microelectronics. Moreover, it was anticipated that this kind of upset would limit the volume of semiconductor devices to a minimum of about 10 μm per side.

Evidence of a small rate of cosmic ray induced upsets in bipolar J-K flip-flops in the space environment (Binder et al., 1975) was presented in 1975 confirming the earlier predictions. Four anomalies were found in the analysis of 17 years of satellite operation. It was suggested that 100MeV heavy ions in the solar wind striking the electronics might be responsible. During the early years of computing there have been many reported cases of electronic anomalies, whose source was unknown at the time. As an example, in 1976, the Cray1 supercomputer at Los Alamos presented an average of 25 memory parity soft errors per month. It was not until 2010 that a study was published, attributing the cause of these anomalies to high-energy neutrons from the cosmic ray background (Normand et al., 2010).

As integration density of DRAM increased to 64k, a significant SEU rate, mainly caused by alpha particle contaminants in package materials was found in terrestrial environments. The first evidence of SEUs at sea level in computer electronics was reported by May and Woods from Intel Corporation in 1978.

Eventually, May and Woods attributed the anomalies to alpha particle from impurities in the packaging modules (May and Woods, 1979).

SEUs at sea level and aircraft altitudes due to cosmic radiation were first predicted (Ziegler and Lanford, 1979) in 1979 by Ziegler and Lanford from IBM Corporation. In 1984 SEU appearances due to cosmic radiation were reported for the first time (Ziegler and Puchner, 2004). The use of low alpha activity materials (May, 1979) mitigated the soft error rate due to this radiation from impurities, leaving cosmic ray as the primary factor of "single event rate" (SER) (Pickel and Blandford, 1978), which is the amount of single events pet unit of time.

However, the increased use of large-scale integration (LSI) technology decreased the volume of the sensitive elements, which implied a corresponding reduction of the critical charge and the number of ion pairs needed to induce a soft error. The resultant SER raise was attributed to a new source, protons from solar events and trapped protons in the Van Allen belts (Wyatt et al., 1979).

The 1980s where characterized by extensive research and development of SEU hardened electronics (Desko et al., 1990; Rockett, 1988; Weaver et al., 1987) and research on the fundamental SEU mechanisms, mostly on memory circuitry (Adams and Gelman, 1984; Blake and Mandel, 1986), since SEUs in combinational logic were rare (May et al., 1984). In 1984 SEUs induced by atmospheric neutrons were predicted in avionics for the first time (Silberberg et al., 1984).

During the 1990s, the prediction of atmospheric neutron induced SEU in avionics was rigorously demonstrated to occur during flight (Taber and Normand, 1992). Furthermore, the concern for SEU increased due to manufacturers reducing the number of SEU hardened components which led to an increased interest for commercially available off-the-shelf (COTS) components, even in space environments (Shirvani and McCluskey, 1998; Underwood, 1998).

Due to its high operating voltages, early SRAM cells were very robust, but with technology scaling, in the last decades, SEUs have become more of a concern, posing a major challenge for the design of memories. SEU susceptibility increases exponentially as voltage decreases and, in contrast, decreases quadratically as feature size decreases. Measurements of neutron accelerated induced upsets in 0.25µm, 0.18µm, 0.13µm and 90nm SRAM showed a SER/bit increase of 8% per generation. The SER of a 90nm SRAM increased of a by 18% for a 10% reduction in voltage (Hazucha et al., 2003). In contrast, more recent results in technology nodes ranging from 250nm through 28nm have shown that the SEU rate per bit has been declining up to the 65nm node (Dixit and Wood, 2011). However, this long term trend has been reversed with results for 40nm SRAM presenting 30% higher bit SER than the previous 65nm technology (Dixit and Wood, 2011). Note that the results provided are based on bit SER. Nonetheless, for every generation the complexity and the number of bits per unit area are increasing and so is the System SER. Recent predictions using Monte-Carlo simulator CORIMS on neutron induced soft errors in SRAMS show that system SER will increase x7 from 130nm to 22nm technology (Ibe et al., 2010).

Embedded DRAM has been widely used in System on Chip (SOC) systems thanks to its density and high performance. At the same technology node, the size of an embedded DRAM bit cell is a quarter of the size of an embedded SRAM cell. With scaling, the voltage reduction has also reduced Qcrit. However, by replacing 2D capacitors (very efficient at collecting radiation charge due to its high area junctions) for 3D capacitors, the collection efficiency has decreased considerably, hence increasing Qcrit. The Qcrit increase due to junction volume scaling is more significant than the Qcrit decrease due to voltage scaling. Because of these, the DRAM bit SER has decreased to around 4x to 5x per generation (R. Baumann, 2005). Then again, the DRAM system SER has remained roughly constant over many generations. In contrast with SRAM, whose SEU susceptibility has increased over the years, the problematic earlier DRAM based on planar cells has evolved to become one of the most robust devices.

4.5.2.1.1. Cell upsets

A cell upset takes place if the deposited charge is greater or equal than the critical charge of the cell, changing its original logical value. These could be single bit upsets (SBUs), multi cell upsets (MCUs) or multiple bit upsets (MBUs).

Single bit upsets (SBUs) are single upsets in a memory cell caused by a single event, i.e. one event producing a single bit error, and are very common on SRAMs.

A single particle can energize two or more memory cells, as shown by (Reed et al., 1997). Multi cell upsets (MCUs), first reported in SRAMs exposed to the harsh space radiation environment (Blake and Mandel, 1986), are multiple bit upsets for one event regardless of the location of the multiple bits, i.e. an FPGA where one routing bit gets an impact from a high energetic particle affecting several memory positions. Hence, MCUs involve both types of upsets, the ones that can be corrected by EDC/ECC as well as those that cannot. Traditionally, MCUs have represented a small fraction of the total number of observed SEU (0-5%) (Maiz et al., 2003). However, in the case of FPGA, high linear energy transfer (LET) heavy ion induced radiation experiments indicate that as geometries shrink the MCU probability significantly increases, accounting for up to 35% of the upsets induced (Quinn et al., 2005). As for SRAM devices, it has been predicted that: 1) the MCU ratio will increase x7 from 130nm down to 22nm; 2) the MCU maximum size (MxN bits rectangular area including failed bits) will exceed as many as 1Mbits in the extreme case; and 3) for 22nm process the maximum bit multiplicity will exceed as many as 100bit (Ibe et al., 2010)

Multiple bit upsets (MBUs) also referred to as single word multiple bit upset (SMUs) (Koga et al., 1993a, 1993b) are a subset of MCUs. And MBU is a multiple bit upset for one event that affects several bits in the same word. This type of deviation cannot be corrected by EDC/ECC. However, it is possible to partially avoid MBUs by using specific layout design of memory cells.

In contrast to cells, bit line upsets are only upset susceptible during a short period of time, the pre-charge period specific from read cycle states. However, susceptibility is dependent on the core cycle frequency. Therefore, bit line upset rates are becoming more important (Schindlbeck, 2005) since recent technologies make use of shorter core cycles, which in turn involve higher susceptibility to upset.

Figure 4-11 shows the sensitive areas that are susceptible to cell and bit line upset. NMOS drains of transistors connected to capacitors are sensitive zones to cell upset. In contrast, the sensitive zones to bit line upsets are the NMOS drains of transistors connected to bit lines (Bougerol et al., 2008).

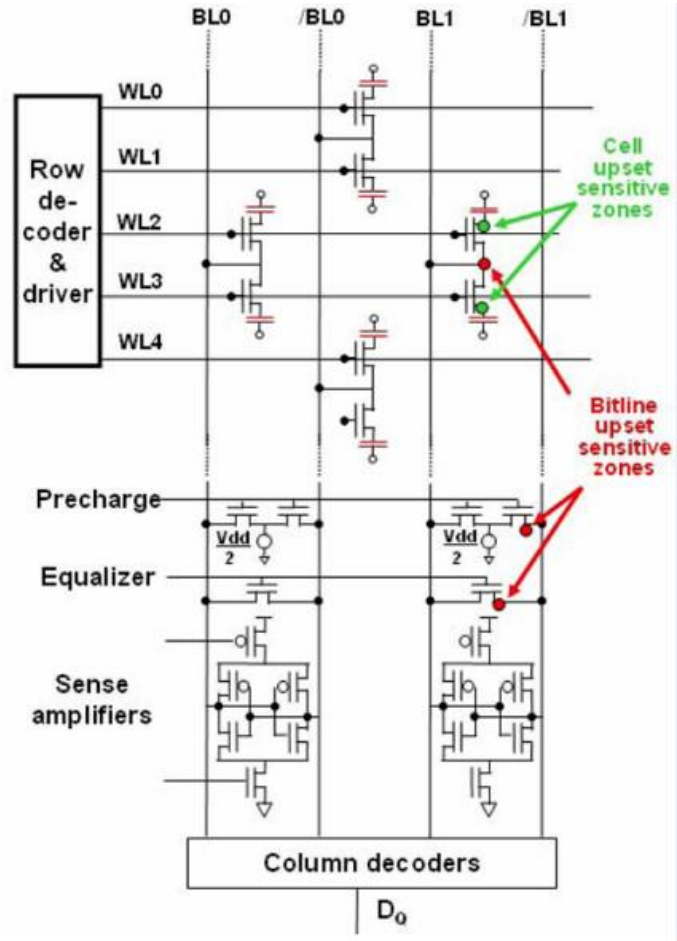


Figure 4-11. Sensitive areas to SEU in a DRAM memory array (Bougerol et al., 2008)

Historically, the occurrence of MCU was attributed to the collection of charges generated by a nuclear spallation reaction as a result of the impact between a secondary ion and the device. As sensitive devices shrink, neighbouring cells present closer physical proximity, increasing the number of cells that can be affected by the impact of a single particle. Nonetheless, novel MCUs are being reported such as "charge sharing among neighbour nodes" (Amusan et al., 2006; Eishi Ibe et al., 2006).

4.5.2.2. Single event transient (SET): an emerging upset mechanism

Without the peripheral logic that interconnects them, sequential logic including embedded SRAM and DRAM would be useless. In general, the scientific community is mostly concerned with the effects of SEUs on sequential logic even though combinational logic is not immune to radiation as single event transients do occur here as well (Baumann, 2002; Buchner et al., 1997; Xiaowei Zhu et al., 2005). However, confusion seems to exist in the literature regarding the terminology used for single event transients. In analog circuits, a SET has also been referred to as "analog single event upset" (Ecoffet et al., 1994). In digital circuits, a transient that causes an incorrect state in the data output of a logic gate has been referred to as "digital single event upset" (Reed et al., 1996).

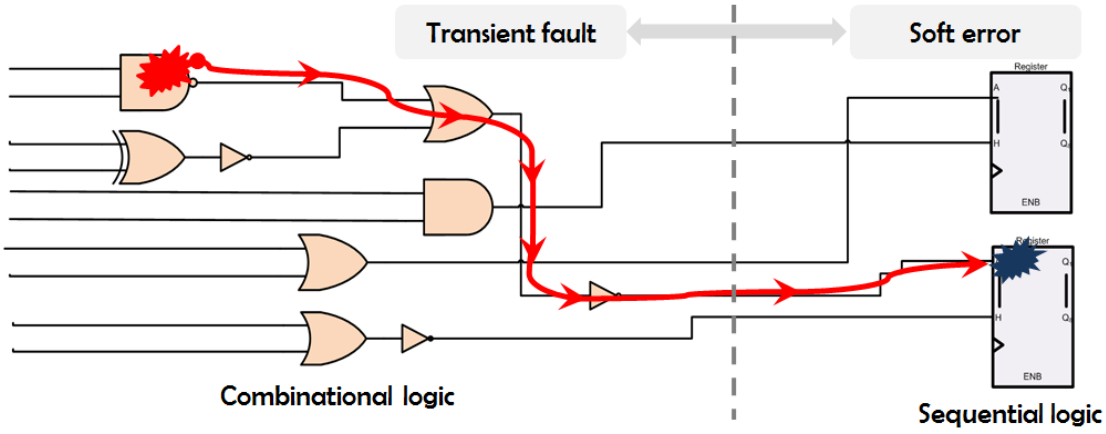


Figure 4-12. Traditional propagation of an SET in combinational logic

Earlier publications often incorporate both phenomena, SET and SEU, together as SEU, perhaps because the effects of an SET can potentially be propagated down the logic line and change the state of a sequential logic element. In this case, the effects are identical to the effects produced by an SEU as shown in Figure 4-12. It is also possible that more than on logic element change their state. This is known as a *single event multiple upset* or SEMU and should not be confused with MBU/MCU.

In contrast with SEUs, SETs were at the time not considered a serious threat to the reliability of semiconductors.

For the purposes of this article, the following definition will apply to the term SET: Single Event Transients (SETs) are analog transient pulses resulting from a single ionising particle, that are large or big enough to momentarily change the output of non latched elements, such as combinational logic, clock line and global control lines to an incorrect logic value. The duration of such pulse is in the order of 100_{ps} (Pouponnot, 2005).

As seen previously in section 5.1.2, different semiconductor technologies show different charge collection and transport mechanisms that lead to different pulses. Depending on the device technology, circuit topology, impact location, particle energy device supply voltage and output load, the resultant SET would have unique characteristics in terms of amplitude, waveform, polarity, duration, etc. Pulses can vary from tenths of picoseconds to tenths of microseconds. The effects of a SET can further be propagated along the logical path, and potentially be latched into one or more flip-flop, latch or register at a distant location from the original charge collection area. Yet, there has not been too much interest in protecting combinational logic since this type of logic has a natural tendency to mask these transient faults. There are inherent masking mechanisms that mitigate the propagation of the glitches, preventing the latch from taking place. These three mechanisms, that can provide a certain level of natural resistance to soft errors, are logical masking, electrical masking, and latch-window masking (Shivakumar et al., 2002; Wirth et al., 2008).

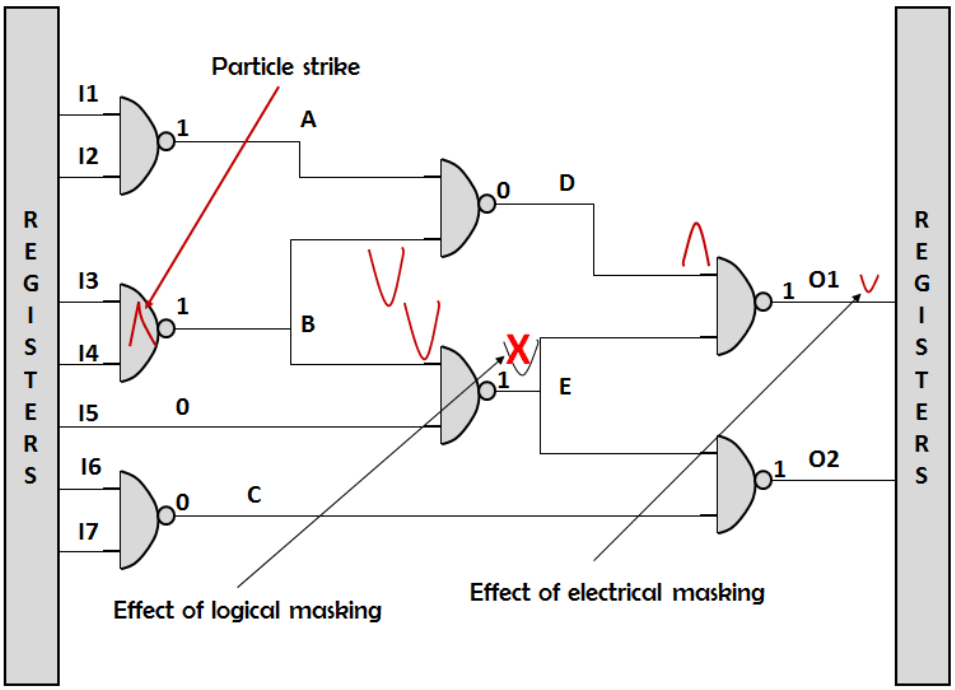


Figure 4-13. Effects of logical and electrical masking on a pipeline stage (Ramanarayanan et al., 2009)

Logical masking takes place when the particle strikes a portion of the combinational logic that, regardless of its output, has no effect on the output of the subsequent gate Figure 4-13. The result of the subsequent gate is solely determined by its other input values. For instance, the output of a NAND gate with an input A equals to '1' and an input B equals to '0' would not be affected by a glitch on the A input since regardless of the value that A has, the gate's output would be '1'.

Electrical masking occurs when, as the signal propagates, due to the electrical properties of the subsequent logic gates, the pulse suffers from attenuation to a point that it is not of sufficient magnitude to upset any downstream state element (Figure 4-13).

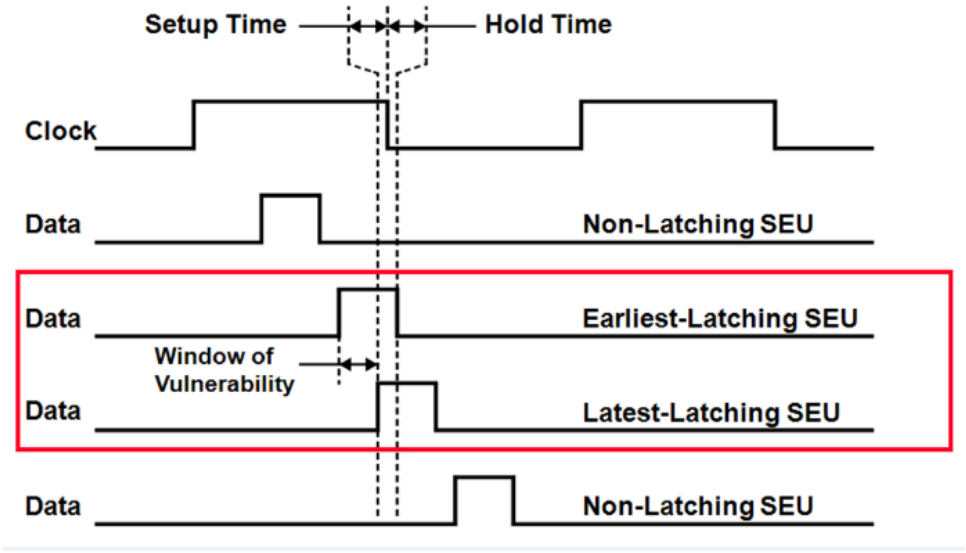


Figure 4-14. Latch window masking; temporal relationship of latching a data SET as an error (Mavis and Eaton, 2002)

Latch window masking, also called timing windows masking, occurs when the undesired pulse reaches a latch at the wrong time of the clock transition (Cha et al., 1993). That is, the pulse does not satisfy the compulsory setup and hold time of the flip-flop. The transient will get latched if the pulse reaches the latch within the "window of vulnerability" (Figure 4-14), hence causing data corruption.

In terms of upset tolerance of single gates, there are two characteristics of interest: glitch generation and glitch propagation (Dhillon et al., 2005). The shape and the magnitude of the voltage glitch generated at the gate's output are determined by the glitch generation characteristics. The voltage magnitude of the glitch depends on the total capacitance of the node while the duration of the glitch depends on the gate's delay. Faster gates lead to wider glitches and therefore better generation characteristics.

Alternatively, the glitch propagation characteristics of a logic gate determine the glitch attenuation as it passes through the gate. Assuming a linear ramp at the output of a gate, where d is the gate propagation delay and w_i is the glitch duration at the gate input, the glitch duration of the gate w_0 can be approximated using Equation 4.4 (Dhillon et al., 2005) as:

$$w_0 = 0 \quad if \quad d > w_i$$

$$w_0 = 2(w_i - d) \quad if \quad 2d > w_i > d$$

$$w_0 = w_i \quad if \quad w_i > 2d$$

Equation 4.4. Approximation of the glitch duration of a gate [83]

According to Equation 4, slower gates will induce more attenuation on glitches than faster gates. Therefore, fast gates have better glitch propagation characteristics. An increase in the gates capacitance would increase the delay of the gate, which in turn, would reduce the glitch propagation characteristics. SETs affecting the clock logic or the reset trees can lead to much larger problems (see Single event functional interrupts section).

In the past, these masking effects are some of the reasons why SETs have not been a dominant contributor in the overall SER. In addition, designers have not been significantly concerned about errors in microprocessor logic because the number of flops on microprocessors was much fewer than the number of memory cells. Since flop protection techniques are more difficult to implement than memory protection mechanisms such as parity or ECC, from 90nm downwards, flop SEU rates are higher than SRAM SEU rates.

SETs are particularly worrisome in safety-critical applications whose memory has been protected to decrease SEU rates. In this type of systems, SET rates can be the dominant reliability failure mechanism.

4.5.2.3. Single event functional interrupt (SEFI)

SEFI represent the most disruptive version of non-destructive SEE. Although this type of anomaly was previously predicted for space environments (Koga et al., 1985), the term single event functional interrupt (SEFI) was first mentioned in 1996 ("EIA/JEDEC STANDARD, Test Procedures for the Measurement of Single-Event Effects in Semiconductor Devices from Heavy Ion Irradiation"

1996). SEFI is defined as all non-destructive failure modes that lead to the malfunction (or interruption of normal operation) of a part or the totality of the device (Bougerol et al., 2008). This definition is in contrast with certain authors that define SEFI as the cause of a higher error rate than expected due to uniformly distributed upsets (Crain et al., 1999; LaBel et al., 1996).

The causes and effects of SEFIs vary from the type of component and the technology used. In general, SEFIs are linked to an upset (SET or SEU) in a control area that configures a specific function, and leads to the loss of that function. In contrast to SEUs and SETs that may or may not affect the operation of the device, every single type of SEFI leads to a direct malfunction. Figure 4-12 shows an SET affecting combinational logic, not affected by the logical and electrical masking mechanisms (as in Figure 4-13), that propagates to a register in a control area within the latch window (as in Figure 4-14). If the register affected is being used by a vital part of the system software, a SEFI could take place.

Table 4-5. Classification of SEFI

Name	Also called	Typical Effect	Recovery procedure	Technology affected	Examples
Logic SEFI	Address error, recoverable bust error, temporary block error	Reading/writing of the wrong row, column; 512-8k addresses in errors	Rewriting of the right value	Complex memories such SDRAM	Fuse latch upsets (SEFLUs)
Soft SEFI	Resettable SEFI	Functionality loss of up to a full memory bank	Refresh cycles	FPGA, microprocessors, complex memories	Stuck block errors
Hard SEFI	Permanent SEFI, Reboot SEFI	Complete loss of functionality	Complete power cycle of the device	FPGA, microprocessors, complex memories	Events that induce data and functionality loss that cannot be recovered

As microcircuits become more complex they also become more susceptible to SEFIs; among those: SDRAMs (Harboe-Sorensen et al., 2007) with complex internal architecture (such as state machine), FLASH memories (Irom and Nguyen, 2007; Nguyen et al., 1999; Oldham et al., 2008), FPGA (Czajkowski et al., 2006) and microprocessors (Czajkowski et al., 2005). Dependent on cause, consequences and recovery procedures, SEFIs can be classified as logic, soft or hard (see Table 4-5).

Logic SEFIs (Bougerol et al., 2008): with regards to memories, it is also called "address error", "recoverable burst error" (R. Ladbury et al., 2006) or "temporary block error" and mainly includes row and column errors. The upset of a row or column register leads to the reading or writing of the wrong row/column. This type of SEFI typically causes between X and 8k addresses in errors where X is the number of addresses per row/column (Bougerol et al., 2008). Rewriting of the right values is used as to recover functionality (Schagaev and Buhanova, 2001).

Examples of logic SEFIs are "fuse latch upsets" also called SEFLUs (Bougerol et al., 2011, 2010) that lead to the wrong addressing of a whole row/column. Manufacturers are experiencing an increasing number of defective cells, therefore adding spare cells and exposing them to reliability tests. If during those tests, a cell is found defective, fuse latches are used to disable the particular row/column. Typical signatures of fuse latch upsets are multiples of X addresses where X is the number of addresses belonging to a column/row.

Soft SEFIs also called "Resettable SEFIs" (Bougerol et al., 2008; Lawrence, 2007) are due to upsets in the device configuration area and usually induce the functionality loss of several thousands of addresses up to a full memory bank. Reconfiguration of the device with a mode register set command can be used as a recovery procedure of the functionality (but not the data). Examples of this are "block SEFIs" also called "stuck block errors", observed in the IBM Luna-ES rev C during heavy ion testing ("NASNGSFC Landsat-7 Project Office, Private Communication," 1995) where an entire row of 1024 addresses was stuck to a

specific value. Since simple writing was not sufficient, device refresh cycles were used to clear the problem. SEUs in selected areas of an FPGA such the JTAG bit serial configuration port can lead to inability of reconfiguration.

Hard SEFIs (Bougerol et al., 2010; Harboe-Sorensen et al., 2007), also called Reboot SEFIs (Bougerol et al., 2008), "permanent SEFIs" (Slayman, 2005), "non resettable errors" (Lawrence, 2007, p. 512) or "persistent non recoverable errors" (R. Ladbury et al., 2006) can be induced by different phenomena and lead to the complete loss of memory functionality. Possible causes of this type of catastrophic SEFI are upsets in the internal state machine, counter registers or activation of special modes. An example of this is an SEU in one of the power on reset registers that can lead to the removal of the entire configuration area. Complete power cycle of the device is compulsory as a recovery procedure.

Fortunately, the probability of SEFI is low compared to other types of SEEs (Slayman, 2005). The reasons for that are:

- The ratio of the periphery logic area to memory array area is very low;
- The critical charge for logic gates is usually higher than for SRAM cells.
- The most part of the periphery logic is combinational, and therefore less susceptible to upsets due to the three inherent masking mechanisms.

SEFIs can also be classified as high current SEFIs if they involve a certain increase in current (Koga et al., 2001a, 2001b).

In addition to SEFIs in complex memories, the energetic particles can also strike other circuits such that the error detection and correction mechanisms affect the functioning of the whole circuit. In FPGAs, SEFIs can cause the device to stop from functioning normally and therefore require a power reset in order to resume normal operations. In microprocessors, SEFIs can induce upsets in the program counter, illegal branching and jumps to undefined states.

4.5.2.4. Single event latchup (SEL) and other destructive effects

Also called “hard errors” or “non reversible errors”, “single event destructive effects” are events that momentarily or permanently change the state of a device or cell/node affecting their functionality. Destructive effects are persistent even after a reset or reconfiguration and a replacement of components may be required.

4.5.2.4.1. Single event latchup

A latchup is an unintended and potentially catastrophic state that affects CMOS devices, characterized by excessive current flow between a power supply and its ground rail.

It can take place due to the interaction between parasitic structures, usually an npn- and a pnp- bipolar transistor. A low resistance path develops between ground and power supply of the device and remains after the triggering event has been removed. Once triggered, a latchup can amplify currents to a point where the device fails as a result of thermal overstress. This electrically induced effect typically occurs in improperly design circuits. However, it was demonstrated (Leavy and Poll, 1969) that a latchup could also be induced via ionising radiation (SEL), including high-energy protons, alpha particles, cosmic rays and heavy ions. The difference between a conventional latchup (electrical) and a single event latchup (SEL) is that latter phenomenon is triggered by an energetic particle instead of an electrical overvoltage.

A classification of different SEL is shown in Table 4-6:

Table 4-6. Classification of SEL

Name	Type of error	Nature	Recovery procedure
Traditional or destructive SEL	Hard	High current	Replacement of components
Non-destructive SEL	Pseudo-hard	Low current	System restart
Microlatchup	Pseudo-hard	Localized, high current	Reducing the power supply voltage below the holding voltage or reset

Parasitic transistors of CMOS devices can be triggered by the strike of high-energy protons, alpha particles, neutrons and heavy ions. An SEL may occur if enough energy, critical charge, is deposited by a given particle within a microscopic region of the device, regardless of the total flux. High currents can lead to metal traces to vaporize, bond wires to fuse open and silicon regions can be melted due to thermal runaway. Hence, the latched condition may potentially destroy the device, affect other surrounding devices and destroy the power supply (traditional or destructive SEL). In certain cases after one or several SEL can make the device more susceptible to future SEUs.

Both high current and low current SELs can occur (K. LaBel et al., 1992). Modern devices may have many different latchup paths, making characterization of those latchup states a challenging task. In some cases, events resulting in localized high current (microlatchups) can remain functional. In order to restore the device to a normal operation, these effects can be tolerated by reducing the power supply voltage below the holding voltage e.g. power off-on reset (PCSE).

Additionally, latent damages have been observed in several types of CMOS devices after non-destructive latchup events (Becker et al., 2002). Becker defines

latent damages as *"structural damages that cause no electrically observable parametric or catastrophic device failure, but can be detected by surface analysis using optical or scanning electron microscopy"*. These type of permanent structural damages are a potential reliability hazard since the interconnect cross-sections in the damaged area may be reduced by one or two orders of magnitude.

Sometimes the SELs are not localized affecting the entire device, but the current may not be high enough to destroy the device (non-destructive SEL). Therefore, SELs are not invariably destructive and can also be categorized as pseudo hard errors.

Temperature is an important factor in SEL susceptibility. Higher temperatures involve a cross section increment and reduction of SEL threshold (Johnston et al., 1991).

SELs can be mitigated through internal fabrication process modification. Silicon on insulator (SOI), silicon on sapphire (SOS) and the use of epitaxial substrates are immune to this type of effects (Miller and Mullin, 1991). However, those are very expensive and their availability normally limited to mission critical systems in space environments (Pouponnot, 2005). Additionally, different layout techniques, like guard drains and guard rings, are often used in CMOS processes. Alternatively, SEL can be circumvented externally through the use of current sensing, watchdogs, etc. Internal methods are trying to keep the event from occurring. With external mechanisms, the event still occurs, but there should be a recovery strategy to deal with them.

4.5.2.4.2. Single event hard error (SHE or SEHR) or stuck bits

Since the mid-1980s certain SRAM devices, when exposed to heavy ions, experienced semi-permanent stored bit patterns or stuck bits with no implication of total dose effects. This form of damage was not reported until 1991 (Koga et al., 1991) and was later studied and renamed as "single hard

error" (SHE) (Dufour et al., 1992). SHE is an unalterable change of state of a memory element associated with semi-permanent damage due to high-localized dose deposition from a single ion track. This type of effect affects memories (SRAM, DRAM, Flash) and latches in logic devices rendering the cell unprogrammable (Dufour et al., 1992).

The cell may have an indeterminate value, also appearing as a permanent fault at the system level. SHEs are considered semi-permanent since some of the stuck bits tend to disappear (in some cases after a day (Duzellier et al., 1993)).

4.5.2.4.3. Single event snapback (SES or SESB)

This type of effect induce high currents in most cases and is particularly difficult to differentiate from high current SELs (Beitman, 1988; Koga and Kolasinski, 1989). While SESBs can take place in technologies immune to SEL, it does not require a four-region structure to arise. In this context, snapback has been confirmed to be particularly susceptible to SOI structures because of their internal design (Dodd et al., 2000).

With regards to SESB and NMOS technology, the parasitic NPN bipolar transistor that exists between the drain and the source amplifies the avalanche current resulting from the impact of an ionising particle. The transistor then opens and remains open.

Like SEL, SESB is also considered a potentially catastrophic event since it can lead to device destruction if not corrected within a short time of occurrence. The main differences between SEL and SESB lie in the amplitude of the current increase, their temperature dependence and recovery conditions. First, unlike destructive SEL, it is often possible to restore normal operation and bring the device out of the high current mode by changing the gate voltage without shutting off the power supply. Secondly, the amplitude of the current increase is much lower for SESB due to its localized nature. Finally, contrary to SEL, SESB is

weakly dependent on temperature (Johnston, 1996). These facts can be used to distinguish between SESB and SEL mechanisms.

4.5.2.4.4. Single event burnout (SEB or SEBO)

SEBO typically occurs in power metal oxide semiconductor field-effect transistors (power MOSFETs) and bipolar transistors since these devices contain parasitic bipolar transistors between the drain and the source (Hohl and Galloway, 1987; Waskiewicz et al., 1986). SEBO creates a permanent short between a source and a drain and involves high currents and localized overheating. If the device is not provided with current limitation capabilities, and the drain-source voltage exceeds the local breakdown voltage of the transistor, the SEBO can lead to the destruction of the device by melting of the silicon in the affected region (Stassinopoulos et al., 1992)(see Figure 4-15).

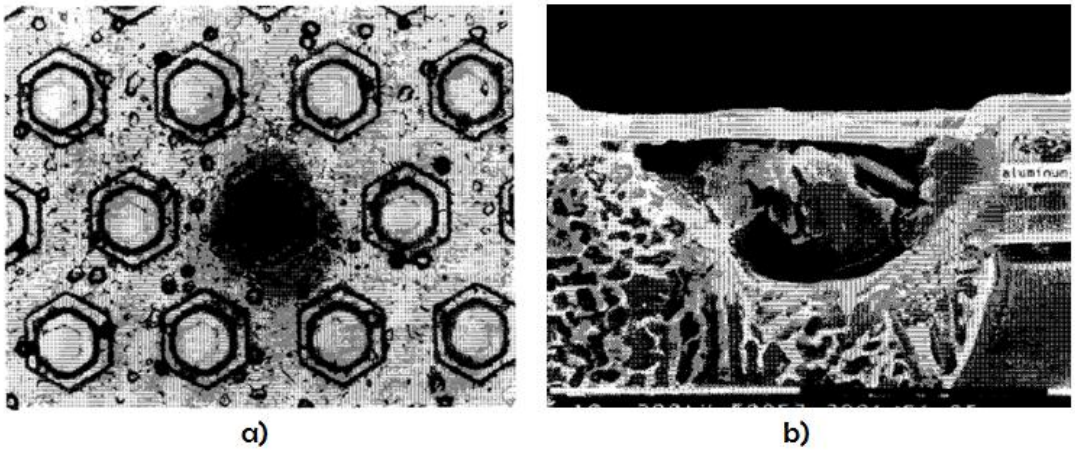


Figure 4-15. IRF 150 power MOSFET burnout: a) Optical view of burnout area on the surface, b) Scanning electron microscope (SEM) sectional view of a burnout area with 1000x magnification (Stassinopoulos et al., 1992)

SEBO has occurred in low voltage devices, however devices with high voltages are more prone to this type of error.

With regards to temperature, it has been shown (Johnson et al., 1992) that higher temperatures decrease the SEBO susceptibility. The probability of SEBO

occurrence is low, but apart from the selection of immune device technology, there are no mitigation techniques.

4.5.2.4.5. Single event gate rupture (SEGR)

It was first observed in non-volatile memories in 1980 (Pickel and Blandford, 1980) and later identified and confirmed in 1984 (Blandford et al., 1984). In 1987 was reported in power MOSFETs (Fischer, 1987) but due to the scaling of CMOS technology SEGR has become a concern in low voltage circuits (Silvestri et al., 2009). This type of single event is often observed with SEB in power MOSFETs. SEGR is triggered by a single ionising particle in a high field region of a gate oxide, creating a localized gate rupture in such area (Sexton et al., 1997). This rupture manifests as a permanent conducting path between the gate and the drain (gate rupture – see Figure 4-16). As a result, the electrical performance is compromised and the functionality of the device may be affected.

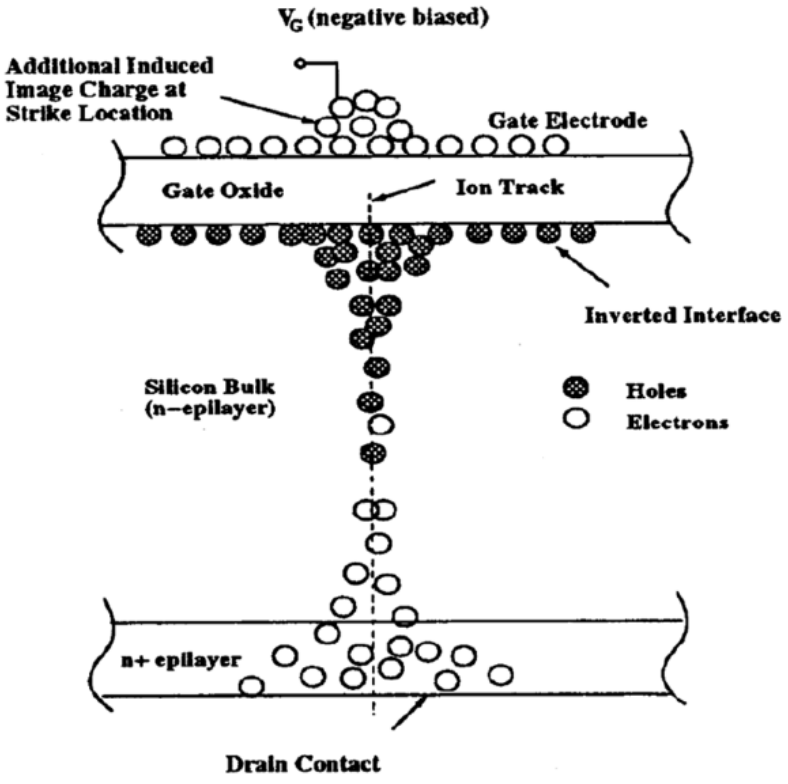


Figure 4-16. SEGR as a result of the impact of a highly energetic particle. Holes from the particle's track aggregate under the gate oxide increasing the high field of the gate oxide to the dielectric breakdown point (Allenspach et al., 1994)

Flash memories (Oldham et al., 2006) and non-volatile SRAM are SEGR susceptible during a write or clear operation due to the large voltage applied to the memory elements. SEGR is not typical of avionics and ground equipment. Like SEBO, the probability of occurrence is low, but should be taken into account in the component selection process.

In order to mitigate SEGR, voltage derating and limiting the available energy to a device can be employed.

4.5.2.4.6. Single event dielectric rupture (SEDR)

Also called "micro damages", SEDR was encountered during heavy ion SEE testing of antifuse FPGA (Katz et al., 1994) and eventually identified as ion induced rupture of antifuses. Similar to the SEGRs observed in power MOSFETs, SEDRs affect non-volatile NMOS devices and non-volatile FPGAs (Katz et al., 1997; Swift and Katz, 1996). SEDRs are triggered by a single ionising particle, and lead to the formation of a conducting path in a high field region of a dielectric.

4.6. Conclusion

This chapter presents the long-term cumulative and short-term effects of radiation in embedded systems. First, we present an overview of the fundamental damage mechanisms and, resulting from such mechanisms we introduce the major macro effects. Secondly, we focus on the short-term degradation induced by ionizing particles, namely single event effects. Thirdly, we describe the physical mechanisms that are responsible for SEE including charge deposition, charge transport, charge collection, to finally fully describe the different circuit responses. As a result, an extensive taxonomy of SEE has been produced, describing their nature, type of degradation, susceptibility, fault rate trends and recoverability.

Radiation can have a major impact on all kind of embedded microelectronics potentially leading to catastrophic failures. As we move to denser semiconductor technologies at lower voltages, system SER will continue to rise and in particular the contribution of single event upsets, single event transients, multi-cell upsets and single event functional interrupts will increase. Error correcting codes are not efficient when dealing with certain multi-bit faults and errors in combination logic. In the case of safety-critical embedded systems, more efforts need to be directed towards research on mitigation techniques for the recent and future undesired effects.

Chapter 5

Chapter 5

FT models

5.1. Models

We define *M* as the known model of a system that performs a given function *F*. Let's imagine a new feature of extreme reliability in that model. In order to express the existence of this new feature, the predicates *P* and *Q* are introduced to determine the state of the model. *P* and *Q* also defined the direction of the time arrow (see Figure 5-1).

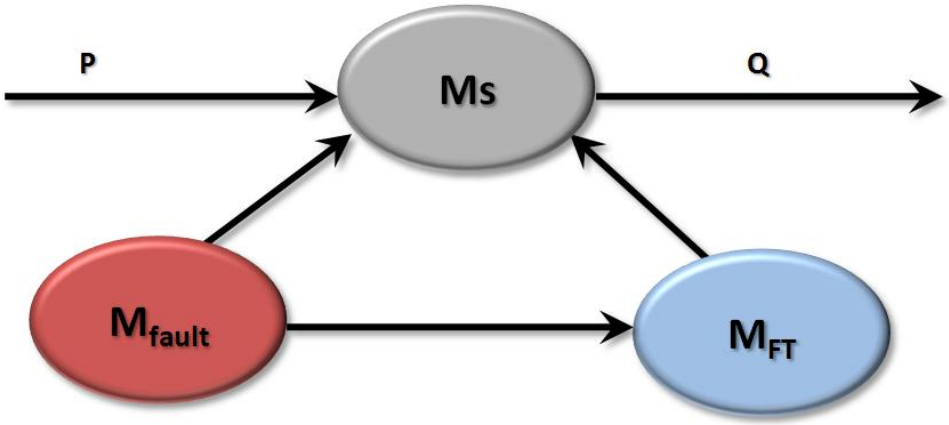


Figure 5-1. New feature of an *FT* system: reliability

To analyse methods for achieving a required level of reliability with performance and power consumption constraints, we offer a combination of the following three models:

- The model of the system M_s
- The model of the faults M_{fault} that a *RT FT* system will be exposed to
- The model of fault tolerance M_{FT} or the new structure that implements *FT*

As shown in Figure 5-1, M_s, M_{fault} and M_{FT} are mutually dependent models. Notice that in this approach development and manufacturing costs of a solution are not considered.

M_{fault} is a description of all faults that a system must tolerate. In binary logic a typical permanent fault can manifest as "stuck at zero" or "stuck at one".

Table 5-1. Typical examples of HW faults

Type of fault	Description	Impact
Byzantine	The behaviour of a component that gives conflicting values to other components	The entire system is affected
Subsystem fault	A temporary or permanent incorrect behaviour of a subsystem	The entire system is affected
Open fault	Resistance on either a line or a block due to a bad connection	The value associated to the line or the block is modified
Bridging fault	Signals S_1 and S_2 are connected unintentionally	The value associated to the line or the block is modified to a different value
Stuck-at fault	The result value is fixed to *0* or *1*	The result value is stuck to *0* or *1*
Bit-flip fault	A state switch from *0* to *1*, or vice versa, when it should not	The result changes its original value

Table 5-1 shows typical examples of HW faults. Hidden faults, also called Latent faults are behavioural faults that exist in the hardware over a long period of time, e.g.: Byzantine faults[22] and fail-stop[23] faults. Both types complicate the design of *FT*; all described faults should be tolerated within a limited and specified period of time. This period actually determines the availability of the system. Fault types differ by their impact, as well as the way they are handled.

Thus, the fault model has its own hierarchy, including single-bit, element, behavioural and subsystem faults. One has to accept that the fault type is varying and some action hierarchy to tolerate them is also required. A detailed fault model is further developed and discussed in Sections 5.2, 5.3 and 5.4.

[22] Byzantine faults occur when a faulty system continues to operate, producing incorrect results sometimes giving the impression that they are working correctly. Dealing with this type of fault is difficult

[23] Fail-stop (also known as fail-silent) faults take place when a faulty unit stops functioning producing no bad output. It either produces not output or produces correct results that clearly indicate that the unit has failed.

Fault encapsulation approaches can help to handle faults: due to deliberate design solutions it is possible to ensure that severe faults in the system do not escalate and remain simpler to handle; therefore making the fault handling practically possible to implement.

RT FT system applications assume long operational life; however, fault-handling schemes are needed much more often towards the end of the device lifecycle. The appropriate techniques for tolerating faults of various types are presented on Table 5-1. As discussed in Section 3.6.1 to tolerate transient faults, time redundancy in hardware (e.g. instruction re-execution) might be effectively used and implemented. System software support is also needed, as the hardware cannot cover all possible faults.

Faults, occurring at the bit level (stuck zero, stuck one and similar) should be efficiently handled *ASAP* (as soon as possible) and *ALAP* (as local as possible), i.e. at the same or nearest level. The term "*level*" in our case means the level in the hardware hierarchy on which the fault should be handled. For instance, when a "*stuck-at zero*" permanent fault has occurred in the register file (RF) with no corrective schemes available, the whole RF has to be replaced, if no other possible reconfigurations were predefined. In turn, when only one RF is integrated in the chip and no other reconfigurations are defined then the whole chip has to be replaced, etc. Pursuing these two principles allows limiting the fault spreading and its impact to a higher level either in the chip or the system as a whole.

To tolerate bit-flip faults, hardware and system software information redundancies might be used, as well as hardware structural support. In this sense parity checking in registers, supported and implemented concurrently by hardware, is described as *HW(δI)*. *HW(δS)* and *HW(δT)* are needed as supportive redundancies, *HW(δS)* describing the additional parity line and comparison logic, and *HW(δT)* describing the additional time needed to update the parity line and executing the comparison. Nonetheless, the main type of redundancy used in this approach is information.

An exact characterization of the distribution of faults for computer systems is extremely difficult due to the number of different factors that determine faults, such temperature, vibration, radiation exposure etc. Besides, discriminating between transient and permanent faults is difficult. The transient-permanent fault ratio varies from 10 to more than 1000 depending on the technology, manufacturing scale, operating conditions, etc. In the case of memories a typical value of hard error rates is in the order of 10-100 FIT whereas for soft errors it can vary between 1000 and more than 5000. The upper bound belongs to aerospace and aviation, principally due to faults induced by alpha particles.

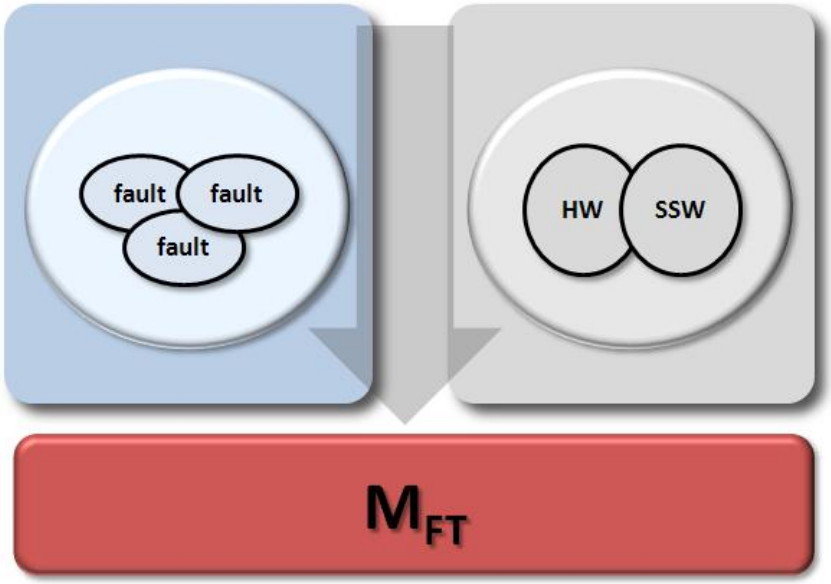


Figure 5-2. Fault tolerance model of a computer system

Figure 5-2 is a combination of Figure 3-2 and Figure 5-1 and presents various faults in the system and various possible solutions. M_{fault} illustrates the fact that the fault types are not separated. For example, Byzantine faults of the system might be "*stuck at zero*" faults of the hardware that were spread throughout the system. The latency of faults becomes crucial in determining the reliability of the system. Consequently different faults require different actions and mechanisms to tolerate them.

The system model of Figure 5-2 has overlapped *SSW* and *HW* ellipses to represent the duality of the system: hardware and system software. Both of them must be involved in the implementation of fault tolerance and real time features. The overlapped *HW* and *SSW* ellipses indicate that HW and SSW functions might be applied to tolerate specific types of hardware faults. Other fault types might also be tolerated by HW or SSW only. M_{ft} is "a conceptual deliverer" of reliability for the *RT FT* system. It has to be effective during the whole operational lifetime of the computer itself. As hardware degrades over time, the fault tolerance mechanisms are more likely to be used towards the end of the lifecycle. FT systems are designed with the assumption that new types of faults do not appear during the operational lifetime of the system, i.e. the system must be designed to be fault tolerant for the set of faults and their types known at design time. All these solutions require careful analysis due to their impact on the system reliability.

In contrast to the usual assumption in reliability modelling, one has to assume that a fault might exist in the system over an arbitrary long period of time (latent fault) and its detection and elimination is not possible "at once". Consequently we accept that FT is a process, and discuss it in the following sections. Using Dijkstra's approach (Dijkstra, 1965) of defining a function as a process described by its algorithm, we consider FT as a function that is also described and implemented by an algorithm.

There are several options to achieve fault tolerance assuming the use of *HW* and *SSW* by using various types of redundancy mentioned above. However, the use of certain redundancy types might cause system performance degradation which is especially true for software measures (Kulkarni et al., 1987; Oh et al., 2002a). Further analysis of performance/reliability degradation should be taken into account.

The introduced system redundancy might be used in a way to tolerate only certain fault types, thus degrading fault coverage, keeping performance at acceptable levels. Software based redundancy might preserve the same type of

fault coverage but with more time redundancy - delays (recovery time degrades, availability degrades). Alternatively, the fault coverage might not preserve the same level, thus the system degrades in terms of reliability.

We assume that regardless of the accuracy of the model, new faults can appear in the system and for some of those our system may not be able to detect, diagnose or recover.

5.2. Fault model

It is unfeasible to describe all possible faults that may occur in a system. In order to make the evaluation of faults possible, they are assumed to behave according to some fault model. A *fault model* (*FM*) is as a way of summarising many fault descriptions at once (Dunn, 1991). Often it is desirable to discuss many different faults at the same time and to describe their common characteristics. Fault models are used to represent in a simple form the consequence of complex physical mechanisms that lead to faults. In the case of electronic systems, the modelling of faults can be implemented at two different levels: at the level of hardware components that implement a system (e.g. memory subsystems, register banks, *ALU*) or at the system level, which is directly related with the information that the system manipulates (e.g. instructions and data program).

The simplest *FM* is to consider the logic gate as a single unit with a constant failure rate, instead of considering different failure rates for the individual transistors that form the unit. As in 4.5.1, analysing the physics of faults to the atomic and molecular level would provide a clear understanding of the failure mechanisms. Such understanding is very helpful in the development of fault models. Primarily based on the work of (Avizienis et al., 2004), we further extend the classification of faults depending on the way they are originated or manifested.

Table 5-2. Classification of faults by origin

Level of abstraction	Structural faults	Transistor level (component faults)		Stuck-open or stuck-off
				Stuck-short or stuck-on
		Gate level (interconnect faults)		Stuck-at *(Armstrong, 1966; Galey et al., 1961)*: s-a-0, s-a-1
				Timing delay *(Smith, 1985)*: path-delay (PDF) and gate-delay (GDF) faults
				Bridging *(Mei, 1974)*
	Functional faults	Pattern sensitive faults (PSF) *(Hayes, 1975)*		Passive PSF
				Active PSF
				Static PSF
		Coupling faults (CF)		
Phase of creation of occurrence *(Landwehr et al., 1994)*	Development faults	e.g. specification faults, implementation and manufacturing faults		
	Operational faults	e.g. aging faults e.g. alpha particle hits		
System boundaries *(Avizienis et al., 2004)*	Internal	e.g. design and implementation faults		
	External	e.g. radiation, temperature changes, power surges from external power supply		
Phenomenological cause	Natural *(Jennings, 1990)*			
	Human-made *(Hugue and Purtilo, 2002)*			
Capability/objective/intent *(Brocklehurst et al., 1994)*	Malicious	Deliberate		
	Non-malicious	Accidental		
		Incompetence		
Nature *(Avizienis et al., 2004)*	Hardware	Cell errors, combinational logic errors …		
	Software	Branch errors, missing instructions, missing pointers …		
Cause	Specification mistakes			
	Defects	Implementation mistakes		e.g. Pentium FDIV bug *(Coe et al., 1995; Price, 1995)*
		Manufacturing defects		Global defects or Gross area defects *(Koren and Koren, 1998)*
				Spot defects *(Koren and Singh, 1990)*
	Operating environment – External disturbances	Thermal stress		
		Heat		
		Electro-migration EM		
		Voltage drop		
		Noise	Electrical overstress	Hot carrier injection HCI (DAHC, CHE, SHE, SGHE)
				Negative Bias temperature instability (NBTI)
				Latchup
			Induced charging	
			Oxide Breakdown	
			Radiation	See Table 4-1, Table 4-2, Table 4-3, Table 4-4, Table 4-5, Table 4-6
			EMP	

5.3. Classification of faults by origin

Faults can be classified differently depending on attributes related to their origin, including their cause, the level at which they take place, the phase of creation, nature, system boundaries, phenomenological causes and intention. Table 5-2 shows a number of faults classified by their origin attributes.

5.3.1. Level of abstraction and fault models

Hardware defects can be the source of *physical faults*. In order to simplify the fault analysis process *Logical faults* can be used to model the manifestation of physical faults on the behaviour of a system. They can be subdivided into *structural faults*, which are related to structural models and modify the interconnection among components, and *functional faults*, which are related to functional models and change the functions of components and circuits.

Component faults are a type of structural faults, which can be applied at the *transistor level*. Some of these are:

- *stuck-open* or *stuck-off*: a transistor is always off and not controllable by gate input
- *stuck-short* or *stuck on*: a transistor is always on and not controllable by gate input

Another type of structural faults, *interconnect faults*, can be applied at the *gate level*. Among these:

- *stuck-at faults (SAF)*: single or multiple lines have a constant value of 0 (*s-a-0 faults*) or 1 (*s-a-1 faults*) regardless of the value of the other signals in the circuit (Armstrong, 1966; Galey et al., 1961)
- *timing or delay faults*: certain defects due to manufacturing or external reasons do not change the logic function of components, but can cause timing violations; faults due to propagation delays along a path (*path-*

delay faults, PDF) or gate (*transition* or *gate delay fault, GDF*) (Smith, 1985), exceeding the limits required for correct operation

- *bridging faults (BFs)*: two or more distinct lines are shorted (Mei, 1974) usually due to particles or shorted metal lines. Depending on the value the bridging could be *AND bridging* (also referred to as *0-dominant*), *OR bridging* (also referred to as *1-dominant*) or *Indeterminate Bridging*. It is obvious that the probability of BF's occurring increases with 1) shorter distances between metal lines due to the use of shrinking technology and 2) the use of long parallel lines (Tehranipoor et al., 2012).

Different types of functional faults that can be applied at the *functional level* are:

- *pattern sensitive faults (PSFs)*: where a fault signal depends on the signal values of nearby components (Hayes, 1975); typical in DRAM, there are three types of PSFs due to changes in the neighbourhood pattern:
  - Passive PSF: the value of a cell remains
  - Active PSF: the value of a cell changes
  - Static PSF: the value of a cell is being forced to a particular state (0/1)
- *coupling faults (CF)*: A subset of SPF, represent a specific pattern sensitivity between two memory cells (Nair et al., 1978); Two memory cells C_j and C_j are coupled if a transition from X to Y in one cell, say C_i, changes the state of the other cell, given that:

$$X \in \{0,1\} \quad and \quad X = \overline{A}$$

  - Idempotent *coupling faults*: a transition $0 \rightarrow 1$, or $1 \rightarrow 0$ in C_i forces the contents of C_j to a specific value $X \in \{0,1\}$
  - Inversion *coupling faults*: a transition $0 \rightarrow 1$, or $1 \rightarrow 0$ in C_i forces an inversion $0 \rightarrow 1$, or $1 \rightarrow 0$ of C_j.

5.3.2. Cause of faults

5.3.2.1. Specification mistakes

Specification mistakes, which take place during the planning and design phases, can be the source of faults (*specification faults*), including incorrect timing, power and environmental requirements. The effect of certain specification faults may be corrected via fault masking.

5.3.2.2. Defects

A hardware defect in electronics is the unintended difference between the implementation and the intended design. *Implementation mistakes*, such as the Pentium FDIV bug (Coe et al., 1995; Price, 1995), are a type of defects. Conversely, imperfections in the fabrication process of state of the art VLSI technologies result in *manufacturing defects*, whose severity increases proportionally with the size and density of the chip.

Manufacturing defects are largely dependent on the specific technology and layout, and include processing and material defects such: dust particles on the chip, conducting layer defects (shorts and opens), oxide defects, scratches and gate oxide pinholes, defects caused by either extra or missing material (Koren and Koren, 1998). Manufacturing defects can be classified as *global defects* (or *gross area defects*), affecting large areas of a wafer and so can be easily detected during manufacturing, or as *spot defects*, which are random, affecting areas comparable to the single device size, and therefore more difficult to be detected (Koren and Singh, 1990).

5.3.2.3. Operating environment

Correct functioning of digital systems is based on the assumption that electrical and timing transistor parameters will remain bounded to certain margins (usually *±15%*). These margin tolerance specified at initial manufacturing can be violated during operating time due to shifts induced by external disturbances.

These mechanisms can produce systematic degradation overtime or abrupt failures of basic components. Transistors can be degraded due to electrical overstress and radiation whereas oxide-breakdown, electrostatic discharge and ionizing radiation are usually the cause of abrupt failures.

Hot carrier injection (*HCI*) has been one of the most common electrical overstress aging mechanism, adversely affecting both *nMOS* and *pMOS* transistors. It occurs when a charge carrier, an electron or a hole, gain enough kinetic energy to break an interface state. Different mechanisms can be responsible for *HCI* including s*ubstrate hot electrons* (*SHE*) (Ning and Yu, 1974), c*hannel hot electrons* (*CHE*) (Cottrell et al., 1979), *drain avalanche hot carriers* (*DAHC*) (Takeda et al., 1983) and *secondarily generated hot electrons* (*SGHE*) (Matsunaga et al., 1980)

Negative Bias temperature instability (*NBTI*) (Schroder and Babcock, 2003) is also a critical reliability concern for p*MOS* transistors (not so much for n*MOS)* and has been a persistent issue for generations below *130nm* (Schroder at. All, 2003, Alam, 2007). Interface traps are generated during negative bias conditions ($V_{gs} = -V_{dd}$). Higher temperatures seem worsen *NBTI*, producing larger voltage, which if maintained over long periods (*NBTI* exhibits logarithmic dependence on time), may significantly increase delays (Kumar, 2006, Kaczer, 2005).

Another example of electrical-overstress mechanism is the *latchup* described in Section 4.5.2.4.1, which can also be triggered electrically (Gregory and Shafer, 1973).

As described in Sections 4.2, 4.3 and 4.4, *non-ionizing radiation* can be the cause of *DDD* while *TID* effects can be induced by ionizing radiation. Other degradation mechanisms can affect interconnection logic, e.g. *electromigration* (*EM*).

In contrast with the previous long-term degradation mechanisms, the effect of noise can produce abrupt failures. Examples of these are faults induced by the

effects of noise including *oxide breakdown*, *electrostatic discharge* (*ESD*) and *ionizing radiation*.

Oxide Breakdown is the destruction of an oxide layer of a semiconductor device, e.g. *time dependent dielectric breakdown* (*TDDB*), *early-life dielectric breakdown* (*ELDB*), and *EOS/ESD-induced dielectric breakdown*.

The Ionizing radiation mechanisms and the faults related to it have already been discussed in Section 4.5

5.3.3. Phase of creation and occurrence of faults

Faults that take place during the manufacturing phase are *development faults* in contrast with *operational faults* that take place during the service delivery of the operation phase (Landwehr et al., 1994), e.g. faults due to radiation as in Chapter 4.

5.3.4. Nature/dimension

According to their nature faults can be categorised as *hardware* (such as combinational and sequential logic defects due to aging, radiation, etc.) or *software* (branch errors, missing instructions and pointers, etc.). The scope of this thesis focuses exclusively on hardware faults and their effects.

5.3.5. System boundaries

With respect to the system boundaries, faults can also be classified as internal (originate inside the system boundary) or external (originate outside the system boundary). *Internal faults* are those that arise from within a system, often due to design flaws. These are usually repeatable for a given set of inputs in the system. In addition, they can also be the result of implementation faults, which if random, are difficult to repeat. *External faults* are those that originate from outside the system, propagating into the system. These are normally the result of interference cause by the physical environment including *environmental faults*

(e.g. radiation, temperature changes), accidental damage from an external system (e.g. power surges from an external power supply), etc.

5.3.6. Phenomenological cause

The key components of embedded systems have an inherent susceptibility to Natural (Jennings, 1990) and human-made (Hugue and Purtilo, 2002) faults. Natural faults are generally random in nature and are caused by natural phenomena, without human participation. These are normally a consequence of environmental overstress. Human-made faults are the result of human action, including design and interaction faults (operational misuses), and are usually due to mistakes in the design, implementation, or use of systems.

5.3.7. Capability/Objective/Intent

Following the previous classification, human-made faults can either be deliberately harmful (*malicious faults)* or can be triggered without purpose and awareness (*non-malicious faults*) (Brocklehurst et al., 1994). *Accidental faults* are due to mistakes and bad decisions as long as they are not made deliberately; these include interaction, design and implementation faults. It is obvious that all natural faults have no intention and therefore are accidental. *Incompetence faults* are faults due to mistakes or bad decisions that were the result of the lack of professional competence.

Table 5-3. Classification of faults by manifestation

Response-timeliness (Qian, 2008)	Omission faults		
	Commission faults		
Dimension	Hardware		
	Software		
	System		
Activation reproducibility (Avizienis et al., 2004)	Solid		
	Elusive	E.g. pattern sensitive faults (effects of temperature, delay in timing due to parasitic capacitance)	
Extent	Local		
	Global		
Persistence/duration	Permanent	Easiest to diagnose; Once the component fails it will never work correctly again	
	Temporary	Transient Intermittent	
Value	Determinate		
	Indeterminate		
Plurality	Single		
	Multiple		
Correlation	Independent		
	Related		
Damage (see Table 4-2)	Soft	Transient-soft Static-soft	
	Hard		
	Pseudo-Hard		
Status	Dormant		
	Active		
Prospect (Laprie, 2008)	Foreseen		
	Unforeseen		
	Foreseeable		
Seriousness	Benign		
	Malicious (Meyer and Pradhan, 1991)	Symmetric	Omissive Transmissive
		Asymmetric (Thambidurai and Park, 1988)	Transmissive (Byzantine) Strictly Omissive (Azadmanesh and Kieckhafer, 2000)
Detectability (Pomeranz and Reddy, 1993)	Detectable	Recoverable DRE (Kadayif et al., 2010; Weaver et al., 2004).	
	Undetectable	Operationally redundant Unrecoverable can lead to DUE and SDC (Kadayif et al., 2010; Weaver et al., 2004).	
	Partially detectable	Under certain conditions, can be detectable and irredundant. Can lead to can lead to DUE and SDC	
Diagnosability			
Containability			
Recoverability			

5.4. Classification of faults by manifestation

Apart from their origin, faults can be classified based on attributes related to their manifestation, including their response, dimension, reproducibility, extent, persistence, value, detectability, etc. Table 5-3 shows a number of faults classified by their manifestation attributes.

5.4.1. Response-timeliness

Let a component C (see Figure 5-3) receive a nonempty input sequence ($V_i \neq null$), consistent with the specification, at time T_i. For V_i, the response V_j at time T_j is correct iff:

- $V_j = W_j$ at time T_j, where W_j is the expected value according with the specification and
- $T_j = T_i + T_d + \Delta t$, where T_d is the minimum delay time of the component, Δt is unpredictable delay time such that $0 \leq \Delta t \leq T_{max}$, given that T_{max} is the maximum unpredictable delay of C

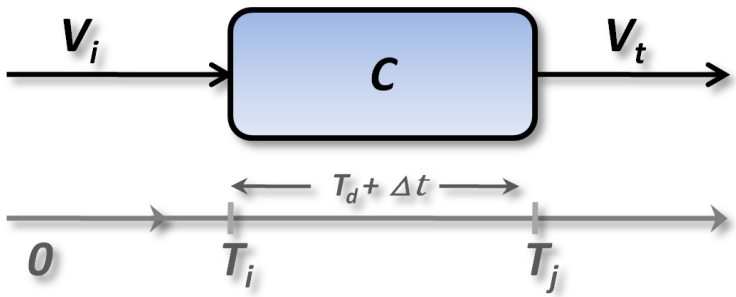


Figure 5-3. Input-response mechanism of a component C with single output

Using the previous definition of *correct response* by (Qian, 2008), there can be four ways with regards to timeliness and expected value, by which a response can deviate from the specification, which leads to the following classification of faults: omission, timing, timely and commission faults:

Omission faults involve the absence of actions when these should be performed. A fault that causes a component *C* not to respond to a nonempty input sequence ($V_i \neq null$) is an *omission fault*. The potential resulting failure would be an *omission failure*, whose response would have the following properties:

- $V_j = null$, $T_j = T_i + T_d + \Delta t$ and
- $V_j = W_j$, $T_j = \infty$

A *timing fault* is a fault that causes a component *C* to respond with the expected value *Wj* to a nonempty input sequence ($V_i \neq null$) either too early or too late. The corresponding failure would be a *timing failure*. Using the previous mathematical notation:

- $V_j = W_j$, either $T_j < T_i + T_d$ or $T_j > T_i + T_d + T_{max}$

A *timely fault* is a fault that causes a component C to respond to a nonempty input sequence ($V_i \neq null$), within the specified time interval, but with a wrong value. The corresponding failure would be a *timely failure*:

- $V_j \neq W_j$, $T_j = T_i + T_d + \Delta t$

Therefore, an omission fault is also timely fault with a null value produced on time.

A commission fault of a component C is any violation from its specified behaviour, with the following properties:

- $V_j \neq W_j$, $T_j = T_i + T_d + \Delta t$ or
- $V_j \neq W_j$, $T_j \neq T_i + T_d + \Delta t$ or
- $V_j = W_j$, $T_j \neq T_i + T_d + \Delta t$

Consequently, a commission fault is a subset of all other three types of faults.

5.4.2. Consistency

Before classifying faults with respect to consistency, the definition of correct response was extended by (Qian, 2008). Qian's definition and fault classification are suitable for systems that are required to produce replicated responses for a given input sequence. Examples of such systems are:

- TMR (Johnson, 1989; von Neumann, 1956) systems (see 3.4.1.1), a non-faulty component is required to send its output to three other components or
- A non-faulty component, which is part of some Byzantine agreement protocol, is required to send its output to other components

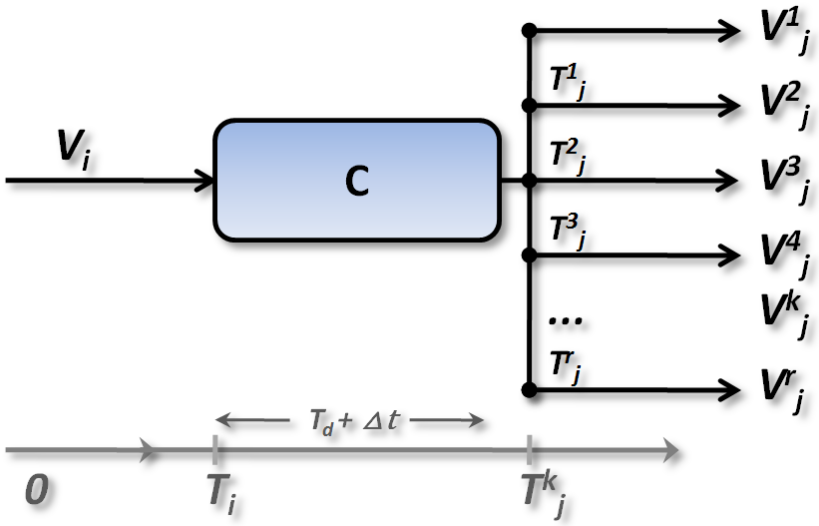


Figure 5-4. Input-response mechanism of a component C with replicated output

Figure 5-4 shows the response mechanism of a non-faulty component C with multiple r identical outputs, as a result of receiving an input sequence V_i at time T_i. The resulting outputs are defined as:

- $V_j = \{V_j^1, V_j^2, V_j^3, \ldots, V_j^r\}$, where V_j^k, $1 \leq k \leq r$, are r outputs
- $T_j = \{T_j^1, T_j^2, T_j^3, \ldots, T_j^r\}$, where V_j^k, $1 \leq k \leq r$, is produced at time T_j^k

For a component C, with an input sequence V_i, received at time T_i, its replicated response is correct (*correct replicated response*) iff:

- $V_j = W_j$, were W_j is the expected vector of replicated outputs; and
- $T_j^k = T_j = T_i + T_d + \Delta\, t$ for all k , $1 \leq k \leq r$

Therefore in a correct replicated response all individual responses must have the correct values "on time" (according to the specification), but not necessarily "*at the same time*". For instance, responses with same values, which take place at different times, nevertheless within the specified interval, would be part of a correct replicated response. Such interval is the *skew interval* and it is the period within which all individual responses are produced. It is defined as $\left|T_j^1 - T_j^k\right|$. For any two outputs:

- $\left|T_j^1 - T_j^k\right| \leq T_{max}$ for all $1,k$, $1 \leq 1, k \leq r$

For replicated-response systems, two types of faults can be considered. A *consistent fault* takes place when individual responses of a component deviate from the specification in an identical manner whereas *inconsistent faults* are the ones that cause any other breach of the specification.

An incorrect replicated response is a *consistent fault* iff:

- $V_j^1 = V_j^k$, and $\left|T_j^1 - T_j^k\right| \leq T_{max}$ for all $1,k$, $1 \leq 1, k \leq r$

Note that a consistent fault causes a component to produce identical values (not necessarily correct values) within the skew interval, although "not on time". A few examples of consistent faults are faults with the following properties are:

- some outputs being on time and the rest are produced early with correct values
- some outputs being late and having correct values but the rest are correct
- all outputs having identically incorrect values

An inconsistent fault is an incorrect replicated response iff its individual responses do not satisfy the consistent failure conditions explained above. A *byzantine fault* (the behaviour of a component that gives conflicting values to other components) is a type of inconsistent fault.

5.4.3. Maintainability: detectability, diagnosability and recoverability

Fault detection, diagnosis and recovery are required to ensure resilience. Testing and diagnosis may be online, offline or a combination of both (Kaegi-Trachsel et al., 2009). Online testing and diagnosis are performed concurrently with system operation whilst offline methods require that the system or subsystem is taken out of service for a specific time. Often, online testing is used for detection while offline diagnosis locates and identifies the fault(s). As soon as the system/subsystem is repaired, offline testing can be used to verify that the repair was successful before placing it back to normal operation.

Test vectors are used by automatic test pattern generation tools (ATPG) (Agrawal and Chakradhar, 1995; Roy et al., 1988) to attempt the detection of all or most modelled fault groups. A *test vector* is a string of n logical values (0,1, or irrelevant X) that are applied to the N corresponding primary inputs (PI) of a circuit, at the same time frame, in order to detect one or more faults (Roy et al., 1988). The specification of a test vector should have two components: the input to be applied and the expected fault-free output (e.g. *t=I/O=0010/11*). A fault will be detected if the output under fault is different than the expected output. If a series of test vectors are applied in a specific order, the term *test sequence* is used, otherwise it is a *test set* (Roy et al., 1988). The term *test pattern* is often used to refer to any of these three terms.

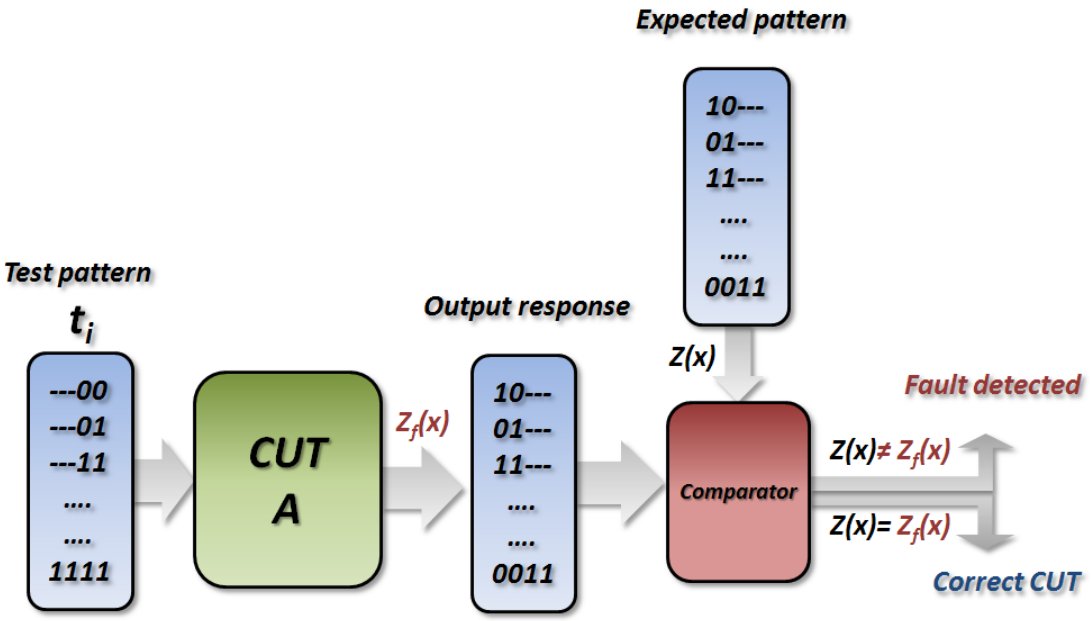


Figure 5-5. Basic testing flow of a circuit under test (CUT)

The testing process involves test pattern *generation* (ATPG), test pattern application in the CUT and output evaluation by an output response analyser (ORA) (Stroud, 2002). Figure 5-5 shows the basic testing flow of a circuit, whose output response after processing a test pattern is compared to the expected pattern (fault-free response pattern that a non-faulty circuit would exhibit). The quality of testing will depend on its fault coverage (defined in Section 2.5.2.3) and speed.

Following (Abramovici et al., 1994), let x be a random input vector, and $Z(x)$ the function of a circuit under test A with an input x. A fault f would transform A into a new circuit A_f with function $Z_f(x)$. Let be T a test set T= $\{t_1, t_2, t_3 \dots t_n\}$ formed by n t_i test vectors where $t_i \geq 1$. In Figure 5-5, the CUT A is tested by applying T and comparing the output response $Z_f(t_i)$ with the expected pattern $Z(t_i)$. A fault is detected if the output response is different than the expected pattern:

$$Z(x) \neq Z_f(x)$$

With regards to fault diagnosis, a test is said to distinguish two faults f_1 and f_2 (*distinguishable/diagnosable faults*) if the output response of the faults are different from each other:

$$Z_{f1}(t) \neq Z_{f2}(t)$$

Conversely, two faults are functionally *equivalent* if all tests that detect f1 also detect f2:

$$f_1 \sim f_2 \quad iff \quad Z_{f1}(t) = Z_{f2}(t) \quad for\ all\ t$$

Functional equivalence can be easily analysed for logic gates. An example of equivalent faults is shown in Figure 5-6a. There is not an existing test that can distinguish between the *s-a-0* faults occurring in the input and output of the *AND*. The same applies for all s-a-1 faults that occur in an *OR* gate.

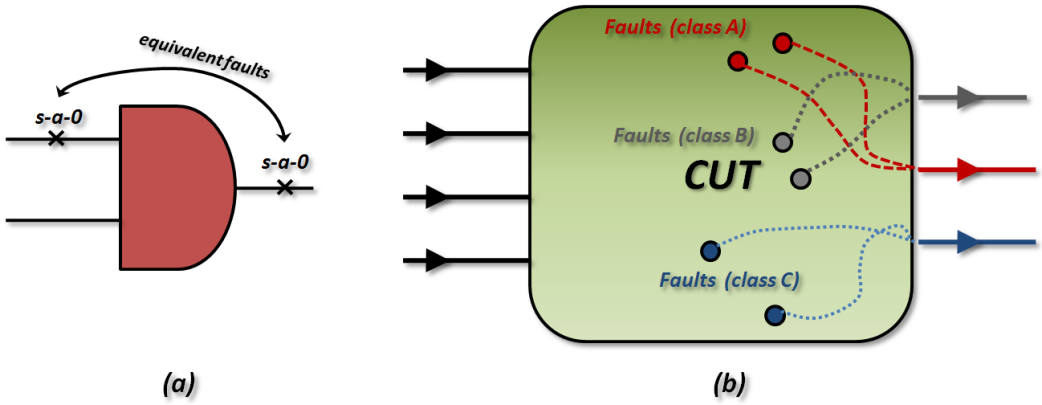


Figure 5-6. Fault diagnosis and equivalent faults. (a) example of equivalent faults. (b)Fault detection anddiagnosis vs vectors

Figure 5-6b shows six faults, two of each class A, B and C, in a CUT. Note that detection, diagnosis together with containment and recovery are some of the goals of testing (as specified in Section 2.5.2.3).

For fault detection at least one vector is needed (fault detection provides only whether the circuit is free of faults or not).

For fault diagnosis at least one vector that produces different responses for every fault class (fault diagnosis aims to determine time, location and type of the detected fault) is needed.

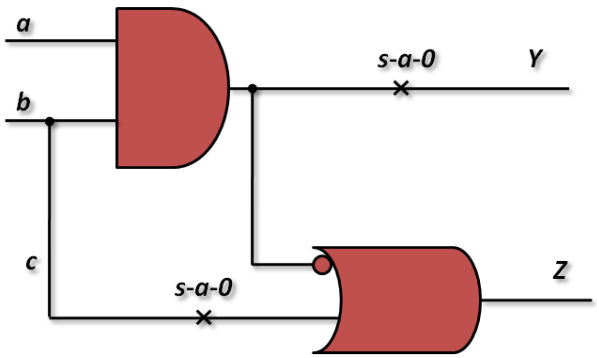


Figure 5-7. Example of non-diagnostic detection equivalence

For combinational circuits with multiple outputs, two types of equivalence can be differentiated (Sandireddy and Agrawal, 2005):

- Diagnostic equivalence: two faults are f_1 *and* f_2 are diagnostically equivalent iff the functions of the two faulty circuits are identical at each output
- Detection equivalence: two faults are f_1 *and* f_2 are detection equivalent iff all tests that detect f_1 also detect f_2, not necessarily with the same output

Figure 5-7 shows an example circuit with two single *s-a-0* faults in the input *c* and output *Y* lines. Both are detection equivalent faults but are not diagnostically equivalent.

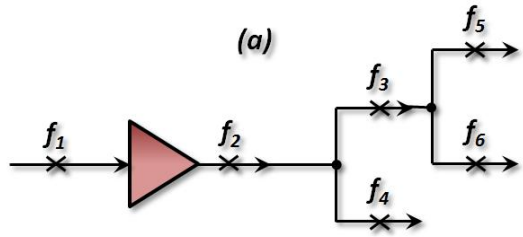
A fault f_2 *dominates* f_1 ($f_2 > f_1$) if the test set for f_1 (T_1) is a subset of the test for f_2 (T_2). All tests pattern of f_1 would detect f_2. Therefore, f_1 implies f_2 and including f_1 in the fault list would be sufficient.

If two faults dominate each other then they are equivalent:

$$f_1 \sim f_2 \quad iff \quad f_1 > f_2 \ and \ \ f_2 > f_1$$

ATPG tools generate test patterns that target possible physical faults according to the fault model (Agrawal and Chakradhar, 1995; Roy et al., 1988). An increase in the complexity of circuits involves bigger fault dictionaries and patterns, slowing down the ATPG process. The implementation of quick detection and diagnostic mechanisms can improve the effectiveness of resilience. One way of creating compact sets a is fault collapsing, which is the process of reducing the number of faults by using redundancy, equivalence and dominance relationships among faults is called fault collapsing (Abramovici and Breuer, 1979). To lessen the burden of test generation, two main types of fault collapsing are used:

- *Fault equivalence collapsing*: uses the notion of fault equivalence to remove most of the equivalent faults from the pattern. Faults of a logic circuit can be divided into N disjoint equivalence subsets S_i, where all faults within a subset are mutually equivalent. A fault set S_i is *collapsed* if it contains one fault from each equivalence subset
- *Fault dominance collapsing*: uses the notion of fault dominance to remove dominating faults from the equivalent collapsed faults. If fault f_2 dominates f_1, then f_2 is removed from the fault list



(b)

Line 1	Line 2	Equivalence	Dominance
f_1	f_2	Y	Either
f_2	f_3	N	f_2 over f_3
f_2	f_4	N	f_2 over f_4
f_3	f_5	N	f_3 over f_5
f_3	f_6	N	f_3 over f_6
f_5	f_6	N	N
f_2	f_4	N	f_2 over f_4
f_3	f_4	N	N

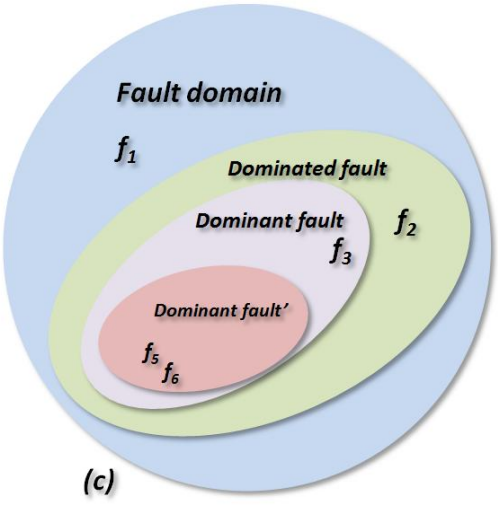


Figure 5-8. Dominance and equivalence relationships of circuit lines

Figure 5-8b shows the dominance and equivalence relationships of a given circuit (Figure 5-8a).

Both equivalence and dominance relationships are transitive. For instance, Figure 5-8c if fault f_2 dominates f_3 and f_3 dominates f_5 then f_2 will dominate f_5. Collapsing algorithms use this transitivity property to reduce the fault patterns (Prasad et al., 2002). If fault detection is the only objective (e.g. fail-safe system that do not require diagnosis), then fault dominance collapsing can be used to further reduce the fault list.

A fault *f* is *detectable*/*testable* if there is a test *T* that is able to test/detect *f*, otherwise *f* is an *undetectable/untestable* fault (Agrawal and Chakradhar, 1995). Undetectable faults can be partitioned into two subsets: partially detectable faults and redundant faults. A test set/sequence is said to be *N-detectable* if all faults are detected at least N times with N different test vectors (McCluskey and Tseng, 2000). The higher the value of N the higher is the fault coverage.

A circuit is *redundant* if the function realized by the circuit without fault(s) is the same as the function realized by the circuit with one or more faults (Carter, 1979). *Redundant faults* are undetectable faults that do not affect the circuit operation (*operationally redundant*). It can be argued why is it of interest to discover redundant faults if they do not affect circuit logical behaviour. Discovering and removing redundant faults from the tests is important for the following reasons:

- In redundant circuits, the presence of undetectable faults can invalidate certain tests, raising problems such as (Friedman, 1967):
  - If f_1 is a detectable fault and f_2 is an undetectable fault, then f can become undetectable in the presence of g. In that case, f_1 is a second-generation redundant fault (Friedman, 1967)
  - If two undetectable single faults g_1 and g_2 are simultaneously present in a system, then they can become detectable

- Due to time and power consumption constrains, it is not feasible to perform a complete search of all possible faults in any given circuit. There are certain needs for minimizing the tests patterns that detect existing faults.
- Additional current drains can be induced due to redundant faults such leakage faults (Mao et al., 1990; Xiaoqing et al., 1996) and gate oxide shorts (Hawkins et al., 2003; Segura and Hawkins, 2005) , which are especially undesirable in low-power devices.
- Redundant defects may indicate a latent reliability problem.

In the case of combinational circuits, all undetectable faults are redundant faults (Abramovici and Breuer, 1979). Testing sequential logic is significantly more difficult that testing combinational logic, whose response is a function of its initial state. In the case of sequential circuits, it has been shown that certain faults for which a test sequence does not exist, under certain conditions, faulty behaviour may be detected. These are *partially/potentially detectable faults,* faults that affect circuit operation under some states (*partially irredundant*), but are not manifested at the outputs for any input sequence under other states (therefore undetectable) (Pomeranz and Reddy, 1993).

The testability features, observability and controllability (defined in 2.5.2.2, are important to increase the effectiveness of fault detection. Therefore observability and controllability would increase the number of testable and tested faults and so the fault coverage. Conversely, lack of testability would increase the number of untestable and untested faults.

In terms of diagnosis, different approaches can be followed: *offline diagnosis* based on fault dictionaries, effect-cause diagnosis also called *online* or *dynamic diagnosis* based approaches, or a combination of these two approaches (Smith, 1997).

Finally, as in Table 5-3, faults can also be classified according to their diagnosability, containability and recoverability.

5.5. FT modelling

When system and fault modelling are developed together, the system behaviour in presence of faults and the control process of FT can be considered at the earliest stages of design. This enables us to take into account mutual dependencies of solutions for reliability, performance and power consumption making the resulting system more efficient and resilient.

When a fault appears, extra redundancy is needed to cope with it. Such redundancy and the ability to use it are key in the implementation of reconfigurability.

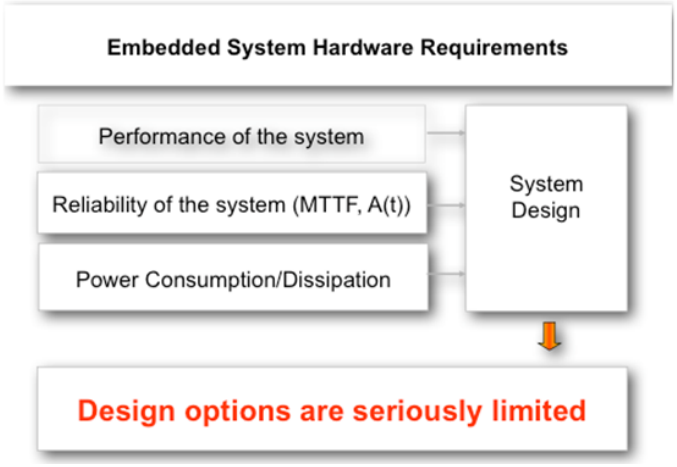


Figure 5-9. Performance, reliability and power concerns on the design of embedded systems

At the same time, redundancy can be used for various purposes and can be involved as an essential part of reconfiguration. System reconfiguration purposes are:

- performance improvement,
- reliability enforcement,
- energy –wise use

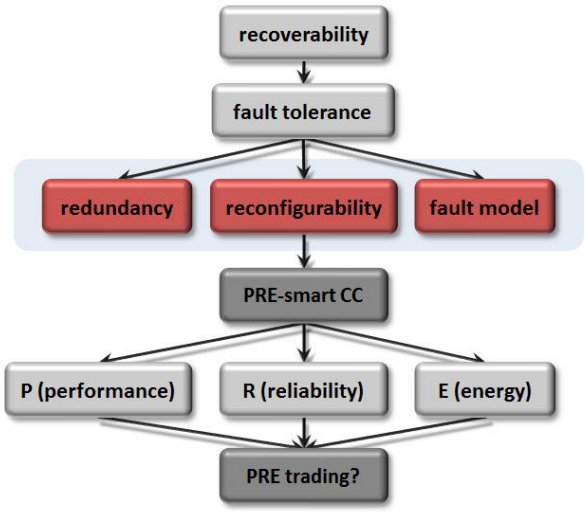


Figure 5-10. Reconfiguration purposes for fault tolerance

Computers developed using this approach have been named PRE-smart computers (Schagaev, 2009; Schagaev et al., 2010). A inheritance of properties is shown on Figure 5-10. Thus PRE (performance-, reliability- and energy-) wise systems might be designed rigorously, using reconfigurability and recoverability as system features introduced at conceptual level. Success of PRE designs use of this approach for connected computers (networking, clusters multiprocessing) depends on careful trading of the redundancy introduced to achieve the required property.

5.5.1. Trading P, R, E

Describing a system in terms of Structure, Information and Time as parameters of redundancy, as well as other properties, (Schagaev, 2001) can help its reliability estimation. Redundancy might be weighted, say, in units or values, with or without relation to the steps of any supportive algorithm, that applies it to form and control configuration. While time and information units are clear (seconds and bits), structure units require some extra efforts. Note also that time, information and structure are considered as independent variables. Structural redundancy for our purposes might be measured using a graph notation:

$dS: < dV, dE >$

where dS denotes introduced structural redundancy, while dV and dE denote extra vertices and edges added in the structure to represent a step of an algorithm.

Our efforts towards a PRE goal can be measured quantitatively with a redundancy vector:

$dR = < dT, dS, dI >$

Time, information and structure as mentioned above are considered as independent variables.

Furthermore, reconfigurability of the system can be used for various purposes (Figure 5-9). To be able to use redundancy and apply reconfigurability we need to consider the introduction of a supportive tool for reconfigurability, a syndrome. The syndrome provides a snapshot of the system state, namely a real-time status of every element of the system in terms of reliability, performance and power awareness.

Lets analyse how a generalized algorithm of fault tolerance might be implemented using mentioned redundancy types and reconfiguration introduced as a system property.

As Figure 5-9 shows the just mentioned mutually contradictive requirements of performance, power consumption and reliability limit the design choice of embedded systems. These constrains become even more serious when the system is implemented on a chip (SoC). In this case reliability, performance and power consumption of the elements are defined and limited by the same technology.

With regards to reliability, the available redundancy hints a potential solution: it is known that $max\ (P_i,P_j,P_k,P_l)$ is achieved when all mentioned probabilities of

independent elements are equal. For example, for SoC it technologically means that we need to introduce in the system different redundancy levels for each mentioned group of elements to equalize group reliabilities.

5.5.2. GAFT: Generalized algorithm of fault tolerance syndrome support

It is well known that fault tolerance of a computer system can be achieved by introducing static redundancy in hardware and system software (HW/SSW). It is also well known that using traditional approaches of fault tolerance (Anderson and Lee, 1981, p. 81; Avizienis, 1971; DeAngelis and Lauro, 1976) is expensive in terms of time, information or hardware overheads. To avoid this, the authors (Schagaev, 1987, 1986; Sogomonian and Schagaev, 1988) proposed to consider fault tolerance not only as a feature but also as a process that can be implemented algorithmically.

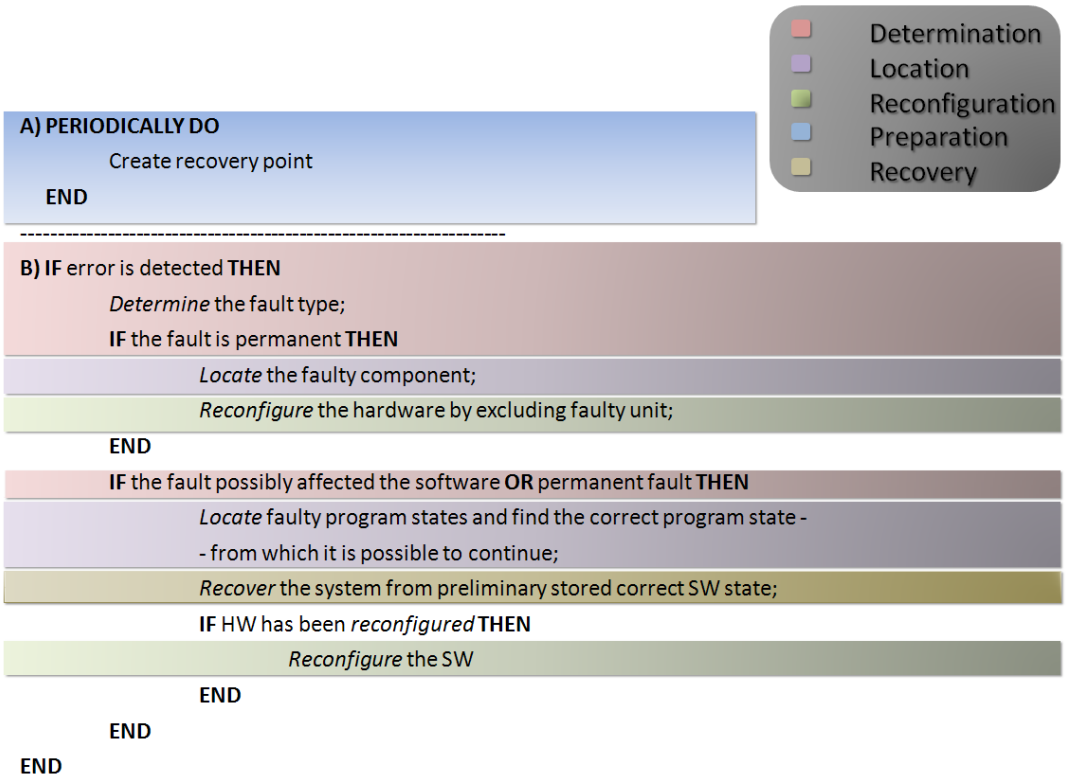


Figure 5-11. GAFT: Generalized algorithm of fault tolerance

A three-step algorithm (Sogomonian and Schagaev, 1988) has been further developed. The outcome of this work is the Generalized Algorithm of Fault Tolerance (GAFT), a five-step fault-handling algorithm (Figure 5-11):

- Detecting faults
- Identifying faults
- Identifying faulty components
- Hardware reconfiguration to achieve a repairable state
- Recovery of a correct state(s) for the system and user SW

The different types of redundancy (information, time and structural either hardware- or software-based) can be used to implement every step of GAFT.

The more complex the system implementation is, the more complex Fault detection and diagnosis will be. This is particularly true for multicore systems (MIMD processors that support vector instructions (SIMD) and pipelined implementations, which are particularly complex. In principle the complexity of GAFT implementations also depends on the complexity of the system, its faults and fault tolerance models.

Hardware support is a faster mechanism than software support to achieve reliability and with a proper design there should be little performance degradation. However, the introduction of static redundancy in HW might be prohibitively expensive in terms of cost and power consumption. Therefore, a process that implements fault tolerance assuming dynamic interaction of existing redundancy types between elements can tackle these problems.

GAFT might be used for comparison and overview of different design solutions of FT systems. It also allows controlling fault coverage at every step of system design, providing a tool to select efficient solutions and estimate overheads.

A substantial redundancy can cover multiple steps in GAFT, such as fault detection and fault recovery; take, for example, TMR systems, when a faulty channel output is overruled by two correct outputs.

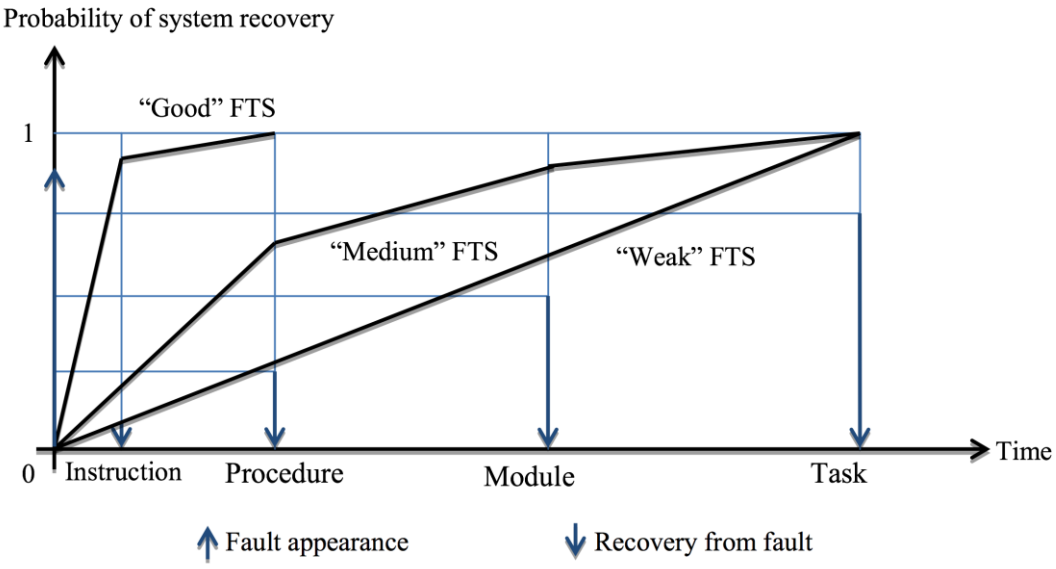


Figure 5-12. System recovery time according the level of implementation of checking and recovery schemes

The implementation of the hardware checking and recovery (step A of GAFT) at different levels causes different timing for GAFT completion (Figure 5-12): microseconds for the instruction level, milliseconds for the procedure level, hundreds of milliseconds for the module level, seconds to tens of seconds at the task level and tens of seconds to minutes or even hours at the system level.

Different implementations would have different properties in terms of timing, fault coverage, types of faults that can be tolerated, power consumption, complexity and cost. Therefore in order to achieve the required specifications, it is wise to combine different checking and recovery schemes in one system.

For example, it might be beneficial to protect the processor and the memory by using hardware schemes at the level of instruction (duplicated processors, triplicate memory) and use higher-level schemes (procedure or module) for the other hardware components due to cost and power constraints. Processor and memory is used at every instruction execution. The implementation levels are

not mutually exclusive; for example the combination of hardware- and software-based checking can significantly improve fault coverage. In general, the higher the implementation level the less hardware support is required, but with higher timing and software coding overhead.

A good fault tolerant system tolerates the vast majority of transient faults within the interval of instruction execution, making them invisible for other instructions (and software). At the same time, our assumptions about transient "live" fault is arbitrary, thus transient faults with longer time range or permanent faults might be detected and recovered differently, for example, at the procedural or task level of system software.

Taking into account that transient faults occur at an order of magnitude more often than permanent faults, transient fault tolerance must be done extremely effective.

In turn, it is necessary to implement special schemes for HW reconfigurability and recoverability to eliminate the impact of permanent faults on the system. There are two types of reconfiguration in GAFT. First a hardware reconfiguration, that implies: 1) temporary isolation and substitution of the suspected/faulty unit for a healthy one, if possible; or 2) replacement of the faulty unit. The SW reconfiguration involves correction of data and/or code in order to restore the system to a working state.

GAFT completion requires three fundamental processes, called in literature P1, P2 and P3 [Stepanyants01]. The error *checking process* P1 is responsible for checking the system state in terms of hardware fault existence/appearance. The second process, the *error recovery process* P2 prepares recovery states when it is scheduled by the system. When a transient fault is detected, its toleration assumes recovering the information that has been modified and restoring hardware states. The third one, the *functional process* P3 is the process of calculation or instruction execution. Let's assume the primary function of the real time critical system as "process three" or P3. If the system ensures full

functionality and transparent application recovery for the process P3 (from a predefined set of faults in a given time frame) then the system is fault tolerant. That is, *we define a system as fault tolerant if and only if it implements GAFT transparently for applications*.

5.5.3. System estates and actions to implement fault tolerance

GAFT above presents an approach for achieving a new and complex property (fault tolerance) considering it as a process with several steps and phases. Further generalization and detailed analysis of system state change is presented below. At any given time, a single processing element (SPE) system can be in one and one only of five possible states: ideal, faulty, erroneous, degraded and failed. Figure 5-13 shows the five S states, the potential T transitions and the M mechanisms involved in fault tolerance. Two different areas can be differentiated: the green area at the bottom half represents the conventional environment with no FT capabilities whereas the red area at the top half represents the possible states and transitions in a dependable environment.

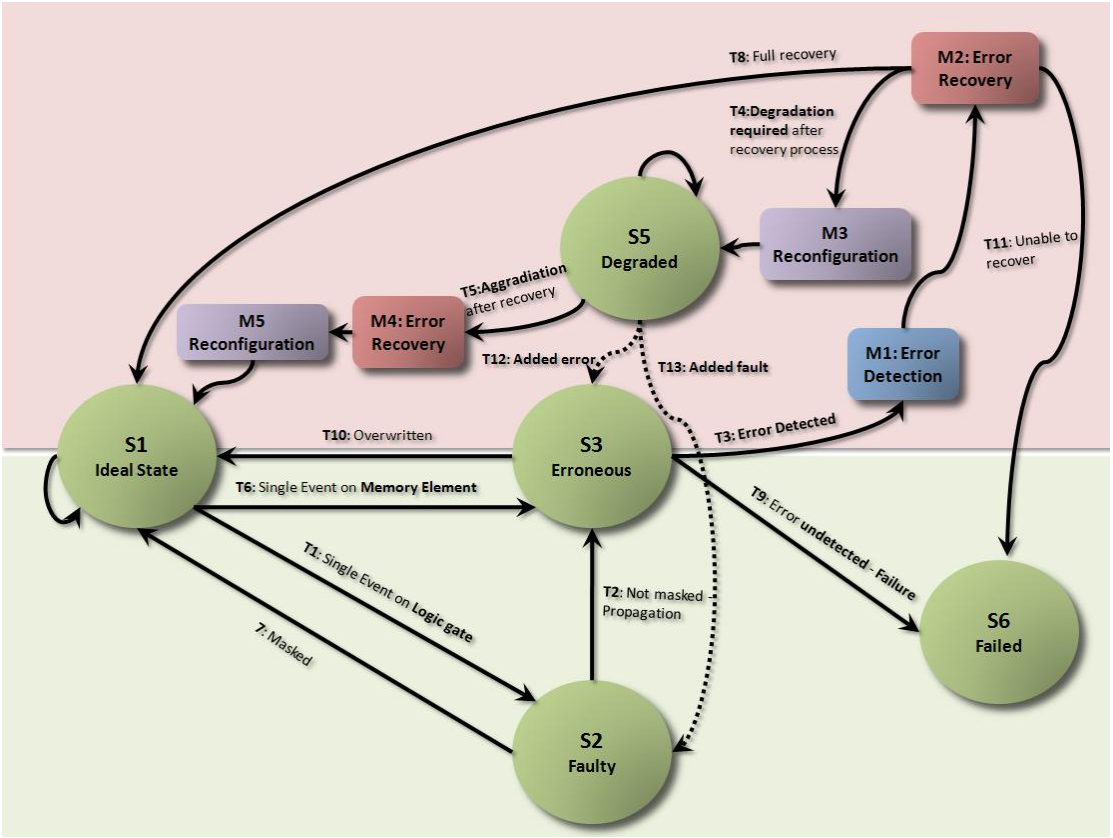


Figure 5-13. System states sequence of actions for FT

Considering S1 as the initial state, a single event can change this ideal state. Note that with the exception of the transition T12, Figure 5-13 does not contemplate the cases where a single event can occur between transitions. If the deviation in the form of voltage transient introduced by such event affects combinational logic, the system would turn (Transition 1) into a faulty state S2. The voltage transient may propagate to sequential logic such a memory cell or latch, potentially flipping the bit, contaminating the data flowing within the system, and leading the system to an erroneous state S3 (soft error in Figure 5-13).

However, there are three masking effects that can prevent transition for this particular event from S2 to S3: logical masking, electrical masking and latch-window masking. Logical masking happens when one of the other inputs of the affected gate is in controlling state so that the output does not vary (e.g., 1 for an AND gate, or 0 for a NAND gate). Electrical masking happens when the voltage transient impacts successive logic gates and propagates through the logic chain fading out before reaching the registered output. Latch-window masking happens when the arrival of the pulse is outside the latching window, usually based on the setup time and hold time of the sequential logic.

Nevertheless, if during the ideal state S1, as a result of the single event the voltage transient affects the sequential logic directly, the system state would transit straightforward to an erroneous state S3 (T6).

The implementation and the coverage of faults within the system can be measured probabilistically, assuming existence of undetected faults. We consider that an undetected fault would lead (T9) to failure in the system (S6), unless the error is overwritten [(e.g.: a memory bit that has flipped can potentially be flipped back to the original value by another event before the fault detection mechanisms were activated or before the error leads to a failure) transiting back to an ideal state. The probability of overwritten errors is very small.

The implemented action of fault detection and recovery mechanisms differs in terms of permanent and transient faults. Faults are initially detected by the by

the fault detection mechanisms (M1 in Figure 5-13). A detected fault that is not recoverable by the recovery mechanisms (M2) would lead to a failure (T11). In most cases, recovery will be possible in two forms: full recovery (T8) and graceful degradation (*T4*). Ideally, a full recovery would turn the system back to the initial state *S1* or, if full recovery is not possible, SSW will make use of the reconfiguration mechanisms (*M3*) to turn into gracefully degraded state S5. In this state the system can continue operating properly. In some cases, depending on the severity of the failure, the operating quality may decrease. This becomes more obvious if a further fault or error takes place. Further graceful degradation may be possible depending on the levels of degradation introduced in the implementation. A good example of degradation support for memory can be found in (Bernstein et al., 1993, 1992).

Recovery from a degraded state takes place once the deviation has been corrected. The recovery mechanisms should be able to return the system back to correct state using additional reconfiguration (M5).

It is clear that a logic framework with holistic principles to follow might help to design and develop an efficient resilient architecture with required properties. These principles are explained in the following chapter.

5.6. Conclusion

In this chapter we have introduced a detailed fault model and analysed it together with generalised methods and models of fault toleration. We have extended the model of faults defined by (Avizienis et al., 2004) and provided a classification suggesting methods for recognition and reaction. Manifestation, detectability diagnosability and recoverability are discussed as one consistent flow proposing adequate solutions for diagnosis and recovery. Note that the proposed fault model is generic and has not been customized for the latest version of the architecture proposed in the following chapter. In fact, the earlier version of the architecture was developed before this fault model. However,

We have introduced the principle of reconfiguration of the system and how this might be used for various purposes - performance, reliability and energy wise gain, improving the efficiency of resilience.

Using PRE- properties, a generalised algorithm of fault tolerance is developed further with a full explanation of system state changes and the actions required to implement GAFT. In contrast with the fault model, the system estate sequence of actions in Figure 5-13 can be directly applied to the newer version of the architecture. In this figure the detection, recovery and reconfiguration mechanisms correspond to the hardware and system software mechanisms available in the ERA architecture.

Chapter 6

Chapter 6

Hardware Support for Resilience

Any new complex property of the system is studied and analysed as a process. This chapter describes how by introducing a) reconfiguration properties and processes into a system to adjust the system for reliability-, performance- and power- wise applications we can extract new structural elements. We present a necessary and detailed argumentation of the syndrome concept, its structural design, possible ways of implementation and its applications. The syndrome properties, its structure and its operations are described using a prototype of a resilient architecture as an example. Detailed explanations of the use and implementation of the syndrome is given within the context of GAFT.

We present a hybrid HW-SSW co-design approach of a resilient architecture with the ability to reconfigure, achieving various levels of dependability in different environments. The system is capable of increasing or decreasing the level of reliability and power consumption by changing the active and passive redundancy via reconfiguration.

In today's MBU scenario, the introduced architecture has advantages to known architectures with a better compromise in area, performance, reliability, power, cost and efficiency.

6.1. ERA concept, system design and hardware elements

The development of this new architecture follows the holistic principles proposed by the ERA paradigm (Schagaev, I., 2010): simplicity, redundancy, reconfigurability, scalability, reliability and resource awareness. To support those principles a new hardware architecture (HW) and system software (SSW) have been developed. Brief introduction of this principles are:

Simplicity: Complexity is difficult to implement and handle efficiently. In addition, big complex systems are more prone to faults, thus lowering reliability.

Reliability: The highest reliability of individual components is preferable but always keeping in mind the cost-efficiency of its implementation.

Redundancy: Deliberate introduction of hardware and software redundancy provides the required level of reconfigurability to reach performance and reliability goals.

Reconfigurability: Apart from the simplicity, reliability and deliberate introduction of redundancy, it is essential to achieve balance between performance, reliability and power. Reconfigurability serves three main purposes: performance, reliability and power awareness. It allows the system to adapt: 1st by recovering from a permanent fault and 2nd by adjusting the requirements of the running application.

Scalability: Scalability should be kept in mind when designing a system so that it can be extended if the requirements change.

Power-awareness: Mission critical systems have significant limitations of hardware resources and power consumptions constraints (e.g. battery life). Thus, for wise resource use, reconfigurability must be introduced.

By following these principles the processes of design and development of a new architecture can be defined.

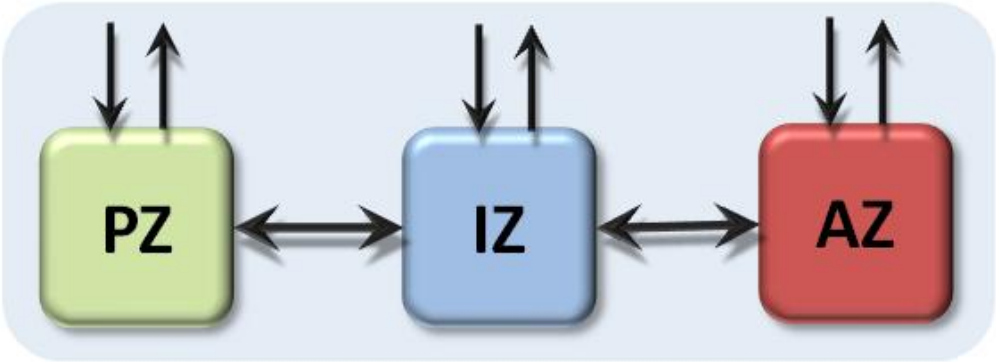


Figure 6-1. System zones from a information processing point of view

Figure 6-1 shows a computer system as three semantically different (from the point of view of information processing/transformation) elements. The principles mentioned above might be applied at the level of each element, which will help into designing a more efficient computer system.

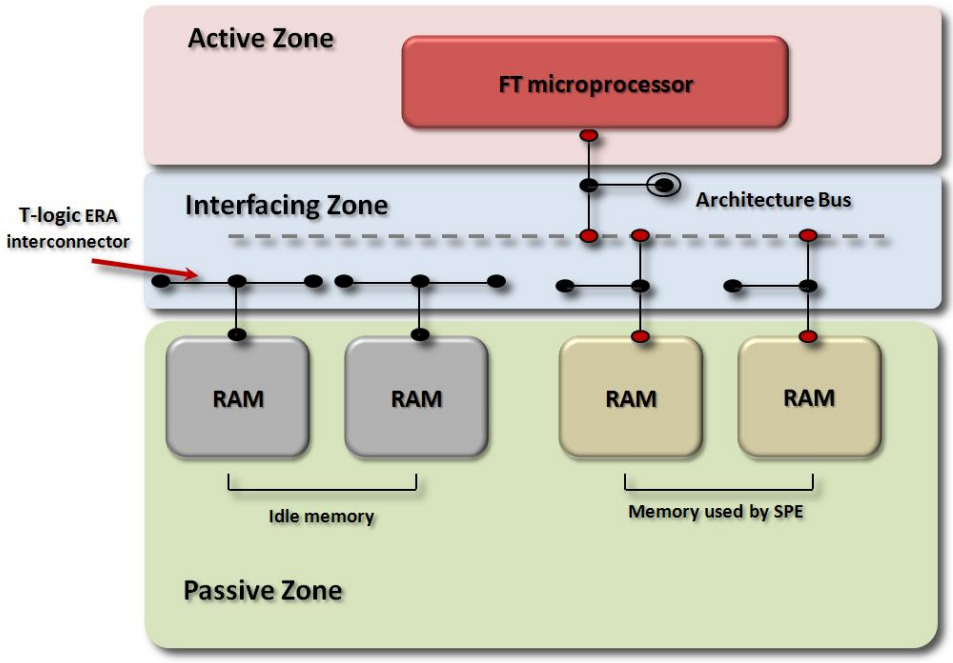


Figure 6-2. Information processing in ERA

In terms of information processing the hardware is based on a Single Processing Element (*SPE*) that is divided into three areas (Figure 6-1): first, the information transformation area – further called active zone (*AZ*) and second the information storage area – called passive zone (*PZ*). The interconnection of these zones is the interfacing zone (*IZ*).

All three zones must have different properties and will use different redundancy mechanisms to tolerate faults and to make system reconfigurability possible and efficient. The proposed architecture structure of each zone is shown in Figure 6-2.

Active Zone: The active zone consists of the microprocessor elements including the arithmetic unit and logic unit. Both units are separated for better fault isolation and easier implementation of hardware tests.

Interfacing Zone: This includes all communication components such as the memory buses and the reconfiguration logic. A configurable bus allows the reconfiguration of the hardware to exclude failed hardware components and go into a degraded state, or to replace the failed component with a working one.

Passive Zone: This includes basic storage systems, such as memory, that do not act by themselves but are handled by controllers or devices.

Minimum deliberate redundancy has been introduced in the form of buffer, register files, replicated memory modules, majority schemes (in terms of HW) and interfacing logic. With regards to SSW, some extra elements required to support fault tolerance are: checkpoint monitor, recovery point monitor, process synchronization and reconfiguration monitor. These are named monitors to express their uninterruptable mode of operation.

As mentioned above, hardware can be considered as three zones, (see Figure 6-1 and Figure 6-2). All elements in these zones have to be reconfigurable for their

own purposes as well as other zones requests. All zones might have different reconfiguration properties.

Reconfiguration might have internal and external reasons. For example, when the system forms a configuration for a task execution it might deliberately and *externally* exclude some hardware elements from configuration due to a transient/permanent fault. On the other hand, checking schemes can enable the reconfiguration of hardware elements for reconfiguration due to task requirements (internal reasons).

Interactions between zones define the level of reconfigurability and flexibility of the architecture. These new hardware reconfiguration abilities must be reflected and supported as new features of the architecture.

6.2. ERA hardware configuration

ERA is based on a first fault tolerant version of the system (ERRIC) resulting from the ONBASS project (ONBASS Consortium, 2004) that has been improved with the improvement of the reconfigurable memory and the addition of a syndrome hardware structure and a memory management unit by the author. With regards to system software, ERA has also benefited from the contribution of (Kaegi-Trachsel and Gutknecht, 2008).

6.2.1. Active Zone

The main principle used in the design of the processor is simplicity. The cost of P1 and P2 implementation of GAFT depends on the structure of the processor and might become prohibitively high. Antola (Antola et al., 1986, p. 1) proved that the overheads necessary to make a CPU fully fault tolerant might easily exceed 100%. Thus, to avoid duplication, there is a requirement to keep the redundancy level needed to implement fault tolerance, as low as possible.

Following the simplicity principle, the instruction set and its implementation within the processor is reduced to the absolute minimum required to support general purpose computing. This allows the careful introduction of redundancy that implements error detection, diagnosis and recovery features. Complicated memory addressing instructions are omitted, as they are not essential.

The instruction set architecture (ISA) consists of only 16 instructions with only two of them for memory access. Such a simplified instruction set generally requires less hardware (the control unit in particular), which increases, by design, the reliability of a single processor. Performance might also be increased as operating clock frequencies can be improved. Extra details on the instruction architecture are explained further in 7.2.

The proposed microprocessor offers clear advantages in comparison to CISC architectures: fewer and simple addressing modes, hardwired design (no microcode), fixed and simple instruction formats. In turn, a simple instruction format allows fetching of two 16-bit instructions per machine cycle. The execution steps are similar to other RISC processors. There is no pipelining and all steps of one single instruction are executed within one memory cycle. Pipelines are one of the most vulnerable elements of modern microprocessors. The relative amount of area of the chip dedicated to pipelines is increasing with scaling design complexities. For instance, instructions can stall in the instruction queue and the longer they reside there, the higher the chances of getting struck by an energetic particle. A transient fault in a latch or a memory cell within the pipeline (e.g. SET or SEU) can propagate and become an error at the microarchitectural level (e.g. Register file or Instruction Register). Consequently, the effects of ionizing radiation in this area can lead to SEFI's, severely decreasing the overall reliability.

The absence of pipelining and caches greatly simplifies the processor design, which, in turn, simplifies the implementation of fault tolerance. A careful introduction of redundancy for checking and recovery allows the processor:

- To detect transient faults during the execution of the instruction
- To abort the current instruction and
- To re-execute it, all transparently to software

There are known arguments that simple ISA's do not have enough instructions to perform most of the application operations. These argue that a small ISA would result in higher compilation effort/time, and that the resultant programs would be bigger, which would increase the amount of memory needed.

However, complex functions are well handled by the compiler instead of having specialized instructions within the ISA dedicated to very particular tasks.

With regards to the size of the programs it is worth mentioning that:

- A bigger ISA such as CISC involves longer operation codes, which in turn, increase the size of programs. Although the program size applications compiled with the proposed microprocessor are still larger than the same applications compiled in other architectures. This is mainly due to the lack of certain instructions such the multiplication instruction, as the corresponding library calls would need additional setup code.
- In an architecture based on a smaller ISA, register references require fewer bits
- It is usually claimed that smaller ISA requires more memory. ERA architecture has unique design property: size of all instructions is 16bit while size of a word is 32. Thus, the instruction density per word is 2 and the previous arguments above about bigger code size for RISC do not stand to ERA
- Since the price of the memory is very low and keeps decreasing this argument also becomes less important

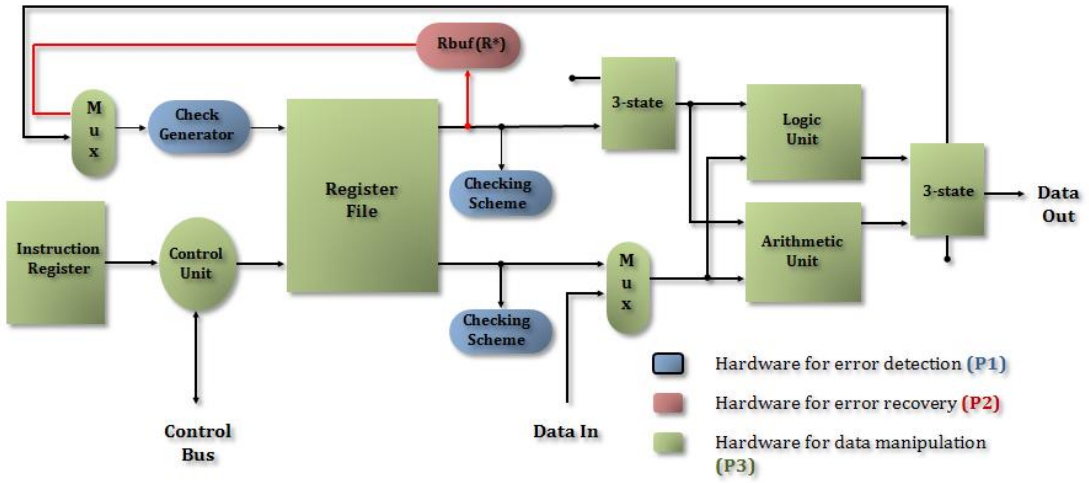


Figure 6-3. Architecture of the active zone of ERA

The processor has a large register file with 32 general purpose registers with a width of 32 bits and no restrictions on their use, which simplifies software development (see Figure 6-3). All standard instructions expect exactly two arbitrary registers as input, and save the result of the operation in one of these two registers, thus overwriting one of the input values.

Memory access is currently possible for 32 bit words at a time, and it has to be aligned. The main structure of the processor architecture is illustrated in Figure 6-3. The instructions are fetched from memory into the instruction register and are decoded by the Control Unit, which also manages the execution of each instruction. Operands for each instruction being executed are fetched from either the register file or memory multiplexed into the Arithmetic or Logic Units (AU and LU respectively).

The output data from AU or LU goes either to memory through the data bus or is written back to the Register File. The current value and type of the data might also indicate an address for branch instructions.

The hardware for fault detection and error recovery process, P1 and P2, are marked in blue and red respectively (Figure 6-3). The hardware for the data manipulation process P3 is marked green.

The simplified architecture presented in Figure 6-3 allows the implementation of GAFT at instruction level with reasonably small reliability (13%) (Schagaev, 2008). As before, the three processes are essential for the guaranteed and successful execution of each instruction.

Two processes P1 and P2 cope with fault checking and recovering of transient errors respectively. P1 is initialized at the start of the instruction execution; P2 is required and initiated when fault has been detected, but it is essential that the pre-modified state is stored at the start of execution of every instruction. P1 and P3 can operate concurrently. P1 and P2 have an influence on each other: the higher the fault detection coverage achieved by P1, the more successful recovery should be (Stepanyants, 2001).

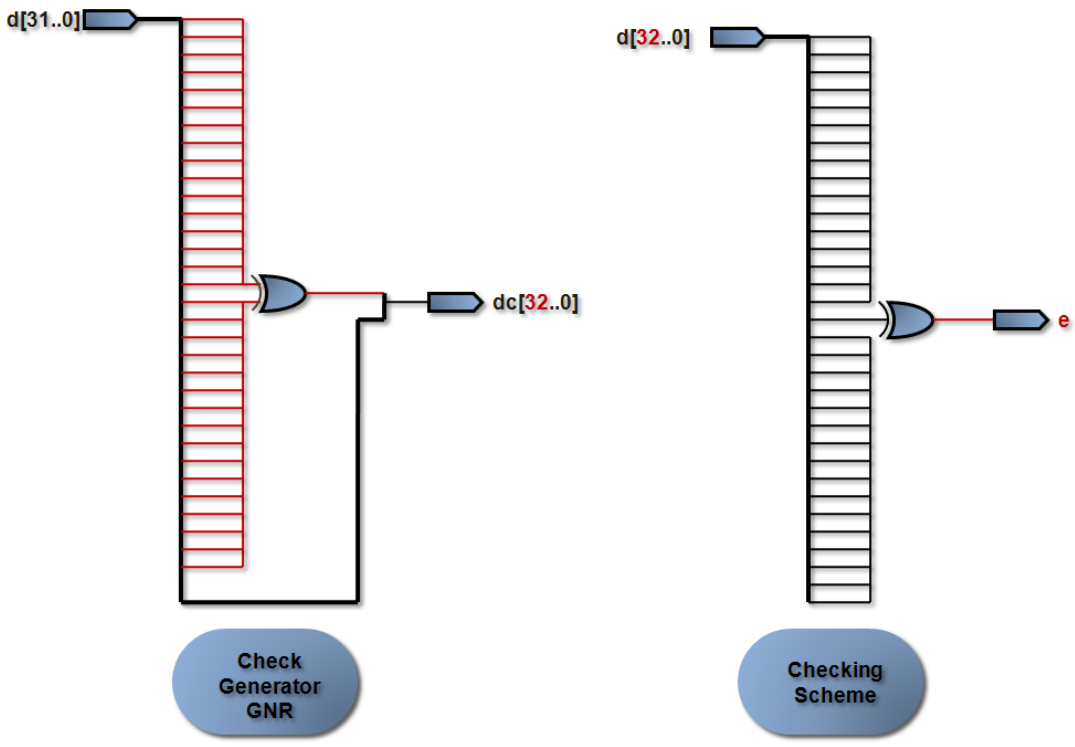


Figure 6-4.Check Generator and Checking Schemes

When data is written into the Register File, the Check Generator (marked in blue in Figure 6-3 and Figure 6-4) generates the checking information for the 32 bits data storing into the register file; this information allows the stored data to be verified later on. The Checking schemes (marked in blue in Figure 6-3 and Figure 6-4) check the data integrity when data are read out from the Register File and when possible, correct the data before the ALU operation takes place.

In order to implement the fault recovery process P2 an extra Register Buffer *Rbuf(R*)* is introduced (marked in red in Figure 6-3). The register buffer is allocated to keep the pre-modified state of operand for the currently executed instruction.

When a fault is detected during instruction execution, it allows the processor to restore to the initial state before the execution of the instruction, enabling the instruction to be repeated. This allows the system to tolerate faults within instruction execution.

The extra Register *Rbuf(R*)*, the checking schemes and the reverse instruction sequencer combined make the implementation of P1 and P2 possible without any perceptible time overheads (13%).

6.2.2. Passive Zone

The proposed memory scheme may be regarded as a collection of 4 blocks or RAM, 16-bit wide (in the case of ERA the blocks have an identical size of 1Mb each: 16x64k). Using 16-bit memory modules instead of 32-bit memory modules increases reliability and reduces, when necessary, the energy required for execution. Additionally, the scheme includes two flash-based ROM modules with replicated bootstrapping firmware and operating system software.

Reliability is increased by means of added working states and configurations that are explained in 6.4.3 Energy-wise operation is improved by means of this architecture ability to activate only certain modules when required (e.g. using a single 16-bit memory module).

The proposed memory scheme allows different configuration schemes that will be explained in 6.4.3. and 6.5.

6.2.3. Interfacing zone

The principles of architecture design explained above relate to the interfacing zone as well. The architecture needs something (logic) in the interfacing area to enhance the flexibility and resilience of system operation between active and passive zones, for wide range of applications where PRE-properties are key requirements. This motivates the ability to have a reconfigurable interfacing zone.

One of the schemes that we propose is known as T-logic. T-logic is a hardware element that provides reconfigurability of the passive zone for performance, reliability and energy-wise operation. This logic should be able to provide minimal configuration – when one processor and memory remain active. T-logic is further explained in the following section.

6.3. ERA reconfigurability

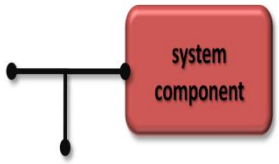
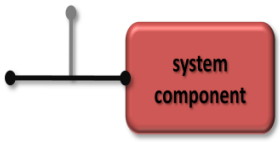
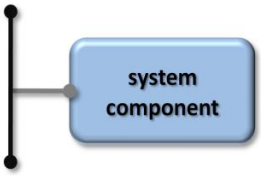
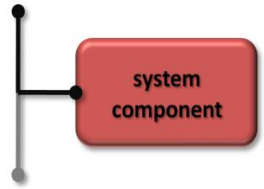
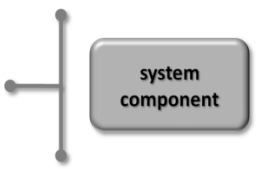
As declared earlier performance, reliability and energy-awareness are required for the next generation of computer systems. Reconfigurability requires hardware and system software implementation and support. In order to be able to change the configuration when necessary (sometimes several times during a single mission) systems should have special elements with specific properties such as extreme reliability, performance and simplicity, supported by independence from faults of the system.

For the purposes declared above we propose a hardware element called T-logic. The main function of a T-element is to connect and disconnect the system component, using "logic rotation" for various types of connections and configurations. Note, that we use the term "logic" since a physical movement or rotation is not feasible and it is clear that any mechanical device that involves

movement would therefore be less reliable. The physical element that provides the functionality for this T-Logic concept is the Memory management unit defined in 6.6.1 and 6.6.2.

T- elements are controlled by hardware through syndrome schemes. Both types of operation, during run time and during diagnostics, are assumed.

Table 6-1. T-LOGIC rotation

position	Description
	The "T" logic connects the active zone to a front element that is connected in redundant mode with a right element. This front element leads the rest of the elements that is connected to.
	The "T-logic" connects the active zone to a front element. In this case the element is working in serial mode.
	The "T-logic" connects the element to two neighbouring elements in redundant mode. The element is leaded by the left side component that is connected to the active zone.
	The "T-logic" connects the element to a left element. system component will be leaded by side element that is connected to the active zone.
	Disconnected the element: The "T-logic" is disconnected from the interconnection scheme. The energy consumption of the element is reduced to the minimum.

In order to connect and disconnect the system component there are several possible "rotations" of a T-element. Table 6-1 describes the basic T-Logic element and shows some of the possible states of a T-element.

The reconfigurable interconnection schemes can execute dynamic reconfiguration of a system, transparently from software. For example, when triple system element configuration (TMR) is used, T-elements might exclude faulty ones from operation, leaving only 2 elements active (DMR). If further degradation, in terms of fault tolerance, takes place, the system can continue operating with up to a single element (Schagaev and Buhanova, 2001).

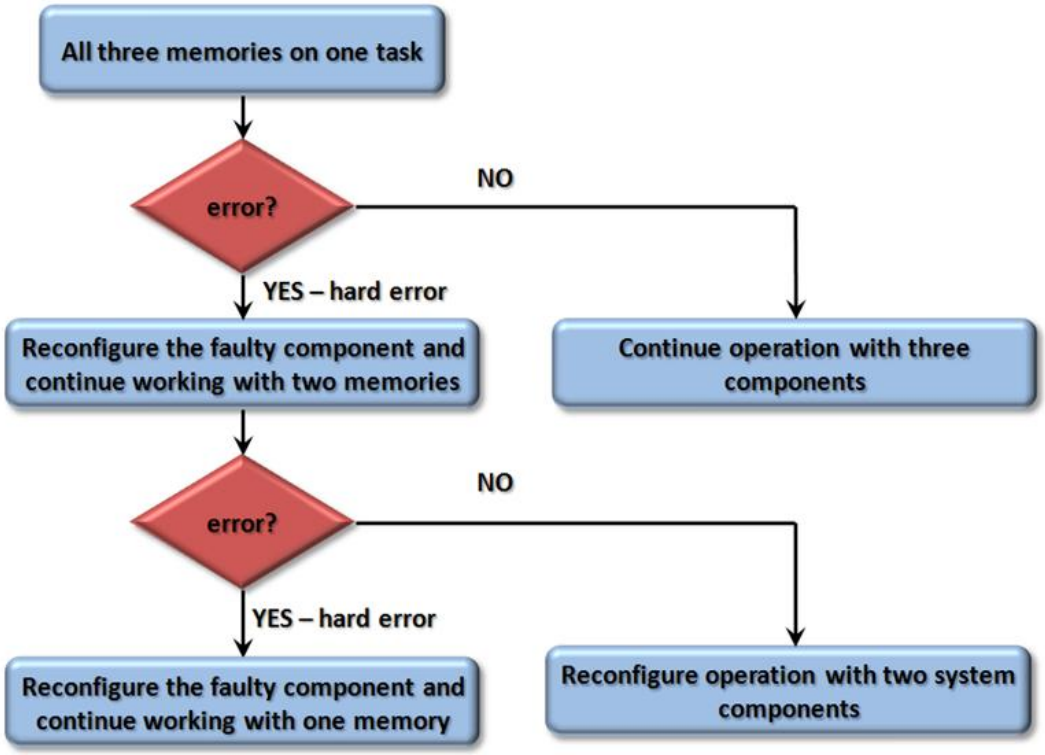


Figure 6-5. Algorithm of reliability configuration using T-LOGIC

Figure 6-5 illustrates how reconfiguration might be used for reliability purposes using T-elements. The figure shows a diagram that exemplifies a scheme with 3 memory modules working in parallel that at some point during operation experiences two permanent failures in two out of three modules.

6.3.1. T logic for memory management

ERA power consumption can also be controlled using T-elements. Existing electronic technologies posses the following drawback: increase in power consumption causes degradation of system reliability. Therefore, an ability to connect and disconnect a system element might be function and requirement in real time and other applications.

Again, for illustrative purpose we use an example when ERA uses three system elements in redundant mode. If the task scheduled does not require a full size configuration, the architecture can be configured to operate using either two elements or a single element on its own.

Hardware configuration should be implemented transparently to application programs. Task defined requirements might be available for a run-time system. Excluding memories will not change the logic of the program.

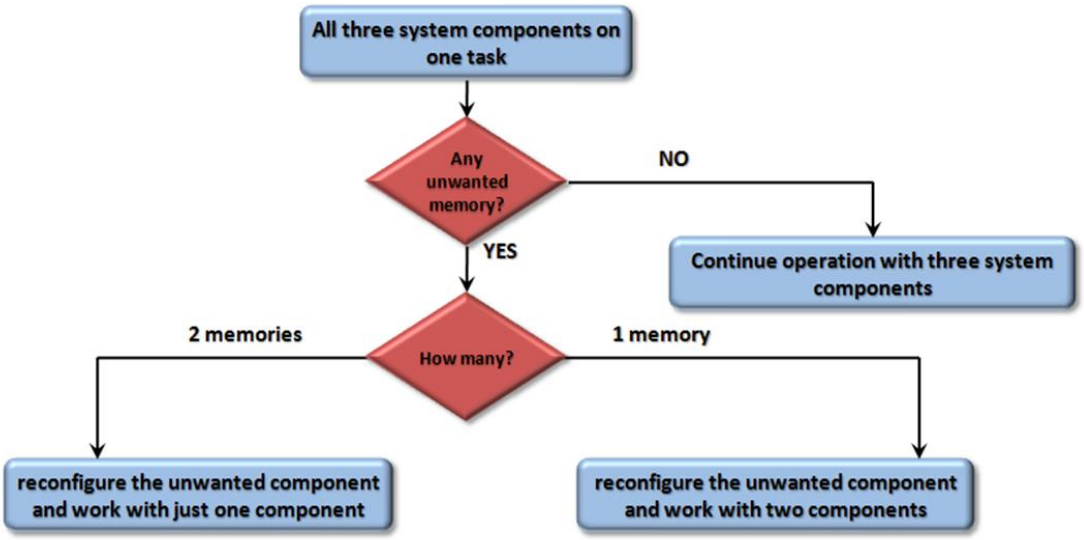


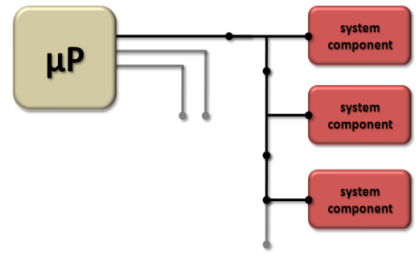
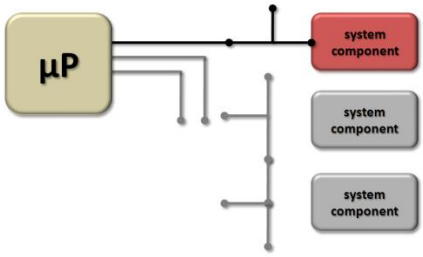
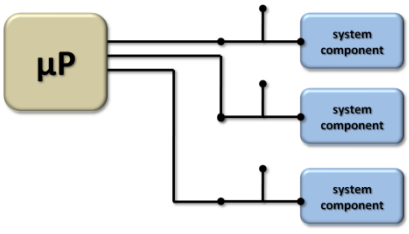
Figure 6-6. Energy-wise algorithm of configuration using T-LOGIC

Figure 6-6 shows the power saving algorithm for a reconfigurable system. The reconfigurable interconnection schemes improve the memory management flexibility. For instance, if the computer architecture has three elements and the task requires maximum capacity, the system will configure all memories in redundant mode so that it can provide maximum capacity.

6.3.2. T-Logic support of configurations in ERA

Possible configurations of a system that uses "T" logic are presented in the Table 6-2 below.

Table 6-2. Possible system configurations using T-LOGIC

configuration	explanation
	"T" configurators connect all three components with the processor. Top system component acts as leading element. The rest system elements compare the results and participate in voting. Thus reliability of this system configuration is high.
	This system configuration serves for maximum energy saving. In this case "T" element connects only one system component with processor, while the rest are idle.
	In this case, all three components are used for maximum hardware capacity. When performance of application is the main priority this configuration fits the purpose.

The first row of the table illustrates a configuration with maximum reliability (three HW elements are available). The power saving of the system could be improved by disconnecting elements from the system and keep them idle (second row). The configuration for maximum capacity required by a task is shown in the third row.

6.4. Syndrome

As mentioned earlier, the new property of the system must be supported by hardware and by SSW implementation of the required processes that define such property. We introduce a special hardware scheme called *syndrome*. The term *syndrome* is new Latin and was originated from the Greek "*syndrome"* where:

- *"Syn-"*, from combination, concurrence
- *"-Dramein"*, base meaning "to run"

For our purposes a *syndrome* is *"a group of related or coincident things, events, actions, signs and symptoms that characterize a particular abnormal condition"*.

Further analysis and development of syndrome concept and application follows.

6.4.1. Syndrome use

The functions of the syndrome are not only passive, presenting a "snapshot-status" of the system, but also active, serving as a tool to control the system configuration and to estimate system conditions. The syndrome can help to answer the question:

WHAT PROVIDES THE FAULT TOLERANCE OF THE SYSTEM? ! ! ! Q1

It is usually assumed that the core logic is ultra reliable and guarantees control of configuration and reconfiguration. Unfortunately, using the homogeneous redundancy may limit the increase in reliability since techniques based on the same type of redundancy are vulnerable to the same threats. Hybrid techniques based on heterogeneous redundancy of components and techniques can be more effective.

Thus even when checking schemes of memory or logic detect faults and reflect this situation in the syndrome, the information may not be useful if the system does not include as well:

- External elements responsible for exercising GAFT and making decision on configuration / reconfiguration if necessary
- Internal elements capable of initiating the required sequence.

Indeed, in a regular computing system when there are faults in the processing logic, to expect that the processor is able to perform self-healing and then control and monitor configuration of the rest of the system is unrealistic. We propose a possible solution as described below.

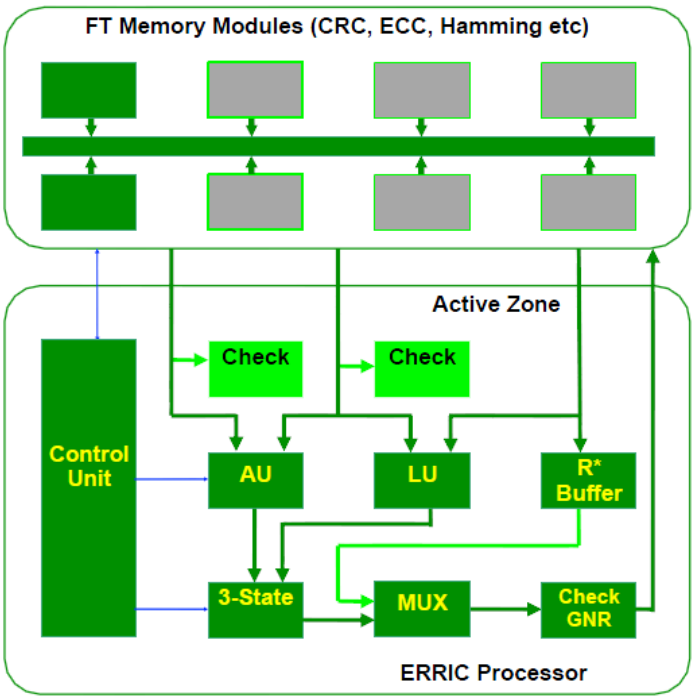


Figure 6-7. Processor structure with "separation of concerns"

Figure 6-7 shows conceptually the ERRIC's active zone divided by two AU and LU elements.

To be able to trust information regarding the status of an element, every checking signal about the condition of registers (not shown), memory units, AU and LU as well as control unit should be aggregated in syndrome. The implementation scheme of fault tolerance separates the passive zone and active zone of the proposed architecture. A clear separation of the functions of

processing (of data operation) and storing (memory) enables us to apply more flexible checking and recovery solutions. The passive zone has the elements to store data, while active zone is used for the data manipulation.

All processor registers (register file) may be protected by parity or other checking schemes. During instruction execution data is loaded from the register file into functional elements for operation and operands are checked. If no fault is detected during instruction execution the operation is considered as successful and the result is stored back into the register file or sent out to memory.

The memory context data might be protected by schemes such as error checking code, Hamming code, or recently proposed schemes (Gössel et al., 2008). If a fault is detected during the course of an instruction, the control unit (by executing GAFT) would attempt to:

- Restore the damaged data
- Repeat the execution of the instruction that presented the fault
- Resume execution

ERA provides a fast and reliable recovery scheme within instruction execution level, transparently to software.

Taking into account the requirement of minimal redundancy the architecture presented on Figure 6-7 above seems efficient – there is no duplication or greater level of reservation applied for active zone. Active zone consists of two non-identical units: arithmetic and logic units respectively (AU and LU).

In terms of power consumption this scheme is also efficient. A question arises:

HOW TO TOLERATE FAULTS IN THE ARITHMETIC OR LOGIC UNITS? Q2

The question Q2 above becomes crucial for the implementation of fault tolerance with minimum redundancy on any system. A possible solution is to apply well-

known mathematical results about the equivalence of arithmetic polynomial for logic functions (Vykhovanets. 2004). That is, a theory where arithmetic functions are represented by Boolean functions. These two groups of results hint us a possible solution: an ISA that consist of logic and arithmetic instructions should be supported by a sequence of functional equivalence of logic operators implemented by arithmetic unit. For every arithmetic instruction a functionally equivalent sequence of logic instructions should be added. Both sequences should be stored in different segments of read-only memory. If a fault occur in the AU a signal sets a flag in the syndrome and sequence of logic operators is executed instead of arithmetic instructions to complete GAFT, and vice versa.

Note that this equivalence and hardware redundancy is used only when a fault is detected and the recovery procedure is initiated. This enables the system to recover from transient errors and execute reconfiguration. After reconfiguration, the system can either continue functioning in normal mode or, in case of an unrecoverable permanent fault, it can provide fail stop sequence of actions.

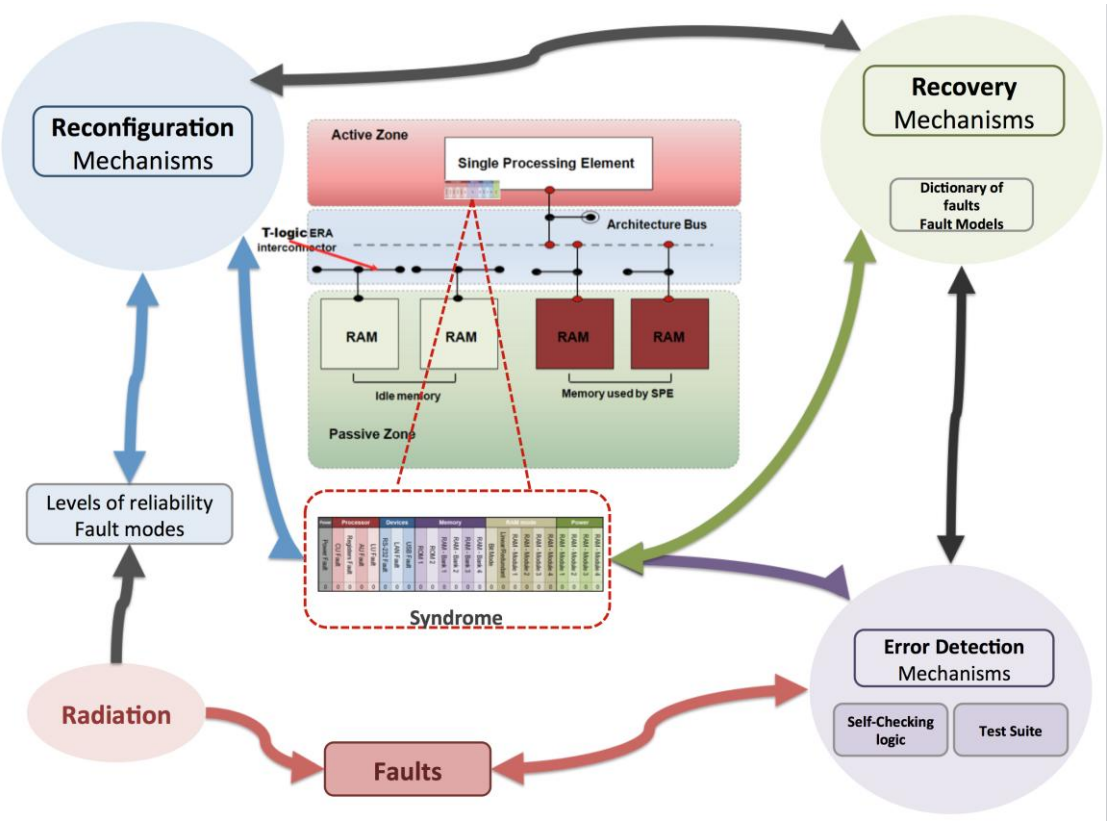


Figure 6-8. Syndrome purposes

The syndrome acts as a control centre for three main functions: fault monitoring, reconfigurability and recovery (Figure 6-8). These three functions serve for the purpose of performance, reliability and power efficiency.

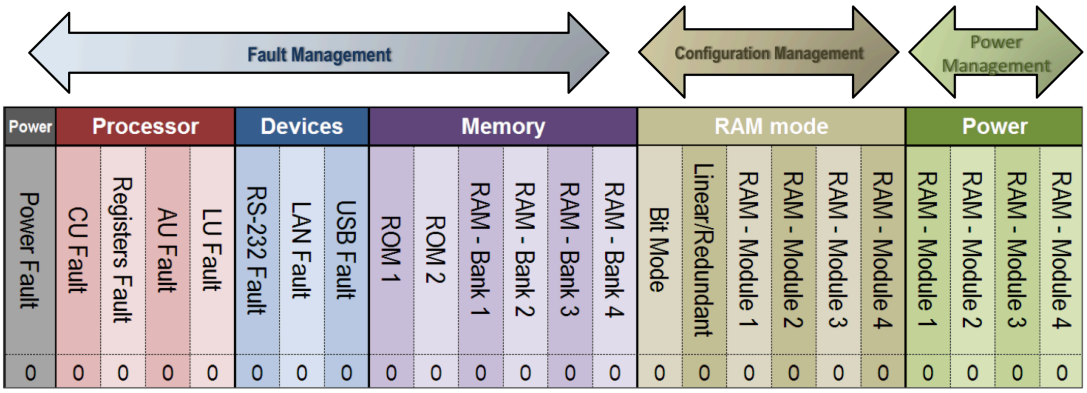


Figure 6-9. Syndrome for reconfigurable architecture

From a low level point of view the syndrome is represented as a special hardware register that will interact with the system via hardware interruptions schemes.

Semantically, the structure of the syndrome is subdivided in three different areas Figure 6-9: Fault control area, Configuration control area and Power Control area.

The Fault control area reflects the hardware status of the different areas of a single processing element: processor, memory and interface. Full "Zero" syndrome in this area indicates that no fault has been detected in the system. If a fault in a specific element is detected, the corresponding bit is set to 1. Each hardware element has sits own representative flag within syndrome. For the ROM memory group that consists of two chips syndrome of ROM condition has two positions with zero whenever the ROM functions correctly. In turn, the static RAM memory group has 4 hardware elements.

The configuration area of syndrome reflects the memory mode that ERA is currently implementing. The Bit mode field defines whether the addressing mode is 16 or 32 bits, whereas the L/R field defines whether the memory banks are in linear or redundant mode.

- Bit Mode: 0 = 16 bits
  1 = 32 bits addressing mode
- LR: 0 = Linear
  1 = Redundant

The power management area reflects the status of the modules in terms of power. Bits Power RAM Module One, Two, Three and Four represent whether the Memory module is powered or not:

- 0 = Power Off
- 1 = Power On

The combination of those three areas: Fault, configuration and Power management control defines the state of the system, e.g.: a memory module could be in the following states: faulty, failed, stand-by, ideal and off state. Checking those 3 areas by a simple reading would determine the status of the memory module. When reconfiguration is set by software, the states of the syndrome might be mirrored in memory. Keeping those states only in SSW is not a universal solution, for example when external element requires an access to a syndrome and local memory has failed further use of available resources and elements becomes impossible.

Without a doubt, the syndrome is one the most critical parts of the system. For reliability purposes, there are three copies of the 32-bit register syndrome connected to a voter within the processing element . Without this replication, a bit flip in the faults area of the syndrome would lead to redundant fault detection processing, whereas a bit flip in the configuration area would likely end up causing a catastrophic failure.

6.4.2. Location access and way of operation of the syndrome

There are two major mechanisms that will be able to detect a fault: hardware logic and SSW. Both, hardware logic (mismatch and voters) directly and SSW via instruction (testing and detection mechanisms) should have access to the syndrome:

SSW events: For SSW to have access to the syndrome it is necessary to implement hardware elements that facilitate reading and writing its value. Note that the current ISA does not include instructions within the processor to do that, unless one of the registers is used as a syndrome. This is similar to the VAX V70 processor that includes a status word placed in a fixed address of the memory.

The register file could be hardened to avoid potential errors in it. At least the syndrome needs to be hardened. For instance, a SEU that causes a bit flip in the configuration area of the syndrome could mistakenly turn off one of the memory modules.

Triplicating the syndrome increases complexity of logic. The voter would be vulnerable as it is a single point of failure (unless the voters are also triplicated). Another option that solves the complexity would be low-level hardening techniques and/or using different technologies (such flash memory) just for the syndrome register. However, that will increase the manufacturing costs. Since, the size of the syndrome is just three bytes, triplication seems the most efficient solution.

As a consequence, a preferred way to access the syndrome that avoids changing the ISA and hardening the register file is the use of the input/output memory lines (mapped in a reserved address) and accessing a TMR Syndrome Scheme that is software independent. Regardless of the hardware implementation, only one syndrome is visible to the system software. An error in one of the syndrome registers is corrected by hardware without software intervention. However, for the purpose of risk analysis it may be useful that the SSW is aware of errors in

the syndrome. Errors within the syndromes are useful information in a potential contingency plan (e.g.: setting the fault tolerance of the system to a higher level in case of recent particle impacts).

Automatic events via hardware detection mechanisms: Special HW interruptions are needed for this; if during the diagnosis of a memory chip the ALU, for instance, signals a problem, a diagnosis of the suspected element should be done first.

The syndrome might be considered independent from other processor hardware elements. A method is needed for synchronizing the operation of the processor with the syndrome. One solution would be polling, where a loop that checks the status of the syndrome is arranged. It has a major disadvantage: the processor is busy reading the syndrome, instead of executing some useful code (wasteful in terms of processing power).

Instead of polling the syndrome waiting for a change, a hardware interruption system is preferred. In this case, the syndrome subsystem is responsible for notifying its current state to the processor. When the syndrome needs the processor's attention, it sends an electrical signal through a dedicated pin in the interrupt controller. In such case, the processor stops its current activity and jumps to execute a function (interrupt handler), which must be associated with the fault manifestation.

By using hardware interruptions, in terms of total execution time, the syndrome will be accessed only when a fault is manifested. Most of the time the fault area of the syndrome will be 0 and the rest of the areas will only be altered when restarting of the system or when changing the memory mode.

The same method might be applied to the control of other devices. Different levels of interruptions are then needed. Active zone hardware should have higher priority than passive zone elements.

If a hardware mechanism sets the syndrome bits, a trap (exception) takes place and the software diagnosis starts. SSW treats the fault and clears the syndrome accordingly. This scheme introduces a requirements to set and re-set syndrome register internally or externally, from the "rest of the system" when ERRIC is used in the form of CC – connected computer structure.

If the syndrome was located within the active zone, in the case of a fault in such area, a neighbour processing element would have difficulties accessing to it. To resolve it there are three options:

- To enable system flexibility in fault handling, one has to implement the syndrome independently from the active area where it can be accessed via hardware by neighbour single elements.
- Software message passing: Instead of hardware access SSW will deal with the status of single elements, by sending un update (periodic updates) of the syndrome to the single element neighbours before changing the memory mode or when a fault has been detected (if feasible in this last case). A syndrome table in a similar fashion to the routing tables used by routers and the different routing algorithms could be shared by different processing elements.
- Both, Hardware access and SW message passing are not mutually exclusive. A combination of both may be possible.

6.4.3. Syndrome: Passive Zone Configurations

A total of 25 memory configurations are possible to operate in reliability, energy or performance wise modes. The characteristics of the proposed memory architecture are given in the following tables: Table 6-3 and Table 6-4.

Table 6-3. 16bit addressing modes in RA

State	Phase	RAM mode		Information - Power				Space (Mb)	Reliability	Speed	Power Consumption	Syndrome
		Bit Mode	Linear/Redundant	RAM - Module 1	RAM - Module 2	RAM - Module 3	RAM - Module 4					
11	4	16	L	A				1	1	2.5	2	111000
12	4	16	L		A			1	1	2.5	2	110100
13	4	16	L			A		1	1	2.5	2	110010
14	4	16	L				A	1	1	2.5	2	110001

Memory modes are subdivided into two major categories, depending on the number of bits read/written at once: 32 bits and 16 bits. At the moment there are 10 different usable memory configurations in 32-bit mode (defined in Table 6-4.) and 15 different configurations in 16 bit mode memories (defined in Table 6-3).

In addition, two modes of operation, depending on the existing amount of redundancy can be selected: (1) Linear mode: where no module is replicated and (2) Redundant mode: with at least one module being replicated. In order to explain the available memory modes, let's consider the letters in the set *{A, B, C, D, x}* as a representation of information stored in the 16-bit memory modules, where:

- A lowercase letter x represents a module that is not in use.
- Identical letters represent identical information in different modules, i.e. information stored in a module is n times replicated into n modules: e.g. AA, AAA, BB and AAAA. In this case n-1 modules are connected in shadow

mode and perform all memory operations concurrently to their respective master memory module. A hardware voter in the memory controller compares the output of the memory modules and, in case of mismatch, triggers an interruption that sets the corresponding fault in the syndrome.

- The pairs AB, BA and CD represent two 16-bit modules combined into a virtual 32-bit module.

6.4.3.1. 32 bit mode

The memory modules in ERA are 16-bit wide. Therefore, two memory chips are combined to allow 32-bit memory access.

State	Phase	RAM mode		Information - Power				Space (Mb)	Reliability	Speed	Power Consumption	Syndrome
		Bit Mode	Linear/Redundant	RAM - Module 1	RAM - Module 2	RAM - Module 3	RAM - Module 4					
1	1	32	R	A	B	A	B	2	2	4	4	111111
10	3	32	L	A	B	C	D	4	1	5	4	101111
2	2	32	R	A	B	A		2	1.5	4	3	111110
3	2	32	R	A	B		B	2	1.5	4	3	
4	2	32	R	A		A	B	2	1.5	4	3	111011
5	2	32	R		B	A	B	2	1.5	4	3	110111
6	3	32	R	A	B			2	1	5	2	111100
9	3	32	R			A	B	2	1	5	2	110011
8	3	32	R	A			B	2	1	5	2	111001
7	3	32	R		B	A		2	1	5	2	110110

Table 6-4. 32 bit addressing modes in RA

Table 6-4 reflects all the supported memory combinations in 32-bit mode. There is only one possible configuration in 32-bit Linear Mode (state 10 in Table 6-4.). Module 1 & 2 are combined and mapped in the memory space. Module 3 & 4 are combined as well and mapped contiguously in the memory. This configuration provides maximum space for the application but does not feature fault tolerance, and not even fault detection at the memory level.

In terms of fault tolerance, the maximum redundancy based on 32-bit addressing is the ABAB configuration (Table 6-4.). The information in the two modules AB is replicated on an extra pair AB. In every reading operation the data from both memory pairs is compared. If there is a mismatch, a checkpoint area in RAM will be the decisive factor in selecting which modules has a fault.

Note that 32-bit modes that involve 3 working modules such as *ABAx, ABxB, AxAB* or *xBAB* are not initial modes (starting in this mode does not offer any advantage). Those four modes would typically involve that an error has occurred in one of the modules and diagnostics is taking place.

6.4.3.2. 16-bit mode

With the main purpose of critical energy saving a 16 bit addressing mode was introduced. This mode is using a single bank of memory. Hence, there is not duplication of memory. The four different memory configurations, currently allowed in this mode are presented in Table 6-3.

When only one single mode is available due to permanent faults in the other three, the system could be restarted in a fresh 16-bit mode loading the different SSW binary codes from a ROM location into RAM. Note that this mode is an emergency mode and does not contemplate the possibility of hot switching from a 32-bit mode. Schemes to change from 32-bit to 16 bit/power saving mode and vice versa are defined by the runtime system. This saving energy mode can be enough to execute a program with performance degradation.

6.5. Graceful Degradation

If one of the memory modules fails, the system can be reconfigured to exclude such module from the current configuration. The ten admissible states in 32-bit mode are given in (Table 6-4.). In terms of fault tolerance, the maximum redundancy based on 32 bit addressing is the ABAB configuration. The transition between these states is completely dependent on soft/hard errors and the

efficiency of recovery mechanisms. However, a voluntary transition between different states is also allowed.

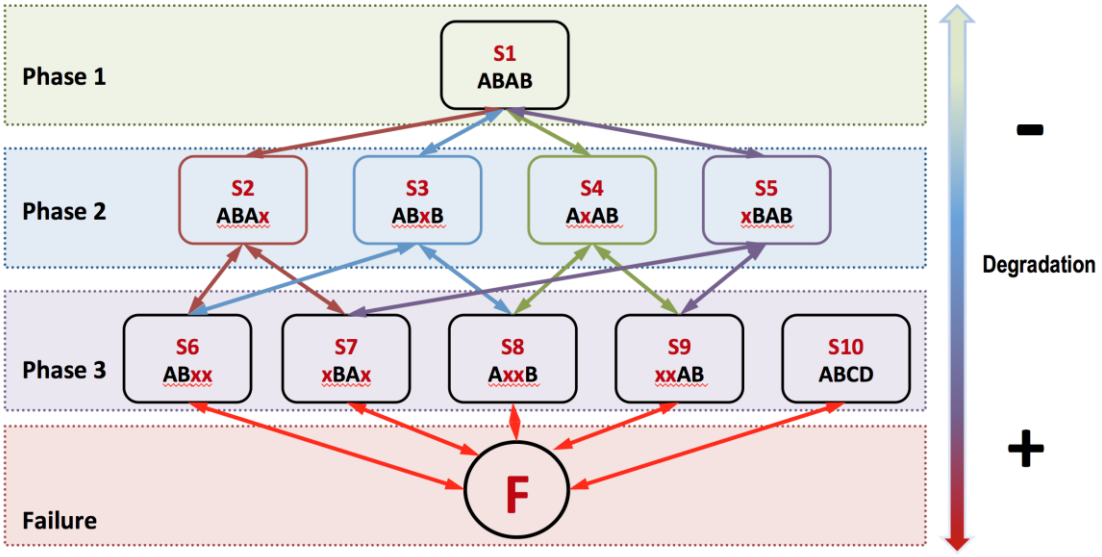


Figure 6-10. 32 bit degradation phases

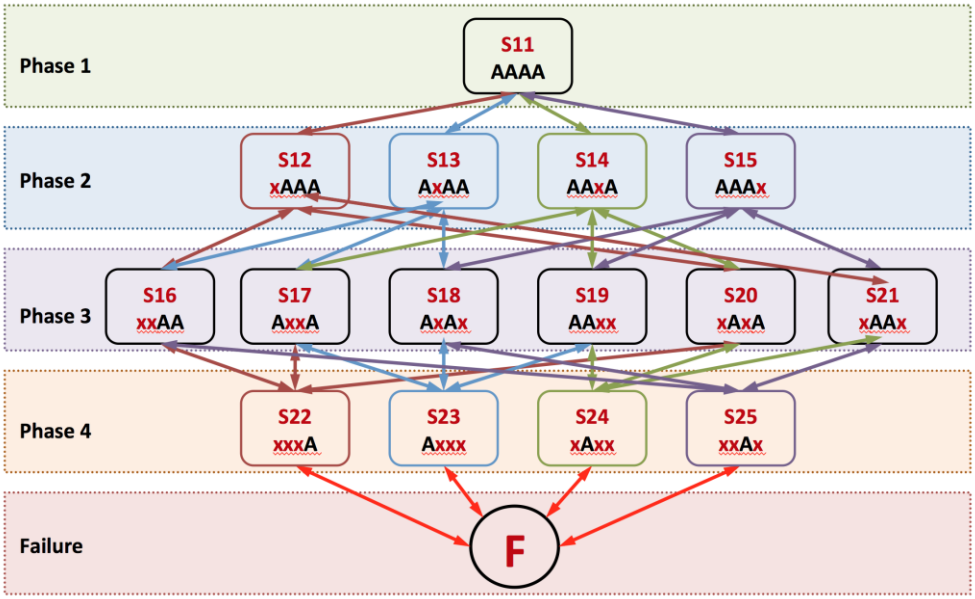


Figure 6-11. 16 bit degradation phases

In case of degradation, successful recovery mechanisms could produce a transition from a degraded phase to an initial phase. The group of states or phases can be distinguished:

- Phase 1: States with full checking and replication of every single bit. The single state in this phase is the initial state and has maximum redundancy for 32-bit: ABAB configuration (see Table 6-4.).
- Phase 2: States in which at least 50% of the bits are replicated. Transition to one of the four different states available in this phase is due to a fault in one of the memory modules of State 1.
- Phase 3: States in which replication of bits does not exist. Six possible states in this phase could potentially lead to a failure.

In a normal scenario with one processing element, if during the operation phase three, a working module experiences another error, reboot of the system is the only way forward. In a multi-processing element scenario and depending on the resources available, reliability needed and current environment, SSW will have to decide whether to:

- cold-switch the same element and restart using the 16-bit mode
- keep using the healthy active area but make use of the neighbour memory elements or,
- cold-switch the execution to another element

6.5.1. Graceful Degradation – Markov analysis

Regarding computational capability and availability of resources, a system can be modelled as being in one of many possible states. The number of states would be large, if fine distinctions are made, or it may be relatively small if similar states are grouped together. Different events can force the system moves from one state to another depending on resource availability and computational changes.

By quantifying the probability of state transitions, State-space modelling can determine the probability of the system being in each specific state; this can be used to obtain some parameters of resilience (reliability, safety, maintainability, safety, etc.).

The sum of all input and output transition probabilities of each state should be 1. The state of the system is characterized by the vector $(S_0,\ S_1,\ S_2,\ S_3, \dots ,\ S_n)$. A transition probability matrix has *N* states. On the *t'th* time-step the system is in exactly one of the available states q_t:

$$q_t \in \{S_1, S_2, S_3, \dots, S_n\}$$

There are discrete time-steps, *t=0*, *t=1*, *...* We are interested on how the system will behave after several time steps. Initial condition of the system: S_0. Given q_t , $q_{t+1}$is conditionally independent of $\{q_{t-1}, q_{t-2}, \dots, q_1, q_0\}$, that is:

$$P(q_{t+1} = s_j | q_t = s_i) = P(q_{t+1} = s_j | q_t = s_i, any\ earlier\ history)$$

The current state q_t determines the probability distribution for the next state q_{t+1}. In order to model of the system we do the following assumptions:

a) System starts in the perfect state

b) Only one fault can occur at a time

c) System does not contain repair – permanent faults

In order to simplify we make a first-order Markov assumption: we say that the probability of an observation at time n only depends on the observation at time n only depends on the observation at time *n-1*. In a sequence $\{q_1, q_2, \dots, q_n\}$

$$P(q_n | q_{t-1}, q_{t-2}, \dots, q_1, q_0) \approx P(q_t | q_{t-1})$$

Using the previous assumption, the joint probability can be expressed as:

$$P(q_1, q_2, \dots, q_n) = \prod\nolimits_{i=1}^{n} P(q_i | q_{i-1})$$

Figure 6-12 presents Markov model of reliability for reconfigurable memory described above:

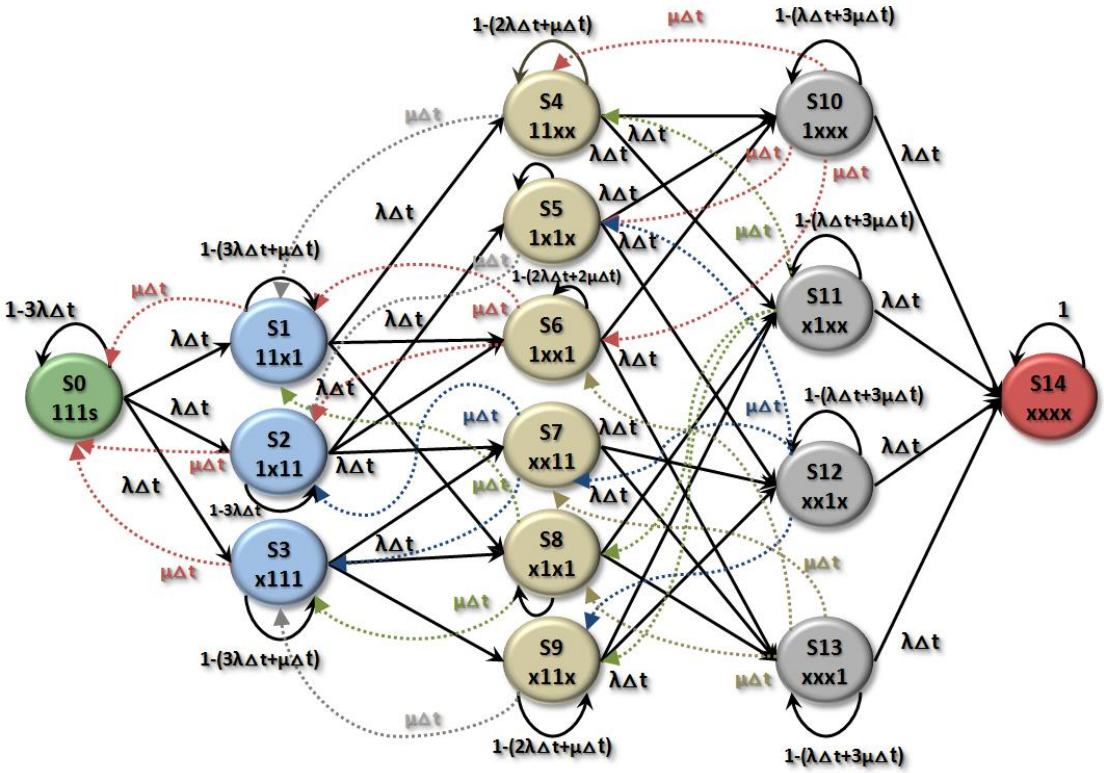


Figure 6-12. Markov model for the ERRIC memory system

The figure shows the transition probability in the case of a four-module memory scheme with a TMR plus a spare configuration.. The circles in the figure represent one of the 15 possible states. The arrows reflect transitions from a source to a destination state.

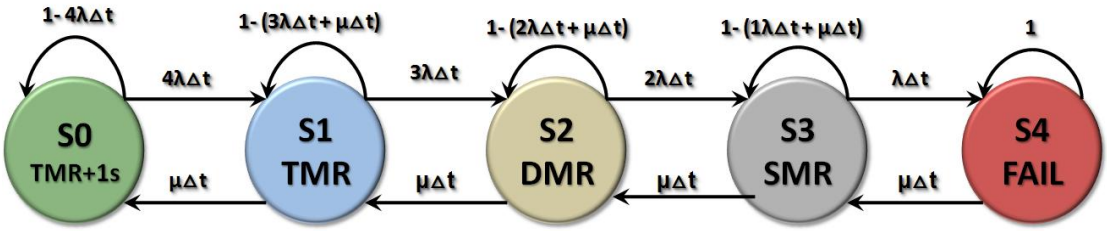


Figure 6-13. Reduced Markov model for the ERRIC memory system

By merging states in the Markov model a simpler equivalent model is created. Figure 6-13 represents such simplification illustrating the probabilities of transition between the original TMR plus spare, a TMR, a DMR, a SMR and a FAIL states.

6.6. Implementation constraints

As explained before in 6.2.1 taking into account the 16-bit instruction size and 32-bit word size of memory organization, at the time of compilation the compiler schedules memory loads on the first 16-bit instruction of a 32-bit two-instruction "packet". This way, memory loads and instruction fetches never occur at the same time. When the memory configuration is set as 16-bit words each read or write precede instruction execution. In the case of a 32-bit memory configuration two instructions might be loaded from memory by one access.

6.6.1. Memory Addressing

Different models could have been followed to implement the ERA's memory addressing scheme. We consider physical addressing and relative addressing.

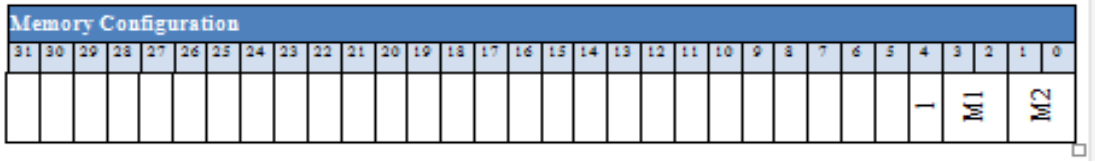


Figure 6-14.Theoretical memory configuration for reconfigurability

Following the first case, as shown in Figure 6-14, the lowest 4 bits of a 32-bit memory address would represent which modules are used to represent the lower and the higher 16 bits. The encoding would be as follows: Bits 2 and 3 correspond to M1 and represent the lower 16 bits whereas bits 0 and 1 correspond to M2 and represent the higher 16 bits.

However, by using this scheme, having the configuration of the modules physically mapped in program code would make reconfiguration difficult to implement. If the binary code reflects which bank the program should run on, then, in case of a fault that requires code to be transferred to another bank (bank switching), a translation would be needed.

Re-compiling of the program is highly unlikely due to the real time constraints of program execution. This also affects recovery time and, in fact, it excludes the chances of recovery in real time.

Besides, if the contents of the memory banks are physically different this will affect the hardware checkers complexity; the checker function would need to compare equal values in case of data comparison and different values in case of address comparison. Thus, separation of concerns principle and a virtual memory approach are preferred.

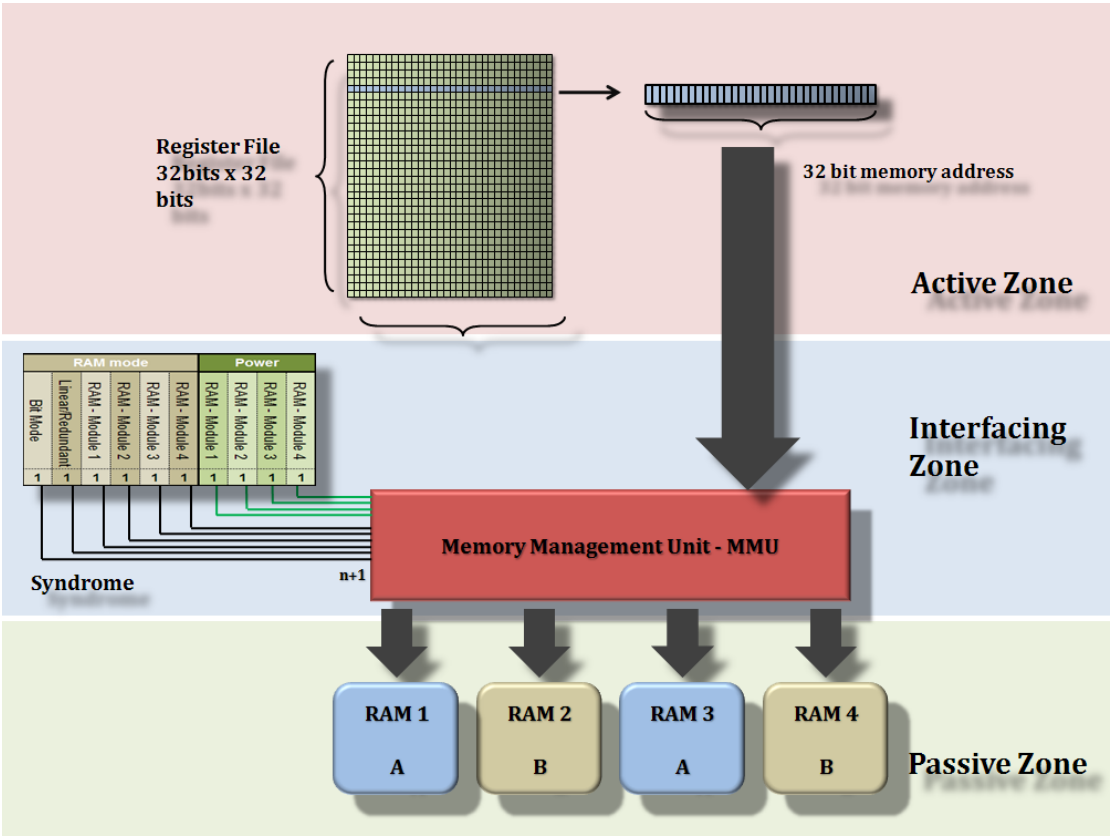


Figure 6-15. MMU and syndrome as memory controllers

By removing M1 and M2 (Figure 6-14), the memory addresses used in a program code refer to a relative position, for instance, within a pair of modules AB. A reconfigurable memory controller (see Figure 6-15) links the relative address to the physical address within a specific memory bank. Such controller is used in a similar fashion to the translation of virtual addresses into physical addresses.

6.6.2. Interfacing Zone: The syndrome as memory addressing controller

As previously seen in Table 6-1 the T-logic ERA element performs configuration and reconfiguration of hardware by providing interconnection and dynamically excluding faulty components from the operational system.

The T-logic interconnector provides flexibility of application of memory elements (32- and 16- bit configurations) and at the same time helps in fault containment. This logic is used to form a hardware configuration scheme adjustable to the program requirements or when a hardware element itself (or architecture) detects hardware faults and thus can't be involved in further calculations.

Note that "*isolation*" might be temporary or permanent, subject to the element's "*health*". The final decision about permanent isolation of an element will take place when testing and recovery procedures are complete.

The four T-logic interconnectors, one for each memory bank, are physically included in the T-logic Management Unit or TLMU. TLMU (see MMU in Figure 6-15) manages the connectivity of the memory, configures and reconfigures the working mode to a 16-bit single memory, 32-bit double memory with master/slave configuration or any of the 14 memory addressing schemes available (Table 6-3 and Table 6-4.). Using the T-logic scheme memory elements could be isolated, switched off for power reduction or doubled in capacity when the maximum storage volume is required, addressing PRE-wise computing.

The Configuration and Power management flags of the syndrome describe the different states of the memory modules. Different values in the configuration area of the syndrome select the bank used and the mode. The output memory lines of the processor determine a location within a memory bank, whereas the Configuration and Power areas of the syndrome specify which banks are to be used and in which mode.

One example of a possible memory configuration (State 1 Table 6-4.) arranged by "T"-logic is presented in Figure 6-15. The example reflects a 32-bit (*Bit mode* = 1) *ABAB* configuration with 2x2 modules duplicated (*Redundant* = 1) working in pairs.

By using this method we can increase the independence of software/hardware configurations for the PRE- purposes. Memory addresses within the code do not need to be arranged, as code integrity is a crucial requirement for safety critical systems.

Let's define the following scenario where it is required to switch data and code from modules 1 and 2 to module 3 and 4. Let's assume that the work mode is *ABxx 32*-bit, which is fast but not very reliable. Assuming that the user (or an online fault detector manager using a fault model) requires a higher level of reliability, transfer from the current *ABxx* mode to an *ABAB* mode is required. An implementation of this example is based on the following algorithm (omitting the testing procedures):

```
 i = 0;
REPEAT
    i++;
    Starting at memory location [0] load the n following words into RF (32x32)
    Change the value of the syndrome to the memory mode XXAB
    Copy the n previously stored words from the RF starting at [0] + i*32*32
    i++
    Change the value of the syndrome to the memory mode ABXX
UNTIL EOM (End of Memory)
Change the value of the syndrome to the memory mode ABAB
```

6.6.3. Access to the syndrome

The design of the TLMU should allow the possibility of neighbour elements accessing the memory elements. Synchronization is therefore required to avoid several elements accessing a specific memory at the same time. Since the syndrome should also be accessible by other elements it may be a good idea that

the syndrome is located in this interfacing area. Having access to the syndrome via TLMU would save logic since only one synchronization element (TLMU) will be used.

6.7. System software support

GAFT might be efficiently implemented by hardware and system software working in a holistic way. In order for a system to be FT it needs to satisfactorily complete three tasks: detection, determination and recovery. Due to the fact that these tasks can be easily decoupled, we will treat them independently. The occurrence of a fault is, in general, a rare event and therefore the error detection, determination and recovery mechanisms are executed with much less frequency than the checking mechanisms.

We present below how with the support of both hardware-checking elements and system software it is possible to mitigate the accumulation of hardware faults that could lead to catastrophic errors. In the case of radiation-induced faults, this combination can remove the effect of one or several SEU before a SEFI takes place. In turn, the SSW needs to be able to support the previously discussed syndrome.

6.7.1. Hardware checking process via SW

Consider a sequence of tests and programs T and P within a system as in Figure 6-16.

Figure 6-16. Ensuring HW integrity through program test execution

The initial test T is executed before any given task execution guaranteeing HW consistency, i.e. it guarantees that there is no fault at time t_0 in the system. However, a permanent fault (e.g. a Hard SEL) that occurs immediately after the first test or during the program execution might be invisible for an arbitrary long time (latent period). Therefore, a second sequential test is required right after the program execution to guarantee that no permanent fault occurred since the last test.

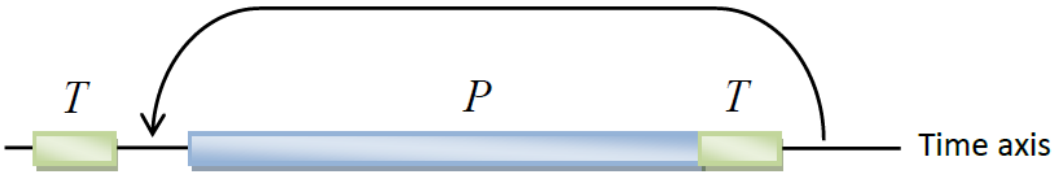


Figure 6-17. Regular sequence of program execution with test of HW integrity to detect permanent faults

For periodic tasks, which are often used in control systems, we slightly adapt this scheme as shown in Figure 6-17. Nevertheless, what if a transient fault occurs during the execution of P? As mentioned in 3.5.2.1.1, at least three cases may take place:

- The effect of the fault lasts until P finishes, T detects the error and the recovery mechanisms are able to restore normal functioning on time (detected recoverable error or DRE)
- The effect of the fault lasts until P finishes, T detects de error but the recovery on time is not possible (detected unrecoverable error or DUE)
- The effect of the fault might not last until P finishes and therefore T would not be able to detect the fault, which in turn would allow the fault to remain undetected, perhaps causing data corruption (silent data corruption or SDC). The corruption could go unnoticed (benign error) or could result in a visible error.

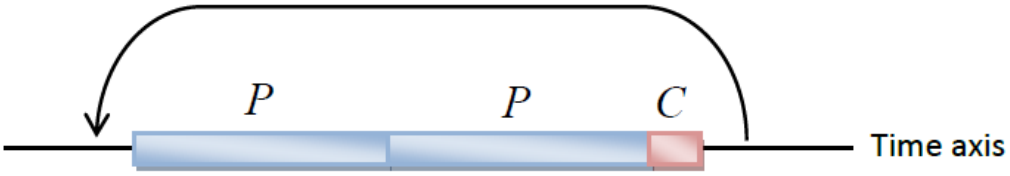


Figure 6-18. Ensuring HW integrity through program test execution to detect transient faults

Transient faults can be detected using time redundancy, re-executing the same program P with comparison C of the result and the result state space. Figure 6-18 illustrates this scenario.

Note that for periodic tasks, the persistent state of the program, i.e. the program state which is used in the next computation as input data must also be compared, as transient faults might affect data that is no longer used in the current computation but that will be used in the next.

Permanent faults however cannot be detected with the comparison scheme alone, as they might affect both executions of P. That is, the scenario in Figure 6-17 can only detect permanent faults whereas the scenario in Figure 6-18 can only detect transient faults.

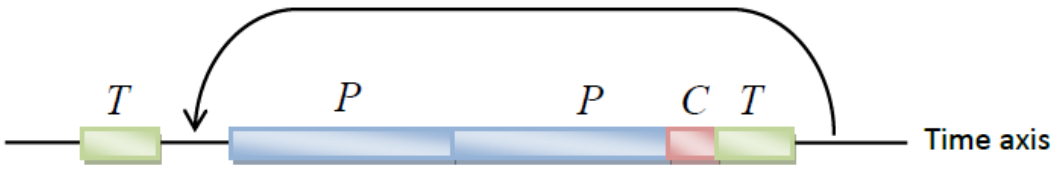


Figure 6-19. Ensuring HW integrity through program test execution to detect both transient and permanent faults

The combined power, in the form of information and time redundancy, illustrated in Figure 6-19 allows the detection of both transient and permanent faults. In this scenario, C is used to detect transient faults and T to detect permanent faults. Assuming that C triggers an error but T does not, it is clear that

a transient fault occurred. Another run of program P with comparison to the previous two runs can identify the run where the transient fault occurred.

In the following analysis, we concentrate on the detection of merely permanent faults, using only T in the analysis. The detection of transient faults can be considered as included in the following analysis if the double execution of P with following C as a whole is treated as task P.

A testing phase is required initially at boot up time to guarantee the correctness of the hardware and also a periodic test before and after the execution of a program. The coverage of the applied tests might vary in the number and type of faults that can be detected and also the set of tested hardware.

Every hardware component has typically at least one assigned test but might also have more than one that could differ on the implementation level. Software based tests need a processor and memory for the test execution even if a peripheral device is tested.

In order to guarantee that faults in other hardware components that are not subject of the test itself do not have an influence on the outcome of the test, the order of the tests must follow the *principle of growing core*: if a test of a hardware component u_i has implicit dependencies on another hardware component u_j, the test of u_j must be executed first.

If the resources needed by a task are known in advance, the testing procedures of the accessed hardware resources only (selective testing), by using the principle of growing core, would be enough. This way, the system stays fully operational even in the case of present faults in some hardware components that are not in use. Spare components can be used for the relocation of programs that were running on faulty hardware components.

Yet, it may be useful to periodically test the full hardware as otherwise, faulty spare components can be considered as fully operational and might be used

again in a subsequent reconfiguration process. A full hardware test also allows the system software to monitor the current full state of the hardware and take appropriate actions when needed.

Timely task completion in real time systems is a key requirement. Consequently, the testing overheads should be reduced as much as possible when/where necessary.

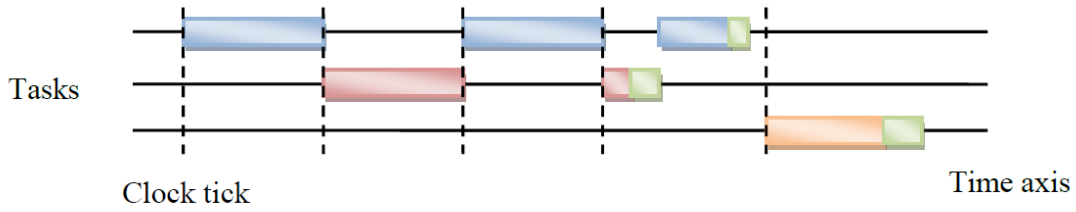


Figure 6-20. Tasks & tests combined

Figure 6-20 exemplifies a scenario of three tasks with their corresponding tests. Analysis of the checking process assumption in this case is based on a scheduler, which distributes time slices to the running processes. In this example, the processes run to completion and are called periodically by the scheduler. Three tasks are running, each with its own test (green boxes) at the end of the task execution. The test only checks the resources that the respective process needs, which results in different test execution times. The task execution is only considered as successful if the test at the end of the task is successful. If the test failed, the task is re-executed by using the same input data set as in the first try.

If no spare components are available in the system, all programs depending on this component must be obviously terminated. If no essential program is affected by this component, the system can continue operating in a degraded mode.

For diagnostic and monitoring purposes the test results should be available to the SSW in that unit or to external systems. Therefore, we propose to organise the test results of hardware in test syndromes.

6.7.2. Software support for reconfiguration

For every hardware component, e.g. register file, ALU, internal bus or device controller, the checking procedures present their result in the form of a syndrome to the software indicating, in binary form, the state of the device. By grouping all syndromes together in one register, the software has a very effective way to check the integrity of the system. In case of a non-zero syndrome further analysis of the hardware conditions are required, especially when the duration of the malfunction is long.

Depending on the checking scheme used, it is not only possible to signal a fault to the runtime, but also to provide him with extra information to ease recovery. For instance, let's define a scenario where the testing schemes discover stuck bits in memory due to a Hard SEL. It would be sufficient to recover programs that access the affected location and not all programs that are using the affected memory module.

Device drivers could for example provide their own testing schemes for their respective device. Especially for devices, one could think of having a combination of hardware and software based testing. I/O devices such as UARTs could effectively be tested by cross connecting the input and output wires using very simple additional hardware logic and sending various bit patterns over this loopback connection.

In case of a detected hardware fault, the syndrome raises a hardware interrupt and SSW takes control of the reconfiguration process. The whole procedure is almost identical in the case of software-based schemes detecting the fault, with the difference that the interrupt that is raised is not hardware but software based. In the case of memory errors, if the current memory configuration does not use a redundant mode, software based checking is the only possible approach.

The general procedure of software support during reconfiguration is listed as follows:

- The hardware checking scheme triggers the syndrome interrupt
- In order to distinguish the fault type, the syndrome interrupt handler then either initiates a *HW Built-In Self-Test* (*BIST*) procedure of the device or runs a SSW based self-test. In the case of SSW tests, writing different memory patterns to the faulty memory address can be used to derive the fault type. The syndrome bit indicating the fault has to be cleared after recovery. If after the test and recovery, the syndrome still shows the fault, the memory module is considered faulty. The affected memory address is still present in one of the processor registers and based on the IRQ return address, the correct register number can be derived by decoding the memory instruction that triggered the fault. In general, all software based testing procedures that test memory must not use the stack (no procedure calls, no data pushed on the stack) until the proper functioning of the used stack locations is ensured.
- In case of a transient fault, the event is logged and the program execution resumed. Logging the events is important as an accumulation of malfunctions in a module or a specified memory location could hint a potential permanent failure in the near future.
- In case of a permanent fault, the current memory configuration is extracted from the syndrome, and the next degradation state is calculated according to the application needs and predefined degradation tables.
- The new calculated memory configuration is written to the syndrome registers and the power of the faulty unit is removed. In some configuration transitions, the SSW has to adapt to the new situation and recover after excluding the faulty unit but before including the new one.
- SSW clears the fault in the syndrome and resumes processing by returning from the interrupt.

Some of the presented transitions in 6.5 need software intervention to fully recover from a permanent fault and to establish a new working software state. We show here a few situations where SW support is needed:

- *Adding/Replacing a module of an already populated bank:* if a memory module of a redundant memory mode (DMR or TMR) that is suspected of presenting faults is replaced by another module, memory content must be replicated to the new module before it is included in the configuration. A small routine following the algorithm described in 6.6.2 is sufficient to perform the copy without modifying the memory during the operation. Before performing that routine, SSW configures the syndrome to include a spare memory bank. After the copy operation, the spare module can be included in the working set. These actions must be performed for example when the system switches from Phase 2 to Phase 1 in Figure 6-10.
- *Failed module in the runtime system area*: The area of memory that contains by convention all runtime system data structures is critical for system operation. When the recovery procedures are unable to overcome the situation, if the module has a replicated pair in redundant mode the affected module is replaced by its replicated version via syndrome. However, if the module is not replicated, resetting the system either via a hardware watchdog or a software initiated power cycle is the option as a last resort. The BIST mechanism automatically identifies the failed module and configures another still working module to bank 1. The runtime system can then restart all critical applications.
- *Fault in a memory module that is not replicated:* this case is the most difficult to handle as the software must adapt to the smaller available memory space.

Instead of graceful degradation, software can also decide to "upgrade" the system in terms of redundancy, i.e. going from a mode with less redundancy to a mode with higher redundancy. The inclusion of a spare module corresponds to the first point in the list above; if an already used module is moved to another

bank, software has to release all data structures residing on that module in case it is in non redundant use and repopulate it with data according to Point 1.

Intentional change of the operating mode to a less redundant mode is of course also possible, and needs no special software measures. As soon as the module is reconfigured to a free bank, the runtime system can start using it.

6.7.3. Hardware condition monitor by system software

A hardware monitor, which is part of the runtime system, is responsible for keeping track of the hardware state. For every hardware component that is managed by the syndrome, the hardware monitor tracks the state in more detail than the syndrome alone can provide. It is also responsible for the execution of all software checking schemes and performs the actual hardware reconfiguration. Thus the hardware monitor must be accessible by the syndrome interrupt handler as well as the runtime system. This monitor should however not directly be accessible by applications; only drivers, which are part of the runtime system, can register checking procedures for their respective hardware component.

When the system is turned on, the BIST procedures embedded in the system are executed. These run tests, using the principle of growing core, ensure the integrity of all devices. If a failure is detected, the syndrome sets the appropriate fault bits. The BIST is also responsible of initiating the system to a predefined working state, i.e. the most reliable mode with all working available resources.

When the BIST finishes and gives control to the runtime system, the hardware monitor first mirrors the current state in software and then reconfigures the system according to the needs of the program. As the syndrome might trigger an interrupt right after boot up, the syndrome interrupt handler has to ensure that the stack pointer is valid and if not initialise it.

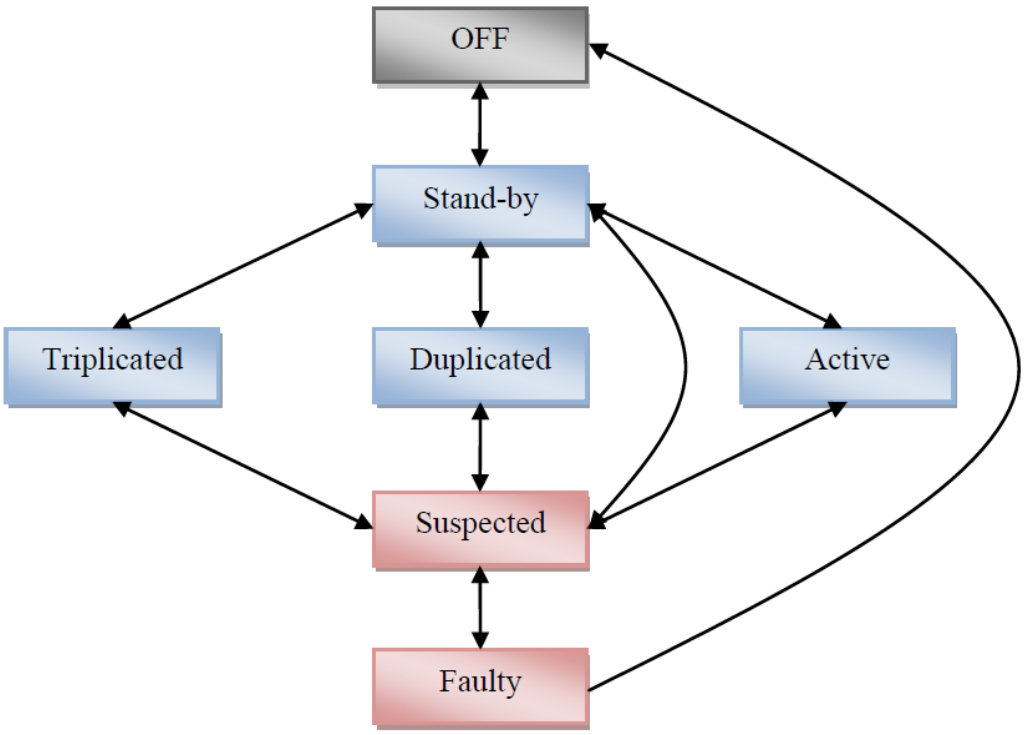


Figure 6-21. Hardware state diagram

Every hardware component managed by the syndrome is in exactly one state of Figure 6-21. This state diagram shows also all possible transitions between states, allowing the hardware monitor to reconfigure the system in a consistent manner. In fact, all of the previously presented cases in the degradation scenarios where software intervention is required are clearly identifiable in Figure 6-21. Intervention is only required if the state transition goes from Stand-by to one of the active cases (marked in blue).

After boot up, all devices are either in state OFF or in one of the blue operation modes. As the BIST automatically configures the memory configuration with the highest reliability possible, the initial states of all devices must be acquired by reading the syndrome. Here is a short list of all possible states and a short description:

- ***OFF:*** the device is currently not in use, powered or isolated for fault containment

- ***Stand-by**:* the device is powered on but not in use. In case of reconfiguration, all transitions go through this state.
- ***Active**:* the device is in use in a non-redundant mode. In case of permanent fault, the runtime system would switch execution to a bank in non-redundant mode.
- ***Duplicated***: the device uses a DMR configuration.
- ***Triplicated***: the device uses a TMR configuration.
- ***Suspected***: As soon as a fault in the hardware is detected, the state of the affected hardware component is set to suspected and the testing procedures are initiated to diagnose the fault. If a device is often in this state, this could be a hint that the device might fail in the near future. For reliability purposes it might be sensible to replace the component with a spare one.
- ***Faulty***: depending on the analysis outcome from the diagnosis mechanisms, the state is then set either to *Faulty* if a permanent fault was diagnosed or back to the previous state if it was only a transient fault.

The state transition diagram of Figure 6-21 is not directly applicable to all devices. A memory bank, for example should during the transition not go through stand by to make sure that the stored data in the attached memory modules are not lost.

Despite the state, periodical hardware checks should be performed on every single hardware component. Even Stand-by states must also be tested. This ensures that hidden faults are detected preventing a possible spread. Thus any state can transition to a suspected one.

It is even possible to restore faulty components to a working condition, as for instance environmental changes could allow a component to function correctly again.

6.8. Programming Language for the Prototype

In 5.5.2 we introduced the three main processes of GAFT: P1 testing and checking, P2 recovery preparation, and P3 recovery that were explained further in 6.7. These concepts have been synthesized into programing language extensions and runtime support. The language extensions use Oberon-07 as a basis, which has been developed during the ONBASS project (ONBASS Consortium, 2004) and used in the implementation of the Minos OS (Kaegi-Trachsel and Gutknecht, 2008).

The simplicity, strong type safety, and built-in safe features of the Oberon language makes it especially suitable for safety critical systems. In order to support all GAFT steps, the language has been modernised with new naming conventions and enumerations and new features have been added to the extensions including: reconfiguration at the runtime system level, memory partitioning, activities and message passing, and object orientation, etc. As in all languages of the Oberon family all these features need runtime support. Language and runtime system are hence closely connected.

6.9. Conclusions

A resilient architecture was proposed including the hardware and the system software elements that can provide efficient performance, reliability and energy-smartness.

The principles of designed followed and the structural properties of active, passive and interfacing zones are introduced. Active zone of hardware was also described with emphasis on recoverability after malfunctions and implementation of checking schemes. A processor with a reduced instruction set and a careful introduction of redundancy including checking-schemes and re-execution at the instruction level can provide higher and efficient reliability.

Reconfigurability of a real-time architecture at the system level was proposed and analysed in the context of each zone. With regards to the interfacing zone, a new element (T-logic), as a basic unit of reconfiguration, and its different configurations were proposed. We analysed and described how the flexibility of the T-elements has a positive effect in reliability and power-smart functioning of the system.

System-level reconfigurability can be achieved using a new hardware element called *Syndrome* that can provide essential knowledge about hardware conditions. We showed the relation of this element with the active, passive and interfacing zones and how it can be used to implement GAFT. Functions of the *Syndrome* for reliability, performance and energy-smart functioning were described and explained.

Taking into account that memory use has, by design, a high impact on system reliability and power consumption, passive zone reconfigurability was analysed and described in detail, including the control of configuration and the phases of hardware degradation.

A Markov model of reliability for passive zone was developed and analysis indicates the reliability gain of the different schemes permitted by the proposed reconfigurable architecture.

System software support of testing and reconfiguration (dealing with system syndrome) was fully described. Shown that in combination of novel hardware architecture and system software all key properties of performance, reliability and energy-wise functioning can be improved.

Chapter 7

Chapter 7

Implementation: Hardware Prototype, Simulation and Testing

There is always a gap between “what we design and analyse” and “what we can implement”. While analysis of the technological domain and trends as well as a new theory of resilient systems were organised in the previous six chapters, as a next step, there is a need to demonstrate “proof of the concept” and do-ability of the proposed ideas on the existing hardware prototype. Thus implementation efforts can be summarised as two-fold:

- Development and debugging of existing manufactured ERA prototype;
- Design and development of a software simulator for the developed ERA hardware

The hardware prototype must work properly, being able to execute the basic functions that will be further incremented, so that the system is prepared for industrial use. In applications, ERA will work as stand alone system using interface with PC or Mac to upload and run programs in a form of bit stream as we use FPGA Altera for this. System software and programs for ERA also have to be developed and therefore there is a need for a software simulator for the ERA architecture.

As long as ERA is designed for embedded and safety critical applications we need to have a simulator as close as possible to the architecture and ERRIC's binary code. Therefore, all our efforts in this chapter are organised around the simulation of the instruction set and the hardware element.

7.1. Instruction Execution

Figure 7-1 shows the execution flow of the proposed microprocessor. As we mentioned in 6.2.1, the execution steps are similar to other RISC processors and since there is no pipelining all steps of one single instruction are executed within one memory cycle.

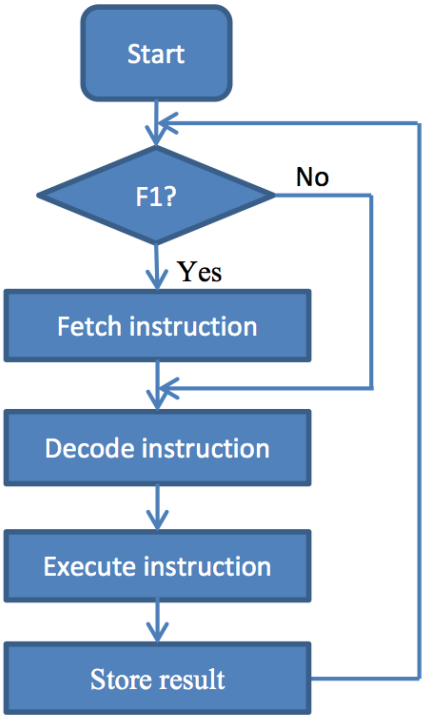


Figure 7-1. Simple version of the Prototype's Instruction Execution flow

The fetching step loads an instruction from main memory storing it into the Instruction Register (IR). Since every instruction is 16-bit wide this step is required every second instruction. Decoding and Execution of the Instruction follows. After execution of the first instruction, the second instruction from the

Instruction Register (*IR*) can be executed without access to memory. This eases the speed gap between processor and memory and reduces their dependency. Finally, storage of the result takes place if the instruction execution has affected the content of registers, processor flags or any other processor state.

The processor has two internal fetching states (F1 and F2) that are required by the memory controller. Again, simplicity is not only applied to the processor but also to the memory controller. Both are designed to avoid possible stalling due to pending memory operations. This can be achieved by interleaving instruction fetches and memory operations.

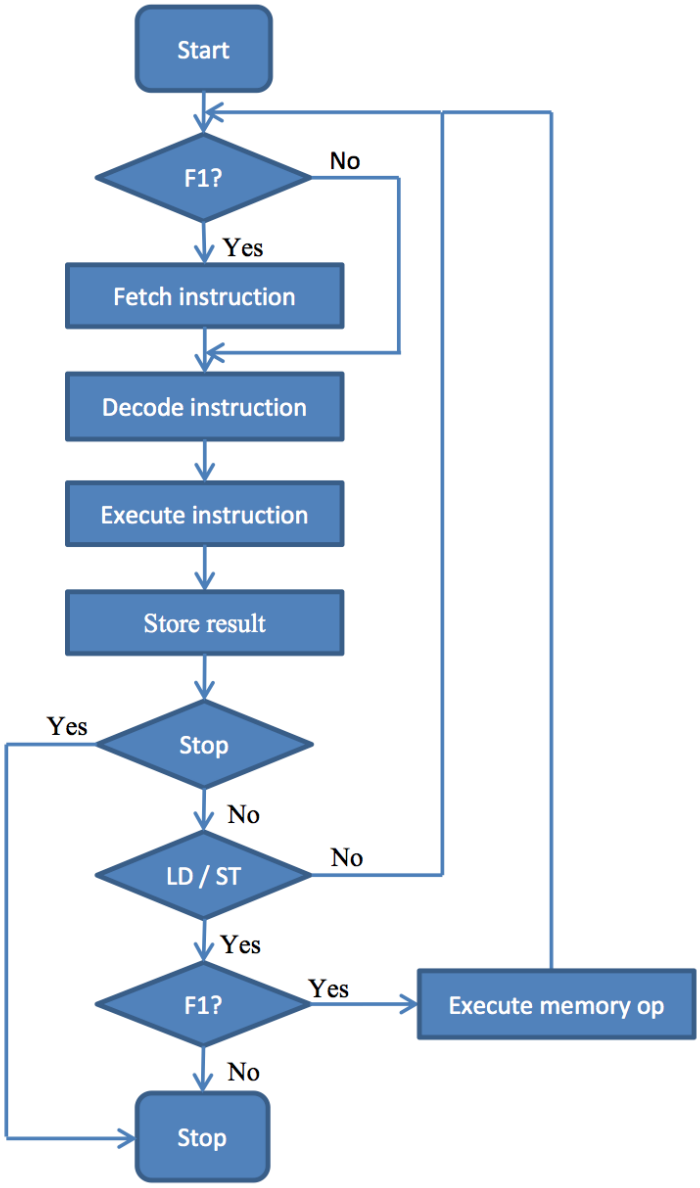


Figure 7-2. Instruction Execution Flow (Extended version)

The fetching step always loads two 16-bit instructions from memory into the internal IR. In sequential instruction execution, the compiler can schedule memory instructions in every second instruction slot where no instruction has to be loaded by the processor. An example of this notion is illustrated Figure 7-2, which is an extended version of the instruction execution flow in Figure 7-1. In the F1 state, the processor fetches an instruction from memory; therefore since the memory unit is busy during this cycle, it cannot execute at the same time an instruction that involves memory. Only when the processor is in the fetching state F2, the processor is able to execute a memory instruction. It is the compiler's responsibility to schedule the instruction in the proper order. We choose to simplify the hardware design at the expense of adding complexity to the compiler. By doing so, we reduce the amount of redundancy and therefore increase system reliability. If the instruction executed is a branch instruction the processor is switched automatically to the F1 state. The reason for that is that the memory controller can only load 32-bit aligned addresses and therefore, the jump destination locations must also be 32-bit aligned. All this factors force the compiler to fill the memory gaps with NOP instructions.

7.2. Instruction Set

As mentioned in 6.2.1, in the ultra reduced instruction set employed, each one of the 16 instructions is encoded into 16 bits and only two of them are memory instructions (load/store)

The instructions are designed as two register instructions. They expect exactly two arbitrary registers as input, and save the result of the operation in the first register, thus overwriting one of the input values. The compiler has to keep in mind that the content of the first register is overwritten. Again we increase the simplicity of design at the expense of compiler's complexity. The impact of the size of the register file on overall performance of processor is a question of further research.

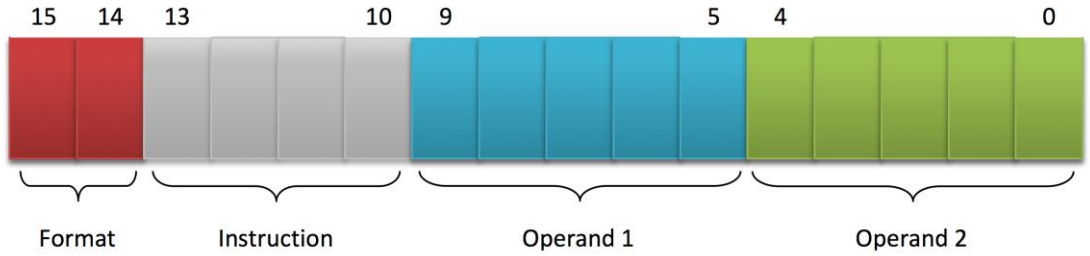


Figure 7-3. Instruction Format

Figure 7-3 illustrates the instruction format divided in four different areas. Bits 15 and 14 (in red) indicate the format of the operation, which could be 8-, 16- or 32-bit. Bits 13 to 10 (in grey) contain one of the 16 different operation codes. Bits 9 to 5 (in blue) and 4 to 0 (in green) contain the first and second operand, which could be any of the 32 general-purpose registers in the *RF*.

The following Table 7-1 lists the current ISA with a short description of every instruction together with their assembler representation:

Table 7-1. Explanation of instructions of current ISA

Name	*Format Code*	*Op. Code*	*Op1*	*Op2*	*Op2*
NOP	01	0000	Ignored	Ignored	Execute no action except increasing the PC
STOP	00	0000	0	0	Stop instruction Execution
TRACE	00	0000	Ri>0	Rj>	Output Ri to debugger. Operand1 or Operand2 must be > 0
RETI	11	0000			Return from interrupt (Address in R31)
LD	11	0001	Ri	Rj	Load 32-bit memory at address Ri into Register Rj(Rj := ∗Ri)
LDA	00	0010	Ignored	Rj	Load the value from the next 32 bit word (rel. to PC) and store it in Rj(Rj := constant). Operand 1 is ignored
ST	11	0011	Ri	Rj	Store content of register Ri to the memory at address Rj(∗Rj := Ri)
MOV	XX	0100	Ri	Rj	Move content of register Ri to reg- ister Rj(Rj := Ri)
ADD	XX	0101	Ri	Rj	Arithmetically add the content of Ri to the content of Rj and store the result in Rj (Rj := Rj + Ri)
SUB	XX	0110	Ri	Rj	Arithmetically subtract the content of Ri to the content of Rj and store the result in Rj (Rj := Rj - Ri)
ASR	XX	0111	Ri	Rj	Shift the content of register Ri arithmetically one bit to the right and store the result in Rj
ASL	XX	1000	Ri	Rj	Shift the content of register Ri arithmetically one bit to the left and store the result in Rj
OR	XX	1001	Ri	Rj	Perform a bitwise logical OR of register Ri with register Rj and store the result in Rj
AND	XX	1010	Ri	Rj	Perform a bitwise logical AND of register Ri with register Rj and store the result in Rj
XOR	XX	1011	Ri	Rj	Perform a bitwise logical XOR of register Ri with register Rj and store the result in Rj
LSL	XX	1100	Ri	Rj	Shift the content of register Ri logically one bit to the right and store the result in Rj
LSR	XX	1101	Ri	Rj	Shift the content of register Ri logically one bit to the left and store the result in Rj
CND	XX	1110	Ri	Rj	Arithmetic comparison of Ri with Rj and store the result in Rj
CBR	XX	1111	Ri	Rj	Jump to address in Rj if Ri is non zero and save PC in Ri

The current architecture comprises 7 control, 5 logic and 4 arithmetic instructions. Whilst most entries in Table 7-1 are self-explanatory some of them need further explanation. A special case of code operation is Opcode 0 which in combination with the format represents four different instructions: *STOP*, *NOP*, *TRACE* and *RETI*. *TRACE* is used for debug purposes and *RETI* is used to exit an interrupt handler.

The compiler needs to be aware that constants cannot be directly encoded in the instructions. *LDA* is another special instruction that loads a constant to the specified register. The next aligned 32 bits aligned after the active program counter store the constant to be loaded. Placing two instructions into one 32-bit word increase code density and performance. As explained earlier, the processor executes the first instruction on the left before executing the second one on the right. Since the program counter has the same value for both instructions it would be problematic to jump directly to a second instruction, which is not 32-bit aligned and has no unique address. Therefore, it is responsibility of the compiler to insert *NOPs* at the right places to prevent cases where instruction reordering fails to fill the gap. .

Table 7-2. CND operation flags

Relation	*Bit Mask*
<	010
≤	110
=	100
>=	101
>	001
≠	011

The *CND* instruction performs an arithmetic comparison of the Registers *Ri* and *Rj* storing the result in *Rj*. The functioning is similar to other platforms: the comparison involves checking of three conditions saving these as flags in the first three bits of *Rj*:

- Bit 0: *R1 > R2*
- Bit 1: *R1 < R2*
- Bit 2: *R1 = R2*

Table 7-2 shows the relations of the comparison operations and their corresponding bitmasks. By applying an appropriate mask to the flags, the result for every possible comparison operation can be used as an argument in a conditional jump or saved as a Boolean value.

At the moment there is no support for unsigned operations. All arithmetic operations treat the values in the operands as signed values. All instructions accept the same register for both arguments with the exception of conditional jump instruction.

More detailed information on the CPU logical structure and the Instruction Set Architecture can be found in Appendix B.

7.3. ERA hardware prototype

Figure 7-4 presents the current custom hardware prototype of the ERA device. The prototype is provided with two flash based ROM modules 128Mb each (8Mb x 16bit) with replicated bootstrapping firmware and operating system software.

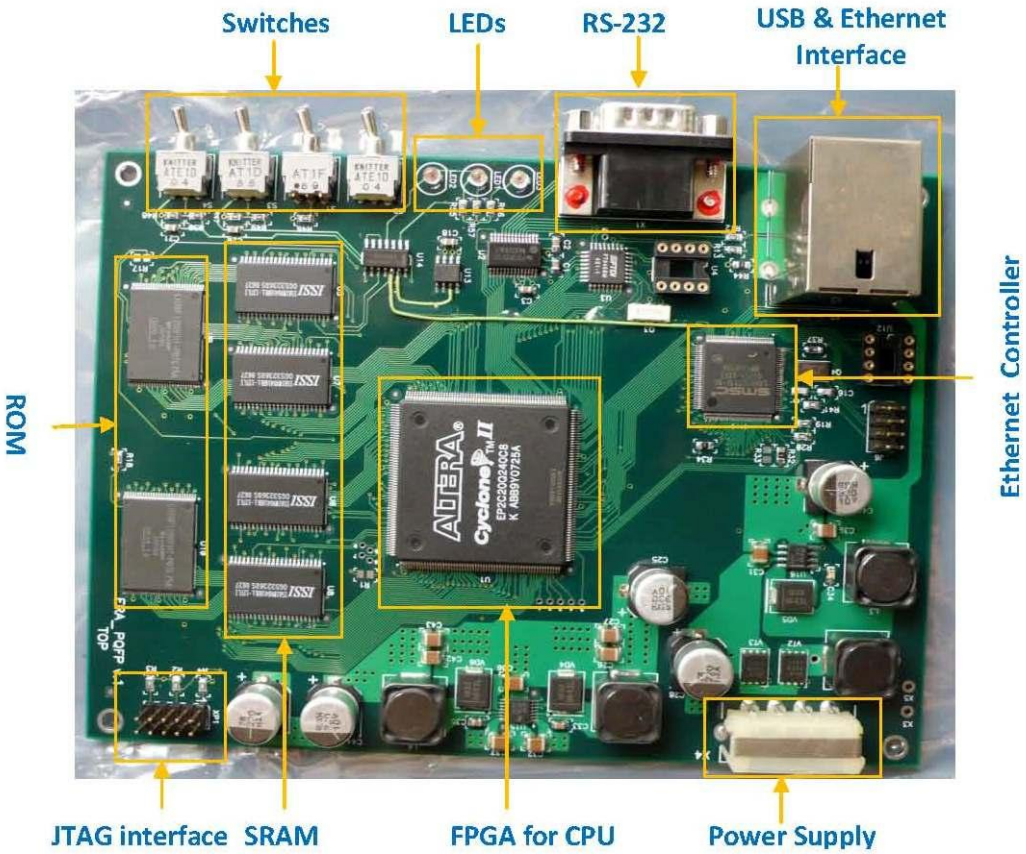


Figure 7-4. ERA prototype board

The proposed memory scheme may be regarded as a collection of 4 blocks of RAM, 16-bit wide, with identical size of 1Mb each (16x64k). Using 16-bit memory modules instead of 32-bit memory modules increases reliability and reduces when necessary energy required for execution.

Reliability is increased by means of added working states and configurations. Energy-wise operation is improving by means of this architecture ability to

activate only modules required - by means of using a single 16-bit memory module, when necessary.

Table 7-3 presents a basic memory map with memory locations occupied by the ERA devices: ROM (2 banks) RAM (4 banks), USB, Ethernet, and UART interface. The 4 RAM modules are 1 Megabit each (64k x 16bit). The flash based ROM modules are 128Mbit each (8Mb x 16bit).

Table 7-3. Device's memory map

Memory Range	*Device Details*	*Device*
0700FFFFH - 07000000H	ISSI-IS61WV6416BLL	SRAM logic module 2 (U7, U8)
0600FFFFH - 06000000H	ISSI-IS61WV6416BLL	SRAM logic module 1 (U5, U6)
057FFFFFH - 05000000H	Sharp-LH28F128BFHT	ROM logic module 1 (U9, U10)
04000000H	FTDI-FT245BM	USB
0300FFFFH - 03000000H	SMSC-LAN91c11i	Ethernet
02000002H - 02000000H	RS232	UART Interface
01000000H	Normal	LEDs (1-2)
00000000H	KNITTER	Switch (3-4)

The SRAM modules representing the units 8,7,5 and 6 are located in the highest part of the memory, followed by the ROM modules (units 9 and 10). The USB, Ethernet, Serial Ports, LED's and switches of the ERA board are mapped in the lower part of memory.

7.3.1. Architectural Comparison

Nowadays the embedded processor market is dominated by the ARM architectures with their RISC processors. Other relevant hardware architectures are the LEON designs that include FT versions of their SPARC processor.

Although 64-bit versions are available for the x86 and ARM architectures, in order to keep consistency, we have chosen to make a comparison of 32-bit version processors including Intel x86's architectures.

Table 7-4. Comparison of Hardware architectures

	ERRIC	*x86*	*SPARC v8*	*ARM7TDMI* *(ARMv5-TE)*	*ARM7TDMI* *Thumb*
ISA type	MISC	CISC	RISC	RISC	RISC
Integer Registers	32x32 bits	8x32 bits	31x32 bits	15x32 bits	8x32 bits + SR, LR
Floating Point Registers	0	Optional 8x32 bits or 8x64bit (8x80 bits internal)	32x32 bits or 16x64 bits or 8x128 bits	Optional 32x32 bits or 16x64 bits	Optional 32x32 bits or 16x64 bits
Vector Registers	0	Optional 8x64 bits or 8x128 bits	0	Optional 32x32 bits or 16x64 bits	0
Address Space	32 bits flat	32 bits, flat or segmented	32 bits flat	32 bits flat	32 bits flat
Instruction Size (bytes)	2	1 – 15	4	4	2
Multi-Processor capable	No	Yes	Yes	Yes	No
Processor Modes	1	3	2	7	7
Data Aligned	Yes	No	Yes	Yes	Yes
MMU	Yes	Yes	Optional	Optional	Optional
Memory Addressing Modes	1	7	2	6	6
Memory Addressing Sizes	32-bit	8,16,32	8,16,32, 64	8,16,32	8,16,32
ISA size	16	332: 138 Integer & Logic, 92 Floating Point	72	53	37
I/O	Memory mapped	Instructions, Memory mapped	Memory mapped	Memory mapped	Memory mapped
Pipeline Length	No Pipeline	Atom: 16 i7: 14 Pentium4: 20-31	Leon3: 7 SPARC64V:15 Ultra-SPARC T2: 8	3	3
Specialities	Very simple ISA, built-in FT	Big ISA and memory operands	Register Window, Delayed Control Transfer	Conditional Instruction Execution	32-bit ARM instructions partly required.

Table 7-4 provides an overview of these hardware architectures and their diverse features. The table is based on data gathered from (Gaisler, 2002; Heise, 2009; Hennessy and Patterson, 2006; Seal, 2000; SPARC International, Inc., 1992).

SPARC and ARM processors are based on a simple Reduced Instruction Set Architecture and therefore more similar to the ERRIC processor, whilst the x86 is based on a Complex Instruction Set with a much larger number of instructions. The table clearly shows the simplicity of ERRIC's ISA with its 16 instructions, which is by large margin smaller than the RISC and CISC architectures. Simple and less powerful instructions come with the cost of longer code compared to the other platforms.

The x86 is a *register-memory* architecture that allows using memory locations directly in instructions. Conversely, ERRIC, ARM and SPARC as *load and store* architectures must first load the argument into a register. The enormous number of instructions of x86 has lead to the situation where the instruction decoder of an Intel Atom CPU occupies more chip's real state than the complete ARM Cortex-A5 (Heise, 2009).

As an example let's examine the load from memory. Prior to the memory access, e.g. in the case array accesses, the absolute memory address must be explicitly calculated and stored in a register. In the SPARC architecture the offset to the base address can first be stored in a distinct register and then added on the fly in the load instruction itself. ARM processors even permit to encode an offset to a base address given in a register directly in the instruction itself.

The instruction set of ARM Thumb is a subset of the standard 32-bit ARM ISA. The Thumb's version targets resource constraint environments where only a 16-bit data bus is available. The address space of the Thumb's is still 32-bit and all registers are 32-bit wide. Nonetheless, while R0 – R7 are directly accessible, R8 – R16 are hidden. ERRIC's ISA, although even more constraint than the Thumb's, is intended to be used as a full instruction set, generic enough to encode all language features. Compared to ARM's, the ERRIC is much simpler and counts with less than half the number of instructions. However, with ERRIC more elaborate instructions need to be emulated, e.g. relative memory accesses or procedure calls. The latter in particular is very efficiently implemented on the ARM by the "load multiple" and "store multiple" instructions, which allows to

efficiently putting all procedure arguments on the stack. In contrast, ERRIC has to individually store all arguments on the stack.

Table 7-5. Supported Adressing Modes

Data addressing mode	*ERRIC*	*x86*	*SPARC v8*	*ARM*	*ERM Thumb*
Register	X	X	(X)	X	X
Register + offset (displacement or based)	-	X	X	X	X
Register + register (indexed)	-	X	X	X	X
Register + scaled register (scaled)	-	X	-	X	-
Register + offset and update register	-	-	-	X	-
Register + register and update register	-	-	-	X	-

Table 7-5 illustrates the different data addressing modes supported by the compared architectures. In some cases, such as SPARC, the architecture does not directly provide an absolute addressing mode. In order to emulate absolute addressing, the SPARC microprocessor uses a register + register mode with a nullified second register. It even provides a register, which is always nullified, so that the absolute addressing emulation does not incur any performance penalty.

$$offset = \begin{Bmatrix} CS: \\ DS: \\ SS: \\ ES: \\ FS: \\ GS: \end{Bmatrix} \begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{bmatrix} + \left[\begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \right] + \begin{bmatrix} None \\ 8-bit \\ 16-bit \\ 32-bit \end{bmatrix}$$

$$offset = Selector: Base + (Index * Scale) + Displacement$$

Figure 7-5. Addressing modes of the x86 architecture

CISC addressing modes are more powerful than RISC ones. Figure 7-5 summarizes the x86 addressing modes. The offset part of a memory address, can be either a static displacement or through an address computation made up of one or more elements. The resulting offset is called effective address and can constitute either positive or negative values except for the scaling factor.

ERRIC provides only absolute memory addressing. Since the addresses must be explicitly computed before the data can be loaded, ERRIC's code requires more instructions.

Table 7-6. Offset Sizes Encoded in instructions

Offset size encoded in instructions (in bits)	*ERRIC*	*x86*	*SPARC v8*	*ARM*	*ERM Thumb*
Unconditional jump call	0	8-32 signed, relative or absolute, direct or indirect	30	24	11
Conditional branch	0	8-32 signed	19	24	8

Table 7-6 compares the offsets sizes that are directly encoded in the instruction. Unless mentioned differently, the offset is always relative to the current instruction pointer. ERRIC does not allow encoding the offset in an instruction. Instead, the offset is always given as an absolute address in a register. Therefore, ERRIC's requires an additional "load constant" instruction, which involves another extra 8 bytes (*Load + NOP + 4 Byte constant*). With regards to safety of code, absolute jumps are desirable to relative jumps. The reason for that is that calculated jumps (relative) are more prone to faults than absolute jumps.

In the following page Table 7-7 presents an overview of the instructions that are required in the compared architectures to be able perform basic operations (such as load, stores, etc.). If an instruction is not available in the ISA, a sequence of instructions is given. In Table 7-7, a "-" sign means that in order to simulate the specific functionality more than a short instruction sequence is required.

All floating-point instructions are omitted, as the ERRIC architecture does not include a floating-point unit and all of them are emulated in software.

Table 7-7. Comparison of Selected Instuctions

Instructions	*ERRIC*	*x86*	*SPARC v8*	*ARM7TDMI (ARMv5-TE)*	*ARM7TDMI Thumb*
Load word	LD	MOV	LD	LDR	LDR
Load byte signed	-	MOVSX	LDSB	LDRSB	LDRSB
Load byte unsigned	LD, LDA, AND-[24]	MOV	LDUB	LDRB	LDRB
Store word	ST	MOV	ST	STR	STR
Store byte	- [25]	MOV	STB	STRB	STRB
ADD	ADD	ADD	ADD	ADD	ADD
ADD (trap if overflow)	-	ADD, INTO	ADDcc, TVS	ADDS, SWIVS	ADD, BVC+4, SWI
Sub	SUB	SUB	SUB	SUB	SUB
Sub (trap if overflow)	-	SUB, INTO	SUBcc, TVS	SUBS, SWIBS	SUB, BVC+4, SWI
Multiply	-	MUL, IMUL	MULX	MUL	MUL
Divide	-	DIV, IDIV	DIVX	-	-
AND	AND	AND	AND	AND	AND
OR	OR	OR	OR	ORR	ORR
XOR	XOR	XOR	XOR	EOR	EOR
NOT	-	NOT	-	-	-
Shift local left	LSL [26]	SHL	SLL	LSL	LSL
Shift local right	LSR [26]	SHR	SRL	LSR	LSR
Shift arithmetic right	ASR[26]	SAR	SRA	-	-
Compare	CND	CMP	SUBcc r0	CMP	CMP
Conditional Branch	CBR	CALL	CALL	BL	BL
Call	CBR	CALL	CALL	BL	BL
Trap	CBR	INT n	TIcc, SIR	SWI	SWI
Return from Interrupt	RETI	IRET	DONE, RETRY, RETURN	MOVS pc, r14	-[27]
NOP	NOP	NOP	SETHI r0, 0	MOV r0, r0	MOV r0, r0

[24] A sequence *LD*, *LDA*, *AND* must be used if the 8-bit data is aligned. Otherwise an LD and a specific number of shift operations must be used

[25] In order to store an 8-bit value the destination address must be loaded, the appropriate bits must be cleared using a bit mask, the argument must be shifted and the written back to memory. It seems clear that omitting the use of 8-bit values would be more efficient

[26] Only one bit

[27] Since Interrupts are always handled in 32-Bit mode and therefore a pure 16-bit Thumb CPU would not support them

7.4. ERA testing and debugging

In order to test the ERRIC processor we must first ensure that the FPGA board and the rest of the elements are working properly. Therefore, testing of the ERA prototype board has been performed via two separate processes: first the testing of the board physical elements and second the functional testing of the soft core.

7.4.1. Testing of the board

First, we test the separate elements of the board using a test case for each element. The test cases employed are written using VHDL and are included in *Appendix A*. The VHDL code is then compiled, using the Quartus II software provided by Altera, to produce a bit stream. The function of the single element of the board is tested by first loading the bit stream file and then by checking the required function. `

As an example, lets examine one of the scenarios corresponding to the tests of the basic functions of read and write of Units 5 and 7 (see U5 and U7 in Table 7-3) SRAM memory modules IS64WV6416BL. It also tests the link between the Unit 1 components (FPGA) and the static memory. What follows is the consequent VHDL code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY eurrica IS PORT (

SW3  : IN STD_LOGIC;
SW4  : IN STD_LOGIC;
        Data_0 : INOUT STD_LOGIC;
        Data_1 : INOUT STD_LOGIC;
        Data_2 : INOUT STD_LOGIC;
        Data_3 : INOUT STD_LOGIC;
        Data_4 : INOUT STD_LOGIC;
        Data_5 : INOUT STD_LOGIC;
        Data_6 : INOUT STD_LOGIC;
        Data_7 : INOUT STD_LOGIC;
        Data_8 : INOUT STD_LOGIC;
        Data_9 : INOUT STD_LOGIC;
        Data_10 : INOUT STD_LOGIC;
        Data_11 : INOUT STD_LOGIC;
        Data_12 : INOUT STD_LOGIC;
        Data_13 : INOUT STD_LOGIC;
        Data_14 : INOUT STD_LOGIC;
        Data_15 : INOUT STD_LOGIC;

        SRAM1_L_CE  : OUT STD_LOGIC;
        SRAM1_L_WE  : OUT STD_LOGIC;
        SRAM1_L_OE  : OUT STD_LOGIC;

        SRAM2_L_CE  : OUT STD_LOGIC;
        SRAM2_L_WE  : OUT STD_LOGIC;
        SRAM2_L_OE  : OUT STD_LOGIC;

        A0:   OUT STD_LOGIC;
        A1:   OUT STD_LOGIC;
        A2:   OUT STD_LOGIC;
        A3:   OUT STD_LOGIC;
        A4:   OUT STD_LOGIC;
        A5:   OUT STD_LOGIC;
        A6:   OUT STD_LOGIC;
        A7:   OUT STD_LOGIC;
        A8:   OUT STD_LOGIC;
        A9:   OUT STD_LOGIC;
        A10:  OUT STD_LOGIC;
        A11:  OUT STD_LOGIC;
        A12:  OUT STD_LOGIC;
        A13:  OUT STD_LOGIC;
        A14:  OUT STD_LOGIC;
        A15:  OUT STD_LOGIC;

        D0:  OUT STD_LOGIC;
D1 : OUT STD_LOGIC);
END eurrica;

ARCHITECTURE behavior of eurrica IS

BEGIN

   D0 <= not SW3;
   D1 <= SW4;

   A0 <= '1';
   A1 <= '0';
   A2 <= '1';
   A3 <= '0';
   A4 <= '1';
```

```
   A5 <= '1';
   A6 <= '0';
   A7 <= '0';
   A8 <= '1';
   A9 <= '0';
   A10 <= '0';
   A11 <= '1';
   A12 <= '0';
   A13 <= '1';
   A14 <= '1';
   A15 <= '0';

sm: PROCESS(SW3)

 BEGIN

if (SW3= '1') then
-- write
   SRAM1_L_CE <= '0';
   SRAM1_L_WE <= '0';
   SRAM1_L_OE <= '0';

   SRAM2_L_CE <= '0';
   SRAM2_L_WE <= '0';
   SRAM2_L_OE <= '0';
   Data_0  <= SW4;
   Data_1  <= SW4;
   Data_2  <= SW4;
   Data_3  <= SW4;
   Data_4  <= SW4;
   Data_5  <= SW4;
   Data_6  <= SW4;
   Data_7  <= SW4;
   Data_8  <= SW4;
   Data_9  <= SW4;
   Data_10 <= SW4;
   Data_11 <= SW4;
   Data_12 <= SW4;
   Data_13 <= SW4;
   Data_14 <= SW4;
   Data_15 <= SW4;
else
-- read
   SRAM1_L_CE <= '0';
   SRAM1_L_WE <= '1';
   SRAM1_L_OE <= '0';

   SRAM2_L_CE <= '0';
   SRAM2_L_WE <= '1';
   SRAM2_L_OE <= '0';
end if;

END PROCESS;
END Behavior;
```

Table 7-8. Test results of reading and writing functions of U5 and U7 SRAM memory modules and their interconnecting elements

Net		EP2C20Q240C8	Tests			
U5/U7	DATA_0	155	0	0	0	+4v
	DATA_1	156	0	0	0	+4v
	DATA_2	157	0	0	0	+4v
	DATA_3	159	0	0	0	+4v
	DATA_4	161	0	0	0	+4v
	DATA_5	162	0	0	0	+4v
	DATA_6	164	0	0	0	+4v
	DATA_7	165	0	0	0	+4v
	DATA_8	166	0	0	0	+4v
	DATA_9	167	0	0	0	+4v
	DATA_10	168	0	0	0	+4v
	DATA_11	170	0	0	0	+4v
	DATA_12	171	0	0	0	+4v
	DATA_13	173	0	0	0	+4v
	DATA_14	174	0	0	0	+4v
	DATA_15	175	0	0	0	+4v
U5/U7	ADDR_0	8	+4v	+4v	+4v	+4v
	ADDR_1	9	0	0	0	0
	ADDR_2	11	+4v	+4v	+4v	+4v
	ADDR_3	13	0	0	0	0
	ADDR_4	14	+4v	+4v	+4v	+4v
	ADDR_5	15	+4v	+4v	+4v	+4v
	ADDR_6	16	0	0	0	0
	ADDR_7	18	0	0	0	0
	ADDR_8	20	+4v	+4v	+4v	+4v
	ADDR_9	21	0	0	0	0
	ADDR_10	37	0	0	0	0
	ADDR_11	38	+4v	+4v	+4v	+4v
	ADDR_12	39	0	0	0	0
	ADDR_13	41	+4v	+4v	+4v	+4v
	ADDR_14	42	+4v	+4v	+4v	+4v
	ADDR_15	44	0	0	0	0
U5	SRAM1_L_CE	233	0	0	0	0
	SRAM1_L_WE	232	+4v	+4v	0	0
	SRAM1_L_OE	231	0	0	0	0
U7	SRAM2_L_CE	230	0	0	0	0
	SRAM2_L_WE	228	+4v	+4v	0	0
	SRAM2_L_OE	226	0	0	0	0
	LED1	125	off	off	on	on
	LED2	178	on	off	on	off
	Switch_3	7	0(off)	0(off)	1(on)	1(on)
	Switch_4	119	0(off)	1(on)	0(off)	1(on)
Results (PASS)			✓	✓	✓	✓

The link configuration is shown on the second column of Table 7-8. The SW3 is to control write data and read data into memory. SW4 specifies the data to test. When SW3 is off ('0'), it performs reading operation. When it is on ('1'), it is to perform writing operation. The address of the SRAM for testing is specified by (A0-A15). The LED1 and LED2 indicate the correct operation of SW3, and SW4. The data input and output from the SRAM (U5, and U7) is check by Voltage meter. Voltage "0" means represents '0' logic state, and Voltage "+4" represents a '1' logic state. Initially, all the memory is set to '0'. The testing is to change the settings of SW3, and SW4 and check against the input and output voltages from the data line of U5, and U7. If they are matching, then the test is passed, and the links are correct and the memory modules are performed required functions.

The first column on Tests is checking memory reading with initial value of '0'. The Voltage on the data line shows the correct results '0'. Then the second column is to change the input value to '1' (SW4=1), because the write control is not changed, so the result should still be '0'. The output is correct and the data still keep on '0'. The third column is to write '0' into the memory (SW3 is setting to write and SW4 is setting to 0) and the result shown on data lines are correct, and fourth column is to write '1' (SW3 is setting to write and SW4 is setting to '1') into the memory modules. The output should be "+4V" on data line. The measures by voltage meter show the correct results.

Once every element of the board has been tested we assume that the board elements are working correctly.

7.4.2. Functional testing of the ERRIC processor

The next step is to confirm that ERRIC processor is working correctly. Similarly to the testing of the board, the testing of the processor is based in several test cases. Again, since the ERRIC processor can only process binary data the test cases consists of bit stream files. However, instead of VHDL, the functional test cases are obtained by using an assembler on an ERRIC's specific pseudo code. A detailed explanation of such assembler is provided in the following section 7.5.

There are several unit test cases for the 16 different instructions of the ERRIC processor. The resulting bit stream file from the assembling process is loaded into the FPGA through the JTAG interface. Then the codes are executed. The input is the number controlled by push bottom of the Altera board and the calculation results is shown out on 7 segment indicators.

What follow is an example of a test case for a specific SUB instruction:

```
-- testing for SUB
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :   00000000000000000000000000000000;  --0        ------------
        01  :   00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :   00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :   01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :   00011000011000110000010000100011;  --4  SUB R3 R3 ; LD (R1) R3
        05  :   00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :   00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :   00000000000000000000000000000100;  --7    --4-- Address
        08  :   00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :   00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :   00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :   00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :   00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :   01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :   00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :   00000000000000000000000000001011;  --15   --    B
        10  :   00000000000000000000000000001010;  --16   --    A
        11  :   00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :   11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :   00100000000000000000000000000000;  --19 --     Mask(>)
        14  :   00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :   00000000000000000000000000000010;  --21 --     2
        16  :   00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :   00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :   00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :   00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :   00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :   00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :   00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :   00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :   10001110100000000010000000000000;
        [1f..2d]  :   10101110000000000010000000000000;
        2e  :   11011110000000000010000000000000;
        [2f..3a]  :   01011000100000000010000000000000;
        3b  :   11011110000000000010000000000000;
        [3c..3d]  :   01111000100000000010000000000000;
        3e  :   00110100000000000010000000000000;
        3f  :   11111111111111110010000000000000;
END;
```

The function of this sequence of code is a loop to continue loading input data, do the SUB calculation, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the

input data, do the SUB operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture (Figure 7-6) is shows both the simulation results and the physical testing results. It performs a 3-3 operation, the subtract function gives the results 0. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows in the square on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

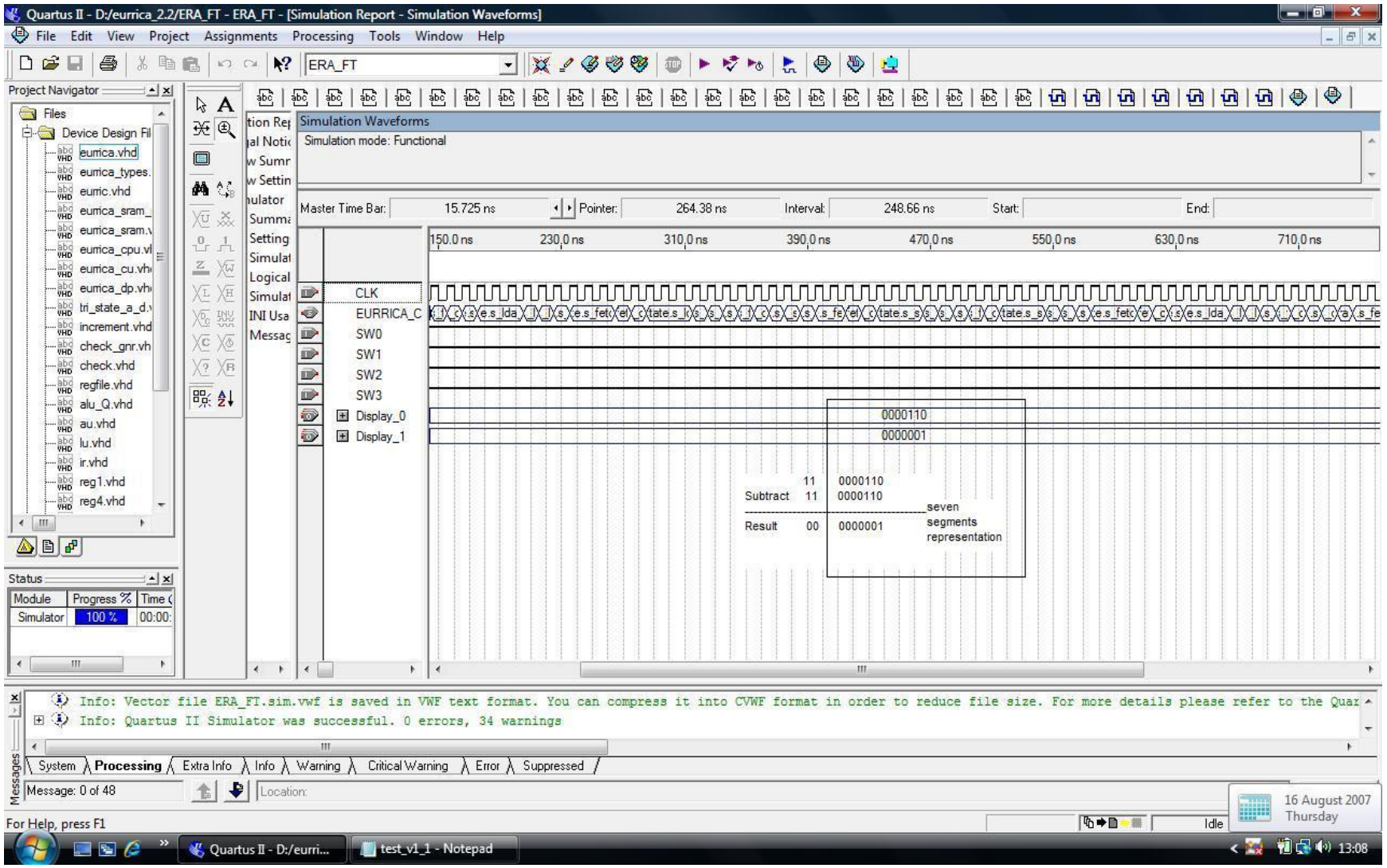


Figure 7-6. Simulation Results of Unit Test of the SUB instruction using Quartus II Simulator

7.5. ERA's assembler

The system software for ERA assembler-level programming consists of the following components:

- An **Assembler** that performs the compilation of source programs written in ERA assembler into executable or object code.
  - Program name: *Assembler.exe*
  - Call form: *assembler [options] source.era*
  - Result: *source.obj* or *source.code*
- A **Linker** that takes several files with object code as input and produces the single result object or executable file depending on the existence of external references in the result.
  - Program name: *Linker.exe*
  - Call form: *linker [options] [entry_point] source1.obj…sourceN.obj*
  - Result: *source1.code*
- A **Runner** that takes an ERA executable file as input, loads it into the memory (using the interface of the **Model** component) and executes it in the simulation mode.
  - Program name: *Runner.exe*
  - Call form: *runner [options] source.code*
  - Result: an output on the console or printing device
- A **Preparator**, an extra component which supports transition from the model to the real ERA board. The Preparator takes the ERA executable file as input and produces the pure binary file which is completely ready to load to the real memory.
  - Program name: *Preparator.exe*
  - Call form: *preparator source.code*
  - Result: *source.bin*

The overall configuration of the assembler (except the Preparator) is shown on the picture below.

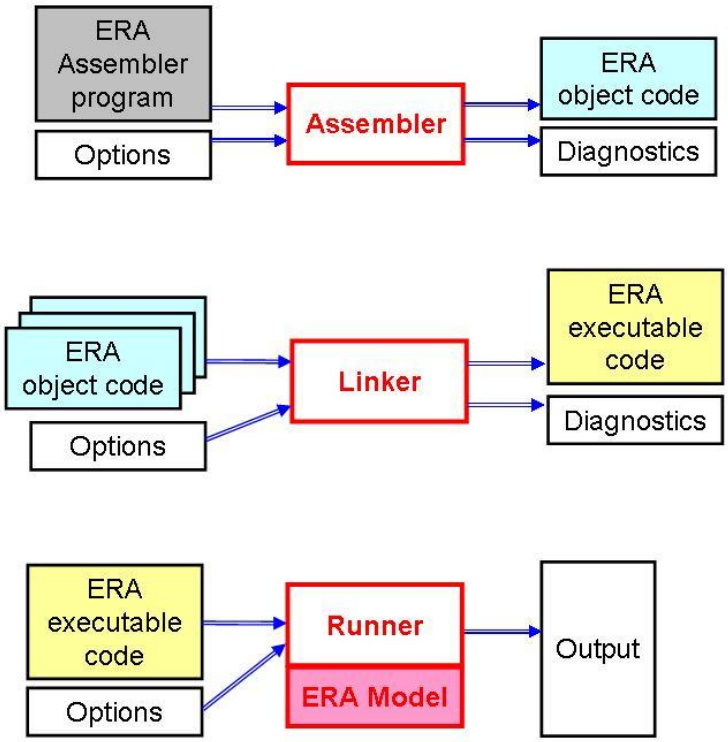


Figure 7-7. Flow of ERRIC testing (top) and flow of ERRIC testing with the help of a disassembler

As an example, below is the source code of a program implementing the simple in-place sorting algorithm. The program is written in the ERA assembler. The assembler's syntax for every ERA instruction was described in 7.2. The sorting algorithm itself is specified in a Pascal-like language like as follows:

```
procedure Sort ( a : array of int; size : int )
begin
    for i := 1 to size
       for j := i to 0 by -1
          if a[j-1] < a[j]
          then
             w := a[j-1];
             a[j-1] := a[j];
             a[j] := w;
          end
       end
    end
end Sort;
```

Additional remarks concerning ERA assembler syntax:

- For debugging purposes two pseudo-instructions have been added to the ERA assembler: DATA and TRACE. DATA instruction just denotes literal data which will go directly to the object code. TRACE instruction causes registers specified in the instructions to be output to the console or to a printing device.
- Labels are specified as identifiers enclosed in angle brackets. The value of a label is the address of the instruction or data immediately following the label.
- Comments have the form // sequence of any characters until end of line

The ERA program implementing the sorting algorithm looks like as follows:

```
// i: R1;  j: R2;  intermediate values: R3, R4, R5

    R1 := 1;        // i := 1
<LoopOuter>
    R3 := Size;
    R3 := *R3;      // R3: Size
    R4 := 1;
    R3 -= R4;
    R3 ?= R1;       // Compare Size-1 and i
    R4 &= R3;       // Extract > sign using R4=1 as mask for >
    R3 := OutOuter;
    if R4 goto R3;  // if i>Size goto OutOuter

    // Organize inner loop
    R2 := R1; NOP;  // j := i
<LoopInner>
    R3 := 0;        // w := 0
    R3 ?= R2;       // Compare j with 0
    R4 := 4;        // Mask for equality
    R3 &= R4;       // Extract equality sign
    R4 := OutInner;
    if R3 goto R4;  // if j=0 exit the inner loop

    // Otherwise, compare two array elements
    // to decide if we need to exchange them.
    // R10: address of j-th element
    // R11: a[j]
    // R12: a[j-1]
    R10 := Array;
    R10 += R2;      // array base address+j
    R11 := *R10;    // R11 := a[j]
    R12 := R10;
    R13 := 1;
    R12 -= R13;     // a+j-1
    R12 := *R12;    // R12 := a[j-1]

    R3  := R12;     // w := a[j-1]
```

```
    R3  ?= R11;     // Compare a[j-1] and a[j]
    R4  := 5;       // Mask for >=
    R4  &= R3;      // Extract > and = signs
    R3  := OutExchange;
    if R4 goto R3;  // if a[j-1] >= a[j] do not perform exchange

    // Otherwise, perform exchange
    R3 := R10;
    R4 := 1;
    R3 -= R4;       // R5: address of (j-1)th element
   *R3 := R11;      // a[j-1] := a[j]
   *R10:= R12;      // a[j] := a[j-1]
<OutExchange>
    // Decreasing j (inner loop)
    R3 := 1;
    R2 -= R3;       // j := j-1
    R4 := LoopInner;
    if R3 goto R4;  // goto LoopInner
<OutInner>
    // Increasing i (outer loop)
    R3 := 1;
    R1 += R3;       // i := i+1
    R4 := LoopOuter;
    if R3 goto R4;  // goto LoopOuter
    NOP;

<OutOuter>
    R15 := Size;
    R15 := *R15;
    R16 := Array;
    TRACE R15,R16;
    STOP; NOP;
<Size>
    DATA  20
<Array>
    DATA 537
    DATA 242
    DATA 114
    DATA 436
    DATA 337
    DATA 296
    DATA 285
    DATA 655
    DATA 639
    DATA 436
    DATA 912
    DATA 520
    DATA 624
    DATA 551
    DATA 600
    DATA 741
    DATA 612
    DATA 943
    DATA 871
    DATA 735
```

Here is the screen snapshot demonstrating the compilation and execution process for the sorting example shown above.

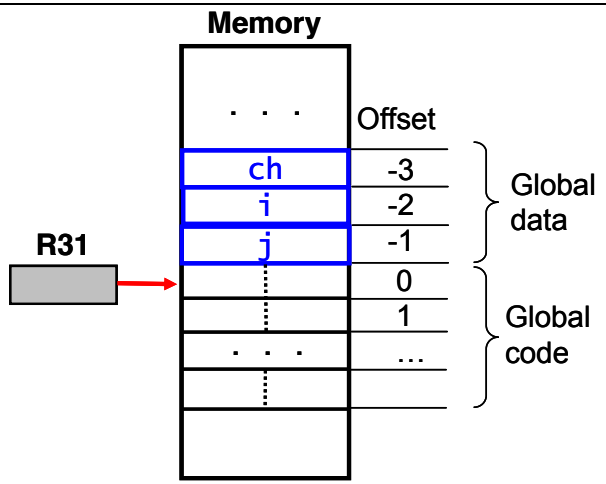
```
C:\Z\ERA Demo>assembler sort.era
ERA Assembler, Version 1.0.0.0 of 26 March 2014, 13:57:19
ERA Model, Version 1.0.0.0
Copyright (c) London Metropolitan University, 2013
source file 'sort.era'is being assembled
assembling is successfully completed

C:\Z\ERA Demo>runner sort.code
ERA Model, Version 1.0.0.0 of 26 March 2013, 14:10:37
Copyright (c) London Metropolitan University, 2013
code file 'sort.code'is being executed
537 242 114 436 337 296 285 655 639 436 912 520 624 551 600 741 612 943
871 735
114 242 285 296 337 436 436 520 537 551 600 612 624 639 655 735 741 871
912 943
execution is successfully completed


C:\Z\ERA Demo>
```

Another example of a simpler program is illustrated in Table 7-9. The table shows the location of data variables and code within the memory structure together with an explanation of the specific line of code and their effect. The *R31* register, set by the program loader, always keeps the base address of the global data and the program code. The register uses negative offsets for the data and non-negative offsets for the code and local data. More examples on how the assembler transforms source code into machine code can be found in Appendix C.

Table 7-9. Example of code trnsformed into assembly code by the assembler

Example 1.
Global data and code

Source code	Memory structure	Code	Assembler. Code	Comments

`char ch;`
`short int i;`
`int j;`
				**Programming convention 1:**
				
				R31register always keeps the base address of the global data (with negative offsets) and the program code (with non-negative offsets).
				
				Initially R31 is set by the progam loader.

`ch := '0';`		`LDA R1` | `NOP`
		`'0'`
		`LDA R2` | `ADD R31,R2`
		`-3`
		`ST R1,R2`
			`R1 := '0';`
			`R2:=-6;`
			`R2+=R31;`
			`*R2:=R1;`
				Get the value of '0' into R1
				Get the address of ch into R2 (as R31+offset)
				Store the value from R1 to ch (pointed to by R2)

`i := 10;`		`LDA R1` | `NOP`
		`10`
		`LDA R2` | `ADD R31,R2`
		`-2`
		`ST R1,R2`
			`R1:=10;`
			`R2:=-4;`
			`R2+=R31;`
			`*R2:=R1;`
				Get the value of 10 into R1
				Get the address of i to into R2 (as R31+offset)
				Store the value from R1 to i (pointed to by R2)

`j := i;`		`LDA R1` | `ADD R31,R1`
		`-1`
		`LDA R2` | `ADD R31,R2`
		`-2`
		`LD R2,R2` | `ST R2,R1`
			`R1:=-2;`
			`R2:=-4;`
			`R2+=R31;`
			`R2:=*R2;`
			`*R1:=R2;`
				Get the offset of j into R1; get the address of j into R1 (as R31+offset)
				Get the offset of i into R1; get the address of I into R2 (as R31+offset)
				Get the value pointed to by R2 (i.e., i) to R2. Store the value from R2 to j (pointed to by R1)

7.6. ERA's simulator: Dissimera

Reading binary code is a painful experience. In order to test and troubleshoot any error of design, bug or incompatibility between the assembler and the VHDL code, a new tool has been develop: a Disassembler and a Simulator in a combined tool that will ease this process. In addition it will allow the simulation of the state of the processor at any given time. Dissimera's main goal is to simulate the basic core features of ERA in a reliable and accurate manner.

The fundamental characteristics of this tool are:

- Disassembling of instructions: Binary-to-ASM and Binary-to-PSEUDOCODE that will complement the assembler.
- Ability to discern data from instructions
- Simulation of the ERA architecture including: Program Counter (PC), Instruction Registry (IR), Register FILE (RF) and memory contents.
- Step-by-Step Execution.
- Breakpoints.
- Overflow warning
- Logging.
- Ability to compare results of simulation execution with the results of Altera execution.

7.6.1. Architecture

The main two functions of Dissimera, Disassembling and Simulation, are embedded into a single software product. The architecture of such software is based on three main modules: the Interface module, the parsing module and the simulation module. The programming language used to implement those is ANSI C and currently targets 80x86 machines.

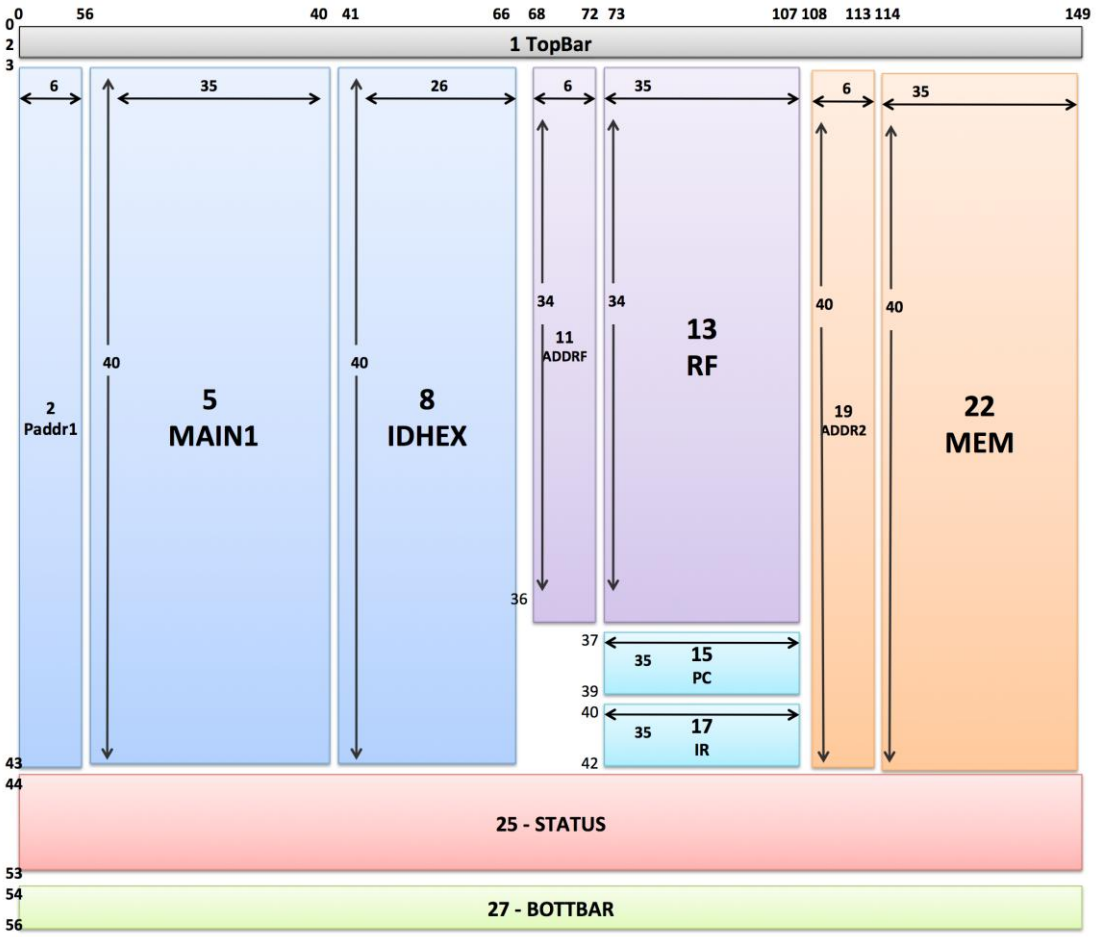


Figure 7-8. Design of the Interface of the current version of the simulator

The Dissimera disassembler reads the output of the assembler introduced in 7.5. Dissimera uses such output, in binary code, as an input, to start the simulation process.

The ***Interface module (IM)*** is based on an *NCurses* API with MIT license to implement the interface under a Windows Console. The IM is built independently from the Dissimera's engine (composed by the parsing and simulation modules) with the intention of improving scalability. It is integrated in a way that escalating to a newer interface or migration to a different operating system will not be problematic.

Figure 7-8 illustrates the different elements of the current design of Dissimera's interface. The current version is based on a single ERRIC's processor with a 32-

bit mode. The *Paddr1* and *MAIN1* elements contain the addresses and binary contents of the 32-bit memory unit that contains the running code. *Paddr2* and *MEM* also contain a copy of the addresses and binary content but allow the user to browse the memory contents during and after execution and check the program results. *IDHEX* shows the Instructions in assembler or the data in hexadecimal numbering. *ADDRF* and *RF* contain the name and value of the 32 registers that compose the register file. *PC* is the program counter or instruction pointer. The Instruction Register or *IR* stores the 32-bit value with the decoded instructions that are about to be executed. The status of execution and a log with extra details is shown in the STATUS element. An example of log execution can be seen in 7.6.2.

The ***parser module (PM)*** involves three different processes. The first process is the *lexical analysis* by which the input binary code is fragmented into meaningful symbols (tokens) in the context of ERA's pseudo-code language. The next process is the *syntactic analysis* of these tokens that define allowable expressions according to the rules of a formal grammar, based on the ISA format introduced in 7.2. . Finally, a *semantic analysis* works out the implications of the validated tokens and takes appropriate action.

These three processes need extra attention. How can we determine the type of a specific value? i.e. How can Dissimera be certain that a 16-bit binary value is either code or data?

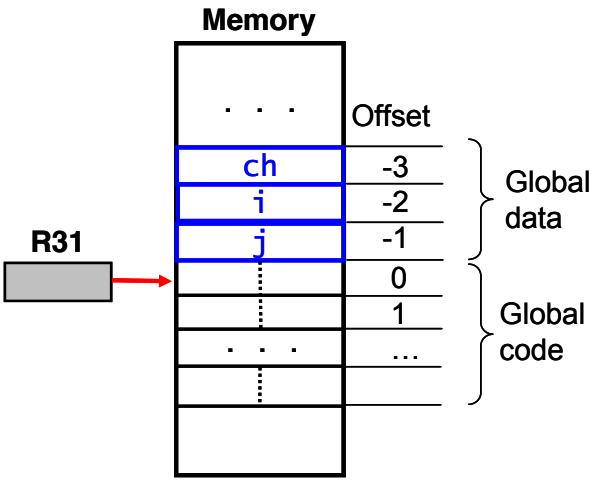


Figure 7-9. Memory allocation of a program in ERA

Figure 7-9 depicts what we already mentioned, the program loader sets R31 with the base address of the global data (negative offsets) and program code (positive offsets). However, in the case of local data, by just examining a single binary value it is not possible to determine its type. E.g.: According to ERA's ISA rules, a *1100010011100001* value can be interpreted:

- as a *C4E1h* data value, or
- as an *LD R7 R1* instruction that loads the 32-bit value of the memory address contained in R7 into the register R1 (i.e.: R1:=*R7).

The type of a value is determined by its context. Dissimera uses the execution context to determine that, by performing two top-down runs before the assembly code is presented on the screen. During the first run the code is fragmented and the starting point of execution is determined. The determination of type takes place in the second run. Dissimera proceeds to silently execute instruction-by-instruction, marking the values as code tokens and decoding them consecutively performing the appropriate jump instructions and following the Program Counter. Once all the code lines are executed the rest of the tokens are marked as local data tokens. This second run also includes error detection mechanisms for bad syntax and buffer overflow. Note that both runs are transparent in terms of user interface. Once the parsing module has finished these processes, the UI contains now the results of disassembling and the simulation process can start.

The ***simulation module (SM)*** is in charge of program execution. This stage benefits from the two previous runs using the output of the parsing stage as an input. Hence, the SM is able to differentiate from code tokens and local and global data tokens.

Below, Figure 7-10 presents a screenshot of the current version of Dissimera and the IM, PM and SM modules.

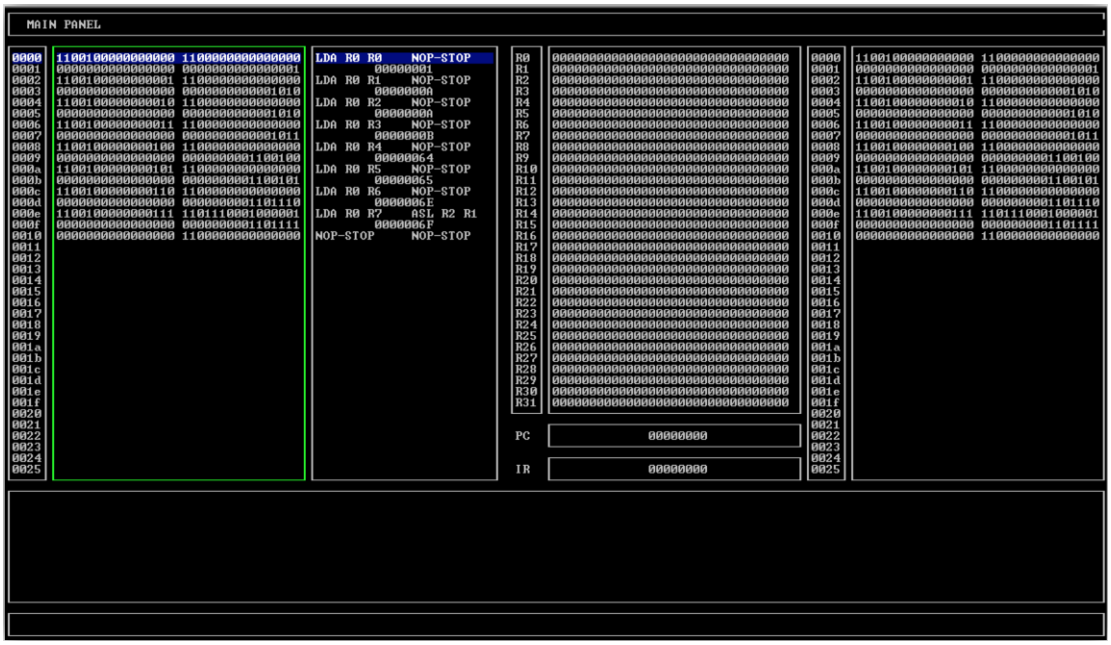


Figure 7-10. Screenshot of the current version of Dissimera

Initially, the execution starts at the address set by program counter (loaded with the content of R31). The *PC* holds the memory address of the next instruction to be executed and is incremented just after fetching the 32 bits from memory containing two instructions. After the processors fetches the memory location stored in the PC, the instruction is loaded in the Instruction Register (*IR*). The instructions are fetched sequentially from memory unless a *CBR* instruction changes the sequence placing a new value in it. Dissimera offers several run modes:

- *Normal Mode*: It is the standard mode. The simulator continuously executes instructions of a program until a STOP instruction is found.
- *Debugging Mode*: which includes the ability to place breakpoints within the code and the ability to perform step-by-step execution. In addition, this mode allows for step-back execution. Simulation can return to a previous state. All this features benefit the debugging of the system.

Dissimera can be used as a tool for analysis and debugging of ERA programs, and more importantly, as a tool for testing and debugging of the hardware architecture. Figure 7-11 depicts two different testing methodologies for ERRIC. Initially, several test programs are developed using the pseudo-code language introduced in 7.5. The *assembler* is then used to obtain the tests programs in binary code compatible with ERRIC.

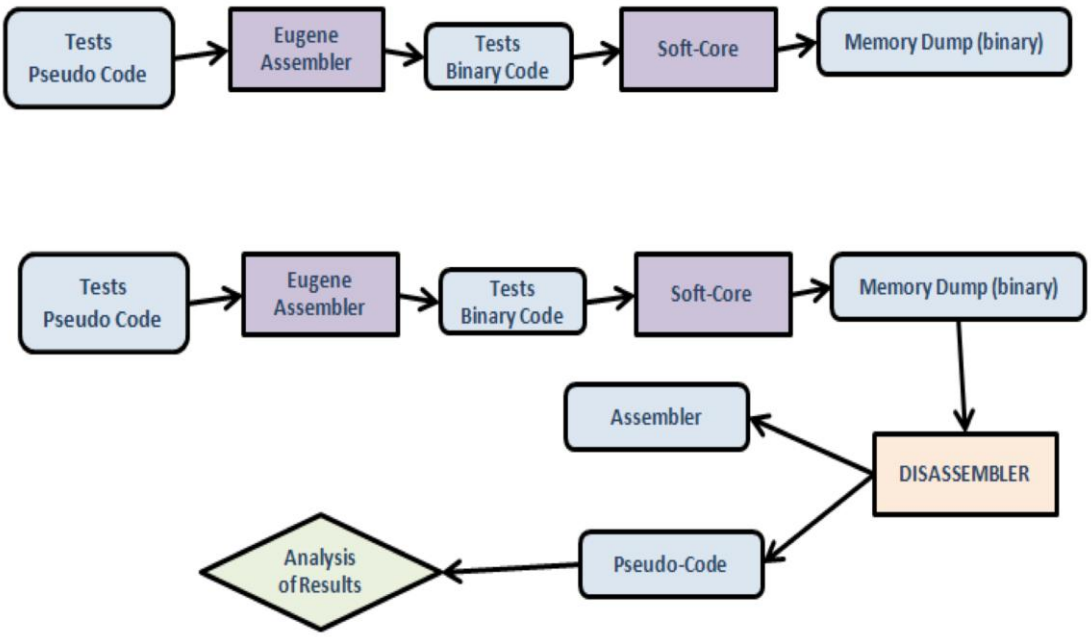


Figure 7-11. Flow of ERRIC testing (top) and flow of ERRIC testing with the help of a disassembler

These tests programs are introduced into ERRIC's soft-core processor through the JTAG Interface of ERA's board. The tests programs are executed with the Quartus II Software and upon execution a memory dump with the results of execution is produced.

After the Soft Processor simulation, the same initial tests programs generated by the assembler are introduced into Dissimera together with the memory dump of execution results produced by Quartus II. Here, the PM of Dissimera performs the disassembling of the binary code. After full execution of the ERA program by the SM, using the Dissimera's IM, the memory dump of the MEM element can be compared and analysed with the previous results of Quartus II. In case of

mismatch, debugging via step-by-step execution of both simulators can help in the detection and location of design and implementation errors.

In the following page, Figure 7-12 Figure 7-13 show the caller graph of Dissimera's main function. A full documentation on the current version of this software, including the implementation details of the IM, PM and SM, libraries used and dependency graphs, together with the source code are included in Appendix D.

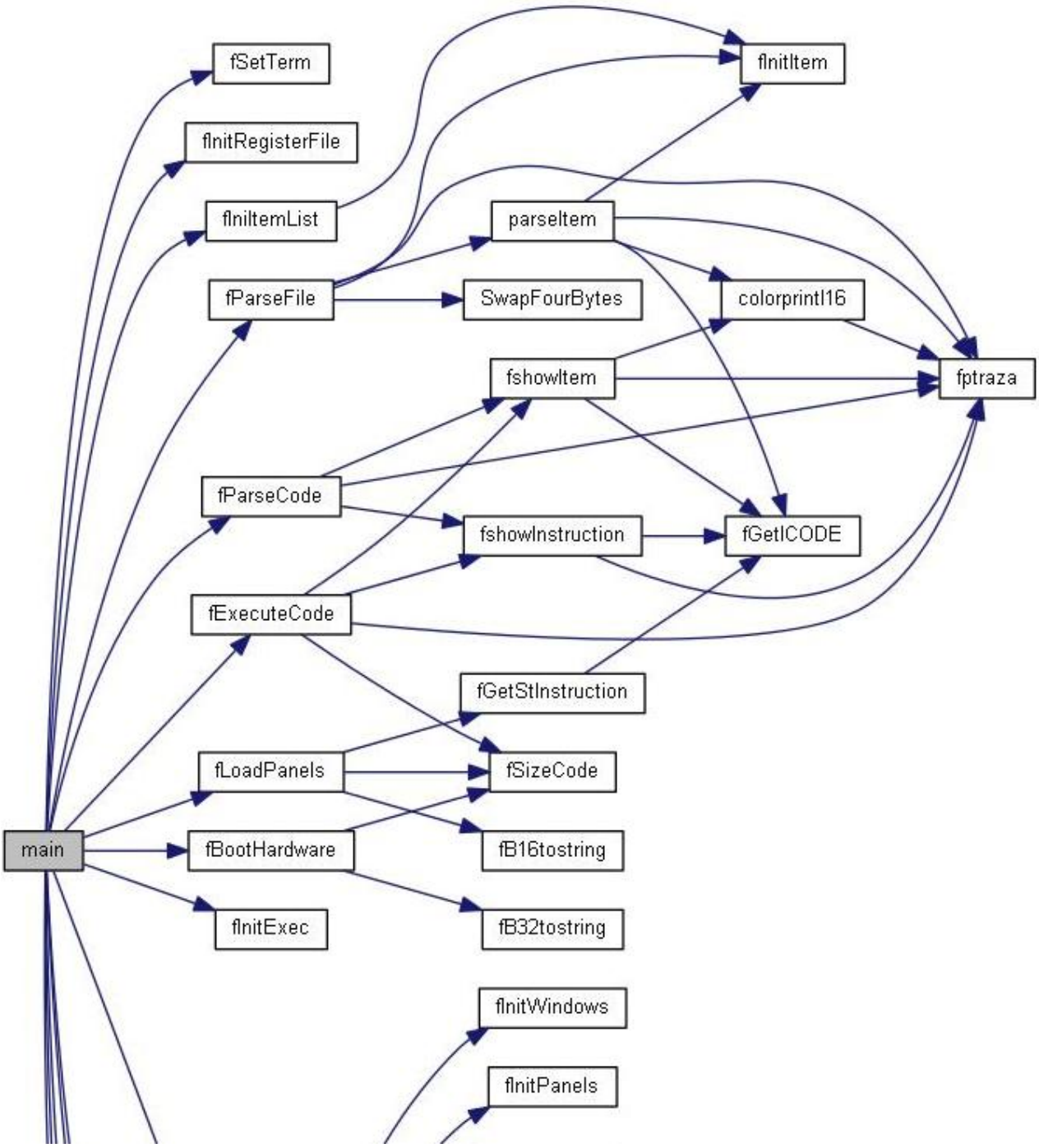


Figure 7-12. Caller Graph of Dissimera's main function 1/2

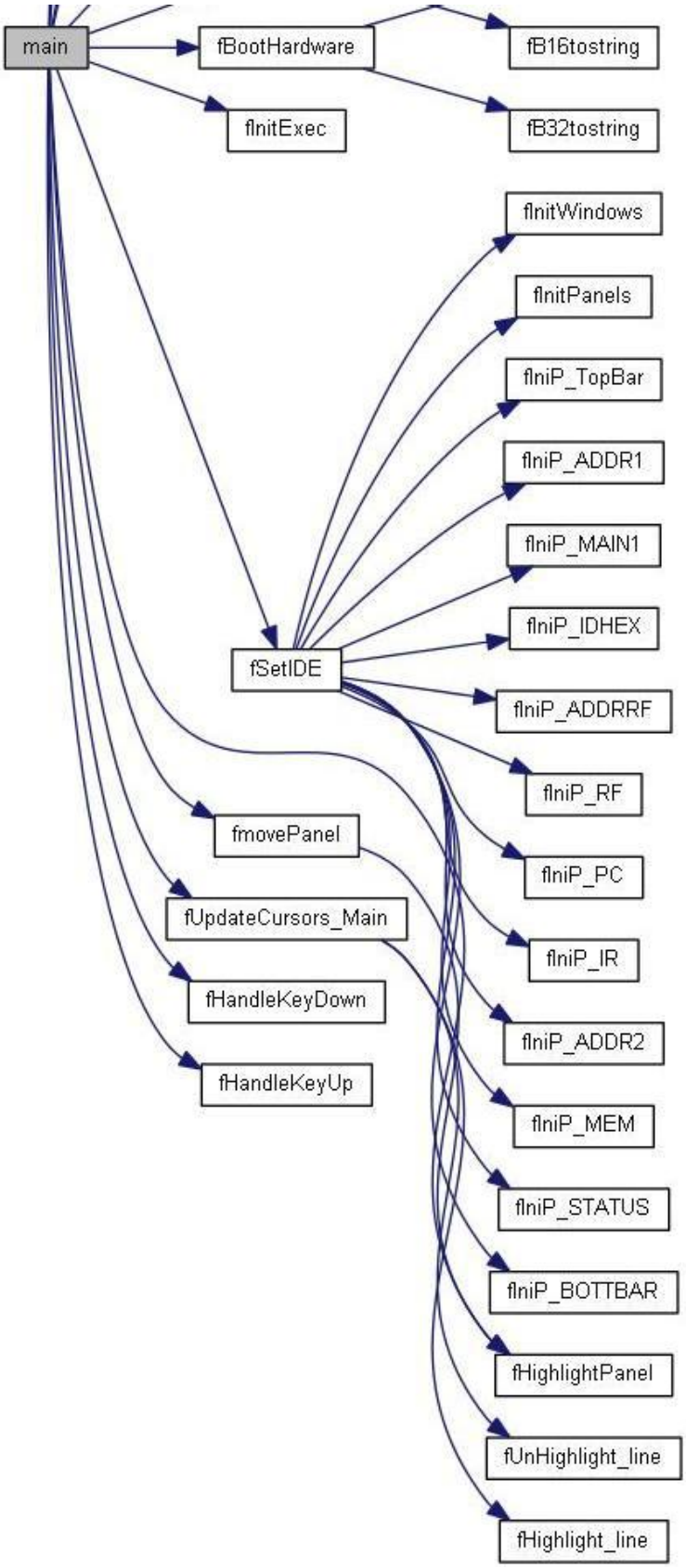


Figure 7-13. Caller Graph of Dissimera's main function 2/3

7.6.2. Disassembler Log Sample

A Dissimera log is divided in four-line units. What follows is an example of a Log File related to the execution of the disassembler. Each unit represents information on 32-bit. The first line of each unit shows the instruction memory address and the hexadecimal value of each 16-bit half. The second line contains the binary values of each half. The third line and fourth lines include information on the assembly instructions of the first and second half respectively, and their pseudo-code meaning.

```
        fname = test.txt
        Number of Bytes = 80 bytes
        Number of 32-bit instruction-data = 20

1       C800          C000
 11 0010 00000 00000   11 0000 00000 00000
  LDA R0 R0  meaning R0:=CONSTANT   meaning  R0:=1    meaning  R0:=1

3       C801          C000
 11 0010 00000 00001   11 0000 00000 00000
  LDA R0 R1  meaning R1:=CONSTANT   meaning  R1:=10   meaning  R1:=10

5       C807          C000
 11 0010 00000 00111   11 0000 00000 00000
  LDA R0 R7  meaning R7:=CONSTANT   meaning  R7:=111   meaning  R7:=111

7       C807          C4E0
 11 0010 00000 00111   11 0001 00111 00000
  LDA R0 R7  meaning R7:=CONSTANT   meaning  R7:=111   meaning  R7:=111

9       C4E1          C802
 11 0001 00111 00001   11 0010 00000 00010
  LD R7 R1  meaning R1:=*R7    meaning  R1:=-939014944
  LDA R0 R2  meaning R2:=CONSTANT   meaning  R2:=10   meaning  R2:=10

11      C803          C000
 11 0010 00000 00011   11 0000 00000 00000
  LDA R0 R3  meaning R3:=CONSTANT   meaning  R3:=11   meaning  R3:=11

13      C804          C000
 11 0010 00000 00100   11 0000 00000 00000
  LDA R0 R4  meaning R4:=CONSTANT   meaning  R4:=100   meaning  R4:=100

15      C805          C000
 11 0010 00000 00101   11 0000 00000 00000
  LDA R0 R5  meaning R5:=CONSTANT   meaning  R5:=101   meaning  R5:=101

17      C806          DC41
 11 0010 00000 00110   11 0111 00010 00001
```

```
  LDA R0 R6  meaning R6:=CONSTANT   meaning  R6:=110    meaning  R6:=110

19        CC07          CC22
 11 0011 00000 00111   11 0011 00001 00010
  ST R0 R7  meaning *R7:=R0   meaning  MEM[111]=1    meaning  MEM[111]=1
  ST R1 R2  meaning *R2:=R meaning  MEM[10]=-939014944  meaning  MEM[10]=-939014944

20        D0A4          0000
 11 0100 00101 00100   00 0000 00000 00000
  MOV R5 R4  meaning R4:=R5 meaning R4:=101
  NOP-STOP R0 R0  meaning STOP instruction
```

7.7. Conclusion

In the development of reliable architectures there is a need for providing accurate testing and debugging of hardware and software. In this chapter we first show the implementation details of hardware architecture that are relevant for simulation.

We show ERRIC's instruction execution flow and explained how it can achieve the decoding and execution of two instructions per single fetch and its implications on system compilers.

We also provide an overview of the hardware prototype and its memory mapping together with an architectural comparison to other relevant RISC and CISC architectures. We introduced Minimal Instruction Set Architecture that simplifies the instruction decoder design and the overall system's reliability. ERA's ISA has 16 instructions, does not have a pipeline and provides only absolute memory addressing. We argue how far from being a drawback, this simplicity is sufficient to perform safety-critical code improving efficiency reliability.

We provide the details of a testing and debugging methodology of a hardware prototype. Finally we showed the features and implementation of an assembler, and a disassembler/simulator as a proof of concept of the architecture. These custom tools are useful, not only for testing and debugging of the hardware prototype, but for the system and application software.

Chapter 8

Chapter 8

Conclusion

Initially, the main objective of this research was to find new ways to improve the fault tolerance of current architectures. We first reviewed the classic theories of reliability and fault tolerance and found that a) the more components a system has the higher the probability of system failure and b) the reliability of a system is often dominated by the reliability of its least reliable component. We concluded that some of the keys to improve reliability would be simplicity of implementation and careful introduction of redundancy.

The terms Reliability, Fault-Tolerance and Dependability do not cover all the attributes of safety-critical applications or, after being redefined over the years, are ambiguous. As a consequence, in Chapter 2 we provided a novel concept of Resilience that encompasses several attributes adapting them to the safety-critical domain. A resilient system, over a specified time interval, under specified environmental and operating conditions (performability), "must be ready" (in terms of availability) to perform its intended function (reliability), guaranteeing the absence of improper system alterations (integrity). It must have the ability to conduct servicing and inspections (testability) so that in case of failure quick restoration to a specified working condition must be achieved (maintainability) can be provided or can discontinue its operation in a safe way (safety).

Furthermore, a resilient system must have the ability to anticipate changes and evolve (evolvability) while executing (adaptability), successfully accommodating changes by reconfiguring elements of the system if necessary (reconfiguration).

Since one of the keys to improve resilience was the careful choice of redundancy and the manner in which this should be applied, we decided to review the different types of redundancy and how such redundancy is translated into functional mechanisms to either avoid or tolerate faults. In Chapter 3 we provided a full classification of fault-tolerant mechanisms based on the type of redundancy employed and study their benefits and drawbacks. Fault avoidance techniques do not guarantee complete removal of faults and present drawbacks such as cost, speed of operation and chip's area. Therefore, fault tolerance mechanisms are needed to further improve the resilience of safety-critical systems. In order to select a specific set of redundancy techniques for the implementation of FT we should first define the different requirements of the particular application. Once the domain and requirements are defined we should select the techniques that are more suitable for such requirements and the level at which the redundancy should be applied.

We realised that in order to improve the existing mechanisms, before researching what it is required from them, we should study and analyse how failures originate, what causes them, under what circumstances, in which contexts, and how often they happen. In Chapter 2 we introduced the concept of vicious cycle that explains our interpretation of the reasons behind the performance and reliability problems that jeopardize the continuation of Moore's Law. We also reviewed the fault-failure lifecycle and defined the necessary concepts of fault, error, failure and catastrophic failure.

Since the majority of hardware faults in current electronics are induced by ionizing radiation we studied the damage mechanisms at the physical level, the sources of error and the micro- and macro- effects of such mechanisms. . As a result Chapter 4 provides an extensive taxonomy of radiation effects describing their nature, type of degradation, susceptibility, fault rate trends and

recoverability. From the study of this taxonomy we conclude that as we moved to denser technologies at lower voltages, system SER will continue to rise and in particular the contribution of SEU, SET, MBU and SEFI will increase. We also conclude that current mitigation techniques are not efficient when dealing with certain types of SEE and/or with the upcoming rates.

In Chapter 5 we explained how any fault tolerant system involves a Model of the System, a Model of Faults and a Model of Fault Tolerance. Consequently, we add value to such system by developing a comprehensive Fault Model suggesting methods for recognition and reaction against faults. We discuss fault manifestation, detectability diagnosability and recoverability and propose adequate solutions for diagnosis and recovery. We have introduced the principle of reconfiguration of the system and how this might be used for various purposes: performance, reliability and energy wise gain, improving the efficiency of resilience. In addition, we introduced GAFT and extend it by providing the different states and actions required to achieve fault tolerance and therefore improve system resilience.

In Chapter 6, using know-how and conclusions acquired in the previous chapters we introduced a hybrid HW-SSW co-design approach of a resilient architecture with the ability to reconfigure, achieving various levels of dependability in different environments. As part of the architecture, we first introduced the syndrome as a new property of the system and analysed it as a process and as a tool for reconfiguration that can provide efficiency of reliability, performance and power consumption.

We also introduced the ERRIC's microprocessor and the ERA architecture defining their active, passive and interfacing zones of information processing. We keep the redundancy level needed to implement fault tolerance, as low as possible. With regards to the active zone, the instruction set and its implementation are reduced to the minimum; coprocessors, pipelining and floating-point units are removed which simplifies the processor design and reduces the complexity and fault rates. We explained the checking schemes and

re-execution of instruction mechanisms within ERRIC and how they can improve reliability.

With regards to the Interfacing zone, we introduced the T-Logic as basic unit of reconfiguration and discuss its various configurations. We introduced the syndrome and explained implementation details and how, in combination with a Memory Management Unit and a Reconfigurable Memory Scheme, it can act as a control centre of three functions: fault monitoring, reconfigurability and recovery. As part of the passive zone, the reconfigurable memory scheme can operate 25 memory configurations and support graceful degradation. We quantify the probability of state transitions and provide a Markov model of reliability for ERA's configurable memory. Finally we described the system software support for testing and reconfiguration. We showed that by combining this novel hardware architecture with the system software, all key properties of performance, reliability and energy-wise functioning could be improved.

In Chapter 7 we provided the implementation details of ERA's hardware prototype. Having a software simulator of a hardware platform at hand is very useful to speed up software development and debugging of applications. We developed an accurate hardware simulator with graphical user frontend called Dissimera. Dissimera's main goal was not speed but to simulate reliably and accurately the basic core features of ERA with fully reproducible results. The simulator is built extendable; once core simulation is achieved, we will escalate from there adding new features with an agile methodology. The development of such disassembler/simulator gives us the possibility of 1) testing and locating errors of design of the soft core processor; 2) understand the smallest details of the ERRIC functionality; 3) Simulation of the current version of the processor and the FT version of the processor; 4) testing and debugging of errors in application and system software. Finally we introduced a testing framework that in combination with Dissimera's, with ERA's assembler and with commercial hardware simulators can properly test and debug not only the ERA's hardware prototype but ERRIC's application and system software.

8.1. Next steps

Arithmetic and logic units are both implemented through the use of logic components. It is known that an arithmetic instruction can be translated into several logic operations. Applying this principle, if an arithmetic unit is suspected of not being able to provide correct service, arithmetic instructions can be translated into logic ones that can be executed by the logic unit of the ALU. Further research could be done on determining if logic operations can be translated into a set of arithmetic ones and how can this be implemented. What would it be the complexity of such translation. Performance would be affected (graceful degradation), but this technique would allow a running program to finish before recovery or fail-safe restart takes place.

The impact of the size of the register file on overall performance of processor is also a question of further research as in 6.2.1.

With regards to Dissimera, although basic functionality has been achieved, the implementation of Dissimera is still a working progress:

- The design is completed and the user interface is fully defined.
- Assembling of pseudo code using ERA assembler/preparator (100% completed).
- Disassembling of binary into human readable code (assembly code) (100% completed).
- The simulator is capable of parsing the binary file resulting from the previous step and is then capable of classifying data and instructions (100% completed).
- Simulation of main memory, register file, program counter and instruction registry is almost completed (90%).

For future revisions of Dissimera we are working on a low-level fault-injection scheme that would support testing of the architecture. We also plan to include

support for: syndrome, extra memory configurations including 16-bit memory configurations.

We are very interested in finding ways to exploit the functionality that the syndrome can provide. We believe that for safety-critical missions such as embedded systems in satellites or space further research is needed. We would like to pursue more research in dependency matrix mapping of symptoms and failure modes. We would like to apply the context sensing (e.g. altitude, latitude, temperature, dynamic events such as solar flares and weather forecasting) and experience to system software in combination with the syndrome.

8.2. Personal contributions

I am responsible for the definition of Resilience in Chapter 2. Some of the attributes are based on individual authors but the combination of those and the particular definition is my own work.

The author is responsible for the two taxonomies: taxonomy of mitigation techniques (Chapter 3), the taxonomy of single event effects (Chapter 4).

The individual contributions of the author in Chapter 5 are the implementation of the Fault Model. GAFT is based on previous work from (Sogomonian and Schagaev, 1988). The author is responsible for extending GAFT by adding the system state changes and the actions to implement fault tolerance.

The ERRIC microprocessor including the active zone, checking schemes, the instruction set architecture and the T-Logic as a concept is the result of previous work from the ONBASS project. The memory management unit that allows implementation of these as a concept is my contribution. The syndrome was a simple idea of Prof. Schagaev that I took on and further developed. I am responsible for the extension of the syndrome concept and its implementation, including the structure, location, access and functionality. The graceful degradation and Markov Models are also my contribution.

As stated earlier, the instruction set was designed before the thesis was started. The assembler is contribution of ETH Zurich. My main personal contribution in Chapter 7 is the framework for testing including 1) the testing of the custom FPGA-based board, 2) the functional testing of the novel architecture and the development of the disassembler and simulator (Dissimera).

References

Abramovici, M., Breuer, M.A., 1979. On Redundancy and Fault Detection in Sequential Circuits. IEEE Trans. Comput. 28, 864–865. doi:10.1109/TC.1979.1675267

Abramovici, M., Breuer, M.A., Friedman, A.D., 1994. Digital Systems Testing & Testable Design, 1st ed. Wiley-IEEE Press.

Adams, J.H., Gelman, A., 1984. The Effects of Solar Flares on Single Event Upset Rates. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.1984.4333485 31, 1212–1216.

Adams, J.H., Silberberg, R., Tsao, C.H., 1982. Cosmic Ray Effects on Microelectronics. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.1982.4335821 29, 169–172.

Agrawal, V.D., Chakradhar, S.T., 1995. Combinational ATPG theorems for identifying untestable faults in sequential circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14, 1155 – 1160. doi:10.1109/43.406717

Alanen, J., Ungar, L.Y., 2011. Comparing software design for testability to hardware DFT and BIST, in: 2011 IEEE AUTOTESTCON. Presented at the 2011 IEEE AUTOTESTCON, pp. 272 –278. doi:10.1109/AUTEST.2011.6058776

Allenspach, M., Brews, J.R., Mouret, I., Schrimpf, R.D., Galloway, K.F., 1994. Evaluation of SEGR threshold in power MOSFETs. Nuclear Science, IEEE Transactions on 41, 2160–2166. doi:10.1109/23.340557

Amusan, O.A., Witulski, A.F., Massengill, L.W., Bhuva, B.L., Fleming, P.R., Alles, M.L., Sternberg, A.L., Black, J.D., Schrimpf, R.D., 2006. Charge Collection and Charge Sharing in a 130 nm CMOS Technology. Nuclear Science, IEEE Transactions on 53, 3253–3258. doi:10.1109/TNS.2006.884788

Anderson, T., Lee, 1981. Fault Tolerance: Principles and Practice. Prentice-Hall.

Antola, A., Erényi, I., Scarabottolo, N., 1986. Transient fault management in systems based on the AMD 2900 microprocessors. Microprocessing and Microprogramming 17, 205–217. doi:10.1016/0165-6074(86)90115-8

Applebaum, S.P., 1965. Steady-State Reliability of Systems of Mutually Independent Subsystems. IEEE Transactions on Reliability R-14, 23 –29. doi:10.1109/TR.1965.5214868

Arimoto, K., Matsuda, Y., Furutani, K., Tsukude, M., Ooishi, T., Mashiko, K., Fujishima, K., 1990. A speed-enhanced DRAM array architecture with embedded ECC. IEEE Journal of Solid-State Circuits 25, 11 –17. doi:10.1109/4.50277

Armstrong, D.B., 1966. On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets. IEEE Transactions on Electronic Computers EC-15, 66 –73. doi:10.1109/PGEC.1966.264376

Asakura, M., Matsuda, Y., Hidaka, H., Tanaka, Y., Fujishima, K., 1990. An experimental 1-Mbit cache DRAM with ECC. IEEE Journal of Solid-State Circuits 25, 5 –10. doi:10.1109/4.50276

Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelick, K., 2006. The landscape of parallel computing research: a view from Berckeley (No. Technical Report No. UCB/EECS-2006-183).

Avizienis, A., 1971. Faulty-Tolerant Computing: An Overview. Computer 4, 5–8.

Avizienis, A., 1976. Fault-Tolerant Systems. IEEE Transactions on Computers C-25, 1304 –1312. doi:10.1109/TC.1976.1674598

Avizienis, A., 1982. The Four-Universe Information System Model for the Study of Fault Tolerance. Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing, Santa Monica, California pp. 6–13.

Avizienis, A., Kelly, J.P.J., 1984. Fault Tolerance by Design Diversity: Concepts and Experiments. Computer 17, 67 –80. doi:10.1109/MC.1984.1659219

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans.\ on Dependable and Secure Computing 1, 11–33.

Azadmanesh, M.H., Kieckhafer, R.M., 2000. Exploiting omissive faults in synchronous approximate agreement. IEEE Transactions on Computers 49, 1031 –1042. doi:10.1109/12.888039

Baeg, S., Wen, S., Wong, R., 2009. SRAM Interleaving Distance Selection With a Soft Error Failure Model. IEEE Transactions on Nuclear Science 56, 2111–2118. doi:10.1109/TNS.2009.2015312

Barlow, R.E., Proschan, F., 1975. Statistical Theory of Reliability and Life Testing: Probability Models,.

Barth, J.L., LaBel, K.A., Poivey, C., 2004. Radiation assurance for the space environment, in: Integrated Circuit Design and Technology, 2004. ICICDT '04. International Conference on. Presented at the Integrated Circuit Design and Technology, 2004. ICICDT '04. International Conference on, pp. 323–333. doi:10.1109/ICICDT.2004.1309976

Baumann, R., 2002. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction, in: Electron Devices Meeting, 2002. IEDM '02. Digest. International. Presented at the Electron Devices Meeting, 2002. IEDM '02. Digest. International, pp. 329–332. doi:10.1109/IEDM.2002.1175845

Baumann, R., 2005. Soft errors in advanced computer systems. Design & Test of Computers, IEEE 22, 258–266. doi:10.1109/MDT.2005.69

Baumann, R., Hossain, T., Murata, S., Kitagawa, H., 1995. Boron compounds as a dominant source of alpha particles in semiconductor devices, in: Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International. Presented at the Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International, pp. 297–302. doi:10.1109/RELPHY.1995.513695

Baumann, R.C., 2001. Soft errors in advanced semiconductor devices-part I: the three radiation sources. Device and Materials Reliability, IEEE Transactions on 1, 17–22. doi:10.1109/7298.946456

Baumann, R.C., 2005. Radiation-induced soft errors in advanced semiconductor technologies. Device and Materials Reliability, IEEE Transactions on 5, 305–316. doi:10.1109/TDMR.2005.853449

Beaudry, M.D., 1978. Performance-Related Reliability Measures for Computing Systems. IEEE Transactions on Computers C-27, 540 –547. doi:10.1109/TC.1978.1675145

Becker, H.N., Miyahira, T.F., Johnston, A.H., 2002. Latent damage in CMOS devices from single-event latchup. Nuclear Science, IEEE Transactions on 49, 3009–3015. doi:10.1109/TNS.2002.805332

Beitman, B.A., 1988. n-channel MOSFET breakdown characteristics and modeling for p-well technologies. Electron Devices, IEEE Transactions on 35, 1935–1941. doi:10.1109/16.7407

Bentoutou, Y., Djaifri, M., 2008. Observations of single-event upsets and multiple-bit upsets in random access memories on-board the Algerian satellite, in: Nuclear Science Symposium Conference Record, 2008. NSS '08. IEEE. Presented at the Nuclear Science Symposium Conference Record, 2008. NSS '08. IEEE, pp. 2568–2570. doi:10.1109/NSSMIC.2008.4774882

Berger, M.J., Coursey, J.S., Zucker, M.A., Chang, J., 2005. ESTAR, PSTAR, and ASTAR: Computer Programs for Calculating Stopping-Power and Range Tables for Electrons, Protons, and Helium Ions (version 1.2.3) [WWW Document]. URL http://physics.nist.gov/Star (accessed 11.9.10).

Bernstein, A.V., Tomfield, Y.L., Schagaev, I.V., 1992. Storage Unit with High Reliability Characteristics. I. Avtomat. i Telemekh 145–152.

Bernstein, A.V., Tomfield, Y.L., Schagaev, I.V., 1993. RAM of High Reliability Properties. II. Avtomat. i Telemekh 169–179.

Binder, D., Smith, E.C., Holman, A.B., 1975. Satellite Anomalies from Galactic Cosmic Rays. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.1975.4328188 22, 2675–2680.

Birolini, A., 2007. Reliability engineering. Springer.

Blake, J.B., Mandel, R., 1986. On-Orbit Observations of Single Event Upset in Harris HM-6508 1K RAMS. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.1986.4334651 33, 1616–1619.

Blandford, J.T., Waskiewicz, A.E., Pickel, J.C., 1984. Cosmic Ray Induced Permanent Damage in MNOS EAROMs. Nuclear Science, IEEE Transactions on 31, 1568–1570. doi:10.1109/TNS.1984.4333551

Boatella, C., Hubert, G., Ecoffet, R., Duzellier, S., 2009. ICARE on-board SAC-C: More than 8 years of SEU & MCU, analysis and prediction. IEEE, pp. 369–374. doi:10.1109/RADECS.2009.5994678

Bodsberg, L., Hokstad, P., 1995. A system approach to reliability and life-cycle cost of process safety-systems. IEEE Transactions on Reliability 44, 179 – 186. doi:10.1109/24.387369

Bogliolo, A., Favalli, M., Damiani, M., 2000. Enabling testability of fault-tolerant circuits by means of IDDQ checkable voters. IEEE Trans. Very Large Scale Integr. Syst. 8, 415–419. doi:10.1109/92.863620

Bose, R.C., Ray-Chaudhuri, D.K., 1960. On a class of error correcting binary group codes. Information and Control 3, 68–79. doi:10.1016/S0019-9958(60)90287-4

Bougerol, A., Miller, F., Buard, N., 2008. SDRAM Architecture & Single Event Effects Revealed with Laser, in: On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International. Presented at the On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International, pp. 283–288. doi:10.1109/IOLTS.2008.40

Bougerol, A., Miller, F., Guibbaud, N., Gaillard, R., Moliere, F., Buard, N., 2010. Use of Laser to Explain Heavy Ion Induced SEFIs in SDRAMs. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.2009.2037418 57, 272–278.

Bougerol, A., Miller, F., Guibbaud, N., Leveugle, R., Carriere, T., Buard, N., 2011. Experimental Demonstration of Pattern Influence on DRAM SEU and SEFI Radiation Sensitivities. Nuclear Science, IEEE Transactions on 58, 1032 – 1039. doi:10.1109/TNS.2011.2107528

Bouricius, W.G., Carter, W.C., Schneider, P.R., 1969. Reliability modeling techniques for self-repairing computer systems, in: Proceedings of the 1969 24th National Conference, ACM '69. ACM, New York, NY, USA, pp. 295–309. doi:10.1145/800195.805940

Bradley, P.D., Normand, E., 1998. Single event upsets in implantable cardioverter defibrillators. Nuclear Science, IEEE Transactions on 45, 2929–2940. doi:10.1109/23.736549

Breuer, M.A., 1973. Testing for Intermittent Faults in Digital Circuits. Computers, IEEE Transactions on C-22, 241 – 246. doi:10.1109/T-C.1973.223701

Brocklehurst, S., Littlewood, B., Olovsson, T., Jonsson, E., 1994. On measurement of operational security. IEEE Aerospace and Electronic Systems Magazine 9, 7 –16. doi:10.1109/62.318876

Buchner, S., Baze, M., Brown, D., McMorrow, D., Melinger, J., 1997. Comparison of error rates in combinational and sequential logic. Nuclear Science, IEEE Transactions on 44, 2209–2216. doi:10.1109/23.659037

Buckle, R., Highleyman, W.H., 2003. The New NonStop Advanced Architecture: A Massive Jump in Processor Reliability. The Connection 24.

Caldwell, D.W., 1998. DSI GDE/Power Anomaly Day 300:Analysis and Resolution.

Carter, W.C., 1979. Hardware Fault Tolerance, in: Computing Systems Reliability. CUP Archive, pp. 211–263.

Carter, W.C., Bouricius, W.G., 1971. A Survey of Fault Tolerant Computer Architecture and its Evaluation. Computer 4, 9 –16. doi:10.1109/C-M.1971.216739

Carter, W.C., Schneider, P.R., 1968. Design of dynamically checked computers, in: IFIP Congress (2). pp. 878–883.

Caywood, J.M., Prickett, B.L., 1983. Radiation-Induced Soft Errors and Floating Gate Memories, in: Reliability Physics Symposium, 1983. 21st Annual. Presented at the Reliability Physics Symposium, 1983. 21st Annual, pp. 167–172. doi:10.1109/IRPS.1983.361979

Cazeaux, J.M., Rossi, D., Metra, C., 2004. New High Speed CMOS Self-Checking Voter, in: Proceedings of the International On-Line Testing Symposium, 10th IEEE, IOLTS '04. IEEE Computer Society, Washington, DC, USA, p. 58–. doi:10.1109/IOLTS.2004.31

Cha, H., Rudnick, E.M., Choi, G.S., Patel, J.H., Iyer, R.K., 1993. A fast and accurate gate-level transient fault simulation environment, in: Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on. Presented at the Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on, pp. 310–319. doi:10.1109/FTCS.1993.627334

Chang-Ming Hsieh, Murley, P.C., O'Brien, R.R., 1983. Collection of charge from alpha-particle tracks in silicon devices. Electron Devices, IEEE Transactions on 30, 686–693. doi:10.1109/T-ED.1983.21190

Chen, C.L., Hsiao, M.Y., 1984. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. IBM Journal of Research and Development 28, 124 –134. doi:10.1147/rd.282.0124

Claeys, C.L., Simoen, E., 2002. Radiation effects in advanced semiconductor materials and devices. Springer.

Coe, T., Mathisen, T., Moler, C., Pratt, V., 1995. Computational aspects of the Pentium affair. IEEE Computational Science Engineering 2, 18 –30. doi:10.1109/99.372929

Conlon, J.C., Lilius, W.A., Tubbesing, F.H., Evaluation, U.S.O. of the D.D.T. and, Engineering, U.S.O. of the U.S. of D. for R. and, Activity, U.S.A.M.S.A., Activity, U.S.A.M.E.T., 1982. Test and evaluation of system reliability, availability, maintainability: a primer. Office of the Director, Defense Test and Evaluation, Under Secretary of Defense for Research and Engineering.

Constantinescu, C., 2003. Trends and challenges in VLSI circuit reliability. IEEE Micro 23, 14 – 19. doi:10.1109/MM.2003.1225959

Constantinescu, C., Parulkar, I., Harper, R., Michalak, S., 2008. Silent Data Corruption - Myth or reality?, in: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. Presented at the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008, pp. 108 –109. doi:10.1109/DSN.2008.4630077

Cottrell, P.E., Troutman, R.R., Ning, T.H., 1979. Hot-electron emission in N-channel IGFET's. IEEE Transactions on Electron Devices 26, 520 – 533. doi:10.1109/T-ED.1979.19456

Crain, S.H., Velazco, R., Alvarez, M.T., Bofill, A., Yu, P., Koga, R., 1999. Radiation effects in a fixed-point digital signal processor, in: Radiation Effects Data Workshop, 1999. Presented at the Radiation Effects Data Workshop, 1999, pp. 30–34. doi:10.1109/REDW.1999.816053

Czajkowski, D.R., Pagey, M.P., Samudrala, P.K., Goksel, M., Viehman, M.J., 2005. Low Power, High-Speed Radiation Hardened Computer & Flight Experiment, in: Aerospace Conference, 2005 IEEE. Presented at the Aerospace Conference, 2005 IEEE, pp. 1–10. doi:10.1109/AERO.2005.1559559

Czajkowski, D.R., Samudrala, P.K., Pagey, M.P., 2006. SEU mitigation for reconfigurable FPGAs, in: Aerospace Conference, 2006 IEEE. Presented at the Aerospace Conference, 2006 IEEE, p. 7 pp. doi:10.1109/AERO.2006.1655957

DeAngelis, D., Lauro, J.A., 1976. Software recovery in the fault-tolerant spaceborne computer, in: Digest Sixth Int. Fault-Tolerant Computing Symp. IEEE Computer Society, Pittsburg, PA, pp. 143–148.

DeCastro, J., Tang, L., Zhang, B., Vachtsevanos, G., 2011. A Safety Verification Approach to Fault-Tolerant Aircraft Supervisory Control, in: AIAA Guidance, Navigation, and Control Conference. American Institute of Aeronautics and Astronautics.

Desko, J.C., Darwish, M.N., Dolly, M.C., Goodwin, C.A., Dawes, W.R., Titus, J.L., 1990. Radiations hardening of a high voltage IC technology (BCDMOS). IEEE Transactions on Nuclear Science 37, 2083–2088. doi:10.1109/23.101234

Dhillon, B.S., 2006. Maintainability, Maintenance, and Reliability for Engineers. CRC Press.

Dhillon, Y.S., Diril, A.U., Chatterjee, A., Metra, C., 2005. Load and Logic Co-Optimization for Design of Soft-Error Resistant Nanometer CMOS Circuits, in: IEEE International On-Line Testing Symposium. IEEE Computer Society, Los Alamitos, CA, USA, pp. 35–40. doi:http://doi.ieeecomputersociety.org/10.1109/IOLTS.2005.41

Dijkstra, E.W., 1965. Solution of a problem in concurrent programming control. Commun. ACM 8, 569. doi:10.1145/365559.365617

Dixit, A., Wood, A., 2011. The impact of new technology on soft error rates, in: Reliability Physics Symposium (IRPS), 2011 IEEE International. pp. 5B.4.1 –5B.4.7. doi:10.1109/IRPS.2011.5784522

Dodd, P.E., 2005. Physics-based simulation of single-event effects. Device and Materials Reliability, IEEE Transactions on 5, 343–357. doi:10.1109/TDMR.2005.855826

Dodd, P.E., Massengill, L.W., 2003. Basic mechanisms and modeling of single-event upset in digital microelectronics. Nuclear Science, IEEE Transactions on 50, 583–602. doi:10.1109/TNS.2003.813129

Dodd, P.E., Shaneyfelt, M.R., Walsh, D.S., Schwank, J.R., Hash, G.L., Loemker, R.A., Draper, B.L., Winokur, P.S., 2000. Single-event upset and snapback in silicon-on-insulator devices and integrated circuits. Nuclear Science, IEEE Transactions on 47, 2165–2174. doi:10.1109/23.903749

Dreslinski, R.G., Wieckowski, M., Blaauw, D., Sylvester, D., Mudge, T., 2010. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. Proceedings of the IEEE 98, 253 –266. doi:10.1109/JPROC.2009.2034764

Dufour, C., Garnier, P., Carriere, T., Beaucour, J., Ecoffet, R., Labrunee, M., 1992. Heavy ion induced single hard errors on submicronic memories [for space application]. Nuclear Science, IEEE Transactions on 39, 1693–1697. doi:10.1109/23.211355

Dugan, J.B., Trivedi, K.S., 1989. Coverage modeling for dependability analysis of fault-tolerant systems. IEEE Transactions on Computers 38, 775 –787. doi:10.1109/12.24286

Dunn, M., 1991. Designer fault models for VLSI, in: IEE Colloquium on Design for Testability. Presented at the IEE Colloquium on Design for Testability, pp. 4/1 –4/5.

Duzellier, S., Ecoffet, R., Falguere, D., Nuns, T., Guibert, L., Hajdas, W., Calvert, M.C., 1997. Low energy proton induced SEE in memories. IEEE Trans. Nucl. Sci. 44, 2306–2310. doi:10.1109/23.659050

Duzellier, S., Falguere, D., Ecoffet, R., 1993. Protons and heavy ions induced stuck bits on large capacity RAMs, in: Radiation and Its Effects on Components and Systems, 1993.,RADECS 93., Second European Conference on. Presented at the Radiation and its Effects on Components and Systems, 1993.,RADECS 93., Second European Conference on, pp. 468–472. doi:10.1109/RADECS.1993.316527

Dyer, C.S., Sims, A.J., Farren, J., Stephen, J., 1990. Measurements of solar flare enhancements to the single event upset environment in the upper atmosphere [avionics]. Nuclear Science, IEEE Transactions on 37, 1929–1937. doi:10.1109/23.101211

Dyer, C.S., Truscott, P.R., Evans, H., Sims, A.J., Hammond, N., Comber, C., 1996. Secondary radiation environments in heavy space vehicles and instruments. Advances in Space Research 17, 53–58. doi:10.1016/0273-1177(95)00512-D

Eckert, D.I., 2001. Odyssey MEEB Analysis: Lockheed-Martin Presentation.

Ecoffet, R., Duzellier, S., Tastet, P., Aicardi, C., Labrunee, M., 1994. Observation of heavy ion induced transients in linear circuits, in: Radiation Effects Data Workshop, 1994 IEEE. Presented at the Radiation Effects Data Workshop, 1994 IEEE, pp. 72–77. doi:10.1109/REDW.1994.633038

ECSS, 2007. Space engineering: Methods for the calculation of radiation received and its effects, and a policy for design margins - ECSS-E-10-12 Draft 0.5, ECSS-E-10-12 Draft 0.5.

Edwards, R., Dyer, C., Normand, E., 2004. Technical standard for atmospheric radiation single event effects, (SEE) on avionics electronics, in: Radiation Effects Data Workshop, 2004 IEEE. Presented at the Radiation Effects Data Workshop, 2004 IEEE, pp. 1–5. doi:10.1109/REDW.2004.1352895

EIA/JEDEC STANDARD,“Test Procedures for the Measurement of Single-Event Effects in Semiconductor Devices from Heavy Ion Irradiation,,” 1996.

Eishi Ibe, Chung, S.S., ShiJie Wen, Hironaru Yamaguchi, Yasuo Yahagi, Hideaki Kameyama, Shigehisa Yamamoto, Takashi Akioka, 2006. Spreading Diversity in Multi-cell Neutron-Induced Upsets with Device Scaling, in: Custom Integrated Circuits Conference, 2006. CICC ’06. IEEE. Presented at the Custom Integrated Circuits Conference, 2006. CICC ’06. IEEE, pp. 437–444. doi:10.1109/CICC.2006.321010

Elsayed, E.A., 1996. Reliability Engineering, Har/Dsk. ed. Prentice Hall.

Felix, J.A., Shaneyfelt, M.R., Schwank, J.R., Dalton, S.M., Dodd, P.E., Witcher, J.B., 2007. Enhanced Degradation in Power MOSFET Devices Due to Heavy Ion Irradiation. Nuclear Science, IEEE Transactions on 54, 2181–2189. doi:10.1109/TNS.2007.910873

Fieseler, P.D., Ardalan, S.M., Frederickson, A.R., 2002. The radiation effects on Galileo spacecraft systems at Jupiter. IEEE Transactions on Nuclear Science 49, 2739–2758. doi:10.1109/TNS.2002.805386

Fischer, T.A., 1987. Heavy-Ion-Induced, Gate-Rupture in Power MOSFETs. Nuclear Science, IEEE Transactions on 34, 1786–1791. doi:10.1109/TNS.1987.4337555

Fisher, J.A., 1983. Very Long Instruction Word architectures and the ELI-512, in: Proceedings of the 10th Annual International Symposium on Computer Architecture. ACM, Stockholm, Sweden, pp. 140–150. doi:10.1145/800046.801649

Fleetwood, D., Pantelides, S., Schrimpf, R., 2008. Oxide Traps, Border Traps, and Interface Traps in SiO2, in: Fleetwood, D., Pantelides, S., Schrimpf, R. (Eds.), Defects in Microelectronic Materials and Devices. CRC Press.

Flynn, M., 1972. Some Computer Organizations and Their Effectiveness. IEEE Trans. Comput. C-21, 948.

Fortes, J.A.B., Raghavendra, C.S., 1985. Gracefully Degradable Processor Arrays. IEEE Transactions on Computers C-34, 1033 –1044. doi:10.1109/TC.1985.1676536

Franklin, M., Saluja, K.K., 1995. Embedded RAM testing, in: , Records of the 1995 IEEE International Workshop on Memory Technology, Design and Testing, 1995. Presented at the , Records of the 1995 IEEE International Workshop on Memory Technology, Design and Testing, 1995, pp. 29 –33. doi:10.1109/MTDT.1995.518078

Friedman, A.D., 1967. Fault Detection in Redundant Circuits. IEEE Transactions on Electronic Computers EC-16, 99 –100. doi:10.1109/PGEC.1967.264621

Furutani, K., Arimoto, K., Miyamoto, H., Kobayashi, T., Yasuda, K., Mashiko, K., 1989. A built-in Hamming code ECC circuit for DRAMs. IEEE Journal of Solid-State Circuits 24, 50 –56. doi:10.1109/4.16301

Gaisler, J., 2002. A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture, in: Proceedings of the 2002 International Conference on Dependable Systems and Networks. IEEE Computer Society, pp. 409–415.

Galey, J.M., Norby, R.E., Roth, J.P., 1961. Techniques for the diagnosis of switching circuit failures, in: Proceedings of the Second Annual Symposium on Switching Circuit Theory and Logical Design, 1961. SWCT 1961. Presented at the Proceedings of the Second Annual Symposium on Switching Circuit Theory and Logical Design, 1961. SWCT 1961, pp. 152 – 160. doi:10.1109/FOCS.1961.33

Garland, D.J., Stainer, F.W., 2013. Modern Electronic Maintenance Principles. Elsevier.

Geer, D., 2007. For Programmers, Multicore Chips Mean Multiple Challenges. Computer 40, 17–19.

Geppert, L., 2004. A static RAM says goodbye to data errors [radiation induced soft errors]. IEEE Spectrum 41, 16 – 17. doi:10.1109/MSPEC.2004.1265121

Gnedenko, B., Pavlov, I.V., Ushakov, I.A., 1999. Statistical Reliability Engineering, 1st ed. Wiley-Interscience.

Goldstein, L., 1979. Controllability/observability analysis of digital circuits. IEEE Transactions on Circuits and Systems 26, 685 – 693. doi:10.1109/TCS.1979.1084687

Gössel, M., Ocheretny, V., Sogomonyan, E., Marienfeld, D., 2008. New Methods of Concurrent Checking. Springer.

Goth, G., 2009. Entering a parallel universe. Commun. ACM 52, 15. doi:10.1145/1562164.1562171

Gountanis, R.J., Viss, N.L., 1966. A method of processor selection for interrupt handling in a multiprocessor system. Proceedings of the IEEE 54, 1812 – 1819. doi:10.1109/PROC.1966.5265

Gregory, B.L., Shafer, B.D., 1973. Latch-Up in CMOS Integrated Circuits. IEEE Transactions on Nuclear Science 20, 293 –299. doi:10.1109/TNS.1973.4327410

Guenzer, C.S., Wolicki, E.A., Allas, R.G., 1979. Single Event Upset of Dynamic Rams by Neutrons and Protons. Nuclear Science, IEEE Transactions on 26, 5048–5052. doi:10.1109/TNS.1979.4330270

Hamming, R.W., 1950. Error correction and error detection coding. Bell System technical journal XXIX.

Hana, H.H., Johnson, B.W., 1986. Concurrent error detection in VLSI circuits using time redundancy. Proc. IEEE Southeastcon 1986 Regional Conf. pp. 208–212.

Haraszti, T.P., 2000. CMOS Memory Circuits. Springer.

Harboe-Sorensen, R., Guerre, F.-X., Lewis, G., 2007. Heavy-Ion SEE Test Concept and Results for DDR-II Memories. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.2007.909747 54, 2125–2130.

Hauge, P.S., Ziegler, J.F., Srinivasan, G.R., 1996. Special issue: terrestrial cosmic rays and soft errors [WWW Document]. IBM J. Res. Dev. URL http://portal.acm.org/citation.cfm?id=226354 (accessed 7.1.10).

Hawkins, C., Keshavarzi, A., Segura, J., 2003. CMOS IC nanometer technology failure mechanisms, in: Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003. Presented at the Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003, pp. 605 – 611. doi:10.1109/CICC.2003.1249470

Hayes, J.P., 1975. Detection oF Pattern-Sensitive Faults in Random-Access Memories. IEEE Transactions on Computers C-24, 150 – 157. doi:10.1109/T-C.1975.224182

Hazucha, P., Karnik, T., Maiz, J., Walstra, S., Bloechel, B., Tschanz, J., Dermer, G., Hareland, S., Armstrong, P., Borkar, S., 2003. Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25- mu;m to 90-nm generation, in: Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International. pp. 21.5.1 – 21.5.4. doi:10.1109/IEDM.2003.1269336

Hazucha, P., Svensson, C., 2000. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. IEEE Transactions on Nuclear Science 47, 2586 –2594. doi:10.1109/23.903813

Heidel, D.F., Marshall, P.W., LaBel, K.A., Schwank, J.R., Rodbell, K.P., Hakey, M.C., Berg, M.D., Dodd, P.E., Friendlich, M.R., Phan, A.D., Seidleck, C.M., Shaneyfelt, M.R., Xapsos, M.A., 2008. Low Energy Proton Single-Event-Upset Test Results on 65 nm SOI SRAM. Nuclear Science, IEEE Transactions on 55, 3394–3400. doi:10.1109/TNS.2008.2005499

Heise, V., 2009. Arm: Nachwuchs fr die die cortex-a- familie. World Wide Web electronic publication.

Hennessy, J.L., Patterson, D.A., 2006. Computer Architecture: A Quantitative Approach, 4th Edition, 4th ed. Morgan Kaufmann.

Hentschke, R., Marques, F., Lima, F., Carro, L., Susin, A., Reis, R., 2002. Analyzing Area and Performance Penalty of Protecting Different Digital Modules with Hamming Code and Triple Modular Redundancy, in: Proceedings of the 15th Symposium on Integrated Circuits and Systems Design, SBCCI '02. IEEE Computer Society, Washington, DC, USA, p. 95–.

Hill, M.D., Rajwar, R., 2001. The Rise and Fall of Multiprocessor Papers in ISCA [WWW Document]. The Rise and Fall of Multiprocessor Papers in the International Symposium on Computer Architecture (ISCA). URL http://pages.cs.wisc.edu/~markhill/mp2001.html (accessed 4.8.10).

Hohl, J.H., Galloway, K.F., 1987. Analytical Model for Single Event Burnout of Power MOSFETs. Nuclear Science, IEEE Transactions on 34, 1275–1280. doi:10.1109/TNS.1987.4337465

Hollnagel, P.E., Wreathall, M.J., Woods, P.D.D., Pariès, J., 2012. Resilience Engineering in Practice: A Guidebook. Ashgate Publishing, Ltd.

Howe, C.L., Weller, R.A., Reed, R.A., Mendenhall, M.H., Schrimpf, R.D., Warren, K.M., Ball, D.R., Massengill, L.W., LaBel, K.A., Howard, J.W., Haddad, N.F., 2005. Role of heavy-ion nuclear reactions in determining on-orbit single event error rates. Nuclear Science, IEEE Transactions on 52, 2182–2188. doi:10.1109/TNS.2005.860683

Hsiao, M.Y., 1970. A class of optimal minimum odd-weight-column SEC-DED codes. IBM J. Res. Dev. 14, 395–401. doi:10.1147/rd.144.0395

Hsieh, C.M., Murley, P.C., O'Brien, R.R., 1981. A field-funneling effect on the collection of alpha-particle-generated carriers in silicon devices. Electron Device Letters, IEEE 2, 103–105. doi:10.1109/EDL.1981.25357

Hughes, H.L., Benedetto, J.M., 2003. Radiation effects and hardening of MOS technology: devices and circuits. IEEE Transactions on Nuclear Science 50, 500 – 521. doi:10.1109/TNS.2003.812928

Hugue, M.M., Purtilo, J., 2002. Guerrilla tactics: motivating design patterns for high-dependability applications, in: 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings. Presented at the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings, pp. 33 – 39. doi:10.1109/SEW.2002.1199447

Hutcheson, G.D., 2009. The Economic Implications of Moore's Law, in: Into the Nano Era. pp. 11–38.

Ibe, E., Taniguchi, H., Yahagi, Y., Shimbo, K. -i., Toba, T., 2010. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. Electron Devices, IEEE Transactions on 57, 1527 –1538. doi:10.1109/TED.2010.2047907

Irom, F., Nguyen, D.N., 2007. Single Event Effect Characterization of High Density Commercial NAND and NOR Nonvolatile Flash Memories. Nuclear Science, IEEE Transactions on 54, 2547–2553. doi:10.1109/TNS.2007.909984

ISO, 2004. ISO, 13381-1, Condition monitoring and diagnostics of machines - prognostics - Part1: General guidelines.

ITRS, 2011. International Technology Roadmap for Semiconductors.

J. R. Schwank, P.E.D., 2003. Charge collection in SOI capacitors and circuits and its effect on SEU hardness. Nuclear Science, IEEE Transactions on 2937 – 2947. doi:10.1109/TNS.2002.805429

Jardine, A.K.S., Lin, D., Banjevic, D., 2006. A review on machinery diagnostics and prognostics implementing condition-based maintenance. Mechanical Systems and Signal Processing 20, 1483–1510. doi:10.1016/j.ymssp.2005.09.012

Jennings, B.F., 1990. Fault detection in microprocessor based systems, in: IEE Colloquium on Fault Tolerant Techniques. Presented at the IEE Colloquium on Fault Tolerant Techniques, pp. 7/1 –718.

Johansson, K., Dyreklev, P., Granbom, O., Calver, M.C., Fourtine, S., Feuillatre, O., 1998. In-flight and ground testing of single event upset sensitivity in static RAMs. Nuclear Science, IEEE Transactions on DOI - 10.1109/23.685251 45, 1628–1632.

Johnson, B.W., 1989. The Design and Analysis of Fault Tolerant Digital Systems. Addison-Wesley.

Johnson, B.W., Aylor, J.H., Hana, H.H., 1988. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. IEEE Journal of Solid-State Circuits 23, 208 –215. doi:10.1109/4.281

Johnson, G.H., Schrimpf, R.D., Galloway, K.F., Koga, R., 1992. Temperature dependence of single-event burnout in n-channel power MOSFETs [for space application]. IEEE Transactions on Nuclear Science 39, 1605–1612. doi:10.1109/23.211342

Johnston, A.H., 1996. The influence of VLSI technology evolution on radiation-induced latchup in space systems. Nuclear Science, IEEE Transactions on 43, 505–521. doi:10.1109/23.490897

Johnston, A.H., Hughlock, B.W., Baze, M.P., Plaag, R.E., 1991. The effect of temperature on single-particle latchup. IEEE Transactions on Nuclear Science 38, 1435–1441. doi:10.1109/23.124129

K. LaBel, E.G. Stassinopoulos, G.J. Brucker, C.A. Stauffer, 1992. SEU tests of a 80386 based flight-computer/data-handling system and of discrete PROM and EEPROM devices, and SEL tests of discrete 80386, 80387, PROM, EEPROM and ASICs, in: Radiation Effects Data Workshop, 1992. Workshop Record., 1992 IEEE. Presented at the Radiation Effects Data Workshop, 1992. Workshop Record., 1992 IEEE, pp. 1–11. doi:10.1109/REDW.1992.247332

Kadayif, I., Sen, H., Koyuncu, S., 2010. Modeling soft errors for data caches and alleviating their effects on data reliability. Microprocess. Microsyst. 34, 200–214. doi:10.1016/j.micpro.2010.04.003

Kaegi-Trachsel, T., Gutknecht, J., 2008. Minos—the design and implementation of an embedded real-time operating system with a perspective of fault tolerance, in: Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on. Presented at the Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on, pp. 649–656. doi:10.1109/IMCSIT.2008.4747312

Kaegi-Trachsel, T., Schagaev, I., Gutknecht, J., 2009. Hardware testing on the level of tasks, in: 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software. Presented at the International Multiconference on Computer Science and Information Technology.

Karimi, F., Lombardi, F., 2002. A scan-BIST environment for testing embedded memories, in: On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International. Presented at the On-Line Testing Workshop, 2002.

Proceedings of the Eighth IEEE International, pp. 211 – 217. doi:10.1109/OLT.2002.1030221

Karnik, T., Hazucha, P., 2004. Characterization of soft errors caused by single event upsets in CMOS processes. IEEE Transactions on Dependable and Secure Computing 1, 128 – 143. doi:10.1109/TDSC.2004.14

Kato, M., Watanabe, K., Okabe, T., 1989. Radiation effects on ion-implanted silicon-dioxide films. IEEE Transactions on Nuclear Science 36, 2199 – 2204. doi:10.1109/23.45425

Katz, R., Barto, R., McKerracher, P., Carkhuff, B., Koga, R., 1994. SEU hardening of field programmable gate arrays (FPGAs) for space applications and device characterization. Nuclear Science, IEEE Transactions on 41, 2179–2186. doi:10.1109/23.340560

Katz, R., LaBel, K., Wang, J.J., Cronquist, B., Koga, R., Penzin, S., Swift, G., 1997. Radiation effects on current field programmable technologies. Nuclear Science, IEEE Transactions on 44, 1945–1956. doi:10.1109/23.658966

Kaufman, L., Johnson, B.W., 2001. Embedded Digital System Reliability and Safety Analysis. NUREG/GR-0020.

Kim, J., Hardavellas, N., Mai, K., Falsafi, B., Hoe, J., 2007. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40. IEEE Computer Society, Washington, DC, USA, pp. 197–209. doi:10.1109/MICRO.2007.28

Kish, L.B., 2002. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. Physics Letters A 305, 144–149. doi:10.1016/S0375-9601(02)01365-8

Koga, R., Crain, S.H., Yu, P., Crawford, K.B., 2001a. SEE sensitivity determination of high-density DRAMs with limited-range heavy ions, in: Radiation Effects Data Workshop, 2001 IEEE. Presented at the Radiation Effects Data Workshop, 2001 IEEE, pp. 182–189. doi:10.1109/REDW.2001.960479

Koga, R., Crain, W.R., Crawford, K.B., Lau, D.D., Pinkerton, S.D., Yi, B.K., Chitty, R., 1991. On the suitability of non-hardened high density SRAMs for space applications. Nuclear Science, IEEE Transactions on 38, 1507–1513. doi:10.1109/23.124139

Koga, R., Crawford, K.B., Grant, P.B., Kolasinski, W.A., Leung, D.L., Lie, T.J., Mayer, D.C., Pinkerton, S.D., Tsubota, T.K., 1993a. Single ion induced multiple-bit

upset in IDT 256K SRAMs, in: Radiation and Its Effects on Components and Systems, 1993.,RADECS 93., Second European Conference on. Presented at the Radiation and its Effects on Components and Systems, 1993.,RADECS 93., Second European Conference on, pp. 485–489. doi:10.1109/RADECS.1993.316526

Koga, R., Kolasinski, W.A., 1989. Heavy ion induced snapback in CMOS devices. Nuclear Science, IEEE Transactions on 36, 2367–2374. doi:10.1109/23.45450

Koga, R., Kolasinski, W.A., Marra, M.T., Hanna, W.A., 1985. Techniques of Microprocessor Testing and SEU-Rate Prediction. Nuclear Science, IEEE Transactions on 32, 4219–4224. doi:10.1109/TNS.1985.4334098

Koga, R., Pinkerton, S.D., Lie, T.J., Crawford, K.B., 1993b. Single-word multiple-bit upsets in static random access devices. Nuclear Science, IEEE Transactions on 40, 1941–1946. doi:10.1109/23.273460

Koga, R., Yu, P., Crawford, K.B., Crain, S.H., Tran, V.T., 2001b. Permanent single event functional interrupts (SEFIs) in 128- and 256-megabit synchronous dynamic random access memories (SDRAMs), in: Radiation Effects Data Workshop, 2001 IEEE. Presented at the Radiation Effects Data Workshop, 2001 IEEE, pp. 6–13.

Koren, I., Koren, Z., 1998. Defect tolerance in VLSI circuits: techniques and yield analysis. Proceedings of the IEEE 86, 1819 –1838. doi:10.1109/5.705525

Koren, I., Krishna, C.M., 2007. Fault-Tolerant Systems. Morgan Kaufmann.

Koren, I., Singh, A.D., 1990. Fault tolerance in VLSI circuits. Computer 23, 73 –83. doi:10.1109/2.56854

Kovalenko, I.N., Kuznetzov, N.Y., Pegg, P.A., 1997. Mathematical Theory of Reliability of Time Dependent Systems with Practical Applications, 1st ed. Wiley.

Kulkarni, G.V., Nicola, F.V., Trivedi, S.K., 1987. Effects of Checkpointing and Queueing on Program Performance. Duke University, Durham, NC, USA.

LaBel, K.A., Gates, M.M., Moran, A.K., Kim, H.S., Seidleck, C.M., Marshall, P., Kinnison, J., Carkhuff, B., 1996. Radiation effect characterization and test methods of single-chip and multi-chip stacked 16 Mbit DRAMs. Nuclear Science, IEEE Transactions on 43, 2974–2981. doi:10.1109/23.556894

Lala, J.H., Harper, R.E., 1994. Architectural principles for safety-critical real-time applications. Proceedings of the IEEE 82, 25 –40. doi:10.1109/5.259424

Landis, D.L., 1989. A self-test system architecture for reconfigurable WSI, in: Test Conference, 1989. Proceedings. Meeting the Tests of Time., International. Presented at the Test Conference, 1989. Proceedings. Meeting the Tests of Time., International, pp. 275 –282. doi:10.1109/TEST.1989.82308

Landwehr, C.E., Bull, A.R., McDermott, J.P., Choi, W.S., 1994. A taxonomy of computer program security flaws. ACM Comput. Surv. 26, 211–254. doi:10.1145/185403.185412

Laplante, P.A., Ovaska, S.J., 2011. Real-Time Systems Design and Analysis: Tools for the Practitioner. John Wiley & Sons.

Laprie, J., 1995. {Dependability - Its Attributes, Impairments and Means}, in: Randell, B., Laprie, J., Kopetz, H., Littlewood, B. (Eds.), Predictably Dependable Computing Systems. Springer-Verlag Heidelberg, pp. 1–24.

Laprie, J., Avizienis, A., 1986. Dependable computing: From concepts to design diversity, in: PROCEEDINGS OF THE IEEE. pp. 629–638.

Laprie, J.-C., 2008. From dependability to resilience 8, G8–G9.

Laprie, J.C.C., Avizienis, A., Kopetz, H. (Eds.), 1992. Dependability: Basic Concepts and Terminology. Springer-Verlag New York, Inc.

Latchoumy, P., Sheik, P., Khader, A., 2011. Survey on Fault Tolerance in Grid Computing.

Lawrence, R.K., 2007. Radiation Characterization of 512Mb SDRAMs, in: Radiation Effects Data Workshop, 2007 IEEE. Presented at the Radiation Effects Data Workshop, 2007 IEEE, pp. 204–207. doi:10.1109/REDW.2007.4342566

Leavy, J.F., Poll, R.A., 1969. Radiation-Induced Integrated Circuit Latchup. Nuclear Science, IEEE Transactions on 16, 96–103. doi:10.1109/TNS.1969.4325510

LEON3-FT SPARC V8 Processor - LEON3FT-RTAX - Data Sheet and User's manual. Aeroflex Gaisler AB, Sweden, 1.1.0.8 edition, 2009.

Li, J., Swartzlander, E., 1992. Concurrent error detection in ALUs by recomputing with rotated operands, in: , 1992 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings. Presented at the ,

1992 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, 1992. Proceedings, pp. 109 –116. doi:10.1109/DFTVS.1992.224374

Lie, C.H., Hwang, C.L., Tillman, F.A., 1977. Availability of Maintained Systems: A State-of-the-Art Survey. A I I E Transactions 9, 247–259. doi:10.1080/05695557708975153

Lin, S., Costello, D.J., 1983. Error Control Coding: Fundamentals and Applications. Prentice Hall.

Mahout, G., Pearce, M., Andrieux, M.-L., Arvidsson, C.-B., Charlton, D.G., Dinkespiler, B., Dowell, J.D., Gallin-Martel, L., Homer, R.J., Jovanovic, P., Kenyon, I.R., Kuyt, G., Lundquist, J., Mandic, I., Martin, O., Shaylor, H.R., Stroynowski, R., Troska, J., Wastie, R.L., Weidberg, A.R., Wilson, J.A., Ye, J., 2000. Irradiation studies of multimode optical fibres for use in ATLAS front-end links. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 446, 426–434. doi:10.1016/S0168-9002(99)01275-9

Maiz, J., Hareland, S., Zhang, K., Armstrong, P., 2003. Characterization of multi-bit soft error events in advanced SRAMs, in: Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International. Presented at the Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International, IEEE, pp. 21.4.1– 21.4.4. doi:10.1109/IEDM.2003.1269335

Mao, W., Gulati, R.K., Goel, D.K., Ciletti, M.D., 1990. QUIETEST: a quiescent current testing methodology for detecting leakage faults, in: , 1990 IEEE International Conference on Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers. Presented at the , 1990 IEEE International Conference on Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers, pp. 280 –283. doi:10.1109/ICCAD.1990.129902

Marshall, R.W., 1963. Microelectronic Devices for Application in Transient Nuclear Radiation Environments. Aerospace and Navigational Electronics, IEEE Transactions on Technical_Paper, 1.4.1–1–1.4.1–6. doi:10.1109/TANE.1963.4502234

Mathew, B., Saab, D.G., 1993. Partial reset: An inexpensive design for testability approach, in: [4th] European Conference on Design Automation, 1993, with the European Event in ASIC Design. Proceedings. Presented at the [4th] European Conference on Design Automation, 1993, with the European Event in ASIC Design. Proceedings, pp. 151 –155. doi:10.1109/EDAC.1993.386484

Matsunaga, J., Momose, H., Iizuka, H., Kohyama, S., 1980. Characterization of two step impact ionization and its influence in NMOS and PMOS VLSI's, in: Electron Devices Meeting, 1980 International. Presented at the Electron Devices Meeting, 1980 International, pp. 736 – 739. doi:10.1109/IEDM.1980.189942

Mavis, D.G., 2002. "Single event transient phenomena—Challenges and solutions. " presented at the Microelectronics Reliability and Qualification Workshop.

Mavis, D.G., Eaton, P.H., 2002. Soft error rate mitigation techniques for modern microcircuits, in: Reliability Physics Symposium Proceedings, 2002. 40th Annual. Presented at the Reliability Physics Symposium Proceedings, 2002. 40th Annual, pp. 216–225. doi:10.1109/RELPHY.2002.996639

May, T., 1979. Soft Errors in VLSI: Present and Future. Components, Hybrids, and Manufacturing Technology, IEEE Transactions on 2, 377–387.

May, T.C., Scott, G.L., Meieran, E.S., Winer, P., Rao, V.R., 1984. Dynamic Fault Imaging of VLSI Random Logic Devices, in: Reliability Physics Symposium, 1984. 22nd Annual. Presented at the Reliability Physics Symposium, 1984. 22nd Annual, pp. 281–283. doi:10.1109/IRPS.1984.362061

May, T.C., Woods, M.H., 1979. Alpha-particle-induced soft errors in dynamic memories. Electron Devices, IEEE Transactions on DOI - 26, 2–9.

McCluskey, E.J., Tseng, C.-W., 2000. Stuck-fault tests vs. actual defects, in: Test Conference, 2000. Proceedings. International. Presented at the Test Conference, 2000. Proceedings. International, pp. 336 –342. doi:10.1109/TEST.2000.894222

McEvoy, D., 1981. The architecture of Tandem's NonStop system, in: Proceedings of the ACM '81 Conference, ACM '81. ACM, New York, NY, USA, p. 245–. doi:10.1145/800175.809886

McGraw-Hill concise encyclopedia of engineering., 2005. . McGraw-Hill, New York.

McMahan, M.A., Leitner, D., Gimpel, T., Morel, J., Ninemire, B., Siero, R., Silver, C., Thatcher, R., 2004. A 16 MeV/nucleon cocktail for heavy ion testing, in: Radiation Effects Data Workshop, 2004 IEEE. Presented at the Radiation Effects Data Workshop, 2004 IEEE, pp. 156–159. doi:10.1109/REDW.2004.1352923

Mei, K.C.Y., 1974. Bridging and Stuck-At Faults. IEEE Transactions on Computers C-23, 720 – 727. doi:10.1109/T-C.1974.224020

Messenger, G.C., Ash, M.S., 1992. The Effects of Radiation on Electronic Systems, 2nd ed. Springer.

Metra, C., Favalli, M., Ricco, B., 1997. Compact and low power on-line self-testing voting scheme, in: , 1997 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1997. Proceedings. Presented at the , 1997 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1997. Proceedings, pp. 137 –145. doi:10.1109/DFTVS.1997.628319

Meyer, F.J., Pradhan, D.K., 1991. Consensus with dual failure modes. IEEE Transactions on Parallel and Distributed Systems 2, 214 –222. doi:10.1109/71.89066

Miller, L.A., Brice, D.K., Prinja, A.K., Picraux, S.T., 1994. Molecular dynamics simulations of bulk displacement threshold energies in Si. Radiation Effects and Defects in Solids: Incorporating Plasma Science and Plasma Technology 129, 127. doi:10.1080/10420159408228889

Miller, L.S., Mullin, J.B., 1991. Electronic materials: from silicon to organics. Springer.

Moon, T.K., 2005. Error Correction Coding: Mathematical Methods and Algorithms. John Wiley & Sons.

Mrstik, B.J., Hughes, H.L., McMarr, P.J., Lawrence, R.K., Ma, D.I., Isaacson, I.P., Walker, R.A., 2000. Hole and electron trapping in ion implanted thermal oxides and SIMOX. IEEE Transactions on Nuclear Science 47, 2189 –2195. doi:10.1109/23.903752

Mukherjee, S.S., Emer, J., Fossum, T., Reinhardt, S.K., 2004. Cache Scrubbing in Microprocessors: Myth or Necessity?, in: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04), PRDC '04. IEEE Computer Society, Washington, DC, USA, pp. 37–42.

Murray, R.M., Day, J.C., Ingham, M.D., Reder, L.J., Williams, B.C., 2013. Engineering Resilient Space Systems: Final Report - Keck Institute for Space Studies.

Nair, R., 2002. Effect of increasing chip density on the evolution of computer architectures. IBM J. Res. Dev. 46, 223–234.

Nair, R., Thatte, S.M., Abraham, J.A., 1978. Efficient Algorithms for Testing Semiconductor Random-Access Memories. IEEE Transactions on Computers C-27, 572 –576. doi:10.1109/TC.1978.1675150

Naseer, R., Bhatti, R.Z., Draper, J., 2006. Analysis of Soft Error Mitigation Techniques for Register Files in IBM Cu-08 90nm Technology, in: 49th IEEE International Midwest Symposium on Circuits and Systems, 2006. MWSCAS '06. Presented at the 49th IEEE International Midwest Symposium on Circuits and Systems, 2006. MWSCAS '06, pp. 515 –519. doi:10.1109/MWSCAS.2006.382112

NASNGSFC Landsat-7 Project Office, Private Communication, 1995.

Neches, R., 2012. Engineered Resilient Systems: A DoD Science and Technology Priority Area.

Neuberger, G., de Lima Kastensmidt, F.G., Reis, R., 2005. An automatic technique for optimizing Reed-Solomon codes to improve fault tolerance in memories. IEEE Design Test of Computers 22, 50 – 58. doi:10.1109/MDT.2005.2

Neuberger, G., Lima, F. de, Carro, L., Reis, R., 2003. A multiple bit upset tolerant SRAM memory. ACM Trans. Des. Autom. Electron. Syst. 8, 577–590. doi:10.1145/944027.944038

Nguyen, D.N., Guertin, S.M., Swift, G.M., Johnston, A.H., 1999. Radiation effects on advanced flash memories. Nuclear Science, IEEE Transactions on 46, 1744–1750. doi:10.1109/23.819148

Nicolaidis, M. (Ed.), 2010. Soft Errors in Modern Electronic Systems, 1st Edition. ed. Springer.

Ning, T.H., Yu, H.N., 1974. Optically induced injection of hot electrons into SiO<inf>2</inf>. Journal of Applied Physics 45, 5373 –5378. doi:10.1063/1.1663246

Nishioka, Y., Ohyu, K., Ohji, Y., Kato, M., da Silva, E.F., J., Ma, T.P., 1989. Radiation hardened micron and submicron MOSFETs containing fluorinated oxides. IEEE Transactions on Nuclear Science 36, 2116 –2123. doi:10.1109/23.45413

Normand, E., Wert, J.L., Quinn, H., Fairbanks, T.D., Michalak, S., Grider, G., Iwanchuk, P., Morrison, J., Wender, S., Johnson, S., 2010. First Record of Single-Event Upset on Ground, Cray-1 Computer at Los Alamos in 1976. Nuclear Science, IEEE Transactions on 57, 3114 –3120. doi:10.1109/TNS.2010.2083687

Northcliffe, L., Schilling, R., 1970. Range and stopping-power tables for heavy ions. Atomic Data and Nuclear Data Tables 7, 233–463.

Oh, N., Shirvani, P.P., McCluskey, E.J., 2002a. Control-flow checking by software signatures. IEEE Transactions on Reliability 51, 111 –122. doi:10.1109/24.994926

Oh, N., Shirvani, P.P., McCluskey, E.J., 2002b. Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability 51, 63 –75. doi:10.1109/24.994913

Oldham, T.R., Ladbury, R.L., Friendlich, M., Kim, H.S., Berg, M.D., Irwin, T.L., Seidleck, C., LaBel, K.A., 2006. SEE and TID Characterization of an Advanced Commercial 2Gbit NAND Flash Nonvolatile Memory. Nuclear Science, IEEE Transactions on 53, 3217–3222. doi:10.1109/TNS.2006.885843

Oldham, T.R., Suhail, M., Friendlich, M.R., Carts, M.A., Ladbury, R.L., Kim, H.S., Berg, M.D., Poivey, C., Buchner, S.P., Sanders, A.B., Seidleck, C.M., LaBel, K.A., 2008. TID and SEE Response of Advanced 4G NAND Flash Memories, in: Radiation Effects Data Workshop, 2008 IEEE. Presented at the Radiation Effects Data Workshop, 2008 IEEE, pp. 31–37. doi:10.1109/REDW.2008.12

Olsen, J., Becher, P.E., Fynbo, P.B., Raaby, P., Schultz, J., 1993. Neutron-induced single event upsets in static RAMS observed a 10 km flight attitude. Nuclear Science, IEEE Transactions on 40, 74–77. doi:10.1109/23.212319

ONBASS Consortium, 2004. ONBASS Consortium, ON Board Active Safety System, European Commission DG Research, contract number AST4-CT-2004-516045.

Owens, B.D., Adams, M.E., Benzce, W.J., Green, G., Shestople, P., 2006. The Effects of Radiation Events on Gravity Probe B. IEEE Transactions on Nuclear Science.

Pankratius, V., Jannesari, A., Tichy, W.F., 2009. Parallelizing Bzip2: A Case Study in Multicore Software Engineering. IEEE Softw. 26, 70–77.

Patel, J.H., Fung, L.Y., 1982. Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. IEEE Transactions on Computers C-31, 589 –595. doi:10.1109/TC.1982.1676055

Paul, S., Cai, F., Zhang, X., Bhunia, S., 2011. Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache. IEEE Transactions on Computers 60, 20 –34. doi:10.1109/TC.2010.203

Pellish, J.A., Reed, R.A., Schrimpf, R.D., Alles, M.L., Varadharajaperumal, M., Niu, G., Sutton, A.K., Diestelhorst, R.M., Espinel, G., Krithivasan, R., Comeau, J.P.,

Cressler, J.D., Vizkelethy, G., Marshall, P.W., Weller, R.A., Mendenhall, M.H., Montes, E.J., 2006. Substrate Engineering Concepts to Mitigate Charge Collection in Deep Trench Isolation Technologies. IEEE Transactions on Nuclear Science 53, 3298 –3305. doi:10.1109/TNS.2006.885798

Pickel, J.C., Blandford, J.T., 1978. Cosmic Ray Induced in MOS Memory Cells. Nuclear Science, IEEE Transactions on 25, 1166–1171. doi:10.1109/TNS.1978.4329508

Pickel, J.C., Blandford, J.T., 1980. Cosmic-Ray-Induced Errors in MOS Devices. IEEE Transactions on Nuclear Science 27, 1006–1015. doi:10.1109/TNS.1980.4330967

Pierce, W.H., 1965. Failure-tolerant computer design. Academic Press, (New York).

Podgorsak, E.B., 2009. Radiation Physics for Medical Physicists. Springer.

Poivey, C., Gee, G., LaBel, K.A., Barth, J.L., 2004. In-flight observations of long-term single event effect (SEE) performance on X-ray Timing Explorer (XTE) solid-state recorders (SSRs) [SRAM], in: 2004 IEEE Radiation Effects Data Workshop. Presented at the 2004 IEEE Radiation Effects Data Workshop, pp. 54–57. doi:10.1109/REDW.2004.1352904

Pomeranz, I., Reddy, S.M., 1993. Classification of faults in synchronous sequential circuits. IEEE Transactions on Computers 42, 1066 –1077. doi:10.1109/12.241596

Pouponnot, A.L.R., 2005. Strategic Use of SEE Mitigation Techniques for the Development of the ESA Microprocessors: Past, Present and Future, in: Proceedings of the 11th IEEE International On-Line Testing Symposium, IOLTS '05. IEEE Computer Society, Washington, DC, USA, pp. 319–323. doi:10.1109/IOLTS.2005.66

Power 6 Specs: IBM Power6 Microprocessor and IBM System p 570, 2007.

Prasad, A.V.S.S., Agrawal, V.D., Atre, M.V., 2002. A new algorithm for global fault collapsing into equivalence and dominance sets, in: Test Conference, 2002. Proceedings. International. Presented at the Test Conference, 2002. Proceedings. International, pp. 391 – 397. doi:10.1109/TEST.2002.1041783

Price, D., 1995. Pentium FDIV flaw-lessons learned. IEEE Micro 15, 86 –88. doi:10.1109/40.372360

Pritchard, B.E., Swift, G.M., Johnston, A.H., 2002. Radiation Effects Predicted, Observed, and Galileo Compared for Spacecraft Systems.

Puchner, H., Kapre, R., Sharifzadeh, S., Majjiga, J., Chao, R., Radaelli, D., Wong, S., 2006. Elimination of Single Event Latchup in 90nm SRAM Technologies, in: Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International. Presented at the Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International, pp. 721 –722. doi:10.1109/RELPHY.2006.251342

Qian, Y., 2008. Information Assurance: Dependability and Security in Networked Systems. Morgan Kaufmann.

Quinn, H., Graham, P., Krone, J., Caffrey, M., Rezgui, S., 2005. Radiation-induced multi-bit upsets in SRAM-based FPGAs. Nuclear Science, IEEE Transactions on 52, 2455–2461. doi:10.1109/TNS.2005.860742

R. Ladbury, M. D Berg, H. Kim, K. LaBel, M. Friendlich, R. Koga, J. George, S. Crain, P. Yu, R. A. Reed, 2006. Radiation Performance of 1 Gbit DDR SDRAMs Fabricated in the 90 nm CMOS Technology Node, in: Radiation Effects Data Workshop, 2006 IEEE. Presented at the Radiation Effects Data Workshop, 2006 IEEE, pp. 126–130. doi:10.1109/REDW.2006.295480

Ramanarayanan, R., Degalahal, V.S., Krishnan, R., Jungsub Kim, Narayanan, V., Yuan Xie, Irwin, M.J., Unlu, K., 2009. Modeling Soft Errors at the Device and Logic Levels for Combinational Circuits. Dependable and Secure Computing, IEEE Transactions on 6, 202–216. doi:10.1109/TDSC.2007.70231

Ravishankar K. Iyer, Z.K., 2003. Hardware and Software Error Detection.

Reed, I.S., Solomon, G., 1960. Polynomial Codes Over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics 8, 300–304.

Reed, R.A., Carts, M.A., Marshall, P.W., Marshall, C.J., Buchner, S., La Macchia, M., Mathes, B., McMorrow, D., 1996. Single Event Upset cross sections at various data rates. Nuclear Science, IEEE Transactions on 43, 2862–2867. doi:10.1109/23.556878

Reed, R.A., Carts, M.A., Marshall, P.W., Marshall, C.J., Musseau, O., McNulty, P.J., Roth, D.R., Buchner, S., Melinger, J., Corbiere, T., 1997. Heavy ion and proton-induced single event multiple upset. IEEE Transactions on Nuclear Science 44, 2224–2229. doi:10.1109/23.659039

Reed, R.A., Weller, R.A., Schrimpf, R.D., Mendenhall, M.H., Warren, K.M., Massengill, L.W., 2006. Implications of Nuclear Reactions for Single Event

Effects Test Methods and Analysis. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.2006.885950 53, 3356–3362.

Reviriego, P., Maestro, J.A., Baeg, S., Wen, S., Wong, R., 2010. Protection of Memories Suffering MCUs Through the Selection of the Optimal Interleaving Distance. IEEE Transactions on Nuclear Science 57, 2124 – 2128. doi:10.1109/TNS.2010.2042818

Reviriego, P., Maestro, J.A., Cervantes, C., 2007. Reliability Analysis of Memories Suffering Multiple Bit Upsets. IEEE Transactions on Device and Materials Reliability 7, 592 –601. doi:10.1109/TDMR.2007.910443

Reynolds, D.A., Metze, G., 1978. Fault Detection Capabilities of Alternating Logic. IEEE Transactions on Computers C-27, 1093 –1098. doi:10.1109/TC.1978.1675011

Roche, P., Gasiot, G., 2005. Impacts of front-end and middle-end process modifications on terrestrial soft error rate. IEEE Transactions on Device and Materials Reliability 5, 382 – 396. doi:10.1109/TDMR.2005.853451

Rockett, L.R., 1988. An SEU-hardened CMOS data latch design. IEEE Transactions on Nuclear Science 35, 1682–1687. doi:10.1109/23.25522

Roy, R.K., Niermann, T.M., Patel, J.H., Abraham, J.A., Saleh, R.A., 1988. Compaction of ATPG-generated test sequences for sequential circuits, in: , IEEE International Conference on Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers. Presented at the , IEEE International Conference on Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers, pp. 382 –385. doi:10.1109/ICCAD.1988.122533

Saleh, A.M., Serrano, J.J., Patel, J.H., 1990. Reliability of scrubbing recovery-techniques for memory systems. IEEE Transactions on Reliability 39, 114 –122. doi:10.1109/24.52622

Sandireddy, R.K.K.R., Agrawal, V.D., 2005. Diagnostic and detection fault collapsing for multiple output circuits, in: Design, Automation and Test in Europe, 2005. Proceedings. Presented at the Design, Automation and Test in Europe, 2005. Proceedings, pp. 1014 – 1019 Vol. 2. doi:10.1109/DATE.2005.121

Schagaev, I., 1986. Algorithms of computation recovery. Automatic and Remote Control 7.

Schagaev, I., 1987. Algorithms to restoring a computing process. Automatic and Remote Control 7.

Schagaev, I., 1989. Yet Another Approach To Classification Of Redundancy, in: Proceedings of FTSD. Prague,Czeschoslovakia, pp. 485–490.

Schagaev, I., 2008. Reliability of malfunction tolerance, in: 2008 International Multiconference on Computer Science and Information Technology. Presented at the 2008 International Multiconference on Computer Science and Information Technology (IMCSIT), Wisla, Poland, pp. 733–737. doi:10.1109/IMCSIT.2008.4747323

Schagaev, I., 2009. ERA: Embedded Reconfigurable Architecture - past present and future.

Schagaev, I., Buhanova, G., 2001. Comparative Study of Fault Tolerant RAM Structures, in: IEEE DSN01. Presented at the IEEE DSN01, Goteborg.

Schagaev, I., Kaegi, T., Gutknetch, J., 2010. ERA: Evolving Reconfigurable Architecture, in: Proceedings of 11th ACIS. Presented at the International Conference on Software Engineering Artificial Intelligence, Networking and Parallel/Distributed Computing, London.

Schagaev, J.Z.I., 2001. Redundancy classification and its applications for fault tolerant computer design. IEEE TESADI-01.

Schindlbeck, G., 2005. Types of soft errors in DRAMs, in: Radiation and Its Effects on Components and Systems, 2005. RADECS 2005. 8th European Conference on. Presented at the Radiation and Its Effects on Components and Systems, 2005. RADECS 2005. 8th European Conference on, pp. PE1–1–PE1–5. doi:10.1109/RADECS.2005.4365591

Schroder, D.K., Babcock, J.A., 2003. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. Journal of Applied Physics 94, 1 –18. doi:10.1063/1.1567461

Seal, D., 2000. ARM architecture reference manual. Addison-Wesley.

Segura, J., Hawkins, C.F., 2005. Bridging Defects, in: CMOS Electronics. John Wiley & Sons, Inc., pp. 199–222.

Seifert, N., Xiaowei Zhu, Massengill, L.W., 2002. Impact of scaling on soft-error rates in commercial microprocessors. Nuclear Science, IEEE Transactions on 49, 3100–3106. doi:10.1109/TNS.2002.805402

Sexton, F.W., Fleetwood, D.M., Shaneyfelt, M.R., Dodd, P.E., Hash, G.L., 1997. Single event gate rupture in thin gate oxides. IEEE Transactions on Nuclear Science 44, 2345 –2352. doi:10.1109/23.659060

Shannon, C.E., 1948. A mathematical theory of communication. Bell System Technical Journal 27, pp 379–423,623–656.

Shedletsky, J.J., 1978. Error Correction by Alternate-Data Retry. IEEE Transactions on Computers C-27, 106 –112. doi:10.1109/TC.1978.1675044

Sherman, L., 2003. Stratus continuous processing technology – the smarter approach to uptime. Stratus Technologies Whitepaper. Technical report, Stratus Technologies.

Shirvani, P.P., McCluskey, E.J., 1998. Fault-Tolerant Systems in A Space Environment: The CRC ARGOS Project. Stanford University.

Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D., Alvisi, L., 2002. Modeling the effect of technology trends on the soft error rate of combinational logic, in: Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on. Presented at the Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pp. 389–398.

Silberberg, R., Tsao, C.H., Letaw, J.R., 1984. Neutron Generated Single-Event Upsets in the Atmosphere. Nuclear Science, IEEE Transactions on 31, 1183 –1185. doi:10.1109/TNS.1984.4333479

Silvestri, M., Gerardin, S., Paccagnella, A., Ghidini, G., 2009. Gate Rupture in Ultra-Thin Gate Oxides Irradiated With Heavy Ions. Nuclear Science, IEEE Transactions on 56, 1964–1970. doi:10.1109/TNS.2009.2022364

Sklaroff, J.R., 1976. Redundancy Management Technique for Space Shuttle Computers. IBM Journal of Research and Development 20, 20 –28. doi:10.1147/rd.201.0020

Slayman, C.W., 2005. Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. Device and Materials Reliability, IEEE Transactions on 5, 397–404. doi:10.1109/TDMR.2005.856487

Slegel, T.J., Averill, R.M., I., Check, M.A., Giamei, B.C., Krumm, B.W., Krygowski, C.A., Li, W.H., Liptay, J.S., MacDougall, J.D., McPherson, T.J., Navarro, J.A., Schwarz, E.M., Shum, K., Webb, C.F., 1999. IBM's S/390 G5 microprocessor design. IEEE Micro 19, 12 –23. doi:10.1109/40.755464

Smith, G.L., 1985. Models for delay faults based on paths. Proc. Int. Test Conf 342–349.

Smith, M., 1997. Application-Specific Integrated Circuits, 1st ed. Addison-Wesley Professional.

Sogomonian, E.., Schagaev, I., 1988. Hardware and software fault tolerance of computer systems. Avtomatika i Telemekhanika 3–39.

SPARC International, Inc., C., 1992. The SPARC Architecture Manual: Version 8. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Srour, J.R., Marshall, C.J., Marshall, P.W., 2003. Review of displacement damage effects in silicon devices. IEEE Transactions on Nuclear Science 50, 653 – 670. doi:10.1109/TNS.2003.813197

Stassinopoulos, E.G., Brucker, G.J., Calvel, P., Baiget, A., Peyrotte, C., Gaillard, R., 1992. Charge generation by heavy ions in power MOSFETs, burnout space predictions and dynamic SEB sensitivity. Nuclear Science, IEEE Transactions on 39, 1704–1711. doi:10.1109/23.211357

Stepanyants, A., 2001. Fault Tolerant Processor and Its Reliability Analysis, in: IEEE DSN01. Presented at the IEEE DSN01, Goteborg.

Storey, D.N., 1996. Safety Critical Computer Systems, 1st ed. Prentice Hall.

Stroud, C.E., 2002. A Designer's Guide to Built-in Self-Test. Springer.

Swift, D.W., John, G.H., 1997. Evaluation of the Space Environment on TOPEX-Poseidon and On-Board Failure of Optocouplers - Document JPL D-14157.

Swift, G., Katz, R., 1996. An experimental survey of heavy ion induced dielectric rupture in Actel Field Programmable Gate Arrays (FPGAs). Nuclear Science, IEEE Transactions on 43, 967–972. doi:10.1109/23.510741

Swift, G.M., Guertin, S.M., 2000. In-flight observations of multiple-bit upset in DRAMs. IEEE Transactions on Nuclear Science 47, 2386–2391. doi:10.1109/23.903781

Taber, A., Normand, E., 1993. Single event upset in avionics. Nuclear Science, IEEE Transactions on 40, 120–126. doi:10.1109/23.212327

Taber, A.H., Normand, E., 1992. Investigation and Characterization of SEU Effects and Hardening Strategies in Avionics. (IBM Report 92-L75-020-2 No. 92-L75-020-2), IBM Report 92-L75-020-2. IBM Report 92-L75-020-2. Defense Technical Information Center.

Takeda, E., Kume, H., Nakagome, Y., Makino, T., Shimizu, A., Asai, S., 1983. An As-P double diffused drain MOSFET for VLSI's. IEEE Transactions on Electron Devices 30, 652 – 657. doi:10.1109/T-ED.1983.21184

Tehranipoor, M., Peng, K., Chakrabarty, K., 2012. Delay Test and Small-Delay Defects, in: Test and Diagnosis for Small-Delay Defects. Springer New York, pp. 21–36.

Test Method for Beam Accelerated Soft Error Rate, 2007.

Thambidurai, P., Park, Y., 1988. Interactive consistency with multiple failure modes, in: , Seventh Symposium on Reliable Distributed Systems, 1988. Proceedings. Presented at the , Seventh Symposium on Reliable Distributed Systems, 1988. Proceedings, pp. 93 –100. doi:10.1109/RELDIS.1988.25784

Turski, W.M., Wasserman, A.I., 1978. Computer programming methodology. SIGSOFT Softw. Eng. Notes 3, 20–21. doi:10.1145/1005888.1005894

Underwood, C.I., 1998. The single-event-effect behaviour of commercial-off-the-shelf memory devices-A decade in low-Earth orbit. Nuclear Science, IEEE Transactions on 45, 1450–1457. doi:10.1109/23.685222

Valstar, J.E., 1965. The Contribution of Testability to the Cost-Effectiveness of a Weapon System. IEEE Transactions on Aerospace AS-3, 52 –59. doi:10.1109/TA.1965.4319758

Velazco, R., Fouillat, P., Reis, R.A. da L., 2007. Radiation effects on embedded systems. Springer.

Von Neumann, J., 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components, in: Automata Studies. Princeton University Press, pp. 43–98.

Wallmark, J.T., Marcus, S.M., 1962. Minimum Size and Maximum Packing Density of Nonredundant Semiconductor Devices. Proceedings of the IRE 50, 286–298. doi:10.1109/JRPROC.1962.288321

Warren, K.M., Weller, R.A., Mendenhall, M.H., Reed, R.A., Ball, D.R., Howe, C.L., Olson, B.D., Alles, M.L., Massengill, L.W., Schrimpf, R.D., Haddad, N.F., Doyle, S.E., McMorrow, D., Melinger, J.S., Lotshaw, W.T., 2005. The contribution of nuclear reactions to heavy ion single event upset cross-section measurements in a high-density SEU hardened SRAM. Nuclear Science, IEEE Transactions on 52, 2125–2131. doi:10.1109/TNS.2005.860677

Waskiewicz, A.E., Groninger, J.W., Strahan, V.H., Long, D.M., 1986. Burnout of Power MOS Transistors with Heavy Ions of Californium-252. Nuclear Science, IEEE Transactions on DOI - 10.1109/TNS.1986.4334670 33, 1710–1713.

Weaver, C., Emer, J., Mukherjee, S.S., Reinhardt, S.K., 2004. Techniques to reduce the soft error rate of a high-performance microprocessor, in: Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on. pp. 264 – 275. doi:10.1109/ISCA.2004.1310780

Weaver, H.T., Axness, C.L., McBrayer, J.D., Browning, J.S., Fu, J.S., Ochoa, A., Koga, R., 1987. An SEU Tolerant Memory Cell Derived from Fundamental Studies of SEU Mechanisms in SRAM. IEEE Transactions on Nuclear Science 34, 1281–1286. doi:10.1109/TNS.1987.4337466

Williams, T.W., Kapur, R., Mercer, M.R., Dennard, R.H., Maly, W., 1996. Iddq testing for high performance CMOS-the next ten years, in: European Design and Test Conference, 1996. ED TC 96. Proceedings. Presented at the European Design and Test Conference, 1996. ED TC 96. Proceedings, pp. 578 –583. doi:10.1109/EDTC.1996.494359

Wirth, G.I., Vieira, M.G., Neto, E.H., Kastensmidt, F.L., 2008. Modeling the sensitivity of CMOS circuits to radiation induced single event transients. Microelectronics Reliability 48, 29–36. doi:10.1016/j.microrel.2007.01.085

Wirth, N., 1983. Programming in Modula-2. Springer-Verlag TELOS.

Wirth, N., 1988. The programming language Oberon. Softw. Pract. Exper. 18, 671–690.

Wirth, N., 1992. Project Oberon: The Design of an Operating System and Compiler. Addison-Wesley Pub (Sd).

Wood, A., 1999. Data Integrity Concepts, Features, and Technology. White paper, Tandem Division, Compaq Computer Corporation.

Woodard, S.E., Metze, G., 1978. Self-checking alternating logic: Sequential circuit design, in: Proceedings of the 5th Annual Symposium on Computer Architecture, ISCA '78. ACM, New York, NY, USA, pp. 114–122. doi:10.1145/800094.803037

Wrobel, F., Hubert, G., Iacconi, P., 2006. A Semi-empirical Approach for Heavy Ion SEU Cross Section Calculations. Nuclear Science, IEEE Transactions on 53, 3271–3276. doi:10.1109/TNS.2006.886200

Wyatt, R.C., McNulty, P.J., Toumbas, P., Rothwell, P.L., Filz, R.C., 1979. Soft Errors Induced by Energetic Protons. Nuclear Science, IEEE Transactions on 26, 4905–4910. doi:10.1109/TNS.1979.4330248

Xiaoqing, W., Saluja, K.K., Kinoshita, K., Tamamoto, H., 1996. Equivalence fault collapsing for transistor leakage faults, in: , IEEE International Workshop on IDDQ Testing, 1996. Presented at the , IEEE International Workshop on IDDQ Testing, 1996, pp. 79 –83. doi:10.1109/IDDQ.1996.557836

Xiaowei Zhu, Baumann, R., Pilch, C., Zhou, J., Jones, J., Cirba, C., 2005. Comparison of product failure rate to the component soft error rates in a multi-core digital signal processor, in: Reliability Physics Symposium, 2005. Proceedings. 43rd Annual. 2005 IEEE International. Presented at the Reliability Physics Symposium, 2005. Proceedings. 43rd Annual. 2005 IEEE International, pp. 209–214. doi:10.1109/RELPHY.2005.1493086

Yu Qingkui, Tang Min, Zhu Hengjing, Zhang Haiming, Zhang Yanwei, Sun Jixing, 2005. Experimental Investigation of Radiation Damage on CCD with Protons and Cobalt 60 Gamma Rays, in: Radiation and Its Effects on Components and Systems, 2005. RADECS 2005. 8th European Conference on. Presented at the Radiation and Its Effects on Components and Systems, 2005. RADECS 2005. 8th European Conference on, pp. LNW2–1–LNW2–5. doi:10.1109/RADECS.2005.4365667

Zhang, B., Tang, L., DeCastro, J., Goebel, K., 2011. Prognostics-enhanced Receding Horizon Mission Planning for Field Unmanned Vehicles, in: AIAA Guidance, Navigation, and Control Conference. American Institute of Aeronautics and Astronautics.

Ziegler, J.F., 1996. Terrestrial cosmic rays [WWW Document]. IBM Journal of Research and Development. URL 10.1147/rd.401.0019

Ziegler, J.F., Lanford, W.A., 1979. Effect of Cosmic Rays on Computer Memories. Science 206, 776–788. doi:10.1126/science.206.4420.776

Ziegler, J.F., Puchner, H., 2004. SER--history, Trends and Challenges: A Guide for Designing with Memory ICs. Cypress.

Ziegler, J.F., Ziegler, M.D., Biersack, J.P., 2010. SRIM - The stopping and range of ions in matter (2010). Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms 268, 1818–1823. doi:10.1016/j.nimb.2010.02.091

Appendix A

ERRIC's CPU Logical Structure

and

Instruction Set

1 CPU Logical Structure

Logically, the Processor Unit consists of the following components:

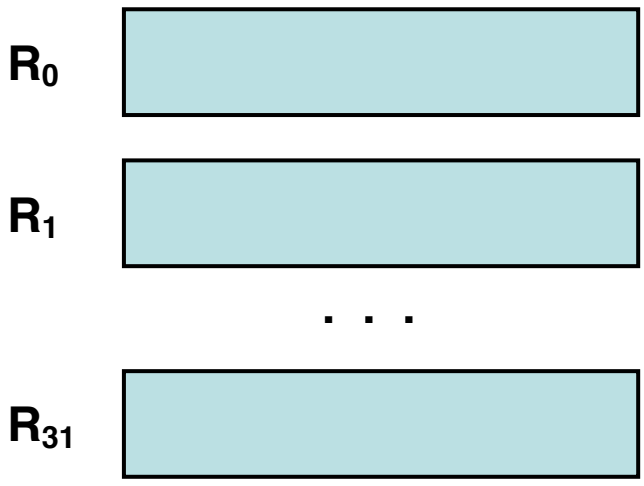
- Central processor unit itself (CPU)
- Common purpose registers (R0, … R31)
- Special purpose registers (IR, PC)
- Random access memory (RAM).

Below is the short characteristics of the components mentioned.

Processor has only two special register IR and PC, and they are not accessible for programmers. The memory image in the example below o shows mapping and using of the general purpose registers (R0,…,R31) for SB, FP etc.

1.1 Common Purpose Registers

- 32 common registers (R0…R31) are of 32 bits each.
- Every register can contain either a value or an address of a memory location.
- An address in a register can address any byte of memory.
- The CPU performs all actions on the operands taken from common registers. There are instructions for loading values from the memory to a register, and for storing a value from a register to the memory.



1.2 Special Purpose Registers

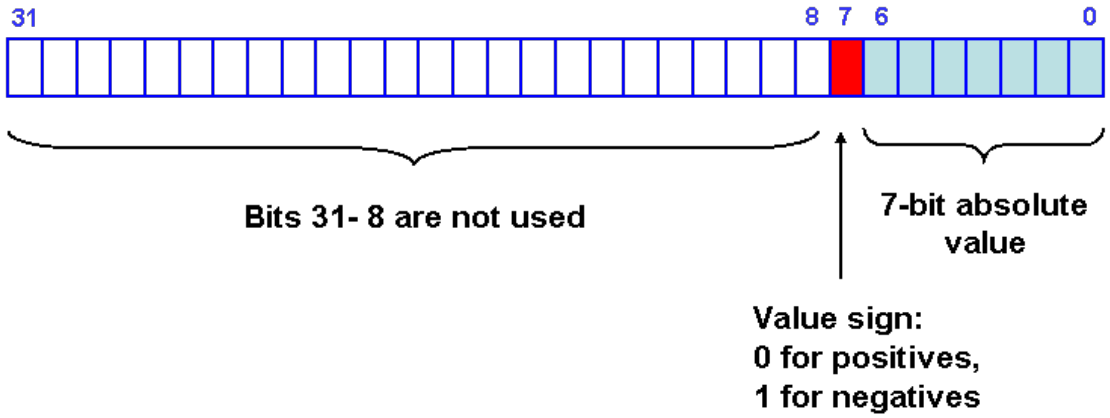
- **PC register** (Program Counter) contains 32-bit address of the leftmost (high) byte of the instruction which is being currently executed. After the current instruction is completed, the address in PC is normally increased by 2 addressing the next instruction (because every instruction occupies 2 bytes, see later). The exception is CBR instruction which can alter this behavior setting the new address on the PC taking it from a common register. There are no other ways to modify the contents of the PC register.

1.3 Supported Values

- The CPU operates with 8-, 16-, and 32-bit values.
- The values of all formats are considered either as **signed integers** (arithmetic ADD and SUB instructions, and arithmetic shift ASL, ASR instructions) or as **bit scales** (logical shift LSL, LSR instructions and logical AND, OR, and XOR instructions).
- Positive integer values are represented **in the direct code** (with 0 in the sign bit). Negative integers are represented **in the two's complement code**. See ISO-XXXX for details.

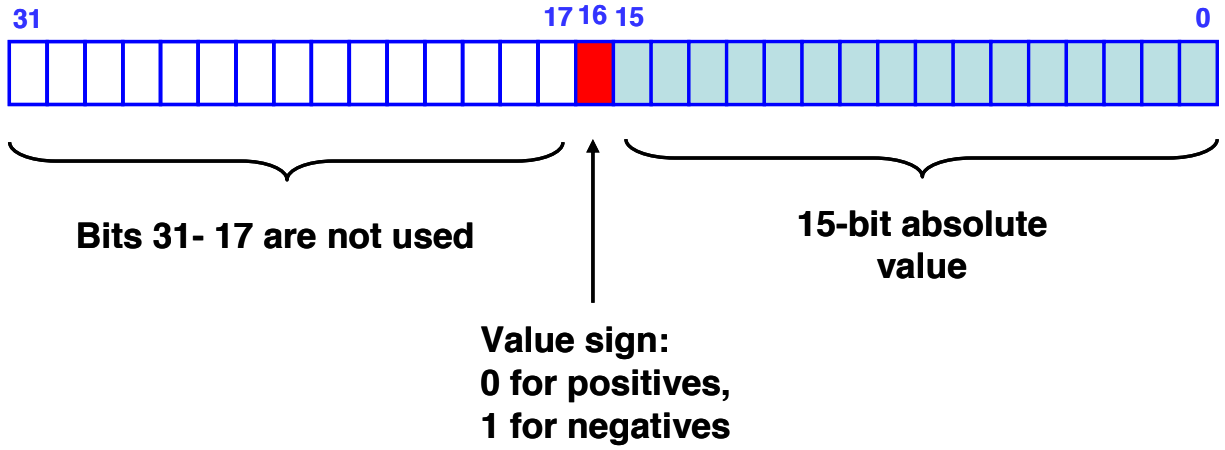
8-bit Values

- The range of possible 8-bit integer values is [-128..127].
- Alignment requrements
- The format of 8-bit signed integers is shown below:



16-bit Values

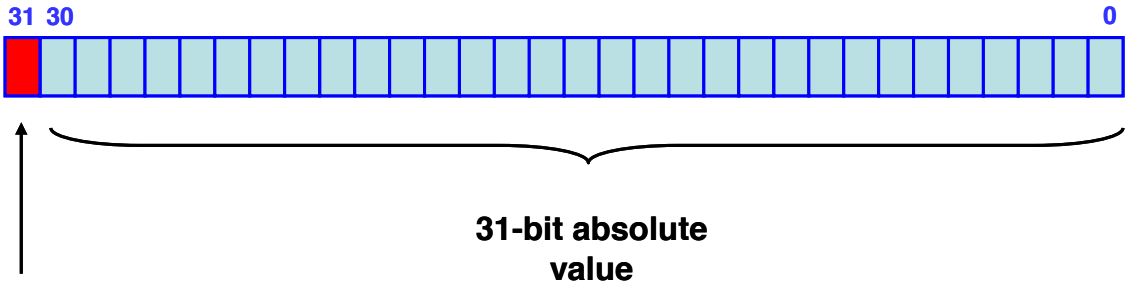
- The range of possible 16-bit integer values is [-32768..32767].
- Alignment requirements
- The format of 16-bit signed integers is shown below:



Here is a mistake. 16 bit value takes 0-15 bits and 15th bit is sign. We count from 0, not from 1, thus 15th bit is a bit number 16.

32-bit Values

- The range of possible 32-bit integer values is [-2147483648..2147483647]
- Alignment requirements
- The format of 32-bit signed integers is shown below:



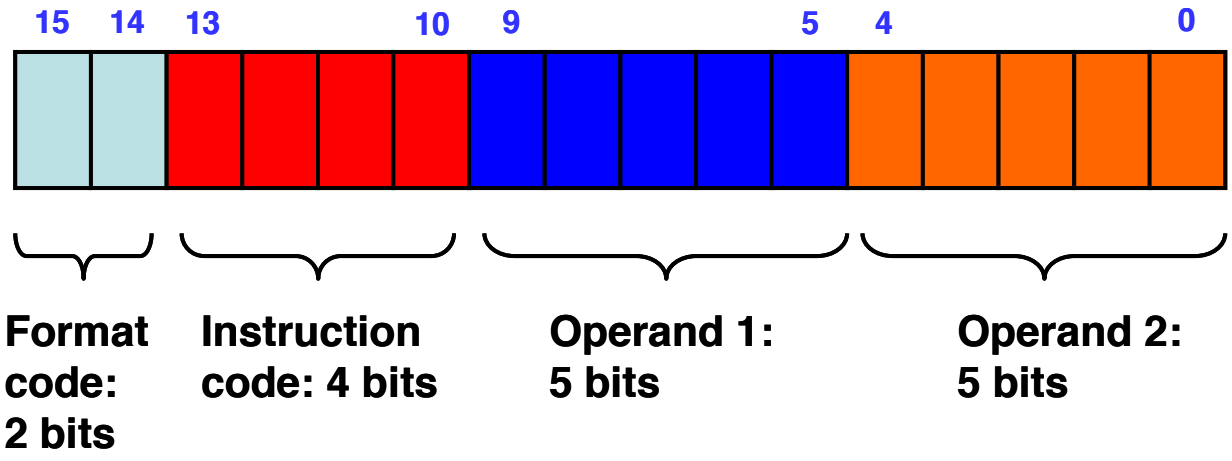
Non-supported Values

- Long (64-bit) integer and floating point values are not directly supported by the CPU. If necessary, the support can be provided by using special software routines using predefined library.
- The floating-point types should be conceptually associated with the 32-bit single-precision and 64-bit double-precision IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

2 Instruction Set

2.1 Common Instruction Format

- Every instruction occupies 16 bits (two bytes).
- **Format code** (bits 15-14 of the instruction) encodes format of operands:
  00 is for 8 bit (lowest 8 bits of operands participate in the operation),
  01 is for 16 bit (lowest 16 bits of operands participate in the operation),
  10 is reserved,
  11 is for 32 bits (entire 32 bits of operands participate in the operation).
- **Instruction code** (bits 13-10 of the instruction) encodes the operation kind of the instruction. There are 16 main kinds of instructions coded by 0x0, 0x1, … 0xF.
- Two **operands** (bits 9-5 and 4-0 of the instruction) always contain register codes (numbers within the range of 0..31).
- Alignment requirements
- The overall instruction's layout is shown below:

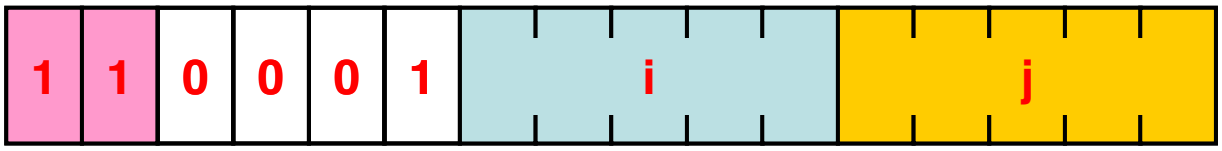


2.1 LD Instruction

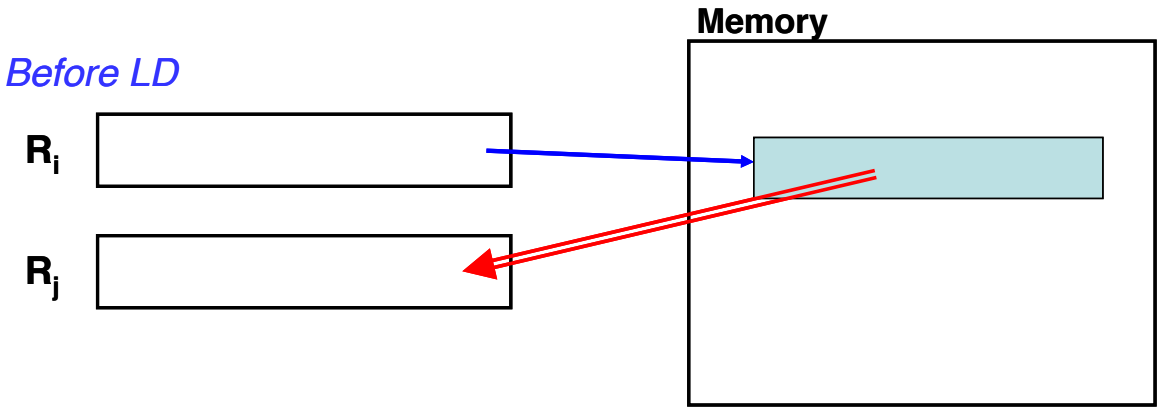
- **LD Ri, Rj Load from the memory**
- **The LD instruction copies the value of a 32-bit memory word pointed to by Ri register, to the Rj register**.
- The suggested assembly statement for the LD instruction is:

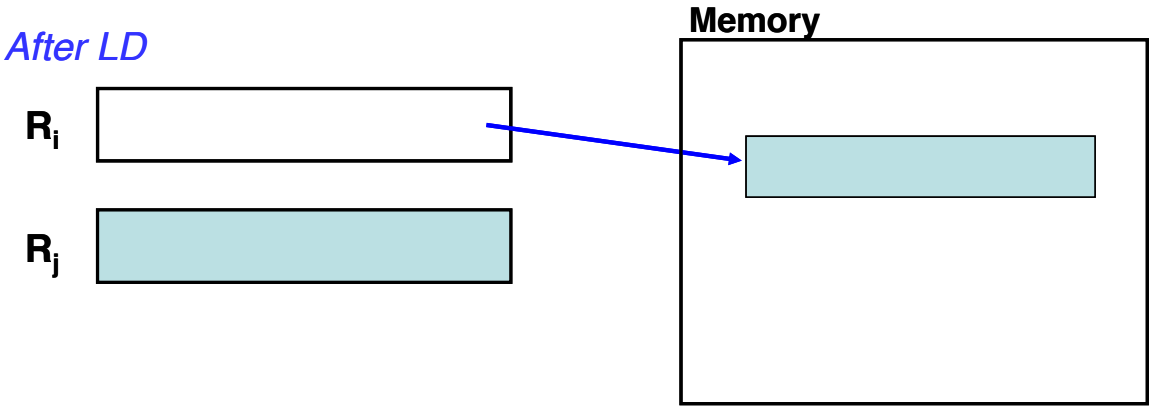
```
Rj := *Ri
```

- The instruction format is as follows:



- The contents of the register Ri is considered as a 32-bit address of a 32-bit memory word.
- Instruction format is always **16 bit**, i.e., the entire 32-bit word is copied from the memory to the register contains 2 isntructions.
- When the instruction is completed, the original contents of the register Rj is lost.
- Registers Ri and Rj can be the same register. If not, the contents of the Ri register (i.e., the address) does not change.
- The effect of the LD instruction is shown below:





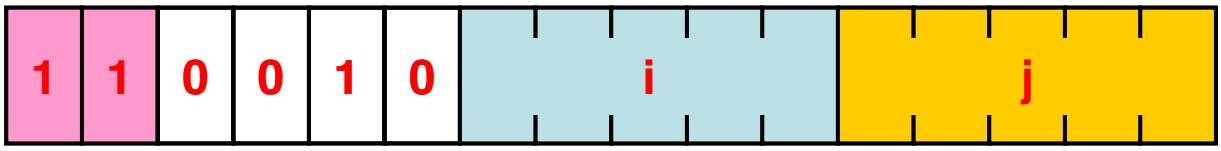
2.2 LDA Instruction (Load address from memory)

- **The LDA instruction takes the value from the next 32-bit word, then adds it to the current value of the Ri register, and stores the result to the Ri register. (LDA Ri, Rj ???)**

Eugene, your semantics is different with ours. For your LDA version, it is efficient for indexed addressing and can facilitate procedural calls. However, it slows down the direct addressing which is more often in programs. When you need directly addressing constants or variables, you need to load a ZERO into Ri and then perform the operation.

For us, the semantics of **LDA Rj** is to load the next 32-bit word into register **Rj**. For index addressing, it can be achieve by two instructions (**LDA Rj; ADD Ri, Rj).** Here, it is the choice between RISC instruction and complex instructions**. I do agree, as we stand for two operand instruction set with explicit "manifestation" of them, and I believe we can progress here immediately if Eugene agree with this approach. Probably write couple of more sentences why we are choosing this approach will be good, three variables two actions sum and access, while semantic of instruction should remind what we are doing – i.e. read or write, not add or delete.**

- The instruction format is as follows:



- The contents of the register Ri is considered as a 32-bit address of a 32-bit memory word.
- The next 32-bit word (pointed to by the PC register) is considered as a constant that loaded.
- When the instruction is completed, the original contents of the register Rj is lost.
- Alignment requirements for the LDA instruction are as follows:
  - The LDA instruction must occupy the left two bytes of the 32-bit word (not necessary please see the example code and early comments).
  - The right two bytes must be left empty. (not necessary, see early comments)
  - The constant must occupy the entire 32-bit word next to the word with the LDA instruction.
- After the instruction has completed its execution, the next instruction to execute is fetched by the address PC+2 (but not PC+1 as for all other instructions).

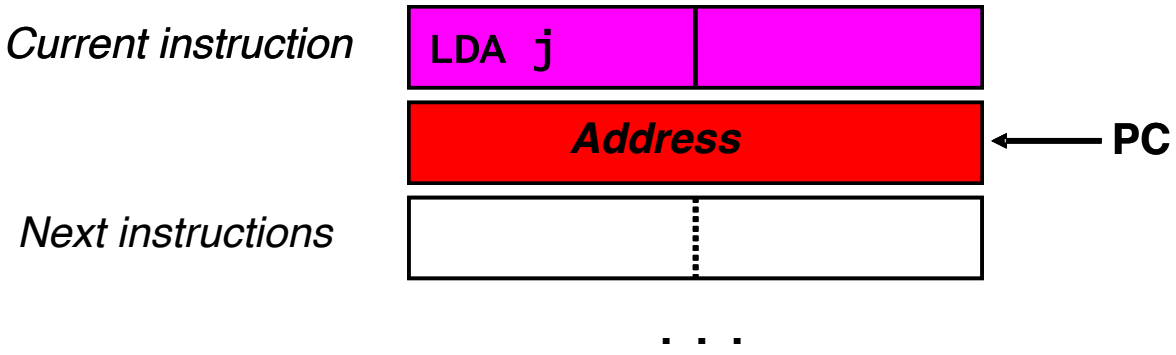
  For 32 bit memory, PC is increased only after 32 bits code (**2 instructions**) or 32 bits data have been processed. Each time, two instructions will be fetched.

  Proper alignment is necessary for dealing with subroutine.

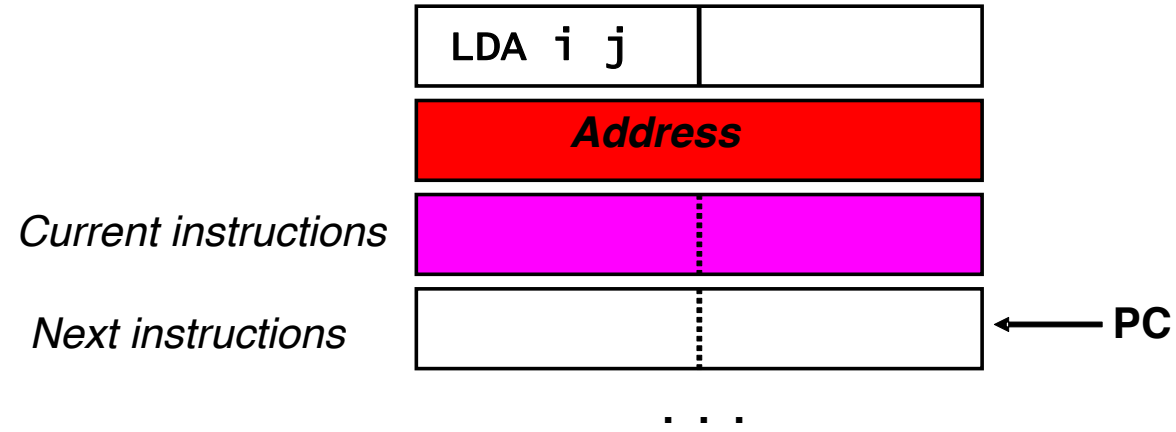
- **The suggested assembly statement for the LDA instruction:**
  HAO, PLS WRITE AS WE DISCUSS, CORRECT AS WE DISCUSS. USING AS MUCH AS POSSIBLE THE SAME FIGURES EUGENE DID.
- **Rj := Address**

- The scheme of how code is being processed is shown below:

Before LDA



After LDA



- The effect of the LDA instruction is shown below

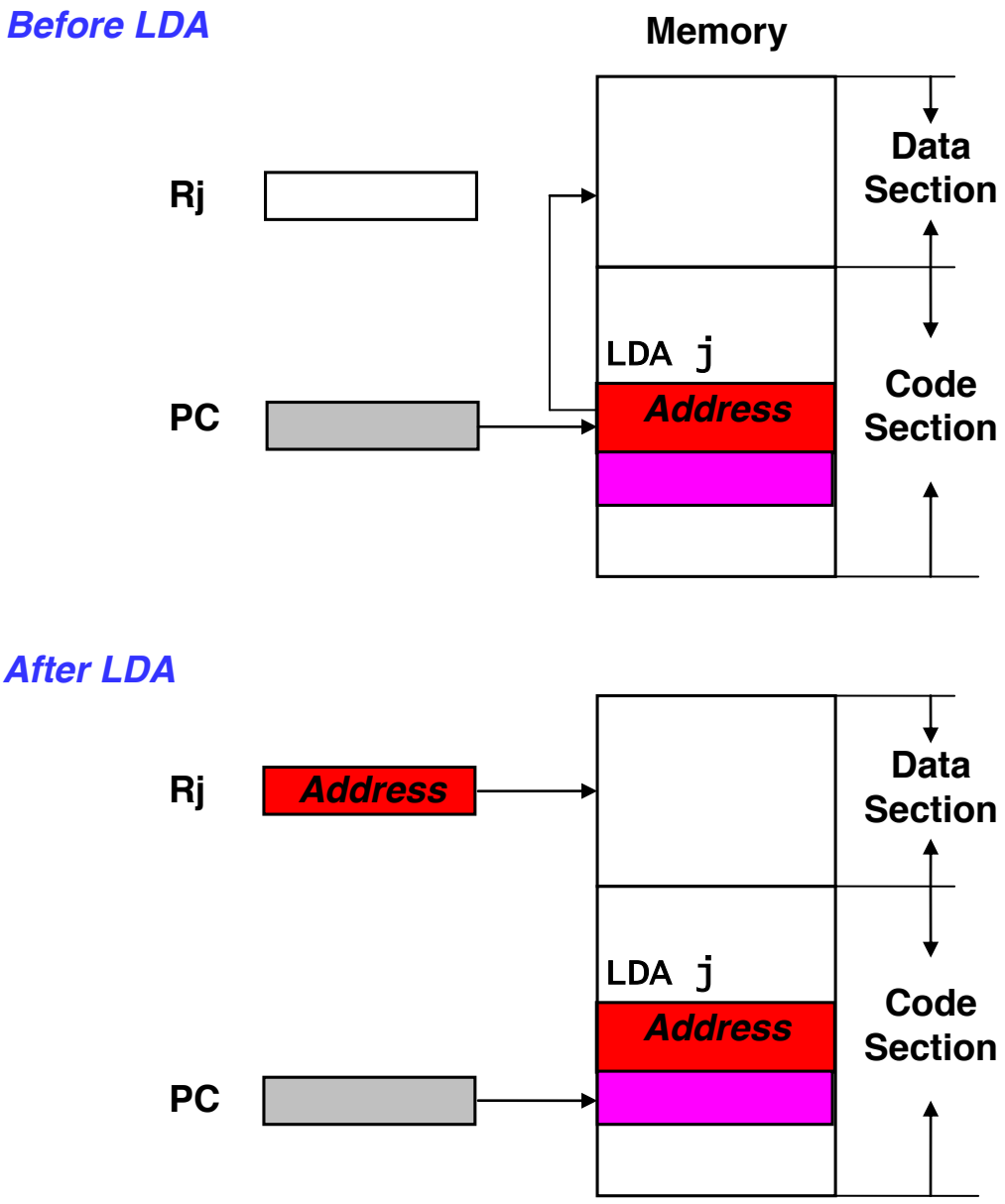
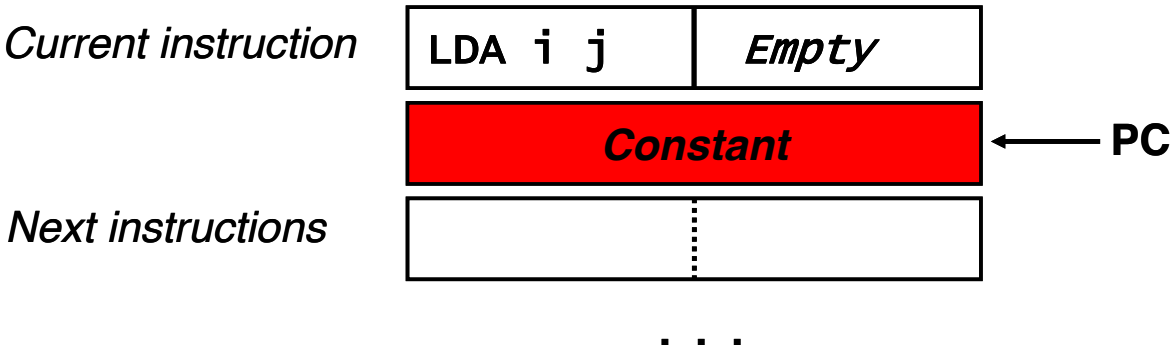


Fig. XX. The Effect of the LDA instruction (LMU version)

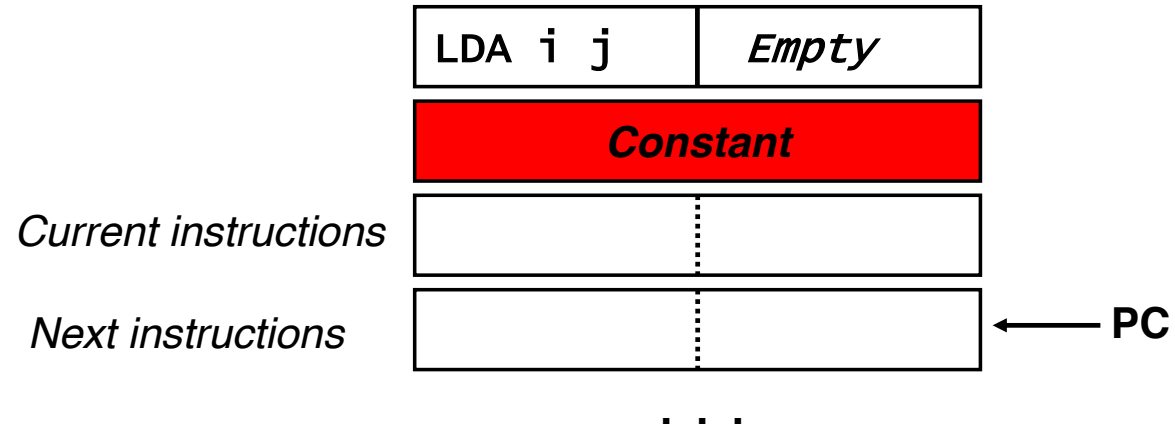
Eugene's version of LDA

-
```
Rj := Ri + constant
Rj := Ri - constant
```
- The scheme of how code is being processed is shown below:

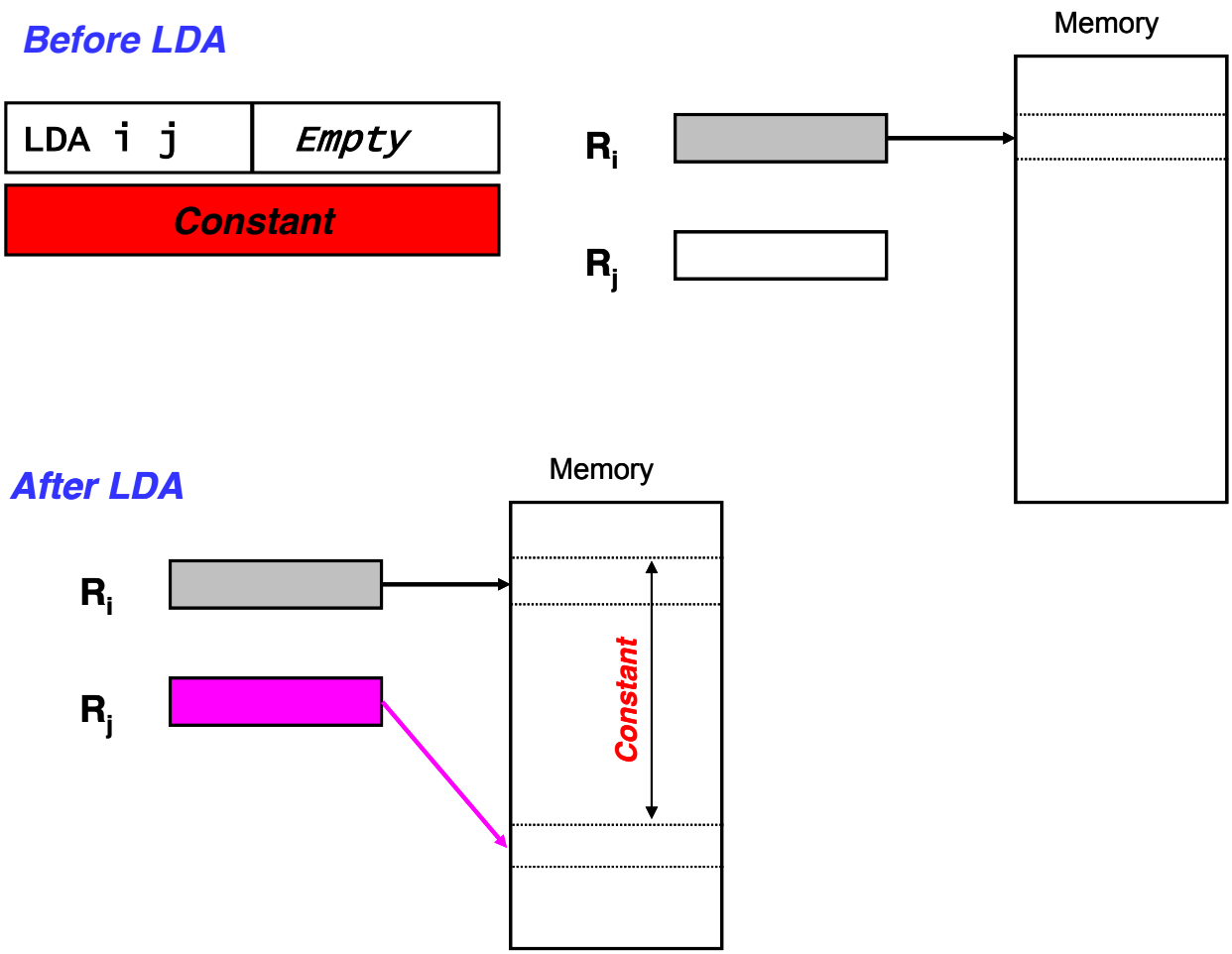
Before LDA



After LDA



- The effect of the LDA instruction is shown below



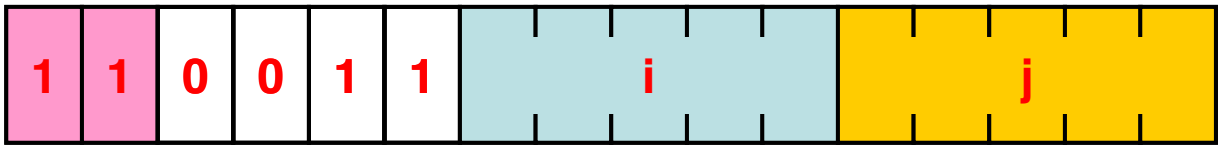
CPU has 32 general registers. It is acceptable to use four of them for procedural calls. It might be interesting to compare with other CPUs for dealing with procedural calls and see how many registers will be involved for other CPUs.

common ones. Therefore, we need the LDA instruction to work both with common registers and with SB/FP. It makes LDA to complicated to implement.

So the overall conclusion is that 1) the common semantics of the LDA instruction presented at the beginning of this section is the most desired one, and 2) we do not need any special purpose registers.

2.3 ST Instruction

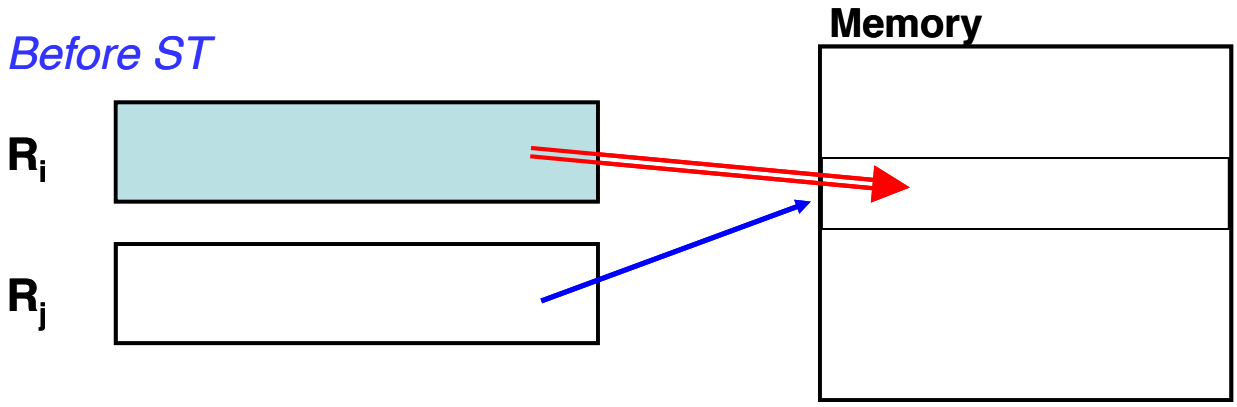
- **The ST instruction copies the value of the register Ri to the memory by address taken from the register Rj**.
- The instruction format is as follows:

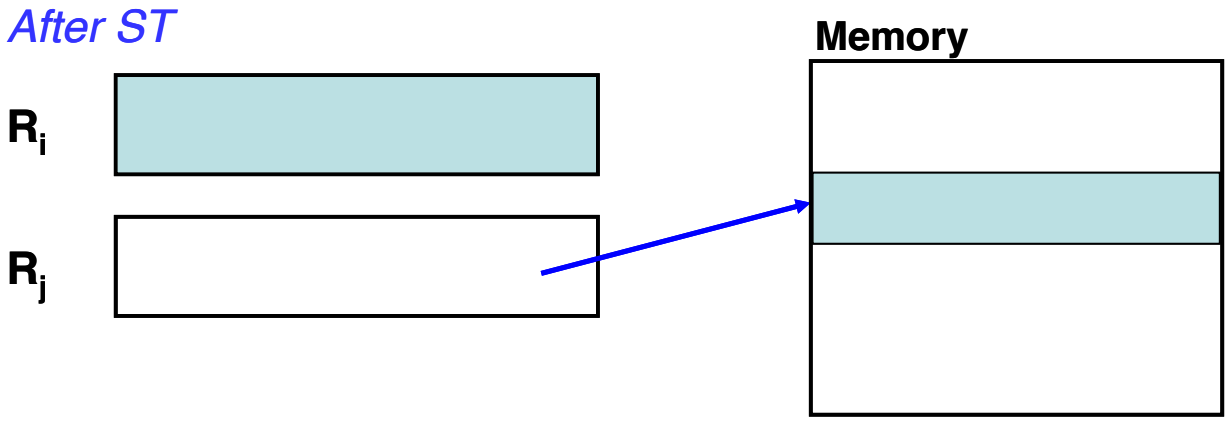


- The contents of the register Ri is treated as an arbitrary value. The contents of the register Rj is considered as a 32-bit address of a 32-bit memory word.
- Instruction format is always **32**, i.e., the entire 32-bit register is copied to the memory.
- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of Ri and Rj registers do not change.
- Suggested assembly statement for the ST instruction is:

```
*Rj := Ri
```

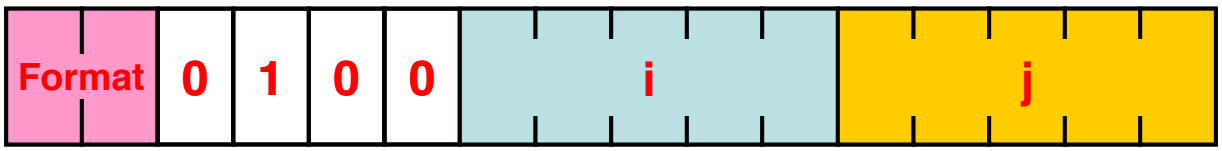
- The effect of the ST instruction is shown below:





2.4 MOV Instruction

- **The MOV instruction copies the value from register Ri to the register Rj**.
- Memory state is not considered in the instruction, and the memory state does not change.
- The instruction format is as follows:



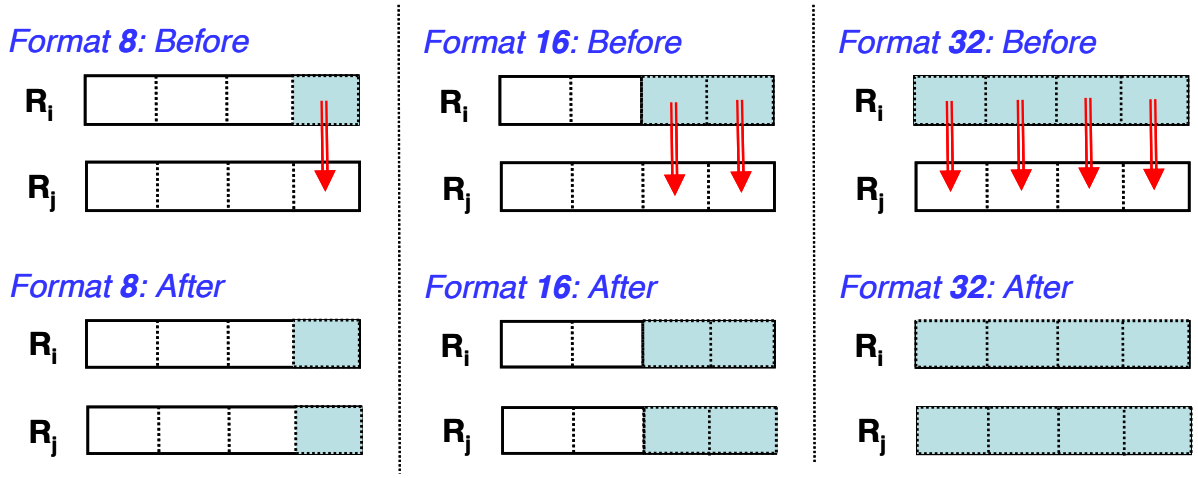
- Suggested assembly statement for the MOV instruction is:

```
Rj := Ri
```

- Additional assembly directives specifying the current instruction format:

```
.format 8     or     .format 16     or     .format 32
```

- Memory state is not considered in the instruction, and the memory state does not change.
- The effect of the MOV instruction is shown below:



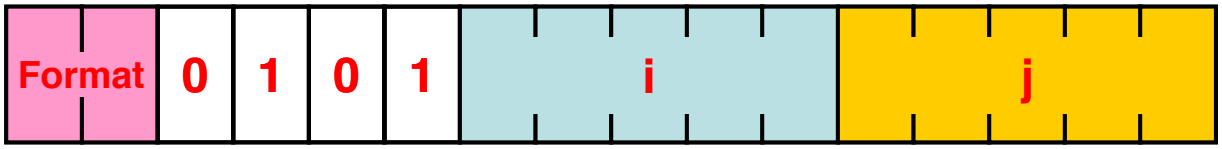
- Instruction format 8: the lowest byte is copied; three highest bytes of Rj remain the same. The original value of Rj's lowest byte is lost.

  Instruction format 16: two lowest bytes are copied; two highest bytes of Rj remain the same. The original value of Rj's two lowest bytes is lost.

  Instruction format 32: the entire 32-bit register is copied. The original value of Rj is lost.
- Memory state is not considered in the instruction, and the memory state does not change.

2.5 ADD Instruction

- **The ADD instruction denotes the two's complement arithmetic addition. The contents of registers Ri and Rj are arithmetically added, and the result is put into the register Rj.**
- The instruction format is as follows:



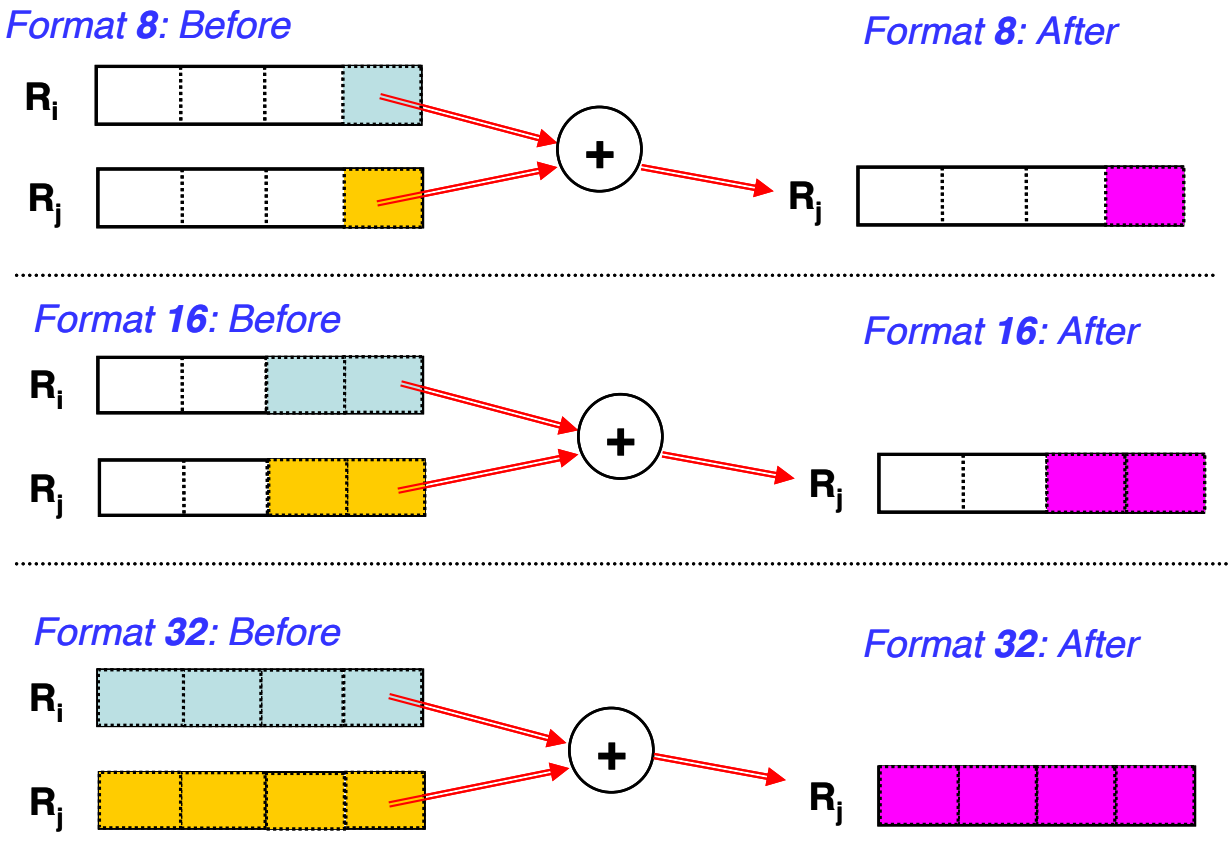
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register.
- If the addition gives a result which cannot be put into the format specified in the instruction, then **overflow** happens: ?????
- **SIGNS MUST BE CHANGED ACCORDINGLY IN BOTH INSTRUCTIONS ARITHMETIC INSTRUCTIONS. PLS CORRECT.**
- Suggested assembly statement for the ADD instruction:

```
Rj += Ri
```

- Additional assembly directives specifying the current instruction format:

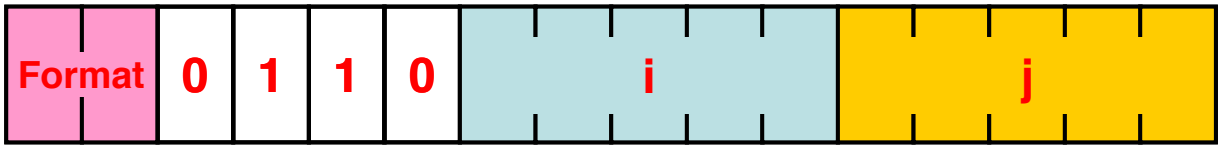
```
.format 8    or    .format 16    or    .format 32
```

- The effect of the ADD instruction is shown below:



2.6 SUB Instruction

- **The SUB instruction denotes the two's complement arithmetic subtraction. The contents of register Ri is subtracted from the contents of the register Rj, and the result is put into the register Rj.**
- The instruction format is as follows:



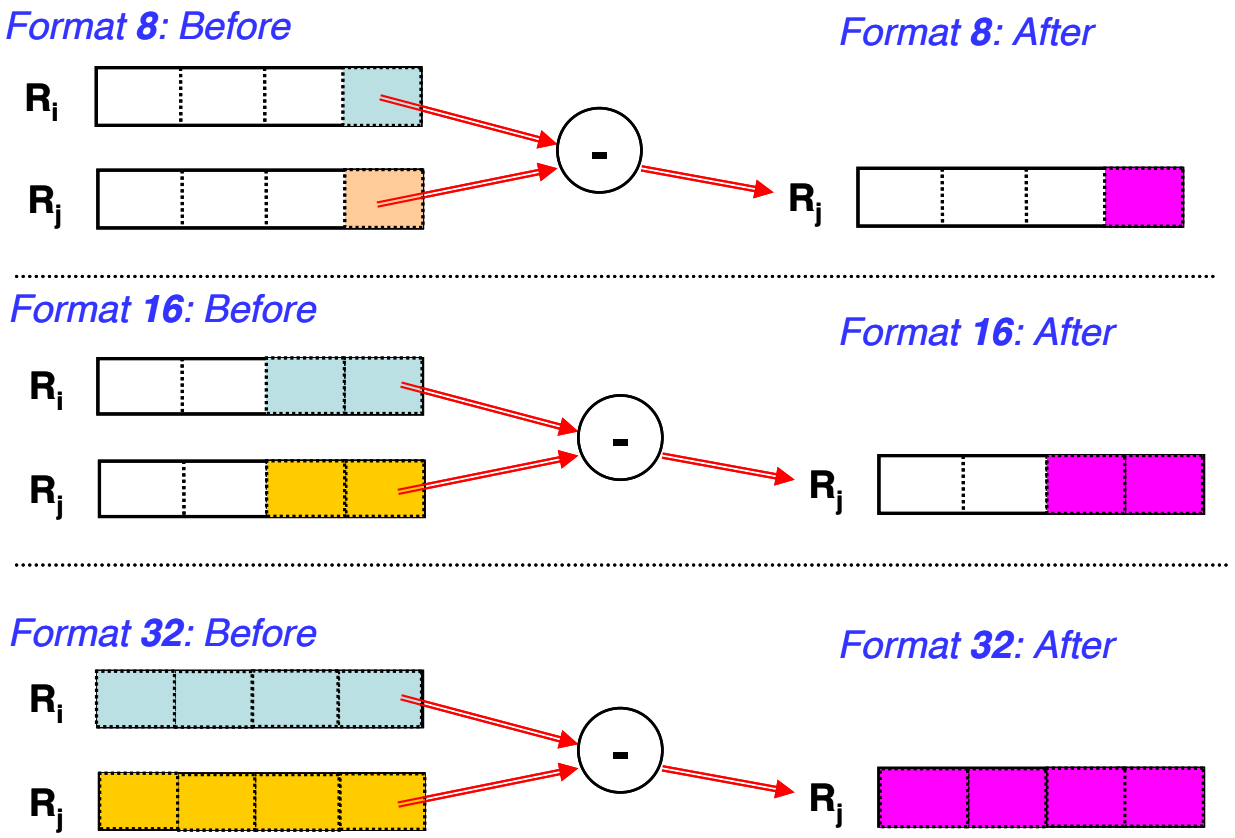
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register.
- If the subtraction gives a result which cannot be put into the format specified in the instruction, then ..... happens: ?????
- Suggested assembly statement for the SUB instruction:

```
Rj -= Ri
```

- Additional assembly directives specifying the current instruction format:

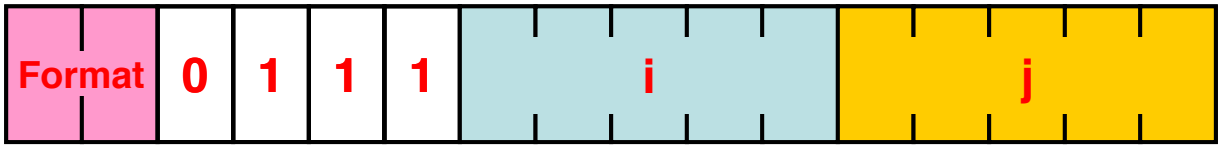
```
.format 8    or    .format 16    or    .format 32
```

- The effect of the SUB instruction is shown below:



2.7 ASR Instruction

- **The ASR instruction arithmetically shifts the contents of the register Ri <u>one bit right</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



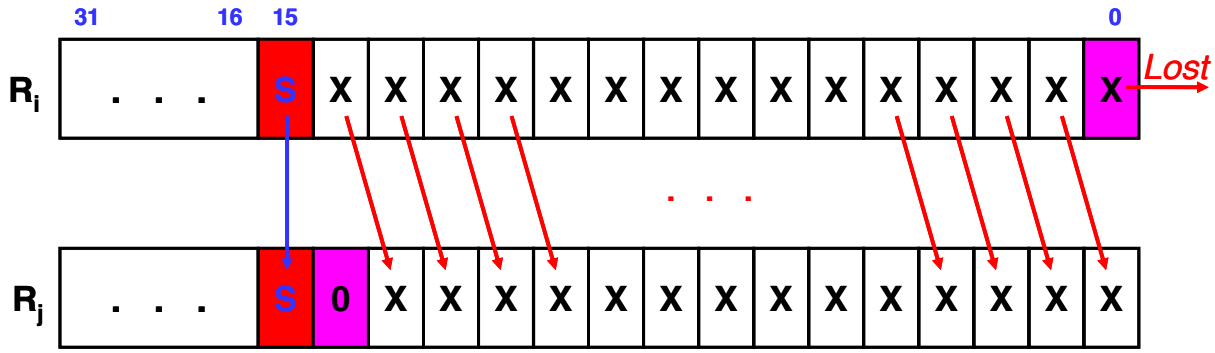
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register.
- Suggested assembly statement for the ASR instruction:

```
Rj >>= Ri;
```

- Additional assembly directives specifying the current instruction format:

```
.format 8    or    .format 16    or    .format 32
```

- *Arithmetic* shift means that the sign bit does not participate in the operation but remains on its usual place.
- The leftmost bit of the operand gets the value of 0. The rightmost bit of the operand is always lost.
- The contents of the Ri register does not change.
- The effect of the ASR instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.
- The effect of the ASR instruction for the case of 16-bit operands is shown below:

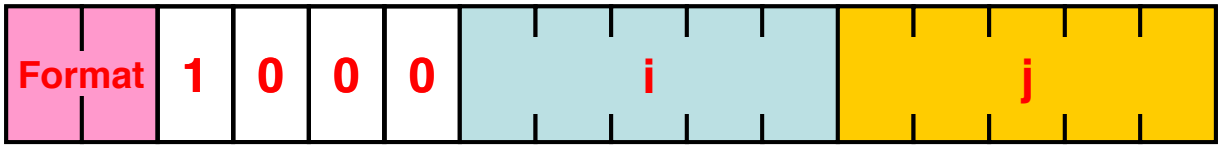


What to do with the high bits of the result for 8 and 16 formats?

- Copy them from the source register.
- Remain them as they were (no modifications).
- Set them to 0s. **BEST OPTION,**

2.8 ASL Instruction

- **The ASL instruction arithmetically shifts the contents of the register Ri <u>one bit left</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



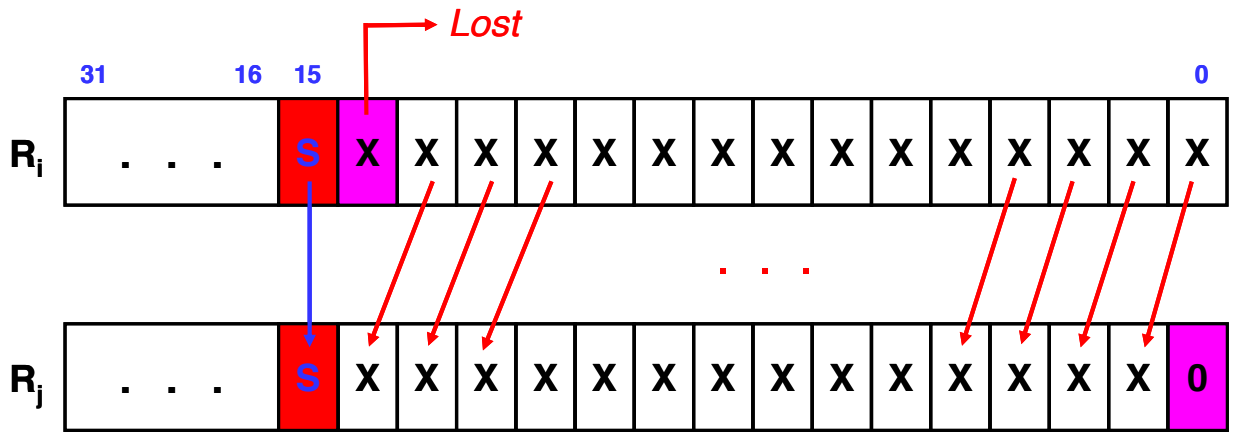
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register.
- Suggested assembly statement for the ASL instruction:

  Rj <<= Ri

- Additional assembly directives specifying the current instruction format:

  .format 8 or .format 16 or .format 32

- *Arithmetic* shift means that the sign bit does not participate in the operation but remains on its usual place.
- The leftmost bit of the operand is always lost. The rightmost bit of the operand gets the value of 0.
- The contents of the Ri register does not change.
- The effect of the ASL instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.

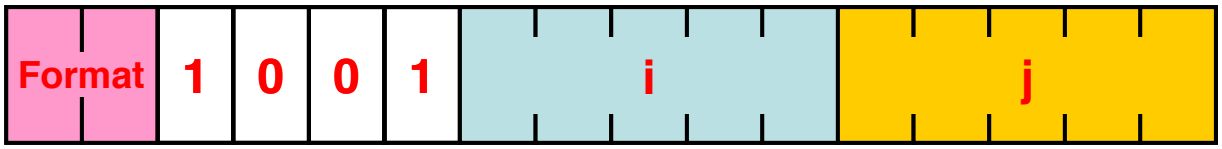


What to do with the high bits of the result for 8 and 16 formats?

- Copy them from the source register.
- Remain them as they were (no modifications).
- Set them to 0s. BEST OPTION

2.9 OR Instruction

- **The OR instruction applies logical addition ("OR") operator to every pair of bits taken from registers Ri and Rj, respectively, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register. If not, the contents of the Ri register does not change.
- Suggested assembly statement for the OR instruction:

```
Rj |= Ri
```

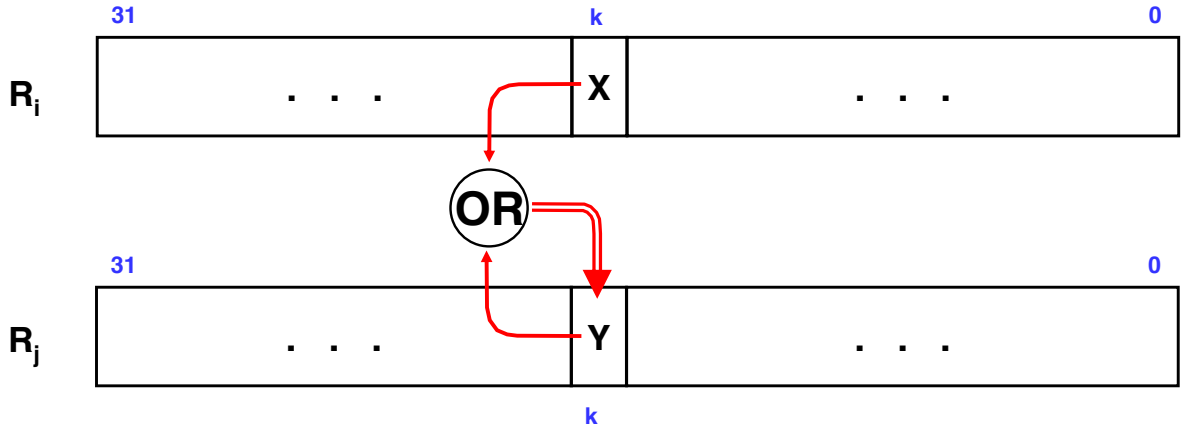
- Additional assembly directives specifying the current instruction format:

```
.format 8    or    .format 16    or    .format 32
```

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every pair of bits independently.
- The rule for the **OR** operation performed on each pair of bits is defined as follows:

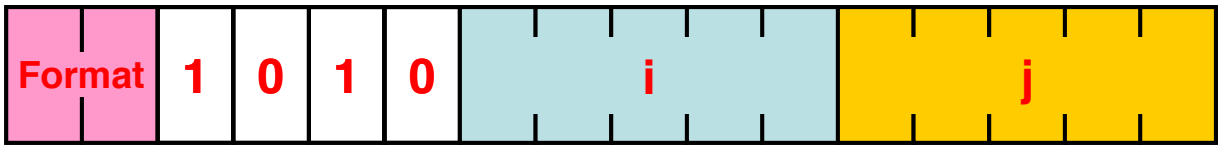
X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	1

- The mechanism of the OR instruction – for one pair of bits - is shown below. Here, k∈[0..31] for format 32, k∈[0..15] for format 16, and k∈[0..7] for format 8.



2.10 AND Instruction

- **The AND instruction applies logical multiplicative ("AND") operator to every pair of bits taken from registers Ri and Rj, respectively, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register. If not, the contents of the Ri register does not change.
- Suggested assembly statement for the AND instruction:

```
Rj &= Ri
```

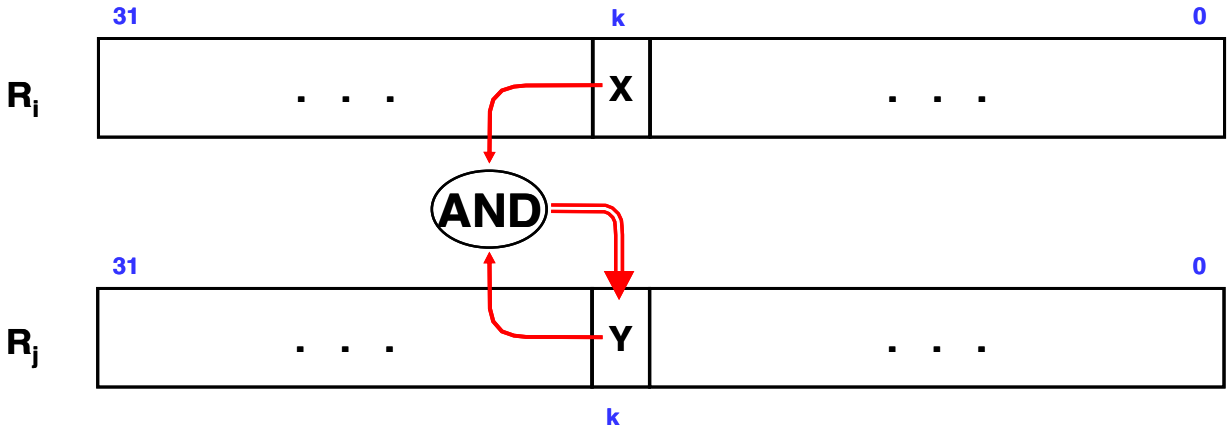
- Additional assembly directives specifying the current instruction format:

```
.format 8   or   .format 16   or   .format 32
```

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every pair of bits independently.
- The rule for the AND operation performed on each pair of bits is defined as follows:

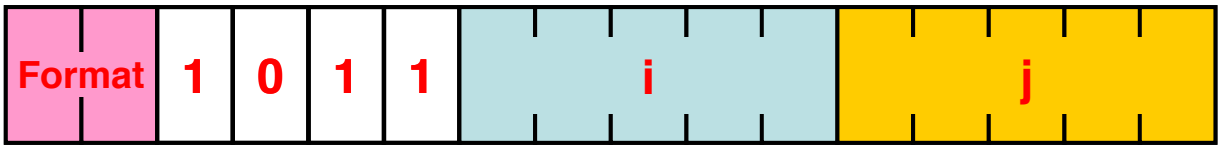
X	Y	Result
0	0	0
0	1	0
1	0	0
1	1	1

- The mechanism of the AND instruction – for one pair of bits - is shown below. Here, k∈[0..31] for format 32, k∈[0..15] for format 16, and k∈[0..7] for format 8.



2.11 XOR Instruction

- **The XOR instruction applies logical exclusive OR ("XOR") operator to every pair of bits taken from registers Ri and Rj, respectively, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register. If not, the contents of the Ri register does not change.
- Suggested assembly statement for the XOR instruction:

```
Rj ^= Ri
```

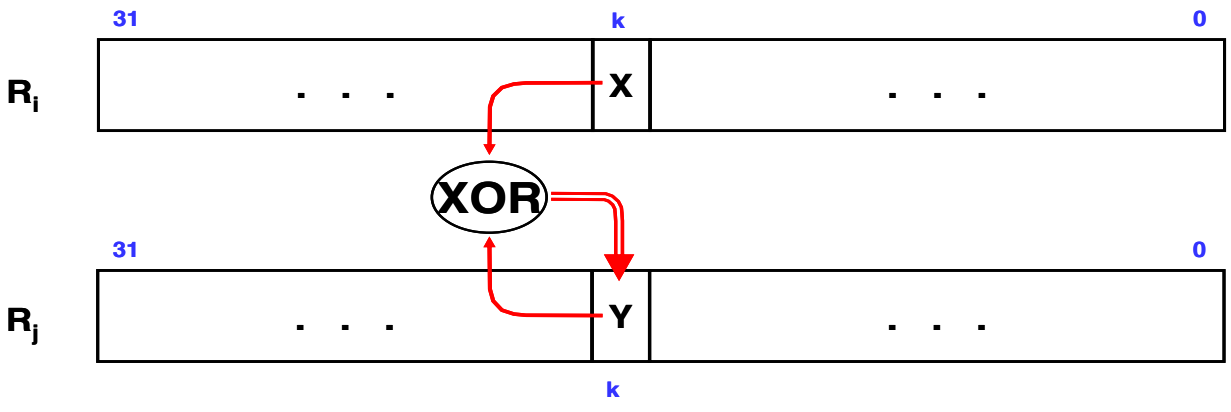
- Additional assembly directives specifying the current instruction format:

```
.format 8    or    .format 16    or    .format 32
```

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every pair of bits independently.
- The rule for the XOR operation performed on each pair of bits is defined as follows:

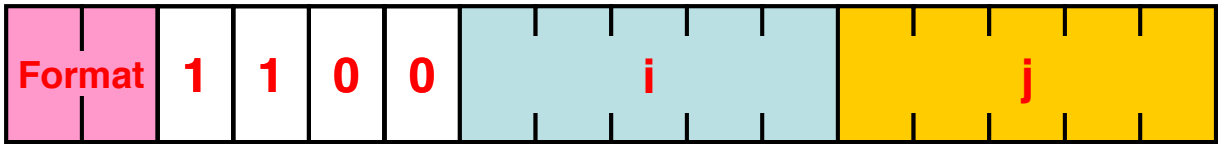
X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	0

- The mechanism of the XOR instruction – for one pair of bits - is shown below. Here, k∈[0..31] for format 32, k∈[0..15] for format 16, and k∈[0..7] for format 8.



2.12 LSL Instruction

- **The LSL instruction logically shifts the contents of the register Ri <u>one bit left</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



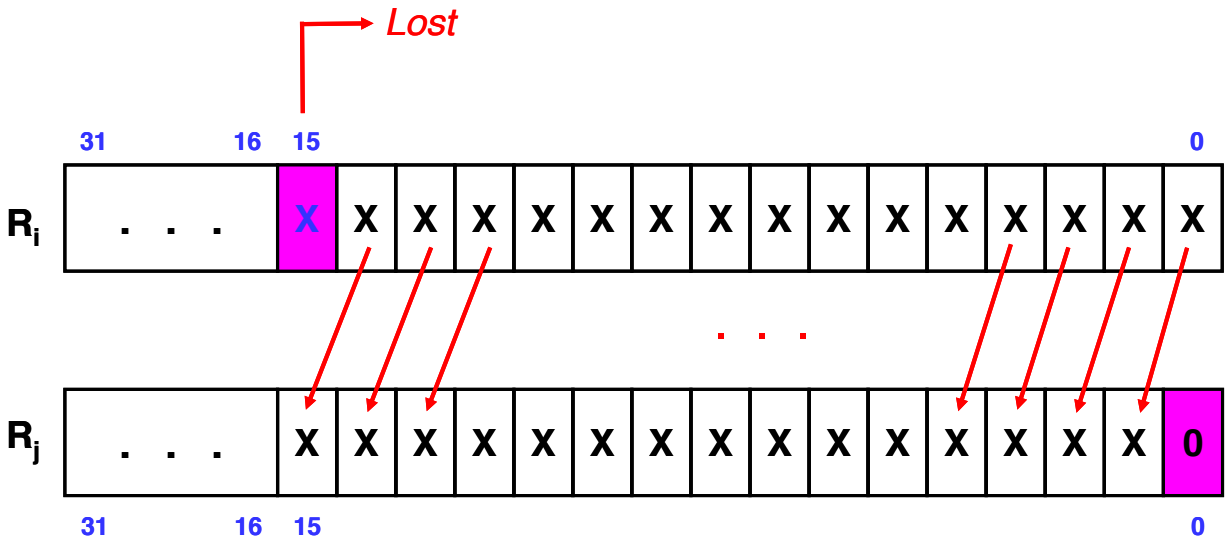
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register. If not, the contents of the Ri register does not change.
- Suggested assembly statement for the LSL instruction:

```
Rj <= Ri
```

- Additional assembly directives specifying the current instruction format:

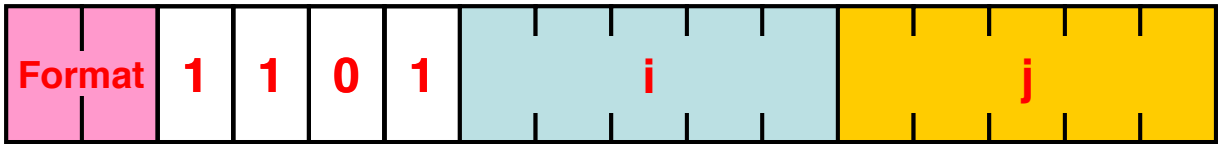
```
.format 8    or    .format 16    or    .format 32
```

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every bit independently.
- The leftmost bit of the operand is always lost. The rightmost bit of the operand gets the value of 0.
- The effect of the LSL instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.



2.13 LSR Instruction

- **The LSR instruction logically shifts the contents of the register Ri <u>one bit right</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



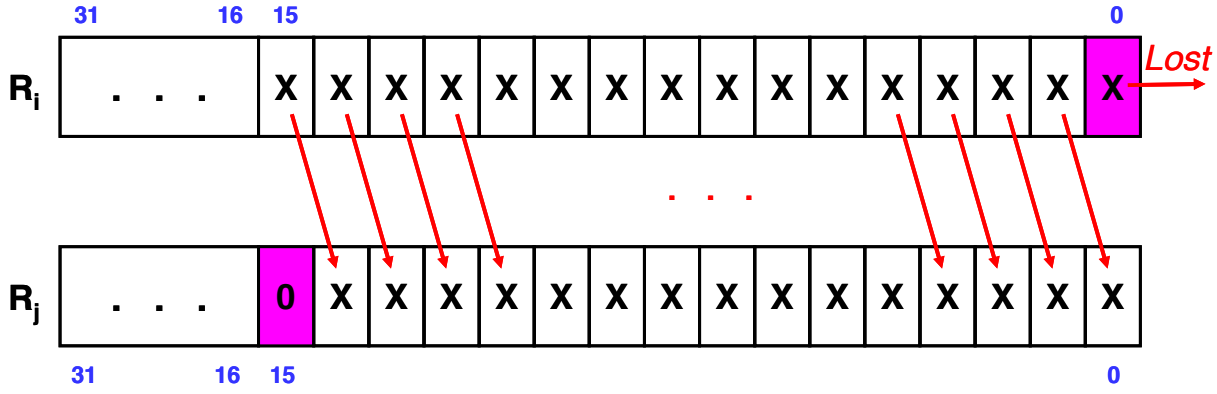
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register. If not, the contents of the Ri register does not change.
- Suggested assembly statement for the LSR instruction:

```
Rj >= Ri
```

- Additional assembly directives specifying the current instruction format:

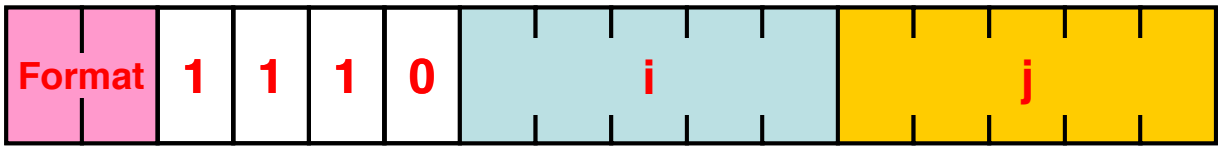
```
.format 8    or    .format 16    or    .format 32
```

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every bit independently.
- The rightmost bit of the operand is always lost. The leftmost bit of the operand gets the value of 0.
- The effect of the LSR instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.



2.14 CND Instruction

- **The CND instruction arithmetically compares the contents of registers Ri and Rj and puts the result of the comparison (as a set of 1-bit signs) to the register Rj.**
- Signs occupy four lowest bits of the result (see next slides for details and for the meaning of the signs).
- The instruction format is as follows:



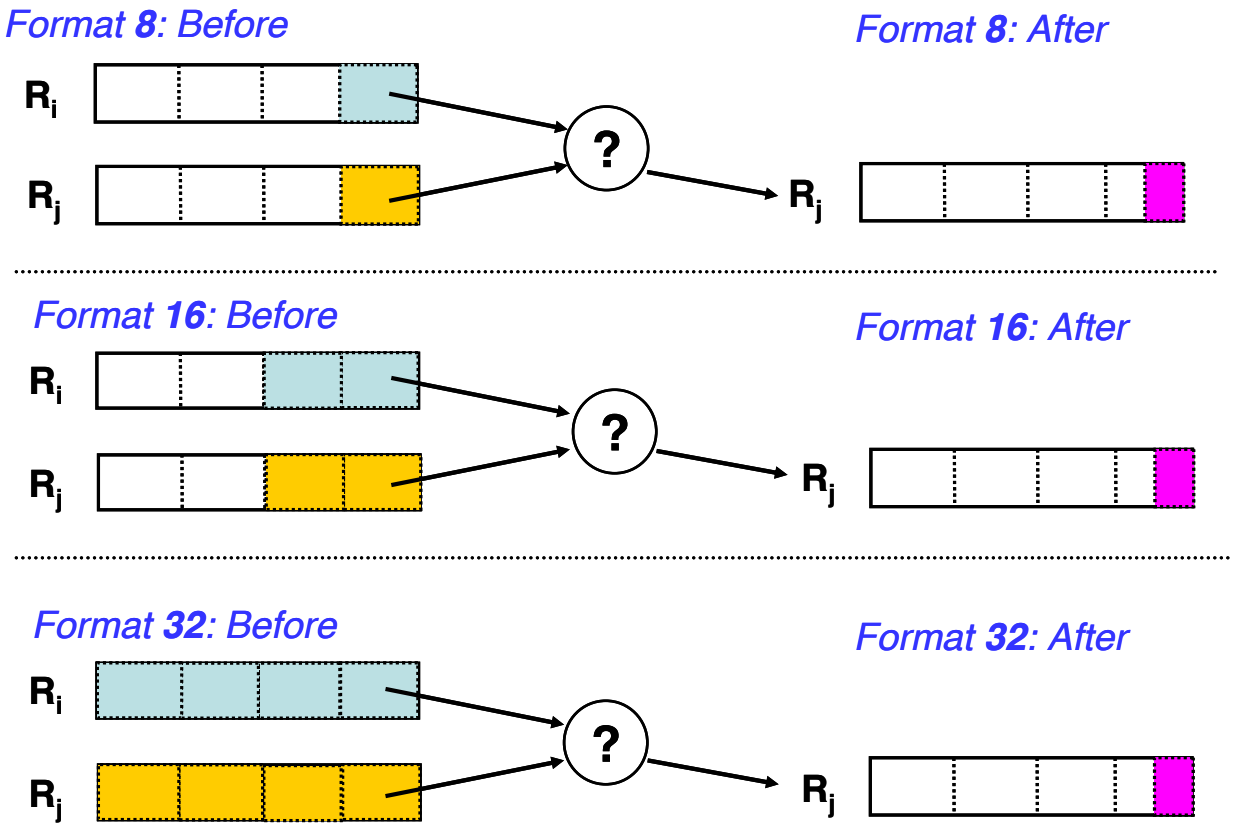
- Memory state is not considered in the instruction, and the memory state does not change.
- Both operands can refer to the same register. If not, the contents of the Ri register does not change.
- Suggested assembly statement for the CND instruction:

```
Rj ?= Ri
```

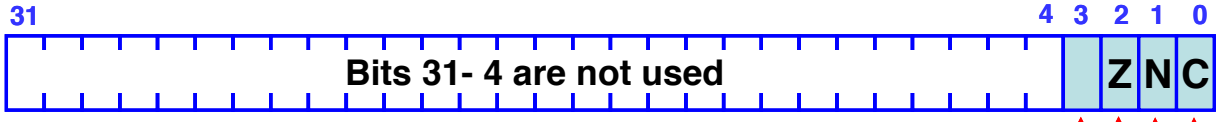
- Additional assembly directives specifying the current instruction format:

```
.format 8    or    .format 16    or    .format 32
```

- The effect of the CND instruction is shown below:



- The result of the instruction (i.e., the contents of the Rj register) is as follows:



- Bit 3: Reserved (always 0)
- Bit 2 (**Z**): **1**, if Ri = Rj, **0**, otherwise
- Bit 1 (**N**): **1**, if Ri < Rj, **0**, otherwise
- Bit 0 (**C**): **1**, if Ri > Rj, **0**, otherwise

- Signs are mutually exclusive: i.e., the semantics of signs assumes that the only one sign is set as the result of the comparison.
- The result of the comparison can be used in an arbitrary way. Perhaps the most important one is to use it for organizing conditional jumps (see CBR instruction).
- Signs can be checked using logical instructions (e.g., AND) together with appropriate masks.

  For example, the ≤ condition can be treated as either < or = conditions, and the corresponding mask is binary 110. Similarly, the ≥ condition is either > or =

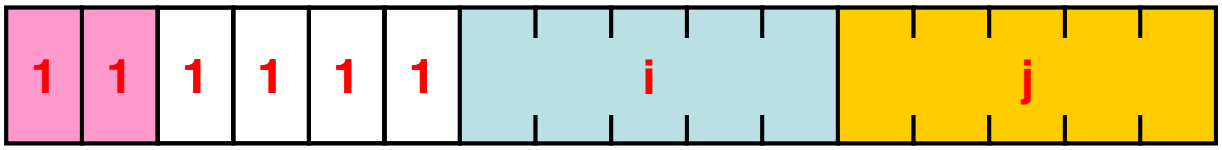
conditions, and the mask is binary 101. Finally, the inequality ≠ condition is either ≤ or ≥ conditions, and the mask for selecting is binary 011.

Therefore, all six possible comparison results (<, ≤, >, ≥, =, ≠) can be extracted using AND instruction with the following masks:

Relation	***Mask***
<	010
≤	110
>	001
≥	101
=	100
≠	011

2.15 CBR Instruction

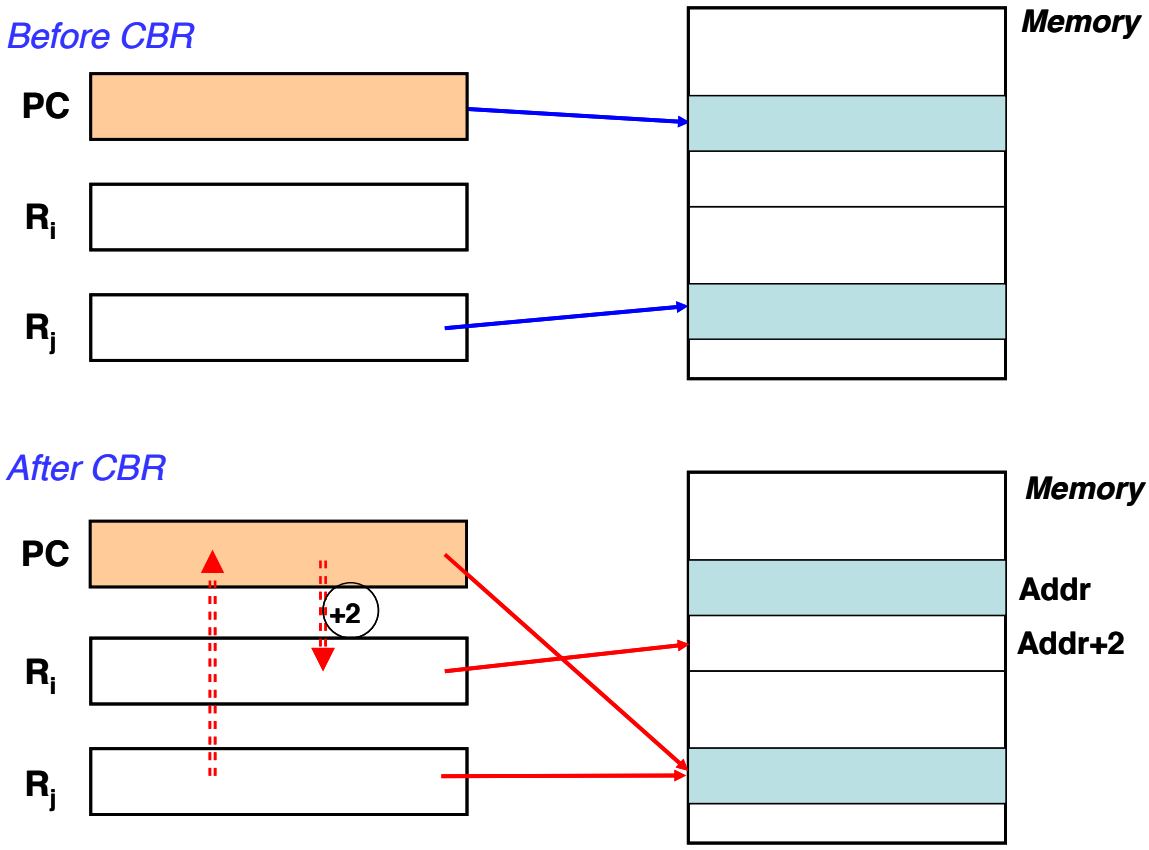
- **The CBR instruction checks the contents of the Ri register. If it is non-zero, then**
  1) the address of the next instruction (i.e., current value of the PC register + 2) is stored in the Ri register, and
  2) the value of the Rj register is set to the PC register. This means that the next instruction will be fetched by the address taken from the Rj register.
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- Format code does not affect the instruction's execution.
- The contents of the Rj register does not change.
- **Suggested assembly statement for the CBR instruction:**

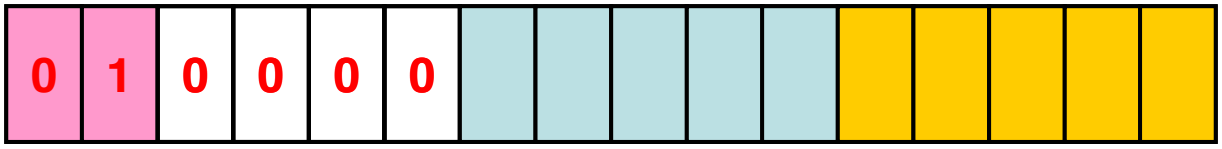
```
if Ri goto Rj
```

- The effect of the CBR instruction (for the case when Ri is non-zero) is shown below:



2.16 NON/STOP Instruction

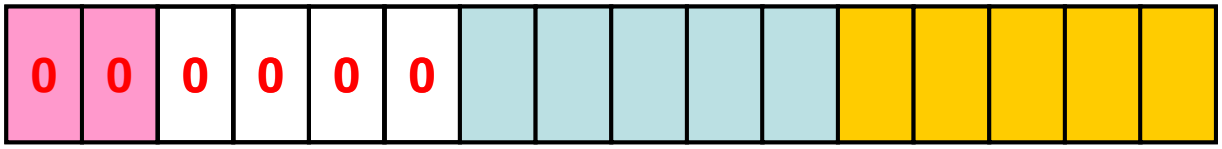
- **The NOP instruction performs no actions, except moving the PC register to the next instruction.**
- The NOP instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The value of the operand part of the instruction (bits from 9 to 0) does not affect the execution.
- The NOP instruction is used as a placeholder (for example to meet alignment requirements), or as a “stub” while code editing or automatic code generation.
- Suggested assembly statement for the NOP instruction:

```
skip
```

- **The STOP instruction causes the program execution to interrupt.**
- The STOP instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The value of the operand part of the instruction (bits from 9 to 0) does not affect the execution.
- Suggested assembly statement for the STOP instruction:
-

```
stop
```

Appendix B

Board elements Testing

Appendix B

Board elements Testing

The board elements tests are illustrated by the scenario based approach. Each time, a test case is used to test each element. The function of the FPGA is tested by loading the bit stream file and by checking the required functions.

The board elements tests are illustrated by the scenario based approach. Each time, a test case is used to test each element. The function of the FPGA is tested by loading the bit stream file and by checking the required functions.

Scenario 1

- Units: U5, U7 static memory modules
- SRAM tests Modules ISSI - IS64WV6416BLL
- Tests the links between U1 (FPGA) components and static memory
- Tests the basic functions of reading and writing from and into U5, U7

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY eurrica IS PORT (

SW3  : IN STD_LOGIC;
SW4  : IN STD_LOGIC;
       Data_0 : INOUT STD_LOGIC;
       Data_1 : INOUT STD_LOGIC;
       Data_2 : INOUT STD_LOGIC;
       Data_3 : INOUT STD_LOGIC;
       Data_4 : INOUT STD_LOGIC;
       Data_5 : INOUT STD_LOGIC;
       Data_6 : INOUT STD_LOGIC;
       Data_7 : INOUT STD_LOGIC;
       Data_8 : INOUT STD_LOGIC;
       Data_9 : INOUT STD_LOGIC;
       Data_10 : INOUT STD_LOGIC;
       Data_11 : INOUT STD_LOGIC;
       Data_12 : INOUT STD_LOGIC;
       Data_13 : INOUT STD_LOGIC;
       Data_14 : INOUT STD_LOGIC;
       Data_15 : INOUT STD_LOGIC;

       SRAM1_L_CE  : OUT STD_LOGIC;
       SRAM1_L_WE  : OUT STD_LOGIC;
       SRAM1_L_OE  : OUT STD_LOGIC;

       SRAM2_L_CE  : OUT STD_LOGIC;
       SRAM2_L_WE  : OUT STD_LOGIC;
       SRAM2_L_OE  : OUT STD_LOGIC;

       A0:   OUT STD_LOGIC;
       A1:   OUT STD_LOGIC;
       A2:   OUT STD_LOGIC;
       A3:   OUT STD_LOGIC;
       A4:   OUT STD_LOGIC;
       A5:   OUT STD_LOGIC;
       A6:   OUT STD_LOGIC;
       A7:   OUT STD_LOGIC;
       A8:   OUT STD_LOGIC;
       A9:   OUT STD_LOGIC;
       A10:  OUT STD_LOGIC;
       A11:  OUT STD_LOGIC;
       A12:  OUT STD_LOGIC;
       A13:  OUT STD_LOGIC;
       A14:  OUT STD_LOGIC;
       A15:  OUT STD_LOGIC;

       D0:  OUT STD_LOGIC;
D1 : OUT STD_LOGIC);
END eurrica;

ARCHITECTURE behavior of eurrica IS
```

```
BEGIN

   D0 <= not SW3;
   D1 <= SW4;

   A0 <= '1';
   A1 <= '0';
   A2 <= '1';
   A3 <= '0';
   A4 <= '1';
   A5 <= '1';
   A6 <= '0';
   A7 <= '0';
   A8 <= '1';
   A9 <= '0';
   A10 <= '0';
   A11 <= '1';
   A12 <= '0';
   A13 <= '1';
   A14 <= '1';
   A15 <= '0';

sm: PROCESS(SW3)

 BEGIN

if (SW3= '1') then

-- write

   SRAM1_L_CE <= '0';
   SRAM1_L_WE <= '0';
   SRAM1_L_OE <= '0';

   SRAM2_L_CE <= '0';
   SRAM2_L_WE <= '0';
   SRAM2_L_OE <= '0';
   Data_0  <= SW4;
   Data_1  <= SW4;
   Data_2  <= SW4;
   Data_3  <= SW4;
   Data_4  <= SW4;
   Data_5  <= SW4;
   Data_6  <= SW4;
   Data_7  <= SW4;
   Data_8  <= SW4;
   Data_9  <= SW4;
   Data_10 <= SW4;
   Data_11 <= SW4;
   Data_12 <= SW4;
   Data_13 <= SW4;
   Data_14 <= SW4;
   Data_15 <= SW4;
else
-- read
   SRAM1_L_CE <= '0';
```

```
   SRAM1_L_WE <= '1';
   SRAM1_L_OE <= '0';

   SRAM2_L_CE <= '0';
   SRAM2_L_WE <= '1';
   SRAM2_L_OE <= '0';

end if;

END PROCESS;

END Behavior;
-----------------------------------------
```

The scenario of the first test is to set the specified links between the FPGA and the static memory modules (U5, U7) and tests writing and reading function. The link configuration is shown on the second column of the table. The SW3 is to control write data and read data into memory. SW4 specifies the data to test. When SW3 is off ('0'), it performs reading operation. When it is on ('1'), it is to perform writing operation. The address of the SRAM for testing is specified by (A0-A15). The LED1 and LED2 indicate the correct operation of SW3, and SW4. The data input and output from the SRAM (U5, and U7) is check by Voltage meter. Voltage "0" represents a '0' logic value, and Voltage "+4" represents a '1' logic value. Initially, all the memory is set to '0'. The testing is to change the settings of SW3, and SW4 and check against the input and output voltages from the data line of U5, and U7. If they are matching, then the test is passed, and the links are correct and the memory modules are performed required functions.

The first column on Tests is checking memory reading with initial value of '0'. The Voltage on the data line shows the correct results '0'. Then the second column is to change the input value to '1' (SW4=1), because the write control is not changed, so the result should still be '0'. The output is correct and the data still keep on '0'. The third column is to write '0' into the memory (SW3 is setting to write and SW4 is setting to 0) and the result shown on data lines are correct, and fourth column is to write '1' (SW3 is setting to write and SW4 is setting to '1') into the memory modules. The output should be "+4V" on data line. The measures by voltage meter show the correct results.

Scenario 1 results

Net		EP2C20Q240C8	Tests			
U5 /U7	DATA_0	155	0	0	0	+4v
	DATA_1	156	0	0	0	+4v
	DATA_2	157	0	0	0	+4v
	DATA_3	159	0	0	0	+4v
	DATA_4	161	0	0	0	+4v
	DATA_5	162	0	0	0	+4v
	DATA_6	164	0	0	0	+4v
	DATA_7	165	0	0	0	+4v
	DATA_8	166	0	0	0	+4v
	DATA_9	167	0	0	0	+4v
	DATA_10	168	0	0	0	+4v
	DATA_11	170	0	0	0	+4v
	DATA_12	171	0	0	0	+4v
	DATA_13	173	0	0	0	+4v
	DATA_14	174	0	0	0	+4v
	DATA_15	175	0	0	0	+4v
U5 /U7	ADDR_0	8	+4v	+4v	+4v	+4v
	ADDR_1	9	0	0	0	0
	ADDR_2	11	+4v	+4v	+4v	+4v
	ADDR_3	13	0	0	0	0
	ADDR_4	14	+4v	+4v	+4v	+4v
	ADDR_5	15	+4v	+4v	+4v	+4v
	ADDR_6	16	0	0	0	0
	ADDR_7	18	0	0	0	0
	ADDR_8	20	+4v	+4v	+4v	+4v
	ADDR_9	21	0	0	0	0
	ADDR_10	37	0	0	0	0
	ADDR_11	38	+4v	+4v	+4v	+4v
	ADDR_12	39	0	0	0	0
	ADDR_13	41	+4v	+4v	+4v	+4v
	ADDR_14	42	+4v	+4v	+4v	+4v
	ADDR_15	44	0	0	0	0
U5	SRAM1_L_CE	233	0	0	0	0
	SRAM1_L_WE	232	+4v	+4v	0	0
	SRAM1_L_OE	231	0	0	0	0
U7	SRAM2_L_CE	230	0	0	0	0
	SRAM2_L_WE	228	+4v	+4v	0	0
	SRAM2_L_OE	226	0	0	0	0
	LED1	125	off	off	on	on
	LED2	178	on	off	on	off
	Switch_3	7	0(off)	0(off)	1(on)	1(on)
	Switch_4	119	0(off)	1(on)	0(off)	1(on)
Results (PASS)			✓	✓	✓	✓

Scenario 2

- U6 and U8 testing
- SRAM tests Modules ISSI - IS64WV6416BLL
- Tests the links between U1 (FPGA) components and static memory
- Tests the basic functions of reading and writing from and into U6, U8

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY eurrica IS PORT (

SW3  : IN STD_LOGIC;
SW4  : IN STD_LOGIC;

        Data_16 : INOUT STD_LOGIC;
        Data_17 : INOUT STD_LOGIC;
        Data_18 : INOUT STD_LOGIC;
        Data_19 : INOUT STD_LOGIC;
        Data_20 : INOUT STD_LOGIC;
        Data_21 : INOUT STD_LOGIC;
        Data_22 : INOUT STD_LOGIC;
        Data_23 : INOUT STD_LOGIC;
        Data_24 : INOUT STD_LOGIC;
        Data_25 : INOUT STD_LOGIC;
        Data_26 : INOUT STD_LOGIC;
        Data_27 : INOUT STD_LOGIC;
        Data_28 : INOUT STD_LOGIC;
        Data_29 : INOUT STD_LOGIC;
        Data_30 : INOUT STD_LOGIC;
        Data_31 : INOUT STD_LOGIC;

        SRAM2_H_CE  : OUT STD_LOGIC;
        SRAM2_H_WE  : OUT STD_LOGIC;
        SRAM2_H_OE  : OUT STD_LOGIC;

        A0:   OUT STD_LOGIC;
        A1:   OUT STD_LOGIC;
        A2:   OUT STD_LOGIC;
        A3:   OUT STD_LOGIC;
        A4:   OUT STD_LOGIC;
        A5:   OUT STD_LOGIC;
        A6:   OUT STD_LOGIC;
        A7:   OUT STD_LOGIC;
        A8:   OUT STD_LOGIC;
        A9:   OUT STD_LOGIC;
        A10:  OUT STD_LOGIC;
        A11:  OUT STD_LOGIC;
        A12:  OUT STD_LOGIC;
        A13:  OUT STD_LOGIC;
        A14:  OUT STD_LOGIC;
        A15:  OUT STD_LOGIC;

        LED1:  OUT STD_LOGIC;
LED2 : OUT STD_LOGIC);
END eurrica;

ARCHITECTURE behavior of eurrica IS

BEGIN
   LED1 <= not SW3;
   LED2 <= SW4;
```

```
   A0 <= '1';
   A1 <= '0';
   A2 <= '1';
   A3 <= '0';
   A4 <= '1';
   A5 <= '1';
   A6 <= '0';
   A7 <= '0';
   A8 <= '1';
   A9 <= '0';
   A10 <= '0';
   A11 <= '1';
   A12 <= '0';
   A13 <= '1';
   A14 <= '1';
   A15 <= '0';

sm: PROCESS(SW3)

 BEGIN

if (SW3 = '1') then

-- write

   SRAM2_H_CE <= '0';
   SRAM2_H_WE <= '0';
   SRAM2_H_OE <= '0';

   Data_16  <= SW4;
   Data_17  <= SW4;
   Data_18  <= SW4;
   Data_19  <= SW4;
   Data_20  <= SW4;
   Data_21  <= SW4;
   Data_22  <= SW4;
   Data_23  <= SW4;
   Data_24  <= SW4;
   Data_25  <= SW4;
   Data_26 <= SW4;
   Data_27 <= SW4;
   Data_28 <= SW4;
   Data_29 <= SW4;
   Data_30 <= SW4;
   Data_31 <= SW4;



else

-- read

   SRAM2_H_CE <= '0';
   SRAM2_H_WE <= '1';
   SRAM2_H_OE <= '0';
```

```
end if;

END PROCESS;

END Behavior;

-------------------------------------
```

The scenario of the second test is to set the specified links between the FPGA and the static memory modules (U6, U8) and tests writing and reading function. The link configuration is shown on the second column of the table. The SW3 is to control write data and read data into memory. SW4 specifies the data to test. When SW3 is off ('0'), it performs reading operation. When it is on ('1'), it is to perform writing operation. The address of the SRAM for testing is specified by (A0-A15). The data input and output is linked to D16-D31. The LED1 and LED2 indicate the correct operation of SW3, and SW4. The data input and output from the SRAM (U6, and U8) is check by Voltage meter. Voltage "0" represents a '0' logic value, and Voltage "+4" represents a '1' logic value. Initially, all the memory is set to '0'. The testing is to change the settings of SW3, and SW4 and check against the input and output voltages from the data line of U6, and U8. If they are matching, then the test is passed, and the links are correct and the memory modules are performed required functions.

The first column on Tests is checking memory reading with initial value of '0'. The Voltage on the data line shows the correct results '0'. Then the second column is to change the input value to '1' (SW4=1), because the write control is not changed, so the result should still be '0'. The output is correct and the data still keep on '0'. The third column is to write '0' into the memory (SW3 is setting to write and SW4 is setting to 0) and the result shown on data lines are correct, and fourth column is to write '1' (SW3 is setting to write and SW4 is setting to '1') into the memory modules. The output should be "+4V" on data line. The measures by voltage meter show the correct results.

Scenario 2 results U6

Net	EP2C20Q240C8	Tests

U6	DATA_16	177	0	0	0	+4v
	DATA_17	184	0	0	0	+4v
	DATA_18	185	0	0	0	+4v
	DATA_19	186	0	0	0	+4v
	DATA_20	187	0	0	0	+4v
	DATA_21	188	0	0	0	+4v
	DATA_22	189	0	0	0	+4v
	DATA_23	191	0	0	0	+4v
	DATA_24	192	0	0	0	+4v
	DATA_25	194	0	0	0	+4v
	DATA_26	195	0	0	0	+4v
	DATA_27	197	0	0	0	+4v
	DATA_28	199	0	0	0	+4v
	DATA_29	200	0	0	0	+4v
	DATA_30	203	0	0	0	+4v
	DATA_31	208	0	0	0	+4v
U6	ADDR_0	8	+4v	+4v	+4v	+4v
	ADDR_1	9	0	0	0	0
	ADDR_2	11	+4v	+4v	+4v	+4v
	ADDR_3	13	0	0	0	0
	ADDR_4	14	+4v	+4v	+4v	+4v
	ADDR_5	15	+4v	+4v	+4v	+4v
	ADDR_6	16	0	0	0	0
	ADDR_7	18	0	0	0	0
	ADDR_8	20	+4v	+4v	+4v	+4v
	ADDR_9	21	0	0	0	0
	ADDR_10	37	0	0	0	0
	ADDR_11	38	+4v	+4v	+4v	+4v
	ADDR_12	39	0	0	0	0
	ADDR_13	41	+4v	+4v	+4v	+4v
	ADDR_14	42	+4v	+4v	+4v	+4v
	ADDR_15	44	0	0	0	0
U6	SRAM1_H_CE	230	0	0	0	0
	SRAM1_H_WE	228	+4v	+4v	0	0
	SRAM1_H_OE	226	0	0	0	0
	LED1	125	off	off	on	on
	LED2	178	on	off	on	off
	Switch_3	7	0(off)	0(off)	1(on)	1(on)
	Switch_4	119	0(off)	1(on)	0(off)	1(on)
	Results (PASS)		✓	✓	✓	✓

Scenario 2 results U8

Net		EP2C20Q240C8	Tests			
U8	DATA_16	177	0	0	0	+4v
	DATA_17	184	0	0	0	+4v
	DATA_18	185	0	0	0	+4v
	DATA_19	186	0	0	0	+4v
	DATA_20	187	0	0	0	+4v
	DATA_21	188	0	0	0	+4v
	DATA_22	189	0	0	0	+4v
	DATA_23	191	0	0	0	+4v
	DATA_24	192	0	0	0	+4v
	DATA_25	194	0	0	0	+4v
	DATA_26	195	0	0	0	+4v
	DATA_27	197	0	0	0	+4v
	DATA_28	199	0	0	0	+4v
	DATA_29	200	0	0	0	+4v
	DATA_30	203	0	0	0	+4v
	DATA_31	208	0	0	0	+4v
U8	ADDR_0	8	+4v	+4v	+4v	+4v
	ADDR_1	9	0	0	0	0
	ADDR_2	11	+4v	+4v	+4v	+4v
	ADDR_3	13	0	0	0	0
	ADDR_4	14	+4v	+4v	+4v	+4v
	ADDR_5	15	+4v	+4v	+4v	+4v
	ADDR_6	16	0	0	0	0
	ADDR_7	18	0	0	0	0
	ADDR_8	20	+4v	+4v	+4v	+4v
	ADDR_9	21	0	0	0	0
	ADDR_10	37	0	0	0	0
	ADDR_11	38	+4v	+4v	+4v	+4v
	ADDR_12	39	0	0	0	0
	ADDR_13	41	+4v	+4v	+4v	+4v
	ADDR_14		+4v	+4v	+4v	+4v
	ADDR_15		0	0	0	0
					42	
U8	SRAM2_H_CE	216	0	0	44	0
	SRAM2_H_WE	232	+4v	+4v	0	0
	SRAM2_H_OE	117	0	0	0	0
	LED1	125	off	off	on	on
	LED2	178	on	off	on	off
	Switch_3	7	0(off)	0(off)	1(on)	1(on)
	Switch_4	119	0(off)	1(on)	0(off)	1(on)
	Results (PASS)		✓	✓	✓	✓

Comments: The SRAM2_H_WE is configured to connect U1 pin 232.

Scenario 3

- U9, and U10- the Read Only Memory (ROM) modules testing
- Tests ROM Modules SHARP - LHF12F17
- Tests the links between U1 (FPGA) components and ROM
- Tests the basic functions of reading and writing of U9, U10

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY eurrica IS PORT (
SW3  : IN STD_LOGIC;
SW4  : IN STD_LOGIC;
-- Data Bus
	Data_0 : INOUT STD_LOGIC;
	Data_1 : INOUT STD_LOGIC;
	Data_2 : INOUT STD_LOGIC;
	Data_3 : INOUT STD_LOGIC;
	Data_4 : INOUT STD_LOGIC;
	Data_5 : INOUT STD_LOGIC;
	Data_6 : INOUT STD_LOGIC;
	Data_7 : INOUT STD_LOGIC;
	Data_8 : INOUT STD_LOGIC;
	Data_9 : INOUT STD_LOGIC;
	Data_10 : INOUT STD_LOGIC;
	Data_11 : INOUT STD_LOGIC;
	Data_12 : INOUT STD_LOGIC;
	Data_13 : INOUT STD_LOGIC;
	Data_14 : INOUT STD_LOGIC;
	Data_15 : INOUT STD_LOGIC;
	Data_16 : INOUT STD_LOGIC;
	Data_17 : INOUT STD_LOGIC;
	Data_18 : INOUT STD_LOGIC;
	Data_19 : INOUT STD_LOGIC;
	Data_20 : INOUT STD_LOGIC;
	Data_21 : INOUT STD_LOGIC;
	Data_22 : INOUT STD_LOGIC;
	Data_23 : INOUT STD_LOGIC;
	Data_24 : INOUT STD_LOGIC;
	Data_25 : INOUT STD_LOGIC;
	Data_26 : INOUT STD_LOGIC;
	Data_27 : INOUT STD_LOGIC;
	Data_28 : INOUT STD_LOGIC;
	Data_29 : INOUT STD_LOGIC;
	Data_30 : INOUT STD_LOGIC;
	Data_31 : INOUT STD_LOGIC;
-- U9 control signals
	ROM1_L_CE  : OUT STD_LOGIC;
	ROM1_L_OE  : OUT STD_LOGIC;
	ROM1_L_WE  : OUT STD_LOGIC;
	ROM1_L_WP  : OUT STD_LOGIC;
-- U10 control signals
	ROM1_H_CE: OUT STD_LOGIC;
	ROM1_H_OE: OUT STD_LOGIC;
	ROM1_H_WE: OUT STD_LOGIC;
	ROM1_H_WP: OUT STD_LOGIC;

-- Address Bus
	A0:   OUT STD_LOGIC;
	A1:   OUT STD_LOGIC;
	A2:   OUT STD_LOGIC;
	A3:   OUT STD_LOGIC;
	A4:   OUT STD_LOGIC;
	A5:   OUT STD_LOGIC;
```

```
        A6:   OUT STD_LOGIC;
        A7:   OUT STD_LOGIC;
        A8:   OUT STD_LOGIC;
        A9:   OUT STD_LOGIC;
        A10:  OUT STD_LOGIC;
        A11:  OUT STD_LOGIC;
        A12:  OUT STD_LOGIC;
        A13:  OUT STD_LOGIC;
        A14:  OUT STD_LOGIC;
        A15:  OUT STD_LOGIC;
        A16:  OUT STD_LOGIC;
        A17:  OUT STD_LOGIC;
        A18:  OUT STD_LOGIC;
        A19:  OUT STD_LOGIC;
        A20:  OUT STD_LOGIC;
        A21:  OUT STD_LOGIC;
        A22:  OUT STD_LOGIC;
        LED1:  OUT STD_LOGIC;
LED2 : OUT STD_LOGIC);
END eurrica;
ARCHITECTURE behavior of eurrica IS
BEGIN
   LED1 <= not SW3;
   LED2 <= SW4;
   A0 <= '1';
   A1 <= '0';
   A2 <= '1';
   A3 <= '0';
   A4 <= '1';
   A5 <= '1';
   A6 <= '0';
   A7 <= '0';
   A8 <= '1';
   A9 <= '0';
   A10 <= '0';
   A11 <= '1';
   A12 <= '0';
   A13 <= '1';
   A14 <= '1';
   A15 <= '0';
   A16 <= '1';
   A17 <= '1';
   A18 <= '1';
   A19 <= '1';
   A20 <= '1';
   A21 <= '1';
   A22 <= '1';
sm: PROCESS(SW3)
 BEGIN
if (SW3 = '1') then
-- write

-- U9 control signals
   ROM1_L_CE <= '0';
   ROM1_L_OE <= '1';
   ROM1_L_WE <= '0';
```

```
   ROM1_L_WP <= '1';

-- U10 control signals
   ROM1_H_CE <='0';
   ROM1_H_OE <='1';
   ROM1_H_WE <='0';
   ROM1_H_WP <='1';

   Data_0  <= SW4;
   Data_1  <= SW4;
   Data_2  <= SW4;
   Data_3  <= SW4;
   Data_4  <= SW4;
   Data_5  <= SW4;
   Data_6  <= SW4;
   Data_7  <= SW4;
   Data_8  <= SW4;
   Data_9  <= SW4;
   Data_10 <= SW4;
   Data_11 <= SW4;
   Data_12 <= SW4;
   Data_13 <= SW4;
   Data_14 <= SW4;
   Data_15 <= SW4;
   Data_16 <= SW4;
   Data_17 <= SW4;
   Data_18 <= SW4;
   Data_19 <= SW4;
   Data_20 <= SW4;
   Data_21 <= SW4;
   Data_22 <= SW4;
   Data_23 <= SW4;
   Data_24 <= SW4;
   Data_25 <= SW4;
   Data_26 <= SW4;
   Data_27 <= SW4;
   Data_28 <= SW4;
   Data_29 <= SW4;
   Data_30 <= SW4;
   Data_31 <= SW4;
else
-- read
-- U9 control signals
   ROM1_L_CE <= '0';
   ROM1_L_OE <= '0';
   ROM1_L_WE <= '1';
   ROM1_L_WP <= '0';
-- U10 control signals

   ROM1_H_CE <='0';
   ROM1_H_OE <='0';
   ROM1_H_WE <='1';
   ROM1_H_WP <='0';

end if;
```

```
END PROCESS;

END Behavior;
-------------------------------------------------------------
```

The scenario of testing ROM is similar with testing RAM, except that the WP is set to '0' and allows read and write. The specified links between the FPGA and the ROM modules (U9, U10) is shown on the second column of the following table. The SW3 is to control write data and read data into memory. SW4 specifies the data to test. When SW3 is off ('0'), it performs reading operation. When it is on ('1'), it is to perform writing operation. The address of the ROM for testing is specified by (A0-A22). The LED1 and LED2 indicate the correct operation of SW3, and SW4. The data input and output (D0-D31) is check by Voltage meter. If Voltage "0" means '0', and Voltage "+4" means '1'. Initially, all the memory is set to '0'. The testing is to change the settings of SW3, and SW4 and check against the input and output voltages from D0-D31. If they are matching, then the test is passed, and the links are correct and the memory modules are performed required functions.

The first column on Tests is checking memory reading with initial value of '0'. The Voltage on the data line shows the correct results '0'. Then the second column is to change the input value to '1' (SW4=1), because the write control is not changed, so the result should still be '0'. The output is correct and the data still keep on '0'. The third column is to write '0' into the memory (SW3 is setting to write and SW4 is setting to 0) and the result shown on data lines are correct, and fourth column is to write '1' (SW3 is setting to write and SW4 is setting to '1') into the memory modules. The output should be "+4V" on data line. The measures by voltage meter show the correct results.

Scenario 3 results

Net		EP2C20Q240C8	Tests			
U9 /U10	DATA_0	PIN_155	0	0	0	+4v
	DATA_1	PIN_156	0	0	0	+4v
	DATA_2	PIN_157	0	0	0	+4v
	DATA_3	PIN_159	0	0	0	+4v
	DATA_4	PIN_161	0	0	0	+4v
	DATA_5	PIN_162	0	0	0	+4v
	DATA_6	PIN_164	0	0	0	+4v
	DATA_7	PIN_165	0	0	0	+4v
	DATA_8	PIN_166	0	0	0	+4v
	DATA_9	PIN_167	0	0	0	+4v
	DATA_10	PIN_168	0	0	0	+4v
	DATA_11	PIN_170	0	0	0	+4v
	DATA_12	PIN_171	0	0	0	+4v
	DATA_13	PIN_173	0	0	0	+4v
	DATA_14	PIN_174	0	0	0	+4v
	DATA_15	PIN_175	0	0	0	+4v
	DATA_16	PIN_177	0	0	0	+4v
	DATA_17	PIN_184	0	0	0	+4v
	DATA_18	PIN_185	0	0	0	+4v
	DATA_19	PIN_186	0	0	0	+4v
	DATA_20	PIN_187	0	0	0	+4v
	DATA_21	PIN_188	0	0	0	+4v
	DATA_22	PIN_189	0	0	0	+4v
	DATA_23	PIN_191	0	0	0	+4v
	DATA_24	PIN_192	0	0	0	+4v
	DATA_25	PIN_194	0	0	0	+4v
	DATA_26	PIN_195	0	0	0	+4v
	DATA_27	PIN_197	0	0	0	+4v
	DATA_28	PIN_199	0	0	0	+4v
	DATA_29	PIN_200	0	0	0	+4v
	DATA_30	PIN_203	0	0	0	+4v
	DATA_31	PIN_208	0	0	0	+4v
U9 /U10	ADDR_0	PIN_8	+4v	+4v	+4v	+4v
	ADDR_1	PIN_9	0	0	0	0
	ADDR_2	PIN_11	+4v	+4v	+4v	+4v
	ADDR_3	PIN_13	0	0	0	0
	ADDR_4	PIN_14	+4v	+4v	+4v	+4v
	ADDR_5	PIN_15	+4v	+4v	+4v	+4v
	ADDR_6	PIN_16	0	0	0	0
	ADDR_7	PIN_18	0	0	0	0
	ADDR_8	PIN_20	+4v	+4v	+4v	+4v
	ADDR_9	PIN_21	0	0	0	0
	ADDR_10	PIN_37	0	0	0	0
	ADDR_11	PIN_38	+4v	+4v	+4v	+4v
	ADDR_12	PIN_39	0	0	0	0
	ADDR_13	PIN_41	+4v	+4v	+4v	+4v
	ADDR_14	PIN_42	+4v	+4v	+4v	+4v
	ADDR_15	PIN_44	0	0	0	0
	ADDR_16	PIN_46	+4v	+4v	+4v	+4v
	ADDR_17	PIN_47	+4v	+4v	+4v	+4v
	ADDR_18	PIN_49	+4v	+4v	+4v	+4v
	ADDR_19	PIN_50	+4v	+4v	+4v	+4v

	ADDR_20	PIN_51	+4v	+4v	+4v	+4v
	ADDR_21	PIN_52	+4v	+4v	+4v	+4v
	ADDR_22	PIN_54	+4v	+4v	+4v	+4v
U9	ROM1_H_CE	PIN_111	0	0	0	0
	ROM1_H_WE	PIN_114	+4v	+4v	+4v	+4v
	ROM1_H_OE	PIN_113	0	0	0	0
	ROM1_H_WP	PIN_116	0	0	+4v	+4v
U10	ROM1_L_CE	PIN_105	0	0	0	0
	ROM1_L_WE	PIN_109	+4v	+4v	+4v	+4v
	ROM1_L_OE	PIN_106	0	0	0	0
	ROM1_L_WP	PIN_110	0	0	+4v	+4v
	LED1	125	off	off	on	on
	LED2	178	on	off	on	off
	Switch_3	7	0(off)	0(off)	1(on)	1(on)
	Switch_4	119	0(off)	1(on)	0(off)	1(on)
	Results		✓	✓	✓	✓

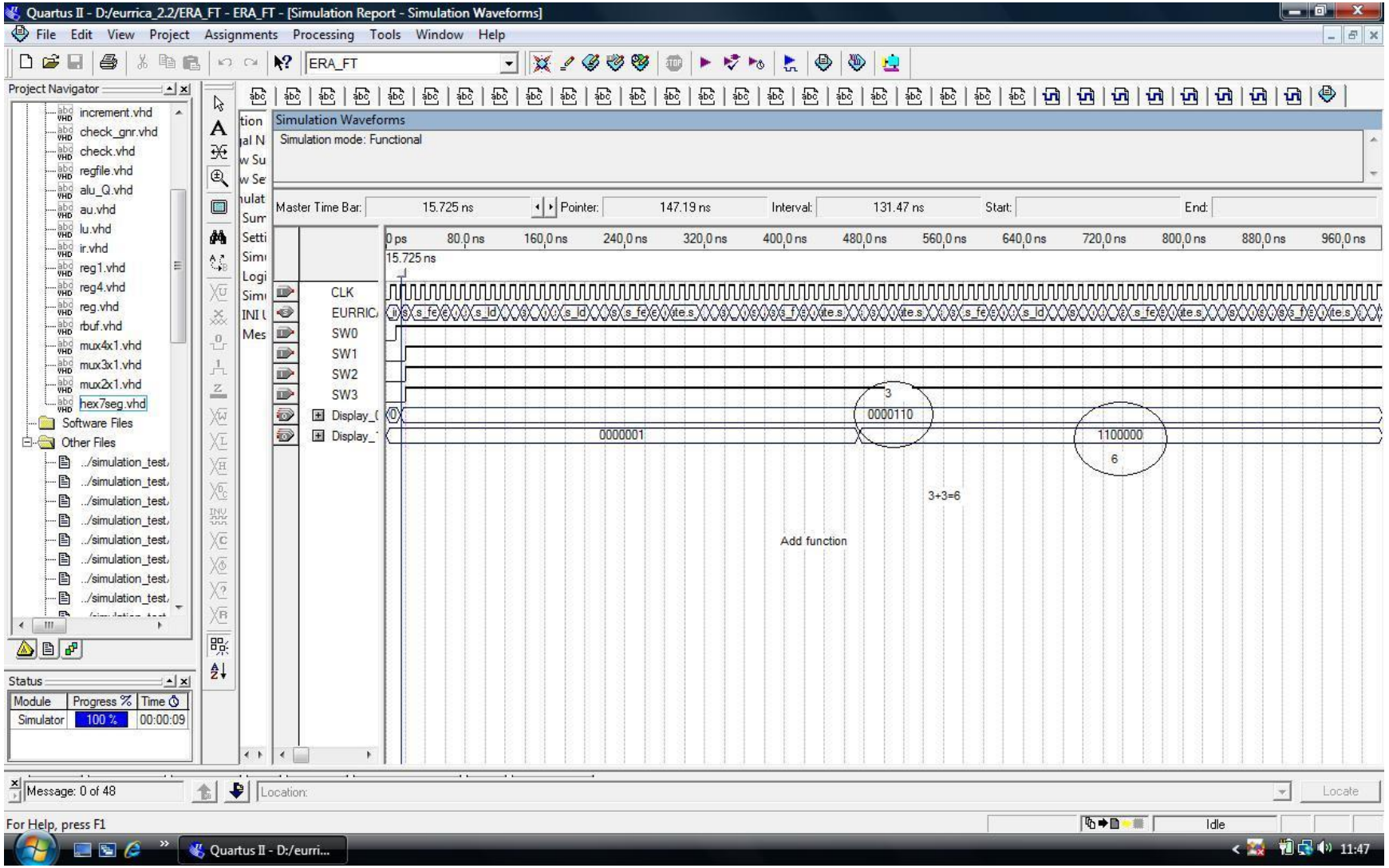
The functional testing of the ERRIC

The sequences of the code to test ERRIC processor are illustrated below. They are loaded with the bits stream file into the FPGA through JTAG interface. Then the codes are executed. The input is the number controlled by push bottom of the Altera board and the calculation results is shown out on 7 segment indicators.

```
-- testing for ADD, Load, CND, CBR
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00010100011000110000010000100011;  --4  ADD R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --      Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --      Mask(>)
        14  :  00000000000000000000100000001101;  --20 --      LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --      2
        16  :  00000000000000000001000001001110;  --22 --      MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --      ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
        [1f..2d]  :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
        [2f..3a]  :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
        [3c..3d]  :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of first sequences of code is a loop to continue loading input data, do the ADD calculation, and output results. To achieve these functions, the instructions - ADD, Load, CND, CBR must function correctly. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the ADD operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs 3+3 function, the ADD function gives the results 6. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows in the circles on the pictures are matched the 7 segments of testing Altera board in all the following tests.

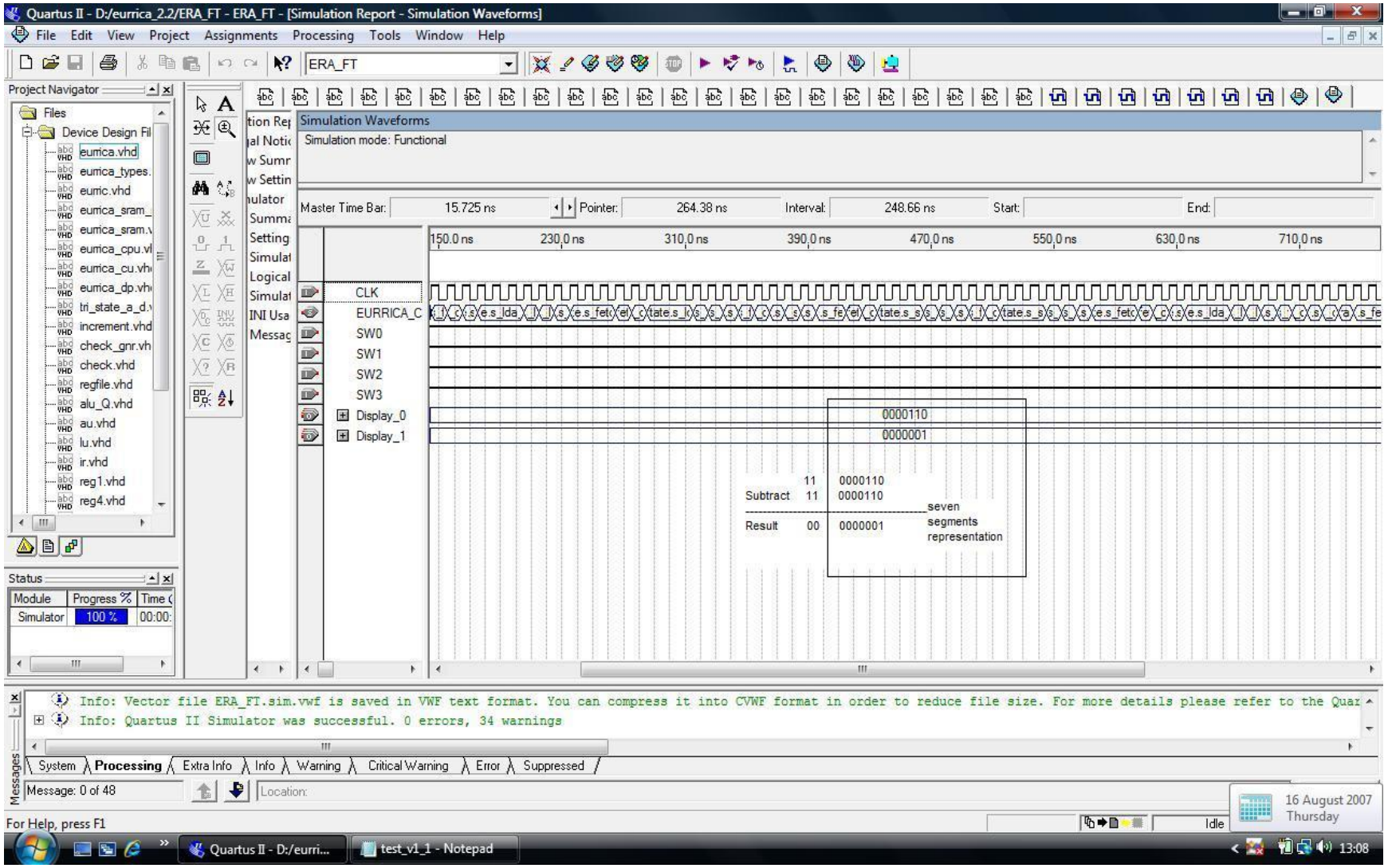


Results of functional testing of ADD, LOAD, CND, CBR instruction

```
-- testing for SUB
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00011000011000110000010000100011;  --4  SUB R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --      Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --      Mask(>)
        14  :  00000000000000000000100000001101;  --20 --      LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --      2
        16  :  00000000000000000001000001001110;  --22 --      MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --      ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
       [1f..2d] :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
       [2f..3a] :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
       [3c..3d] :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is a loop to continue loading input data, do the SUB calculation, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the SUB operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs 3-3 function, the subtract function gives the results 0. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows in the square on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

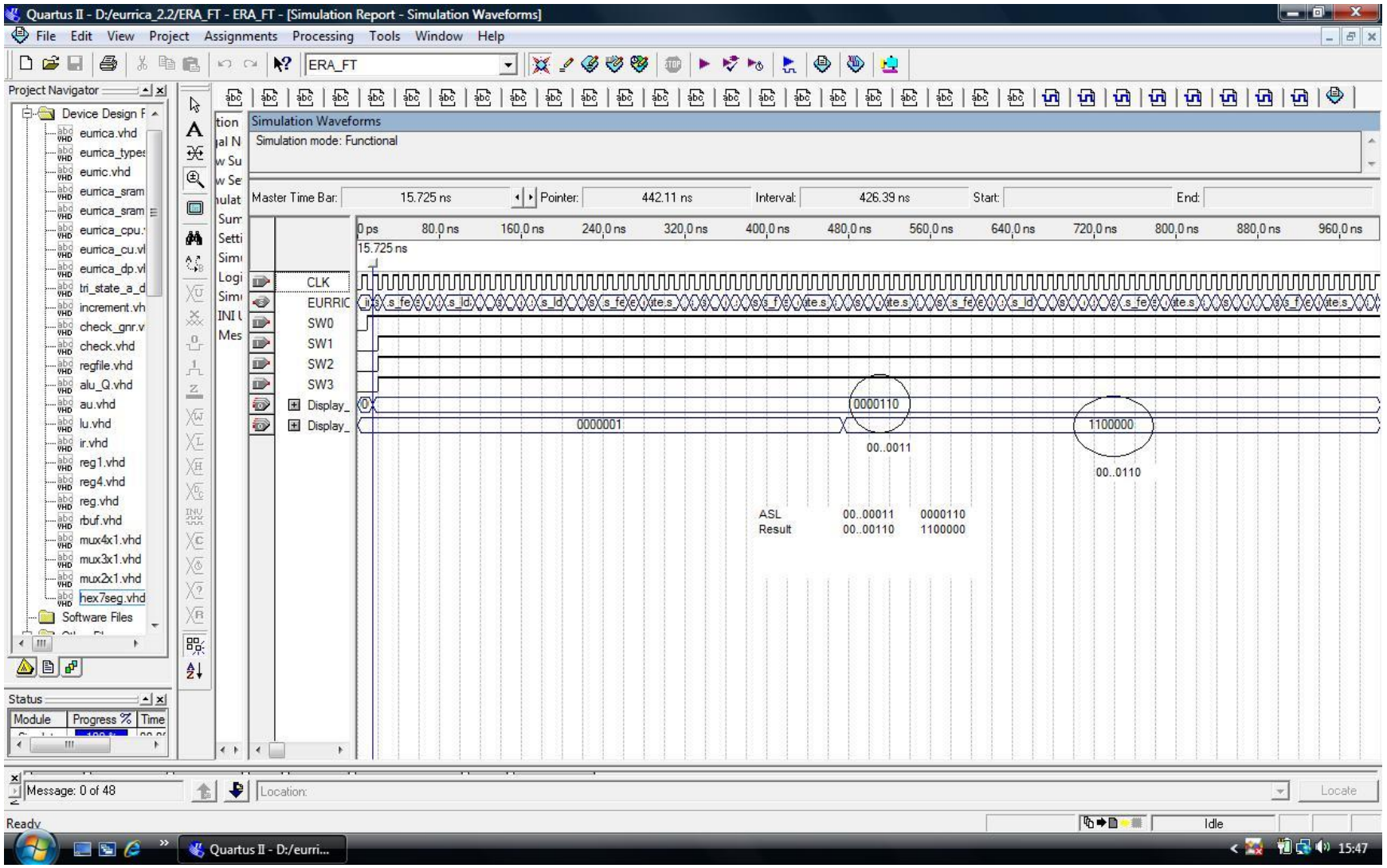


Testing results of SUB

```
-- testing for ASL
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00100000011000110000010000100011;  --4  ASL R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --     Mask(>)
        14  :  00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --     2
        16  :  00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
       [1f..2d] :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
       [2f..3a] :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
       [3c..3d] :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is a loop to continue loading input data, do the ASL calculation, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the ASL operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs arithmetically left shift function on input data '3', the function gives the results '6'. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows in the square on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

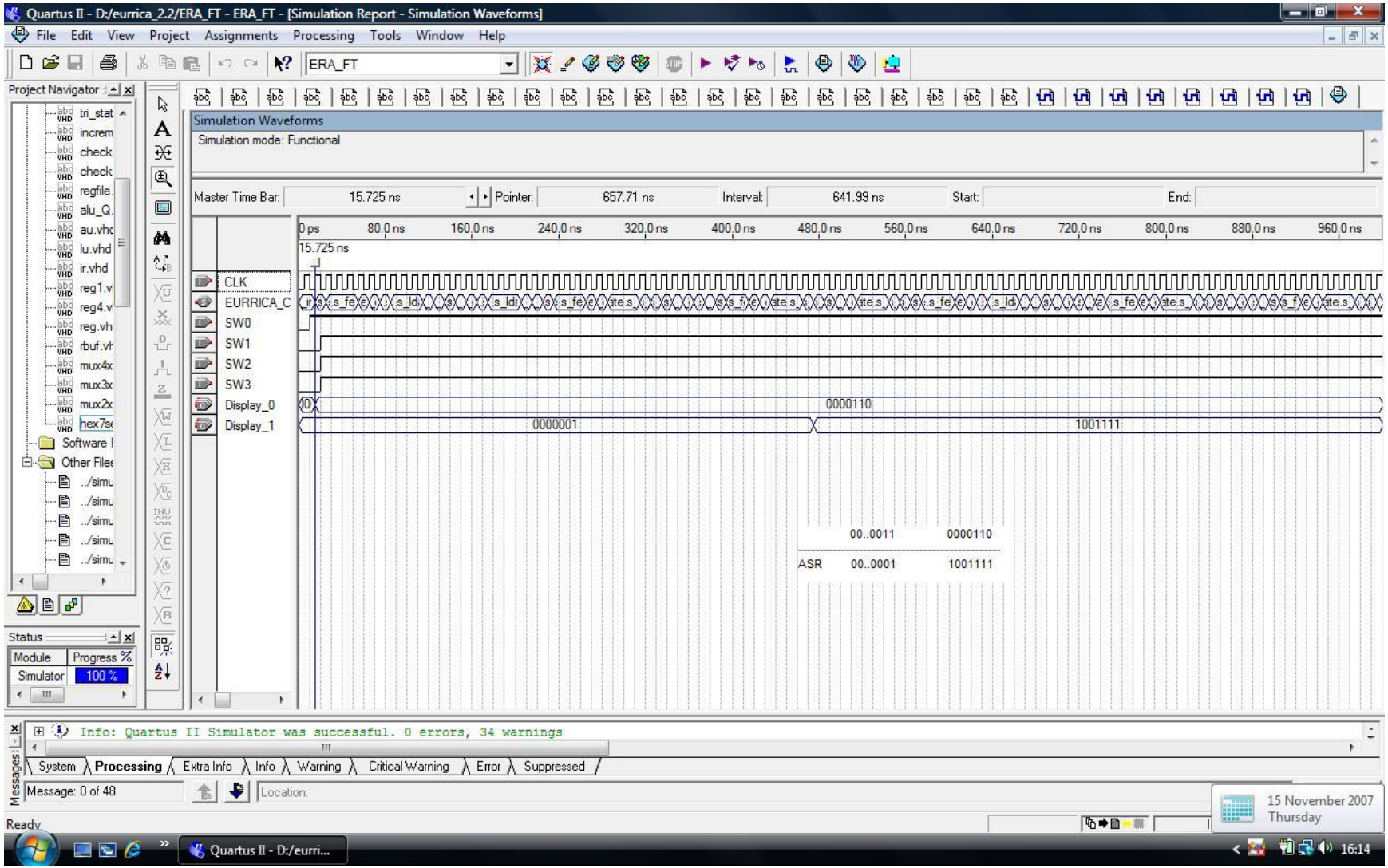


The results of testing ASL instructions

```
-- testing for ASR
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00011100011000110000010000100011;  --4  ASR R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --     Mask(>)
        14  :  00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --     2
        16  :  00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
        [1f..2d]  :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
        [2f..3a]  :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
        [3c..3d]  :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is to test arithmetic shift right instruction. It is a loop to continue loading input data, do the ASR calculation, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the ASR operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs arithmetically right shift function on input data "0..011", the function gives the results "..0..01". The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows in the square on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

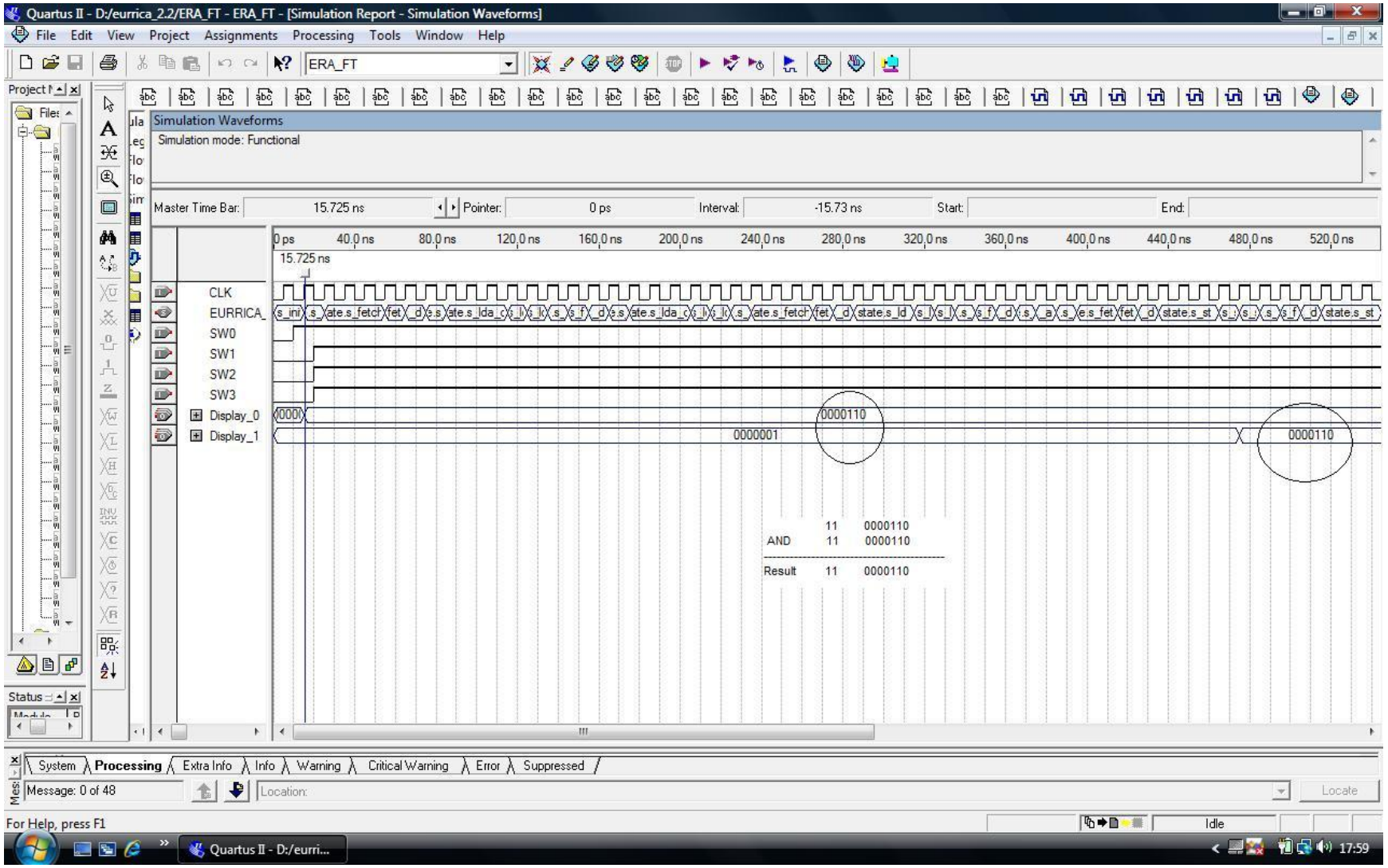


The results of ASR

```
-- testing for AND
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00101000011000110000010000100011;  --4  AND R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --      Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --      Mask(>)
        14  :  00000000000000000000100000001101;  --20 --      LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --      2
        16  :  00000000000000000001000001001110;  --22 --      MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --      ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
       [1f..2d] :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
       [2f..3a] :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
       [3c..3d] :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is to test logic AND instruction. It is a loop to continue loading input data, do the logically AND function, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the AND operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs logical AND function on input data “0..011”, the function gives the results “0..011”. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows in the square on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

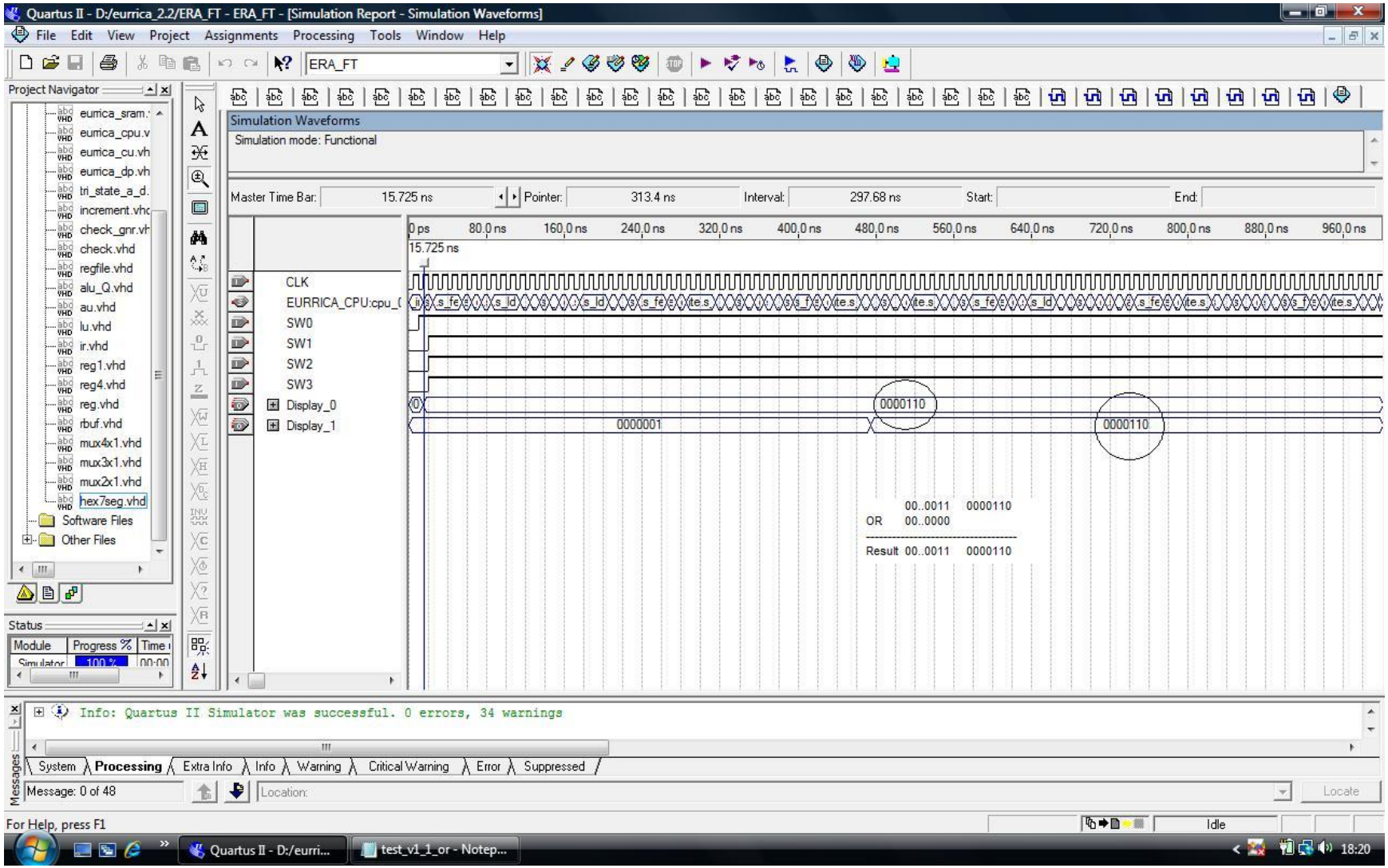


The testing results of AND instructions

```
-- testing for OR
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00100100001000110000010000100011;  --4  OR R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --     Mask(>)
        14  :  00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --     2
        16  :  00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
        [1f..2d]  :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
        [2f..3a]  :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
        [3c..3d]  :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is to test logic OR instruction. It is a loop to continue loading input data, do the logically OR function, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the OR operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs logical OR function on input data “0..011” and “0..0..00”, the function gives the results “0..011”. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

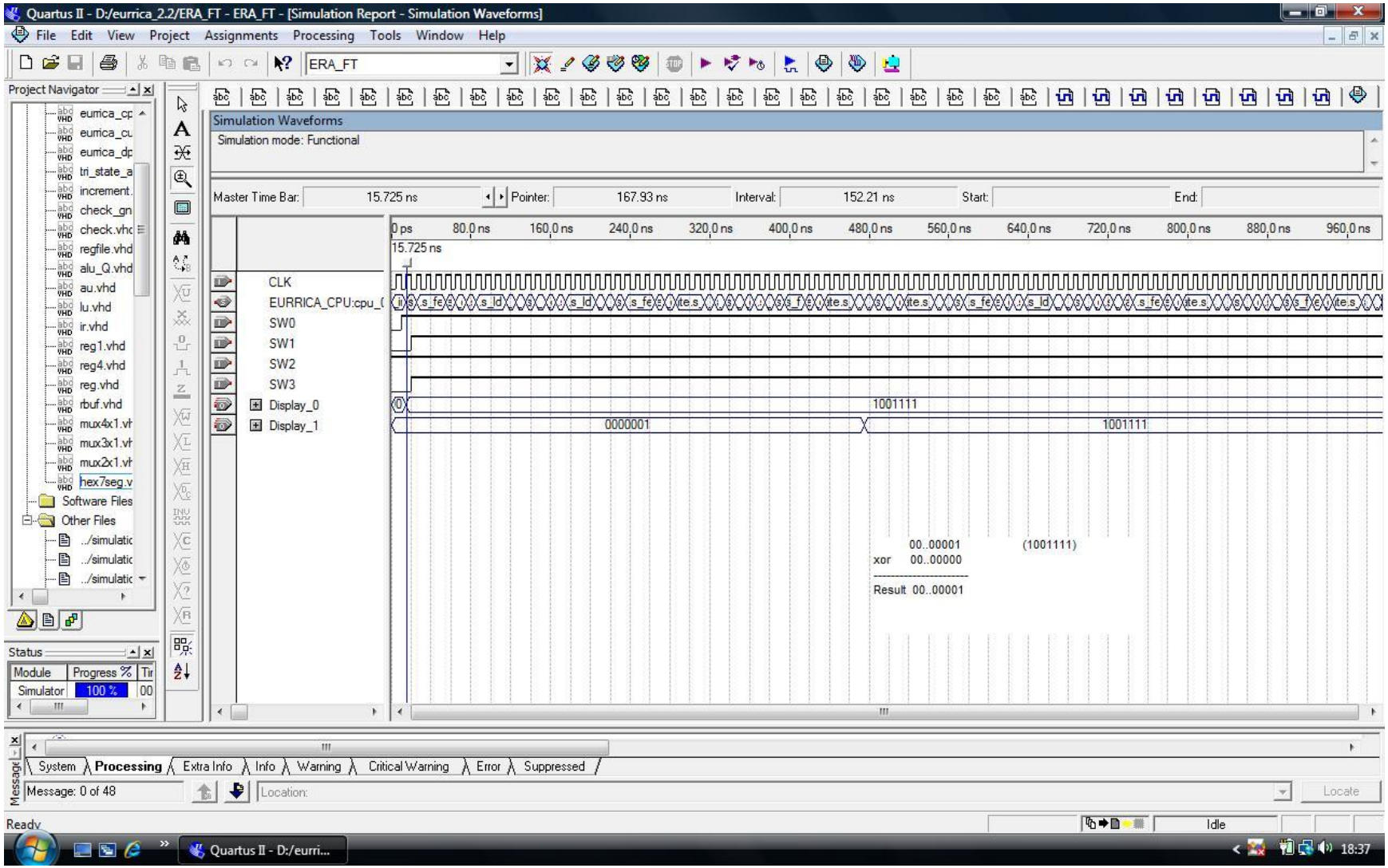


The testing results of the OR instruction

```
-- testing for XOR
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00101100001000110000010000100011;  --4  XOR R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5    --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --     Mask(>)
        14  :  00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --     2
        16  :  00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
        [1f..2d]  :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
        [2f..3a]  :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
        [3c..3d]  :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is to test logic XOR instruction. It is a loop to continue loading input data, do the logically XOR function, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the XOR operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs logical XOR function on input data “0..001” and “0..000”, the function gives the results “0..001”. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

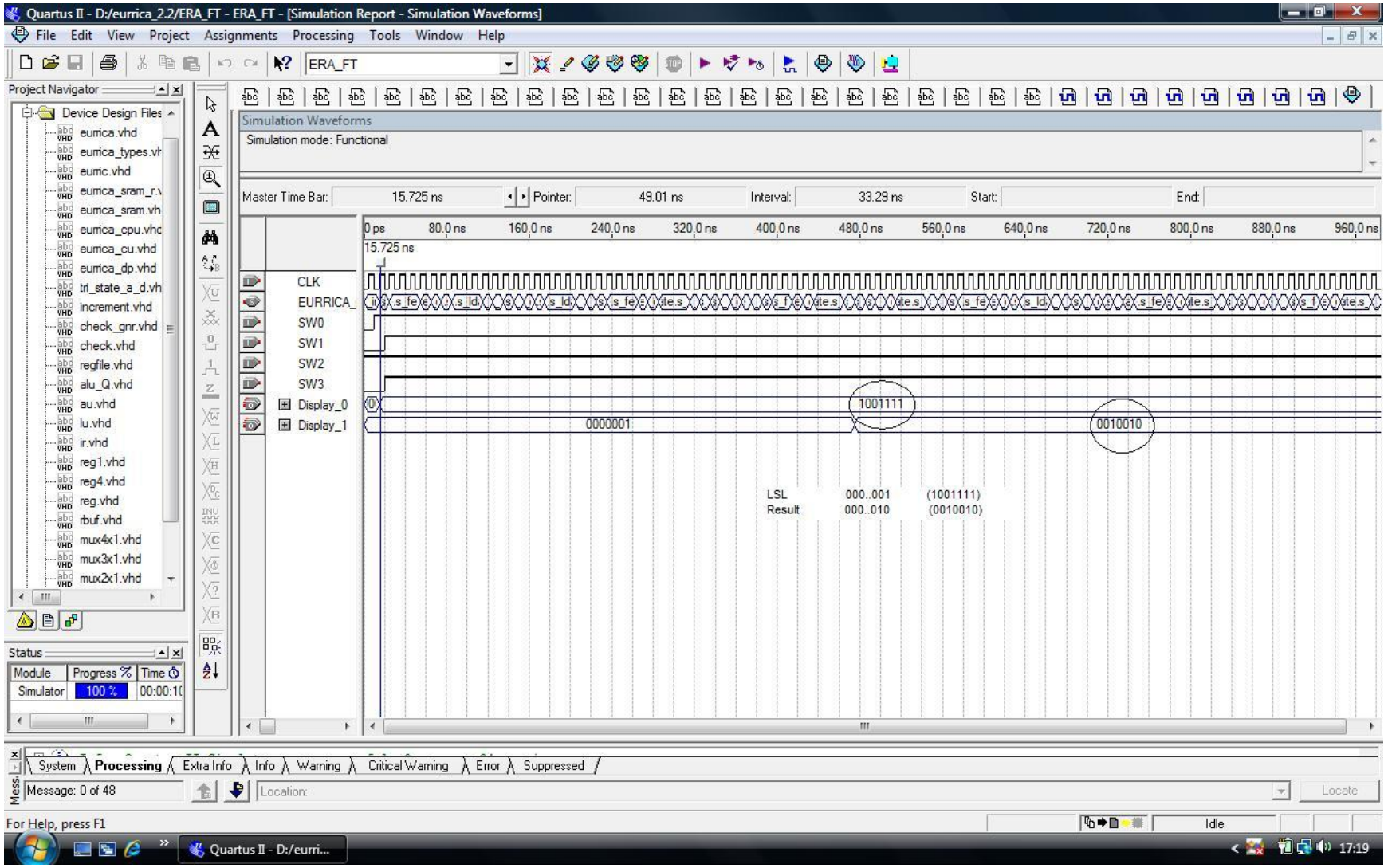


The testing results of the XOR instruction

```
-- testing for LSL
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00110000011000110000010000100011;  --4  LSL R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --     Mask(>)
        14  :  00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --     2
        16  :  00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
        [1f..2d]  :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
        [2f..3a]  :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
        [3c..3d]  :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is to test logic left shift instruction. It is a loop to continue loading input data, do the logically left shift function, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the logic left shift operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs logical left function on input data “0..001”, the function gives the results “0..010”. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows on the following pictures are matched the 7 segments of testing Altera board in all the following tests.

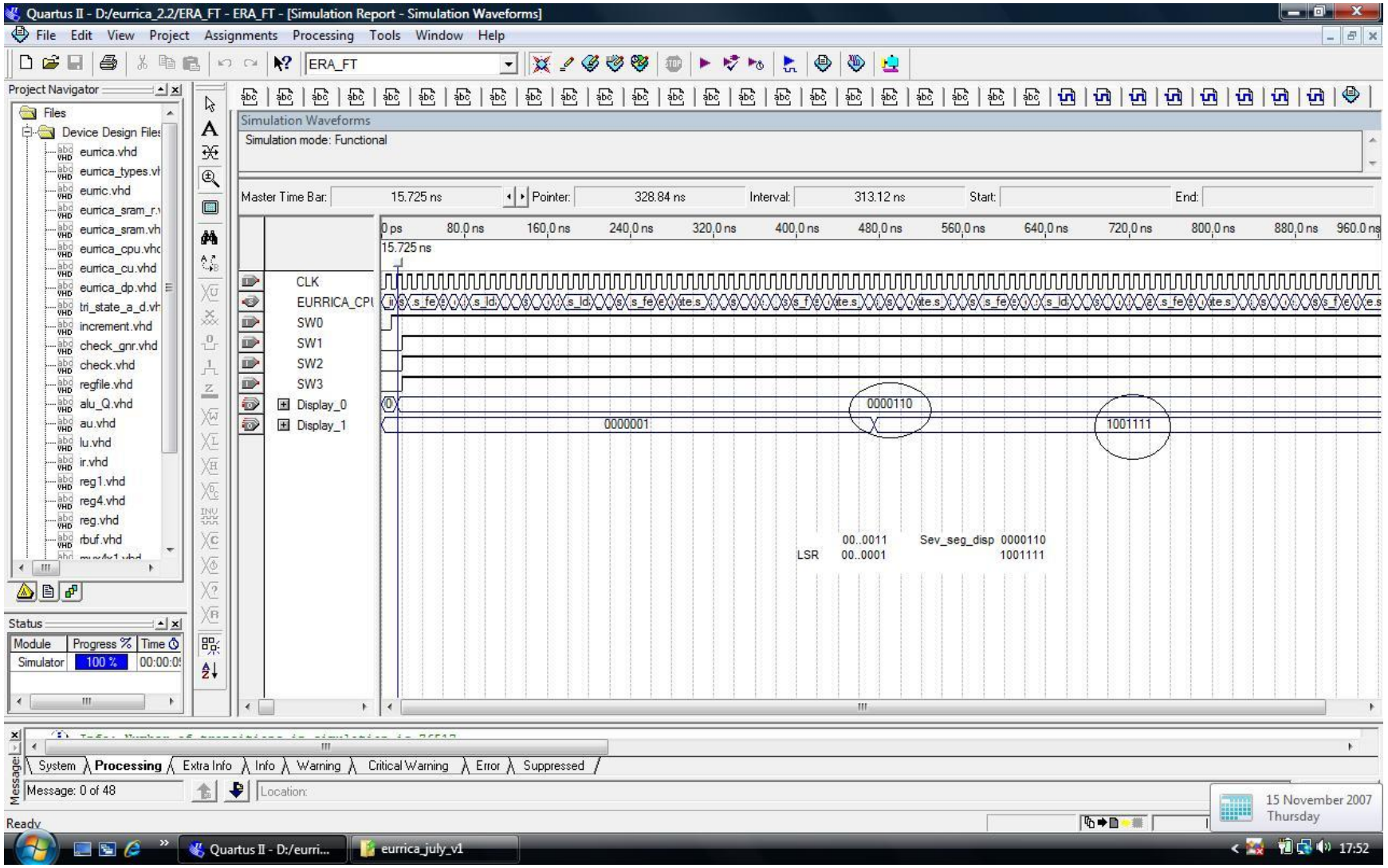


The testing result of LSL

```
-- testing for LSR
WIDTH=32;
DEPTH=64;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        00  :  00000000000000000000000000000000;  --0       ------------
        01  :  00001000000000100000100000000001;  --1  LDA R2   ; LDA R1
        02  :  00100000000000000000000000000000;  --2  (input)  port1(addr)
        03  :  01000000000000000000000000000000;  --3  (output) port2(addr)
        04  :  00110100011000110000010000100011;  --4  LSR R3 R3 ; LD (R1) R3
        05  :  00001100010000110000110001000011;  --5     --     ; ST (R2) R3
        06  :  00111000101000010000100000000101;  --6  CBR R5  R1 ; LDA R5
        07  :  00000000000000000000000000000100;  --7    --4-- Address
        08  :  00001000000001100000100000000101;  --8  --LDA R6 ; LDA R5
        09  :  00000000000000000000000000010110;  --9  --      22(jump1)
        0a  :  00000000000000000000000000010101;  --10 --      21(jump2)
        0b  :  00001000000010000000100000000111;  --11 --LDA R8    ;  LDA R7
        0c  :  00000000000000000000000000011100;  --12  --     28(jump3)
        0d  :  01000000000000000000000000000000;  --13  --(output)   port2(addr)
        0e  :  00001000000010100000100000001001;  --14 --LDA R10   ;  LDA R9
        0f  :  00000000000000000000000000001011;  --15   --    B
        10  :  00000000000000000000000000001010;  --16   --    A
        11  :  00001000000011000000100000001011;  --17 --LDA R12   ;  LDA R11
        12  :  11000000000000000000000000000000;  --18 --     Mask(<=)
        13  :  00100000000000000000000000000000;  --19 --     Mask(>)
        14  :  00000000000000000000100000001101;  --20 --     LD (PC) R13
        15  :  00000000000000000000000000000010;  --21 --     2
        16  :  00000000000000000001000001001110;  --22 --     MV  R2  R14
        17  :  00000000000000000001010000101110;  --23 --     ADD R1  R14
        18  :  00111100011100000001000111010000;  --24 -- CND R3 R16 ; mv  R14 R16
        19  :  00111000101100000010100101110000;  --25 -- CBR R5 R16;  AND R11 R16
        1a  :  00111101101011110000010010001111;  --26 -- CND R13 R15; LD  R4  R15
        1b  :  00111001111001110010100110001111;  --27 -- CBR R7 R15 ; AND R12 R15
        1c  :  00111000110000010000110101001000;  --28 -- CBR R6 R1;   ST  R10 R8
        1d  :  00111000110000010000110100101000;  --29 -- CBR R6 R1;   ST  R9  R8
        1e  :  10001110100000000010000000000000;
        [1f..2d]  :  10101110000000000010000000000000;
        2e  :  11011110000000000010000000000000;
        [2f..3a]  :  01011000100000000010000000000000;
        3b  :  11011110000000000010000000000000;
        [3c..3d]  :  01111000100000000010000000000000;
        3e  :  00110100000000000010000000000000;
        3f  :  11111111111111110010000000000000;
END;
```

The function of this sequence of code is to test logic right shift instruction. It is a loop to continue loading input data, do the logically left shift function, and output results. First, the input port memory address is loaded into R1, output port memory address is loaded into R2. Then load the input data, do the logic left right operation on the loaded data, and then output the results. After the operation, it jumps back to the next loop ready to the next tests.

The following picture is showed both the simulation results and the physical testing results. It performs logical left function on input data “0..011”, the function gives the results “0..001”. The results in simulation and indication on testing board (7 Segment digital) are matched to prove that the tests are passed. The number shows on the following pictures are matched the 7 segments of testing Altera board in all the following tests.



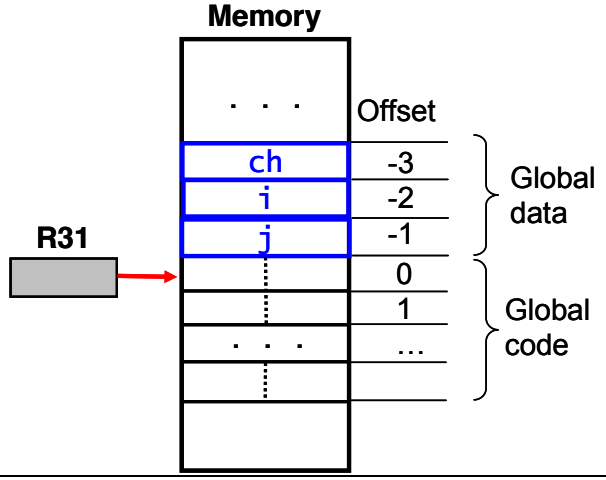
The testing results of LSR

Appendix C

Assembler Code Examples

Example 1.
Global data and code

Source code	Memory structure	Code	Ass.Code	Comments

`char ch;`
`short int i;`
`int j;`
				**Programming convention 1:**
				
				R31 register always keeps the base address of the global data (with negative offsets) and the program code (with non-negative offsets).
				
				Initially R31 is set by the progam loader.

`ch := '0';`		`LDA R1` `NOP` `'0'`	`R1 := '0';`	Get the value of '0' into R1
		`LDA R2` `ADD R31,R2` `-3`	`R2:=-6;` `R2+=R31;`	Get the address of ch into R2 (as R31+offset)
		`ST R1,R2`	`*R2:=R1;`	Store the value from R1 to ch (pointed to by R2)
`i := 10;`		`LDA R1` `NOP` `10`	`R1:=10;`	Get the value of 10 into R1
		`LDA R2` `ADD R31,R2` `-2`	`R2:=-4;` `R2+=R31;`	Get the address of i to into R2 (as R31+offset)
		`ST R1,R2`	`*R2:=R1;`	Store the value from R1 to i (pointed to by R2)
`j := i;`		`LDA R1` `ADD R31,R1` `-1`	`R1:=-2;`	Get the offset of j into R1; get the address of j into R1 (as R31+offset)
		`LDA R2` `ADD R31,R2` `-2`	`R2:=-4;` `R2+=R31;`	Get the offset of i into R1; get the address of I into R2 (as R31+offset)
		`LD R2,R2` `ST R2,R1`	`R2:=*R2;` `*R1:=R2;`	Get the value pointed to by R2 (i.e., i) to R2. Store the value from R2 to j (pointed to by R1)

Example 2.
Jumps Without Context Switch: Gotos Within Routine Code

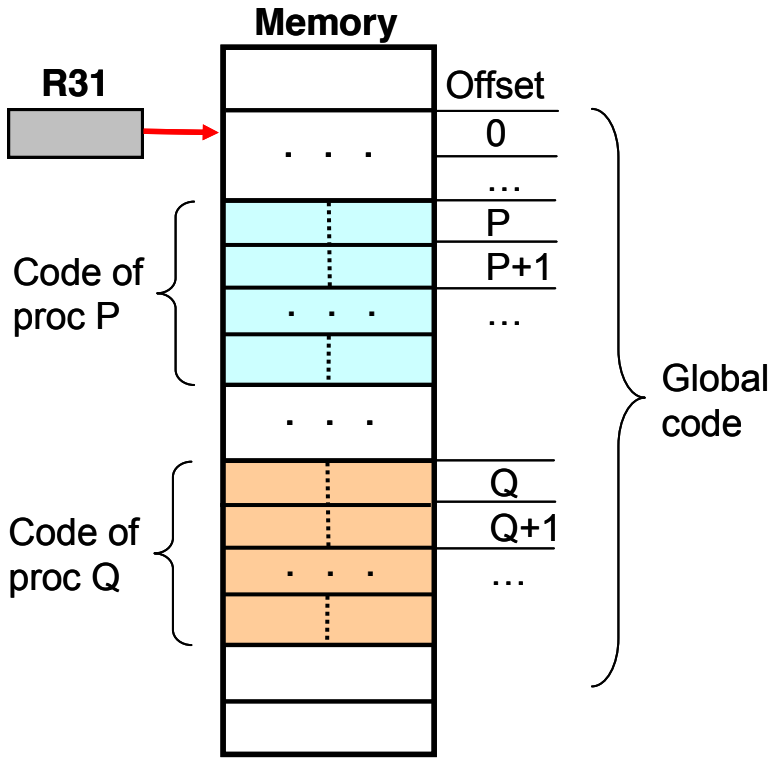
Source code	Code base	Offset	Code	Assembly Code	Comments
					Label is treated as the relative address of the instruction which follows directly after the label
Stmts before goto L; Stmts between L: Stmts after	R31 --->	0	Code before	Code before	
		...	. . .		
		N	LDA R1 | ADD R1,R31	R1 := L; R1 += R31;	Calculate physical address of the target instruction: codebase+label
		N+1	L		
		N+2	CBR R1,R1	if R1 goto R1;	Unconditional branch: we use R1 as the "condition" because it is guaranteed non-null.
		N+3	Code between	Code between	
		...	. . .		
		L	Code after	<L> Code after	
		...	. . .		

Example 3.
Jumps with Context Switch: Procedure Call

Source code

```
  . . .
proc P;
  var x, y, z, t : int;
begin
  . . .
  Q(x,7,&y,z+t);
  . . .
end P;

proc Q(a,b:int;c:int&;d:int);
begin
  . . .
  return;
end Q;
  . . .
```

Code structure



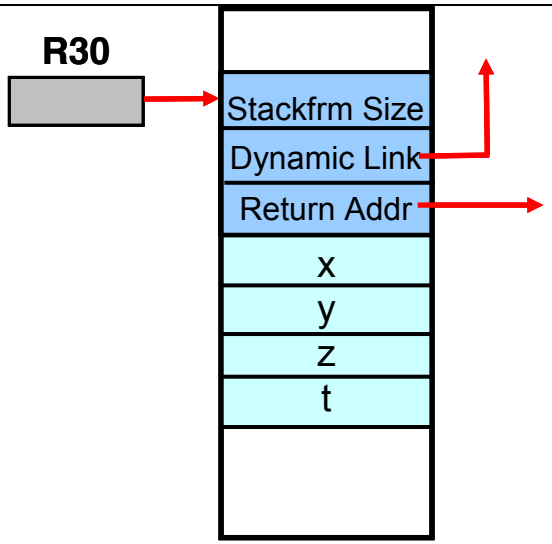
Comment

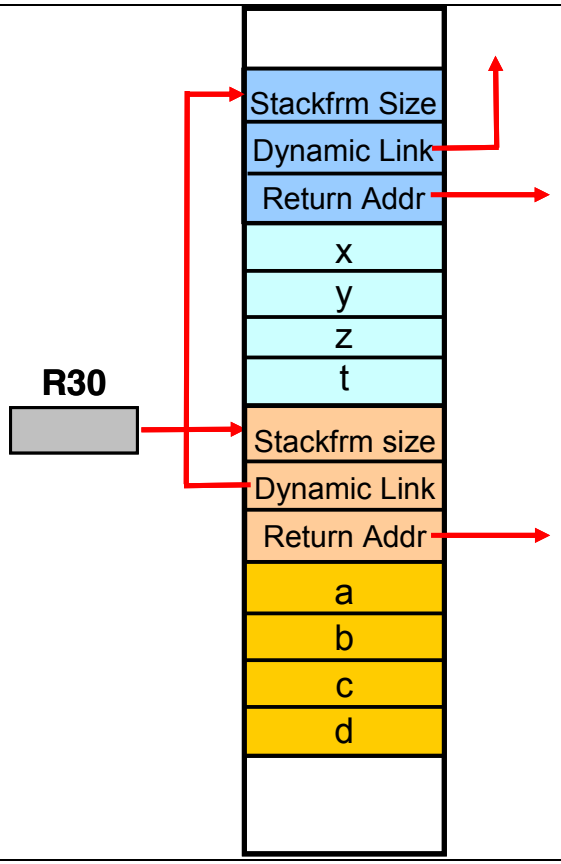
This example illustrates different ways of passing parameters:

1. The value of a standalone variable
2. A constant
3. An address of a variable ("passing by reference")
4. The value of an expression (evaluate and pass)

Procedure value is treated as the relative address (of common code base R31) of the first procedure's instruction.

Source code	Memory structure before call to Q	Memory structure after call to Q	
			Programming convention 2: R30 register always keeps the base address of the current stackframe which stores local data of the latest procedure call (together with some





additional information).

The R30's contents changes when calling a routine and returning from a routine (see example below).

R30 is set by the program loader and initially points to the global data (i.e., R30=R31).

Programming convention 3:

R29 register is used for passing the return value (if any) from the callee to the caller.

Code for calling routine

```
...
Q(x,7,&y,z+t);
...
```

```
// Create the stackframe for Q (permanent part of the call)
R1 := *R30;           // get the size of the stackframe
R1 += R30;            // R1 points to the start of the new stackfrm
R2 := 1; R2 += R1;    // R2 points to the 2nd word of the new stackframe
*R2 := R30;           // Store dynamic link: pointer to the old stackframe

// Evaluate and store actual arguments (this part can vary)
// First actual: x
R2 := 3; R2 += R30;   // Get the address of x (based by R30)
R2 := *R2;            // Get the value of x
R3 := 3; R3 += R1;    // Get the address for the 1st actual
*R3 := R2;            // Store the value of the 1st actual
```

	// Second actual: constant 7
	R2 := 7; // Get the value of the 2nd actual
	R3 := 4; R2 += R1; // Get the address for the 2nd actual
	*R3 := R2; // Store the value of the 2nd actual
	// Third actual: address of y
	R2 := 4; R2 += R30; // Get the address of y
	R3 := 5; R3 += R1; // Get the address for the 3rd actual
	*R3 := R2; // Store the address of y as the 3rd actual
	// Fourth actual: z+t
	R2 := 5; R2 += R30; // Get the address of z
	R2 := *R2; // Get the value of z
	R3 := 6; R3 += R30; // Get the address of t
	R3 := *R3; // Get the value of t
	R2 += R3; // Get the sum z+t
	R3 := 6; R3 += R1; // Get the address for the 4th actual
	*R3 := R2; // Store the sum z+t as the 4th actual
	
	// Jump to the code of proc Q (permanent part of the call)
	R30 := R1; // Make new stackframe the current one
	R2 := Q; R2 += R31; // Get the address of the procedure Q
	if R2 goto R2; // Return address is stored in R2;

	Code for the routine being called (Routine's "standard prologue")

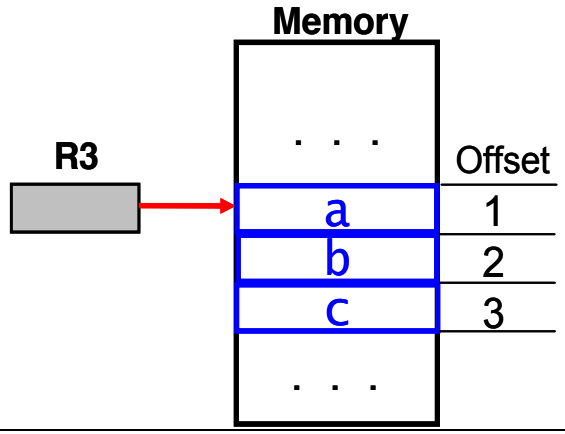
	R1 := <own-stackframe-size>;
	*R30 := R1; // Store the size of the own stackframe
	R1 := 2; R1 += R30; // The address for storing return address
	*R1 := R2; // Store return address
	. . . // The routine code itself

	Code for returning from a routine (Routine's "standard epilogue")

return;	R1 := 2; R1 += R30; // Get the address of the RetAddr in the current stkfrm
	R1 := *R1; // Get the return address itself
	R2 := 1; R2 += R30; // Get the address of the dynamic link
	R2 := *R2; // Get the dynamic link itself
	R30 := R2; // Restore the address of the previous stackframe
	if R1 goto R1; // Return to the caller

Example 4.
Dynamic Data Structures

Source code	Memory structure	Code	Assembly Code	Comments

`struct S`
`{ int a;`
`  int b;`
`  int c; };`
 	
`. . .`
 	
`S* p = new S;`
 	
`. . .`
				

`p->a = 7;`		`LDA R4` `ST R4,R3` 7	`R4 := 7;` `*R3 := R4;`	Get the address of `a` field (`R3`) and store the value of `3` by this address
`p->b = p->c;`		`LDA R4` `ADD R3,R4` 1	`R4:=1;` `R4+=R3;`	Set `1` (as field `b`'s offset) to `R4`, and then get the address of `b` as `R3+1`
		`LDA R5` `ADD R3,R5` 2	`R5:=2;` `R5+=R3;`	Set `2` (as field `c`'s offset) to `R5`, and then get the address of `c` as `R3+2`
		`LD R5,R5` `ST R4,R5`	`R5:=*R5;` `*R4:=R5;`	Load `c` to `R5` and then store it by the address of `b` (from `R4`)

Appendix D

Dissimera's Source Code Documentation

Dissimera's Source Code and documentation

Dissimera

Disassembler/Simulator of the ERA architecture

Victor Castano
Version 0.1a
16 March 2012

Abstract

What follows is a detailed documentation of the simulator's source code for the ERA architecture. can help others to:

1) understand the simulator and its design at a low level, and
2) co-participate in the development/upgrade of the tool.

Alternatively, a dynamic website has been deployed at a temporary location at:

http://victorccc.com/index.html.

By the time this document is read, this website will include newer developments. Hence, it is recommendable to have a look at it to see the latest updates.

Table of Contents

Introduction

Reading binary code is a painful experience for any programmer. In order to test and troubleshoot any error of design, bug or incompatibility between linker and the VHDL code I have decided to develop a Disassembler and a Simulator in a combined tool that will ease this process. In addition it will allow the simulation of the state of the processor at any given time.

The fundamental characteristics of this tool will be:

- Disassembling of instructions: Binary-to-ASM and and Binary-to-PSEUDOCODE that will complement Zouev's assembler.
- Ability to discern data from instructions
- Simulation of the ERA architecture including: Program Counter (PC), Instruction Registry (IR), Register FILE (RF), memory contents and Syndrome Structure.
- Step by Step Execution.
- Breakpoints.
- Overflow warning
- Logging.
- Ability to compare results of simulation execution with the results of Altera execution.

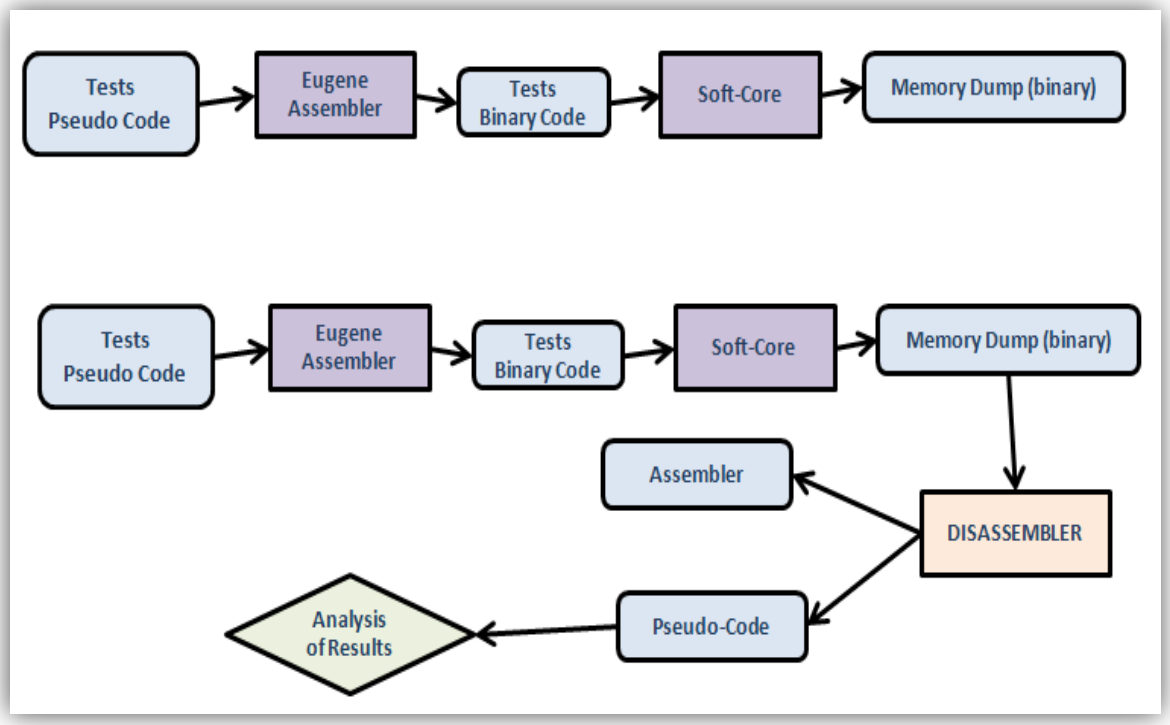


Figure 1. Flow of ERRIC testing (top) and Flow of ERRIC testing with the help of a disassembler

The development of such disassembler/simulator gives us the possibility of 1) testing and location of errors of design of the soft core processor; 2) understand the smallest details of the ERRIC functionality; 3) Simulation of the current version of the processor and the FT version of the processor and 4) Fault Injection.

Basic disassembling (1) and ability to differentiate data from instructions are already implemented. The rest of the features will be developed in the next two months. I believe that this tool could have an application in a learning environment. It could be easily modified to be used in any computing architecture or software engineer module to explain how computers work at the very low level.

Interface and current state

The design is completed and the user interface is fully defined (see Figure 2 and Figure 3 below):

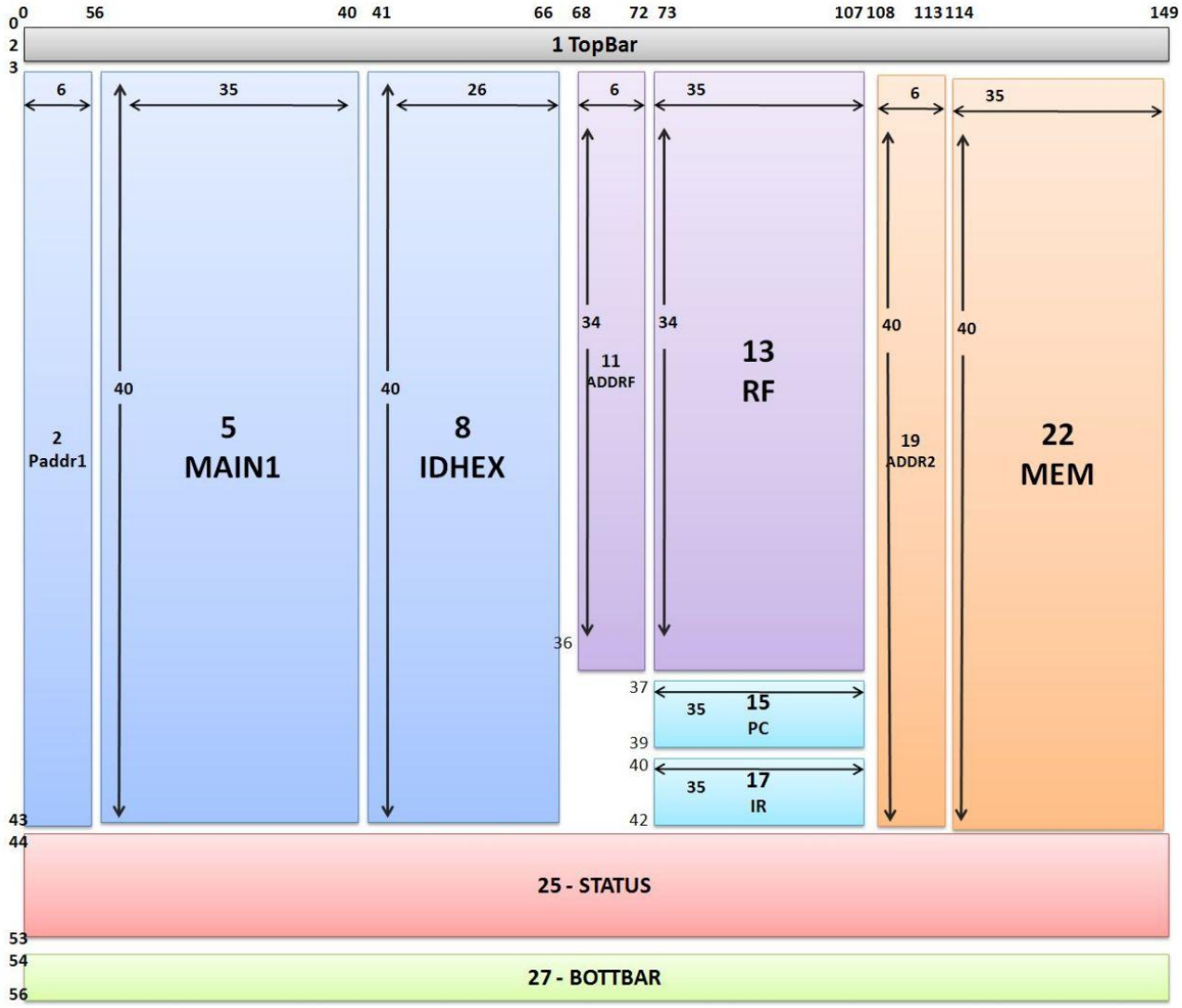


Figure 2. Design of the Interface of the current version of the simulator v0.1a

The implementation is still a working progress. Nonetheless, several parts have already been completed :

1) The design is completed and the user interface is fully defined.

2) Assembling of pseudo code using ERA assembler/preparator (100% completed).

3) Disassembling of binary into human readable code (assembly code) (100% completed).

4) The simulator is capable of parsing the binary file resulting from the previous step and is then capable of classifying data and instructions (100% completed).

5) Simulation of main memory, register file, program counter and instruction registry is almost completed (90%).

Figure 3. Interface of the current version of the simulator v0.1a

Log sample

Example of Log file related to the execution of the disassembler

fname = log.txt

```
 Number of Bytes = 68 bytes
 Number of 32-bit instruction-data = 17

 0            C800                 C000
   11 0010 00000 00000    11 0000 00000 00000
    LDA    R0   R0             NOP-STOP   R0   R0


 1            0000                 0001
   00 0000 00000 00000    00 0000 00000 00001
    NOP-STOP    R0   R0             NOP-STOP   R0   R1


 2            C801                 C000
   11 0010 00000 00001    11 0000 00000 00000
    LDA    R0   R1             NOP-STOP   R0   R0


 3            0000                 000A
   00 0000 00000 00000    00 0000 00000 01010
    NOP-STOP    R0   R0             NOP-STOP   R0   R10


 4            C802                 C000
   11 0010 00000 00010    11 0000 00000 00000
    LDA    R0   R2             NOP-STOP   R0   R0


 5            0000                 000A
   00 0000 00000 00000    00 0000 00000 01010
    NOP-STOP    R0   R0             NOP-STOP   R0   R10


 6            C803                 C000
   11 0010 00000 00011    11 0000 00000 00000
    LDA    R0   R3             NOP-STOP   R0   R0


 7            0000                 000B
   00 0000 00000 00000    00 0000 00000 01011
    NOP-STOP    R0   R0             NOP-STOP   R0   R11


 8            C804                 C000
   11 0010 00000 00100    11 0000 00000 00000
    LDA    R0   R4             NOP-STOP   R0   R0


 9            0000                 0064
```

```
  00 0000 00000 00000    00 0000 00011 00100
   NOP-STOP    R0   R0             NOP-STOP   R3   R4


10             C805                  C000
  11 0010 00000 00101    11 0000 00000 00000
   LDA     R0   R5             NOP-STOP   R0   R0


11             0000                  0065
  00 0000 00000 00000    00 0000 00011 00101
   NOP-STOP    R0   R0             NOP-STOP   R3   R5


12             C806                  C000
  11 0010 00000 00110    11 0000 00000 00000
   LDA     R0   R6             NOP-STOP   R0   R0


13             0000                  006E
  00 0000 00000 00000    00 0000 00011 01110
   NOP-STOP    R0   R0             NOP-STOP   R3   R14


14             C807                  DC41
  11 0010 00000 00111    11 0111 00010 00001
   LDA     R0   R7             ASL   R2   R1


15             0000                  006F
  00 0000 00000 00000    00 0000 00011 01111
   NOP-STOP    R0   R0             NOP-STOP   R3   R15


16             0000                  C000
  00 0000 00000 00000    11 0000 00000 00000
   NOP-STOP    R0   R0             NOP-STOP   R0   R0



Size of File          = 68 bytes
Number of Elements   = 17


1              C800                  C000
  11 0010 00000 00000    11 0000 00000 00000
   LDA R0 R0   meaning R0:=CONSTANT    meaning   R0:=1     meaning   R0:=1


3              C801                  C000
  11 0010 00000 00001    11 0000 00000 00000
   LDA R0 R1   meaning R1:=CONSTANT    meaning   R1:=10     meaning   R1:=10


5              C802                  C000
```

11 0010 00000 00010 11 0000 00000 00000
LDA R0 R2 meaning R2:=CONSTANT meaning R2:=10 meaning R2:=10

7 C803 C000
11 0010 00000 00011 11 0000 00000 00000
LDA R0 R3 meaning R3:=CONSTANT meaning R3:=11 meaning R3:=11

9 C804 C000
11 0010 00000 00100 11 0000 00000 00000
LDA R0 R4 meaning R4:=CONSTANT meaning R4:=100 meaning R4:=100

11 C805 C000
11 0010 00000 00101 11 0000 00000 00000
LDA R0 R5 meaning R5:=CONSTANT meaning R5:=101 meaning R5:=101

13 C806 C000
11 0010 00000 00110 11 0000 00000 00000
LDA R0 R6 meaning R6:=CONSTANT meaning R6:=110 meaning R6:=110

15 C807 DC41
11 0010 00000 00111 11 0111 00010 00001
LDA R0 R7 meaning R7:=CONSTANT meaning R7:=111 meaning R7:=111

17 0000 C000
00 0000 00000 00000 11 0000 00000 00000
NOP-STOP R0 R0 meaning STOP instruction
NOP-STOP R0 R0 meaning NOP instruction

Todo List

Global fParseFile (char *iniFileName, s_item *pCode)

To load, not only the code list, but also the RAM. Disabling the comment for "pRAM[i-1]=pCode[i].HLLL;" will be the only thing left to do.

File viewbin.cpp

Bug List

Global main (int argc, char *argv[])

File viewbin.cpp

File Index

File List

Here is a list of all files with brief descriptions:

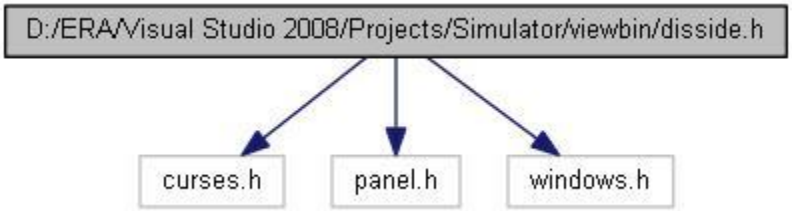
File Documentation

D:/ERA/Visual Studio 2008/Projects/Simulator/viewbin/disside.h File Reference

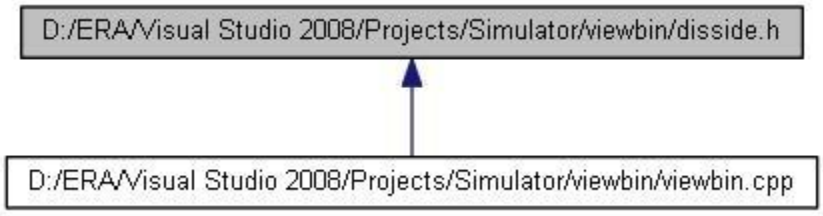
Header file for the Visual Interface of Dissimera.

```
#include <curses.h>
#include <panel.h>
#include <windows.h>
```

Include dependency graph for disside.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **s_coord**
- struct **s_panel**
- struct **s_winborder**
- struct **s_window**

Defines

- #define **ESCAPE_KEY** 27
- #define **CTRL_LEFT_KEY** 443
- #define **CTRL_RIGHT_KEY** 444
- #define **CTRL_UP_KEY** 480
- #define **CTRL_DOWN_KEY** 481
- #define **KEY_F1** 265
- #define **KEY_F2** 266
- #define **KEY_F3** 267
- #define **KEY_F4** 268
- #define **KEY_F5** 269
- #define **KEY_F6** 270
- #define **KEY_F7** 271
- #define **KEY_F8** 272
- #define **KEY_F9** 273

- #define **KEY_F10** 274
- #define **KEY_F11** 275
- #define **KEY_F12** 276
- #define **KEY_PGDOWN** 338
- #define **KEY_PGUP** 339
- #define **NUMPANELS** 30
- #define **NUMWINDOWS** 30
- #define **BUFFERWIN** 250
- #define **P_NULL** -1
- #define **P_TOPBAR** 0
- #define **P_SUBTOPBAR** 1
- #define **P_ADDR1** 2
- #define **P_SUBADDR1** 3
- #define **P_PADDR1** 4
- #define **P_MAIN1** 5
- #define **P_SUBMAIN1** 6
- #define **P_PMAIN1** 7
- #define **P_IDHEX** 8
- #define **P_SUBIDHEX** 9
- #define **P_PIDHEX** 10
- #define **P_ADDRRF** 11
- #define **P_SUBADDRRF** 12
- #define **P_RF** 13
- #define **P_SUBRF** 14
- #define **P_PC** 15
- #define **P_SUBPC** 16
- #define **P_IR** 17
- #define **P_SUBIR** 18
- #define **P_ADDR2** 19
- #define **P_SUBADDR2** 20
- #define **P_PADDR2** 21
- #define **P_MEM** 22
- #define **P_SUBMEM** 23
- #define **P_PMEM** 24
- #define **P_STATUS** 25
- #define **P_SUBSTATUS** 26
- #define **P_BOTTBAR** 27
- #define **P_SUBBOTTBAR** 28
- #define **E_NORMALRUN** -1
- #define **E_NOP** -1
- #define **E_STEPBSTEP** 1

Functions

- int **fInitWindows** (**s_window** *p_window)
- int **fInitPanels** (**s_panel** *p_panel)
- void **fHighlight_line** (WINDOW *pwin, int pline, unsigned int pbackgcolor)
- void **fUnHighlight_line** (WINDOW *pwin, int pline)
- int **fUpdateCursors_Main** (**s_window** *p_window, int p_iLastIntr, int p_iNewInstr, unsigned int pbackgcolor)
- int **fSetTerm** ()
  It sets a long buffer and Size of the terminal (maximizing it).
- void **fIniP_TopBar** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_ADDR1** (**s_window** *p_window, **s_panel** *p_panel, unsigned short int p_numInstr)

- void **fIniP_MAIN1** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_IDHEX** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_ADDRRF** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_RF** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_PC** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_IR** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_ADDR2** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_MEM** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_STATUS** (**s_window** *p_window, **s_panel** *p_panel)
- void **fIniP_BOTTBAR** (**s_window** *p_window, **s_panel** *p_panel)
- int **fSetIDE** (**s_panel** *p_panel, **s_window** *p_window, unsigned short int p_numInstr)
- void **fHighlightPanel** (short int pwin, char pflag, **s_window** *p_window)
- short **fmovePanel** (int pkey, short *pActiveP, **s_panel** *p_panels, **s_window** *p_window)
- int **fBootHardware** (**s_panel** *p_panels, **s_window** *p_window, **s_item** *pCode, unsigned *pRF, **s_item** *plRAM, unsigned pPC)
- int **fLoadPanels** (**s_panel** *p_panels, **s_window** *p_window, **s_item** *pCode, unsigned *pRF, **s_item** *plRAM, unsigned pPC)
- int **fExecuteCode** (**s_panel** *p_panels, **s_window** *p_window, **s_item** *pCode, unsigned *pRF, **s_item** *plRAM, unsigned *pPC, **s_exec** *p_exec, int p_nIexec)
- int **fHandleKeyDown** (**s_panel** *p_panels, **s_window** *p_window, **s_item** *pCode, unsigned *pRF, **s_item** *plRAM, unsigned *pPC, **s_exec** *p_exec)
- int **fHandleKeyUp** (**s_panel** *p_panels, **s_window** *p_window, **s_item** *pCode, unsigned *pRF, **s_item** *plRAM, unsigned *pPC, **s_exec** *p_exec)

Detailed Description

Header file for the Visual Interface of Dissimera.

```
\author        Victor Castano
\version  0.1a
```

\date 08\03\2012 It contains the variables and functions related to the interface

Definition in file **disside.h**.

Data Structure Documentation

struct s_coord

Definition at line 104 of file disside.h.

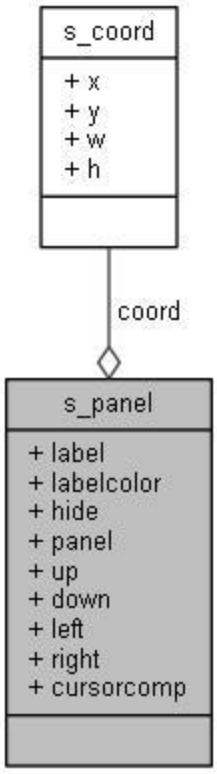
Data Fields:

unsigned int	h	
unsigned int	w	
unsigned int	x	
unsigned int	y	

struct s_panel

Definition at line 111 of file disside.h.

Collaboration diagram for s_panel:



Data Fields:

s_coord	coord	
short int	cursorcomp	
short int	down	
unsigned char	hide	
char	label	
int	labelcolor	
short int	left	
PANEL *	panel	
short int	right	
short int	up	

struct s_winborder

The parameters elements are

- ls: character to be used for the left side of the window
- rs: character to be used for the right side of the window
- ts: character to be used for the top side of the window
- bs: character to be used for the bottom side of the window
- tl: character to be used for the top left corner of the window
- tr: character to be used for the top right corner of the window
- bl: character to be used for the bottom left corner of the window
- br: character to be used for the bottom right corner of the window

Definition at line 137 of file disside.h.

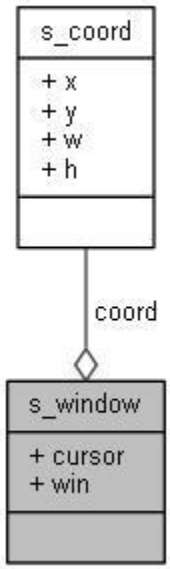
Data Fields:

chtype	bl	
chtype	br	
chtype	bs	
chtype	ls	
chtype	rs	
chtype	tl	
chtype	tr	
chtype	ts	

struct s_window

Definition at line 145 of file disside.h.

Collaboration diagram for s_window:



Data Fields:

s_coord	coord	
short int	cursor	
WINDOW *	win	

Define Documentation

#define BUFFERWIN 250

Definition at line 50 of file disside.h.

Referenced by fIniP_ADDR1(), fIniP_IDHEX(), and fIniP_MAIN1().

#define CTRL_DOWN_KEY 481

Definition at line 26 of file disside.h.

Referenced by fmovePanel(), and main().

#define CTRL_LEFT_KEY 443

Definition at line 23 of file disside.h.
Referenced by fmovePanel(), and main().

#define CTRL_RIGHT_KEY 444

Definition at line 24 of file disside.h.
Referenced by fmovePanel(), and main().

#define CTRL_UP_KEY 480

Definition at line 25 of file disside.h.
Referenced by fmovePanel(), and main().

#define E_NOP -1

Definition at line 99 of file disside.h.

#define E_NORMALRUN -1

Definition at line 98 of file disside.h.

#define E_STEPBSTEP 1

Definition at line 100 of file disside.h.

#define ESCAPE_KEY 27

Definition at line 22 of file disside.h.
Referenced by main().

#define KEY_F1 265

Definition at line 27 of file disside.h.

#define KEY_F10 274

Definition at line 36 of file disside.h.
Referenced by main().

#define KEY_F11 275

Definition at line 37 of file disside.h.

#define KEY_F12 276

Definition at line 38 of file disside.h.

#define KEY_F2 266

Definition at line 28 of file disside.h.

#define KEY_F3 267

Definition at line 29 of file disside.h.

#define KEY_F4 268

Definition at line 30 of file disside.h.

#define KEY_F5 269

Definition at line 31 of file disside.h.
Referenced by main().

#define KEY_F6 270

Definition at line 32 of file disside.h.

#define KEY_F7 271

Definition at line 33 of file disside.h.

#define KEY_F8 272

Definition at line 34 of file disside.h.

#define KEY_F9 273

Definition at line 35 of file disside.h.
Referenced by main().

#define KEY_PGDOWN 338

Definition at line 43 of file disside.h.
Referenced by main().

#define KEY_PGUP 339

Definition at line 44 of file disside.h.
Referenced by main().

#define NUMPANELS 30

Definition at line 48 of file disside.h.
Referenced by fInitPanels(), and main().

#define NUMWINDOWS 30

Definition at line 49 of file disside.h.
Referenced by fInitWindows(), and main().

#define P_ADDR1 2

Definition at line 57 of file disside.h.
Referenced by fHighlightPanel(), fIniP_ADDR1(), fIniP_MAIN1(), and fUpdateCursors_Main().

#define P_ADDR2 19

Definition at line 81 of file disside.h.
Referenced by fHighlightPanel(), fIniP_ADDR2(), fIniP_IR(), fIniP_MEM(), fIniP_PC(), and fIniP_RF().

#define P_ADDRRF 11

Definition at line 69 of file disside.h.
Referenced by fIniP_ADDRRF(), fIniP_IDHEX(), and fIniP_RF().

#define P_BOTTBAR 27

Definition at line 92 of file disside.h.
Referenced by fIniP_BOTTBAR(), and fIniP_STATUS().

#define P_IDHEX 8

Definition at line 65 of file disside.h.
Referenced by fHighlightPanel(), fIniP_ADDRRF(), fIniP_IDHEX(), fIniP_IR(), fIniP_MAIN1(), fIniP_PC(), fLoadPanels(), and fUpdateCursors_Main().

#define P_IR 17

Definition at line 78 of file disside.h.
Referenced by fBootHardware(), fIniP_IR(), and fIniP_PC().

#define P_MAIN1 5

Definition at line 61 of file disside.h.
Referenced by fHighlightPanel(), fIniP_ADDR1(), fIniP_IDHEX(), fIniP_MAIN1(), fIniP_STATUS(), fIniP_TopBar(), fLoadPanels(), fUpdateCursors_Main(), and main().

#define P_MEM 22

Definition at line 85 of file disside.h.
Referenced by fHighlightPanel(), fIniP_ADDR2(), and fIniP_MEM().

#define P_NULL -1

Definition at line 52 of file disside.h.

Referenced by fIniP_ADDR1(), fIniP_BOTTBAR(), fIniP_MEM(), fIniP_STATUS(), fIniP_TopBar(), fInitPanels(), and fmovePanel().

#define P_PADDR1 4

Definition at line 59 of file disside.h.

#define P_PADDR2 21

Definition at line 83 of file disside.h.

#define P_PC 15

Definition at line 75 of file disside.h.

Referenced by fBootHardware(), fIniP_ADDRRF(), fIniP_IR(), fIniP_PC(), and fIniP_RF().

#define P_PIDHEX 10

Definition at line 67 of file disside.h.

#define P_PMAIN1 7

Definition at line 63 of file disside.h.

#define P_PMEM 24

Definition at line 87 of file disside.h.

#define P_RF 13

Definition at line 72 of file disside.h.

Referenced by fBootHardware(), fIniP_ADDR2(), fIniP_ADDRRF(), fIniP_PC(), and fIniP_RF().

#define P_STATUS 25

Definition at line 89 of file disside.h.

Referenced by fIniP_ADDR1(), fIniP_ADDR2(), fIniP_BOTTBAR(), fIniP_IDHEX(), fIniP_IR(), fIniP_MAIN1(), fIniP_MEM(), and fIniP_STATUS().

#define P_SUBADDR1 3

Definition at line 58 of file disside.h.

Referenced by fIniP_ADDR1().

#define P_SUBADDR2 20

Definition at line 82 of file disside.h.
Referenced by fIniP_ADDR2().

#define P_SUBADDRRF 12

Definition at line 70 of file disside.h.
Referenced by fIniP_ADDRRF().

#define P_SUBBOTTBAR 28

Definition at line 93 of file disside.h.
Referenced by fIniP_BOTTBAR().

#define P_SUBIDHEX 9

Definition at line 66 of file disside.h.
Referenced by fIniP_IDHEX(), and fLoadPanels().

#define P_SUBIR 18

Definition at line 79 of file disside.h.
Referenced by fIniP_IR().

#define P_SUBMAIN1 6

Definition at line 62 of file disside.h.
Referenced by fIniP_MAIN1(), and fLoadPanels().

#define P_SUBMEM 23

Definition at line 86 of file disside.h.
Referenced by fIniP_MEM().

#define P_SUBPC 16

Definition at line 76 of file disside.h.
Referenced by fIniP_PC().

#define P_SUBRF 14

Definition at line 73 of file disside.h.
Referenced by fIniP_RF().

#define P_SUBSTATUS 26

Definition at line 90 of file disside.h.
Referenced by fBootHardware(), fIniP_STATUS(), fLoadPanels(), fmovePanel(), and fSetIDE().

#define P_SUBTOPBAR 1

Definition at line 55 of file disside.h.

Referenced by fIniP_TopBar(), and main().

#define P_TOPBAR 0

Definition at line 54 of file disside.h.

Referenced by fIniP_ADDR1(), fIniP_ADDR2(), fIniP_ADDRRF(), fIniP_IDHEX(), fIniP_MAIN1(), fIniP_MEM(), fIniP_RF(), and fIniP_TopBar().

Function Documentation

int fBootHardware (s_panel * *p_panels*, s_window * *p_window*, s_item * *pCode*, unsigned * *pRF*, s_item * *pIRAM*, unsigned *pPC*)

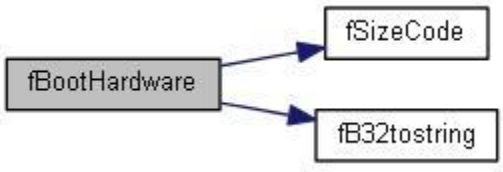
Definition at line 1050 of file disside.h.

Referenced by main().

```
{
     unsigned int   lSizeCode = 0;
     unsigned int   i=0;
     unsigned int   lIR=0;
     char           lstInstr[64]="";
     char           lstreg[64]="";

     waddstr(p_window[P_SUBSTATUS].win, " Booting Hardware ...  ");
     wrefresh(p_window[P_SUBSTATUS].win);


     //Obtaining number of Elements
     lSizeCode = fSizeCode(pCode);

////////////////////////////////////////////////////////////////
/*
     //Loading the MEMORY PANEL
     for (i=1; i<=lSizeCode; i++)
     {
          fB16tostring(pCode[i].HL,lstInstr);
          mvwaddnstr(p_window[P_MEM].win, i, 0,lstInstr,-1);
          fB16tostring(pCode[i].LL,lstInstr);
          mvwaddnstr(p_window[P_MEM].win, i, 18, lstInstr,-1);
     }
     wrefresh(p_window[P_MEM].win);
*/
     //Loading the Register File
     for (i=0; i<32; i++)
     {
          fB32tostring(pRF[i],lstreg);
          mvwaddnstr(p_window[P_RF].win, i+1, 1,lstreg,-1);
     }
     wrefresh(p_window[P_RF].win);


     //Loading Program Counter
     strcpy(lstreg, "");
     sprintf(lstreg, "%08X", pPC);
     mvwaddnstr(p_window[P_PC].win, 1, 14, lstreg,-1);
     wrefresh(p_window[P_PC].win);

     //Loading Instruction Register
     strcpy(lstreg, "");
```

```
     sprintf(lstreg, "%08X", lIR);
     mvwaddnstr(p_window[P_IR].win, 1, 14, lstreg,-1);
     wrefresh(p_window[P_IR].win);
////////////////////////////////////////////////////////////////// end comment

     waddstr(p_window[P_SUBSTATUS].win, "Hardware Booted\n");
     wrefresh(p_window[P_SUBSTATUS].win);

     return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

int fExecuteCode (s_panel * *p_panels*, s_window * *p_window*, s_item * *pCode*, unsigned * *pRF*, s_item * *pIRAM*, unsigned * *pPC*, s_exec * *p_exec*, int *p_nIexec*)

Definition at line 1175 of file disside.h.

Referenced by main().

```
{
     // p_nIexec represents the number of instructions to be executed
     //        (-1 means normal execution, 0 means no execution)
     // fExecuteCode function returns the number of instructions executed.
     //       -1 as a result means an error has occured during execution.

     //s_item            item;
     unsigned int        lsize=0;
     unsigned int        i=0;
     unsigned short int  lEOLOAD=0;          // Variable that will represent which load is active
High (1) Load or Low (2) Load
     bool                lEOCODE=0;          // Variable that will represent whether the END OF
CODE has been reached
     s_instruction       lInstr;             // Local var that stores the instruction: either
High or Low
     int                      lnInstEx=0;         // Number of instructions executed

     //Obtain number of Elements
     //lsize=pCode[0].linenum;
     lsize=fSizeCode(pCode);

     fptraza(gflog, 1, "\n Executing code");
     fptraza(gflog, 1, "\n Size of Array       = %d bytes", lsize*4);
     fptraza(gflog, 3, "\n Number of Elements  = %d \n", lsize);

     //Check if the Code list is empty
     if (lsize <=0)
     {
          return -1;     //Empty code list
     }

     /*   Browse the list of Code (High and Low part) starting by the first element
          By default every item was set as T_32BITDATA, we are now browsing only instruction by
instruction
          in order to mark instructions from the data
     */
```

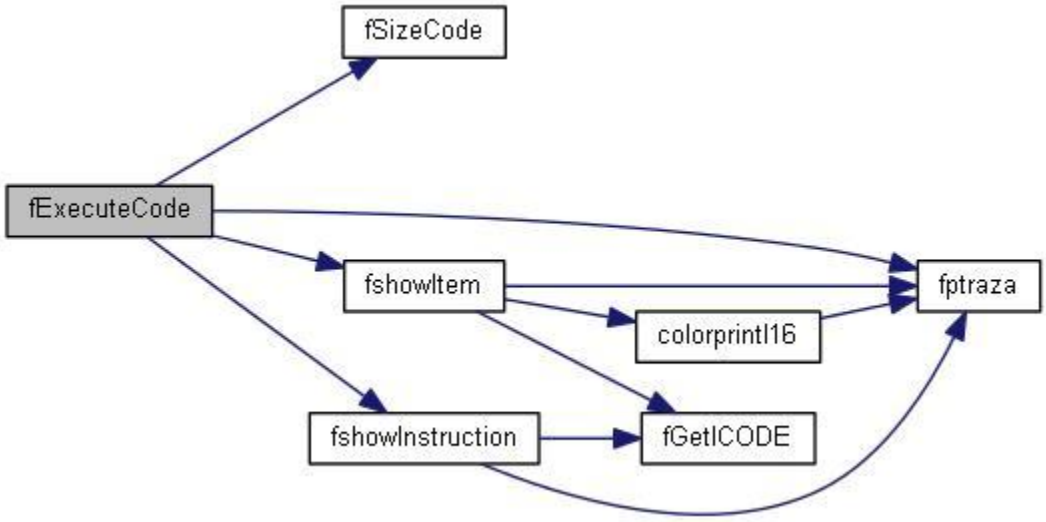
```
     i=*pPC+1; // Position in the code list is 1+ due to the first element being used for
configuration
     do
     {
          fshowItem(&pCode[i]);
          //pCode[i].type=T_INSTRUCTIONS;         // The item is definitely an instruction load
          lEOLOAD=1;
          do //Browse firts the high and then the low part (specified by lEOLOAD)
          {
               if (lEOLOAD==1)
               {
                    lInstr=pCode[i].InstrA;
               }
               else
               {
                    lInstr=pCode[i].InstrB;
                    fptraza(gflog, 1, "\n");
               }

               fshowInstruction(&lInstr);
               p_exec->uiLastInstr=*pPC;


               switch (lInstr.i_code)
               {
               case 0:   //The NOP instruction performs no actions, except moving the PC register
to the next instruction.
                    switch (lInstr.f_code)
                    {
                    case 0:                              // x00b STOP instruction
                         fptraza(gflog, 1, " meaning STOP instruction ");
                         lEOCODE=true;            // End of CODE
                         break;
                    case 1:                              // x01b NOP instruction
                         //break;                 //Assuming that 01 and 11 for f_code are a NOP
instruction (check with Igor, Thomas and Eugene
                    case 3:                              // QUESTION: x11b According to the
Document Instruction Set this is not possible but it seems to be a NOP instruction
                         fptraza(gflog, 1, " meaning NOP instruction ");
                         break;
                    default:
                         fptraza(gflog, 1, "THIS IS NOT POSSIBLE - a 01??? in the format code?");
                    }

                    break;
               case 1:   // The LD instruction copies the value of a 32-bit memory word pointed
to by Ri into Rj
                    fptraza(gflog, 1, " meaning R%d:=*R%d ", lInstr.Op2, lInstr.Op1);
                    //Loading Rj with the vaule from the memory location
                    pRF[lInstr.Op2]=plRAM[lInstr.Op1].HLLL;
                    fptraza(gflog, 1, "   meaning  R%d:=%d ", lInstr.Op2,
plRAM[lInstr.Op1].HLLL);
                    break;
               case 2:   // The LDA instruction takes the value from the next 32-bit word and stores
the result into Rj
                         // constant stored in the Next 32bit location
                    fptraza(gflog, 1, " meaning R%d:=CONSTANT", lInstr.Op2);
                    fptraza(gflog, 1, "   meaning  R%d:=%d ", lInstr.Op2, pCode[i+1].HLLL);
                    pRF[lInstr.Op2]=pCode[i+1].HLLL;   //Loading Rj with the next 32bit word
                    fptraza(gflog, 1, "   meaning  R%d:=%d ", lInstr.Op2, pRF[lInstr.Op2]);

                    if (lEOLOAD ==1)
                    {
                         lnInstEx++; // Not sure if this is correct. Does LDA execute the low 16
bits????
                         lEOLOAD++;     // Jumping the load
                    }
                    i++; // Jumping to the following instruction (the one after the data)

                    break;
```

```
               case 3:   // The ST instruction copies the value of Ri to the memory by address
taken from Rj
                    fptraza(gflog, 1, " meaning *R%d:=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, "   meaning  MEM[%d]=%d ", pRF[lInstr.Op2],
pRF[lInstr.Op1]);
                    plRAM[pRF[lInstr.Op2]].HLLL = pRF[lInstr.Op1];         // MEM(Rj)=Ri
                    fptraza(gflog, 1, "   meaning  MEM[%d]=%d ", pRF[lInstr.Op2],
plRAM[pRF[lInstr.Op2]].HLLL);
                    break;
               case 4:   // The MOV instruction copies the value from Ri to the Rj
                         // Assuming that format code is fixed to 11
                    fptraza(gflog, 1, " meaning R%d:=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d:=%d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2]=pRF[lInstr.Op1];   // Rj=Ri
                    break;
               case 5:   // The ADD instruction denotes the two's complement arithmetic addition.
                         // The contents of Ri and Rj are arithmetically added, and the result
is put into Rj.
                         // Assuming that format code is fixed to 11
                    fptraza(gflog, 1, " meaning R%d+=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=R%d+R%d", lInstr.Op2,lInstr.Op2,
lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=%d+%d", lInstr.Op2,pRF[lInstr.Op2],
pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] + pRF[lInstr.Op2];        // Rj=Rj+Ri
                    break;
               case 6:   /*   The SUB instruction denotes the two's complement arithmetic
subtraction.
                              The content of Ri is subtracted from the contents of Rj, and the
result
                              is put into the register Rj
                         */
                    fptraza(gflog, 1, " meaning R%d-=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=R%d-R%d", lInstr.Op2,lInstr.Op2,
lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=%d-%d", lInstr.Op2,pRF[lInstr.Op2],
pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] - pRF[lInstr.Op2];        // Rj=Rj+Ri
                    break;
               case 7:   // The ASR instruction arithmetically shifts Ri one bit right, and puts
the result into Rj.
                    fptraza(gflog, 1, " meaning R%d >>= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d >>= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] >> 1;               // one bit right
shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d >>= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 8:   /*   The ASL instruction arithmetically shifts the contents of Ri one
bit left, and puts
                         the result into the register Rj. */
                    fptraza(gflog, 1, " meaning R%d <<= R%d", lInstr.Op2, lInstr.Op1);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    fptraza(gflog, 1, " meaning R%d <<= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] << 1;               // one bit left
shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d <<= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 9:   /*   The OR instruction applies logical addition ("OR") operator to every
pair
                              of bits taken from Ri and Rj, respectively, and puts the result into
Rj. */
                    fptraza(gflog, 1, " meaning R%d |= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d |= %d", lInstr.Op2,  pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] | pRF[lInstr.Op1];
                    fptraza(gflog, 1, " meaning R%d |= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 10: /*    The AND instruction applies logical multiplicative ("AND") operator
to
                              every pair of bits taken from Ri and Rj, respectively, and puts the
                              result into Rj */
```

```
                    fptraza(gflog, 1, " meaning R%d &= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d &= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] & pRF[lInstr.Op1];
                    fptraza(gflog, 1, " meaning R%d &= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 11: /*    The XOR instruction applies logical exclusive OR ("XOR") operator
to every
                              pair of bits taken from Ri and Rj, respectively, and puts the result
into Rj. */
                    fptraza(gflog, 1, " meaning R%d ^= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d ^= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = ( pRF[lInstr.Op2] || pRF[lInstr.Op1] ) && !(pRF[lInstr.Op2]
&& pRF[lInstr.Op1]);
                    fptraza(gflog, 1, " meaning R%d ^= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    break;
               case 12: /*    The LSL instruction logically shifts the contents of Ri one bit left,
and
                              puts the result into the Rj.  */
                    fptraza(gflog, 1, " meaning R%d <= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d <= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] << 1;               // one bit logicleft
right shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d <= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 13:
                    /*   The LSR instruction logically shifts the contents of Ri one bit right,
and
                         puts the result into the register Rj.   */
                    fptraza(gflog, 1, " meaning R%d >= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d >= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] >> 1;               // one bit logic right
shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d >= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 14:
                    /*   The CND instruction arithmetically compares the contents Ri and Rj
                         and puts the result of the comparison (as a set of 1-bit signs) to Rj.
*/
                    fptraza(gflog, 1, " meaning R%d ?= R%d", lInstr.Op2, lInstr.Op1);
                    // Victor When executing we can check the result and show it
                    break;
               case 15:
                    /*   The CBR instruction checks the contents of Ri. If it is non-zero,
                         then:
                              1)   the address of the next instruction (i.e., current value of
                                   the PC register + 1) is stored in the Ri register, and
                              2)   the value of Rj is set to the PC register. This means that
                                   the next instruction will be fetched by the address taken Rj
                    */
                    fptraza(gflog, 1, " meaning if R%d GOTO R%d", lInstr.Op1, lInstr.Op2);
                    break;
               default:
                    return -1; //This should never happen;
               }

               lEOLOAD++;
               lnInstEx++;

          } while (lEOLOAD <= 2);  // Loop finishes when every part of the load (High and Low)
has been processed

          i++;
          (*pPC)++;

     }while (lnInstEx<p_nIexec && lEOCODE==FALSE);

     p exec->nInstrEx=lnInstEx;
     return lnInstEx;
```

```
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

int fHandleKeyDown (s_panel * *p_panels*, s_window * *p_window*, s_item * *pCode*, unsigned * *pRF*, s_item * *pIRAM*, unsigned * *pPC*, s_exec * *p_exec*)

Definition at line 1390 of file disside.h.

Referenced by main().

```
{

     return 0;
}
```

Here is the caller graph for this function:

int fHandleKeyUp (s_panel * *p_panels*, s_window * *p_window*, s_item * *pCode*, unsigned * *pRF*, s_item * *pIRAM*, unsigned * *pPC*, s_exec * *p_exec*)

Definition at line 1399 of file disside.h.

Referenced by main().

```
{

     return 0;
}
```

Here is the caller graph for this function:

void fHighlight_line (WINDOW * *pwin*, int *pline*, unsigned int *pbackgcolor*)

Definition at line 193 of file disside.h.

Referenced by fUpdateCursors_Main().

```
{
     mvwchgat(pwin, pline, 1, pwin->_maxx-2,  pbackgcolor, 0, NULL);
     wrefresh(pwin);
}
```

Here is the caller graph for this function:

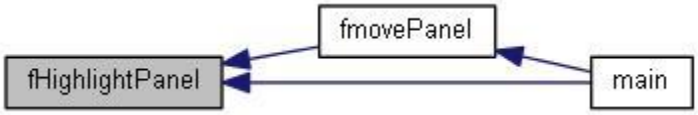
void fHighlightPanel (short int *pwin*, char *pflag*, s_window * *p_window*)

Definition at line 879 of file disside.h.

Referenced by fmovePanel(), and main().

```
{
     if (pflag==1)
     // Enable highlighting of current Panel
     {
          //wattrset(p_win, COLOR_PAIR(2) | A_BOLD);
          wattron(p_window[pwin].win, COLOR_PAIR(2) | A_BOLD);
          box(p window[pwin].win, 0 , 0);
          wrefresh(p_window[pwin].win);
          wattroff(p_window[pwin].win, COLOR_PAIR(2) | A_BOLD);
     }
     else
     {
          // Disable highlighting of current Panel
          wattron(p_window[pwin].win, COLOR_PAIR(0));
          box(p_window[pwin].win, 0 , 0);
          wrefresh(p_window[pwin].win);
          wattron(p_window[pwin].win, COLOR_PAIR(0));

     }

     //Refreshing the subwindows if there is any
     switch(pwin)
     {
          case P_ADDR1:
          case P_MAIN1:
          case P_IDHEX:
          case P_ADDR2:
          case P_MEM:
               wrefresh(p_window[pwin+1].win);
               wattroff(p_window[pwin+1].win, COLOR_PAIR(2) | A_BOLD);
               wrefresh(p_window[pwin+1].win); // Refreshing the Subwindow or pad in this case
(next window)
               break;
          default:
               break;
     }

     return;
}
```

Here is the caller graph for this function:



void fIniP_ADDR1 (s_window * *p_window*, s_panel * *p_panel*, unsigned short int *p_numInstr*)

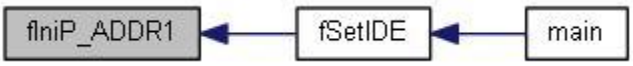
Definition at line 304 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                 coord = {0,0,0,0};
     char               ls_label[255]="";
     int                     lcolor=1;
     unsigned char      lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     // We first Create and Initialize the Window
     i = P_ADDR1;

     coord.x=0; coord.y=3; coord.h=40; coord.w=6;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);
     wrefresh(p_window[i].win);


     // Creating subwindows within the window
     p_window[P_SUBADDR1].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     if ( p_window[P_SUBADDR1].win == NULL)
     {
          addstr("Unable to create new subwindow");
          refresh();
          endwin();
          return;
     }
     //wrefresh(p_window[P_SUBADDR1].win);
     p_window[P_SUBADDR1].win = newpad(BUFFERWIN, coord.w-2);
     if ( p_window[P_SUBADDR1].win == NULL)
     {
          addstr("Unable to create new pad");
          refresh();
          endwin();
          return;
     }


     //Creating and Setting-Up a new Panel
     p panel[i].coord = coord;
     sprintf(ls_label, " ADDRESS 1 PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_TOPBAR;
     p_panel[i].down=P_STATUS;
     p panel[i].left=P NULL;
     p_panel[i].right=P_MAIN1;
     p_panel[i].cursorcomp=0;
     ///////p_panel[i].panel = new_panel(p_window[P_SUBADDR1].win);

     //wrefresh(p_window[i].win);

     //Writing HEX Addresses on pad
     unsigned short lhexadd=0;
     char           lsaddr[10];
     for (lhexadd=0; lhexadd<p_numInstr-1; lhexadd++)
     {
          sprintf(lsaddr,"%04x", lhexadd);
          mvwaddnstr(p_window[P_SUBADDR1].win, lhexadd,0,lsaddr,-1);
          //prefresh(p_window[P_SUBADDR1].win,0,0,coord.y+1,coord.x+1,coord.h+1,coord.w);
          //wrefresh(p_window[P_SUBADDR1].win);
     }

     //wrefresh(p_window[i].win);
```

```
     prefresh(p_window[P_SUBADDR1].win,0,0,coord.y+1,coord.x+1,coord.h+1,coord.w);

     //char ch=getch();
//   wrefresh(p_window[i].win);
//   prefresh(p_window[P_SUBADDR1].win,1,0,coord.y+1,coord.x+1,coord.h+1,coord.w);
     /*
     ch=getch();
     prefresh(p_window[P_SUBADDR1].win,2,0,coord.y+1,coord.x+1,coord.h+1,coord.w);
     ch=getch();
     prefresh(p_window[P_SUBADDR1].win,3,0,coord.y+1,coord.x+1,coord.h+1,coord.w);
     */
     return;
}
```

Here is the caller graph for this function:



void flniP_ADDR2 (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 672 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char                ls_label[255]="";
     int                      lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;
     unsigned short      lhexadd=0;
     char                lsaddr[10];

     i = P_ADDR2;

     coord.x=108; coord.y=3; coord.h=40; coord.w=6;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);

     // Creating subwindows within the window
     p_window[P_SUBADDR2].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     wrefresh(p_window[i].win);

     //Creating and Setting-Up a new Panelp_panel[i].coord = coord;
     sprintf(ls_label, " Address Memory PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_TOPBAR;
     p_panel[i].down=P_STATUS;
     p panel[i].left=P_RF;
     p_panel[i].right=P_MEM;
     p_panel[i].cursorcomp=0;
     p_panel[i].panel = new_panel(p_window[i].win);

     //Writing HEX Addresses on panel
     for (lhexadd=0; lhexadd<coord.h-2; lhexadd++)
     {
          sprintf(lsaddr,"%04x", lhexadd);
          mvwaddnstr(p_window[i].win, lhexadd+1,1,lsaddr,-1);
     }
     prefresh(p_window[i].win,108,0,3,0,40,6);
     //wrefresh(p window[i].win);

     return;
}
```

Here is the caller graph for this function:

void flniP_ADDRRF (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 499 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                 coord = {0,0,0,0};
     char               ls_label[255]="";
     char               lsreg[5]="R0";
     int                     lcolor=1;
     unsigned char      lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;
     unsigned short int  j = 0;

     i = P_ADDRRF;

     coord.x=68; coord.y=3; coord.h=34; coord.w=5;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);

     // Creating subwindows within the window
     p_window[P_SUBADDRRF].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     wrefresh(p_window[i].win);

     //Creating and Setting-Up a new Panel
     p_panel[i].coord = coord;
     sprintf(ls_label, " Address Registers PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_TOPBAR;
     p_panel[i].down=P_PC;
     p_panel[i].left=P_IDHEX;
     p_panel[i].right=P_RF;
     p_panel[i].cursorcomp=0;
     p panel[i].panel = new panel(p window[i].win);


     //Writing Register names on panel
     for (j=0; j<32; j++)
     {
          sprintf(lsreg,"R%d", j);
          mvwaddnstr(p_window[i].win, j+1,1,lsreg,-1);
     }
     wrefresh(p window[i].win);

     return;
}
```

Here is the caller graph for this function:

void flniP_BOTTBAR (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 792 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char                ls_label[255]="";
     int                      lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_BOTTBAR;

     coord.x=0; coord.y=54; coord.h=3; coord.w=149;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p window[i].win, 0, 0);

     // Creating subwindows within the window
     //CORREGIRRRRRRRRRRRRRRRRRRR
     p window[P SUBBOTTBAR].win=derwin(p window[i].win, coord.h-2, coord.w-2, 1, 1);
     wrefresh(p_window[i].win);

     //Creating and Setting-Up a new Panelp_panel[i].coord = coord;
     p_panel[i].coord = coord;
     sprintf(ls_label, " BOTTBAR ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p panel[i].hide= lhide;


     p_panel[i].up=P_STATUS;
     p panel[i].down=P NULL;
     p_panel[i].left=P_NULL;
     p_panel[i].right=P_NULL;
     p_panel[i].cursorcomp=0;
     p_panel[i].panel = new_panel(p_window[i].win);

     return;
}
```

Here is the caller graph for this function:

void flniP_IDHEX (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 436 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char                ls label[255]="";
     int                      lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_IDHEX;

     coord.x=41; coord.y=3; coord.h=40; coord.w=26;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);
     wrefresh(p_window[i].win);

     // Creating subwindows within the window
     p_window[P_SUBIDHEX].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     if ( p_window[P_SUBIDHEX].win == NULL)
     {
          addstr("Unable to create new subwindow");
```

```
          refresh();
          endwin();
          return;
     }

     //wrefresh(p_window[P_SUBIDHEX].win);
     //wrefresh(p_window[i].win);
     p_window[P_SUBIDHEX].win = newpad(BUFFERWIN, coord.w-2);
     if ( p_window[P_SUBIDHEX].win == NULL)
     {
          addstr("Unable to create subpad");
          refresh();
          endwin();
          return;
     }


     //Creating and Setting-Up a new Panel
     p_panel[i].coord = coord;
     sprintf(ls_label, " Instruction/Data Decoding PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p panel[i].hide= lhide;

     p_panel[i].up=P_TOPBAR;
     p_panel[i].down=P_STATUS;
     p_panel[i].left=P_MAIN1;
     p panel[i].right=P ADDRRF;
     p_panel[i].cursorcomp=0;

     //testing
     /*
     char           lstInstr[64]="";
     strcpy(lstInstr, "Victor1");
     mvwaddnstr(p_window[P_SUBIDHEX].win, 0,0,lstInstr,-1);
     prefresh(p_window[P_SUBIDHEX].win,0,0,coord.y+1,coord.x+2,coord.h+1,coord.w+3);
     */

     return;
}
```

Here is the caller graph for this function:

void flniP_IR (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 637 of file disside.h.

Referenced by fSetIDE().

```
{
     s coord                coord = {0,0,0,0};
     char                ls_label[255]="";
     int                      lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_IR;

     coord.x=73; coord.y=40; coord.h=3; coord.w=35;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);

     // Creating subwindows within the window
     p_window[P_SUBIR].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     wrefresh(p_window[i].win);
```

```
     //Creating and Setting-Up a new Panelp_panel[i].coord = coord;
     sprintf(ls_label, " Instruction Register ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_PC;
     p_panel[i].down=P_STATUS;
     p_panel[i].left=P_IDHEX;
     p_panel[i].right=P_ADDR2;
     p_panel[i].cursorcomp=0;
     p_panel[i].panel = new_panel(p_window[i].win);

     return;
}
```

Here is the caller graph for this function:

void flniP_MAIN1 (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 384 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                  coord = {0,0,0,0};
     char                ls_label[255]="";
     int                      lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_MAIN1;

     coord.x=6; coord.y=3; coord.h=40; coord.w=35;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);
     wrefresh(p_window[i].win);

     // Creating subwindows within the window
     p_window[P_SUBMAIN1].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     if ( p window[P SUBMAIN1].win == NULL)
     {
          addstr("Unable to create new subwindow");
          refresh();
          endwin();
          return;
     }

     //wrefresh(p_window[P_SUBMAIN1].win);
     p window[P SUBMAIN1].win = newpad(BUFFERWIN, coord.w-2);
     if ( p_window[P_SUBMAIN1].win == NULL)
     {
          addstr("Unable to create subpad");
          refresh();
          endwin();
          return;
     }

     //Creating and Setting-Up a new Panel
     p_panel[i].coord = coord;
     sprintf(ls_label, " MAIN PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_TOPBAR;
```

```
     p_panel[i].down=P_STATUS;
     p_panel[i].left=P_ADDR1;
     p_panel[i].right=P_IDHEX;
     p_panel[i].cursorcomp=0;

     return;
}
```

Here is the caller graph for this function:

void flniP_MEM (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 719 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char                ls_label[255]="";
     int                    lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_MEM;

     coord.x=114; coord.y=3; coord.h=40; coord.w=35;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);

     // Creating subwindows within the window
     p_window[P_SUBMEM].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     wrefresh(p_window[i].win);

     //Creating and Setting-Up a new Panelp_panel[i].coord = coord;
     sprintf(ls_label, " MEMORY PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_TOPBAR;
     p_panel[i].down=P_STATUS;
     p_panel[i].left=P_ADDR2;
     p_panel[i].right=P_NULL;
     p_panel[i].cursorcomp=0;
     p_panel[i].panel = new panel(p_window[i].win);

     return;
}
```

Here is the caller graph for this function:

void flniP_PC (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 601 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char                ls_label[255]="";
     int                    lcolor=1;
```

```
    unsigned char       lhide=0;       // TRUE if panel is hidden
    unsigned short int  i = 0;

    i = P_PC;

    coord.x=73; coord.y=37; coord.h=3; coord.w=35;
    p_window[i].coord = coord;
    p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
    box(p_window[i].win, 0, 0);

    // Creating subwindows within the window
    p_window[P_SUBPC].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
    wrefresh(p_window[i].win);

    //Creating and Setting-Up a new Panel
    p_panel[i].coord = coord;
    sprintf(ls_label, " Program COUNTER ");
    sprintf(p_panel[i].label, ls_label);
    p_panel[i].labelcolor = lcolor;
    p_panel[i].hide= lhide;

    p_panel[i].up=P_RF;
    p_panel[i].down=P_IR;
    p_panel[i].left=P_IDHEX;
    p_panel[i].right=P_ADDR2;
    p_panel[i].cursorcomp=0;
    p_panel[i].panel = new_panel(p_window[i].win);

    return;
}
```

Here is the caller graph for this function:

void flniP_RF (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 547 of file disside.h.

Referenced by fSetIDE().

```
{
    s_coord                 coord = {0,0,0,0};
    char                ls_label[255]="";
    int                     lcolor=1;
    unsigned char       lhide=0;       // TRUE if panel is hidden
    unsigned short int  i = 0;
    int                     x=0;
    int                     y=0;


    i = P_RF;


    coord.x=73; coord.y=3; coord.h=34; coord.w=35;
    p_window[i].coord = coord;
    p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
    box(p_window[i].win, 0, 0);

    // Creating subwindows within the window
    p_window[P_SUBRF].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
    wrefresh(p_window[i].win);

    //Creating and Setting-Up a new Panel
    p_panel[i].coord = coord;
    sprintf(ls_label, " REGISTER FILE PANEL ");
    sprintf(p_panel[i].label, ls_label);
    p_panel[i].labelcolor = lcolor;
    p_panel[i].hide= lhide;
```

```
     p_panel[i].up=P_TOPBAR;
     p_panel[i].down=P_PC;
     p_panel[i].left=P_ADDRRF;
     p_panel[i].right=P_ADDR2;
     p_panel[i].cursorcomp=0;
     p_panel[i].panel = new_panel(p_window[i].win);

//   getyx(p_window[i].win,y,x);
//   wmove(p_window[i].win,2,2);




     /*
     //Highlighting current cursor
               fHighlight_line(p_window[*pActiveP].
                    g_wins[gactiveP].win,gCinstr+1,COLOR_PAIR(3)|A_BOLD);   // Main/present
panel
               fHighlight_line(g_wins[P_ADDR1].win,gCinstr+1,COLOR_PAIR(3)|A_BOLD);    //
ADDR1 Panel
               fHighlight_line(g_wins[P_IDHEX].win,gCinstr+1,COLOR_PAIR(3)|A_BOLD);    //
IDHEX Panel
               */

     return;
}
```

Here is the caller graph for this function:

void flniP_STATUS (s_window * *p_window*, s_panel * *p_panel*)

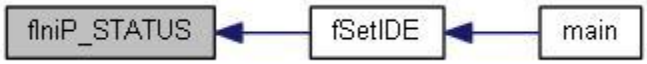
Definition at line 754 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char               ls_label[255]="";
     int                    lcolor=1;
     unsigned char      lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_STATUS;

     coord.x=0; coord.y=43; coord.h=11; coord.w=149;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);
     wrefresh(p_window[i].win);

     // Creating subwindows within the window
     p_window[P_SUBSTATUS].win=derwin(p_window[i].win, coord.h-2, coord.w-4, 1, 1);
     // Enabling Scroll in the subwindow
     scrollok(p_window[P_SUBSTATUS].win,TRUE);
     wbkgd(p_window[P_SUBSTATUS].win,COLOR_PAIR(2));

     p_panel[i].coord = coord;
     sprintf(ls_label, " STATUS ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_MAIN1;
     p_panel[i].down=P_BOTTBAR;
     p_panel[i].left=P_NULL;
     p_panel[i].right=P_NULL;
```

```
     p_panel[i].cursorcomp=0;
     //p_panel[i].panel = new_panel(p_window[i].win);

     return;
}
```

Here is the caller graph for this function:



void flniP_TopBar (s_window * *p_window*, s_panel * *p_panel*)

Definition at line 261 of file disside.h.

Referenced by fSetIDE().

```
{
     s_coord                coord = {0,0,0,0};
     char               ls label[255]="";
     int                    lcolor=1;
     unsigned char      lhide=0;       // TRUE if panel is hidden
     unsigned short int  i = 0;

     i = P_TOPBAR;

     coord.x=0; coord.y=0; coord.h=3; coord.w=149;
     p_window[i].coord = coord;
     p_window[i].win = newwin(coord.h, coord.w, coord.y, coord.x);
     box(p_window[i].win, 0, 0);
     wrefresh(p_window[i].win);

     // Creating subwindows within the window
     p_window[P_SUBTOPBAR].win=derwin(p_window[i].win, coord.h-2, coord.w-2, 1, 1);
     //wbkgd(p_window[P_SUBTOPBAR].win,COLOR_PAIR(3));

     //Writing welcome message on panel
     mvwaddnstr(p_window[P_SUBTOPBAR].win, 0,0," Welcome to the ERA
disassembler/simulator\n",-1);
     wrefresh(p_window[P_SUBTOPBAR].win);
     wrefresh(p_window[i].win);

     //Creating and Setting-Up a new Panel
     p_panel[i].coord = coord;
     sprintf(ls label, " TOP BAR PANEL ");
     sprintf(p_panel[i].label, ls_label);
     p_panel[i].labelcolor = lcolor;
     p_panel[i].hide= lhide;

     p_panel[i].up=P_NULL;
     p_panel[i].down=P_MAIN1;
     p_panel[i].left=P_NULL;
     p_panel[i].right=P_NULL;
     p panel[i].cursorcomp=0;
     p_panel[i].panel = new_panel(p_window[i].win);

     return;
}
```

Here is the caller graph for this function:

int flnitPanels (s_panel * *p_panel*)

Definition at line 170 of file disside.h.

Referenced by fSetIDE().

```
{
     unsigned int i=0;

     for (i=0; i<NUMPANELS; i++)
     {
          p_panel[i].coord.h=0;
          p_panel[i].coord.w=0;
          p_panel[i].coord.x=0;
          p_panel[i].coord.y=0;
          p_panel[i].cursorcomp=0;
          p_panel[i].down=P_NULL;
          p_panel[i].up=P_NULL;
          p_panel[i].left=P_NULL;
          p_panel[i].right=P_NULL;
          p_panel[i].hide=0;
          strcpy(p_panel[i].label," ");
          p_panel[i].labelcolor=0;
          p_panel[i].panel=NULL;
     }
     return 0;
}
```

Here is the caller graph for this function:

int flnitWindows (s_window * *p_window*)

Definition at line 154 of file disside.h.

Referenced by fSetIDE().

```
{
     unsigned int i=0;

     for (i=0; i<NUMWINDOWS; i++)
     {
          p_window[i].coord.h=0;
          p_window[i].coord.w=0;
          p_window[i].coord.x=0;
          p_window[i].coord.y=0;
          p_window[i].cursor=0;
          p_window[i].win=NULL;
     }
     return 0;
}
```

Here is the caller graph for this function:

int fLoadPanels (s_panel * *p_panels*, s_window * *p_window*, s_item * *pCode*, unsigned * *pRF*, s_item * *pIRAM*, unsigned *pPC*)

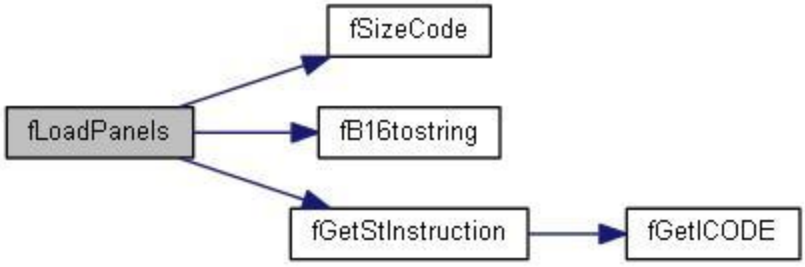
Definition at line 1104 of file disside.h.

Referenced by main().

```
{
     unsigned int   lSizeCode = 0;
     unsigned int   i=0;
     char           lstInstr[64]="";
     char           lsIcode[20]="";
     s_coord                  coord = {0,0,0,0};
```

```
    //Obtain number of Elements
    lSizeCode = fSizeCode(pCode);

    waddstr(p_window[P_SUBSTATUS].win, " Loading Panels ....   ");
    wrefresh(p_window[P_SUBSTATUS].win);

    //Browsing the list
    for (i=0; i<lSizeCode; i++)
    {
         //Loading coordinates of the MAIN 1 window
         coord=p_window[P_MAIN1].coord;

         // Printing binary in MAIN 1
         fB16tostring(pCode[i].HL,lstInstr);
         //mvwaddnstr(p_window[P_PMAIN1].win, i, 0,lstInstr,-1);
         mvwaddnstr(p_window[P_SUBMAIN1].win, i, 0,lstInstr,-1);
         fB16tostring(pCode[i].LL,lstInstr);
         mvwaddnstr(p_window[P_SUBMAIN1].win, i, 17,lstInstr,-1);
         //mvwaddnstr(p_window[P_PMAIN1].win, i, 18, lstInstr,-1);
         prefresh(p_window[P_SUBMAIN1].win,0,0,coord.y+1,coord.x+2,coord.h+1,coord.w+3);

         strcpy(lstInstr, "");

         //Loading coordinates of the IDHEX window
         coord=p_window[P_IDHEX].coord;

         // Printing Instruction
         if (pCode[i].type==T_INSTRUCTIONS)
         {
              fGetStInstruction(pCode[i].InstrA, lstInstr);
              mvwaddnstr(p window[P SUBIDHEX].win, i, 0, lstInstr,-1);

              fGetStInstruction(pCode[i].InstrB, lstInstr);
              mvwaddnstr(p_window[P_SUBIDHEX].win, i, 13, lstInstr,-1);
         }
         else
         {
              sprintf(lstInstr, "%08X", pCode[i].HLLL);
              //mvwaddnstr(p window[P SUBIDHEX].win, i, 8, lstInstr,-1);
              //mvwaddnstr(p_window[P_SUBIDHEX].win, i, 8, "xxx",-1);
              int k=0;
              k=mvwaddnstr(p_window[P_SUBIDHEX].win, i, 8, "xxx",-1);
              if (k!=OK)
              {
                   addstr("Unable to create subpad");
                   refresh();
                   endwin();
                   return -1; // Testing
              }
         }

         //Refreshing SUBIDHEX window
         prefresh(p_window[P_SUBIDHEX].win,0,0,coord.y+1,coord.x+1,coord.h+1,coord.w);
         prefresh(p_window[P_SUBMAIN1].win,0,0,coord.y+1,coord.x+2,coord.h+1,coord.w+3);
    }

    waddstr(p_window[P_SUBSTATUS].win, "Panels Loaded\n");
    wrefresh(p_window[P_SUBSTATUS].win);

    return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

short fmovePanel (int *pkey*, short * *pActiveP*, s_panel * *p_panels*, s_window * *p_window*)

Definition at line 920 of file disside.h.

Referenced by main().

```
{
     short int ltempP=*pActiveP;   // Backing up Active panel parameter
     char      lstMessage[255]=""; // Message for the status window (sprintf)

     switch(pkey)
     {
          case CTRL_LEFT_KEY: // CTRL+LEFT
               if (p_panels[ltempP].left != P_NULL)
               {
                    ltempP=*pActiveP;

                    // Disable highlighting of current Panel
                    fHighlightPanel(ltempP, 0, p_window);

                    // Switching to the left Panel
                    *pActiveP=p_panels[ltempP].left;

                    // Remembering the way in in orther to return back (right in this case)
                    p_panels[*pActiveP].right=ltempP;

                    // Enable highlighting of current Panel
                    fHighlightPanel(*pActiveP, 1, p_window);
               }
               else
               {
                    beep();
               }
               break;
          case CTRL RIGHT KEY: // CTRL+RIGHT KEY
               if (p_panels[ltempP].right != P_NULL)
               {
                    ltempP=*pActiveP;

                    // Disable highlighting of current Panel
                    fHighlightPanel(ltempP, 0, p_window);

                    // Switching to the right Panel
                    *pActiveP=p panels[ltempP].right;

                    // Remembering the way in in orther to return back (left in this case)
                    p_panels[*pActiveP].left=ltempP;

                    // Enable highlighting of current Panel
                    fHighlightPanel(*pActiveP, 1, p_window);
```

```
            }
            else
            {
                 beep();
            }
            break;
       case CTRL_UP_KEY: // CTRL+UP KEY
            if (p_panels[ltempP].up != P_NULL)
            {
                 ltempP=*pActiveP;

                 // Disable highlighting of current Panel
                 fHighlightPanel(ltempP, 0, p_window);

                 // Switching to the upper Panel
                 *pActiveP=p_panels[ltempP].up;

                 // Remembering the way in in orther to return back (down in this case)
                 p_panels[*pActiveP].down=ltempP;

                 // Enable highlighting of current Panel
                 fHighlightPanel(*pActiveP, 1, p_window);
            }
            else
            {
                 beep();
            }
            break;
       case CTRL_DOWN_KEY: // CTRL+DOWN KEY
            if (p_panels[ltempP].down != P_NULL)
            {
                 ltempP=*pActiveP;

                 // Disable highlighting of current Panel
                 //fHighlightPanel(p_window[ltempP].win, 0);
                 fHighlightPanel(ltempP, 0, p_window);

                 // Switching to the down Panel
                 *pActiveP=p_panels[ltempP].down;

                 // Remembering the way in in orther to return back (up in this case)
                 p_panels[*pActiveP].up=ltempP;

                 // Enable highlighting of current Panel
                 //fHighlightPanel(p_window[*pActiveP].win, 1);
                 fHighlightPanel(*pActiveP, 1, p_window);
            }
            else
            {
                 beep();
            }
            break;

       default:
            break;
   }

   // Updating the status subwindow with the new move
   if (*pActiveP!= ltempP)  // Preventing from showing the message when moving to the same panel
   {
       sprintf(lstMessage," Moving to the%s \n",p panels[*pActiveP].label);
       waddstr(p_window[P_SUBSTATUS].win, lstMessage);
       wrefresh(p_window[P_SUBSTATUS].win);
   }

       //   ltempP=*pActiveP;
            // Disable highlighting of current Panel
       //   fHighlightPanel(p_window[ltempP].win, 0);

            // Highlighting MAIN panel
       //   *pActiveP=P_IDHEX;
```

```
          //   fHighlightPanel(p_window[*pActiveP].win, 1);


          /*
               //Highlighting current cursor
               fHighlight_line(p_window[*pActiveP].
                    g_wins[gactiveP].win,gCinstr+1,COLOR_PAIR(3)|A_BOLD);   // Main/present
panel
               fHighlight_line(g_wins[P_ADDR1].win,gCinstr+1,COLOR_PAIR(3)|A_BOLD);    //
ADDR1 Panel
               fHighlight_line(g_wins[P_IDHEX].win,gCinstr+1,COLOR_PAIR(3)|A_BOLD);    //
IDHEX Panel
          */

return *pActiveP;
}
```

Here is the call graph for this function:

Here is the caller graph for this function:

int fSetIDE (s_panel * *p_panel*, s_window * *p_window*, unsigned short int *p_numInstr*)

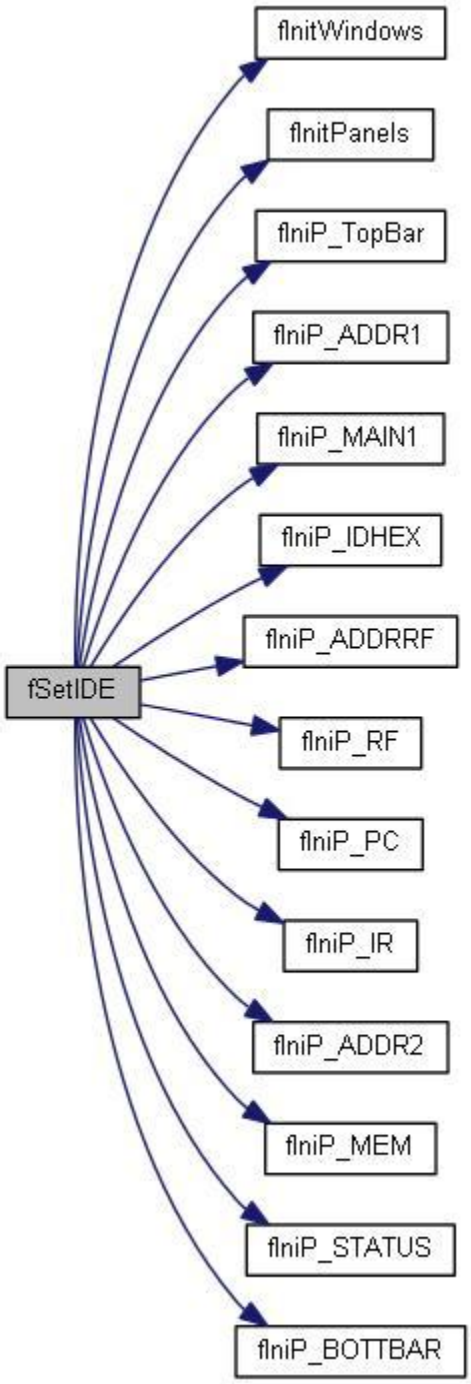
Definition at line 830 of file disside.h.

Referenced by main().

```
{
     unsigned short int  i = 0;
     s_coord                  coord = {0,0,0,0};
     char                ls_label[255]="";
     int                      lcolor=1;
     unsigned char       lhide=0;       // TRUE if panel is hidden
     //s_winborder              lwinb={'â”,', 'â”,', 'â”€', 'â”€', 'â”Œ', 'â”□ ', 'â””', 'â”˜'};

     fInitWindows(p_window);
     fInitPanels(p_panel);

     // Initializing panels
     fIniP_TopBar(p_window, p_panel);
     fIniP_ADDR1(p_window, p_panel, p_numInstr);
     fIniP_MAIN1(p_window, p_panel);
     fIniP_IDHEX(p_window, p_panel);

     fIniP_ADDRRF(p_window, p_panel);
     fIniP_RF(p_window, p_panel);
     fIniP_PC(p_window, p_panel);
     fIniP_IR(p_window, p_panel);
     fIniP_ADDR2(p_window, p_panel);
     fIniP_MEM(p_window, p_panel);

     fIniP_STATUS(p_window, p_panel);
     fIniP_BOTTBAR(p_window, p_panel);

     //Writing PC and IR
     //coord.x=73; coord.y=37; coord.h=3; coord.w=35;
//   mvaddnstr(38,69,"PC",-1);
//   mvaddnstr(41,69,"IR",-1);

     waddstr(p_window[P_SUBSTATUS].win, " All panels Initialized\n");
     wrefresh(p_window[P_SUBSTATUS].win);
```

```
    // Refreshing all panels
    /*
    refresh();
    update_panels();
    doupdate();
    */


    return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

int fSetTerm ()

It sets a long buffer and Size of the terminal (maximizing it).

Author:

Victor Castano

Date:

08\03\2012

This function uses windows specific libreries/functions. Hence, the code is only portable with minor modifications

Returns:

int 0 for correct execution

Definition at line 228 of file disside.h.

Referenced by main().

```
{
     HWND hWnd;
     HANDLE hOut;
    CONSOLE_SCREEN_BUFFER_INFO SBInfo;
     COORD NewSBSize;
     SMALL_RECT DisplayArea = {0, 0, 0, 0};
     char lsNamewnd[255] ="ERA Disassambler v0.1 June-2011 - Victor Castano";

     hOut = GetStdHandle(STD_OUTPUT_HANDLE);
     GetConsoleScreenBufferInfo(hOut, &SBInfo);

     SetConsoleTitle(lsNamewnd);
     hWnd = FindWindow(NULL, lsNamewnd);

     NewSBSize = GetLargestConsoleWindowSize(hOut);
     // The following two lines will overwrite the detected
     // maximum resolution for the console. Comment if appropiate
     NewSBSize.X=192;
     NewSBSize.Y=58;
     DisplayArea.Right = NewSBSize.X-1;
    DisplayArea.Bottom = NewSBSize.Y-1;

     SetConsoleScreenBufferSize(hOut, NewSBSize);
     SetConsoleWindowInfo(hOut, TRUE, &DisplayArea);
     ShowWindow(hWnd, SW_MAXIMIZE);
     //printf("\n right = %d, bottom = %d", DisplayArea.Right, DisplayArea.Bottom);
     //getchar();

     return 0;
}
```

Here is the caller graph for this function:

void fUnHighlight_line (WINDOW * *pwin*, int *pline*)

Definition at line 199 of file disside.h.

Referenced by fUpdateCursors_Main().

```
{
     //wattron(p_win, COLOR_PAIR(0));
     mvwchgat(pwin, pline, 1, pwin->_maxx-2,  COLOR_PAIR(0), 0, NULL);
     wrefresh(pwin);
}
```

Here is the caller graph for this function:

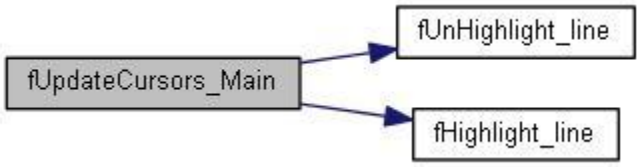
int fUpdateCursors_Main (s_window * *p_window*, int *p_iLastIntr*, int *p_iNewInstr*, unsigned int *pbackgcolor*)

Definition at line 207 of file disside.h.

Referenced by main().

```
{
     fUnHighlight_line(p_window[P_MAIN1].win,p_iLastIntr+1);     // Main Panel
     fUnHighlight_line(p_window[P_ADDR1].win,p_iLastIntr+1);     // ADDR1 Panel
     fUnHighlight_line(p_window[P_IDHEX].win,p_iLastIntr+1);     // IDHEX Panel

     fHighlight line(p window[P MAIN1].win,p iNewInstr+1,pbackgcolor);     // Main Panel
     fHighlight_line(p_window[P_ADDR1].win,p_iNewInstr+1,pbackgcolor);     // ADDR1 Panel
     fHighlight_line(p_window[P_IDHEX].win,p_iNewInstr+1,pbackgcolor);     // IDHEX Panel
     return 0;
}
```

Here is the call graph for this function:



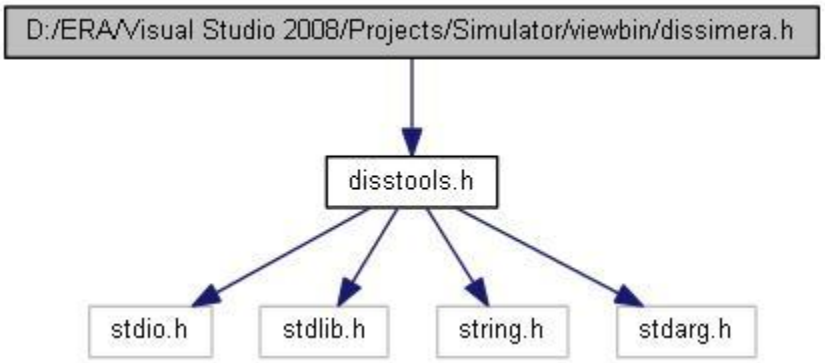
Here is the caller graph for this function:

D:/ERA/Visual Studio 2008/Projects/Simulator/viewbin/dissimera.h File Reference

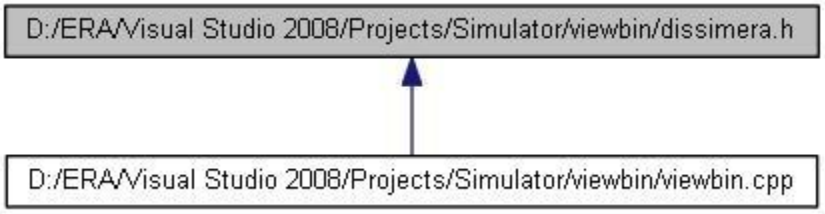
General Header file for Dissimera.

```
#include "disstools.h"
```

Include dependency graph for dissimera.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **s_instruction**
- *Structure that represents the instruction*. struct **s_item**
- *Structure that represents either a data or a instruction word*. struct **s_exec**

Structure that represents the state of the execution. Defines

- #define **MAX_LIST_SIZE** 2000
  Size of the array of 32bit instructions.
- #define **RAM_ELEMENTS** 2000
  Size of the RAM = 2000x32bits=64000bits=7.8125 kilobytes.
- #define **STBUFFER** 1000
  Size of the terminal buffer.
- #define **T_INSTRUCTIONS** 1
  Type of load: T_INSTRUCTIONS 1 = 2x16 Instructions.
- #define **T_32DATA** 3
  Type of load: T_32DATA 3 = 32 bit Data.
- #define **T_SPECIAL** 8
  Type of load: T_SPECIAL 8 = Configuration load (such position 0)
- #define **F_STOP** 0

Stop Instruction FormatCode = 00 InstructionCode = 0000.

- #define **F_NOP** 3
  Nop Instruction.
- #define **I_NOPSTOP** 0
  NOP/STOP instruction.
- #define **I_LD** 1
  LD instruction.
- #define **I_LDA** 2
  LDA instruction.
- #define **I_ST** 3
  ST instruction.
- #define **I_MOV** 4
  MOV instruction.
- #define **I_ADD** 5
  ADD instruction.
- #define **I_SUB** 6
  SUB instruction.
- #define **I_ASR** 7
  ASR instruction.
- #define **I_ASL** 8
  ASL instruction.
- #define **I_OR** 9
  OR instruction.
- #define **I_AND** 10
  AND instruction.
- #define **I_XOR** 11
  XOR instruction.
- #define **I_LSL** 12
  LSL instruction.
- #define **I_LSR** 13
  LSR instruction.
- #define **I_CND** 14
  CND instruction.
- #define **I_CBR** 15
  CBR instruction.
- #define **LSDEBUB** 0
  LSDEBUB tobedone.
- #define **LNDEBUG** 1
  LSDEBUB tobedone.
- #define **LNORMAL** 2
  LSDEBUB tobedone.

Functions

- int **fInitRegisterFile** (unsigned int *pRF)
  It initializes the register file structure.
- int **fInitExec** (**s_exec** *p_exec)

It initializes Execution structure.

- int **fInitRAM** (unsigned int pNumelem, unsigned int *pRAM)
  It initializes RAM structure.
- void **fInitItem** (**s_item** *item)
  It initializes an Item structure.
- void **fIniItemList** (int pNumelem, **s_item** *pItemlist)
  It initializes the Code/Data list specified whose number of elements is pNumelem.
- int **fGetICODE** (char codenum, char *sIcode)
  It returns an output parameter sIcode with the ASCII equivalent of the instruction code.
- int **fGetStInstruction** (**s_instruction** p_instr, char *pstInstr)
  *It returns an output parameter *pstInstr with the ASCII equivalent of the instruction.*
- void **fset_type** (**s_item** *item, char type)
  It sets the type of the item/load.
- short int **SwapTwoBytes** (short int w)
  It converts two bytes from Little Endian to Big Endian.
- int **SwapFourBytes** (int dw)
  It converts four bytes from Little Endian to Big Endian.
- void **fB16tostring** (unsigned int n, char *pschar)
  It prints a 16 digit binary integer n as a binary in text.
- void **fB32tostring** (unsigned int n, char *pschar)
  It prints a 32 digit binary integer n as a binary in text.
- unsigned int **fASL** (unsigned int pnum)
  ERA arithmetic shift left operation (keeping the sign bit)
- unsigned int **fASR** (unsigned int pnum)
- **s_item parseItem** (unsigned int numline, unsigned int instr32)
  It will return an structure of the 2x16bit instructions filled.
- int **fParseFile** (char *iniFileName, **s_item** *pCode)
  It parses the bin file that results of Eugene's assembler/preparator.
- int **fSizeCode** (**s_item** *pCode)
- int **fshowItem** (**s_item** *item)
  fuction that shows an item on the screen using fptraza.
- int **fshowInstruction** (**s_instruction** *pInstr)
  It shows the instruction included as an input parameter.
- int **fParseCode** (**s_item** *pCode, unsigned *pRF, **s_item** *plRAM, unsigned *pPC)
  It browses the Code list and discriminates between data and instructions.

Detailed Description

General Header file for Dissimera.

```
\author        Victor Castano
\version  0.1a
```

\date 08\03\2012 It contains a collection of variables and functions related to the simulator

Warning:

Definition in file **dissimera.h**.

Data Structure Documentation

struct s_instruction

Structure that represents the instruction.

Definition at line 159 of file dissimera.h.

Data Fields:

unsigned char	f_code	Format Code (2bits)
unsigned char	i_code	Instruction Code (4bits)
unsigned char	Op1	First operand (5bits)
unsigned char	Op2	Second operand (5bits)

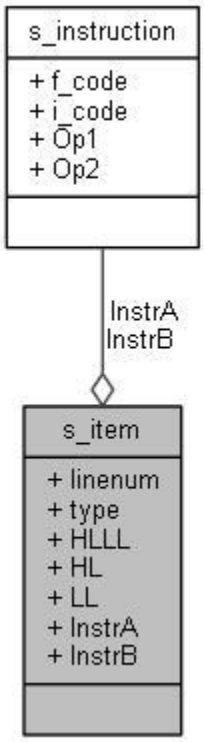
struct s_item

Structure that represents either a data or a instruction word.

Position 0 of the array of s_items is deliberately left empty for future configuration purposes. It seems that 5+4+4=13 elements would be more than enough for future configuration. For instance: Number of elements in the array. I know, I know. Redundant data. But It will be easier to handle. If needed I will swap the array for a dynamic list (at the moment this is just to test instructions on EURRICA.

Definition at line 175 of file dissimera.h.

Collaboration diagram for s_item:



Data Fields:

unsigned short int	HL	Higher load (16) of the 32 bits.

unsigned int	HLLL	32 bit instruction/data
s_instruction	InstrA	
s_instruction	InstrB	
unsigned short int	linenum	the x coordinate
unsigned short int	LL	Lower load (16) of the 32 bits.
unsigned char	type	Type of load: 1=2x16 Instructions, 2=Instruction+Data , 3=32 bit Data, 8=Configuration load (such as position 0) By default every load will be treated as Data

struct s_exec

Structure that represents the state of the execution.

In the future it will include timing and stats.

Definition at line 193 of file dissimera.h.

Data Fields:

unsigned int	nInstrEx	Number of Instructions Executed.
unsigned int	uiLastInstr	Last Instruction Executed.

Define Documentation

#define F_NOP 3

Nop Instruction.

Definition at line 132 of file dissimera.h.
Referenced by fGetStInstruction().

#define F_STOP 0

Stop Instruction FormatCode = 00 InstructionCode = 0000.

Definition at line 131 of file dissimera.h.
Referenced by fGetStInstruction().

#define I_ADD 5

ADD instruction.

Definition at line 139 of file dissimera.h.

#define I_AND 10

AND instruction.

Definition at line 144 of file dissimera.h.

#define I_ASL 8

ASL instruction.

Definition at line 142 of file dissimera.h.

#define I_ASR 7

ASR instruction.

Definition at line 141 of file dissimera.h.

#define I_CBR 15

CBR instruction.

Definition at line 149 of file dissimera.h.

#define I_CND 14

CND instruction.

Definition at line 148 of file dissimera.h.

#define I_LD 1

LD instruction.

Definition at line 135 of file dissimera.h.

#define I_LDA 2

LDA instruction.

Definition at line 136 of file dissimera.h.

#define I_LSL 12

LSL instruction.

Definition at line 146 of file dissimera.h.

#define I_LSR 13

LSR instruction.

Definition at line 147 of file dissimera.h.

#define I_MOV 4

MOV instruction.

Definition at line 138 of file dissimera.h.

#define I_NOPSTOP 0

NOP/STOP instruction.

Definition at line 134 of file dissimera.h.

#define I_OR 9

OR instruction.

Definition at line 143 of file dissimera.h.

#define I_ST 3

ST instruction.

Definition at line 137 of file dissimera.h.

#define I_SUB 6

SUB instruction.

Definition at line 140 of file dissimera.h.

#define I_XOR 11

XOR instruction.

Definition at line 145 of file dissimera.h.

#define LNDEBUG 1

LSDEBUB tobedone.

Definition at line 152 of file dissimera.h.

#define LNORMAL 2

LSDEBUB tobedone.

Definition at line 153 of file dissimera.h.

#define LSDEBUB 0

LSDEBUB tobedone.

Definition at line 151 of file dissimera.h.

#define MAX_LIST_SIZE 2000

Size of the array of 32bit instructions.

Definition at line 121 of file dissimera.h.
Referenced by main().

#define RAM_ELEMENTS 2000

Size of the RAM = 2000x32bits=64000bits=7.8125 kilobytes.

Definition at line 122 of file dissimera.h.

#define STBUFFER 1000

Size of the terminal buffer.

Definition at line 123 of file dissimera.h.

#define T_32DATA 3

Type of load: T_32DATA 3 = 32 bit Data.

Definition at line 128 of file dissimera.h.
Referenced by fInitItem().

#define T_INSTRUCTIONS 1

Type of load: T_INSTRUCTIONS 1 = 2x16 Instructions.

Definition at line 126 of file dissimera.h.
Referenced by fLoadPanels(), and fParseCode().

#define T_SPECIAL 8

Type of load: T_SPECIAL 8 = Configuration load (such position 0)

Definition at line 129 of file dissimera.h.
Referenced by fParseFile().

Function Documentation

unsigned int fASL (unsigned int *pnum*)

ERA arithmetic shift left operation (keeping the sign bit)

Author:
Victor Castano

Date:
08\03\2012

It shifts the content of register Ri arithmetically one bit to the left and store the result in Ri.

Memory state is not considered in the instruction, and the memory state does not change.

Both operands can refer to the same register.

Suggested assembly statement for the ASL instruction: Rj <<= Ri

Additional assembly directives specifying the current instruction format: .format 8 or .format 16 or .format 32

Arithmetic shift means that the sign bit does not participate in the operation but remains on its usual place.

The leftmost bit of the operand is always lost.

The rightmost bit of the operand gets the value of 0.

The contents of the Ri register does not change.

The effect of the ASL instruction for format 16 is shown below.

The operation for formats 8 and 32 is performed in the similar way.

Parameters:

pnum	is contents of the register Ri that should be arithmeticaly shifted to the left

Returns:
void

Definition at line 537 of file dissimera.h.

```
{
     unsigned int i=0;
     unsigned int j=0;

     // If bit in position 0 (the most left position) is 1 ... (negative number)
     // preparing mask to keep the sign
     if(pnum&(0x80000000>>i))
     {
          j=0x80000000;
     }
     else
     {
          j=0x00000000;
     }

     i=pnum & (0X3FFFFFFF);   // selecting bits 14-0
     i=i<<1;                       // Shifting them one to the left
     i=i|j;                        // Or with the mask (keeping the sign if needed)

     return i;
```

```
}
```

unsigned int fASR (unsigned int *pnum*)

Definition at line 561 of file dissimera.h.

```
{
     unsigned int i=0;
     unsigned int j=0;

     // If bit in position 0 (the most left position) is 1 ... (negative number)
     // preparing mask to keep the sign
     if(pnum&(0x80000000>>i))
     {
          j=0xC0000000;
     }
     else
     {
          j=0x40000000;
     }

     // To be
finisheddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddd
     //i=pnum & (0X3FFFFFFF);
     //i=i<<1;
     //i=i|j;

//   colorprintI32(i);
//  fptraza(gflog, 1,"  i = %d\n",i);

     return i;
}
```

void fB16tostring (unsigned int *n*, char * *pschar*)

It prints a 16 digit binary integer n as a binary in text.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

n	is the integer with the number
**pschar*	is the string with the binary number to be returned

Returns:

void

Definition at line 467 of file dissimera.h.

Referenced by fLoadPanels().

```
{
    unsigned int    i=0;
     char           lsnum[20]="";
    for(i = 0; i<16; i++)
     {
        if(n&(0x8000>>i))
          {
               strcat(lsnum, "1");
          }
          else
          {
               strcat(lsnum, "0");
```

```
        }

        //if(i == 1) fptraza(gflog, 1," "); /*put a space between bytes*/
        //if(i == 5) fptraza(gflog, 1," "); /*put a space between bytes*/
      //if(i == 10) fptraza(gflog, 1," "); /*put a space between bytes*/
    }
     strcpy(pschar, lsnum);
}
```

Here is the caller graph for this function:

void fB32tostring (unsigned int *n*, char * *pschar*)

It prints a 32 digit binary integer n as a binary in text.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

n	is the integer with the number
**pschar*	is the string with the binary number to be returned

Returns:

void

Definition at line 497 of file dissimera.h.

Referenced by fBootHardware().

```
{
    unsigned int    i=0;
     char           lsnum[40]="";

    for(i = 0; i<32; i++) {
        if(n&(0x80000000>>i)) strcat(lsnum, "1"); else strcat(lsnum, "0");
          //if(i == 1) fptraza(gflog, 1," "); /*put a space between bytes*/
          //if(i == 5) fptraza(gflog, 1," "); /*put a space between bytes*/
        //if(i == 10) fptraza(gflog, 1," "); /*put a space between bytes*/
          //if(i == 15) fptraza(gflog, 1,"      "); /*put a space between bytes*/
          //if(i == 17) fptraza(gflog, 1," "); /*put a space between bytes*/
          //if(i == 21) fptraza(gflog, 1," "); /*put a space between bytes*/
          //if(i == 26) fptraza(gflog, 1," "); /*put a space between bytes*/
    }

     strcpy(pschar, lsnum);
}
```

Here is the caller graph for this function:

int fGetICODE (char *codenum*, char * *sIcode*)

It returns an output parameter sIcode with the ASCII equivalent of the instruction code.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

codenum	is the number of elements
**sIcode*	is the output parameter with the ASCII equivalent of the instruction code

Returns:

int 0 for correct execution

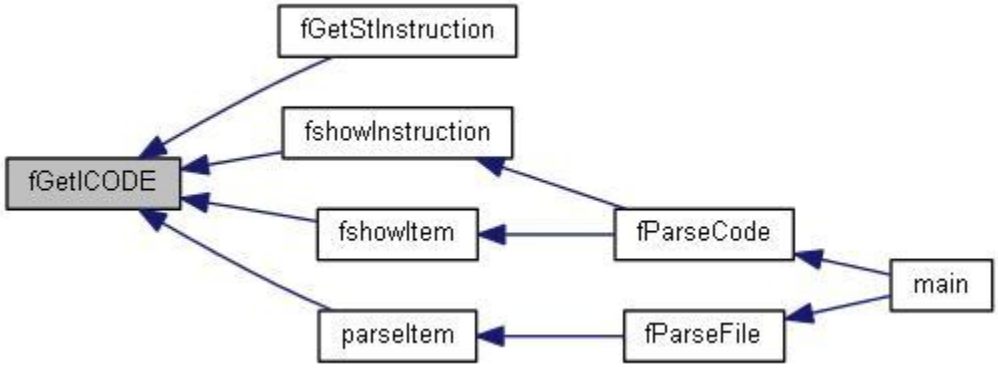
Definition at line 307 of file dissimera.h.

Referenced by fGetStInstruction(), fshowInstruction(), fshowItem(), and parseItem().

```
{
     strcpy(sIcode," ");
     switch (codenum)
     {
     case 0:
          strcpy(sIcode, "NOP-STOP"); //The NOP instruction performs no actions, except moving
the PC register to the next instruction.
          break;
     case 1:
          strcpy(sIcode, "LD");         //The LD instruction copies the value of a 32-bit memory
word pointed to by Ri register, to the Rj register.
          break;
     case 2:
          strcpy(sIcode, "LDA");
          break;
     case 3:
          strcpy(sIcode, "ST");
          break;
     case 4:
          strcpy(sIcode, "MOV");
          break;
     case 5:
          strcpy(sIcode, "ADD");
          break;
     case 6:
          strcpy(sIcode, "SUB");
          break;
     case 7:
          strcpy(sIcode, "ASL");
          break;
     case 8:
          strcpy(sIcode, "ASR");
          break;
     case 9:
          strcpy(sIcode, "OR");
          break;
     case 10:
          strcpy(sIcode, "AND");
          break;
     case 11:
          strcpy(sIcode, "XOR");
          break;
     case 12:
          strcpy(sIcode, "LSL");
          break;
     case 13:
          strcpy(sIcode, "LSR");
          break;
     case 14:
          strcpy(sIcode, "CND");
          break;
     case 15:
          strcpy(sIcode, "CBR");
          break;
```

```
     default:
          return 1; //This would never happen;
     }

     return 0;
}
```

Here is the caller graph for this function:



int fGetStInstruction (s_instruction *p_instr*, char * *pstInstr*)

It returns an output parameter *pstInstr with the ASCII equivalent of the instruction.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

p_instr	is structure with the instruction and the operands
**pstInstr*	is the output parameter with the ASCII equivalent of the instruction code and parameters

Returns:

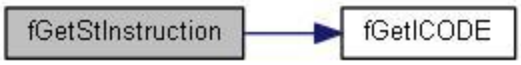
int 0 for correct execution

Definition at line 377 of file dissimera.h.

Referenced by fLoadPanels().

```
{
     char lstInstr[20]="";
     char lsIcode[20]="";

     if( p_instr.i_code == 0)
     {
          if (p_instr.f_code == F_NOP)sprintf(lstInstr, "NOP");
          if (p_instr.f_code == F_STOP) sprintf(lstInstr, "STOP");
     }
     else
     {
          fGetICODE(p_instr.i_code, lsIcode);
          sprintf(lstInstr, "%s R%d R%d", lsIcode, p_instr.Op1, p_instr.Op2);
     }

     strcpy(pstInstr, lstInstr);

     return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

void fInitItemList (int *pNumelem*, s_item * *pItemlist*)

It initializes the Code/Data list specified whose number of elements is pNumelem.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

pNumelem	is the number of elements
pItemlist	the list of elements

Returns:

void

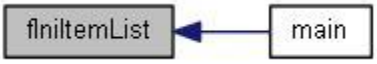
Definition at line 287 of file dissimera.h.

Referenced by main().

```
{
     int i=0;

     for (i=0; i<pNumelem; i++)
     {
          fInitItem(&pItemlist[i]);
     }

     return;
}
```

Here is the call graph for this function:

Here is the caller graph for this function:



int fInitExec (s_exec * *p_exec*)

It initializes Execution structure.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

*p_exec	is a reference to the execution structure

Returns:

int 0 for correct execution

Definition at line 226 of file dissimera.h.

Referenced by main().

```
{
     p_exec->nInstrEx = 0;
     p_exec->uiLastInstr = 0;

     return 0;
}
```

Here is the caller graph for this function:

void flnitItem (s_item * *item*)

It initializes an Item structure.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

*item	is a reference to the RAMs tructure

Returns:

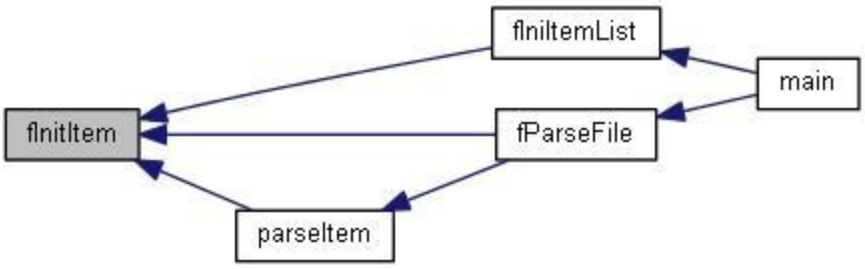
void

Definition at line 261 of file dissimera.h.

Referenced by flniItemList(), fParseFile(), and parseItem().

```
{
     item->linenum=0;
     item->type=T_32DATA;               // By default every load will be treated as a 32bit data
     item->HLLL=0;
     item->InstrA.f_code=0;
     item->InstrA.i_code=0;
     item->InstrA.Op1=0;
     item->InstrA.Op2=0;
     item->InstrB.f_code=0;
     item->InstrB.i_code=0;
     item->InstrB.Op1=0;
     item->InstrB.Op2=0;
     item->HL=0;
     item->LL=0;
     return;
}
```

Here is the caller graph for this function:



int fInitRAM (unsigned int *pNumelem*, unsigned int * *pRAM*)

It initializes RAM structure.

Author:
Victor Castano

Date:
08\03\2012

Parameters:

pNumelem	is the number of elements to be initialized
**pRAM*	is a reference to the RAM structure

Returns:
int 0 for correct execution

Definition at line 243 of file dissimera.h.

```
{
     unsigned int i=0;

     for (i=0; i<pNumelem; i++)
     {
          pRAM[i]=0;
     }
     return 0;
}
```

int fInitRegisterFile (unsigned int * *pRF*)

It initializes the register file structure.

Author:
Victor Castano

Date:
08\03\2012

Returns:
int 0 for correct execution

Definition at line 207 of file dissimera.h.

Referenced by main().

```
{
     unsigned char i=0;

     for (i=0; i<32; i++)
     {
```

```
          pRF[i]=0;
     }
     return 0;
}
```

Here is the caller graph for this function:

int fParseCode (s_item * *pCode*, unsigned * *pRF*, s_item * *pIRAM*, unsigned * *pPC*)

It browses the Code list and discriminates between data and instructions.

Author:

Victor Castano

Version:

0.1a

Date:

08\03\2012

Parameters:

**pCode*	
**pRF*	
**plRAM*	
**pPC*	

Returns:

int 0 for correct execution

fParseCode browses the Code list and discriminates between data and instructions It browses instructions starting in the address 0 and jumps from one to the next setting up the load to T_INSTRUCTIONS leaving the rest as DATA. Any data stored after the STOP instruction will not be browsed.

Definition at line 786 of file dissimera.h.

Referenced by main().

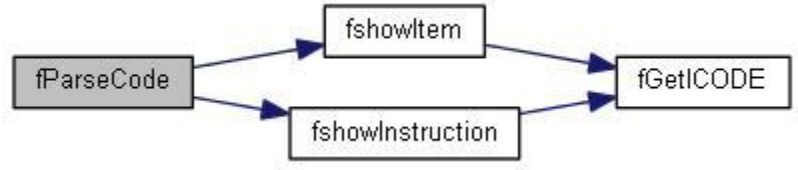
```
{
     //s_item            item;
     unsigned int        lsize=0;
     unsigned int        i=0;
     unsigned short int  lEOLOAD=0;                    // Variable that will represent which load
is active High (1) Load or Low (2) Load
     bool                lEOCODE=0;                     // Variable that will represent whether
the END OF CODE has been reached
     s_instruction       lInstr;                        // Local var that stores the instruction:
either High or Low


     //Obtain number of Elements
     lsize=pCode[0].linenum;
     fptraza(gflog, 1, "\n Size of File        = %d bytes", lsize*4);
     fptraza(gflog, 3, "\n Number of Elements  = %d \n", lsize);

     //Check if the Code list is empty
     if (lsize <=0)
     {
          return -1;
     }

     /*   Browse the list of Code (High and Low part) starting by the first element
```

```
          By default every item was set as T_32BITDATA, we are now browsing only instruction by
instruction
          in order to mark instructions from the data
     */

     i=*pPC+1; // Position in the code list is 1+ due to the first element being used for
configuration
     do
     {
          fshowItem(&pCode[i]);
          pCode[i].type=T_INSTRUCTIONS;      // The item is definitely an instruction load
          lEOLOAD=1;
          do //Browse firts the high and then the low part (specified by lEOLOAD)
          {
               if (lEOLOAD==1)
               {
                    lInstr=pCode[i].InstrA;
               }
               else
               {
                    lInstr=pCode[i].InstrB;
                    fptraza(gflog, 1, "\n");
               }

               fshowInstruction(&lInstr);

               switch (lInstr.i_code)
               {
               case 0:   //The # instruction performs no actions, except moving the PC register
to the next instruction.
                    switch (lInstr.f_code)
                    {
                    case 0:                              // x00b STOP instruction
                         fptraza(gflog, 1, " meaning STOP instruction ");
                         lEOCODE=true;            // End of CODE
                         break;
                    case 1:                              // x01b NOP instruction
                         //break;                 //Assuming that 01 and 11 for f_code are a NOP
instruction (check with Igor, Thomas and Eugene
                    case 3:                              // QUESTION: x11b According to the
Document Instruction Set this is not possible but it seems to be a NOP instruction
                         fptraza(gflog, 1, " meaning NOP instruction ");
                         break;
                    default:
                         fptraza(gflog, 1, "THIS IS NOT POSSIBLE - a 01??? in the format code?");
                    }

                    break;
               case 1:   // The LD instruction copies the value of a 32-bit memory word pointed
to by Ri into Rj
                    fptraza(gflog, 1, " meaning R%d:=*R%d ", lInstr.Op2, lInstr.Op1);
                    //Loading Rj with the vaule from the memory location
                    pRF[lInstr.Op2]=plRAM[lInstr.Op1].HLLL;
                    fptraza(gflog, 1, "   meaning  R%d:=%d ", lInstr.Op2,
plRAM[lInstr.Op1].HLLL);
                    break;
               case 2:   // The LDA instruction takes the value from the next 32-bit word and stores
the result into Rj
                         // constant stored in the Next 32bit location
                    fptraza(gflog, 1, " meaning R%d:=CONSTANT", lInstr.Op2);
                    fptraza(gflog, 1, "   meaning  R%d:=%d ", lInstr.Op2, pCode[i+1].HLLL);
                    pRF[lInstr.Op2]=pCode[i+1].HLLL;   //Loading Rj with the next 32bit word
                    fptraza(gflog, 1, "   meaning  R%d:=%d ", lInstr.Op2, pRF[lInstr.Op2]);
                    if (lEOLOAD ==1) lEOLOAD++;   // Jumping the load
                    i++; // Jumping to the following instruction (the one after the data)
                    break;
               case 3:   // The ST instruction copies the value of Ri to the memory by address
taken from Rj
                    fptraza(gflog, 1, " meaning *R%d:=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, "   meaning  MEM[%d]=%d ", pRF[lInstr.Op2],
pRF[lInstr.Op1]);
```

```
                    plRAM[pRF[lInstr.Op2]].HLLL = pRF[lInstr.Op1];         // MEM(Rj)=Ri
                    fptraza(gflog, 1, "   meaning  MEM[%d]=%d ", pRF[lInstr.Op2],
plRAM[pRF[lInstr.Op2]].HLLL);
                    break;
               case 4:   // The MOV instruction copies the value from Ri to the Rj
                         // Assuming that format code is fixed to 11
                    fptraza(gflog, 1, " meaning R%d:=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d:=%d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2]=pRF[lInstr.Op1];   // Rj=Ri
                    break;
               case 5:   // The ADD instruction denotes the two's complement arithmetic addition.
                         // The contents of Ri and Rj are arithmetically added, and the result
is put into Rj.
                         // Assuming that format code is fixed to 11
                    fptraza(gflog, 1, " meaning R%d+=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=R%d+R%d", lInstr.Op2,lInstr.Op2,
lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=%d+%d", lInstr.Op2,pRF[lInstr.Op2],
pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] + pRF[lInstr.Op2];        // Rj=Rj+Ri
                    break;
               case 6:   /*   The SUB instruction denotes the two's complement arithmetic
subtraction.
                              The content of Ri is subtracted from the contents of Rj, and the
result
                              is put into the register Rj
                         */
                    fptraza(gflog, 1, " meaning R%d-=R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=R%d-R%d", lInstr.Op2,lInstr.Op2,
lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d=%d-%d", lInstr.Op2,pRF[lInstr.Op2],
pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] - pRF[lInstr.Op2];        // Rj=Rj+Ri
                    break;
               case 7:   // The ASR instruction arithmetically shifts Ri one bit right, and puts
the result into Rj.
                    fptraza(gflog, 1, " meaning R%d >>= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d >>= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] >> 1;               // one bit right
shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d >>= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 8:   /*   The ASL instruction arithmetically shifts the contents of Ri one
bit left, and puts
                         the result into the register Rj. */
                    fptraza(gflog, 1, " meaning R%d <<= R%d", lInstr.Op2, lInstr.Op1);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    fptraza(gflog, 1, " meaning R%d <<= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] << 1;               // one bit left
shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d <<= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 9:   /*   The OR instruction applies logical addition ("OR") operator to every
pair
                              of bits taken from Ri and Rj, respectively, and puts the result into
Rj. */
                    fptraza(gflog, 1, " meaning R%d |= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d |= %d", lInstr.Op2,  pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] | pRF[lInstr.Op1];
                    fptraza(gflog, 1, " meaning R%d |= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 10: /*    The AND instruction applies logical multiplicative ("AND") operator
to
                              every pair of bits taken from Ri and Rj, respectively, and puts the
                              result into Rj */
                    fptraza(gflog, 1, " meaning R%d &= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d &= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] & pRF[lInstr.Op1];
                    fptraza(gflog, 1, " meaning R%d &= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
```

```
               case 11: /*    The XOR instruction applies logical exclusive OR ("XOR") operator
to every
                              pair of bits taken from Ri and Rj, respectively, and puts the result
into Rj. */
                    fptraza(gflog, 1, " meaning R%d ^= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d ^= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = ( pRF[lInstr.Op2] || pRF[lInstr.Op1] ) && !(pRF[lInstr.Op2]
&& pRF[lInstr.Op1]);
                    fptraza(gflog, 1, " meaning R%d ^= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    break;
               case 12: /*    The LSL instruction logically shifts the contents of Ri one bit left,
and
                              puts the result into the Rj.  */
                    fptraza(gflog, 1, " meaning R%d <= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d <= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] << 1;               // one bit logicleft
right shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d <= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 13:
                    /*   The LSR instruction logically shifts the contents of Ri one bit right,
and
                         puts the result into the register Rj.   */
                    fptraza(gflog, 1, " meaning R%d >= R%d", lInstr.Op2, lInstr.Op1);
                    fptraza(gflog, 1, " meaning R%d >= %d", lInstr.Op2, pRF[lInstr.Op1]);
                    pRF[lInstr.Op2] = pRF[lInstr.Op1]; // Rj = Ri
                    pRF[lInstr.Op2] = pRF[lInstr.Op2] >> 1;               // one bit logic right
shitfing of Rj
                    fptraza(gflog, 1, " meaning R%d >= %d", lInstr.Op2, pRF[lInstr.Op2]);
                    break;
               case 14:
                    /*   The CND instruction arithmetically compares the contents Ri and Rj
                         and puts the result of the comparison (as a set of 1-bit signs) to Rj.
*/
                    fptraza(gflog, 1, " meaning R%d ?= R%d", lInstr.Op2, lInstr.Op1);
                    // Victor When executing we can check the result and show it
                    break;
               case 15:
                    /*   The CBR instruction checks the contents of Ri. If it is non-zero,
                         then:
                              1)   the address of the next instruction (i.e., current value of
                                   the PC register + 1) is stored in the Ri register, and
                              2)   the value of Rj is set to the PC register. This means that
                                   the next instruction will be fetched by the address taken Rj
                    */
                    fptraza(gflog, 1, " meaning if R%d GOTO R%d", lInstr.Op1, lInstr.Op2);
                    break;
               default:
                    return 1; //This would never happen;
               }
               lEOLOAD++;
          } while (lEOLOAD <= 2);  // Loop finishes when every part of the load (High and Low)
has been processed
          i++;
          (*pPC)++;
     }while (!lEOCODE);

     return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

int fParseFile (char * *iniFileName*, s_item * *pCode*)

It parses the bin file that results of Eugene's assembler/preparator.

Author:

Victor Castano

Date:

08\03\2012

This function loads data into 1) the code list and 2) the RAM (not yet)

Parameters:

**iniFileName*	is the bin file to be parsed
**pCode*	is the code list returned

Todo:

To load, not only the code list, but also the RAM. Disabling the comment for "pRAM[i-1]=pCode[i].HLLL;" will be the only thing left to do.

Returns:

int 0 for correct execution

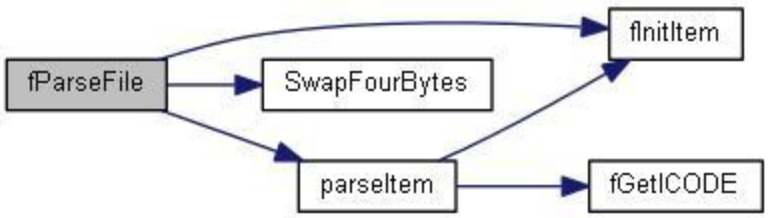
Definition at line 662 of file dissimera.h.

Referenced by main().

```
{
     FILE                *fp;
     unsigned int        instr32=0;
     unsigned short int  i=0;
     unsigned long       lSize;

     fp=fopen(iniFileName,"rb");
     if (!fp)
     {
          fptraza(gflog, 1,"Unable to open file!");
          return -1;
     }
     fptraza(gflog, 1,"fname = %s\n", iniFileName);

     // ========================================
     //The following would browse a file an put it into a buffer
     // ========================================
     // obtain file size
     fseek (fp , 0 , SEEK_END);
     lSize = ftell (fp);
     rewind (fp);
     //fptraza(gflog, 1,"\n Number of Bytes = %d bytes \n 32-bit-instructions = %d \n 16-bit
instructions = %d \n",lSize, lSize/4, lSize/2);
     fptraza(gflog, 1, "\n Number of Bytes = %d bytes \n Number of 32-bit instruction-data = %d
\n",lSize, lSize/4);

     fInitItem(&pCode[0]);         // Initializing the first config element in position 0
     pCode[0].type=T_SPECIAL; // Setting the load as special

     //Browse the file and store the contents into 1)Code List and 2)RAM
     rewind (fp);
     for(i = 1; i <= lSize/4; i++)                          // Position 0 of the array is
deliberately left empty for future configuration purposes
     {
```

```
          pCode[i].linenum=i;
          fread(&pCode[i].HLLL, sizeof(int), 1, fp);
          pCode[i].HLLL = SwapFourBytes(pCode[i].HLLL);

          // Store elements into code list
          pCode[i] = parseItem(i, pCode[i].HLLL);

          // Store elements into RAM
          //pRAM[i-1]=pCode[i].HLLL;

          pCode[0].linenum++;                                    // Increase the number of
elements in the array of Code
          fptraza(gflog, 1,"\n\n");
     }

     // terminate
     fclose(fp);

     return 0;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

void fset_type (s_item * *item*, char *type*)

It sets the type of the item/load.

Author:
Victor Castano

Date:
08\03\2012

Parameters:

**item*	is the structure with the item
type	is the type of Load

Returns:
void

Definition at line 407 of file dissimera.h.

```
{
     item->type = type;
     return;
}
```

int fshowInstruction (s_instruction * *pInstr*)

It shows the instruction included as an input parameter.

Author:

Victor Castano

Version:

0.1a

Date:

08\03\2012

Parameters:

*pInstr	

Returns:

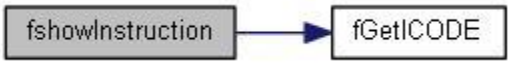
int 0 for correct execution

Definition at line 760 of file dissimera.h.

Referenced by fExecuteCode(), and fParseCode().

```
{
     unsigned int    numchar=0;
     char lsIcode[20]=" ";

     fGetICODE(pInstr->i_code, lsIcode);
     numchar=fptraza(gflog, 1,"    %s R%d R%d ", lsIcode, pInstr->Op1, pInstr->Op2);
     return numchar;
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

fshowItem (s_item * *item*)

fuction that shows an item on the screen using fptraza.

```
\author        Victor Castano
\version  0.1a
\date          08\03\2012
```

Parameters:

*item	

Returns:

int 0 for correct execution

Definition at line 733 of file dissimera.h.

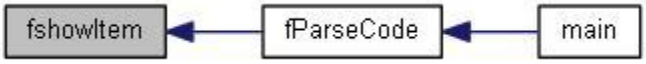
Referenced by fExecuteCode(), and fParseCode().

```
{
     char lsIcodeA[20]=" ";
     char lsIcodeB[20]=" ";

     fptraza(gflog, 1, "\n\n\n %d  ", item->linenum);
     fptraza(gflog, 1,"         %04X               %04X \n   ",item->HL, item->LL);
     colorprintI16(item->HL);
     fptraza(gflog, 1,"   ");
```

```
     colorprintI16(item->LL);
     fptraza(gflog, 1,"\n");

     fGetICODE(item->InstrA.i_code, lsIcodeA);
     fGetICODE(item->InstrB.i_code, lsIcodeB);
     //numchar=fptraza(gflog, 1,"    %s R%d R%d           %s R%d R%d\n", lsIcodeA,
item->InstrA.Op1, item->InstrA.Op2, lsIcodeB, item->InstrB.Op1, item->InstrB.Op2);
     return 0;
}
```

Here is the call graph for this function:

Here is the caller graph for this function:



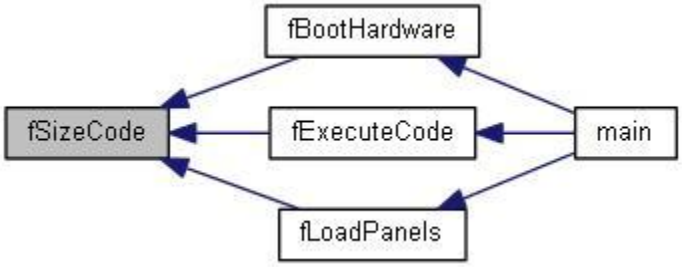
int fSizeCode (s_item * *pCode*)

Definition at line 715 of file dissimera.h.

Referenced by fBootHardware(), fExecuteCode(), and fLoadPanels().

```
{
     unsigned int size=0;

     size = pCode[0].linenum;
     return size;
}
```

Here is the caller graph for this function:



s_item parseItem (unsigned int *numline*, unsigned int *instr32*)

It will return an structure of the 2x16bit instructions filled.

Author:

Victor Castano

Date:

08\03\2012

Parameters:

numline	
instr32	

Returns:

s_item: an structure containing the 2x16bit instructions

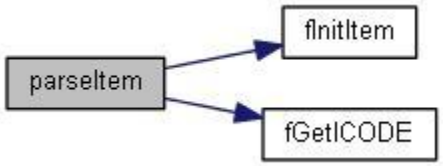
If the input parameter numline is 0 then linenum will not be filled within the structure. It is important to discriminate data from instructions. This is not trivial: setting up the load to T_INSTRUCTIONS leaving the rest as DATA. Any data stored after the STOP instruction will not be browsed. When the current load is an instruction we can determine which one is the next instruction

Definition at line 606 of file dissimera.h.

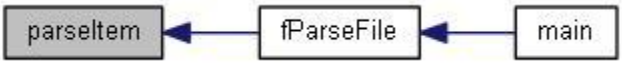
Referenced by fParseFile().

```
{
     s_item              item;
     unsigned int   l instr32 = instr32;    //Backup of instruction due to bit operations
modifying original

     //Initialize item
     fInitItem(&item);

     if (numline!=0)
     {
          item.linenum = numline;
     }


     //fptraza(gflog, 1," \n LIne: %d \n", item.linenum);
     fptraza(gflog, 1," \n %d ", item.linenum-1);

     //Loading the structure
     item.HLLL=instr32;
     item.HL = (short int)((l instr32 & 0xFFFF0000)>>16);
     item.LL= (short int)(instr32 & 0x0000FFFF);
     fptraza(gflog, 1,"         %04X              %04X \n   ",item.HL, item.LL);
     colorprintI16(item.HL);
     fptraza(gflog, 1,"   ");
     colorprintI16(item.LL);
     fptraza(gflog, 1,"\n");
     item.InstrA.f_code = (char)((item.HL & 0xC000)>>14);
     item.InstrB.f_code = (char)((item.LL & 0xC000)>>14);
     item.InstrA.i_code = (char)((item.HL & 0x3C00)>>10);
     item.InstrB.i_code = (char)((item.LL & 0x3C00)>>10);
     item.InstrA.Op1 = (char)((item.HL & 0x03E0)>>5);
     item.InstrB.Op1 = (char)((item.LL & 0x03E0)>>5);
     item.InstrA.Op2 = (char)((item.HL & 0x001F));
     item.InstrB.Op2 = (char)((item.LL & 0x001F));

     char lsIcodeA[20]=" ";
     char lsIcodeB[20]=" ";
     fGetICODE(item.InstrA.i_code, lsIcodeA);
     fGetICODE(item.InstrB.i_code, lsIcodeB);
     fptraza(gflog, 1,"    %s   R%d  R%d           %s  R%d  R%d\n", lsIcodeA, item.InstrA.Op1,
item.InstrA.Op2, lsIcodeB, item.InstrB.Op1, item.InstrB.Op2);

     return item;

}
```

Here is the call graph for this function:



Here is the caller graph for this function:



int SwapFourBytes (int *dw*)

It converts four bytes from Little Endian to Big Endian.

Author:
Victor Castano

Date:
08\03\2012

Eugene's bin files from assembler-preparator seem to be in Big Endian format. This function and the function SwapTwoBytes will help to convert from Little Endian to Big Endian.

Parameters:

dw	is the int with the four bytes in Little Endian format

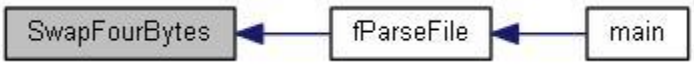
Returns:
int with the two bytes in Big Endian format

Definition at line 446 of file dissimera.h.

Referenced by fParseFile().

```
{
    register int tmp;
    tmp =  (dw & 0x000000FF);
    tmp = ((dw & 0x0000FF00) >> 0x08) | (tmp << 0x08);
    tmp = ((dw & 0x00FF0000) >> 0x10) | (tmp << 0x08);
    tmp = ((dw & 0xFF000000) >> 0x18) | (tmp << 0x08);
    return(tmp);
}
```

Here is the caller graph for this function:



short int SwapTwoBytes (short int *w*)

It converts two bytes from Little Endian to Big Endian.

Author:
Victor Castano

Date:
08\03\2012

Eugene's bin files from assembler-preparator seem to be in Big Endian format. This function and the function SwapFourBytes will help to convert from Little Endian to Big Endian.

Parameters:

w	is the short int with the two bytes in Little Endian format

Returns:
short int with the two bytes in Big Endian format

Definition at line 427 of file dissimera.h.

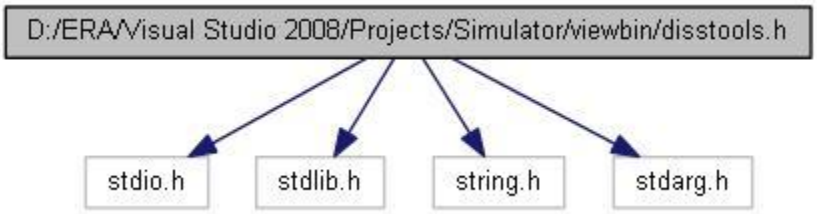
```
{
      register short int tmp;
      tmp =  (w & 0x00FF);
      tmp = ((w & 0xFF00) >> 0x08) | (tmp << 0x08);
      return(tmp);
}
```

D:/ERA/Visual Studio 2008/Projects/Simulator/viewbin/disstools.h File Reference

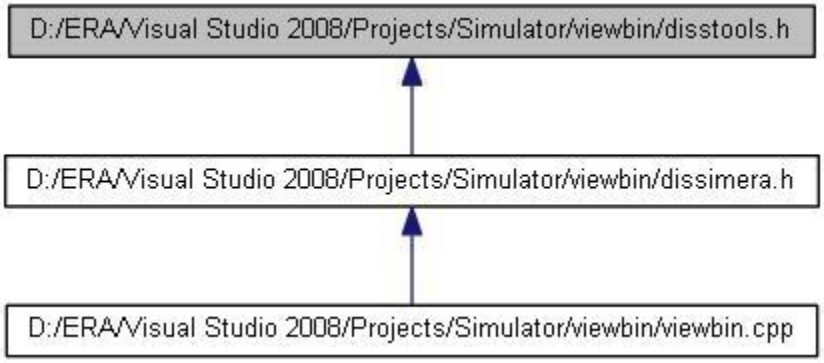
Header file for the general tools of Dissimera.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
```

Include dependency graph for disstools.h:



This graph shows which files directly or indirectly include this file:



Functions

- int **fptraza** (FILE *file, short int level, char *format,...)
- void **colorprintI16** (short int n)
- void **colorprintI8** (unsigned char n)
- void **colorprintI32** (unsigned int n)

Variables

- FILE * **gflog** = NULL
- short int **gTLEVEL** = 0

Detailed Description

Header file for the general tools of Dissimera.

It contains the generic variables and functions that could be used in other projects, not only in Dissemera.

```
\author        Victor Castano
```

```
\version  0.1a
```
\date 08\03\2012 It contains the generic variables and functions that could be used in other projects, not only in Dissemera.

Definition in file **disstools.h**.

Function Documentation

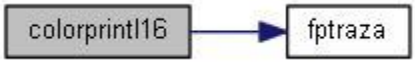
void colorprintl16 (short int *n*)

Definition at line 56 of file disstools.h.

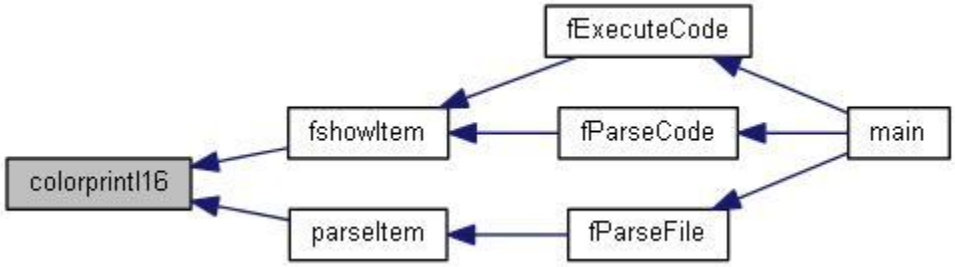
Referenced by fshowItem(), and parseItem().

```
{
    unsigned int i;
    for(i = 0; i<16; i++) {
        if(n&(0x8000>>i)) fptraza(gflog, 1, "1"); else fptraza(gflog, 1, "0");
          if(i == 1) fptraza(gflog, 1," "); /*put a space between bytes*/
          if(i == 5) fptraza(gflog, 1," "); /*put a space between bytes*/
        if(i == 10) fptraza(gflog, 1," "); /*put a space between bytes*/
    }
}
```

Here is the call graph for this function:



Here is the caller graph for this function:

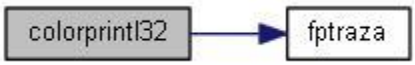


void colorprintl32 (unsigned int *n*)

Definition at line 84 of file disstools.h.

```
{
    unsigned int i;
    for(i = 0; i<32; i++) {
        if(n&(0x80000000>>i)) fptraza(gflog, 1,"1"); else fptraza(gflog, 1,"0");
          if(i == 1) fptraza(gflog, 1," "); /*put a space between bytes*/
          if(i == 5) fptraza(gflog, 1," "); /*put a space between bytes*/
        if(i == 10) fptraza(gflog, 1," "); /*put a space between bytes*/
          if(i == 15) fptraza(gflog, 1,"      "); /*put a space between bytes*/
          if(i == 17) fptraza(gflog, 1," "); /*put a space between bytes*/
          if(i == 21) fptraza(gflog, 1," "); /*put a space between bytes*/
          if(i == 26) fptraza(gflog, 1," "); /*put a space between bytes*/
    }
}
```

Here is the call graph for this function:



void colorprintl8 (unsigned char *n*)

Definition at line 69 of file disstools.h.

```
{
    unsigned int i;
    for(i = 0; i<8; i++) {
        if(n&(0x80>>i)) fptraza(gflog, 1, "1"); else fptraza(gflog, 1, "0");
          if(i == 1) fptraza(gflog, 1," "); /*put a space between bytes*/
          if(i == 5) fptraza(gflog, 1," "); /*put a space between bytes*/
        if(i == 10) fptraza(gflog, 1," "); /*put a space between bytes*/
    }
}
```

Here is the call graph for this function:

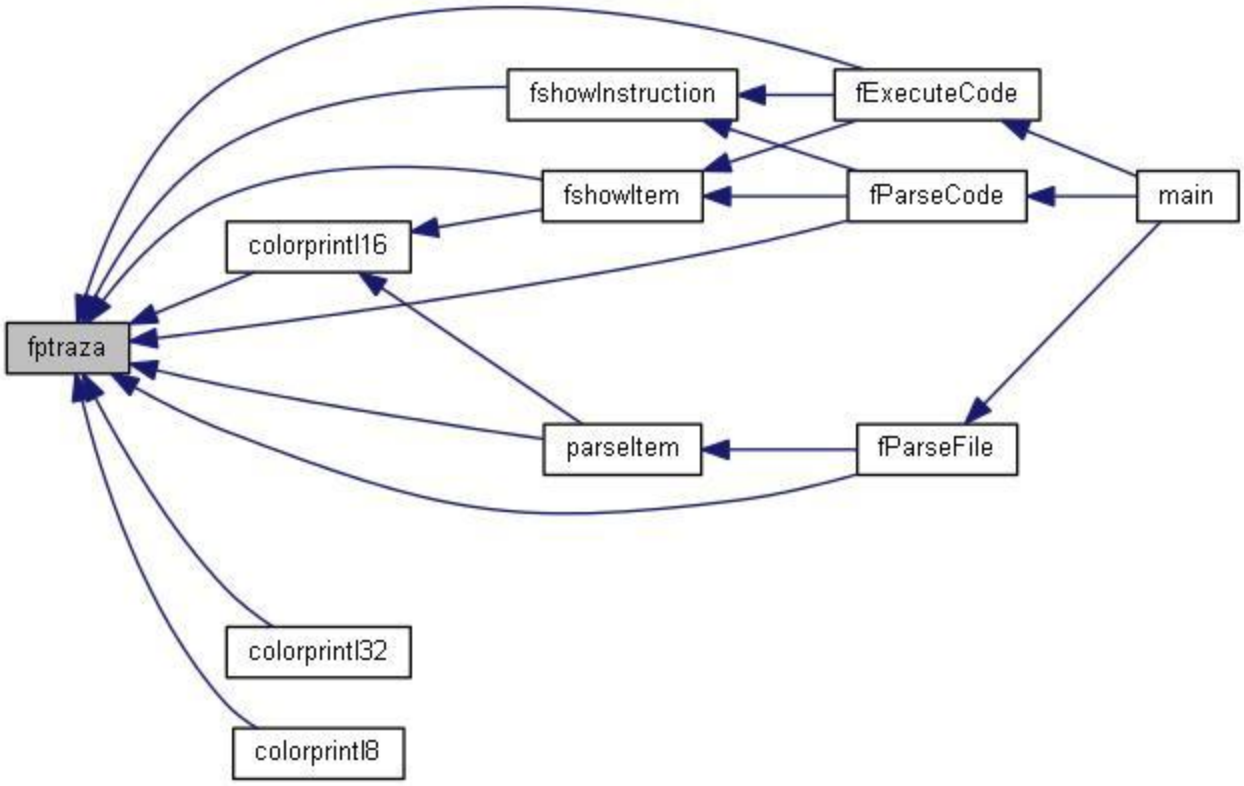
int fptraza (FILE * *file*, short int *level*, char * *format*, ...)

Definition at line 31 of file disstools.h.

Referenced by colorprintI16(), colorprintI32(), colorprintI8(), fExecuteCode(), fParseCode(), fParseFile(), fshowInstruction(), fshowItem(), and parseItem().

```
{
     int i = 0;
     va_list args;
     char buf[255];

     if (gTLEVEL > level)
     {
          return i;
     }

     va_start(args, format);
     vsprintf(buf, format, args);
     va_end(args);

     i= fprintf(file, buf, args);

     return i;
}
```

Here is the caller graph for this function:



Variable Documentation

FILE* gflog = NULL

Definition at line 24 of file disstools.h.

Referenced by colorprintI16(), colorprintI32(), colorprintI8(), fExecuteCode(), fParseCode(), fParseFile(), fshowInstruction(), fshowItem(), main(), and parseItem().

short int gTLEVEL = 0

Definition at line 27 of file disstools.h.

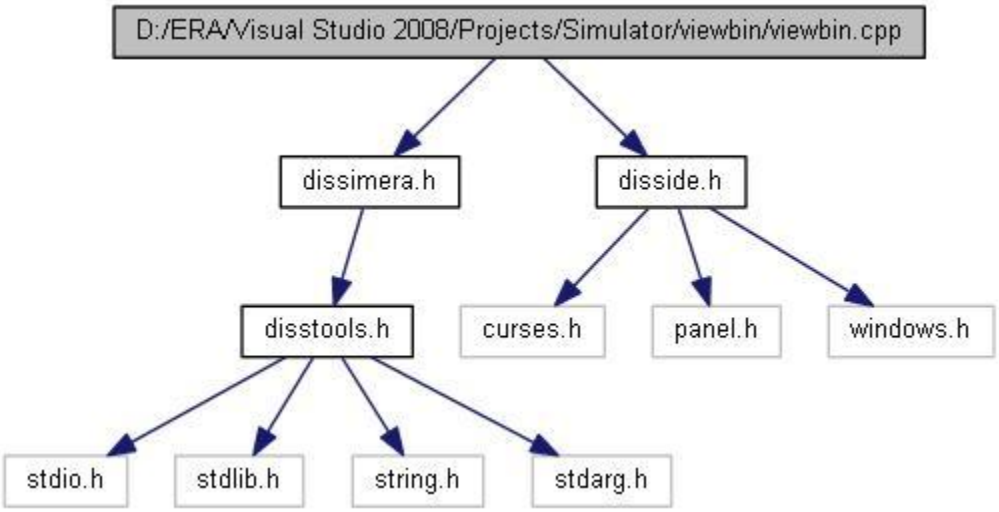
Referenced by fptraza().

D:/ERA/Visual Studio 2008/Projects/Simulator/viewbin/viewbin.cpp File Reference

Main source file for the Dissimera simulator.

```
#include "dissimera.h"
#include "disside.h"
```

Include dependency graph for viewbin.cpp:



Defines

- #define **_CRT_SECURE_NO_DEPRECATE**

Functions

- int **main** (int argc, char *argv[])
  Main fuction of the disassembler/simulator DISSEMERA.

Detailed Description

Main source file for the Dissimera simulator.

```
\author        Victor Castano
\version  0.1a
```

\date 08\03\2012 This project started as a Little utility to transform a bin file (ready for altera)into a human readable version and ended up as a disassembler/simulator of the ERA architecture. It contains the main function and it is the starting running point of the application. Todo:

```
          13/09/2011
          - Crear una variable (global??) que tenga la ventana activa en todo momento
          - Otra variable con modo edit on cuando se presione enter
          - Cada ventana deberia tener el valor del cursor actual

          14/09/2011
```

```
- Crear un array de ventanas que se puede pasar como parametro por referencia
  a las funciones que modifiquen las ventanas
- Ventana activa
- Played with fInitExec(&g_exec)
- note tested

19/09/2011
- Fixed pad and windows
- Worked on Status Panel Scroll

20/09/2011
- Will: F10 , F5 and F9 (rollback) compilation.
```

- Will: Scroll down 1 and 3 panels at once. Bug:

Warning:

See also:

dissimera.h for documentation of variables and functions of the dissasembler/simulator
disside.h for documentation of variables and functions of the User Interface

Definition in file **viewbin.cpp**.

Define Documentation

#define _CRT_SECURE_NO_DEPRECATE

Definition at line 40 of file viewbin.cpp.

Function Documentation

int main (int *argc*, char * *argv*[])

Main fuction of the disassembler/simulator DISSEMERA.

Author:
Victor Castano

Version:
0.1a

Date:
08\03\2012

Bug:

Warning:

Parameters:

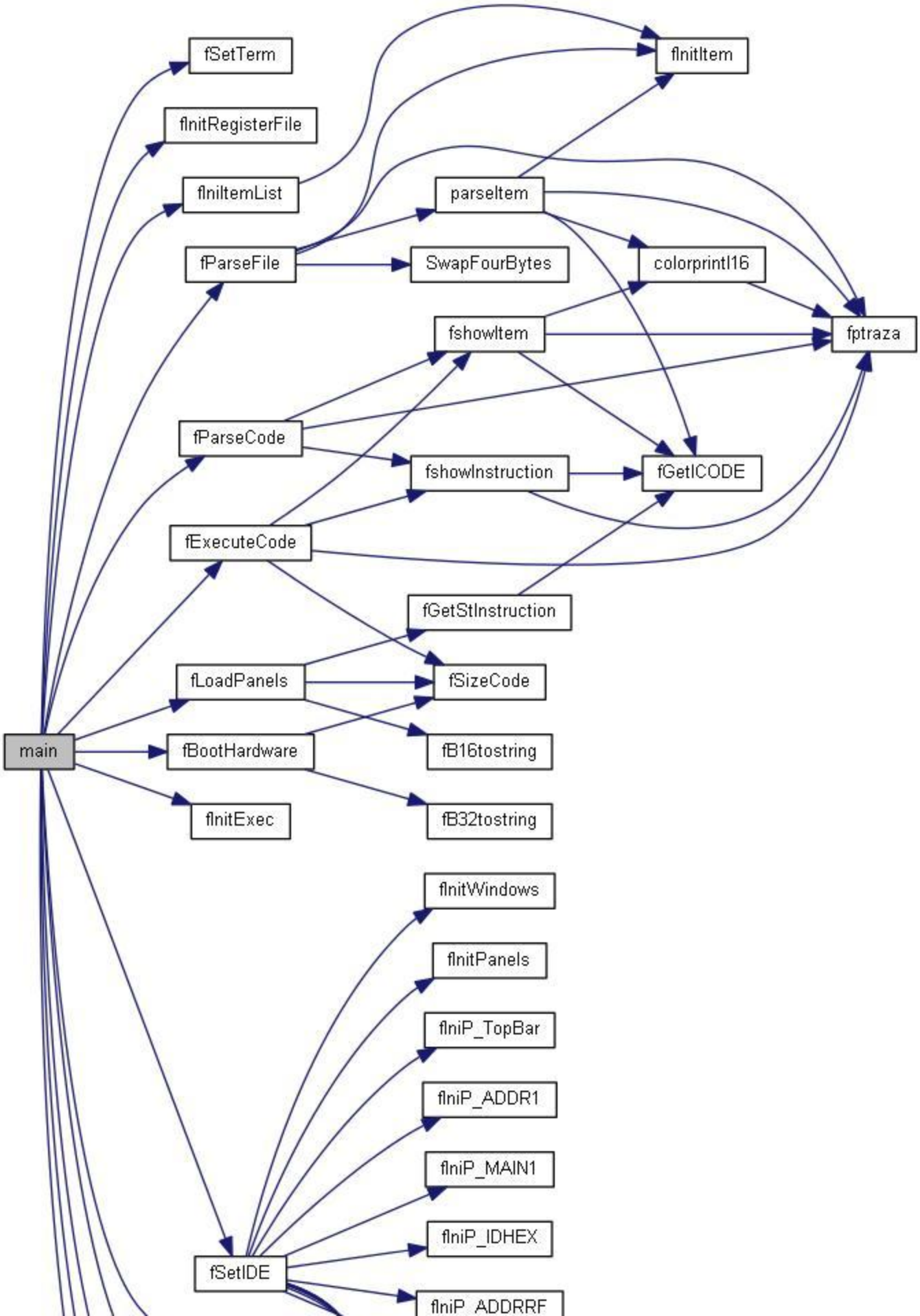
argc	argument count: contains the number of arguments passed to the program
argv	argument vector: is a one-dimensional array of strings. Each string is one of the arguments that was passed to the program.

Definition at line 57 of file viewbin.cpp.

```
{
     char           gSlogname[255]="log.txt";     // gSlogname
     unsigned int   gPC=0;                            // Program counter
     unsigned int   gIR=0;                            // Instruction register
     //unsigned int gNI=0;                            // Next Instruction
     unsigned int   gRF[32];                       // Brief description after the member //
Register File
     s_item              gLcode[MAX_LIST_SIZE];        // List of Code
     s_item              glistRAM[MAX_LIST_SIZE]; // List of RAM
     s_window       g_wins[NUMWINDOWS];
     s panel             g panels[NUMPANELS];
     s_exec              g_exec;
     short int      gactiveP=P_MAIN1;
     short int      gCinstr=0;
     short int      gtempP=P MAIN1;
     int                 gkey=0;


     gflog = fopen(gSlogname, "w");
     if (gflog == NULL)
     {
          perror("failed to open log file");
          return EXIT_FAILURE;
     }

     //Setting up the terminal to max. resolution
     fSetTerm();

     //Initializating the Register File
     fInitRegisterFile(gRF);

     // Initializating the RAM
     //fInitRAM(RAM_ELEMENTS, gRAM);

     // Initializating the list of code
     fIniItemList(MAX_LIST_SIZE, gLcode);
     fIniItemList(MAX_LIST_SIZE, glistRAM);

     //Initializating the Execution structure
     fInitExec(&g exec);

     //   Parsing the bin file from Eugene's bin file (after assembling/preparation) and
     //   1) loading data into the code list and 2) loading the RAM
     fParseFile(argv[1], gLcode);

     // Making an identical copy of the code list into a RAM list
     memcpy(glistRAM, gLcode, MAX_LIST_SIZE*sizeof(s_item));

     //   Parsing the array of code and discriminating data and instructions
     //   - gPC should be 0 for ERA in order to start in memory location 0
     fParseCode(gLcode, gRF, glistRAM, &gPC);

     // Initializating Hardware:Register FILE PC and IR;
     fInitRegisterFile(gRF);
     gPC=0;
     gIR=0;

     //Setting up Curses
     initscr();
     cbreak();
     noecho();
     keypad(stdscr, TRUE);
     curs_set(0);

     /* Initialize all the colors */
     start_color();
     init_pair(0, COLOR_WHITE, COLOR_BLACK);
     init_pair(1, COLOR_BLUE, COLOR_BLACK);
     init pair(2, COLOR GREEN, COLOR BLACK);
     init_pair(3, COLOR_RED, COLOR_BLUE);
     init_pair(4, COLOR_RED, COLOR_BLACK);
```

```
     init_pair(5, COLOR_MAGENTA, COLOR_BLACK);
     init_pair(6, COLOR_YELLOW, COLOR_BLACK);
     init pair(7, COLOR WHITE, COLOR CYAN);

     // Setting up the IDE frame
     fSetIDE(g_panels, g_wins, gLcode[0].linenum);

     // Loading Panels with data/instrucions
     fLoadPanels(g_panels, g_wins, gLcode, gRF, glistRAM, gPC);

     // Loading Hardware with content
     fBootHardware(g_panels, g_wins, gLcode, gRF, glistRAM, gPC);
//   getch();


     // Active window by default is MAIN (highlighting)
     gactiveP=P_MAIN1;
     //fHighlightPanel(g_wins[gactiveP].win, 1);

     fHighlightPanel(gactiveP, 1, g_wins);

     //Highlighting current Instruction

     fUpdateCursors_Main(g_wins, g_exec.nInstrEx, gPC,COLOR_PAIR(3|A_BOLD));

     do
     {
          switch(gkey)
          {
               case CTRL_LEFT_KEY:  // CTRL+LEFT
                    fmovePanel(gkey, &gactiveP, g_panels,g_wins);
                    break;
               case CTRL_RIGHT_KEY: // CTRL+RIGHT KEY
                    fmovePanel(gkey, &gactiveP, g_panels,g_wins);
                    break;
               case CTRL_UP_KEY:   // CTRL+UP KEY
                    fmovePanel(gkey, &gactiveP, g_panels,g_wins);
                    break;
               case CTRL_DOWN_KEY: //CTRL+DOWN KEY
                    fmovePanel(gkey, &gactiveP, g panels,g wins);
                    break;
               case KEY_F5:        // F5 Execution
                    // Execution code still needed
                    break;
               case KEY_F9:        // F9 Rollback
                    fExecuteCode(g_panels, g_wins, gLcode, gRF, glistRAM, &gPC, &g_exec, 13);
                    break;
               case KEY_F10:       // F10 Execution/Debugging step by step
                    fExecuteCode(g panels, g wins, gLcode, gRF, glistRAM, &gPC, &g exec, 13);
                    break;
               case KEY_DOWN: //DOWN KEY
                    fHandleKeyDown(g_panels, g_wins, gLcode, gRF, glistRAM, &gPC, &g_exec);
                    break;
               case KEY_UP:   //UP KEY
                    fHandleKeyUp(g_panels, g_wins, gLcode, gRF, glistRAM, &gPC, &g_exec);
                    break;
               case KEY PGDOWN://PAGE DOWN
                    break;
               case KEY_PGUP: //PAGE UP
                    break;
               default:
                    beep();
                    break;
          }

          // Show current window on TOP panel
          werase(g_wins[P_SUBTOPBAR].win);
          mvwaddnstr(g_wins[P_SUBTOPBAR].win, 0,1,g_panels[gactiveP].label,-1);
          wrefresh(g wins[P SUBTOPBAR].win);

     }while((gkey = getch()) != ESCAPE_KEY);
```

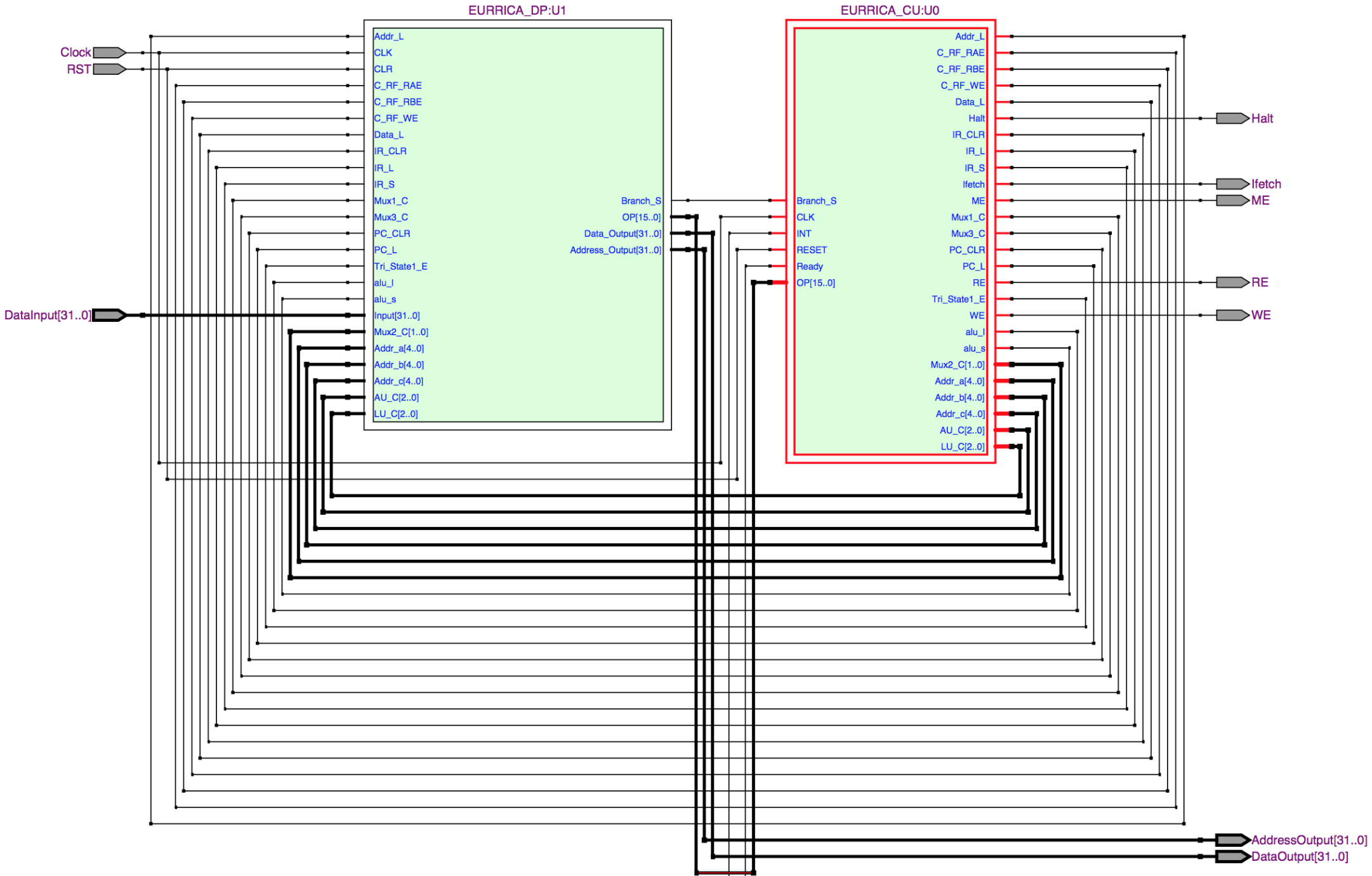
```
    endwin();

    return 0;
```
}Here is the call graph for this function:


main
fSetTerm
flnitRegisterFile
flniItemList
fParseFile
SwapFourBytes
flnitItem
parseItem
colorprintl16
fptraza
fshowItem
fParseCode
fshowInstruction
fGetICODE
fExecuteCode
fGetStInstruction
fLoadPanels
fSizeCode
fBootHardware
fB16tostring
flnitExec
fB32tostring
flnitWindows
flnitPanels
fIniP_TopBar
fIniP_ADDR1
fIniP_MAIN1
fIniP_IDHEX
fSetIDE
fIniP_ADDRRF

Index

Appendix E

An example of parts of the RTL Design of ERRIC's CPU


EURRICA_DP:U1
Addr_L
CLK
CLR
C_RF_RAE
C_RF_RBE
C_RF_WE
Data_L
IR_CLR
IR_L
IR_S
Mux1_C
Mux3_C
PC_CLR
PC_L
Tri_State1_E
alu_l
alu_s
Input[31..0]
Mux2_C[1..0]
Addr_a[4..0]
Addr_b[4..0]
Addr_c[4..0]
AU_C[2..0]
LU_C[2..0]
Branch_S
OP[15..0]
Data_Output[31..0]
Address_Output[31..0]
EURRICA_CU:U0
Branch_S
CLK
INT
RESET
Ready
OP[15..0]
Addr_L
C_RF_RAE
C_RF_RBE
C_RF_WE
Data_L
Halt
IR_CLR
IR_L
IR_S
Ifetch
ME
Mux1_C
Mux3_C
PC_CLR
PC_L
RE
Tri_State1_E
WE
alu_l
alu_s
Mux2_C[1..0]
Addr_a[4..0]
Addr_b[4..0]
Addr_c[4..0]
AU_C[2..0]
LU_C[2..0]
Clock
RST
DataInput[31..0]
Halt
Ifetch
ME
RE
WE
AddressOutput[31..0]
DataOutput[31..0]