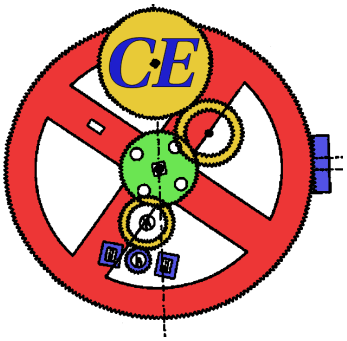


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2012

MSc THESIS

An Adaptive Defect-Tolerant Multiprocessor Array Architecture

Georgios Smaragdos

CE-MS-2012-08

Abstract

Recent trends in transistor technology have dictated the constant reduction of device size. One negative effect stemming from the reduction in size and increased complexity is reduced reliability. This paper is centered around the matter of fault recovery, in the subject of device fault-tolerance, and graceful system degradation in the presence of hard faults. Using a sparing strategy to re-use functional pipeline stages of faulty cores, we take advantage of the natural redundancy of multi-cores. This is done by the incorporation of a re-configurable network in which the cores of the system sit upon and has the ability to re-direct the data flow from the faulty pipeline stages of damaged cores to spare functional ones. The implementation requires the absence of global signals and thus pipeline stage operation needs to be decoupled. We also develop the bi-directional switch required for the network and implement a 4-core working example of our architecture as proof of concept and to evaluate the design. The 4-core design can guarantee correct functionality with 75% of system non-functional, in the best case scenario. The Defect-Tolerant pipeline has overhead of 1.92% in execution cycles and 14.4% in terms of operating frequency, for our custom made stress-marks. Such a system implemented with un-pipelined interconnect would lead to a pipeline with 50% lower frequency and $\times 2.1$ longer overall execution time when the system has no faults. With our architecture and pipelined interconnect the frequency overhead is reduced by 34% and the overall execution time cost by 28% in the full 4-core system. The total execution time overhead, for our stress-marks, in the complete system ranges from $\times 1.5$ to $\times 3.8$ compared to the baseline, depending on the number of defects in the array. The area overhead is around 69% and power consumption, without incorporating any advanced power saving technique, is estimated between $\times 4$ to $\times 5$ times higher compared to the baseline.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

An Adaptive Defect-Tolerant Multiprocessor Array Architecture

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Georgios Smaragdos
born in Thessaloniki, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

An Adaptive Defect-Tolerant Multiprocessor Array Architecture

by Georgios Smaragdos

Abstract

Recent trends in transistor technology have dictated the constant reduction of device size. One negative effect stemming from the reduction in size and increased complexity is reduced reliability. This paper is centered around the matter of fault recovery, in the subject of device fault-tolerance, and graceful system degradation in the presence of hard faults. Using a sparing strategy to re-use functional pipeline stages of faulty cores, we take advantage of the natural redundancy of multi-cores. This is done by the incorporation of a re-configurable network in which the cores of the system sit upon and has the ability to re-direct the data flow from the faulty pipeline stages of damaged cores to spare functional ones. The implementation requires the absence of global signals and thus pipeline stage operation needs to be decoupled. We also develop the bi-directional switch required for the network and implement a 4-core working example of our architecture as proof of concept and to evaluate the design. The 4-core design can guarantee correct functionality with 75% of system non-functional, in the best case scenario. The Defect-Tolerant pipeline has overhead of 1.92% in execution cycles and 14.4% in terms of operating frequency, for our custom made stress-marks. Such a system implemented with un-pipelined interconnect would lead to a pipeline with 50% lower frequency and $\times 2.1$ longer overall execution time when the system has no faults. With our architecture and pipelined interconnect the frequency overhead is reduced by 34% and the overall execution time cost by 28% in the full 4-core system. The total execution time overhead, for our stress-marks, in the complete system ranges from $\times 1.5$ to $\times 3.8$ compared to the baseline, depending on the number of defects in the array. The area overhead is around 69% and power consumption, without incorporating any advanced power saving technique, is estimated between $\times 4$ to $\times 5$ times higher compared to the baseline.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-08

Committee Members :

Advisor:	Ioannis Sourdis, CSE, Chalmers University Of Technology
Advisor:	Christos Strydis, Department of Neuroscience, Erasmus MC
Member:	Georgi Gaydadjiev, CE, TU Delft
Member:	Edoardo Charbon, CAS, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisors Sourdis Giannis and Strydis Christos for their guidance and patience during the work for this thesis. The work would not be possible without the long discussions and brainstorming with them and their constant attention to detail. Also I would like to thank Siskos Dimitris (Probably still EKA) and Rob Seepers for their invaluable help with the design and evaluation tools. Finally Stavros (levies) Tzilis and Alirad Malek for the help in the calculation of the probabilities for the defect coverage analysis. Thanks guys....

I would also like to thank our secretary, Lidwina, and our sysadmins, Erik and Eef, for their support within the group.

I am very grateful to my parents and family for their constant support.I would not be able to finish without them and I will always appreciate their love and patience. I would also like to thank all the friends that kept me company and helped me all these years in Greece and the Netherlands...Especially Aggelos, Dimitris, Alekos and Roula, Giorgos (all of them...Damn we are a lot), Babis, Sotiria, past and current room-mates in the Netherlands.

Georgios Smaragdos
Delft, The Netherlands
June 28, 2012

1 Introduction

"The beginning is the most important part of the work."
-Plato

Recent engineering trends in the design of computer systems have resulted in the emergence of new challenges for the designers. With transistor device sizes now in the realm of nanometers and their density increasing, one of the important factors that suffers, is reliability. Transistors of smaller size are more susceptible to cosmic rays [19], noise [31] and voltage variations, which increases the rate of soft faults, but also more sensitive to wear-out phenomena [10], which results in permanent faults that can render a device unusable. Smaller technologies have increasingly lower threshold voltages increasing the possibility of error and it is predicted that soft errors are more likely in smaller sizes while average time of functional degradation also falls, as illustrated in Figure 1.1 and Figure 1.2. Moreover, it is argued that for the same reasons, the ability of the designer to create a reliable system will become increasingly more difficult in the future [7].

Adding to this is the increasing use of Embedded and Computers systems in general, in research, logistics and everyday life. Such systems tend to become more complex and advanced as time passes. The complexity does wonders to improve the efficiency and ability of the systems but also has a negative effect on their reliability. Moreover there is a number of applications whereby, due to certain characteristics, a fault is unacceptable.

These applications, that used to rely on simpler and more reliable devices until now, become more advanced as current trends demand. Such systems can include any one or more of these characteristics :

- The process of repairing a damaged component in the application is very costly or practically impossible.
- The application operates in hazardous environments that could alter data acquired or processed, or increase the probability of failure to the extend that the device becomes inefficient.
- The application is critical for the safety or survival of human life.

Systems that fall under this category could be anything from space applications and costly research devices, like particle accelerators, to navigational computers and biomedical systems, such as implants.

Circumstances force today's designers to design devices that are resistant to faults and develop strategies to that direction which are characterized as fault tolerance. Fault tolerance procedures can be divided into 3 fields:

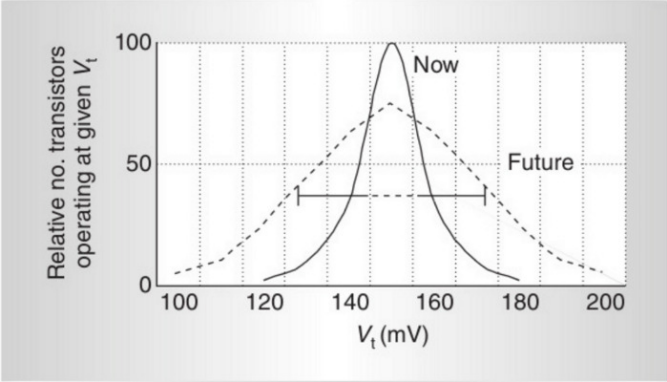


Figure 1.1: Future trend on transistor Voltage Variation [7]

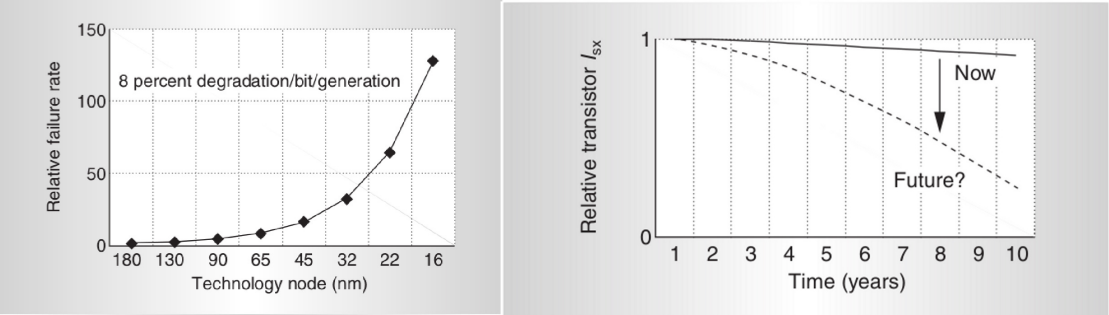


Figure 1.2: Figures showing the projected rate of soft failure and device degradation over time [7]

- Fault Detection
- Fault Diagnosis
- and Fault Recovery

Fault Detection has to do with the mechanisms that detect that a fault is present on a device. *Fault Diagnosis* has to do with determining the component that a fault originated from and the nature of said fault. Lastly, *Fault Recovery* has to do with mechanisms that, by using the information from the previous tasks, attempt to repair or bypass the fault according to its nature and available resources. This thesis is concentrated on the last task; that of Fault Recovery from permanent faults (*defects*).

1.1 Thesis Motivation

There is already a significant body research done in the field of fault tolerance and especially in the field of permanent fault recovery. Most techniques seem to follow 2 ideas: *sparing* and that of *matching*.

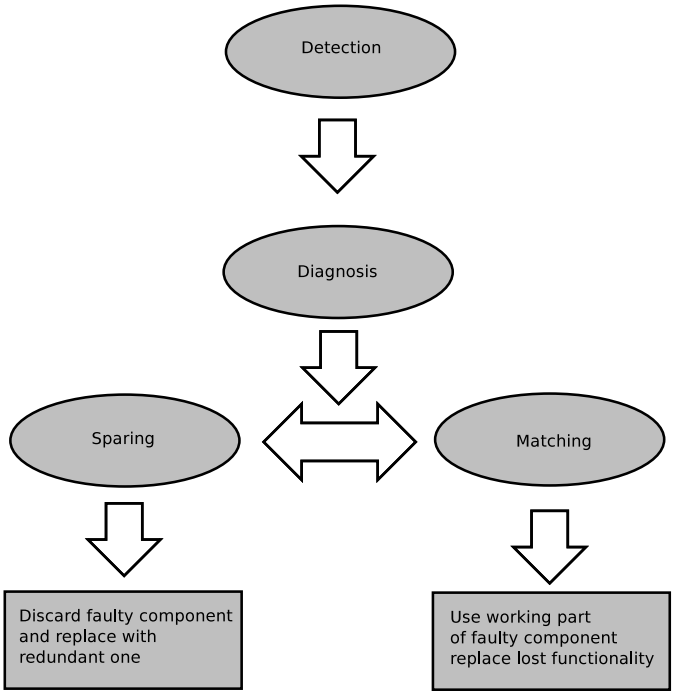


Figure 1.3: Typical Defect Tolerance Tasks.

The general concept is to divide the design into basic blocks identical to each other, otherwise called *Substitutable Resources*. Sparing refers to the strategy of discarding the faulty block and replacing it by another that is available. Matching, on the other hand, tries to determine if the faulty block can be partially used and has mechanisms that allows the block to at least continue to operate with a reduced functionality (Figure 1.3). Obviously matching can achieve better use of the hardware since it uses even the faulty components when possible, albeit with partial functionality. On other hand, sparing is quite simpler to implement and does not require advanced fault-diagnosis mechanisms to determine if a resource is partially usable. Usually, these strategies are achieved by redundancy of resources(or data redundancy if possible). Devices have either extra redundant components, like memories [27] or even smaller components like branch predictors [8] , adders [21] etc. The redundant resources are used in case of a damaged resource, taking its place so the system can have complete functionality. An obvious problem with these ideas is the cost. The ,design has complete copies of components not used at all and wasting space when the design operates without problems; a cost that sometimes applications cannot afford.

Yet, a recent trend in technology could use the idea of redundancy more efficiently under certain circumstances; the extended use of multi-core systems. The use of multi-cores has become a major way of improving performance since it has become increasingly more difficult to optimize a single chip to achieve significant performance benefits. Under these circumstances the use of extra copies of already proven core designs in the same chip has become more cost efficient than the development of new ones for the creation

of new products. Since these systems use identical components, they naturally have a significant amount of regularity. If the architecture allows it, in case of fault in a core, the spare and undamaged components could be used, under the one of the two previous strategies, to amortize the performance loss created by the fault. So, designing a Fault Tolerant multi-core array is potentially an interesting idea considering both its wide use in recent years and its inherent characteristics, that could be suitable for a fault-tolerant implementation.

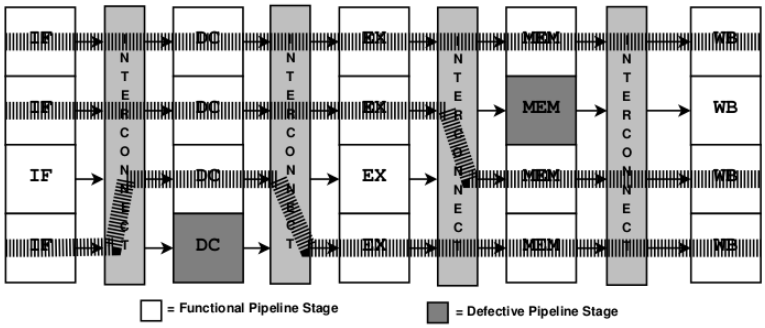
1.2 Problem Statement

In this thesis, the problem of designing a defect-tolerant multi-core array, which can assure graceful degradation in the presence of hard faults, is explored, motivated by what was previously described. As our starting point we use a basic 5 stage RISC processor architecture, suitable for the initial design as it is a common paradigm. The first step is to define the granularity level of the Substitutable Resource. There are 3 typical options one could choose in the case of a multi-core array:

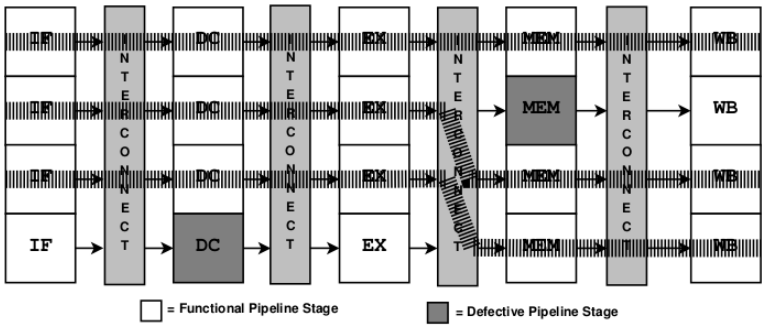
- Component Level
- Stage Level
- Core Level

In the first case, the component becomes the Substitutable Resource. This could be anything from an adder to a branch predictor to cache memories. This option has the most flexibility when it comes to defect tolerance and can produce more efficient designs but the mechanisms to implement fault tolerance at that level are costly and complex. On the opposite side, one can choose the core as the Substitutable Resource. A choice that is certainly easy to implement but not really efficient, since the only straightforward way of implementing defect tolerance is by core redundancy, reducing flexibility and actual defect tolerance. The middle ground would be to use the pipeline stage level. This choice is much more efficient than substituting the whole core, thus providing satisfactory defect-tolerance, but has less complexity in implementation than using component level granularity. Such a system could be able to use a sparing strategy when there are defects in the cores by sharing the functional pipeline stages between the dis-functional cores and reconfiguring the array thus producing a new functional core, bypassing the defects. This can be implemented by putting the stages of the different cores on a reconfigurable interconnect. Thus, by re-configuring the network, the system can change the direction of the data flow, bypassing the faulty stages (Figure 1.4). If the functionality of such a architecture can be proven, the design can be taken one step forward, in the future, by systems that also include re-configurable hardware as wildcard stages being able to replace any faulty stage, if a spare is not available, and provide increased flexibility.

This problem has 2 basic design challenges. First is the algorithmic calculation of the reconfiguration. Since, depending on the defecting stages, there are more than one ways to reconfigure the array, there must be an algorithm that can produce an optimal



(a) Configuration A



(b) Configuration B

Figure 1.4: Example of different configurations concerning the same stage defects

configuration for each case of damaged stages and in a reasonable amount of time for the system to be practical. The second, which is also the subject of this thesis, is the architectural redesigning of the cores to implement the fault-tolerant mechanisms and also the design of the re-configurable network used to connect the core stages. We demand from the proposed system to have certain functional characteristics. First of all we assume that the reconfiguration is taking place off-line. In other words, when a defect is detected, the cores are halted until the reconfiguration is completed. For this idea to be practical it is required from the architecture, despite whatever change done to the array, that the programs running in the cores will be able to run exactly as they are without the need to re-compile them. In short, every possible configuration should be able to work using the same application binaries. The design, implementation and evaluation of such an architecture and its respective interconnect is the purpose of this thesis.

1.3 Thesis Goals

This thesis seeks to make the following contributions:

- Explore the challenges and characteristics of implementing a system as described by the problem statement: A multi-core, defect-tolerant system ,with stage granularity

using a re-configurable interconnect.

- Design and implement an architecture based on the 5-stage RISC pipeline that achieves to decouple the various processor stages so as to make the re-configuration process transparent to the application without any additional application compiling overhead.
- Design and implement the components needed for a re-configurable interconnect suitable for our purposes, mainly the interconnect switches and interconnect pipeline registers.
- Design and implement a example of a full multi-core array using our architecture for the processors and interconnect.
- Evaluate our multi-core with the purpose of recognizing the main overhead of the architectural changes and interconnect implementation. One of the main questions to answer is whether our initial expectation that the performance overhead of the interconnect of the system is at acceptable levels, thus making the idea of the re-configurable interconnect practical.
- Propose future upgrades and improvements according to the evaluation results.

1.4 Thesis Overview

This thesis is organized as follows: In chapter 2 we make a brief presentation of related work in the subject fault recovery and tolerance, in general. Chapter 3 gives the basic concepts needed to follow the remainder of this thesis and also a few additional subjects. These concern the core configuration-selection algorithm a practical system such as the one we describe here could be using and also a description of the DeSyRe project that our work is be linked to. In chapter 4 we present the implementation of the processor architecture, the interconnect components and, finally, a complete example of a 4-Core system using our design. In chapter 5 we describe the development and evaluation tools used, the experimental setup and the evaluation results for our design. Lastly, in chapter 6, we present the conclusions stemming from the evaluation of our work and provide suggestions for future work.

2 Related Work

"Consider the past and you shall know the future."
-Chinese Proverb

In the first chapter the motivation and goals of this thesis were presented. As stated in the previous chapter, there is, already, a number of previous research done in the field of fault tolerance. These solutions vary from design to design depending on the chosen granularity that affects the performance/defect-tolerance trade-offs and nature of the architectures. This chapter will attempt to present a number of these research works and describe their general approach towards the problem.

This chapter is organized as follows: In section 2.1 some important research is presented that used the core as a Substitutable Resource(aka Core Redundancy). The next section refers to solutions for fault tolerance developed for Superscalar cores. Section 2.3 presents a number of research papers that exploit *field programmable* solutions for the creation of reliable systems. Section 2.4 presents two proposed architectures that used stage-level granularity to cope with the fault tolerance issue, which is also the approach of this works's design, *StageNet* and *CCA*. Finally, a summary of the chapter is provided in the last section.

2.1 The Core As a Substitutable Resource

One approach that has been studied for fault tolerance is that of core redundancy. This solution, in general, has the advantage of less complex implementation and less performance costs, compared to other granularity choices, but tends to be less defect-tolerant than other solutions. However, it could make them inefficient in applications with high fault probability. There is a number of issues that needs to be addressed in such designs. These include challenges for *fault detection* and *diagnosis*, device *self-healing*, *memory re-scheduling*,*core isolation* and *thread migration*.

Specifically one such work provides a solution that tries to tackle the problem of fault isolation [3]. A typical issue when using redundant cores, is the mechanism used to isolate and retire the faulty core and redirect the data flow to the functional part of the device. Here, an architecture is proposed that provides configurable mechanisms for isolation of the damaged part of the chip and its deactivation but in an as efficient and cost effective method as possible. Thus, it can provide a device that ensures graceful degradation in the presence of faults. The architecture splits the system in to domains that are isolated. A fault in one domain will not affect the other domains. The isolation is achieved by using techniques in the interconnect which are characterized as *configurable isolation*. These are a number of design techniques that provide "optional logical fault isolation for shared components"

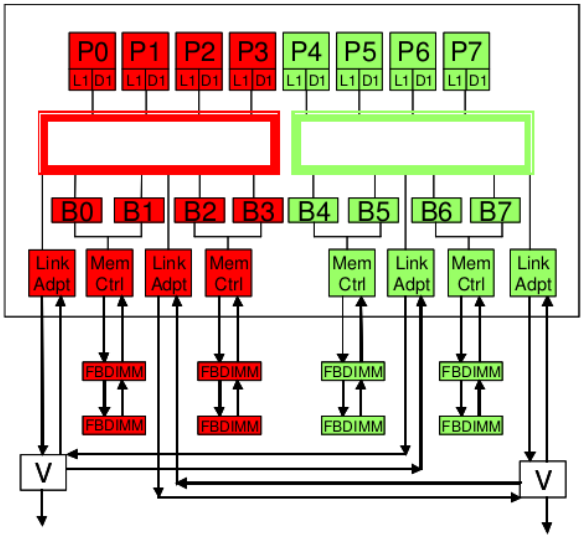


Figure 2.1: Reconfigurable Isolation. The differently colored domain in the system signify isolated domain in the system by the interconnect [3].

Other works try to reduce cost with more effective coherency check mechanisms between redundant components, such as memories. In the case of [14] the authors propose such an improved system for multiprocessor systems with distributed shared-memory. Since such checks between redundant components are required and are part of the critical path, better optimized checking systems can provide overall performance improvement. In addition to this, the paper provides a design that simplifies the coherence-checking mechanisms and permits their re-use by different cores.

Michael D. Powell, et al. propose an architecture with advanced migration mechanisms between cores in a multi-core system [20]. This provides a system that can salvage a damaged core by enabling it to run the instructions that it can still execute, despite the presence of the fault, and send the rest of the instructions, that the damaged core cannot execute, to the fully functional ones. This way the architecture can partially reuse a damaged core when typically this core would we useless.

Lastly, there is also the *ElatiC* architecture proposed by Dennis Sylvester, David Blaauw, and Eric Karl [29]. Here, a complete architecture is developed that copes with defect diagnosis, self healing and performance tuning for multi-core devices.

As a whole, it is apparent that the core-redundancy approach can provide some fault tolerance with good performance trade-offs but only to a certain extent, since using the whole core as a Substitutable Resource lacks flexibility and this limits defect-tolerance.

2.2 Fault-Tolerant Superscalar Cores

Some research has been done for fault tolerance in *superscalar architectures*. These concentrate on exploiting the redundancy of structures that are unique in this case of processors, on the evaluation of the impact defects have on such systems and the

development of metrics to evaluate performance in the presence of faults.

Some efforts concentrate on the design of certain components of multi-core systems and their impact on the performance of reliable designs. One such research concentrates on the switch architecture of superscalar core clusters. It evaluates various switch designs and proposes a switch architecture for reliable CMPs called *Bulletproof* [11].

An architecture concentrated on fault tolerance that exploits the redundancy in the architectural characteristics of superscalar designs was developed by A. Bower, Paul G. Shealy, Sule Ozev and Daniel J. Sorin [9]. This was a technique called *Self-Repairing Array Structures (SRAS)*. Here, the designers take advantage of the regularity of two typical structures of the superscalar architecture, the reorder buffer and the branch history table. The technique includes the mechanisms that detect a faulty part of the arrays and isolate the faulty rows by phasing them out.

Lastly,Premkishore Shivakumar et al. in their research propose a new metric in performance evaluation of both fully functional and damaged chips [24]. The goal is to evaluate the defect susceptibility of multi-core chips and the performance/efficiency in the use of redundancy techniques in such systems for various device technologies.

2.3 Field-Programmable Logic Approaches

The inherent flexibility that *field programmable* devices have and the natural regularity in devices such as FPGAs make them also a good candidate for deploying fault tolerant designs. There is a number of proposals in this case. Some concentrate on providing field programmable logic to implement defect tolerance specifically on the control logic of a device. Recent research efforts also include devices utilizing the natural regularity of FPGAs to produce reliable architectures, with most noted their usage in space applications.

More specifically, some designers propose a mechanism protecting from faults using field-programmable control logic [32, 4]. The control unit is equipped with logic that allows it to detect errors during run-time. When this happens, the control can re-program itself to ensure correct device functionality, with some performance degradation. Since the faults usually do not have an impact to all types of instructions, the control unit is able to restore itself back to normal functionality, as long as there are no uncommitted instructions in flight that are influenced by the damage.

Also, as stated before, the regularity of the FPGA fabric gave motivation for increased use in reliable systems. Already NASA has recognized this aspect alongside with the redundancy that FPGAs have and used them for fault-tolerant designs [25]. Another example of FPGA usage is the *Dependable Multiprocessor* (DM) technology. It was developed for use on payload and robotics missions in space aviation. It provides a cluster with high performance, high dependability and power efficiency [22].

Additional research in the field, using FPGAs was also done by Debayan Bhaduri and Sandeep Shukla [6]. In their research, they present how a tool created for computing thermal perturbations and interconnect noise can be actually used to compute performance/cost trade-offs for reliable systems based on re-configurable hardware. The original scheme used *Markov Random Fields* and *Belief Propagation* methods for its

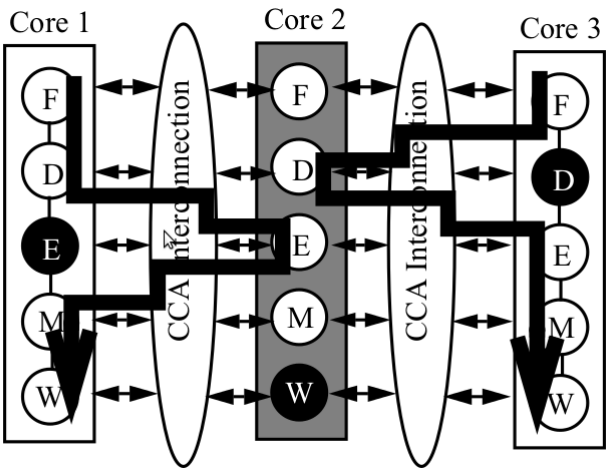


Figure 2.2: The CCA Architecture [23]

purpose. The researchers used a loopy Belief Propagation scheme to accomplish their goals of using the tool to compute the needed trade-offs.

Lastly, other works include efforts that explore the fault diagnosis in FPGA logic blocks, with emphasis on the attempt of reusing faulty blocks [13] and the development of re-configurable design techniques that cover detection, diagnosis and recovery with reusable damaged parts to extend the reliability capability beyond that of what the redundant parts can provide [28]. Furthermore, there is work on the implementation of architectures that allow partial reconfiguration of re-configurable logic while applications are still in execution, ensuring uninterrupted operation [26].

It is clear that the inherent characteristics of re-programmable and re-configurable devices make them suitable for fault tolerant designs. Their flexibility makes them ideal for fine-grain reconfiguration and application of component redundancy and so provide wide defect tolerance. But these designs also inherit the disadvantages of such hardware. These are high performance costs and usually high power consumption.

2.4 Pipeline Stage as Substitutable Resource

Between the core-redundancy techniques that offer good performance results but ensure limited defect coverage, and component redundancy techniques using fine-grain hardware, that seem to have better coverage but higher cost in performance and power, lies the solution of using the pipeline stage as substitutable resource. This idea, which was also used for the design presented in this thesis, has been explored by two other proposed architectures, *StageNet* [15, 16, 17] and the *Core Cannibalization Architecture* (CCA) [23].

The CCA introduces the idea of the *Cannibalizable Core* (Figure 2.2). A core in the system is characterized as such if it is possible for neighboring cores to borrow stages from it to ensure correct functionality in case of fault. So, this core can share its pipeline stages if it is not operational to ensure functionality to the rest, characterized as *non-*

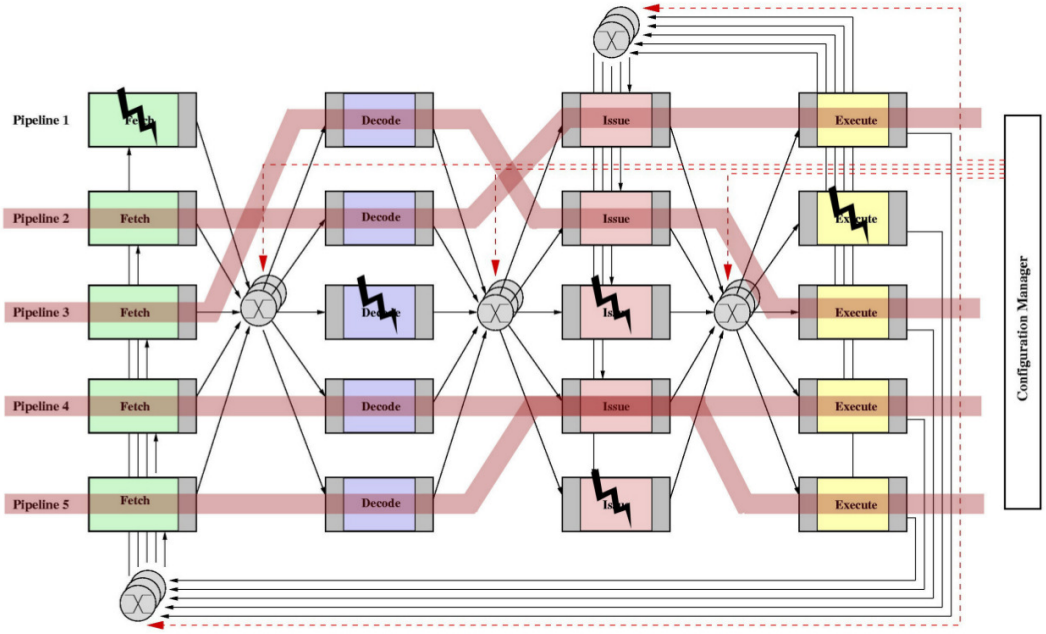


Figure 2.3: The StageNet Architecture [15, 16, 17]

cannibaliazable (NC). The re-routing of data-flow is achieved by multiplexers between the stages of the core choosing either to receive/send data from the previous/next stages of the same core or one of its neighboring ones. The sharing of stages between non-immediate neighboring cores introduces extra pipeline stages in the system, so that an increase in clock period would not be necessary. In order not to make the complexity of the control too high and reduce the number of possible extra cycles in the pipeline, the architecture does not allow a NC core that requires extra stages to borrow more than one stage. This constraint has, though, an impact on the flexibility of the system and limits the defect tolerance coverage. To accommodate for the extra stages, the branch feedback is latched and also feedbacks are added to accommodate the extra data conflicts that come from the extra pipeline stages to achieve instruction bypassing. Also, the design uses buffering in the input and output of stages to ensure stall-signal propagation without the loss of data. This is an expensive method in terms of area but ensures no performance degradation because of the extra stages when a stall occurs. Also the design assigns the jump/branch address computation logic to fetch stage because it is critical for performance for it to be near the fetch stage. So, even though the logic servicing branch/jumps is actually in the decode stage, the address computation always remains in the same core as the fetch stage even if the decode stage is replaced in the case of a fault. This constraint forcing limitations on stage replacement additionally reduces somewhat the design's flexibility in defect coverage.

An architecture that also implements this premise is StageNet (Figure 2.3). It accomplishes this goal by decoupling the various stages of a 5-stage processor and connecting them together using crossbars. When an erroneous stage is detected the crossbars reroute the data from the faulty processor stage into another one of the cluster. This enables

the design to be very flexible in the reconfiguration of the cluster, saving itself from the constraints found in the CCA. The use of crossbars in this design, though, has important performance costs compared to the original 5-stage pipeline. In the crossbar, the delay is the same despite the destination of the data. In every case the performance cost is the maximum possible regardless of weather the cores sharing stages are immediate neighbors or not or even when there is no sharing, when all cores are operational. Furthermore, in order to reduce the number of inputs in the crossbars, the designers tried to reduce the feedback loops needed in the datapath to the very essentials. The finished design included only the WB and Branch resolve feedbacks. It was decided to completely remove all data-forwarding by just stalling the pipeline in case of hazards and detecting them in the issue stage using hardware (*scoreboard*). This effectively eliminates the need to make hazard control checks using feedback lines between stages (leading to a greater number of inputs in the crossbars), that would otherwise be needed. Bypassing is achieved by a *bypass cache* saving the data that might be needed and using them according to the information stored in the scoreboard. This also has the impact of including the bypass logic, the data memory and the execution FUs in the same stage in order again to eliminate the need for more inputs in the crossbars that would hinder performance. Also flush signals are eliminated since each stage is flushing itself using a status register in case of a branch mis-prediction.

2.5 Summary

In this chapter we presented a number of research works that was previously performed on the field of fault recovery and fault tolerance in general. These include strategies that use core granularity providing good performance but limited fault tolerance; fine-grained granularity using reconfigurable logic, having good tolerance and flexibility but significant power and performance costs; and superscalar based architectures. In the middle between coarse-grain and fine-grain hardware usage for fault-tolerant design lies the solution of using the stage as the substitutable resource. This is the approach used in the design that will be presented in the following chapters. Other previous works using the same premise are the CCA architecture and the StageNet. The CCA achieves good performance but includes constraints that reduce flexibility. StageNet solves this by using crossbars to connect all stages in the cluster, thus achieving full flexibility by paying the performance/area cost of the crossbar usage.

3 Background

"A good short-story writer has an instinct for sketching in just enough background to ground the specific story."
-Lynn Abbey

The previous chapters have indicated the motive behind and nature of fault-tolerance research and more specifically of the work presented in this thesis. Naturally, there are a number of design concepts that one needs to be familiar with when tackling this problem. This chapter serves the purpose of presenting said concepts that are needed to understand and follow the content of the work included in this document.

This chapter is organized as follows: Section 3.1 gives a brief presentation of the DeSyRe project, whose framework is linked with the work presented here. The next section is dedicated to the architectural concepts that are required to follow the design implementation presented in the next chapters. It was previously mentioned that the proposed design has two main challenges: that of the algorithmic computation of the core array reconfiguration and the actual architectural design of the array's cores and interconnect network, which is the subject of this thesis. In section 3.3, the work that was was done in conjunction with this thesis in order to tackle the first challenge is briefly presented along with some of its links to the architectural work. Finally, the chapter concludes with a small summary.

3.1 The DeSyRe project

The *on-Demand System Reliability* (DeSyRe) project [1] is a research project with the purpose of developing a fault-tolerant SoC framework for embedded systems. Its goal is to create reliable devices out of unreliable components with reduced performance and power costs and with as much as possible defect tolerance. For this reason, it defines a framework that spans from runtime and software support to hardware and technology. To achieve its target, the framework is divided into 2 types of partitioning: The *physical* and the *logical* partitioning. The design that is presented is this thesis can be seamlessly linked to the DeSyRe's framework organization.

The physical partitioning is referring to the technology used for the various components of the systems, with emphasis on the fault reliability of each section of components. This approach divides the components of the system into *fault-free* (FF) sections and *fault-prone* (FP) sections. The FF part of the design has as goal to provide general reliable control of the FP part. The FF part is currently included by necessity and, since it is expensive, the intention is to keep it as small as possible. The FP section is the main bulk of the system and practically provides all the desired functionality. This section can be implemented with less reliable components, with the FF providing the necessary

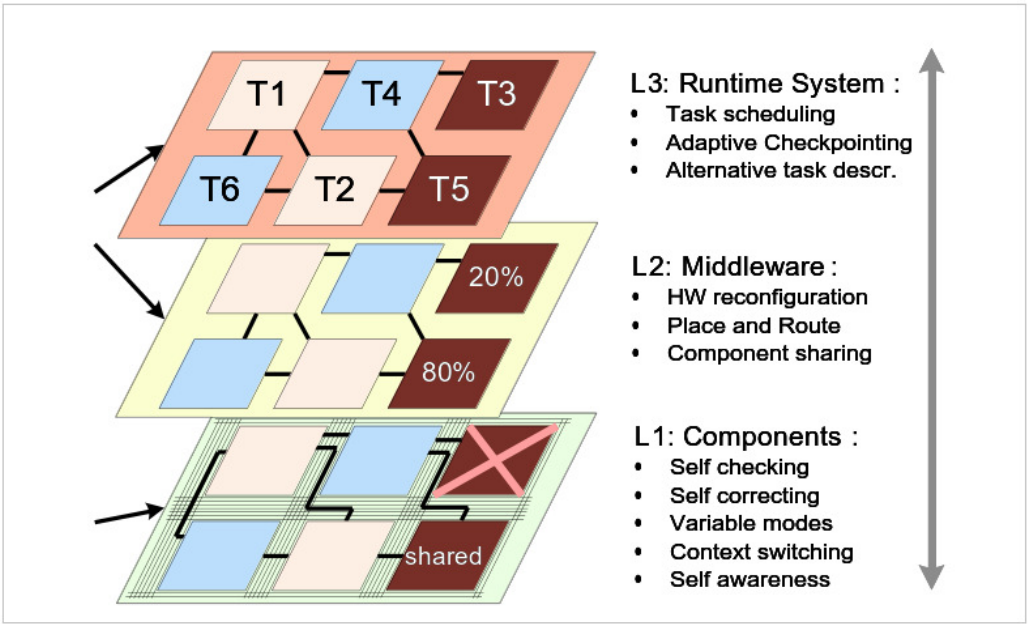


Figure 3.1: Logical partitioning of a DeSyRe SoC: Components layer (SoC functionality), Middleware layer and Runtime-System layer (SoC management) [1].

services to guarantee fault-tolerance. Any kind of reconfiguration, communication with the outside environment and functionality checks on the FP section is handled by the FF.

The second type of partitioning is the logical partitioning. It refers to the functionality of the parts comprising the device. Through this partitioning the system is divided into 3 layers:

- The Component Layer
- The Middleware; and
- The Runtime System.

The component layer refers to the components delivering the functionality, that could themselves possibly even provide fault tolerance. The middleware is responsible for implementing any reconfiguration on the components in the presence of faults and provides a link between the components and the runtime system. The runtime system is the upper level in this partitioning and is, amongst others, responsible for the general control of the system QoS and making decisions on the reconfiguration of the system (Figure 3.1). The partitions, although discrete, can be linked, judging by the nature of each partitioning. Practically speaking, the middleware and runtime systems should be implemented in the FF part of the system and the components on the FP.

The architecture of the work presented in this thesis fits on this framework. In our system we have a number of cores with decoupled pipeline stages, that can, through a

reconfigurable interconnect, share the spare parts of faulty cores to create a functional one, all according to a selection algorithm. The actual multi-core system can be part of the component layer. As a middleware system we can assume the software that produces the bit-stream controlling the network reconfiguration and interconnect switches. Finally, the Runtime System can include the selection algorithm that chooses the most efficient core re-configuration in the presence of a new fault. The part of the device that selects the reconfiguration and the software producing the interconnect control signals and the interconnect will be implemented in the FF part of the device and the multi-core system, with their respective registers in the FP. With this in mind, the architecture can even be later upgraded within the rules of the same framework and include even more functionality in the future.

3.2 Architectural Basis

One of the purposes of this thesis is to describe a fault tolerant processor array architecture. In other words, the fundamental design and structure of the array system. As the design uses the classic 5 Stage RISC processor pipeline paradigm [18] as a basis, some fundamentals of the architecture are needed for someone to be able to follow the implementation of the design. So, in this section these fundamentals will be presented along with concepts that are useful to understand the challenges and solutions given in the implementation of the design.

3.2.1 RISC Instruction Set Architecture

An *Instruction-Set Architecture* (ISA) is the set of instructions that a processor is able to execute. It is the interface between the hardware and the programmer and it refers to assembly language instructions. It defines the types of operations that the hardware must be able to execute and the form of compilation that must be done on the higher level programming input. Depending on the type of processor, there are many approaches one can follow to create an ISA. The paradigm used in this thesis design is the RISC architecture. RISC ISAs have a certain number of properties :

- All operations on data are done in the contents of registers and usually affect the entirety of the register's contents.
- All operations that change the status of the main memory are only the store/load operations moving data from/to registers to/from the memory.
- The various formats of instructions in the ISA are limited in number and usually have the same bit length.

This creates quite a simplified structure. This is done intentionally so to greatly simplify the implementation of the pipeline on the processor's datapath. In our case, the baseline architecture has 32-bit wide instructions and a total count of 16 data registers (*register file*), with register 0 always having the value zero as it is typical. The are 3 types of instructions in our baseline architecture:

- *Arithmetic operations*: These include ALU operations such as adding the contents of 2 or 3 registers between them or with an immediate, comparisons of the same nature, shifting operations on data and loading immediate values on the Register File (RF).
- *Memory-Access operations*: As the name reveals, these are operations that move data from the main memory to the RF and back.
- *Control-Flow Operations*: These are instructions that change the control flow of the program. These can be either direct or conditional based on a comparison.

Since the processor is an integer processor there are no floating-point operations implemented on the architecture.

3.2.2 The Baseline Pipeline

The various tasks in a processor can be made independent for the purpose of pipelining the design. Pipelining is a method of increasing performance of the architecture by increasing the *throughput* of the processor. Throughput refers to the rate of completing instructions at a given time. Pipelining techniques improve this by enabling the various tasks of the datapath to run in parallel. This way the execution of each instruction can begin before the next one finishes, in the same fashion that a factory production line functions. In theory, the goal is for one new instruction to be inserted on every cycle.

The various tasks or sections in the datapath, called *pipeline stages*, are separated by register arrays (*pipeline registers*), that save the results of the previous task to be used by the next stage on the next clock cycle (Figure 3.2). In our paradigm the pipeline stages are organized as follows:

- *Instruction Fetch* (IF): In the IF stage, the *program counter* (PC) is calculated for the next instruction which is is read by the program memory for its output to be used in the next stage.
- *Decode Stage* (DEC): Here, we have the decoding of the instruction according to the output of the program memory. Depending on the instruction shifting and registers reads are also performed in preparation of the execution of the instruction in the following stages. Also in this stage we have the resolution of the direct branches that update the PC accordingly.
- *Execution Stage* (EX): In this stage, we have the logic that executes the arithmetic operations and conditional branch resolutions. It is also where a memory address is calculated in when a memory access is required.
- *Memory Stage* (MEM): This is the stage wherein the data memory resides and where memory operations are executed.
- *Write Back* (WB): Here, the execution of an instruction concludes by writing back to the RF any value that was produced by the operation and, therefore, needs to be saved.

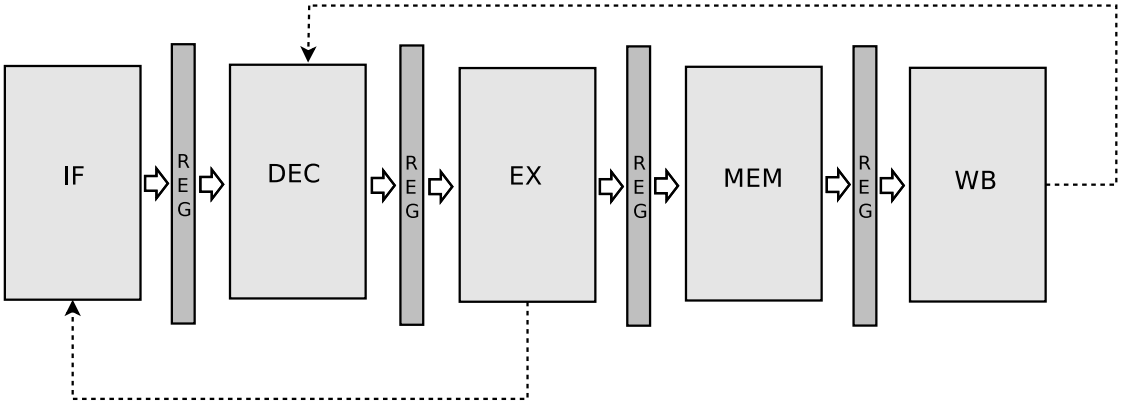


Figure 3.2: The typical 5 Stage Pipeline Datapath, which is also the structure of the Baseline processor for this thesis

These stages comprise the complete datapath of the baseline processor. In theory, it can provide a throughput of completing one instruction every cycle. Yet, in reality, this is not the case. The parallel execution creates a number of complications in execution that hinders instruction throughput. These complications are knows as hazards.

3.2.3 Pipeline Hazards

There are a number of occasions that the parallel execution of the instructions in a pipeline creates situations that hinder the uneventful completion of operations. The solutions to these are not without cost and they usually tend to decreased the speed-up a pipeline could theoretically achieve in terms of throughput. These are identified as *pipeline hazards*. They can be divided into 3 types:

- *Structural Hazards*
- *Data Hazards or Data Conflicts*
- *Control-Flow Hazards*

Structural Hazards occur when there is competition between the parallel operations for the use of a hardware resource. For example, both the IF stage and MEM stages access the memory if both the program and the data are stored in the same main memory. Similarly, the RF is used by both WB and DEC every cycle. These conflicts are easy to solve either by time multiplexing, as it happens with the RF, considering a module as 2 separate modules, like it can be done with the program and data memories, or by outright replicating the resource so all competing stages can have one in their disposal.

Data conflicts on the other hand are quite more difficult to solve and cannot be tackled without cost in performance. Data conflicts occur because instructions have the ability to start execution before the previous one completes. In this case, there is a chance that an instruction requires a value that was produced by a previous operation (either arithmetic or memory access) and is not yet written back to the RF by the time the

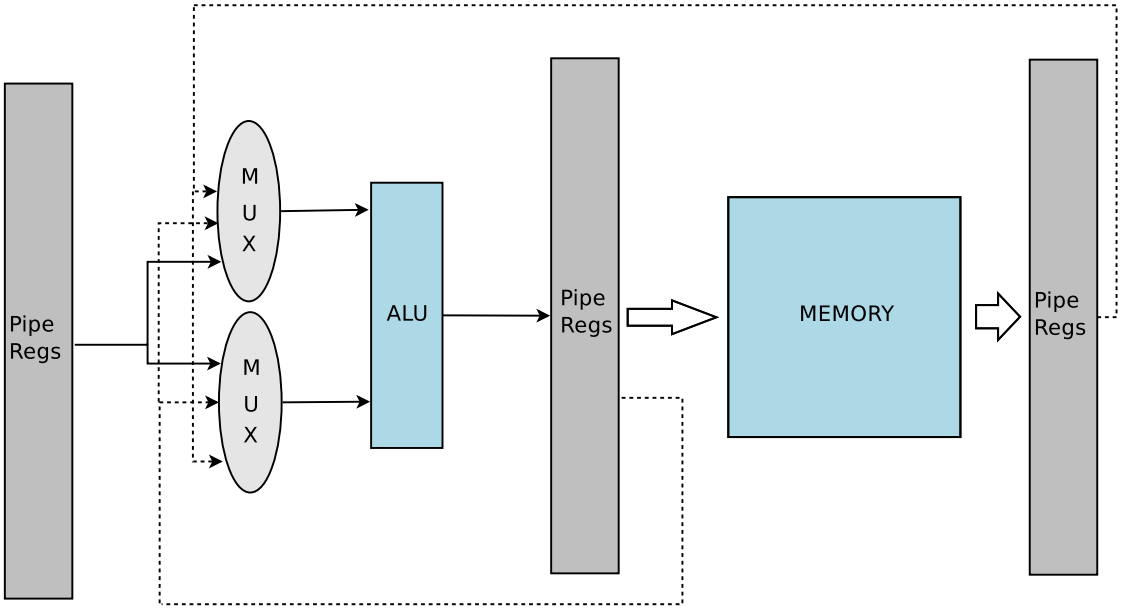


Figure 3.3: The typical bypass mechanism of a 5 Stage pipeline datapath

instruction reaches the DEC stage. There are 2 ways that this can be solved on a typical 5-stage pipeline. Either *Stalling* the pipeline when a conflict occurs until the required value is written back or *Bypassing* the normal progression of execution and forward the needed value from the latter stages to the instruction that requires it (Figure 3.3). Obviously, stalling has important cost when it comes to throughput. Bypassing, on the other hand, does not, but there is in the 5 stage pipeline a case of data conflict that cannot be solved by bypassing. That is, if an instruction needs a value in the EX that is provided by the memory but has not yet been produced by the MEM stage. The more stages a pipeline has the more possibilities of data conflicts emerge and greater cost is paid to address them.

Control Hazards occur in case of a branch. Before a branch is resolved, other instructions enter the pipeline. This means that it is possible for unwanted instructions to be executed since their will enter the instruction pipeline before the actual control flow of the program is known. The usual solution is for the datapath to consider always that a branch is not taken, and so no change in the control flow occurs, and continue to fetch instruction as normal. If the branch is truly not taken, then there is no harm done in the execution of the program. If, on the other hand, there is a branch *mis-prediction*, the instructions that entered the pipeline before the branch resolution need to be canceled by *flushing* them from the pipeline through reset signals to the pipeline regs, a solution that also holds a throughput cost.

Our baseline core includes all of these typical techniques to defend against pipeline hazards with a small difference. It does not implement the stalling, in the one case of data conflict that is unavoidable, in hardware but has its compiler inserting nops in the program, simulating the same behavior. This choice actually creates some complications in tackling the problem of re-designing the architecture for our purposes, as will be

shown, in chapter 4.

3.3 Heuristic Search for Defect Tolerant Adaptive Multiprocessor Arrays

In the problem of defining a design of a processor array that can share its stages between dis-functional cores for the purpose of creating new ones there is an typical problem that arises. Since, there are more than one ways, usually, to reconfigure the array, there is the matter of finding the most efficient way to make the reconfiguration. If it is not chosen correctly, it may result in less efficient designs being chosen and affect the ability of the hardware to gracefully degrade in the presence of faults. The CCA architecture [23], seen in the previous chapter, solves this problem by putting simple constraints on the reconfiguration to guarantee that the rearrangement of stages will not affect the performance to unacceptable levels. Yet these constraints affect flexibility and defect tolerance and might not always lead to optimal solutions. StageNet[15, 16, 17] on the other hand uses crossbars to connect inputs and outputs of the various stages of the array. Since crossbars have uniform delay despite what input or output a crossbar uses, the delay penalty for every configuration is the same. So, a selection algorithm is not needed.

On the other hand, crossbars are expensive in area and performance costs that are present in full even when there are no faults in the array. In our case we, use a reconfigurable network to achieve stage sharing. So the choice of configuration directly affects the design's performance. An algorithm for optimal selection is imperative. This problem was explored by Vasileios Vasilikos in [30], in a work complementary to the current thesis.

That work had the purpose of analyzing the problem and describe it as a *formal computational problem* to ensure its in depth exploration. The next step was to elaborate into the definition of array performance in order to create suitable approximation metrics, in other words *Heuristics*, in order to define the optimality of each solution that results from the computation of the reconfiguration. Finally, the work proposed and implemented a number of *search algorithms* to achieve the function in a way that the computational time/accuracy trade off can be adjustable and evaluates the results.

In the design, for the array to achieve the goal of stage sharing, it is imperative for the stages in the pipeline to be decoupled. The algorithm perceives the structure as a homogeneous array comprised of N processors each having M pipeline stages (Figure 3.4). The algorithm assumes that the architecture cannot use the same stage for the same processor, which is exactly how the architectural design was implemented. So in the end it does a heuristic check for a functional reconfiguration of the array with the goal of best performance. One of the main parameters that has impact on performance is the influence of the interconnect. Since the idea is to place decoupled stages on a reconfigurable network, the delay is not uniform, as it is the case with StageNet. To avoid the wire delays of stage sharing, the wires in the network are pipelined. This introduces, practically, extra stages in the pipeline. In the algorithm, as well an in the architecture, we assume that each time the data flow must travel from a stage

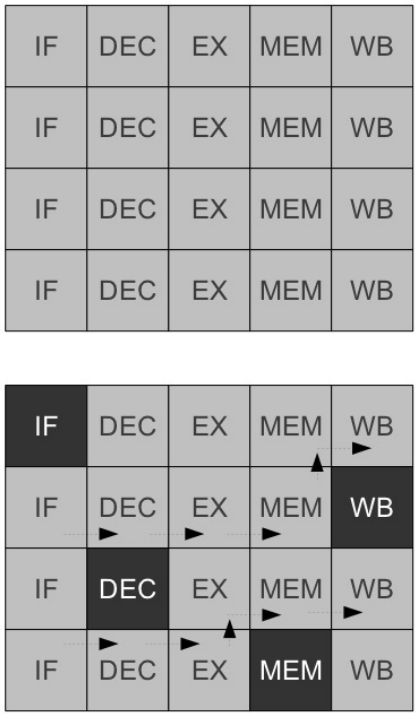


Figure 3.4: How the algorithm perceives a 4x5 core array with and without fault. Each stage is a tile in the fabric. When faults occur (deep gray tiles) the algorithm tries to find suitable re-configurations

of one core to a stage in the immediate neighboring core, one extra pipeline stage is needed to be inserted. Extra stages affect throughput in various aspects, like more data conflicts, slower forwarding etc. These will be analyzed further in the architectural implementation. So one major aspect that the algorithm takes into consideration is the minimization of the extra pipeline stages in the interconnect wires.

There are also many other aspects that could affect the correct choice of configurations. Since the algorithm makes no assumptions as to the type of processor or the type of the interconnect, it is designed to include these restrictions that come up from the architectural design, as input parameters. The more accurate these parameters are the more accurate the end computation will be. Some examples of such parameters, that originate from the architectural exploration of the problem, could be:

- Extra stage number upper bound;
- Number of cables between communication paths;
- Minimizing distance between critical stage pairs;
- Wildcard stages that could be implemented with re-configurable hardware;
- Minimizing the area of the re-configurable hardware when in use;

- One directional interconnect wires

Extra stage number upper bound :An upper bound in the total number of inserted stages in a reconfiguration could be useful to determine the demands of the hazard control logic and design it accordingly. Also after some point of inserting stages it is obvious the design will become too slow to justify re configuration.

Number of cables between communication paths: A parameter that should be taken into account is the available number of cables in the communication paths of the interconnect, since it could interfere with the availability of some configurations. Feedback cables should also be taken into account as they too require additional cables unless the existing ones are bi directional.

Minimizing distance between critical stage pairs: There are certain stage pairs in the typical 5-stage pipeline that are more critical to be close enough because the delay between them has important effect on the datapath throughput. These are the stages that share feedback. In particular, the EX/IF stage pair that shares the branch resolution feedback and the WB/DEC stage pair with the write-back feedback. Of the two, most important is the WB/DEC pair that is more often used. If the datapath has a bypass mechanism, then an additional feedback exists between the EX/MEM stage pair. This replaces the write-back feedback in importance since, with this bypass, we do not care how long it will take for the data to be written back in the RF.

Wildcard stages that could be implemented with re configurable hardware: Reconfigurable hardware can replace any possible faulty stage and so increases the possible configurations in every case.

Minimizing the area of the re-configurable hardware when in use: It is favorable to use as little as possible of the re-configurable hardware in a new configuration due to its performance and power costs. It would be preferable to avoid the use of this hardware to replace a faulty stage, if such a choice is possible. Also, if more than one possible configuration exist, the one most preferred is the one that places the smallest/fastest stage on the reconfigurable hardware.

One-directional interconnect wires: It is obvious, that if the switching nodes of the network have any kind of restrictions on wire direction there is an impact on the selection algorithm's flexibility. Yet, such a limitation can be enforced if the device has certain area of power constraints.

The work concludes with an extended analysis of the various explored computational methods and makes assessments in terms of computational speed and accuracy. The result of the work contributes greatly to the correct choice of selection algorithms for an actual device, depending on the device's demands in computational speed and accuracy(optimality) of the new reconfiguration after the occurrence of faults. According to this work the the best computational speed can be achieved by a *Greedy* algorithm, in the expense of accuracy. On the other hand better accuracy is achieved by the use of a *Genetic* algorithm. When a middle ground between the two factors is needed, the use of a *Seeded* algorithm or a faster tuned Genetic one can be incorporated.

3.4 Summary

In this chapter, the basic background required for the presentation of our work was provided. This included a small presentation of the DeSyRe project framework in which our work can be linked with. Next, we presented the basic architecture theory of the 5-stage RISC pipeline, that is the basis of our own architecture. Finally, the previous section showed the parallel work completed on the problem of finding a suitable heuristic-search algorithm needed for determining the reconfiguration of the array core in case of fault and its effect on the architectural characteristics. With the necessary background provided, the next chapter is devoted to presenting the design challenges in modifying the typical 5-stage RISC architecture to achieve the ability of stage sharing, the implementation of our design and the array's interconnect implementing the possible reconfigurations.

The Defect-Tolerant Architecture

4

"Imagination is the beginning of creation. You imagine what you desire, you will what you imagine and at last you create what you will."
-George Bernard Shaw

As it was previously mentioned, the subject of this thesis is to develop and evaluate an architecture that can fulfill the functionality needed to design a complete defect-tolerant array based on a reconfigurable network. The design of such a system requires the analysis of the problem, the identification of the challenges and complications such a design would have. Afterwards the implementation of the said architecture and the array's interconnect to produce a functional example of the approach, is possible. We begin with a classical 5-stage RISC architecture, which is an improved version of a demo processor provided by our design tools alongside its respective C compiler, modified to serve as our Baseline core. Later we modify it to create our own version of the architecture capable of handling the project challenges.

The rest of the chapter is organized as follows: Initially we present the main operation concepts of our design and the challenges that stem from them, which need to be addressed by our architecture. These are (i) the method of handling data conflicts, (ii) the stalling scheme in our processors, and (iii) the pipeline flushing mechanisms. The next 3 sections are devoted to presenting the possible solutions to these problems and the solutions that were eventually implemented in the final design. The last section addresses the problem of the implementation of the array interconnect and describes a complete example of a 4 core array made using our architecture.

4.1 General Design Premise

Going back to the two related works done on the same premise-the StageNet [15, 16, 17] and the CCA [23]-we can see a number of characteristics. On the one hand, the CCA achieves its goals by a simple and effective method but introduces constraints in the reconfiguration of the fabric to prevent significant performance loss. This reduces flexibility and the potential defect tolerance, as a whole. StageNet on the other hand, solves this problem with the use of crossbars. Thus, by decoupling the stages and connecting them through crossbars, it achieves better flexibility at the cost of crossbar use that affects performance, area and power consumption. The challenge we force on is to provide the same flexibility as StageNet but with a more efficient interconnect that has scalable performance to the number of present defects.

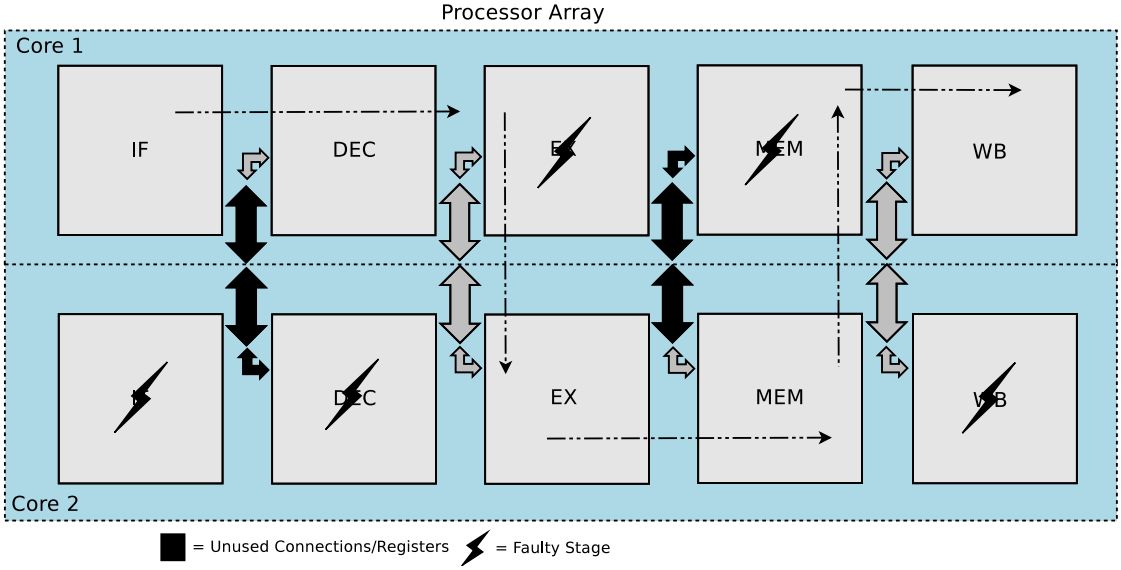


Figure 4.1: Basic visualization of redirecting dataflow by reconfiguring interconnect

4.1.1 A More Efficient Interconnect

The main problem is how can we design a more efficient interconnect suitable for the design concept. One interesting alternative to the crossbars for connecting every stage in the cluster is to use reconfigurable interconnects. When there is a defect, the interconnect just re-configures the system, depending on the area of the defect, to create new cores from the spare parts of the dis-functional processors (Figure 4.1). This has the advantage of not paying the full performance cost when everything works well in the cluster, like in the StageNet. In the presence of a fault, the device is forced to be inactive, and a system that can perform the heuristic search as described in 3.3 (also in [30]) can check if there are enough spare parts from the faulty core(s) to create a new ones by leading the dataflow through the spare functional stages. In other words, extra cores shall be created (adding to the performance of the array) that would not normally exist in the presence of defects, just by tweaking the controls of the interconnect switches. In case there are more than one possible solution, the heuristic algorithm can choose the best according to its input parameters.

4.1.2 Design Improvements and Requirements

With the above initial design in mind we can begin looking for problems and proposing possible design improvements. An obvious concern one could have for the interconnect would be the wire delay. The time that it takes for the data to travel through the interconnect could significantly influence the critical path of the design and increased the cycle time of the processor(s). A good solution to this would be to pipeline the interconnect (Figure 4.2). This would ultimately mean that extra pipeline stages would be inserted in the pipeline for this design to function with substantially lower delay overhead. Actually, through this idea we attempt to address a very important aspect:

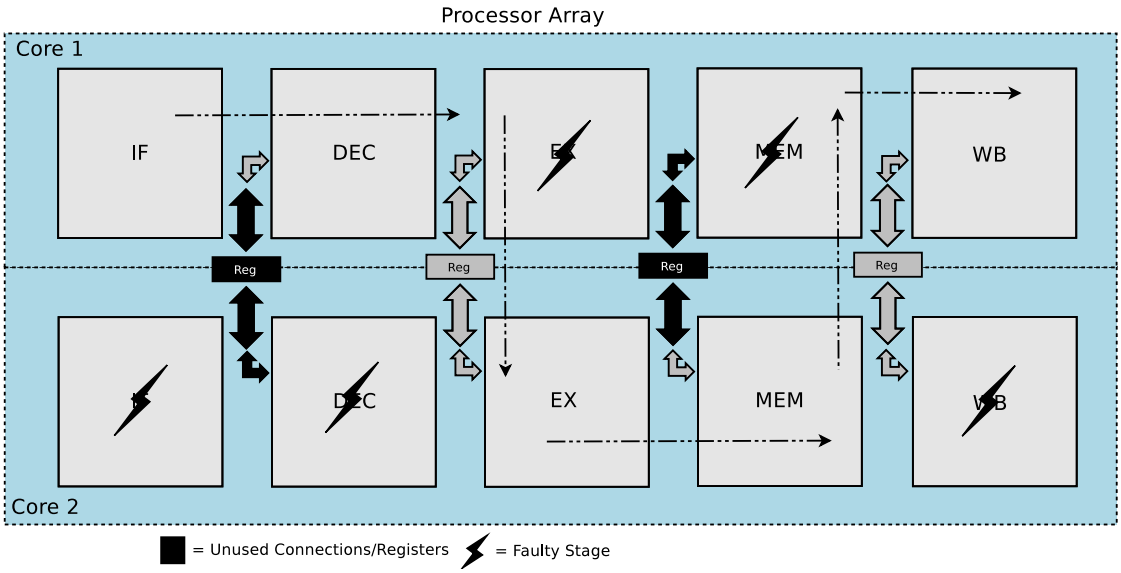


Figure 4.2: Basic visualization of redirecting dataflow by pipelined reconfigurable interconnect. The number of stages in the newly created core are 7 instead of the original 5.

Whether by using such a pipelined interconnect in our array we can keep the operation frequency overhead in acceptable levels, paying just a reasonable performance cost for the presence of the interconnect switches and the interconnect pipeline registers in the critical path.

We specify that each core in the array has its own *plane* within which the wire delays are in acceptable levels, practically meaning that feedbacks between stages of the same core do not need to be pipelined. On the other hand, we assume (perhaps pessimistically) that the wire delays for data to travel from one core's plane to its immediate neighbor need one extra register level to avoid significant wire delay penalty. The further the data need to travel from their original core, the greater number of extra empty stages (or *bubble stages*) need to be inserted into the respective pipeline. The immediate effect of this method means that what we eventually get is a design with a variable number of pipeline stages requiring from the final architecture to be able to cope with this aspect. Such a system is required to include a number of desirable attributes in order to be functional as follows:

- The reconfiguration functionality is transparent to the application level. The system needs to run with exactly the same application binaries for every configuration so it won't have to recompile the applications in case of a re-configuration. As a consequence the complier must not do any kind of optimizations that are affected by the number of the pipeline stages in the datapath. Any optimization is preferable to be done in hardware.

- The end design must avoid global signaling as much as possible. Thus, the system must not require global control (such as stalls and flush signals) since is it impossible

to send global control signals without severely lowering clock frequency. All control should be localized and every stage decoupled. It is preferable to design a local control scheme for the pipeline stages of this architecture.

- A desirable attribute is also that re-programmable/configurable hardware must be limited to the interconnect logic which, basically, means that only the re-configuration of the interconnect should be required to change the dataflow and no change should be needed in the core architecture.

4.1.3 The Stage Issue in Detail

The main challenge that needs to be addressed in our design is the complication of the variable number of stages in the pipeline that are created by pipelining the wires in the re-configured datapath. This complicates many characteristics of the pipeline functionality.

The control logic of the hazards itself, for example, needs to adapt in order to accommodate for the variable number of stages. Since there are no crossbars here, the issue that could affect performance overhead is not so much the number of inputs going into the interconnect but the distance each wire has to travel. Since the main feedback loops of the WB and the branch resolution cannot be removed, maybe it is possible to place the hazard control outside the pipeline, in re-configurable or other fashion for it to adapt to the variable stages, without much cost. Otherwise the possibility of local control needs to be considered. An other question to be considered, would be whether to implement data forwarding instead of just stalling in a data conflict, which will benefit the performance. Depending on the architecture and the interconnect it might have a reasonable cost on area and complexity.

In case of forwarding not being applicable, when considering data hazards specifically, then the scoreboard/status register scheme in the StageNet architecture could also work here since the scoreboard only sees what comes into the issue stage to judge about possible hazards and is oblivious to the number of stages (local control). Yet, this might require rearrangement of the Baseline pipeline to a design that has a separate DE and IS (issue) stage and possibly a combined EX/MEM stage, as in the StageNet. Merging already heavy stages, such as the EX and MEM, together is not a good idea. Moreover, there is the problem of preventing the scoreboard from filling completely. Furthermore, the fetch stage needs to be stalled anyway which means that additional wires are needed from the IS to the IF stage. All this without beginning to consider the servicing of control hazards and the fact that it is preferable to avoid global stall and reset signals.

Besides the above complications one immediate change that was done in our design, due to the variable stage issue, was in the Baseline compiler. As mentioned previously the Baseline core simulates the stalling in a data hazard by detecting the conflict at compile time and adding nops in the program. Since there is no way for the compiler to know the number of pipeline stages beforehand, this inevitably must change. In our architecture, we modify the original compiler to just do its basic function and moving all necessary hazard control functions in hardware. As such, we are certain that the compiler will not take an action that would prevent the application from running with

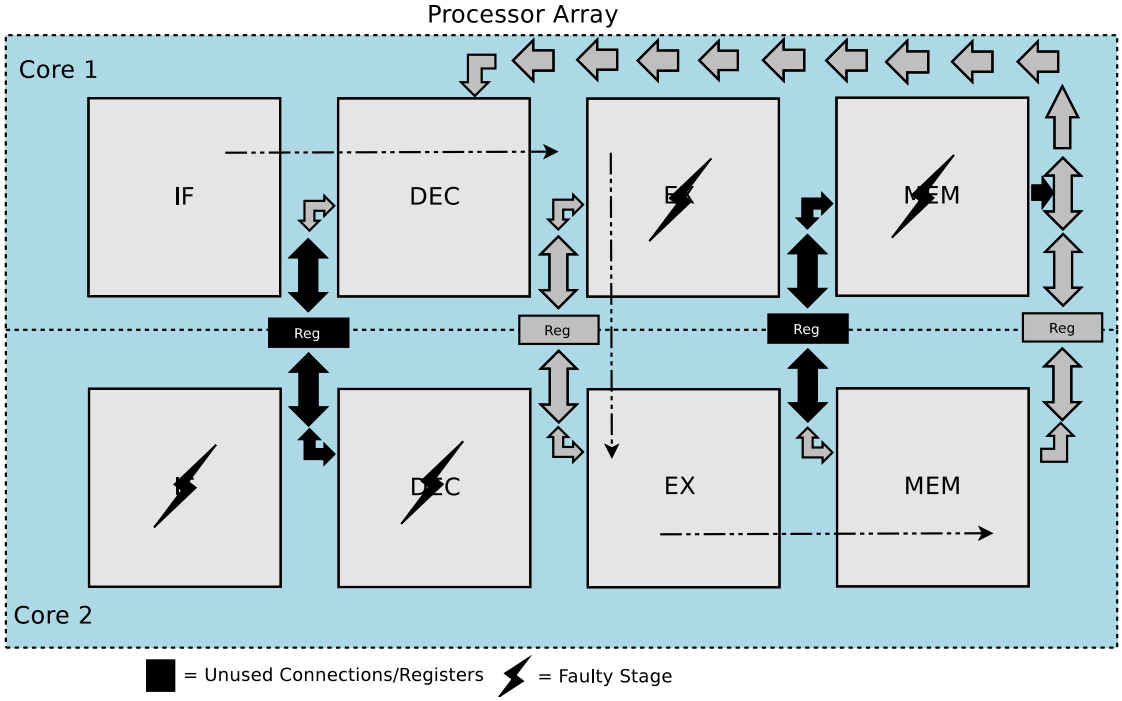


Figure 4.3: Basic visualization of redirecting dataflow by pipelined reconfigurable interconnect with the WB merged in the DEC stage. The number of stages in the newly created core is now 6.

the same binaries on every possible configuration. Obviously, the stalling function must be incorporated also in the hardware.

Having in mind that the more stages we have in the pipeline the more data conflicts occur, one immediate optimization was also done in the architecture. The incorporation of the WB stage in the interconnect. The Write back even though logically does separate actions, physically it shares the bulk of its logic with the DEC stage, practically the RF. Besides that, the only dedicated hardware to the WB is then a few multiplexors. If these were placed physically to the Decode, the only thing that is left in the WB are wires sending the data to the RF in the DEC stage, meaning the write-back feedback. Since it is just wires these can be easily included in the interconnect. This enables us to physically include each stage with its latter pipeline registers and organize the modules of the processor to 4 stages (Figure 4.3). The logic tasks are still the same but we can now approach the architecture as a 4-stage pipeline.

As a whole, the main issues stemming from the variable stage characteristic and need to be addressed to make our idea a functional design are three:

- *Data-Hazard resolution*
- *Control-Hazard resolution and the Pipeline-Flush Scheme*
- *Absence of Global Stall*

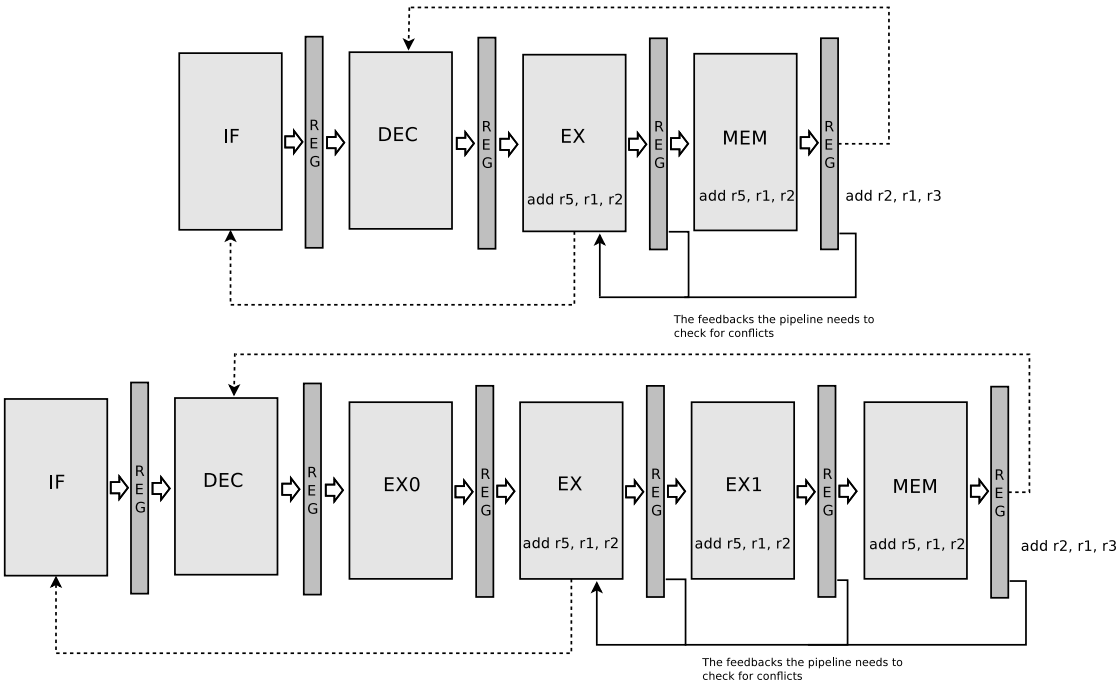


Figure 4.4: Data conflict feedbacks in a normal pipeline and a reconfigured one with 2 extra empty stages (EX0 and EX1)

4.2 Data-Hazards Resolution

Since we have a variable pipeline, the time for a value (produced by an instruction) to be committed is not fixed. By inserting more stages in a pipeline the direct result is that the instruction window in which a data conflict may occur, increases. More precisely, for every single pipeline stage inserted due to reconfiguration, we have to be able to check every new stage for dependencies in order to detect all data conflicts. Moreover, if the pipeline incorporates value bypassing to service the conflicts, a feedback line is needed from each new stage to the stages that might require a forwarding value, mainly the EX and MEM stages. For example, let's consider a very simple sequence that produces a conflict:

add r2, r1, r3
add r5, r1, r2
or....
add r2, r1, r3
any instruction
add r5, r1, r2

As they are, in the typical 5-stage pipeline, these would create a data hazard on register r2. The architecture would need to check after EX and MEM stages to detect the dependency and forward the value in case bypassing was implemented. In case the

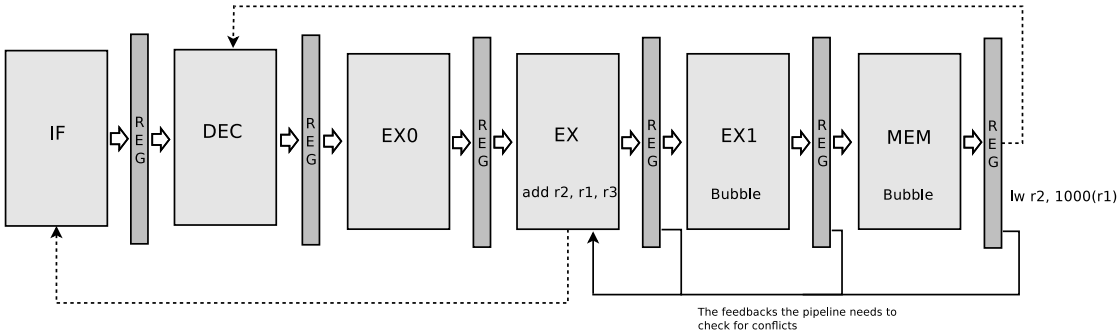


Figure 4.5: Data conflict requiring bubble inserting in a reconfigured pipeline with 2 extra empty stages (EX0 and EX1). Here 2 bubbles need to be inserted (by compiler inserted nops or pipeline stalling) for the add to wait for the value to be produced.

two *add* instructions were separated by more instructions though, the value of r2 coming from the first add would be written back in time and there would be no data hazard. Let us consider now a case of reconfiguration where the EX stage is damaged and the core used the EX stage of an immediate neighbor. This would insert 2 bubble stages in the pipeline: One for the data from the DEC to travel to the neighboring core's plain and one more to come back to the original core's MEM stage (Figure 4.4). In the case of the previous add instructions, with the extra stages even this sequence now would produce a data hazard:

add r2, r1, r3
any instruction
any instruction
add r5, r1, r2

The instruction window of conflicts has increased by 1 and will increase every time an extra stage is inserted after the EX stage in a configuration. This is only for this specific type of conflict. The hardware must be able to handle all of them for all possible array configurations.

Additionally, there are also the conflicts that cannot be handled with bypassing and require stalling; these create additional issues. This happens when the conflict exists between a memory operation and a following arithmetic operation. The value needed in the arithmetic operation is not produced yet, so we cannot use bypassing. For the pipeline to handle this either the control unit in hardware must stall the arithmetic operation or the compiler must schedule a nop between the two instructions, simulating the same action. In a possible reconfiguration, for every stage inserted between the EX and MEM stage, besides the fact that the conflict instruction window increases, an additional stall is needed to make sure the memory produced value is ready. Since the number of extra stages depends on the distance between the shared stages, the number of stalls needed is variable too, just like the number of stages (Figure 4.5).

There are some solutions one can consider to implement data-hazard resolution in a

processor in this fashion:

- *Remove data forwarding altogether and address the conflicts through stalls*; or
- *Use a Reconfigurable control unit*; or
- *Use a Programmable control unit*; or
- *Use a mechanism that includes a kind of Conflict Table to support servicing data conflicts, oblivious of the number of stages*; or
- *Pipeline State Saving*

4.2.1 Removal Of Forwarding

One idea that could simplify the problem is to abandon forwarding support and have the data conflicts serviced using stalls. This could be done using a scoreboard that detects the conflicts, keeps the instructions and stalls the pipeline until the sources are ready to use. This can be done locally and, so, in our case it is not influenced by the variable number of stages in the pipeline. This could also address the problem of the global stalls since only the stage that includes the scoreboard is stalled. But to really achieve this one needs a way to make sure that, for the worst case of all possible configurations, the scoreboard is big enough so it will not fill up and create a need to stall previous stages. This can also be achieved through the compiler (by rearranging instructions and adding nops) and even by macro instructions, as in the StageNet, to ensure that the conflicts in the instruction flow are not greater in number than the scoreboard can handle. A large enough scoreboard to handle all possible conflicts on its own is considerably costly. The use of compiler optimizing the instruction flow is also not applicable because of the variable stages, unless the compiler assumes the maximum number of stages, issuing larger delays when not required. Finally the use of microcode is costly and quite complex. With that being said, the main disadvantage of the removal the forwarding technique is, of course, the performance cost in execution cycles.

4.2.2 A re-configurable control unit

Another way to possibly address the issue is a reconfigurable control unit that changes according to each new sharing configuration. The possible configurations can be predetermined since there is an upper limit in the number of inserted stages (pipelined wires). Yet, this solution is desirable since it is preferable to keep the control unit hardwired through the different configurations as part of the actual datapath implemented in ASIC and have only the switches re-configure the data flow for the creation of the new configuration. First of all, a control in re-configurable hardware is slower, affecting operating frequency considerably, and also power hungry. Additionally systems with different number of processors also have different predetermined possible configurations, meaning that the control logic needed to be redesigned explicitly for every possible number of processors in a system. More importanly this solution is still uses global control signals, which their use is not preferable.

Dest	Value	V	R
2	345	1	0
3	576575	1	1
6	235266	0	1

Figure 4.6: The aspect of a Conflict Table

4.2.3 A re-programmable control unit

A similar approach to the reconfigurable unit, but this time all possibilities are pre-designed into the control unit according to the possible configurations the datapath can have. The functionalities can be chosen according to a configuration code that can be given to the control unit from the general control of the system, using the heuristic search algorithm of section 3.3. This way the unit does not have to be in reconfigurable hardware. This certainly has better performance and power efficiency than the control with re-configurable hardware. Yet, there is still the same issues of explicit control logic for every possible number of processor in a system and that this is still a solution implementing global control.

4.2.4 A Conflict Table

There is, yet, another solution that could be used to solve the problem, similar to what the StageNet used. A Table or scoreboard-like design implementing localized control that could save the destination and result of every uncommitted instructions that produces a result in the pipeline. Instead of checking for conflicts in every stage and forwarding directly each value from a stage when it is needed, the conflicts can be checked and forwarded to and from the Table without the need to fetch them from other stages (on most occasions) using long wires, including the conflicts created because of possible bubble stages could be inserted (Figure 4.6).

When an instruction, that has to write its result to the Register File (RF), is decoded, the destination is immediately written in the Dest field and the valid bit (V) becomes 1. When a value is produced, either by the EX or the MEM stage, it is also immediately written in the value field and Ready (R) becomes one. When a result comes from the WB and is written in the RF, the V bit becomes 0 again. Every instruction that needs a value from the RF, checks first if its source is in the Table and if it is valid. In case it is valid (so there is dependence with a previous instruction), it checks the R bit. When it is ready, then the instruction uses the value in the Table instead of loading it from the RF and continues to execute. In case it is not ready, then the instruction stalls until the value is produced and written in the Table. The feedback lines to the Table can be pipelined if needed in case of stage sharing between cores, that would require longer wires, to keep the frequency to acceptable levels. This way the design can support forwarding without

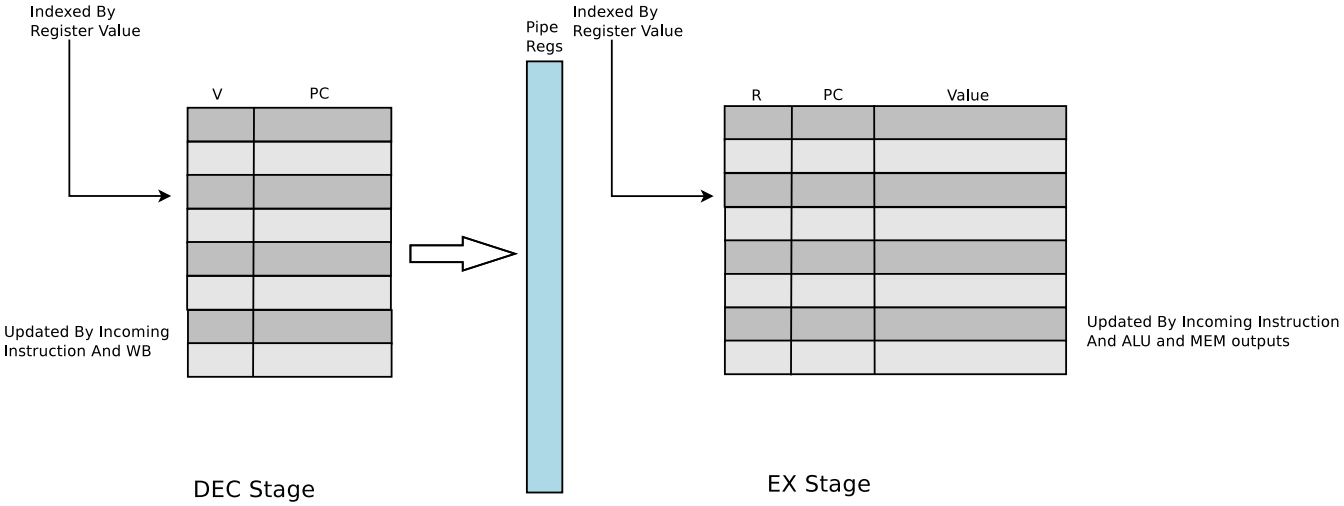


Figure 4.7: The aspect of a Conflict Table with the valid field in the DEC stage

checking pipeline registers and can work while being oblivious to the number of stages in the datapath. In every configuration the functionality is the same. The Table should be large enough to service the maximum number of data dependencies that can occur in the worst case configuration and it is dependent on the maximum number of stages. Here of course we still have the problem of global stalls that must be addressed.

A good place to integrate the table would be the EX stage since results of the ALU that are to be written in the table, would not be forced to pass through a slow (and perhaps pipelined) feedback loop. Also, the fact that the table would be in the same stage where the branch resolution is calculated simplifies the flushing scheme (explained below).

Yet, there are a couple of issues with this approach that need to be solved for it to become a functional design. First, there is the problem of indexing. It is not a good idea to check every single value in the table to find a possible dependency. An obvious solution is to have a 3-field table (value, V, R) that is equal to the number of registers in the RF and indexed by the register number. This way the presence of a dependency can be found quickly according to the information coming from the Decode stage. This also eliminates the need for an upper limit in the stage number mentioned earlier since there is the capability of storing every valid register value at each time. Therefore, the instructions that are halted by a dependency can be always serviced with the current value of the register they would need.

Another issue is the update of the V and R bits on the table. The information that a register value is written back to the RF, so it is invalid in the conflict table, comes to the decode stage. With the table needing to be updated while residing in the EX stage there would be a need for a feedback loop from the WB to chance the V bit. An extra feedback line would be wise to be avoided, since it can issue extra delays in execution cycles. A solution would be to move the V to the Decode stage and send its needed value to the table in the EX stage through pipeline registers (Figure 4.7). This way it can be updated immediately when each instruction is decoded or written back.

Another problem is the matter of distinguishing whether a register value in the RF is valid if there are 2 instances of that value in flight in the pipeline (if the table does not match the size of the RF). For instance, 2 consecutive adds with the same destination register: When the first add is written back, it will chance the V bit and show the register value as valid which would not be the case until the second add is written back leading to false execution. A solution to that would be to also keep the PC of each instruction beside the valid bit. When an instruction is written back, it validates the value only if its PC matches the one of the last instruction that changed the register value.

Yet another matter with this solution is updating the R bit. The only case when a value that is needed by an operation is not ready in the conflict table is if it is produced by a load instruction, since the table is placed in the EX and the arithmetic operations write their values there immediately. So, a natural solution would be to change the ready bit to 0 each time a load instruction enters the EX stage. Here, the PC also should be saved for the same reason as for the V bit. The R bit becomes ready when the correct value comes from the feedback between the EX and MEM stages.

Lastly, with this solution you cannot service St/Ld dependences as one could by traditional bypassing without having to pay with stall cycles. Instead, here this is treated in the same way as a dependency between a load and any other arithmetic operation. This could be addressed by an additional table in the MEM stage, but it is a cost in logic and complexity that may not be worth the benefits.

Finally, the presence of a scoreboard-like structure in the DEC stage introduces the problem of having to partially update it in case of a branch mis-predict. In that case there will be entries in the table that need to be flushed because of the mis-predict and entries that preceded the branch and must not be erased. Servicing that would increase the complexity even more.

4.2.5 Pipeline State Saving

The previous conflict-table scheme has the disadvantage of being quite expensive and complex to implement. What is implemented, at a minimum, is essentially a second RF plus additional fields for determining the correct order and validity of the values. Moreover, the need for a part of the table in the Decode stage, also forces a change in the reset scheme in the case of branch mis-prediction, besides adding logic to the Decode Stage. The table tracking the existence of the values still traveling within the pipeline in the case of a mis-prediction may contain information that needs to be dropped too, if they are referring to instructions of an invalidated instruction flow. As a consequence, even more logic is required for correct functionality.

A different way to keep track of what is going on in the pipeline and providing the needed values is to save copies of the essential information of the state the pipeline registers of latter stages. Information that might contain values needed to resolve a possible conflict. These values can be saved in the stages where an executing instruction might require an uncommitted value, eg EX and MEM stages (Figure 4.8). By storing this information in a register array using a FIFO-like scheme for inserting/removing entries (or FIFO buffers), an exact instance of the next pipeline stages is saved, which can then be used to provide the requested value or, if it is not available, wait until is it

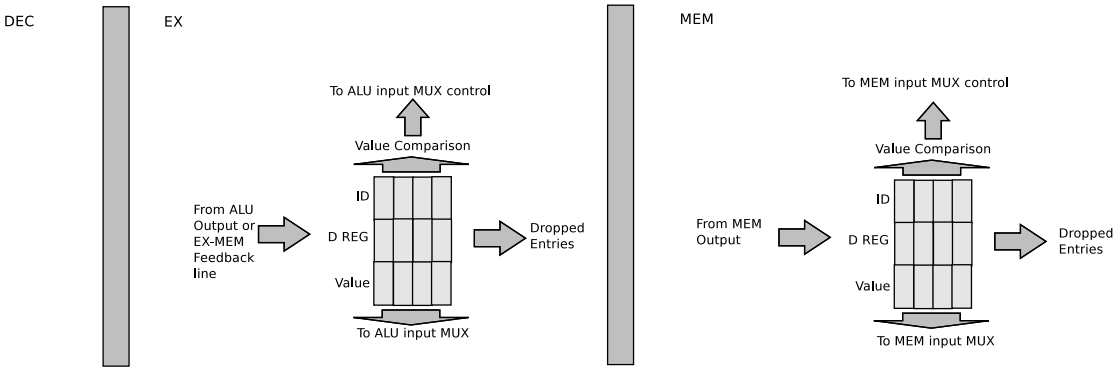


Figure 4.8: General Premise of Pipeline State Saving.

provided though a feedback line.

In general, one structure that could be used includes FIFO buffers that keep 3 types of information about the instructions traveling in the pipeline:

- The instruction type which can be modeled by a 2 bit ID. There are three types of instructions (using a 2 bit field we have one value to spare) that affect the conflict resolution scheme that have to do with where a new value is produced. One type are the instructions that do not produce a result to be written in the RF and we can ignore when it comes to data conflicts. The other instructions either produce a value in the EX stage (arithmetic operations) or in the MEM stage (load operations). According to the ID, the logic can recognize whether an instruction (i) produces a value from the ALU, (ii) is a LD or (iii) is an instruction that does not produce a value to be written in the RF.
- The destination register (if applicable).
- The actual value if, of course, makes sense for the instruction type.

The size of the FIFO buffers can be determined from the number of processors(N) in the processor array since the maximum number of *bubble stages* that could be inserted in a possible re-configuration is determined by N. The greater the number of processors, the greater is the maximum distance wires must travel for the two farthest apart processors to share resources, and thus the higher is the maximum number of bubble stages that could be inserted in the pipeline. The FIFO buffers must be deep enough to store enough values for the worst case configuration, in which we have the maximum number of stages in the pipeline.

Under the worst case scenario, a fault could be in the MEM stage and the stage that replaces the damaged stage is taken from the core farthest away form the faulty one. In this case the highest number of extra stages will be inserted after EX, which would lead to an immediate increase in the dependence window.

The bubble stages are inserted in the path after both the original EX and new MEM stages with a new configuration replacing the faulty part (we assume, as always, that the WB is in hardware terms practically part of the interconnect). By having 2

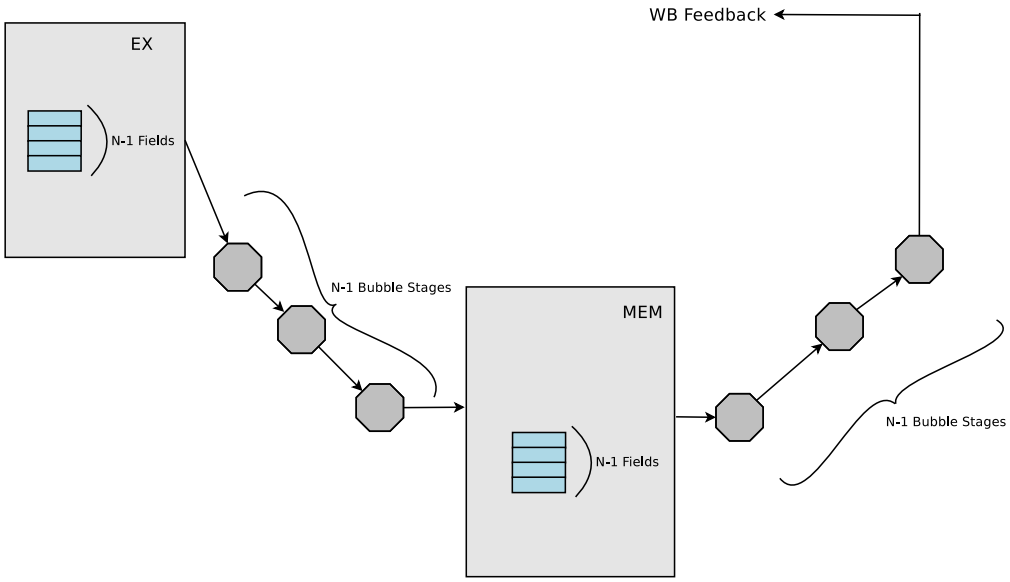


Figure 4.9: Representation of the reconfigured array with the 2 (N-1) state saving FIFO buffers visible.

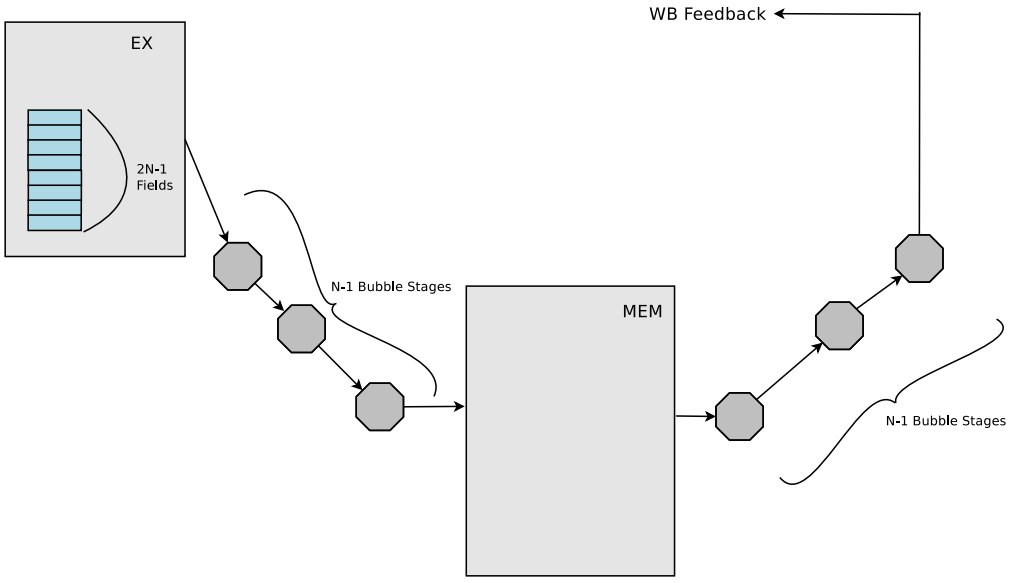


Figure 4.10: Representation of the reconfigured array with the 2 (N-1) state saving FIFO buffers merged into a (2N-1) buffer in EX.

arrays in the EX and MEM stage the bypassing of the values can be serviced properly (Figure 4.9). Here, the logic in EX provides the values that instructions need and are still traveling in the bubble stages between EX and MEM stages and the logic in MEM services the dependencies between the memory operations and instructions in the bubble stages between MEM and WB.

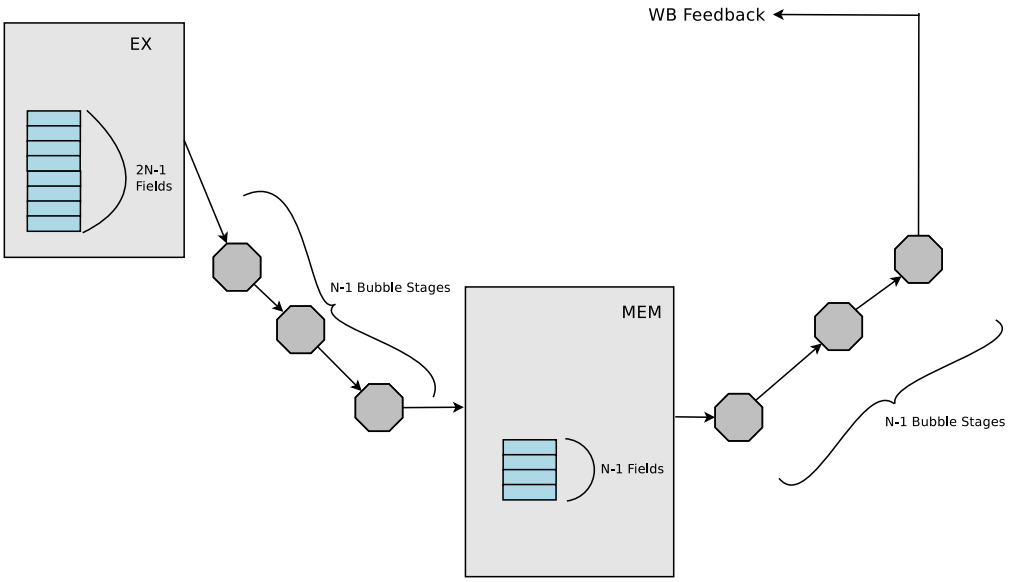


Figure 4.11: Representation of the reconfigured array with the 2 state saving FIFO buffers visible. A (2N-1) buffer in the EX and another (N-1) one inserted in MEM for additional dependence coverage.

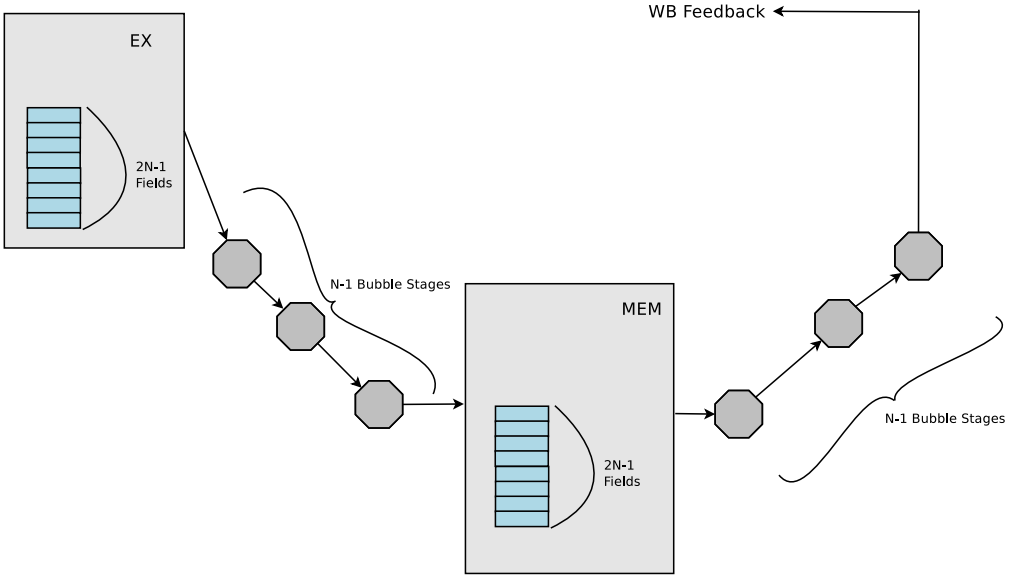


Figure 4.12: Representation of the reconfigured array with the 2 (2N-1) state saving FIFO buffers in EX and MEM.

With this design, though, there is a major limitation. If there is a possible dependence between an arithmetic operation and an instruction that has moved past the MEM stage, and thus the entry in the FIFO buffers is dropped, the only option to service the conflict is to receive the needed value from a feedback line coming from MEM. One quick and

better solution, that would have no cost of logic in total, would be to actually move the MEM FIFO structure to the EX stage. Practically this means implementing only one FIFO buffer with 2N-1 entries (not 2N-2 since here we also have to account for the MEM stage besides the bubble stages) (Figure 4.10). Although this design is not useful for dependencies that are created by values coming from the memory (values produced by loads) every other dependency can be resolved in the EX stage immediately.

In case of a dependency between a load operation and any other instruction, the other instruction is forced to wait in the EX stage until the value is available in a feedback line. The instruction stalls the pipeline until then and continues normally when it is available. We can use the spare instruction ID value in the EX FIFO buffer to determine when a load instruction has its value available or not yet arrived by the MEM stage. This process covers dependencies between store and load instructions that in a traditional bypassing scheme could be resolved with bypassing and not by stalling the pipeline. By adding some more logic there is a way of resolving also these dependencies and improving the mechanism: Putting an additional FIFO buffer with N-1 fields in the MEM stage used specifically to resolve dependencies between store and load operations (Figure 4.11).

The store/load operation dependencies are, now, detected in both stages. In case the difference between them is equal or less than N instructions then the dependency can be resolved by the FIFO buffer logic in the MEM stage and the pipeline can continue without stalls. When, on the other hand, the difference between them is larger than that then the buffer in the MEM stage cannot resolve the dependency. The needed value will have already been written back and dropped by the MEM FIFO buffer by the time the store instruction arrives to the stage to be executed. In this case there is no way to retrieve the needed value from the RF. As a consequence, the store has to stall earlier in the EX stage and wait for the required value to come from a feedback line.

This issue can be resolved by increasing the depth of the FIFO buffer in the MEM stage to the same size as that of the EX stage buffer (Figure 4.12). This way the mechanism can service all ST/LD dependencies, as the MEM FIFO buffer still saves the needed value in a possible conflict even after it is written back. The additional cost in cycle time and logic can vary since, in this mechanism, the logic in the EX stage does not have to detect these dependencies and can be simpler, but the FIFO buffer in the MEM stage has to be doubled.

Feedback Functionality

It is apparent from the previous scheme that there is an additional feedback needed, besides the WB, to service dependencies: A feedback from the MEM to the EX stage to send the needed values that are produced by the memory. Since we consider the WB to be part of the interconnect between stages, what needs to be addressed is how this other feedback will send the values needed for the bypass to the EX stage and if the delay that this feedback introduces in case bubble stages are present, will create problems for the correct functionality, therefore necessitating the addition of more logic.

As it holds in all cases, the 2N-1 buffers are actually deep enough to compensate

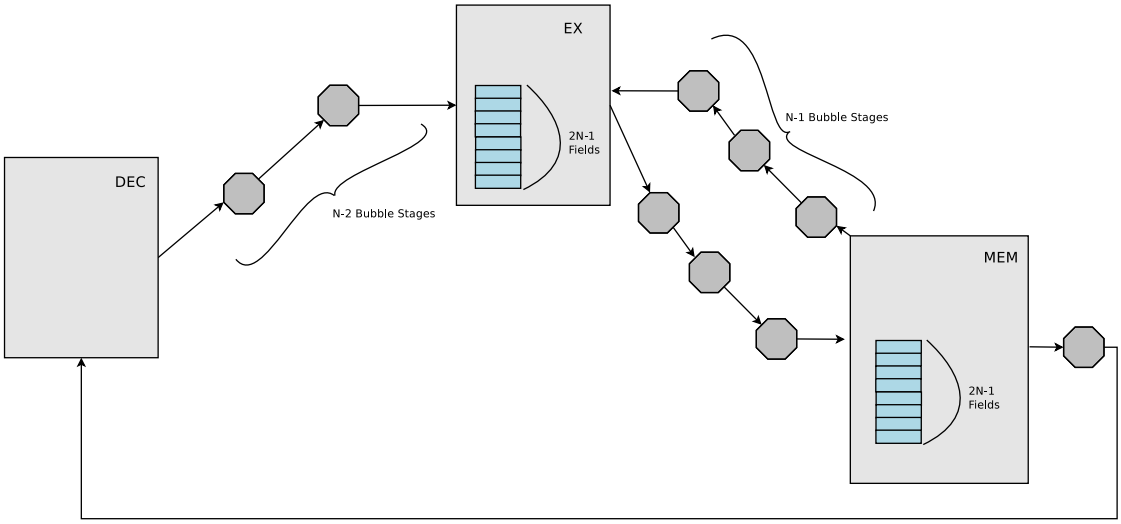


Figure 4.13: Representation of the reconfigured array with the 2 2N-1 state keeping tables in EX and MEM in the extreme case of maximum inserted stages in the areas that influence EX/MEM feedback delays

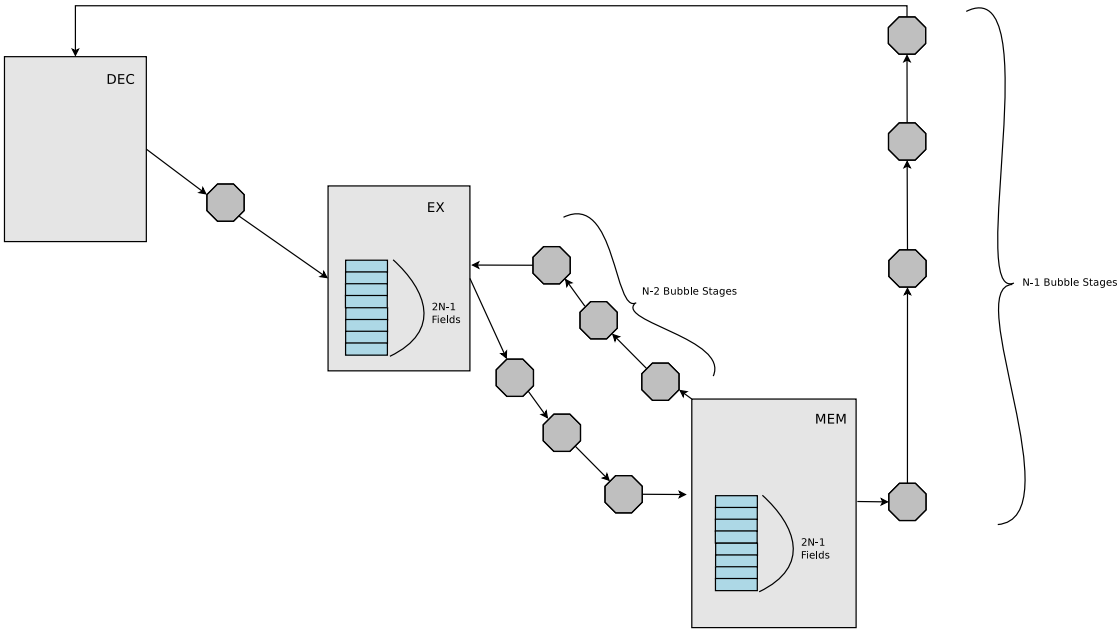


Figure 4.14: Representation of the reconfigured array with the 2 2N-1 state keeping tables in EX and MEM in similarly extreme configuration case as Figure 4.13

for the feedback line delay so that every entry saved in the FIFO buffers will not be dropped before it is required. The maximum distance the feedback must cover is the same as the distance between the EX and MEM stage. To avoid higher frequency latencies the feedback line is pipelined with the same number of stages as between the

EX and MEM stages, which can be maximum N-1 bubble stages. The FIFO buffers are stalled when their respective stages are also stalled when a dependency is detected. The above mentioned MEM/EX feedback line will send the values produced from the memory immediately back to the EX to be saved in the FIFO buffer in case they are needed. The buffers have 2N-1 entries when the maximum steps between two processors sharing resources is N-1. This means that the maximum distance which can exist in the pipeline between two conflicting instructions before a dependency is detected and the buffers stall, is exactly 2N-1 stages. As a result the buffers are deep enough so that they keep the entry of the needed value long enough for every future in flight instruction that has left the DEC stage, and thus not being able to take their sources for the RF, to find its needed sources in a FIFO buffer entry before they are dropped.

For example consider some extreme configurations. Figure 4.13 depicts one of the occasions on which the maximum possible bubble stages are inserted in the pipeline and the buffer requires to guarantee correct functionality. The greatest risk of dropping a value would arise if in this configuration an instruction that is at the DEC stage has a dependency to a load instruction that is at the same time present in the bubble stage between the MEM and the WB feedback. In this is the case, the source value is also, at this time, in the first bubble stage of the MEM/EX feedback line and the repective entry in the EX buffer in the Nth position. As the instruction and value travel though the pipeline and the entry in the EX FIFO buffer progress downward, instruction and value will reach the EX stage at the same time, while the entry in the FIFO buffer will be in 2N-2 position. From there the instruction can take the correct value as it will be written in the entry of the buffer. In the next cycles this value is dropped but it does not create any problems since any new instructions, after the first one mention above, could receive this value from the RF itself.

In the other extreme situation where we have more bubble stages after the MEM stage and only one between the DEC and EX stages (Figure 4.14) the buffer with 2N-1 depth has also no problem in keeping the values needed without dropping one to soon. This time actually the buffer itself is stalled along side the EX stage when a conflict is detected and so an entry representing a data conflict will not drop lower than the N-1 entry in the buffer before the conflict is detected and then waits for the values to travel through the EX/MEM feedback, thus having no danger of dropping the needed entry too early.

4.3 Control-Hazard resolution and the Pipeline Flush Scheme

In case of a branch mis-prediction, a number of invalid instructions are fetched before branch resolution occurs. These need to be flushed for the datapath to work properly. Since the goal is not to have long wires in the processor arrays, a scheme is needed for flushing the pipeline without using global signals that would also ensure the correct commitment of the of instructions.

For the flushing problem, a scheme similar to what is done in the StageNet can be used. In the case of our design it is actually somewhat simpler to implement compared

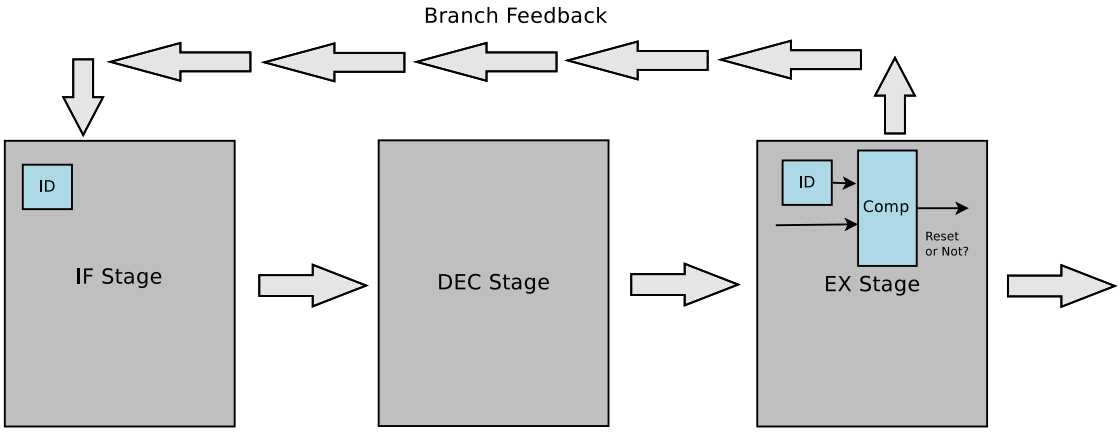


Figure 4.15: The flushing mechanism using ID registers and instruction stream ids

to the StageNet because of the lack of a separate issue stage and the existence of hazard control in the same stage with as branch-resolution logic. The IF and EX stages could include an ID register to identify whether each instruction is valid or part of an invalid instruction stream due to a mis-prediction (Figure 4.14). More precisely, each instruction has one additional bit as a stream ID traveling with it in the pipeline. Its value is given in the IF stage, as the instruction is fetched, by the ID register of this stage. This bit is checked in the EX when the instruction arrives to it. In case it matches the value already stored in the EX ID register it is allowed to continue, otherwise it is flushed and the instruction is not allowed to make any changes in the FIFO buffers. When a mis-prediction is detected, the value of the ID register in EX (alongside with the ID value of the branch instruction) are inverted. When the correct PC in the IF stage is loaded according to the branch that is executed, the IF ID register is also flipped. As a result, the invalid instructions which were fetched before the branch resolution was completed and before the PC value corrected, are marked with a different value in the ID bit than the current value in the EX ID register. So naturally, they will be flushed when they arrive to the EX stage and will not be executed. The correct instructions will have, on the other hand, the same ID value as the ID register in the EX stage and be executed normally.

4.4 Avoiding Global Stall

As previously stated, we prefer to avoid global signals in general. So, alongside a flushing mechanism that operates locally, we also need a similar stalling scheme. In resolving the stall issue here, there are 2 methods that could be used, each with its own merits:

- Propagate the stall signal from stage to stage on every clock cycle.
- Implement the stall function by flush/reload instead of actual pipeline stalling.

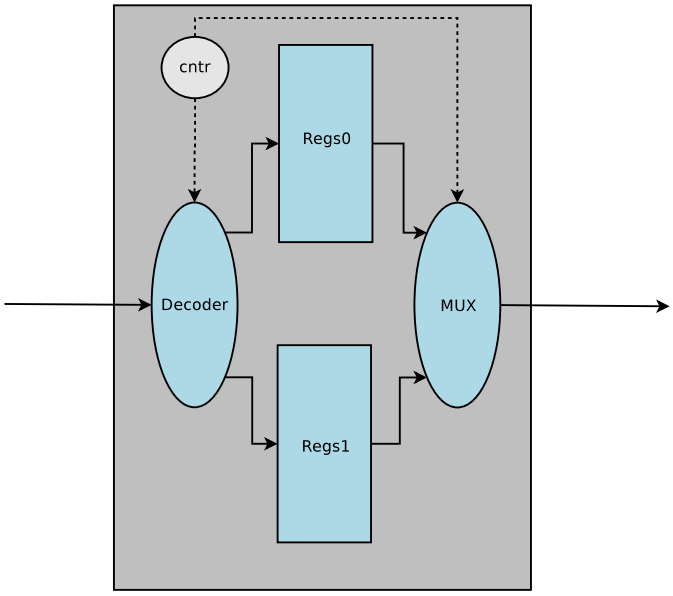


Figure 4.16: Pipeline-register design for implementing stall propagation without data loss.

4.4.1 The Stall Propagation Mechanism

This is a method that is used in typical computer networks when the need of packet stalling arises, such as in the case of congestion [5]. The packets in every node, along the path of transfer, need to stop yet global stall signals are infeasible, just like in our case. The way this is solved is by using *double buffers* in the output of every node and ensuring that consecutive packets are not written on the same buffer. The stall signal is transmitted from the node where it was produced to the previous node and stalls it. Then, on every cycle the stall signal ripples through all the previous nodes. The double buffers ensure that no packet is lost because of the delay of the rippling stall signal. The transfer can, then, start again the same way. The begin signal,which gives the signal for operation to recommence, is just rippling through the nodes just like the stall signal. In our case there could be double pipeline registers on every stage doing exactly the same operation as the buffers (Figure 4.17).

A control in each pipeline-register unit can ensure that consecutive instructions are written to different group of identical registers and that the correct group is read to send its contents to the next stage to ensure the in-order operation of the datapath (Figure 4.16). If the pipeline needs to be stalled because of a data hazard, the stall signal ripples through the stages prior to the one that produced the stall. So, all stages are consecutively stalled without loss of data due to the double pipeline stages.

4.4.2 The Flush/Reload Mechanism

Another typical method one can use to implement the needed stalls is to actually drop an instruction the moment it produces a stall and all the other fetched instructions after

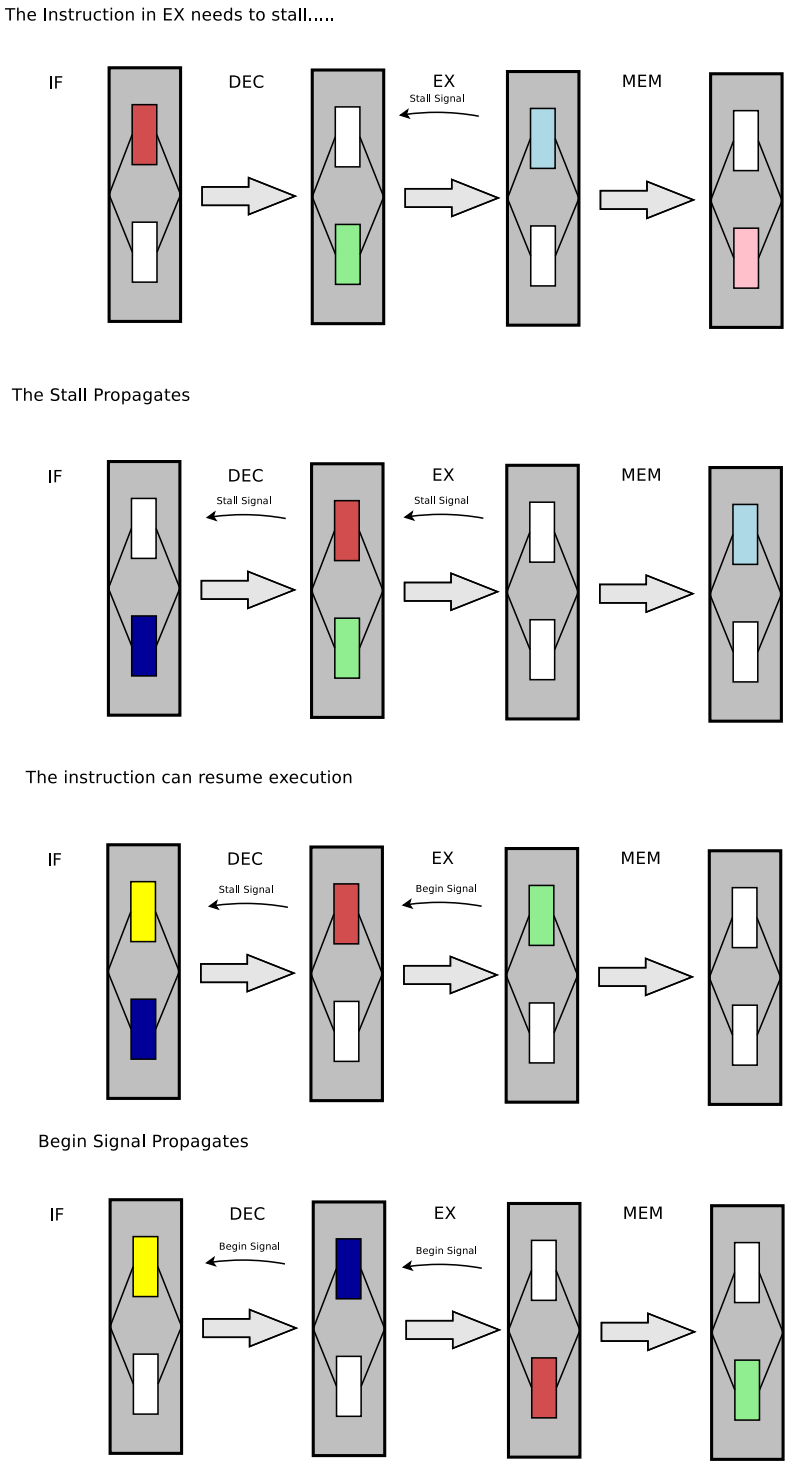


Figure 4.17: Stall and begin signals propagating to the previous pipeline stages. The design avoids data loss with the extra pipeline registers.

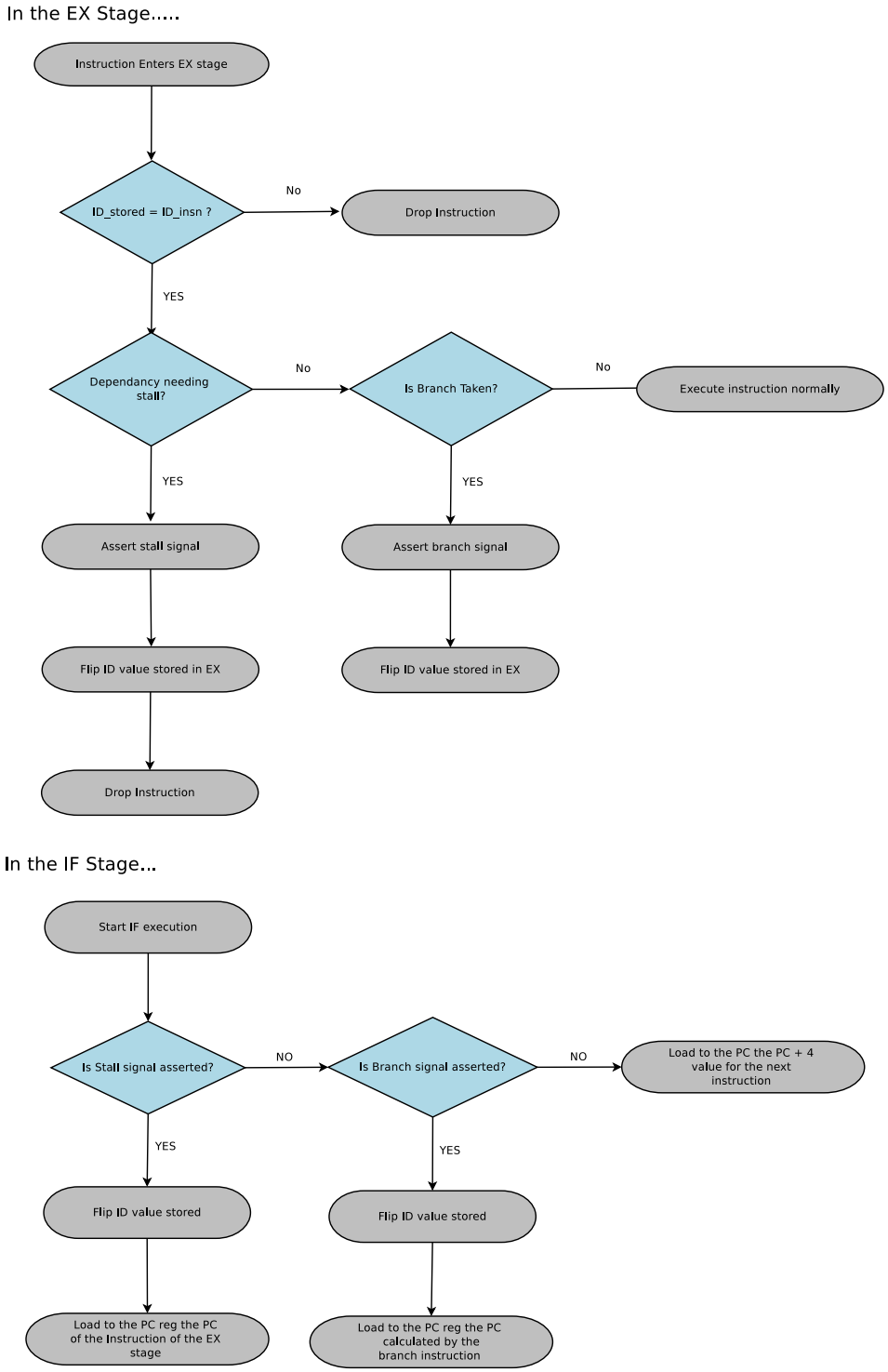


Figure 4.18: Flow charts describing the complete Flush/Stall mechanism in both EX and IF stages

it and reload it in the IF stage with the expectation that, by the time the instruction reaches again the stage in which the need for pausing the pipeline arose, the situation will be resolved and the execution can continue [12]. In our case, it is quite easily applicable since the need for stalls due to a dependency can only arise if an instruction requires a value in the EX stage. So, for the flushing of the pipeline we can use exactly the same mechanism used to flush the pipeline in case of a branch mis-prediction. The only change would be to add some more control logic that activates the mechanism not only in a case of a branch but also when a dependency needing a stall is detected and of course to incorporate this case in the PC selection in the IF stage. This is the method we included in our architecture.

The complete mechanism works as follows (also Figure 4.18):

1. In the Execution Stage

- When an instruction reaches the EX stage it is first checked whether it is valid by the comparing its ID with the ID stored in the EX stage. In case it is not, because of a branch mis-prediction or previous pipeline stall, the instruction is dropped.
- In case it is valid, first the logic should check whether the instruction has a dependency that needs to stall (flush/reload actually) the pipeline. If I has to, it asserts the stall signal going through the feedback to the IF stage and flips the ID value stored in the Execution.
- In case a stall is not needed and the instruction is a branch that needs to be taken it activates the mechanism again as described in the reset scheme above.

2. In the Instruction Fetch stage

- The logic first checks the stall signal coming from the EX stage. In case it is (1), then it flips the ID stored in the IF stage and reloads the PC of the instruction that caused the dependency, also coming from the EX through the branch feedback.
- In case a stall is not needed and if there is a branch needed to be taken, the logic works exactly as described in the reset scheme, and updates the PC with the address coming from the branch execution.
- Otherwise it just typically loads the updated PC (PC+4) for the next instruction.

4.4.3 Method Comparison

Both schemes have their own advantages and disadvantages in terms of performance, area/power and design complexity. In general, in terms of execution cycles, the stall propagation seems to be better since it wastes less cycles than the flushing and reloading of the pipeline but it is much more complex and requires a considerable amount of more logic to implement, having impact on the area and thus power consumption of the design, while also having a small impact on the critical path (aka operating frequency). In more detail:

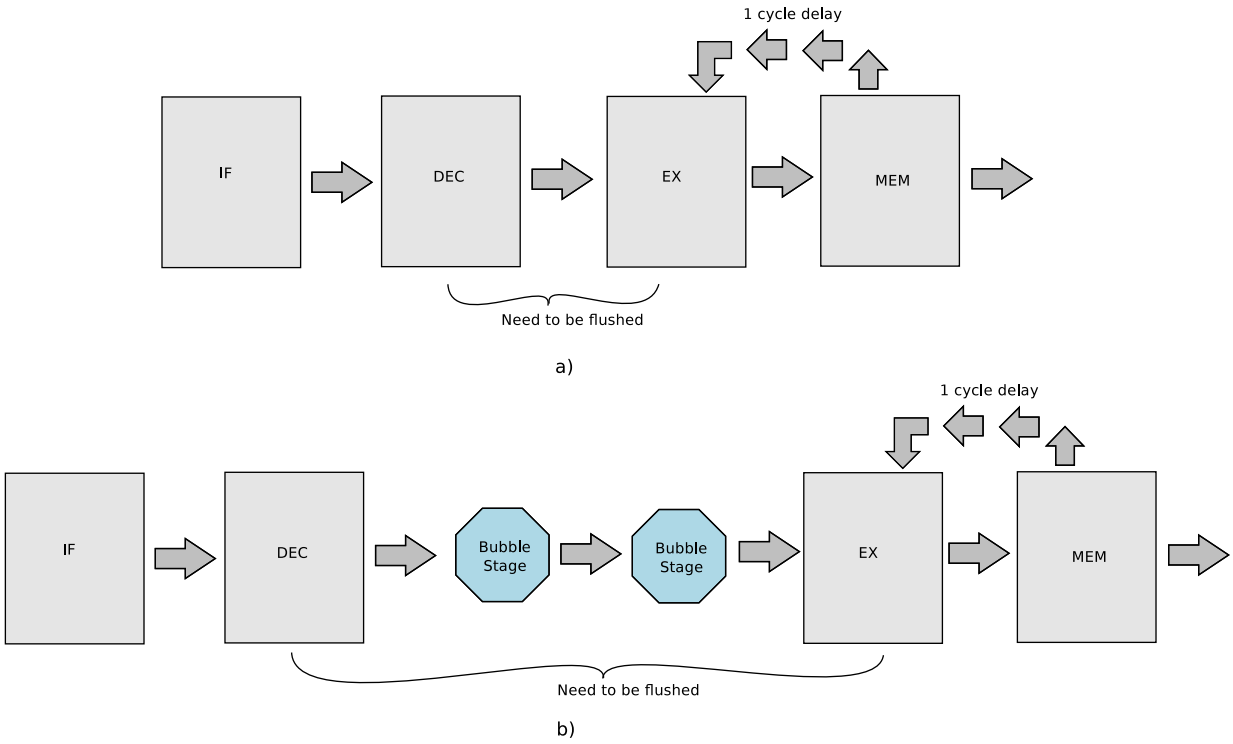


Figure 4.19: Instructions that need to be dropped in a) the initial configuration and b) with the insertion of extra stages before EX.

4.4.3.1 Execution Cycles

A main point of comparison between the 2 methods is the amount of cycles wasted in the case of a dependency needing stalls. Since there is a complete bypass system in the design there is only one case in which a stall is unavoidable: A data dependency between an instruction requiring a value in the EX stage and a LD instruction not yet being executed and sent back to the execution stage. The impact of each method on such a stall would be, as always, depending on the processor configuration.

In the case of initial configuration, where all stages are operational, the architecture is typical to the 5-stage pipeline. So it is obvious, that with this architecture, a scheme with stall propagation would just waste the 1 typical cycle to forward the needed value to the EX stage. On the other hand, a flush/reload scheme would have to pay the same price as a branch mis-prediction and waste 2 cycles instead of 1 since it would flush the instruction coming from the DEC stage and fetch the instruction that created the dependency again (Figure 4.19a). So, here stall propagation has the advantage. It must be noted that this kind of dependency, in this configuration, arises only in the case of data dependency between a LD operation and the instruction immediately after it since there are no bubble stages present widening the dependency window between operations and creating delays in the forwarding of the values coming from the memory.

In the case of a re-configuration, in the presence of a possible fault in one of the stages, the impact of each method in execution cycles can be influenced by the place the

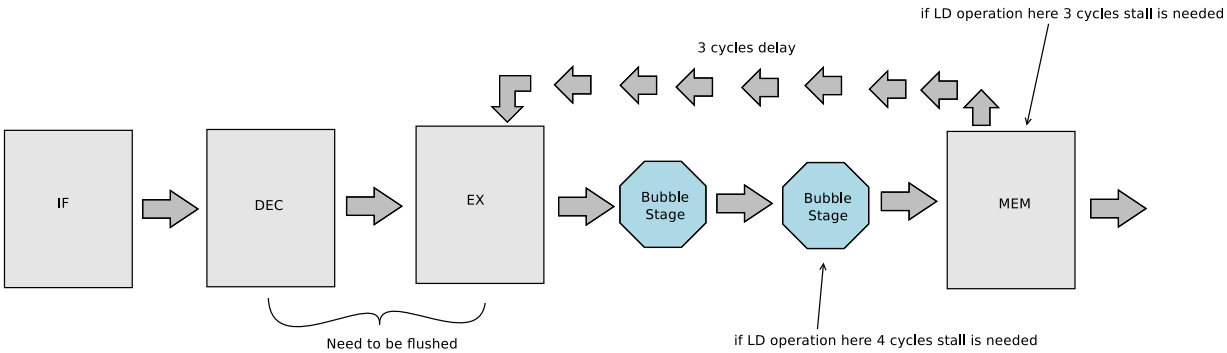


Figure 4.20: Instructions that need to be dropped in with the insertion of extra stages after EX.

bubble stages will be inserted in the pipeline, meaning which stages have to be replaced. Obviously bubble stages inserted between DEC and EX, EX and MEM and after MEM stage have a direct impact on the instruction window in which data dependencies can occur and the delay of the forwarding.

In the case of having bubble stages after the MEM stages, the impact of the stalling is exactly the same as with the initial configuration since, although the dependability window widens, the requirement for stalling the pipeline is exactly the same as before. The extra dependencies can be bypassed immediately through the FIFO buffers and the cost of the stall for both methods is also the same.

When there are bubble stages only between DEC and EX stages, not only the dependency window widens but also the cost for the flush/reload method is increased since more instructions will be issued until a possible dependency needing a stall is detected and all these need to be dropped (Figure 4.19b). In the case of this configuration, the flush/reload scheme wastes considerably more cycles to serve this kind of data hazards than the stall propagation solution. Yet, still the case where a flush/reload would be needed would be as rare as in the initial configuration of the system.

If, on the other hand, bubble stages exist only between EX and MEM stage the 2 methods have a good chance to have exactly the same cost in terms of cycles (Figure 4.20), the reason being that the delay of the forwarding from MEM in cycles is now more than the instructions that need to be squashed in the case of flushing the pipeline. For instance, if a value needed 4 cycles to be forwarded from memory, in the case of the stall propagation, the datapath will need to stall for 4 cycles. If the system does flush and reload, it would have to do it twice until the forwarded value is ready for use and also waste 4 cycles. If the value needs 3 cycles though, the flush reload method would be 1 cycle more wasteful than the stall, a difference that becomes less significant as the bubble stages, and thus the delays, increase in number. It is interesting to note that, in this case, the dependencies needing stalling increase since the delay of sending the correct value though the memory feedback increases also due to the presence of bubble stages between EX and MEM stages.

In conclusion, it is simple to have in mind that in a flush/reload scheme the price one would have to pay to service a data dependency that must wait to be handled with by-

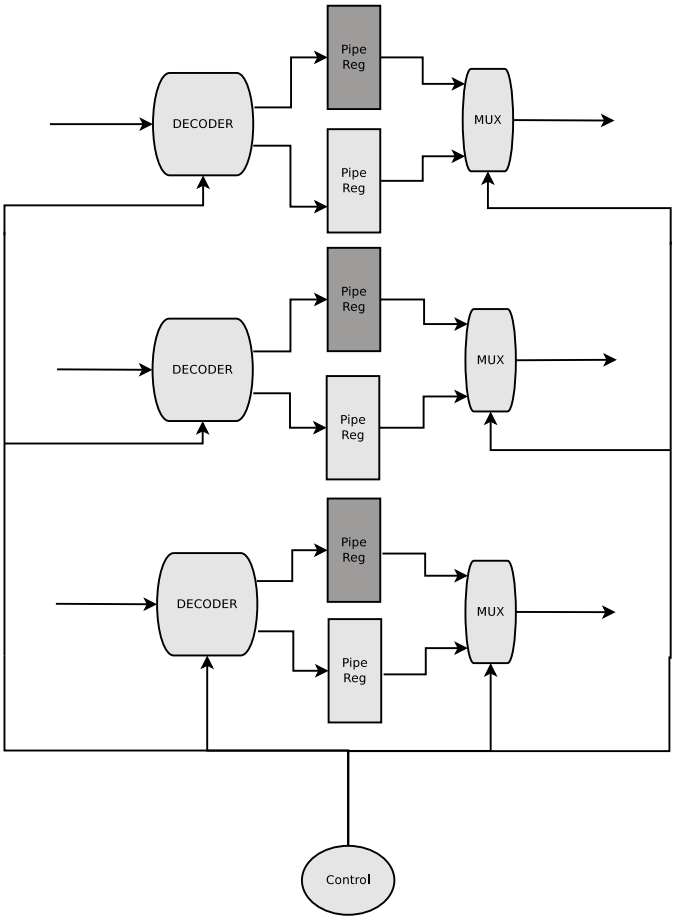


Figure 4.21: Additional logic (in light gray) needed for employing double buffering in the MEM/EX pipeline register. For every register, an additional one is needed, one decoder and MUX.

passing, would be the same as a branch mis-prediction. Depending on the configuration this could be at best the same as the cost of stall propagation, although usually the cost is a little higher. Interestingly, the case where the cost of both methods can be exactly equal are reconfigurations where the presence of data hazards needing stall or flushing are encountered more often.

4.4.3.2 Design complexity, area/power and critical path

In terms of complexity and area/power the flush/reload method seems to have the upper hand. To implement a stall propagate method a considerable amount of logic is needed to function properly under the design's demands. First of all, the use of double the number of registers in the datapath is an obvious cost. The baseline pipeline has in total 28 pipeline registers:

- 4 in IF/DEC;

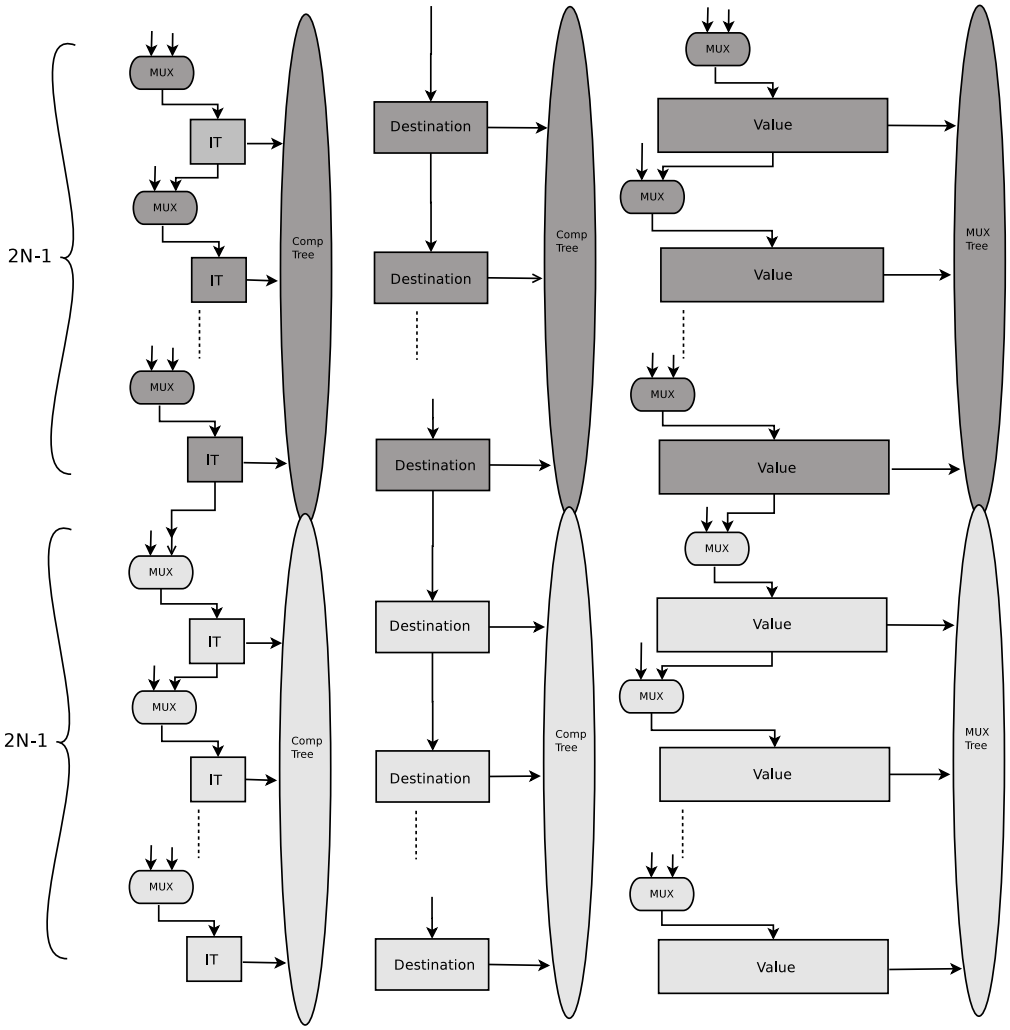


Figure 4.22: Additional logic (in light gray) needed in the each state-saving buffer (in EX and MEM) for the employing double buffering technique. Practically, the FIFOs need to be doubled

- 13 in DEC/EX;
- 8 in EX/MEM; and
- 3 in MEM/WB

And this is without accounting for the 1 bit flip/flops needed to send the control signals from stage to stage. The total pipeline registers will have to be increased to 56 with the double buffering. This would also mean that the pipelining done in the re-configurable network in the case of a re-configuration of the system also needs double the registers, meaning that double buffering must be present to all pipelined cables on the interconnect.

There is also the case of the control logic needed to implement the double buffering. The scheme needs a decoder and a multiplexer before and after every pipeline register, including the ones carrying the control signals, not counted above (Figure 4.21). So, for every single pipeline register in the design there is also the area cost of an extra decoder and multiplexer needed for the functionality. Moreover, because of the double buffering the maximum amount of uncommitted instructions that can be saved within the pipeline doubles and so the instruction window in which data hazards may occur also widens. As a result, the possible dependencies increase in number, which might lead to even more delays in execution. Also, more states need to be saved by the FIFO buffers implementing the bypass.

The FIFO buffers need to be doubled and from 2N-1 they need to become 4N-2, with N being the number of cores in the system (Figure 4.22). For a 4-core system this would mean 14 more registers, 7 more for the buffer in the EX and 7 more for the buffer in the MEM stage. Furthermore, we must take into account the logic each FIFO buffer needs for serving the bypass method. This includes the logic that detects and bypasses a needed value and the logic that writes a value coming to the buffer from the memory in the correct place (for the array in the EX stage). These costs increase considerably as the number of cores increase.

Besides the area cost that results in higher power consumption, there is certainly an impact on the critical path of the design and, therefore, the overall operating frequency. The decoders and multiplexers needed to choose the correct value from the double buffered pipeline registers are, of course, in the critical path that would have a minor impact on the operating frequency. This cost becomes even higher as more bubble stages are inserted in the case of a possible configuration. In addition, the arrays being used for bypassing are also in the critical path and their increase in depth has a direct result in the critical path of the stages they are included in since they need to be indexed sequentially for the design to be able to detect the sequence of the instructions in the pipeline easily without additional logic and control IDs etc. Of course, whether this has any overall significant influence in the operation frequency is relative to each stage's size in when compared to the complete design.

On the other hand, a flush/reload approach has very little effect on the logic of the design. There is no need for double buffering and, so, no need for any increase on the register arrays implementing bypass. Also, it uses exactly the same logic that the reset scheme does for serving the branch mis-predictions. The only cost would be that the branch feedback needs to be some bits wider and some additional logic (a few muxes) in the IF and EX stage is needed to implemented the extra cases in the choice of PC and flushing the instructions coming in the EX stage. All these aspects may mitigate the advantage the stall-propagation method has in overall performance, gained by the difference in execution cycles. Finally, the flush/reload method is considerably easier to implement and verify because it shares logic with the reset scheme for the branching.

These conclusions led us to choose to implement the Flush/Reload Mechanism in our architecture. The stall propagation should give better performance results in terms of executions cycles but the cost in complexity, additional logic and power is considerable, especially when we expect an overall increase in these aspects with the rest of the architectural changes in our design. Moreover, using this technique along-side with the

state saving for bypassing can lead to considerable costs in terms of critical-path delay, diminishing the gain from the execution-cycles advantage of stall propagation. On the other hand, the flush/reload scheme uses much of the same logic used in pipeline flushing in case of branch mis-prediction and has no need to extend the bypassing arrays in EX and MEM. Moreover, a important reason to choose this method was also the tools we had available to develop our design. The main tool was the Coware Processor Designer. It has a number of advantages, that will be discussed later, but one characteristic of the tool is that it does not easily let you tamper with the internal function of the pipelining, i.e. the pipeline registers. Although usually this is an advantage, because it automates the pipeline of the designs, in the case of using double buffering it effectively becomes a hindrance. What is, in fact, needed is to use the tool exactly the opposite way it was supposed to. On the other hand, to describe the behavior of the Flush/Reload mechanisms is much more natural and simple to implement and verify.

4.5 The Network Interconnect and the Complete System

The complete system is comprised of the cores' stages, each of them including its respected pipeline registers and considered a different node, and the interconnect using switches to reroute the data according to the configuration of the core array. If data must travel from one core to another it passes from one switch to the one directly above or below through a pipelined wire, depending on the configuration. Except for the switches that forward the data coming out of the pipeline resisters from one stage to the next, there are also additional switches to guide the feedbacks from MEM and and EX stages(memory values for bypassing and branch feedbacks) to their respective destinations that could be different than the stages' pipeline-register outputs.

The Interconnect Switch

An important element of the complete core is the interconnect switch between the core stages. Their functionality can have an important impact on the overall design. It comes, us usual, to the question of area/power vs flexibility. The more flexibility one could include on the switching nodes, the better configurations can be achieved in terms of performance and efficiency, paying the cost of the functionality in area/power. The switches in general, besides having the input and output for routing the data between stages of the same core, also need two input/output wires for sending/receiving the data to/from the above or below core in the case of a re-configuration (Figure 4.23). We can assume that each switch can be used only once for routing data and new cores created by a reconfiguration cannot use the same path. The in/out wires are required since it is imperative for the system to be able to route data to both directions. The switches are controlled by a 3-bit signal (Table 4.1).

Similarly, bi-directional registers are also required to implement the interconnect pipelining. These registers include a 1 bit control to determine the data direction (also Figure 4.23).

In/out wires are implemented in synthesis using *Tri-state buffers*. Tri-State buffers have the ability to assign their wires, beside the typical 0 and 1, also a state of high

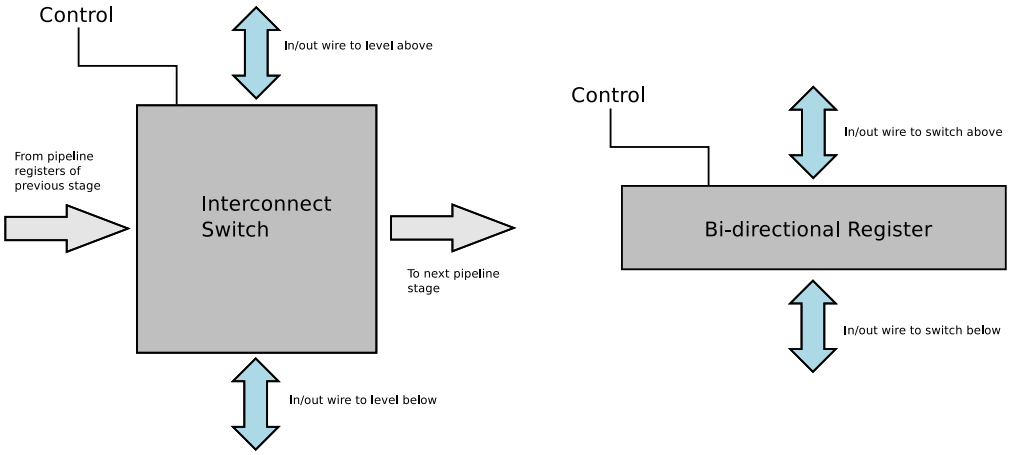


Figure 4.23: Bi-directional switch and register used in the interconnect.

Control Value	State
000	From Input to Output - North and South ports not used
001	From Input to North and South port not used
010	From Input to South and North port not used
011	From North to Output - Input and South ports not used
100	From Input to Output - From North to South
101	From South to Output - Input and North ports not used
110	From Input to Output - From South to North
when others	From Input to Output - North and South ports not used

Table 4.1: Control Inputs for Interconnect Switch with the ability to send data from the level above to the level below(and vice versa), while the normal input and output of the switch are used.

impedance. This effectively isolates the wire. With these devices it is possible to create logic circuits with common in/out ports (bi-directional), isolating one part of the circuit when the other is in use (Figure 4.24). Their main disadvantage is that the wires using tri-states are actually slower.

An interesting ability that the switch could have is also the ability to send data from the level above to the level below (and vice versa), while the normal input and output of the switch are used, instead of just servicing only one path. This can give the system the ability to be more versatile in the choice of re-configuration and result in better configurations in case of a fault; For example, in a case of an array with 4 cores where the first and last cores are damaged but together they can form a new one. If the switches between the two middle cores are unable to service them because they are already used by the cores that still work properly, the two damaged cores are actually unable to share their functional resources. If, on the other hand, the switch could do that, a 3rd core could be created from the two faulty ones. This of course would mean a more complex design for the switch and, as a result, a more costly one.

In the final design we used this latter switch that gives flexibility to the design. Our

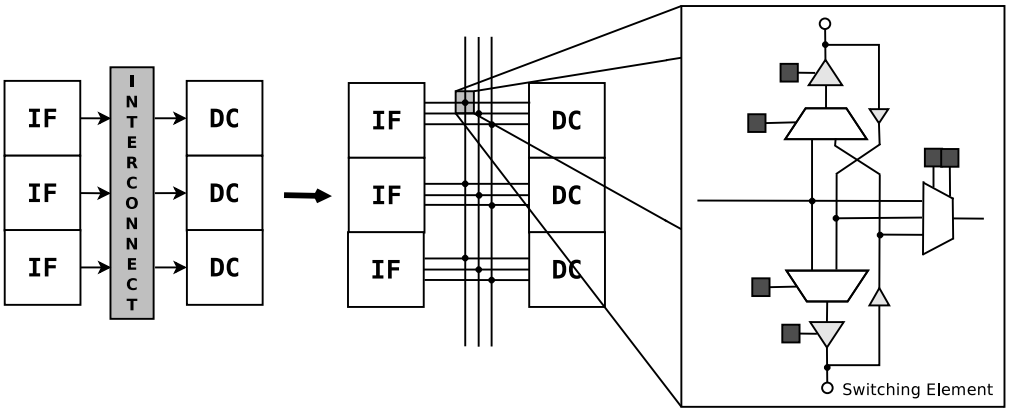


Figure 4.24: Bi-directional switch in detail.

experimental core is an array of 4 cores. For a 4-core array, 16 switches are needed for passing the data though the pipeline and 8 more for the feedback loops, plus 24 bi-directional interconnect register arrays (Figure 4.25).

4.6 Summary

In this chapter we described in detail the general premise of the design and analyzed the basic demands and challenges of the architecture. The main difficulty was that the system had to be designed for a variable number of pipeline stages. Three are the main issues that had to be overcome: the Data Hazard handling, the Control Hazards and Pipeline Flushing scheme and the absence of Global Stall signals. Then, these problems were presented in detail along with the alternative solutions with their advantages and disadvantages and determined the design methods with which they were eventually tackled in our final array. Finally, we described the features of the interconnect for a complete 4 core array. In the next chapter, we present the evaluation results from the experiments done in that 4-core array.

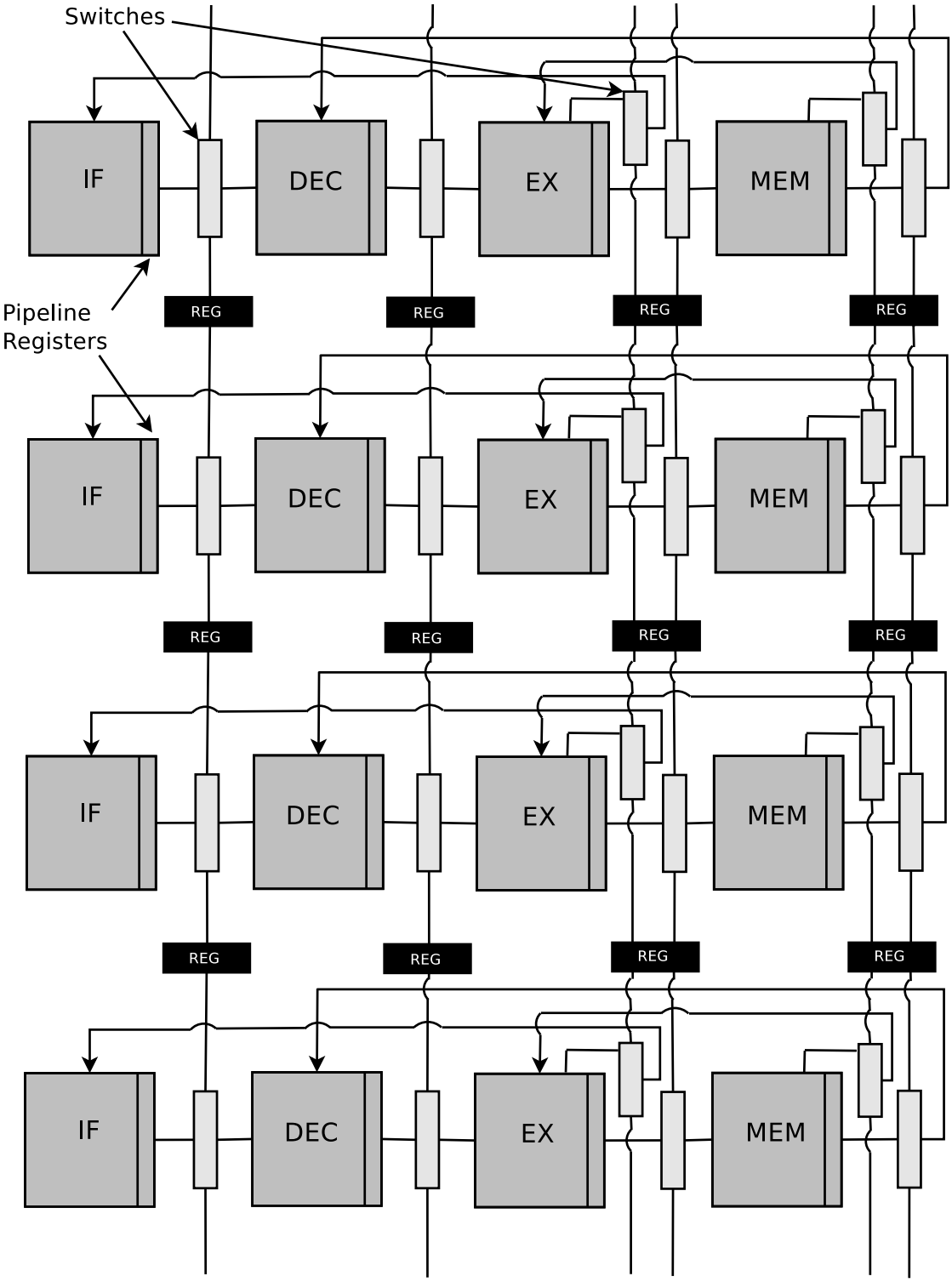


Figure 4.25: The complete 4 core system using our architectural design.

5 Design Process and Evaluation

"Results! Why, man, I have gotten a lot of results. I know several thousand things that won't work."
-Thomas A. Edison

Until this point we have presented our architecture's background and implementation and described a 4-core system as an example of our design. Now we seek to present the development and experimental setup of the design process and, also, the experimental results used to evaluate the new architecture.

The chapter is organized as follows: In the first section we present the development and evaluation tools used for our architecture. In the next section we provide the experimental setup and define the metrics of evaluating the design 's characteristics. Finally, the chapter ends with a presentation of the final results in terms of performance, area, power consumption and reliability.

5.1 Development and Evaluation Tools

Obviously, an important aspect influencing every design is the tools one has available to implement the design ideas. The flexibility, depth and realization process is directly linked to them. In our case, the tools also gave us the means to automate many of the processes and gave us time to concentrate on more important aspect and challenges of the design. VHDL design provides the ability to describe hardware and simulate behavior, but it is not a simple task to choose and setup the tools needed for area, timing and power analysis of the design.

Our library of tools included the Coware Processor and Compiler Designer, Mentor Graphics Modelsim for simulation purposes and switching analysis, and Synopsis Design Compiler tools for area,power and timing analysis (Figure 5.1).

5.1.1 Processor and Compiler Designer

The Coware Processor/Compiler Designer suite is an automated tool that enables the quick design of embedded processors, while also providing the ability to produce their respective C compiler. The modeling is based on the "Language for Instruction-Set Architectures" or LISA language. The LISA code enables the programmer to describe in abstract fashion the behavior of his/her custom processor, in a more or less sequential fashion, in a C-like philosophy. The language is organized in such a way to be able to describe any custom architecture that is based on an ISA. From there, the user can either use the built-in simulator to validate the design's functionality or even use the processor-generator tool [2].

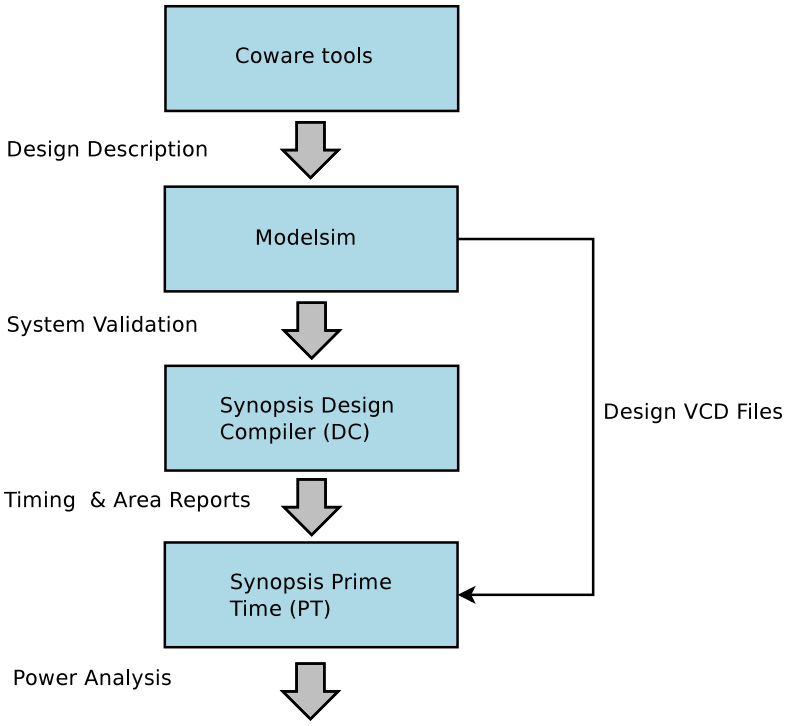


Figure 5.1: General Tool Flow

The suite has a built-in tool that can produce synthesizable RTL descriptions of the LISA described design. From that, point on one can also check functionality using typical RTL simulators, like Modelsim. The design process goes as follows:

- Designers can describe their Instruction Set Architecture within the tool itself. The suite is powerful enough to handle the description of many widely used types of processors like VLIW processors, RISC processors, DSPs, ASIPs and special-purpose co-processors. It even provides demos for the most basic of them, such as 5-stage RISC and VLIW processors, which can serve as learning examples or a basis for the user's custom designs.

- By using the ISA and the LISA language the designers can describe the hardware behavior in a standardized fashion. This gives the users the ability to devise quite complex designs with very little design-time overhead, utilizing the high abstraction level of the language. The same changes or coding choices in actual VHDL or Verilog, depending on the goals of the designer, could take considerably more time to complete. Also, in this case the designers can use the Compiler Designer tool, which is a simplified version of the CoSy tool [2], to develop a compiler for their design. The process here is also largely automated. Using the ISA, the users can define the matching to C language commands and, using the provided linker and assembler, they have a complete working compiler for their processor, with which then can run C benchmarks to validate and evaluate the design. What is also notable is that the compiler tool has a number of parameters that one can tweak

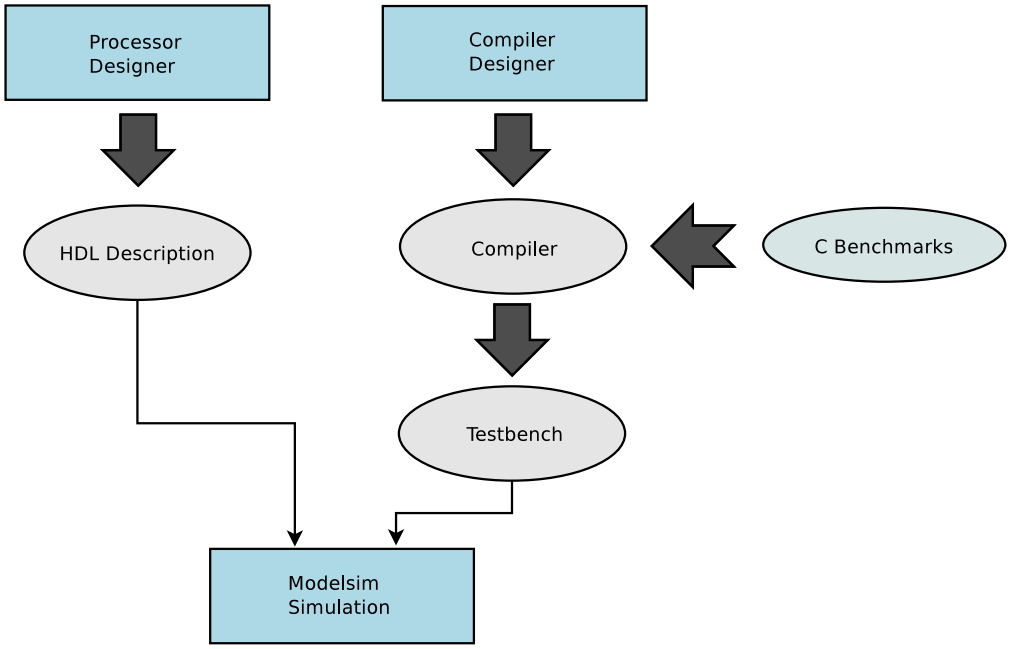


Figure 5.2: Development Tool Flow

to produce a compiler suitable to his goal. Parameters include scheduling options, compiler optimizations, resource allocation etc. The demo designs also come with their demo compilers.

- Lastly, when the designers have a working processor and compiler description, They can either use the built-in debugger and simulator tools to validate their design or even use the Processor Generator tool to produce an RTL description of the architecture. The generator has also a number of tweaking options and can provide both VHDL and Verilog code and even produce the basic structure of simulating the design in various simulators including ModelSim and Cadence. The code is also synthesizable so it can be used for timing , area and power evaluations.

The process we followed in this thesis was similar (Figure 5.2). Using the demo design for the 5-Stage RISC processor, we firstly produced an optimized version of it, suitable to serve us a Baseline design that operated according to the basic 5-Stage RISC design as described in chapter 3. Then in the LISA description we described our own architecture for one core, while also tweaking the compiler to remove any optimizations that might create problems with the aspect of variable pipeline stages in our design. The pipeline functionality, as well as instances simulating reconfigured cores(with more stages) with the same architecture, were then validated by using C and assembly test programs. This was done with the RTL description, produced by the Processor Generator and simulated in ModelSim. The last part of the process was the implementation of the 4-core system. The switches and bi-directional registers were implemented in VHDL using Xylinx ISE suite and simulated in ModelSim again. Then, the system was completed by inserting

the switches and registers in the VHDL generated code of the single core, by replicating the core and then testing the final system in ModelSim.

These tools have a number of advantages for our purposes. First of all, the main advantage was the presence of the Compiler Designer (CD). Because of the nature of our architecture we want a compiler that does not do anything more than translating the C code to working assembly avoiding the presence of optimizations that might not work with our architecture. The presence of the CD gave us the ability to do so without the need to redesign the compiler but to tweak it for our needs with just a few mouse clicks. Another advantage would be the abstraction of offered by the LISA language enabling the implementation of changes with only some lines of code which could take a significant amount of extra time if one wrote directly in VHDL, while at the same time enabling us to produce testable and synthesizable VHDL code of the design.

Yet, the usage of the tool does not come without its difficulties. First of all, it naturally had a learning curve one has to overcome to use it efficiently. Also, the fact that the LISA code is a sequential language describing hardware, which is mainly parallel in philosophy, can create obstacles in the generation of one from the other. One must have a good understanding of the model so he can anticipate what kind of LISA code will produce what kind of VHDL to make sure the LISA description does not produce code that is too costly. And, even then, one cannot achieve that always. Additionally, the generated code is quite complex and, as it is generated, not always optimal. For example, it tends to implement functions with many dozens of wires while, intuitively, a designer writing directly VHDL could do the same with 10 or 20 wires. This has a impact on the final overheads of the architecture. It, also, makes development more complex if, as in our case, the designer has a need to alter the VHDL code directly. Finally, the sequential nature of the description means that the model is event driven. This adds limitations that in actual hardware could be overcome, like the absence of asynchronous stalls for example, which in our case was one of the reasons the stall-propagation techniques were not chosen for the implementation of our design. Generally speaking, it is actually the aspects that make Lisatek powerful that create problems in our case: The ease of the high level creation of a computer architecture. Sometimes, implementation details are too important and avoiding them creates complications. Yet, even with these limitations, the practicality of the model for quick design, synthesizable code generation and the presence of a parameter-driven compiler tool are quite enough to serve the needs for this architecture and justify the tool's usage.

5.1.2 Synthesis and Power analysis tools

The next step is to evaluate our design. The simulation in Modelsim can give us the performance in execution cycles. But for time, area and power results, on the other hand, synthesis and power analysis is required to obtain the necessary reports.

First we have to synthesize our design to acquire timing and area results. Synthesis is the procedure where a hardware description on an high level HDL, VHDL in our case, is translated into gate-level. Practically speaking, synthesis implements the VHDL design in terms of logic gates. Our synthesis tool is the Synopsis Design Compiler. For the tool to be able to implement the design it requires, except for the HDL file of our

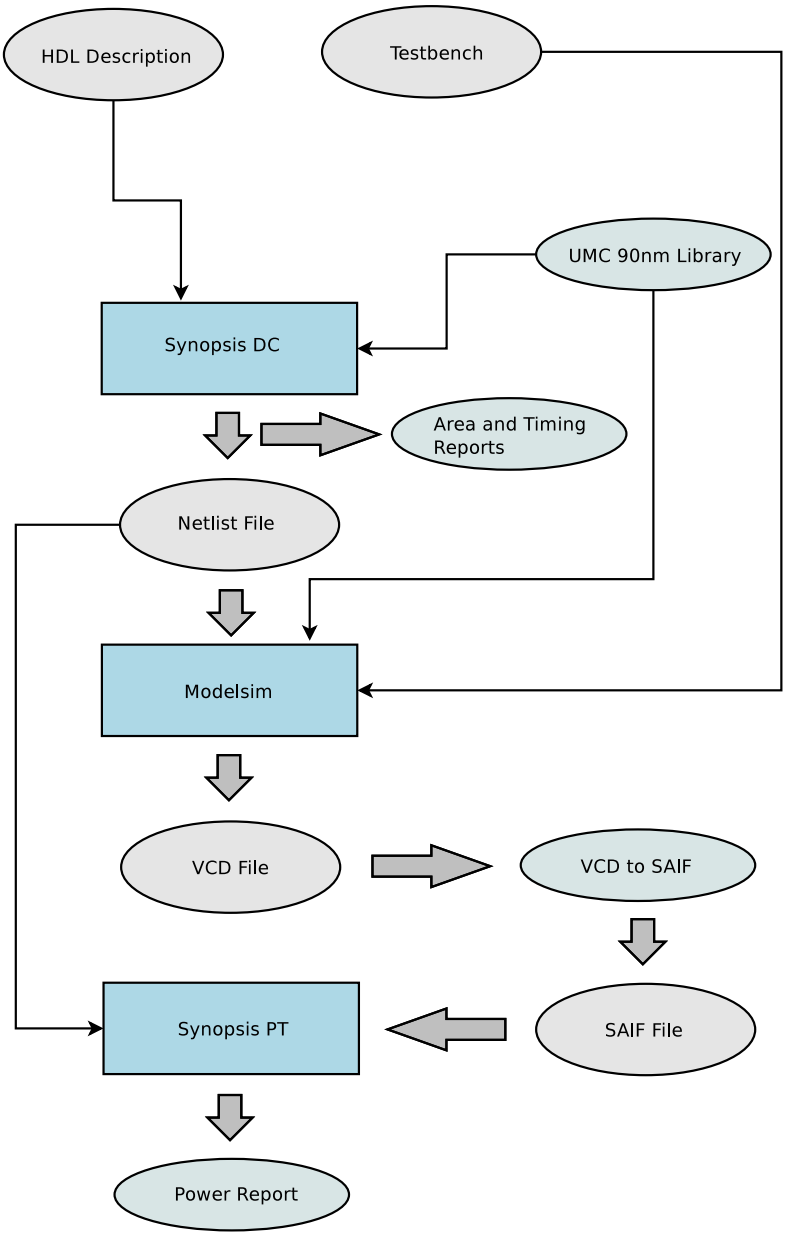


Figure 5.3: Synthesis and Power Analysis procedure

design, a device library is also needed. The device library has description models for all the basic building blocks to synthesize a hardware design such as MUXs, gates and flop-flops while also containing models that have to do with wires. Each model comes with its annotations concerning timing delays, area sizes etc. According to the library information, the tool can report on estimates about timing and area characteristics of the synthesized design. In our case, we use the UMC 90nm SP (Standard Purpose) library. The final output is a netlist file in VHDL form.

The netlist file can, then, be used to produce the scripts needed for power analysis.

The tool we used for this purpose was Synopsis Prime Time. Before we can make estimations on power consumption in our design we need to acquire information on the switching activity of our design for our various test programs. We need to produce a VCD (Value-Change Dump) file with the help of Modelsim. Although the netlist file created by the synthesis and the test-benches is produced by the Design Compiler, Modelsim can also create the VCD file containing the needed description on switching activity, for each of our test programs. Since PT does not work with VCD files, we need to convert the VCD to SAIF (Switching-Activity Interchange Format) form. This is done by the "vdctosaif" application included with the Synopsis tools. Finally, Synopsis PT can use the SAIF file to produce reports on power consumption on each of our benchmarks. The full tool flow for the timing, area and power analysis can be seen in Figure 5.3.

5.2 Experimental Setup

Here, we discuss the experimental setup in which we proceeded to evaluate the final outcome of the design. Our setup is made up to recognize the design's characteristics in terms area, frequency and general execution time overhead and power consumption compared to our Baseline core.

5.2.1 Area, Timing and Power Estimation

The frequency and general timing results come from the synthesis report of the Synopsis DC tool. The DC provides also a complete area analysis. Timing in Synopsis is calculated as *slack time (ST)*,being either negative or positive slack. When one defines the setup options for the process he also defines the maximum *clock period* (T) for the design in *nanoseconds* (ns). DC then calculates all delay between all possible inputs and outputs and detects the longest path. The final report provides the delay of every part of the design and subtracts the longest delay of the slowest part of the pipeline alongside the extra delays for data arrival and gives the slack time, or in other words how much time of the maximum clock period the critical path of the design does not cover or by how much it exceeds it. By subtracting/adding the slack from/to the maximum delay we get the design's maximum clock period. Then, we can calculate *Operating Frequency* through:

$\mathbf{F} = 1/T * 10^9$ - where T is the clock period (in ns) and F the operating frequency measured in *Hertz* (Hz).

It must be pointed out that the timing results of the synthesis are in actuality estimates based on the delay model of the technology library. Accurate, real-life measurements can only be made by place-and-route procedures. Yet, for basic evaluation, results these estimates are useful to make conclusions.

For the area estimation a design is divided into 2 parts. The cell area that is comprised of the area taken by the logic components of the design and the net area concerning the area taken by the wiring. The UMC libraries, that we are using, came with a model of zero area wires. Meaning that it is impossible for it to provide the net area for a design.

Yet, the size of the net area is usually only a fraction of the cell area of the design, as, for examle, in the interconnect the majority of the area is coming from the logic of the switches. So, just by the cell area calculation the designer can get a good estimation of area overheads. The cell or logic unit that the library uses as basic design component are dimensionless objects that with a relative size according to the models included on the library. According to those sizes the total area is calculated. To have an actual area computation in square micrometers would also require place and route.

For the power estimation the Synopsis PT can give accurate estimates on power consumption since it also has information about the switching activity of the designs according to the different benchmarks used to test them. For every different benchmark a SAIF must be produced to calculate the power consumption for each testing program. From the provided report an average consumption can be deduced for our benchmarks.

5.2.2 Performance in Execution Cycles

To assess the cost of the architectural changes in our designs, compared to the baseline, we measure the clock cycles spent on Benchmark execution for every design using ModelSim simulation. As benchmarks, we use small custom C programs that are designed to test the tolerance of the designs to the hazards that seem to usually hinder the number of cycles required to finish execution. Practically speaking, they are extreme occasions of code programs might include that have direct influence on pipeline execution, used to judge the impact of our modifications on these occasions (*stress-marks*). We have chosen to use 3 types of benchmarks each concentrating on code prone to creating delays plus 1 normal C example of code:

- Function-Argument Heavy Code;
- Heavy Read-After-Write Conflict Code;
- Heavy Branch non-taken Code; and
- Normal C "for" loop Code.

Function Argument Heavy Code: This is a case of code including a function with a large number of arguments. This type of code is prone to creating Data Hazards between arithmetic and memory operations due to the initial memory operations. Useful to assess the impact of the Flush/Reload mechanism used to address the pipeline stalling.

Heavy Read After Write Conflict Code: This program has code producing a large amount of RAW hazards. We expect that all of them can be serviced by the bypass mechanism and should not under normal operating conditions show any overhead in our architecture. This is not the case though in the case of a reconfiguration that would increase the feedbacks' delays or flush penalties.

Heavy Branch non-taken Code: A loop with non-taken branches is also used to assess the Flush/Reload mechanism but in terms of branch mis-prediction this time. Here again we do not expect any changes in the case of normal operating conditions but a significant impact at a configuration that influences the penalty paid for flushing the pipeline.

Normal C "for" loop Code : All the previous micro-benchmarks include a typical C "for" loop. The iterations themselves, besides what lies within the loop, produce hazards that could affect program execution. And, since a typical loop is a type of code quite often used, we also include a micro-benchmark with a void "for" loop. As before, our architecture should be able to run the program in the same cycle count in normal conditions as the Baseline Pipeline, expecting an overhead to appear in configurations that influence the penalties for flushing and feedback delays.

Using the time analysis alongside the results of the benchmarks, we can also compute general execution time for our design. These benchmarks have a size of about 1000-2000 execution cycles in the Baseline processor. The size was chosen for convenience, since power analysis works with the same benchmarks. Large benchmarks would take a prohibitive amount of time for the power analysis to be completed. So, programs with this size are not too large enough for the power analysis to be prohibitively long, while they are large enough to help reach conclusions on the performance overheads of the architecture.

5.2.3 Test Designs

Now that we have presented the process of acquiring the result, we can provide a few details about the designs we wish to evaluate. The experiments were done in (a) the Baseline pipeline, (b) our new Defect-Tolerant pipeline as would operate in a 4-core system without faults and (c) in two different pipeline configurations representing 2 worst case configurations that could be present on the 4-core system.

In more detail the timing and area analysis were done on one core using the Baseline pipeline, on one core using the new pipeline without interconnect logic and the 4-core complete system using the new pipeline that includes the switches and full interconnect operating normally. Results in area and timing delay do not change with the alternative possible configurations in the 4-core system. We also include timing delay results for a 4-core design with un-pipelined interconnect to show the benefit of pipelining the interconnect wiring. The power analysis was done on the same designs on all 4 benchmarks. The reason was that we wanted to show both the overhead of architectural changes alone on the pipeline and the complete system including the interconnect. In the case of execution cycles and execution time, we also want to explore the overhead of the possible worst case configurations. For this reason, we also run the benchmarks on 2 more possible worst-case pipeline configurations that suffer the largest slowdown that a reconfigured core can suffer from.

To be more precise, the 1st scenario of worst case configurations is the one in which the top core of the system can create just one core by using the MEM stage of the bottom core (Figure 5.4). In this case, the system adds the maximum possible number of stages between EX and MEM stages suffering from the longest possible delays in both WB and forwarding feedback. This, at the same time, increases the instruction window that data hazards might occur in, while at the same time the pipeline requirement to wait for the transfer of values from MEM to EX for bypassing, is the maximum. Therefore, this should have a significant overhead in execution time.

In the 2nd scenario, which is the worst case possible in our systems, the new core

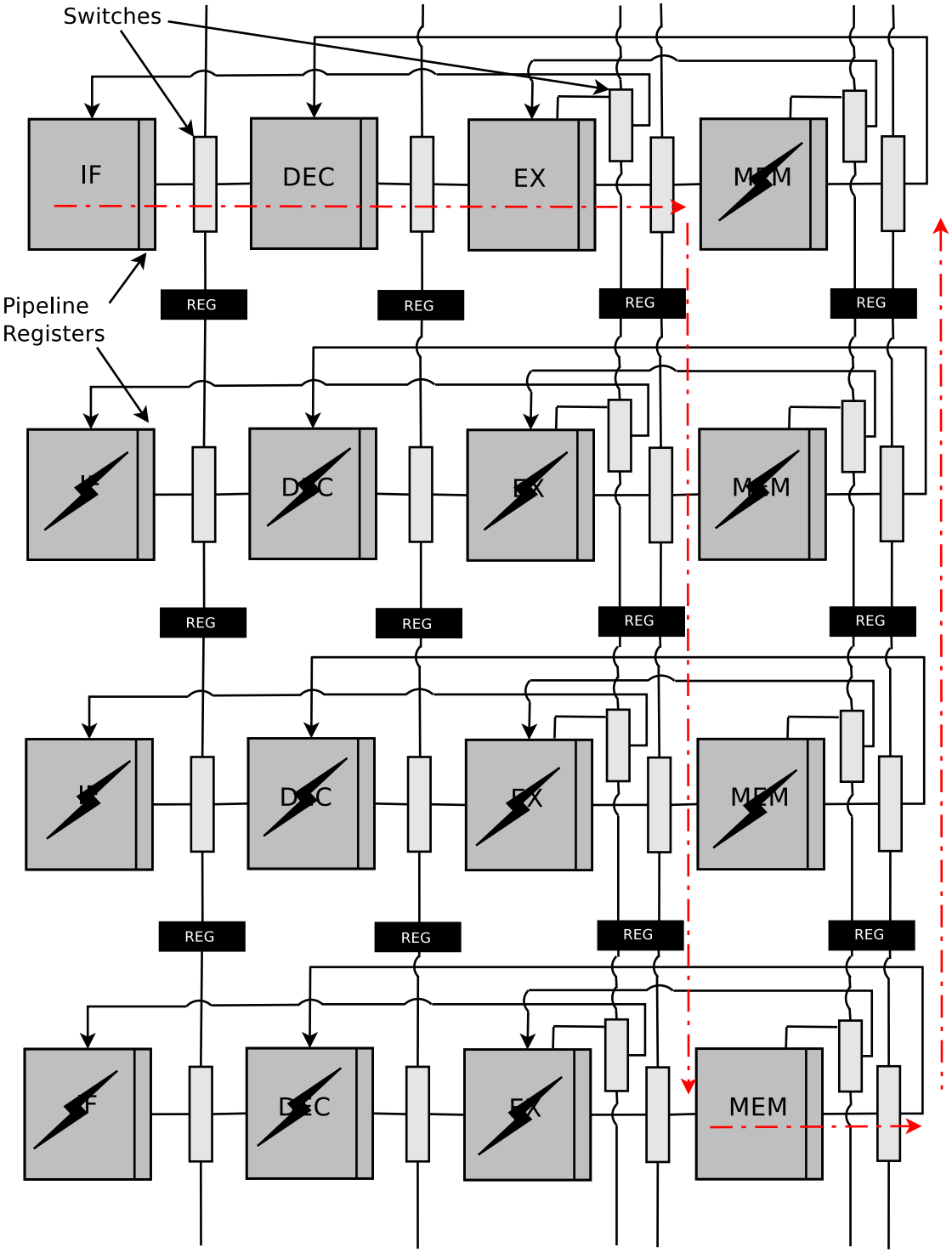


Figure 5.4: 1st worst case configuration scenario used for our experiments

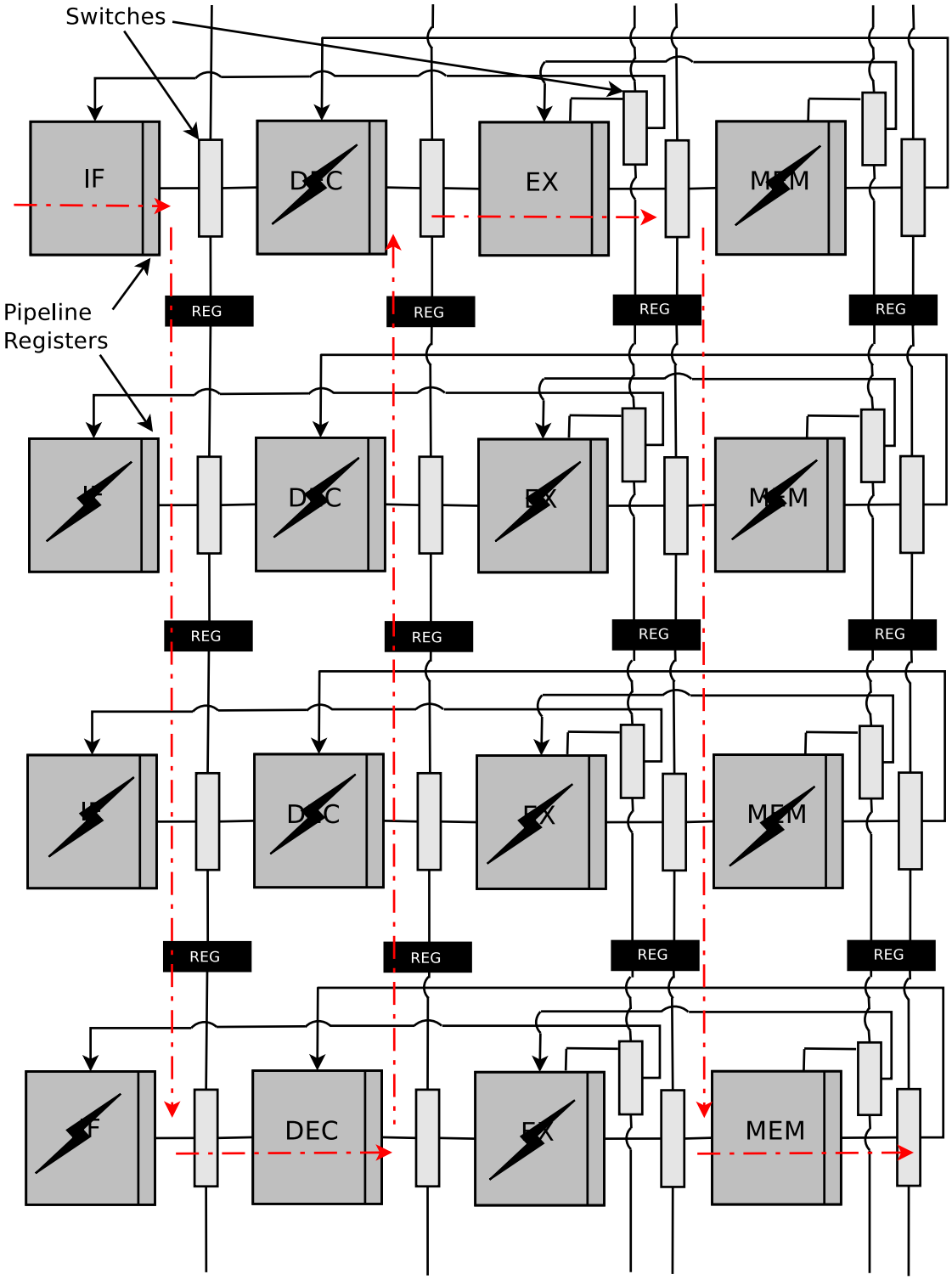


Figure 5.5: 2nd worst case configuration scenario used for our experiments

uses the DEC stage from the bottom core besides the MEM stage (Figure 5.5). Here, additionally to the overhead of the previous configuration, we have an increased cost of the pipeline flushing in the case of branch mis-predictions and data hazard stalls because of the extra stages before and after DEC.

5.3 Design Evaluation

In this section, we present the results of our evaluation. Intuitively, we expect that the end result has an important fault coverage using stage granularity. Even if the new architecture does add a performance overhead and has not as good a performance as a non-defect-tolerant multi-core system using the baseline core, the fault tolerance capabilities can give an edge in total performance, in the long term. The reason is that, as time passes, faults render cores unusable and the performance degradation of the non-fault-tolerant system will be rapid, while our system can salvage some of the performance lost by creating new cores that may be slower than initially yet will still be functional while in normal circumstances there would be none able to work. The evaluation is done with the goal of recognizing what costs paid for incorporating the designs that give our system the defect tolerance. In that way the most costly part can be detected and possible optimizations can be defined that would make performance degradation even more graceful in the presence of faults. These are categorized and presented according to the experimental setup described in the previous section.

5.3.1 Performance

In Figure 5.6 we can see our benchmark results in terms of execution cycles, while in Figure 5.7 we can see the difference between the test and baseline pipelines in percentages. We can see that, as predicted, between the Baseline pipeline and our Defect-Tolerant pipeline for our multi-core system there is only a small difference in execution cycles in the case of the Argument Heavy test. There is a difference of about 12% in the certain benchmark and can be awarded to the Flush/Reload mechanism. When a conflict is detected needing a stall, the flushing of the pipeline wastes 2 instructions instead of the 1 that is spent on stalling the pipeline in the Baseline core. So, this leads to the 12% cost in execution cycles.

On the other hand, our bypassing mechanism and the fact that the Flush/Reload mechanism for branch mis-predictions is able to cope with the hazards in exactly the same way as the Baseline, results in our pipeline having no additional overhead. In total, adding the benchmarks together we come up with just a small overhead of 1.92% in execution cycles in our pipeline's design, which is also the performance penalty of our system's cores in the absence of faults. Interesting are the results in the two worst-case configurations, where the defects as so many that only one processor can be made up from scavenged stages. The overheads are considerable, but these are, after all, the worst case options that could only occur long after all fault free cores are gone in the system, which would be entirely disabled if the baseline design would have been used.

In the case of the 1st worst-case configuration, we see that the most resistant benchmark in the slowdown is the RAW benchmark. Here, although we have more data

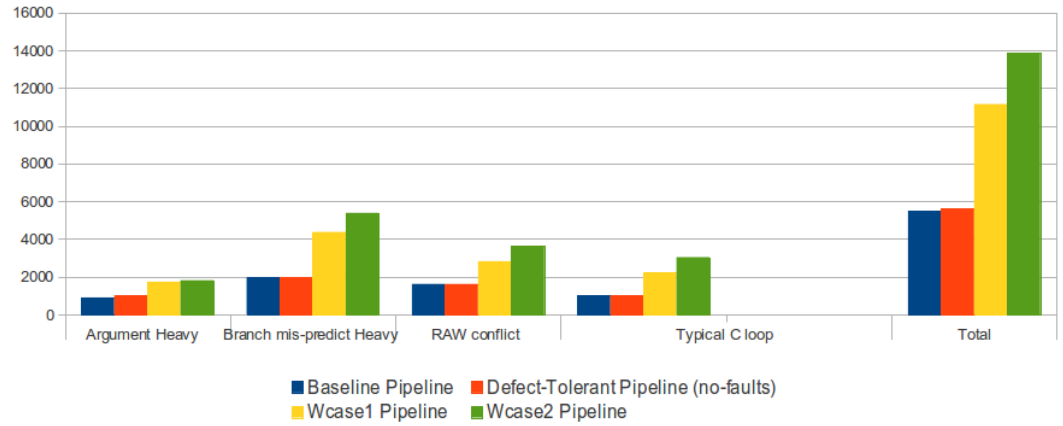


Figure 5.6: Performance of the test designs in execution cycles for each micro-benchmark.

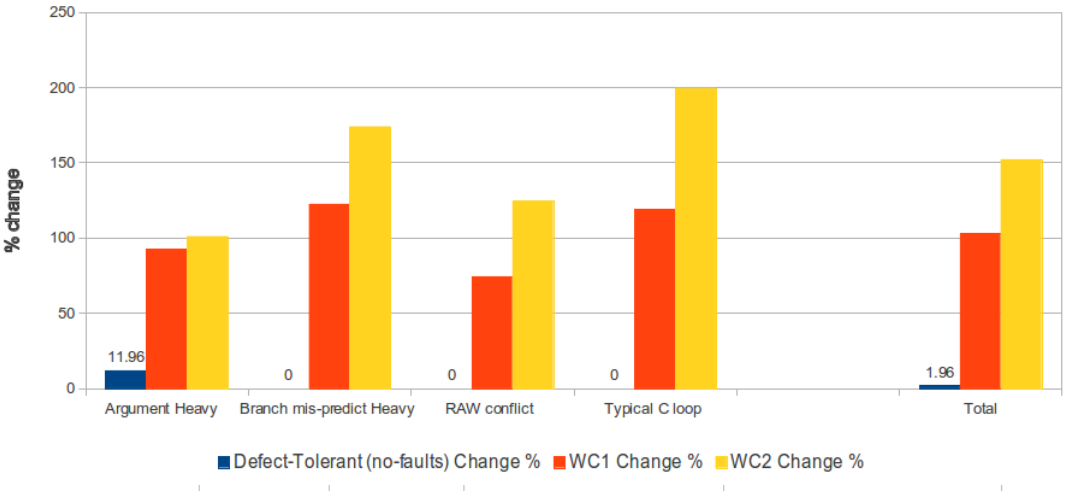


Figure 5.7: Change in execution cycles for the test designs compared to the Baseline core.

conflicts, the presence of the bypassing mechanism can service a number of them, hence that benchmark has the smallest amount of slowdown. Largest cost can be seen in the void-loop benchmark. This is to be expected since the complete absence of instructions within the loop brings up, in this case, conflict between the variables that are used in the loop iterations which, under normal circumstances would not come up; hence, the increased slowdown. The largest overhead can been seen during the mis-predict that the extra data conflicts add up to the penalties paid in the mis-prediction conflicts. Overall, this configuration results in a core that takes twice the number of execution cycles to complete the benchmarks, compared to the baseline pipeline.

For the 2nd worst-case pipeline the pattern is roughly the same, albeit with increased overhead due to the increased penalty for pipeline flushing found in this case. The only

Design	Slack(ns)	Clock Period(ns)	Period Increase
Baseline Pipeline	0.00	5.00	-
Defect-Tolerant Pipeline	-0.84	5.84	16.8%
4C w/o pipelined interc.	-5.24	10.24	75% – 105%
4C w/ pipelined interc.	-2.54	7.54	34.2% – 50.8%

Design	Op. Frequency(Mhz)	Frequency Reduction
Baseline Pipeline	200	-
Defect-Tolerant Pipeline	171.2	-14.4%
4C w/o pipelined interc.	97.6	-42.9% – -51.2%
4C w/ pipelined interc.	132.6	-22.5% – -33.7%

Table 5.1: Timing measurements for our test systems. Visible in the table are the slack time, clock period, operating frequency. Also, in the last column we can see the percentage change in period and frequency between the design compared to the Baseline. In the 4-core system without pipelining in the interconnect (4C w/o pipelined interc.) and the full 4-core system we can see the change compared to both the Defect-Tolerant and the Baseline Pipeline respectively.

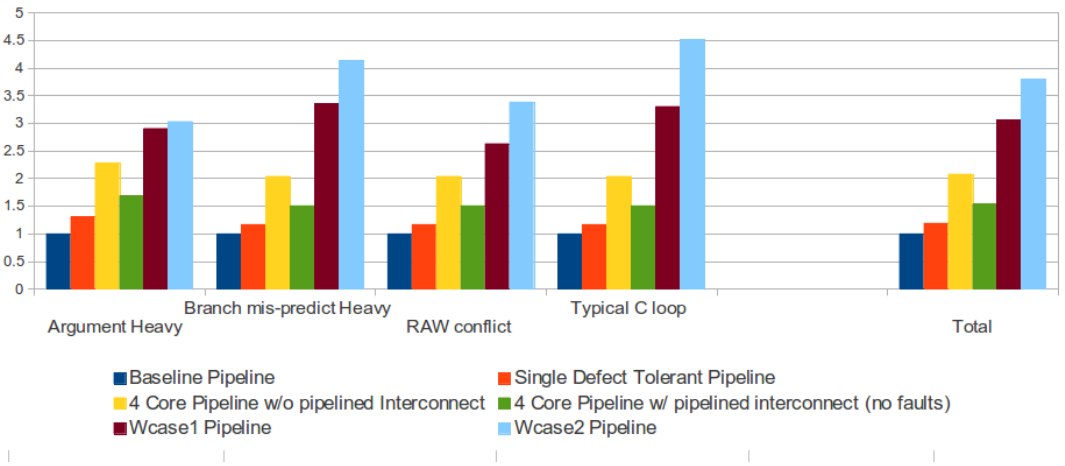


Figure 5.8: Performance in terms of execution time normalized to the Baseline values

difference is that, now, the least influenced benchmark is the argument-heavy test since the amount of conflicts for its case remains the same along with the wasted instructions needed to resolve them. The resulting pipeline takes about 2.5x more cycles to finish execution.

The other main aspect of performance is the operating frequency. Reports on timing were taken in the Baseline pipeline, a Defect-Tolerant pipeline without the interconnect logic to evaluate the overhead of the architectural changes and the complete 4-core system, without and with pipelining in the interconnect, to assess overall operating frequency. The interesting part, which this work is also trying to conclude on, is the interconnect impact on the operating frequency. The lowest it is, the most viable

Design	Area in Logic Cells	Comments
Baseline	86197	–
Single Defect-Tolerant Core	145800	69.14% compared to Baseline
4 Core System	846571	–
4 Core System Interconnect only	263371	31.11% of total System Area

Table 5.2: Area Measurements from DC Synthesis. Seen also the difference of the Single Core with the baseline core and the percentage of the interconnect in the end design.

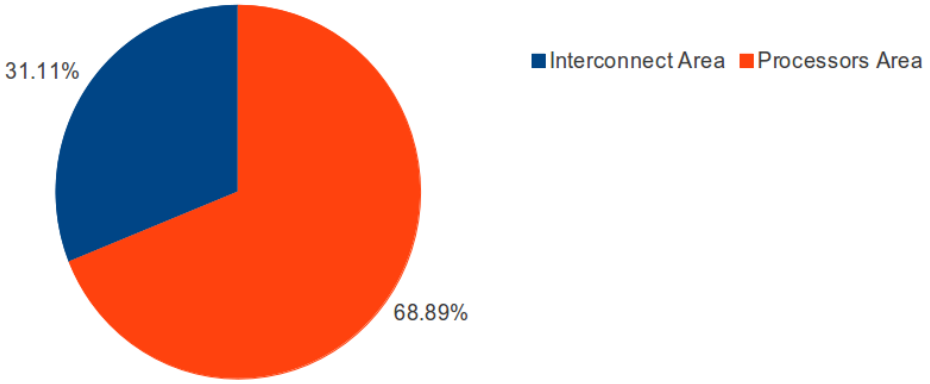


Figure 5.9: Area distribution in 4 core system

the solution will be. This we can find by comparing the operating frequency between the Defect-Tolerant pipeline without the interconnect logic and the 4-core system that include the interconnect elements and pipeline registers.

From the timing report we can compute the operating frequency of the design (Table 5.1). We can see that the Baseline pipeline has a frequency of 200Mhz. Our architectural changes yield a pipeline design with 171MHz; a 14.4% reduction. This results from our architectural changes that have to do with the bypassing mechanism. The FIFO buffers are practically a series of comparators, registers and multiplexers. All are operated sequentially so as to take advantage of the in-order nature of the the architecture to ensure correct value bypassing. This, along with the other subsystems that ensure stage decoupling, are all on the critical path adding to the frequency penalty. The full 4-core system with un-pipelined interconnect has an operating frequency of 97.6MHz which is a huge reduction of 42.9% compared to the Defect-Tolerant single pipeline. The reduction in operating frequency when the pipelined interconnect is considered is just 22.5%, showing the benefit of the pipelining. This gives an overall reduction compared to the baseline of 33.7%.

Using the results on execution cycles and frequency, we can derive the total execution time for our benchmarks. These can be seen in Figure 5.8. The values in the figure are normalized to the Baseline execution time.

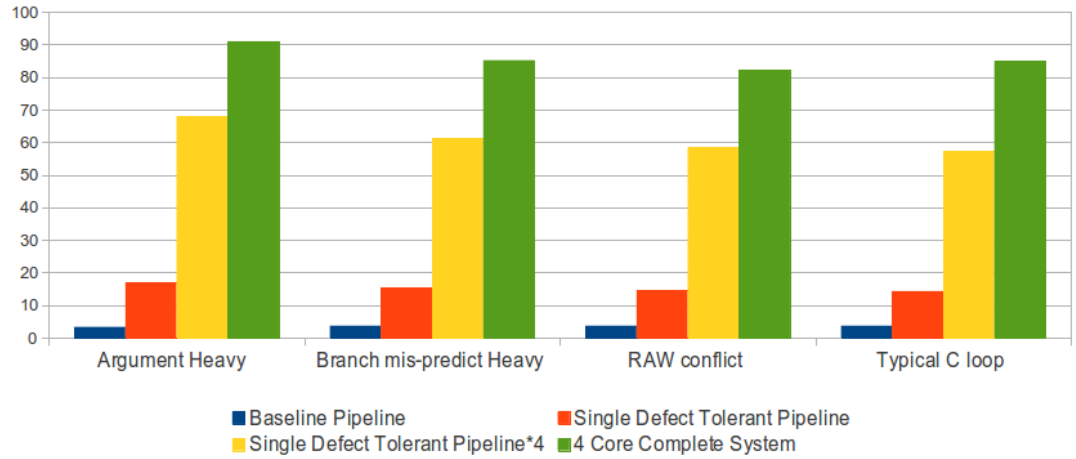


Figure 5.10: Power Consumption for each benchmark (in mW). Depicted here is the Baseline, Single Defect-Tolerant pipelines and the Defect-Tolerant pipeline multiplied by 4 (for comparison purposes) and the complete 4-core system to show the interconnect's consumption.

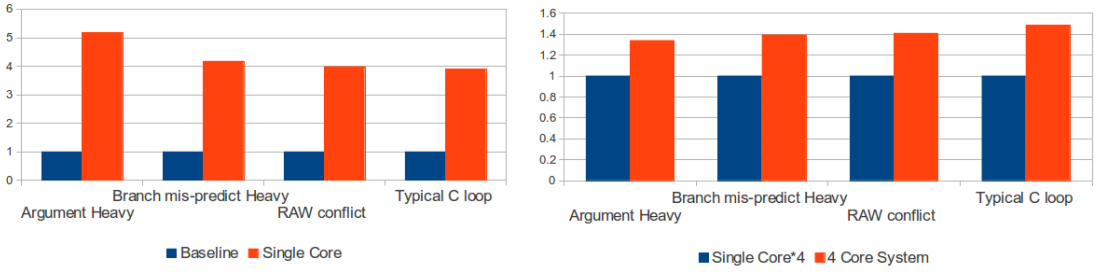


Figure 5.11: Normalized power consumption. Left chart between Baseline and Defect-Tolerant pipeline. Right chart between Defect-Tolerant pipeline*4 and full 4-core system.

5.3.2 Area and Power

The added logic that is needed for the decoupling of the stages and the resolution of the various design challenges in the architecture is expected to need a considerable amount of area. To the complete system we must add the area overhead of the switching interconnect. Of course, with the additional logic comes also additional power consumption, especially if we consider the nature of the added logic which is expected to have heavy usage and switching activity. To that we must also add the limitations of the processor generator in the generation of optimal VHDL code.

Table 5.2 depicts the area measurements for the Baseline, Single Defect-Tolerant pipelines, the 4-core system and the pure interconnect area of the 4-core system in logic cells. We can see our architectural changes incur a 69% area overhead. This stems from the bypass mechanisms in EX an MEM, the bypass arrays and the additional logic required by the Flush/Reload for data and control hazards resolution. Also, an

interesting fact is that the interconnect in our 4-core system covers about 31% of the total area of the system (Figure 5.9).

In terms of power, we expect a similar overhead due to the usage of the additional logic. The increased switching activity in the additional logic and the activity of the interconnect have a direct influence on the consumption. In Figure 5.10 we can see the power consumption for every micro-benchmark. From the Baseline to the Defect-Tolerant pipeline we can see an increase in power between 4-5 times. By multiplying the Defect-Tolerant pipeline values by 4 and comparing them with the full system we can derive the interconnect influence on power consumption. As can be seen, the interconnect increases the power consumption by 1.2x to 1.5x in a fully functional 4-core system. A better depiction can be seen in Figure 5.11, where the values are normalized by the Baseline numbers and the Defect-Tolerant pipeline*4 values.

5.3.3 Defect Tolerance

What remains is to make an analysis on the defect tolerance of the new architecture compared to the baseline core. We will concentrate on two metrics. (1) The average number of operational cores in a system for a given defect rate and (2) the probability of guaranteed functionality for a given defect rate or, in other words, the probability of having at least one core operational for a certain defect rate. The defect rate is defined as the probability of failure of a device with a given area. This area is the area of the baseline core in our measurements. The rate values for our measurements range from 0.1 to 0.9.

For a system using the baseline core we can consider the core as the main component. With this in mind, using the various defect rates, we can calculate the probability of having a certain number of operational cores (i) for each rate. The probability of having exactly i operational cores for a certain defect rate P_{sd} is:

$$P(N,i) = \underbrace{\frac{N!}{i! * (N-i)!}}_{\text{Binomical coeff.}} * P_{sd}^{N-i} * (1-P_{sd})^{i} \tag{5.1}$$

Where N the number of cores in the system. With these probabilities we can then compute the average number of functional cores for different defect rates. Also by adding all P(N,i) for i equal or greater that 1 we can compute the probability that at least one core will we operational at a given defect rate or what we define as the *reliability factor*.

To compute the same probabilities for the Defect-Tolerant system we consider now the stage as the main component. As previously, the rates were considered according to the area of the baseline and so the rates need to be scaled according to the estimated area of the stage. For the sake of simplicity we consider that every stage has the same area and, thus, the same probability to fail. The rates are scaled then according to the area difference between the stage of the Defect-Tolerant and the baseline core (taking into account of course the fact that the Defect-Tolerant pipeline is larger than the Baseline).

Here we consider the array as columns of identical stages. In this case we computed first the probability of having at least M number of cores functional. This would in

turn mean that we must compute the probability of having at least M number of every different pipeline stage in the array being functional, computed as:

$$P(N,M) = \sum_{M \leq i \leq N} P(N,i) = \sum_{M \leq i \leq N} \frac{N!}{i! * (N-i)!} * P_{sd}^{N-i} * (1 - P_{sd})^{i} \tag{5.2}$$

Since we consider every stage to have the same defect rate, the P(N,M) value will be the same for all stages. The total probability of having at least M functional cores in the system would then be $P(N,M)^4$. So the probability of having exactly M cores operational in the system can be found by :

$$P_M = P(N,M) - P(N,M+1) \tag{5.3}$$

For this last value we can again, like before, compute average number of operational cores and the reliability factor. Another useful case would be the tolerance of a clustered defect-tolerant system, where cores are split to teams and stages can be shared only within the same team. This would make re-configuration less flexible but it would reduce area overheads (and thus defect probability). It would also reduce performance and power cost (i.e less switching) costs improving the efficiency of the design. The same metrics as before can be easily computed through the same equations just by considering each cluster as a separate array and summing the probabilities of the possible combinations that could occur for each case of functional cores.

So, to evaluate the defect tolerance of the various system designs we use these two metrics. The average number of operating cores in the array for each defect rate and more importantly the reliability factor that highlights guarantee of functionality. Besides the comparison between the 4-core Baseline and Defect-Tolerant pipeline it is also interesting to check another case. Since the Defect Tolerant core is so area costly is it valuable to compare it with the case of using redundant cores instead of the stage reconfigurability to provide defect tolerance. So, we also include this case in our analysis. For area equal to that of the 4-core Defect-Tolerant processors in our system, we could have a 4-core Baseline system with 2 redundant cores to provide defect tolerance, which is the 3rd option in our computations.

What we expect from the defect tolerant architecture is to have better average number of operating cores than the baseline system after a certain low defect rate and to become more efficient than the system with the 2 extra redundant cores, at some high defect rate. Indeed, the average, working number of cores for the Defect-Tolerant is higher compared to the Baseline for rates above 0.3 and also greater than the Baseline system with redundant cores at the quite high rate of 0.63 and above (Figure 5.12).

In the case of the reliability factor our defect-tolerant design manages to sustain value a close to 1 even for very extreme cases of defect rates. The Baseline system begins to have noticeable reduction in the reliability factor after 0.4 defect rate, while the redundant-cores case is quite reliable for even high defect rates and has noticeable reduction for defect rates higher than 0.63 (Figure 5.13).

As stated before an interesting case is also the idea of clustering cores within the array. This does reduce the flexibility of the design but also reduces area and performance costs

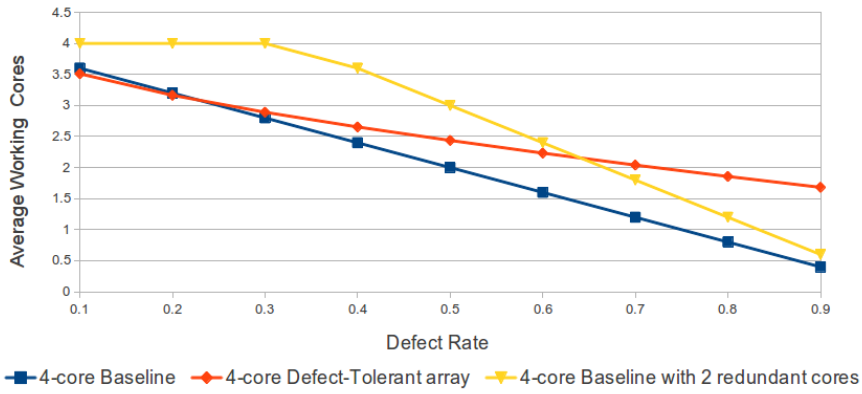


Figure 5.12: Average number of functional cores as a function of defect rate (the higher the average the more efficient the design).

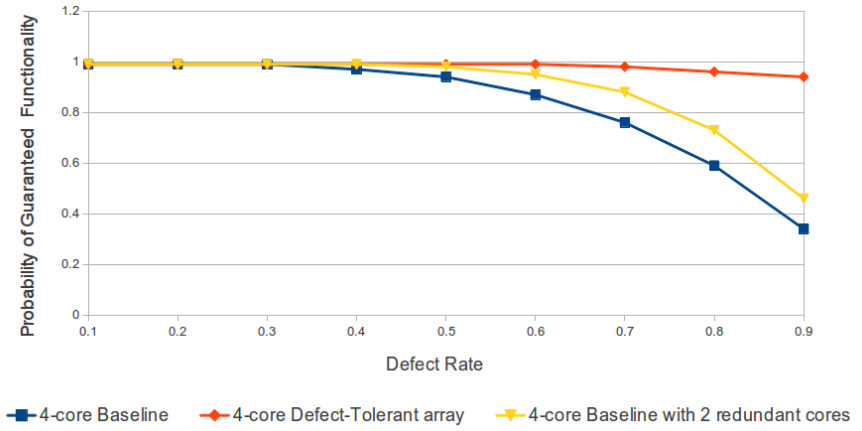


Figure 5.13: Reliability Factor. Probability of guaranteed functionality as a function of defect rate.

since the logic needed to provide the stage sharing can be considerably less. As such, it can have improved efficiency compared to the core redundancy case. Specifically, in a 4-core defect-tolerant system with the core split in two clusters of two cores, the complete processor area in the defect-tolerant cores is equal to a baseline 4-core system with one redundant core providing defect tolerance. The clustered array actually has better average functional cores than the core redundancy case for defect rates higher that 0.55 instead of 0.63 (Figure 5.14) and although there is a small decrease in the reliability factor than previously, the factor difference with the core redundancy case is noticeably larger (Figure 5.15).

All in all the defect-tolerant architecture shows significant benefits in defect tolerance compared to the baseline. Compared to the core redundancy paradigm this is true for very high defect rates. The clustering of core remedies somewhat this fact and fairs

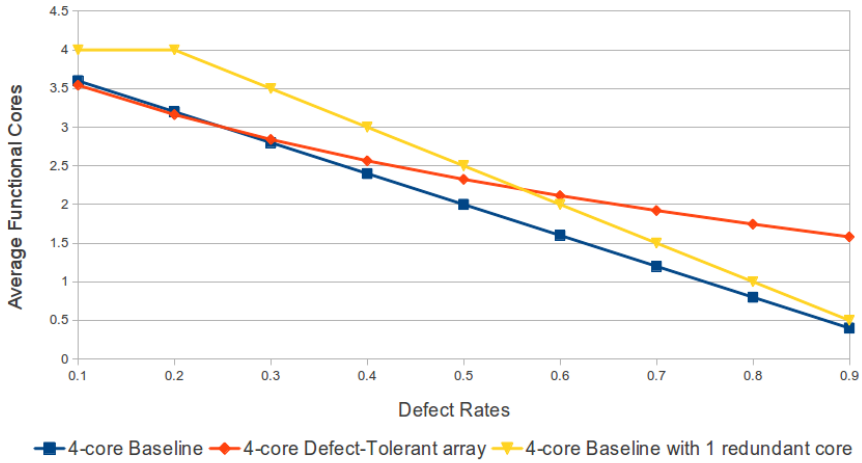


Figure 5.14: Average number of functional cores as a function of defect rate for clustered array and the respective core redundancy array (the higher the average the more efficient the design).

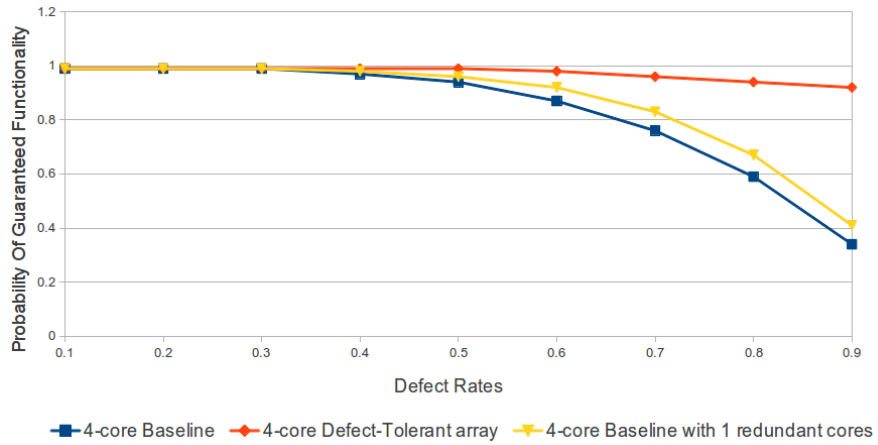


Figure 5.15: Reliability Factor for clustered array and the respective core redundancy array. Probability of guaranteed functionality as a function of defect rate.

better compared to the core-redundancy case.

5.4 Cost Estimation Functions

Using the detailed results by the synthesis it is possible to create very simple linear cost functions that could predict costs for systems with a different number of cores than 4. These, of course, are not accurate measurements but the homogeneous and symmetrical scaling of architectural changes and interconnect-logic requirements for a given number of cores make these functions a sufficient estimation and are useful as a general indication of

device overheads. These functions compute the cost in relation to the baseline RISC core. If the result of the area function is 2, for example, this indicates that the system area is 2 times the area of a system using the baseline core. The cost functions cover execution cycles (non-fault operation and worst-case configuration), clock period, execution time and area. N is again as before the number of cores in the array.

The execution cycles for the worst-case configuration pipeline:

$$C_n = \underbrace{1.016}_{\text{pipeline changes}} + \underbrace{0.33(N-1)}_{\text{extra stages before EX}} + \underbrace{0.16(N-1)}_{\text{extra stages after EX}} \tag{5.4}$$

The execution cycles for the non-fault pipeline:

$$C_n = \underbrace{1.016}_{\text{pipeline changes}} \tag{5.5}$$

Cycle-time estimation:

$$\text{Est.Cycle Time} = \underbrace{0.8}_{\text{pipeline changes}} + \underbrace{0.7}_{\text{switch presence}} + \underbrace{0.33(N-1)}_{\text{maximum steps to furthest core away}} \tag{5.6}$$

$$C_t = 1 + \frac{\text{Est.Cycle Time}}{\text{Baseline Clock Period}} \tag{5.7}$$

Area estimation:

$$C_a = 1 + \underbrace{0.173N}_{\text{pipeline changes}} + \underbrace{0.76}_{\text{interconnect}} \tag{5.8}$$

Finally, from these equations we can also derive costs for the case of core clustering, as described previously in the defect tolerance analysis. For M number of clusters in the array the functions are:

The execution cycles for the worst-case configuration pipeline:

$$C_n = \underbrace{1.016}_{\text{pipeline changes}} + \underbrace{0.33(N/M-1)}_{\text{extra stages before EX}} + \underbrace{0.16(N/M-1)}_{\text{extra stages after EX}} \tag{5.9}$$

The execution cycles for the non-fault pipeline:

$$C_n = \underbrace{1.016}_{\text{pipeline changes}} \tag{5.10}$$

Cycle-time estimation:

$$\text{Est.Cycle Time} = \underbrace{0.8}_{\text{pipeline changes}} + \underbrace{0.7}_{\text{switch presence}} + \underbrace{0.33(N/M-1)}_{\text{maximum steps to furthest core away}} \tag{5.11}$$

$$C_t = 1 + \frac{\text{Est.Cycle Time}}{\text{Baseline Clock Period}} \tag{5.12}$$

Area estimation:

$$C_a = 1 + \underbrace{0.173N/M}_{\text{pipeline changes}} + \underbrace{0.76}_{\text{interconnect}} \tag{5.13}$$

5.5 Summary

In this chapter, three were the main subjects that were discussed. First, the development and evaluation process, tool flow of our architecture and characteristics of the various tools that were used throughout the implementation of the work. Second, the experimental setup that was made in order to evaluate the outcome of the work and the goals behind it. Lastly, we made a presentation of the evaluation results including performance, area, power analysis and reliability. In terms of performance the fault-free pipeline operated 1.5 times slower than the baseline, a result of the architectural changes providing defect tolerance. The major overhead came in the form of area and power. The new core had about 69% greater area without including the interconnect. In the final system the interconnect comprised about 31% of the total system area. Power consumption was also 4-5 times higher and with the interconnect adding about 30% to 40% additional consumption. In terms of reliability the benefit to the non-tolerant system was significant but compared to the core redundancy alternative the benefit was obvious for very high defect rates. Finally, we create some simple linear cost functions for performance and area estimation for N number of cores in an array. In the next chapter we will conclude this thesis by presenting our conclusions and suggestions for future work on the subject.

6 Conclusions and Future Work

"If you want a happy ending, that depends, of course, on where you stop your story."
-Orson Welles

In the process of this thesis we tried to explore the challenges and characteristics of the architectural design of a Fault Tolerant multi-core system based on a re-configurable network. The main idea was to exploit stage granularity defect tolerance using the said re-configurable network. The motivation was the expectation that such a network if pipelined would have a reasonable overhead in overall performance. We described the architectural exploration that identified the challenges and possible solutions to them and presented the final solutions given to our design. We also presented the implementation of the array interconnect, mainly the interconnect switches and pipeline registers, and we showed an example of a 4-core array. Then we described the development and evaluation process and provided the measurement results. In this chapter we discuss some conclusions based on our evaluation and make some suggestions on future work.

6.1 Evaluation Conclusions

The results acquired through our experiments concentrated on assessing the performance overheads of our solution, that of a defect-tolerant multi-core processor using stage granularity and re-configurable interconnect. The immediate benefit of the accomplished architecture comes from the creation of a system that can guarantee graceful degradation of performance in hazardous conditions, where either manufacturing or environmental hard faults are common. Although the new architecture exhibits some slowdown when working normally and creates even slower cores in case of re-configuration, the benefit comes in the long-term performance. Through the reconfigurations, new cores are created, even if they are slower, whereas in a system using the Baseline core these would not be present and performance degradation would be rapid.

Our architectural changes, that made the implementation possible, have an overhead of about 1.9% in terms of execution cycles for our stress-marks and a timing overhead of 14.4% in terms of operation frequency. At the same time, in the final 4-core array the interconnect, including bi-directional switches and registers, has an overhead of 22.5% in operating frequency stemming from the extra logic added, which is an important improvement compared to the amount of wiring delays incurred by in the interconnect if it was not pipelined (about 43%). In total execution time, the overhead of a core in the 4-core array is 1.5x slower compared to the Baseline architecture. The main source of delay here are the architectural changes needed to decouple the stages and free them from global control and the interconnect delays. In terms of architectural changes most costs come from the bypassing mechanism. The final mechanism, although it can resolve

almost any dependency just like the Baseline core (apart from a certain type of data hazard that wastes an extra cycle than normal), is costly and present in the critical path. Extra overhead is added by the flushing, branch resolve and stall mechanisms, of course, yet it is much more limited than the overhead of bypassing.

For the re-configured cores, we have also established the slowest possible re-configured pipeline that can be setup in the system (in the presence of faults) that has a slowdown of about 3.8. This represents, though, a case in which only one core could be made functional and the whole system would not be working at all under normal circumstances, perhaps even for many faults before the one that lead to the worst case condition. The performance of all other re-configured cores lay between the performance of this worst-case scenario and the non-fault operation.

Although the performance cost of the system is variable, the considerable overhead comes in the form of area and power. The area overhead between the Baseline and Defect-Tolerant pipeline is about 69% and that has also an influence in the systems power consumption, which is higher by about 4.5 times, on average, for our stress-marks. An addition to these comes by the interconnect area and power consumption. The interconnect in a 4-core system comprises about 31% of the total area and increases power consumption by about 35% on average. Clearly, this increase in terms of the architectural changes comes from the hazard-resolution mechanisms. The register array used includes trees of comparators, MUXs and registers all having considerable area and switching activity in the duration of the operation. For the interconnect overheads we must include the influence of the code generator provided by the development tools. Although always producing code with correct functionality, because of the nature of the process of translating sequential behavior onto hardware description, the produced code is not optimal. This increases the need for area and power in the interconnect switches, that would otherwise be much lower in a more optimal code written manually by a designer. Yet, this limitation was not enough to outweigh the benefits in expediting the design process complexity provided by the Processor and Compiler Designer tools.

In terms of defect tolerance we used 2 metrics. The average number of functional cores for a given defect rate and the reliability factor, expressing the possibility of guaranteed functionality. Compared to a 4-core system using the baseline the benefits are clear. For defect rates higher than 0.3 the defect-tolerant architecture becomes more efficient with even more clear benefits for higher defect rates. In terms of reliability factor, the probability of guaranteed functionality for the baseline falls noticeably compared to the defect-tolerant one for rates higher than 0.55. Compared to the case of the redundant core system, the area overhead of our architecture shows its disadvantage. The redundant core system has very a comparable reliability factor to the defect-tolerant factor and only shows noticeable difference for rates higher than 0.65 which are extreme occasions of defects. Around the same high defect rate, is also the point where the defect-tolerant architecture has better average functional core values. Until this point core redundancy is more efficient.

All in all, the end system has some performance overhead and significant area and power overheads. Since the interconnect shows reasonable overhead and its area and switching activity is closely tied to the architectural changes, one good case for optimizations would be the processor architecture and especially the bypass mechanism.

Also, the implementation of a more optimal compiler that does not interfere with the variable number of stages in the system would also be a good bet for improvement in performance, mainly in the re-configured cores. Finally, the best case for improvement would be the optimization of the interconnect. It must be pointed out that the performance, area and power overheads are scalable according to the number of cores in the system. Since there are so much tied to the bypass mechanism and it's size is closely related to the number of processors in the system. As the number of processors increase so does the performance,area and power overheads for the system.

In terms of defect tolerance compared to the Baseline core the benefits are sufficient but the high cost of area may make the solution of just using core redundancy more efficient. Unless the area overhead is reduced, the defect-tolerant techniques shown in this processor have significant benefits for very high defect rate environments. A solution to reduce the large area effect is to cluster the array. As mentioned in the previous chapter, the loss of flexibility is largely balanced out by the reduction in the required area. Thus, compared to the respective array using core redundancy the clustered defect-tolerant architecture becomes more attractive for lower defect rates than before. An added plus is that the performance of the cores is also improved because of the reduced logic compared to the complete unclustered 4-core array.

6.2 Thesis Contributions

Through this thesis we accomplished:

- To explore the challenges and characteristics of implementing a system as described in our problem statement. A multi-core fault tolerant system, with stage granularity using a re-configurable interconnect. The challenges were concentrated mainly on three things. The *Data-Hazard Resolution*, the *Control-Hazards Resolution* and the *Removal of Global Stalls*.

- To design and implement an architecture based on the 5-stage RISC pipeline that achieves to decouple the various processor stages so as to make re-configurations straightforward and achievable without any additional application compiling overhead. Then we devised, compared and eventually implemented solutions for the 3 main challenges as described previously.

- To describe the usage of the interconnect components and implement them. The main components were interconnect switches and registers equipped with bi-directional ports implemented with tri-states.

- To implement an example of a 4-core system using our processor and interconnect designs as a functional proof of concept.

- To evaluate our multi-core with the purpose of recognizing the main overheads of the architectural changes and interconnect implementation in terms of performance, area, power consumption and reliability. One of the main questions is to conclude on the cost of the interconnect. The overhead in operating frequency is

around 22% (about 33% compared to the initial baseline frequency) due to the incorporation of the interconnect.

6.3 Future Work

Based on the experience gained while developing our architecture and the its evaluation results we can now make some suggestions on future work that can be explored based on the subject of this thesis:

- Incorporation of more advanced hardware optimizations to improve performance. A good example of this would be the inclusion of branch predictors in the system. A better prediction scheme, although it would not make much difference in normal functionality, it would greatly benefit the reconfigured cores in the presence of faults that, depending on the configuration, could pay increased cost for flushing the pipeline in the case of mis-predictions. Another example would be the use of micro-ops in the architecture that could benefit the execution of the applications and overall execution time.

- An interesting idea is the use of a better compiler that could incorporate optimizations that are not dependent on static stage number in the pipeline. A compiler for instance, that could try to diminish the RAW hazards within a certain instruction window, would have the potential to greatly benefit the performance of the reconfigured cores in the design. This could be done by a more efficient selection of registers in the instructions that would avoid conflicts if possible. Depending on the size of the instruction window, that could range from zero to the maximum number of extra stages that could be inserted in the pipeline (N-1, with N the number of cores), there would be certain trade-offs: The greater the window the greater the performance benefit, but also the greater the compilation effort, which should be very tolerable.

- An optimized interconnect switch and register to decrease the performance cost of the current implementation. Such an interconnect could influence the overall performance greatly.

- Techniques such as these presented in this thesis, may be a little extravagant for simple RISC architectures, but this work could be a stepping stone for more involved architectures. For example, the superscalar paradigm, although is more complex and has many different challenges compared to the basic RISC architecture, its inherent nature creates the potential to use its own hardware structures to achieve decoupling of its pipeline stages and implement a superscalar based, fault-tolerant core-array placed on a reconfigurable interconnect. Additionally, more advanced architectures, such as superscalar designs, have to potential to amortize/hide overhead much better than the simple RISC architecture.

- The use of extra re-configurable hardware as stage wild-cards to provide greater flexibility and defect coverage. In the interconnect, one could include a node that

has re-configurable hardware to replicate the architecture of a faulty stage, if a spare is not available. This would include a number of challenges such as how it would be added to the interconnect and the hardware's fine-grain pipelining that would be needed to balance out the performance overhead of the re-configurable hardware. On the other hand, the fault tolerance of the design should rise dramatically with the addition of such nodes.

Appendix: Tool Usage

A

This Appendix will attempt to give some more information on the tools usage, used in the CE lab for architectural development and evaluation, and mostly concentrates on simulation, synthesis and power analysis. Its intent is to be used as a manual for future use of the tools.

A.1 Design Simulation

The simulation procedure is comprised by 3 parts: The production of the vhdl code, the compiling of the C benchmarks and their compilation and simulation in Modelsim. The steps that are needed in more detail are :

- When the coding in the Processor and Compiler Designer is done and there is the need to test the design use the *Build all* command. This will create both the design described in lisa language and also the respective compiler.
- When the process is finished without errors go to the Processor Generator to generate the RTL code for your design. There are numerous option that can be tweaked such as coding strategies, asynchronous/synchronous reset signals etc.For more details see Coware Documentation.
- In *2009.1.1/bin* folder make sure you have the your C Benchmarks and the *linker.cmd* file. In the bin the 2 script files (imaginatively named script1 and script2) must also be placed. Running these with a C program as argument will compile the benchmark with the designed compiler and automatically produce the memory mapping for the Modelsim simulation. The difference is that script2 also opens up a vi terminal that shows the compiled assembly operations to be mapped in the program memory in with the user can edit before the mapping.
- Source the Modelsim files on a terminal.In our case it was : *"source /opt/applics/bin/modelsim-6.3d.sh"*
- Go into the Modelsim directory (sim_modelsim folder) of the respective Lisa project.
- Run *"./CreateMakefile"* (for generating the work library + creating the Makefile.vmake using the vmake utility)
- Recompile either by running again *"./CreateMakefile"* or by running *"make -f Makefile.vmake"*. (Not always needed)

- Run *"make gui"* for opening modelsim and a waveform for the automatically created Testbench that was made by the Processor Generator.
- To run a different benchmark re-compile it using the script files and then go again to the Modelsim folder and re-make the project for the simulator with the *./CreateMakefile* command and open the application again with *"make gui"*. You can now run the design with your compiled benchmark.

A.2 Synthesis and Power analysis

After the creation and description of the design in VHDL and testing on a simulation in ModelSim one can synthesize his design.

A.2.1 Synthesis in Synopsis DC

- Set DC environment by running the following TCL script:

  source /opt/applics/bin/synthesis-D2010.sh
- Create a DC setup file *.synopsys_dc.setup*, set all references to the technology libraries to be used. Make sure there is a copy of the DC setup file in the current design directory.
- Start DC with: *dc_shell-t*
- Compile design by running a TCL script: *dc_shell: source aname.tcl*. The TCL script (aname.tcl) contains the compile options and design constraints. These include the design files in vhdl that are used, taken by the *designlist.tcl* file that lists all the RTL files for the design. These file must be done correctly for the synthesis to finish properly.
- Write the synthesized gate netlist to a VHDL file (command can be added to script file): *dc_shell: write –format vhdl –hierarchy –output topdesignname.vhd*. The netlist will be used later for the power analysis. Make sure that the final design specified in the *designlist.tcl* is the top design for the write command to produce the correct netlist. Now that the synthesize is done, reports on area and timing are produced for all design files and written in the folder specified in the *aname.tcl*.
- Close DC: *dc_shell: exit*

A.2.2 Annotating switching activity (ModelSim)

This process is needed to produce the vcd file for every C benchmark to compute power consumption for each of them.

- Start ModelSim. Create a new ModelSim project (in a new, clean directory, e.g. C:/design/). Add the original testbench.vhd and the *topdesignname.vhd* to the

ModelSim project. Note: GENERIC statements present in the original VHDL files are removed from the netlist description. Remove GENERIC statements from the component instantiation in the testbench. Also in the new project you need to add the vhdl files that describe the memories and any extra custom made libraries alongside the memory mappings.

- The netlist file contains numerous references to logic gate components. A new library must be created to make ModelSim aware of these components. In ModelSim: File— new—library create a new library and logical mapping to it Library name: fsd0a_a_generic_core
- Close current project: File—Close Press compile, select library fsd0a_a_generic_core and compile fsd0a_a_generic_core_Vtables.vhd and fsd0a_a_generic_core_Vcomponents.vhd from the source C:/design/fsd0a_a_generic_core/
- Finally, open the project again and add the file (add to project—existing file) fsd0a_a_generic_core_VITAL.vhd from the source C:/design/fsd0a_a_generic_core/

  Note: the files fsd0a_a_generic_core_VITAL.vhd, fsd0a_a_generic_core_Vtables.vhd, and fsd0a_a_generic_core_Vcomponents.vhd can be found in one of the sub maps of the technology libraries. Also, these naming is specific to UMC90nm SP technology.
- Create a run.do script file to annotate the switching activity:

  vsim work.testbench

  vcd file ./topdesignname.vcd

  add wave -r *

  vcd add -r *

  run *becnmark duration*

  vcd flush

  vcd off

  quit -sim
- Place the .do file in the current project directory and run the script in ModelSim: ModelSim: do run.do A file *topdesignname.vcd* is created and placed in the current project directory.
- To run for different benchmark just produce the memory mapping of the new test, like it is described in the first section, and copy–paste on the old one in the Modelsim project and re-run the .do script.

NOTE: If you are compiling a design with more than one memories the netlist names them with a different name so you will have to produce identical module vhdl descriptions for the memories, name them as they are in the netlist and add them to the Modelsim project. Also if memories are not bound when compiling/simulating , even though no errors occur, make a new clean Modelsim project and try again.

A.2.3 Power Analysis in Synopsis PT

If you do the previous process correctly you should have a topdesignname.vcd file for every C benchmark tested in the Synthesis design folder. For the power analysis you need to do:

- First we must create the SAIF files, fron the VCD files, that are by Synopsis PT. Use command : *vcd2saif –input topdesignname.vcd –output topdesignname.saif.* This command requires the Synopsis DC environment set.
- Before we start Primetime, a naming mismatch between the netlist and SAIF file needs to be solved. The port, net and instance names in the VCD/SAIF file are all lower case. The netlist (VHDL) is case insensitive and may contain upper/mixed case port, net and instance names. In Primetime, the SAIF file is read and the switching activity is annotated on nets, pins, ports and cells in the current design (compiled netlist). This process is case sensitive. This, means, for a successful annotation process, all net, port and instance names in the netlist should be identical to the names in the SAIF file, and thus should be lower case. The easiest way is to convert the netlist file (topdesignname.vhd) to a fully lower case file:

  dd if=topdesignname.vhd of=topdesignname conv=lcase

  This produces a topdesignname file with no extension all in lower case. Delete/Move the previous one from the folder and add the .vhd extension on the new file.
- Create a Primetime setup script .synopsys_pt.setup and place it in the project directory. This is also where you specify which .saif will be used.
- Set the Primetime environment and start PT:

  source /opt/applics/bin/primetime-D2010.sh
- Start PT shell: *pt_shell*
- Compute power of design by running a TCL script: *source aname_pt.tcl*
- The report is written in the folder specified in the *source aname_pt.tcl*
- Repeat the procedure for each .saif file each representing on benchmark.

Note: for examples of the setup files and script files of DC and PT, see the directories of the designs of the various test designs done for this project. The setup files are ready to use for any of these designs. The script files are a very good basis for basically any other design as well, but they will need modifications.

Bibliography

[1] *Desyre project official website : http://www.desyre.eu/.*

[2] *Coware processor designer and compiler designer manual*, product version v2009 ed., April 2009.

[3] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith, *Configurable isolation: building high availability systems with commodity multi-core processors*, Proceedings of the 34th annual international symposium on Computer architecture (New York, NY, USA), ISCA 07, ACM,, 2007, pp. 470–481.

[4] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao, *Reliable systems on unreliable fabrics*, Design Test of Computers, IEEE **25** (2008), no. 4, 322–332.

[5] Aydin O. Balkan, Gang Qu, and Uzi Vishkin, *A mesh-of-trees interconnection network for single-chip parallel processing*, International Conference on Application-specific Systems, Architectures and Processors, ASAP '06, 2006, pp. 73–80.

[6] D. Bhaduri, S. K. Shukla, P. Graham, and M. Gokhale, *Reliability analysis of fault-tolerant reconfigurable architectures*, IEEE Int. workshop on Design & Test of Defect-Tolerant Nanoscale Archit. (NANOARCH), May 2005.

[7] S. Borkar, *Designing reliable systems from unreliable components: the challenges of transistor variability and degradation*, Micro, IEEE **25** (2005), no. 6, 10 – 16.

[8] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin, *Tolerating hard faults in microprocessor array structures*, In Proc. of the 2004 International Conference on Dependable Systems and Networks, 2004, p. 51.

[9] Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin, *Tolerating hard faults in microprocessor array structures*, Proceedings of the 2004 International Conference on Dependable Systems and Networks (Washington, DC, USA), IEEE Computer Society, 2009, pp. 51–.

[10] A. Christou, *Electromigration and electronic device degradation*, John Wiley and Sons, Inc., 1994.

[11] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky, *Bulletproof: A defect-tolerant cmp switch architecture*, In Proceedings of the 12th International Symposium on High Performance Computer Architecture, 2006, pp. 3–14.

[12] Siskos Dimitrios, *A co-processor for a secure implantable medical device*, Master's thesis, TU Delft, 2011.

[13] G.N. Gaydadjiev, S. Tzilis, and I. Sourdis, *Fine-grain fault diagnosis for fpga logic blocks*, Int. Conf. on Field-Programmable Technology (FPT 2010), Dec 2010.

[14] Brian T. Gold, Babak Falsafi, and Hoe James C, *Chip-level redundancy in distributed shared-memory multiprocessors*, Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing (Washington, DC, USA), PRDC 09, IEEE Computer Society, 2009, pp. 191–201.

[15] S. Gupta, Shuguang Feng, A. Ansari, J. Blome, and S. Mahlke, *The stagenet fabric for constructing resilient multicore systems*, IEEE/ACM Int. Symp. on Microarchitecture (MICRO-41), nov. 2008, pp. 141–151.

[16] S. Gupta, Shuguang Feng, A. Ansari, and S. Mahlke, *Stagenet: A reconfigurable fabric for constructing dependable cmps*, IEEE Transactions on Computers **60** (2011), no. 1, 5–19.

[17] Shantanu Gupta, Shuguang Feng, Jason Blome, and Scott Mahlke, *Stagenet: A reconfigurable cmp fabric for resilient systems*, Reconfigurable and Adaptive Architecture Workshop (RAAW), dec. 2007.

[18] John L. Hennessy and David A. Patterson, *Computer architecture - a quantitative approach (3. ed.)*, Morgan Kaufmann Publishers, 2003.

[19] J.Zeigler, *Terrestrial cosmic ray intensities*, IBM Journal of Research and Development **42(1)** (1998), 117–139.

[20] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee, *Architectural core salvaging in a multi-core processor for hard-error tolerance*, Proceedings of the 36th annual international symposium on Computer architecture (New York, NY, USA), ISCA 09, ACM, 2009, pp. 93–104.

[21] Danny P.Riemens, *Exploring suitable adder designs for biomedical implants*, Master's thesis, Computer Engineering, Department of Electrical Engineering Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2010.

[22] J. Ramos, J. Samson, D. Lupia, I. Troxel, R. Subramaniyan, G. Cieslewski, A. Jacobs, J. Greco, J. Curreri, M. Fischer, E. Grobelny, A. Georgr, and R. Some, *High-performance, dependable multiprocessor*, IEEE Aerospace Conference, 2006.

[23] Bogdan F. Romanescu and Daniel J. Sorin, *Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults*, Proceedings of the 17th international conference on Parallel architectures and compilation techniques (NY, USA), PACT 08, 2008, pp. 43–51.

[24] Premkishore Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger, *Exploiting microarchitectural redundancy for defect tolerance*, 21st Int. Conf on Computer Design, oct. 2003, pp. 481–488.

[25] Jr Shuler and Robert, *Fpgas with reconfigurable fault-tolerant redundancy*, Tech. report, NASA Tech briefs MSC-24464-1, NASA Center: Johnson Space Center, 2010.

[26] B. Skaggs, J. Emmert, C. Stroud, and M. Abramovici, *Dynamic fault tolerance in fpgas via partial reconfiguration*, IEEE Symp. on Field-Programmable Custom Computing Machines, 2000.

[27] L. Spainhower and T. Gregg, *Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective.*, IBM Journal of Research and Development **43** (1999), no. 6, 863–873.

[28] C. E. Stroud, J. M. Emmert, and M. Abramovici, *Online fault tolerance for fpga logic blocks*, IEEE Trans. Very Large Scale Integr. Syst. **15** (2007), 216–226.

[29] D. Sylvester, D. Blaauw, and E. Karl, *Elastic: An adaptive self-healing architecture for unpredictable silicon*, Design Test of Computers, IEEE **23** (2006), no. 6, 484–490.

[30] Vasileios Vasilikos, *Heuristic search for defect tolerant adaptive multiprocessor arrays*, Master's thesis, Computer Engineering Department of Electrical Engineering Faculty of Electrical Engineering, Mathematics and Computer Science Delft University of Technology, 2011.

[31] S. Vrudhula, D. Blaauw, and S. Sirichotiyakul, *Estimation of the likelihood of capacitive coupling noise*, In Proc. of the 39th Design Automation Conference, 2002, pp. 653–658.

[32] I. Wagner, V. Bertacco, and T. Austin, *Shielding against design flaws with field repairable control logic*, Design Automation Conference, 2006 43rd ACM/IEEE, 2006, pp. 344–347.

Curriculum Vitae

Georgios Smaragdos was born in Thessaloniki, Greece, on July 3rd 1983. He graduated from the 4th Senior High School (Lyceum) of Kalamaria in 2001 with a GPA of 18.1/20.0 (Very Good). After participating in the national examinations for a university acceptance, he was admitted to the Technical University of Crete (TUC), in Chania, in the department of Electronics & Computer Engineering (5 year Diploma). The title of his Diploma Thesis was "Design Of Current Mode CMOS Integrated Filters " under the supervision of Matthias Bucher.

While working on his Diploma Thesis, in August 2008 he was admitted to the Delft University of Technology (TUD), in the Netherlands, in the MSc program of Computer Engineering, where he continues his studies until now. He continued his work of his first diploma Thesis in parallel to the Master studies. He graduated from TUC with a GPA of 7.06/10.0 (Very Good) in May of 2012. His research interests include Reconfigurable hardware, Computer Architecture, Fault-Tolerant Computing and Embedded Systems.