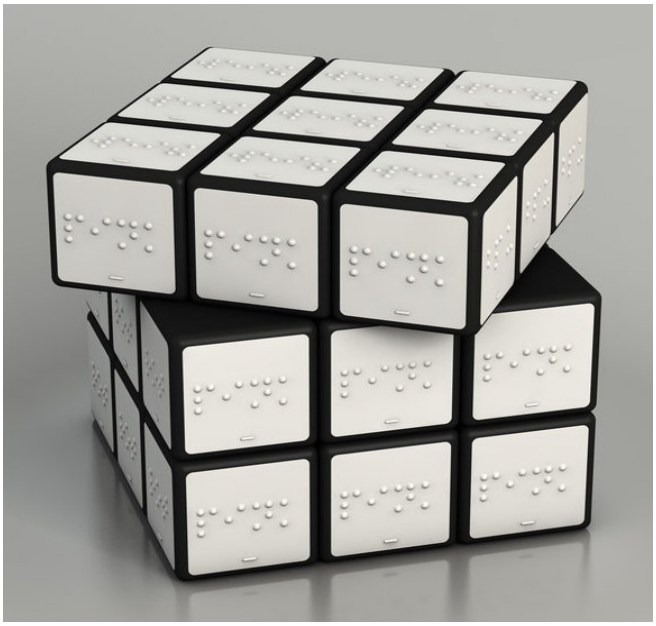
CHALMERS



A Defect-Tolerant Mixed-Grain Reconfigurable Multiprocessor Array

Master of Science Thesis in Embedded Electronics System Design

DANISH ANIS KHAN

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, September 2013

A Defect-Tolerant Mixed-Grain Reconfigurable Multiprocessor Array
DANISH ANIS KHAN

Examiner: Dr. Ioannis Sourdis

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 31-772 10 00

Cover:
Rubik's Cube for blind persons,
© http://www.konstantindatz.de

Department of Computer Science and Engineering
Göteborg, Sweden, September 2013

Abstract

Defect tolerance at chip level is currently an evolving field. With continues improvement in transistor feature size and device count the reliability of hardware has became a major concern for manufactures and designers alike. Hence a solution that can address defects at lower level while maintaining a small implementation cost could not only help in keeping the manufacturing cost low but also serves as a base for future reliable yet cost effective devices. The topic of this thesis is related to a similar approach for a RISC processor. We began from previously designed coarse grain implementation of a defect tolerant multiprocessor array and supplement it with a fine-grain "wild-card" like block which could replace any one of the defective pipeline stages in the array when required. Thus would improve the availability of the multiprocessor array at high defect rates. However by doing so the performance of implemented pipeline stages suffers from inherit gate delay of the reconfigurable substrate. Therefore the design has been modified to provide functionality at reasonable performance cost. The proposed design offers graceful degradation and in a worst case scenario exhibits a performance & power overhead of 10X and 1.75X respectively as compare to the baseline processor. With respect to area utilization the proposed fine-grain block requires 2.6X the area of single baseline processor, while in terms of availability the benefit of this approach in a 4 core coarse-grain defect tolerant array becomes apparent at defect rates above 1.3 faults per core.

Keywords: Defect tolerance, RISC Processor, FPGA, Mixed-grain, Reconfigurable, Pipeline stage sharing, RAW hazards, Dependable, ASIC, Virtex-5, Performance Improvement.

Contents

List of Figures

List of Tables

List of Abbreviation

TMR Triple Module Redundancy

DMR Dual Module Redundancy

FPGA Field Programmable Gate Array

ASIC Application Specific Integrated Circuit

ALU Arithmetic and Logic Unit

MUX Multiplexer

SEU Single Event Upsets

VHDL Very High Speed Integrated Circuit Hardware Description Language

RTL Register Transfer Language

RISC Reduced Instruction-Set Computer

SoC System On-Chip

FT Fault Tolerant

DT Defect Tolerant

LUT Look-up Table

LISA Language for Instruction-Set Architectures

Acknowledgements

I would like to begin by thanking Almighty Allah for all his blessings and enabling me to complete this task with the best of my abilities. I would like to extend my sincere gratitude to my thesis supervisor Dr.Ioannis Sourdis for his guidance, advice, suggestions and encouragements that made this thesis possible. Special thanks to George Smaragdos & Robert Seepers from Erasmus MC for their technical help with the tools and invaluable suggestions right from the beginning of my thesis work. Many thanks to Stavros Tzilis for helping me in configuration generation part and Alirad Malek in dependability analysis. It has been a great pleasure working with you all.

I am also grateful to my country Pakistan for proving me an encouraging educational environment up till my bachelor's studies and Institute of Space Technology, Pakistan for providing me the opportunity to experience new horizons of knowledge in a world class educational institution. I would like to dedicate this work to my parents who have always been my source of inspiration, motivation and determination in everything I do. Last but not least I would like to thank my family for their love and prayers and friends for their kind support and help throughout my Master's studies.

Danish Anis Khan

Gothenburg, Sweden
September 2013

Introduction 1

Since the invention of transistor in 1947 and then Integrated circuits in 1958, the world of electronics has grown exponentially while the size of a single device has shrank from centimeters to nanometers. This growth in technology and its applications has uncovered many new challenges for researchers in past decades. The most important challenge among them is to minimize the number of defects which unfortunately increases with technology scaling [1]. With current semiconductor industry trend of keeping up with Moore's law the defect rate of semiconductor devices is expected to increase further in coming years.

Besides feature size, there are other factors that affect the life span and performance of a semiconductor device such as atmospheric radiation, temperature and quality of materials used in manufacturing process etc. Design complexity is also increasing with every new functionality and tight area and power constraints. These factors push designers to compromise in reliability and the design becomes more vulnerable to manufacturing defects and unexpected failures.

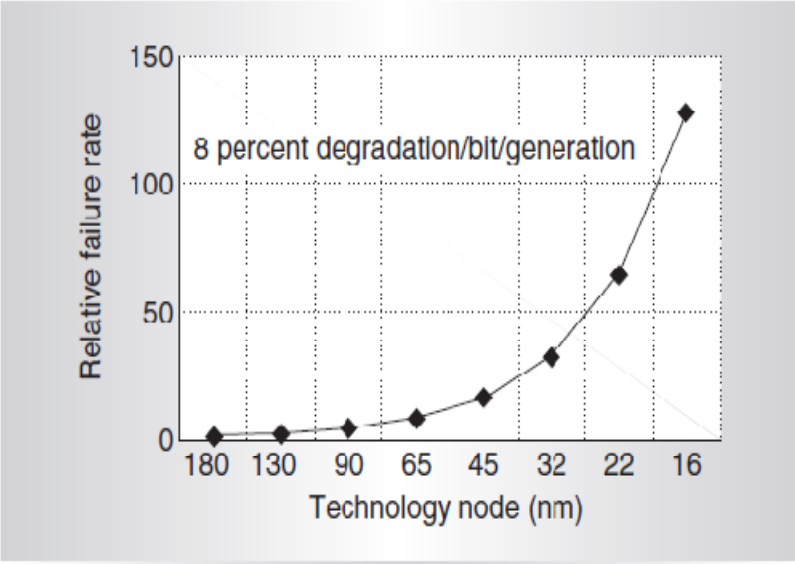


Fig 1.1 Soft-error failure rate over time [1]

In order to overcome these issues, techniques such as error detection & correction and defect tolerance are widely used in embedded designs. Implementation of such techniques is also considered mandatory in safety critical applications such as aerospace, automotive and medical lifesaving equipments. Common defect tolerance technique includes the usage

of spare components in the design which can be replaced with the defective ones when required. The focus of this thesis is also on a similar technique for RISC micro-processors. This chapter will discuss motivation behind this work, problems that are addressed and objectives of this thesis work.

1.1 Motivation

Safety critical applications require any design to remain functional even in the presence of faults; hence fault tolerance is an essential feature in all such designs. In embedded systems defect tolerance can be achieved by software, hardware techniques or their combination. Different techniques are used in each of these approaches and have their own advantages and disadvantages in terms of cost and performance of the final product. Hardware based approaches have better performance and recovery rates but the design suffers from higher implementation and operating costs, while software based techniques enjoys lower cost at the expense of lower performance and recovery rates.

An approach that exploits advantages of both hardware and software techniques can provide promising results in terms of both performance and reliability. One such technique is to detect faults with the help of software and then use spare hardware resources to replace faulty ones or to discard the faulty components altogether and utilize remaining resources to accomplish required tasks. Such techniques introduce considerable overhead on one hand in terms of area and power while on the other in terms of performance and efficiency.

In embedded systems a multiprocessor array possess inherit fault tolerance capabilities which can be further enhanced by making individual processor stages interchangeable. In this way working components in a faulty processor core will be available as spares for the remaining array. This concept of sparing of resources is exploited most recently [2] in which reconfiguration of processor pipeline stages is used in multiprocessor array as a fault recovery approach. With graceful degradation this technique offers a better performance – cost ratio in comparison to other solutions such as core level redundancy which can provide high performance but suffers badly at higher defect rates while a full reconfigurable solution such as FPGA can tolerate much higher defect rates but at the cost of extensive performance degradation.

1.2 Problem Statement

A research project DeSyRe [3] (On-Demand System Reliability) is currently under way. The aim of this project is to develop a fault tolerant SoC for embedded applications. The SoC will consist of a fault-free and a fault-prone region. The actual functionality will be placed in the fault-prone area while the overall system management is performed from the fault-free section. The fault-prone area of the chip will made from a reconfigurable substrate that supports substitution at sub-component level. One of such component that will be implemented on this reconfigurable substrate is an adaptive defect-tolerant multiprocessor

array [3]. It is a reconfigurable multiprocessor array in which pipeline stages can be shared among different processor cores through reconfigurable interconnects. The multiprocessor array consists of four 32-bit RISC processors especially designed to accommodate pipeline stage sharing.

It is desired that a fine-grain reconfigurable node is included with the existing multiprocessor array. This reconfigurable node should act as a “wild-card” and able to dynamically replace any faulty pipeline stage when required. Since this reconfigurable node will be implemented as a fine-grain FPGA like block therefore any pipeline stage implemented on this node will suffer from higher gate delays that are associated with reconfigurable hardware. In order to provide dynamic replacement of any pipeline stage and compensate for extra delay added by the fine-grain FPGA block a suitable approach needs to be defined.

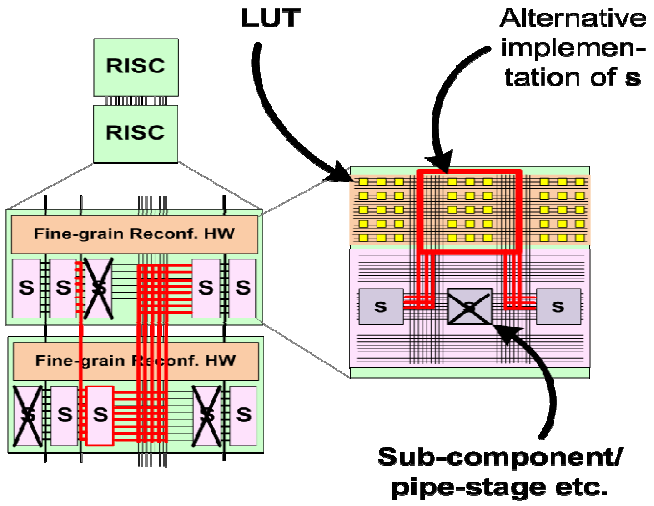


Fig 1.2 Coarse and fine-grain reconfigurable array system [3]

1.3 Objective

The aim and objective of this thesis work are as follows:

1. Design and implement a fine-grain reconfigurable block for multiprocessor array to instantiate there faulty processor parts.

2. Improve the performance of reconfigurable hardware so that the performance gap can be reduced between fine-grain implementation and ASIC based fault tolerant multiprocessor array. This is achieved by attempting to pipeline and/or introduce parallelism to the parts instantiated in fine-grain reconfigurable array.

3. Generate reconfiguration settings for multiprocessor array with fine-grain reconfigurable block to dynamically replace faulty processor parts.

4. Evaluate and measure performance, area and power overhead of the proposed solution and determine the improvement in the availability of multiprocessor array.

1.4 Method

In order to achieve the objectives mentioned in previous section within the time frame of this thesis following methods are used:

For Objective-1:
RTL modifications along with the Interconnect-Switch logic presented in [2] will be used to connect extra reconfigurable pipeline stage with the multiprocessor array.

For Objective-2:
Following two methods will be attempted to improve the performance of the fine-grain block by up to 5X of original.

1- By pipelining the stage instantiated in the fine-grain reconfigurable block.
2- By implementing multiple parallel instances of any pipeline stage in the fine-grain reconfigurable block.

For Objective-3:
This task requires modification in the source code of heuristic search algorithm that determines the number of working cores and their configuration for any defective array input. The program will provide switch settings of reconfigurable array and the type of pipeline stage fine-grain block instantiates for that particular configuration.

For Objective-4:
Xilinx X-power and Cadence RTL compiler will be used to determine power consumption of the reconfigurable hardware. Xilinx ISE can provide logic area utilization and latency information. Availability of multiprocessor array with fine-grain reconfigurable block will be determined by obtaining its fault coverage with all possible reconfigurations.

1.5 Thesis Outline

The rest of the thesis is organized as follows:

Chapter-2 Includes background information. Starting with brief summary of related research and their comparison, it then provides introduction to DeSyRe project and coarse grain multiprocessor array.

Chapter-3 Discusses implementation of fine-grain block for coarse grain multiprocessor array. It covers modifications in each pipeline stage and generation of array configuration bits.

Chapter-4 Presents methods used in the evaluation of our design and experimental results of that evaluation which include area, performance, power & dependability analysis.

Chapter-5 Provides conclusions of the work presented and give suggestions for future work and improvements.

Background 2

This chapter will talk about basic concepts behind this thesis and previous research in the same area. It includes comparison of different approaches and discusses the basics of multiprocessor array. Starting with section 2.1, where we see how substitutable resources can be used for defect tolerance, we then shed light on some related work and their comparison in section 2.2. At the end in section 2.3 the basic concept of coarse grain reconfigurable multiprocessor array is discussed in detail. The work presented in this thesis is also an extension of this reconfigurable multiprocessor array.

2.1 Sparing Substitutable Units

The concept of sparing resources to avoid faults is already common in data storage. DRAM and SRAM includes spare row and column cells which replaces defective ones when required. In general there are two main techniques that are commonly used in data storage devices at present. The first one is called Perfect Component Model [4] which handles the process of fault detection and then replace faulty components with spare ones in backend, thus hides faults from the user and the device appears fault free all the time. This approach is mainly used in DRAMs and SRAMs. The second technique known as Defect map [4] reports the faulty components to the user and their usage is later avoided by recording their location at user level. This approach is common in magnetic disk storage such as computer hard drives.

One issue with the sparing of resources approach is that it is not possible to have spares in design for all resources, power supply and clock nets for instance remain unique in the design and can be designated as non repairable resources. This problem can be overcome by minimizing the quantity of such components in the design or by improving their reliability. For this reason modern FPGAs usually have multiple clocks and IO nets that acts as spare components in implemented design. Granularity is another factor in sparing of components approach that limits the effectiveness and yield. The reason for this is twofold. First the defect rate of any component is proportional to the die area the component occupies and hence larger substitutable components means higher defect rates. Secondly a minor defect may result in discarding of entire components which are already limited in the design due to their larger size. Hence any design exploiting this approach needs to address the issues mentioned earlier.

2.2 Related Work

In this section we will discuss some previous efforts and approaches that exploit the concept of using redundant resources as spares. In section 2.2.1 we will get some insight of previous work that uses core level redundancy for fault tolerance. Next in section 2.2.2 some techniques in coarse grain reconfiguration are explored and then in section 2.2.3 we will see recent work on similar approach but at a finer granularity.

2.2.1 Core Level Redundancy

One way of having fault tolerance in processors is to use multiple cores as redundant elements. This approach has been adapted in many research articles in the past. The main challenges associated with this approach include fault detection, fault isolation, recovery and task migration from defective core. In [5] a chip-multiprocessor CMP is proposed that uses core domains to detect and contain faults. These processor domains works in DMR (Dual Modular Redundancy) or TMR (Triple Modular Redundancy) to recover from faults. In order to contain faults only in the effected domain they proposed a technique called "configurable isolation". Cores in each group shares memory controller and communication interfaces in pairs.

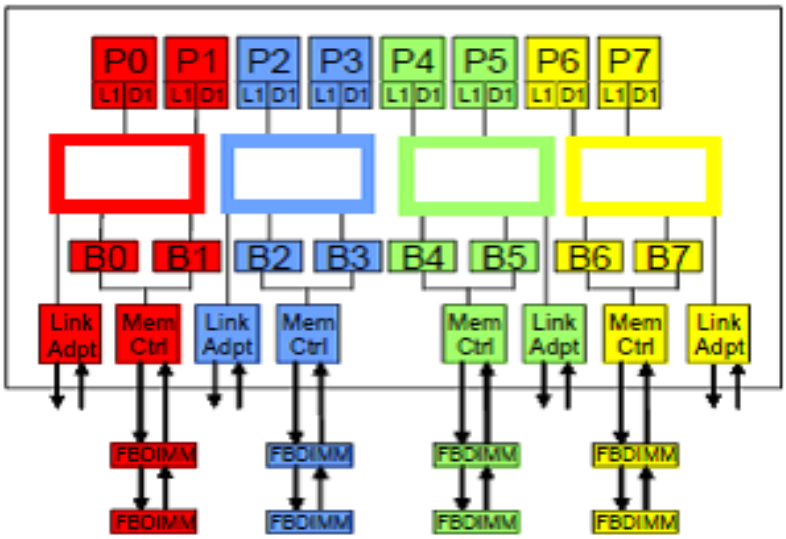


Fig 2.1 Four Fault Isolation Domains configured in TMR and DMR [5]

As an illustration of this approach Fig 2.1 shows 8 core configuration of proposed CMP in TMR-DMR mode. Each colored domain contains two cores in DMR with shared resources. These colored domains are then used in both TMR and DMR fashion to compensate for any fault when required.

Another effort involves a DMR variation called dynamic core coupling [6]. In this technique Instead of static coupling cores in CMP can verify each other's output in a dynamic manner. This reduces design overheads by avoiding dedicated communication link and spare components and improves reliability by allowing more flexible and configurable coupling among cores. This means that any two available cores can be coupled to run a new thread. In their proposed scheme, fault recovery is employed in two ways. For soft errors by backward error recovery (BER) scheme, where both cores will rollback to a last known working state. For hard faults a forward recovery scheme is used in which a third core is brought in to coupling in TMR fashion and faulty core is bypassed.

In another approach architectural redundancy is exploited to reuse a defective core in the event of hard faults [7]. The proposed technique is called "architectural core salvaging". Here in the event of hard faults the working portion of defective core remains functional. Only those tasks that can no longer be executed in the defective core are migrated or exchanged with other fully functional core in CPU. Author claimed that this technique can cover larger core die area and require only minor architectural changes to implement.

Yet another approach for core level fault tolerance is presented in [8]. It is called ElastIC architecture. It uses dynamic reliability management (DRM) scheme to deal with manufacturing defects and wear-out problems by applying voltage and frequency scaling techniques. Each processing units in this approach contains sensors to measure reliability, performance and power utilization. These measurements are then used by top software layer to initialize required bias voltages and clock delays to mask defects in that particular core. To summarize this discussion we can say that core redundancy is a suitable option for fault tolerance in microprocessors where sufficient resources, power & area budget is available. For tighter constraints and higher defect rates these approaches turn out to be less efficient and costly to implement.

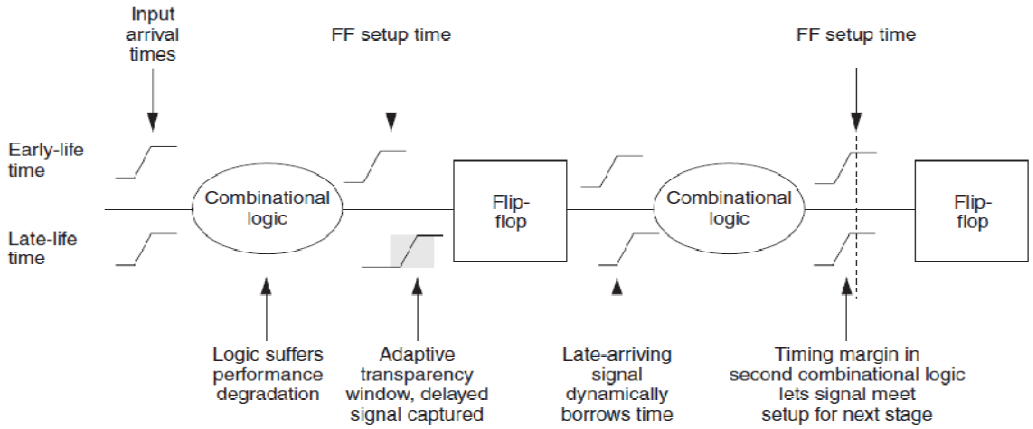


Fig 2.2 Performance degradation is mask by tunable flip-flops [8]

2.2.2 Fine-grain reconfiguration

With the advancements in programmable logic, FPGAs are becoming more viable solution for fault tolerant designs. Their inherent redundancy in the form of LUTs and multiple I/O banks provides excellent base for defect tolerance. Moreover in some solutions only their reprogramability is exploited to address manufacturing defects in integrated circuits. One such example is an approach proposed by Ilya Wagner, Valeria Bertacco & Todd Austin [9]. In this method a FPGA module is used to check control flow in a processor pipeline. The map of faulty states is loaded in the FPGA module which is then compared with processor states in each cycle and upon match the processor is switched to a low performance mode. The normal performance mode will be restored once the faulty sate is passed. As this method is solely used to counter manufacturing defects therefore defects that are need to be patched are identified by other means a priory.

In another approach the architectural redundancy of FPGA is used to recover from faults as mentioned earlier. For instance NASA's Johnson Space center proposed a reconfigurable fault tolerance technique for space applications [10]. With this technique the hardware redundancy in FPGA chip can be reduced via mode selection in order to obtain more logic area. In this way the same FPGA chip can accommodate larger designs when reliability requirements are low. In another work instead of replacing defective components with spare ones a technique to reuse faulty components is presented [11]. Here a diagnostic scheme is used for FPGA cells at a finer grain to locate and categorize faulty elements. The working sections are then utilized to perform other operations.

Dynamic partial reconfiguration is another interesting development in FPGAs. In this approach configuration of a particular portion in FPGA can be changed without causing any interruption in the working of remaining design. This technique has been exploited to recover from both soft & hard faults [12, 13]. Despite of its effectiveness this technique is expensive to implement as it requires extra control logic to detect faults in FPGA and perform partial reconfiguration to recover from these faults. This extra logic also needed to be fault tolerant in order to make the overall system reliable. From the above discussion we can conclude that FPGA based fault tolerance techniques have an edge over previous approaches mainly because of their inherent redundancy and re-programmability. But tradeoff associated with these approaches includes higher performance, area & power overheads that limits their feasibility in space & medical applications.

2.2.3 Coarse grain Reconfiguration

With core level reconfiguration some level of fault tolerance can be achieved in exchange of significantly high area and power overheads. While on the other hand by using a fine-grain reconfiguration we can obtain much better defect coverage but will have to bear higher performance losses and implementation cost. Hence any solution that leverages the features of both coarse and fine-grain reconfiguration can potentially lower the gap between high performance and reliability at a reasonable cost. One such solution is to use pipeline stages as a unit of reconfiguration in processor based designs.

This idea has been explored in [14] as StageNet and in [15] as Core Cannibalization Architecture (CCA). Another architecture that exploited this concept is called VIPER (Virtual Pipelines for Enhanced Reliability) [16] and more recently COBRA (Comprehensive Bundle-Based Reliable Architecture [17] which is an improved version of VIPER. The topic of this thesis is also based on a similar approach.

In StageNet as shown in fig 2.5; 5 pipeline stages of a general purpose processor are modified so that they can be detached when required to form new cores. Their proposed technique uses crossbar interconnects to facilitate pipeline stage sharing and borrowing. Some of the enhancements done in pipeline stages include inclusion of scoreboard in issue stage as a substitute of forwarding logic for hazard resolution. Extra program counters and register cache is used to support multiple threads on shared stages. Global flush signal is replaced by status register in each stage which allows stages to flush their contents automatically when branch miss-prediction occurs. Moreover in order to improve performance a data cache is also included in memory stage which can reduce the number of stalls required in any thread.

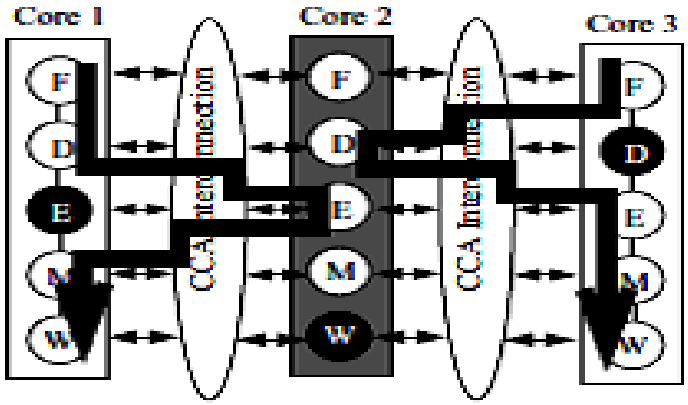


Fig 2.3 Core Cannibalization Architecture [15]

The channel width of crossbar link also has an impact on performance which is demonstrated in the paper by using three different channel widths. One last point to note about this architecture is that their proposed crossbar link imposes a constant overhead on performance, irrespective of any configuration.

The Core Cannibalization Architecture (CCA) (fig 2.3) also demonstrates pipeline stage borrowing in multi processor environment. The key differences between CCA and StageNet lies in the interconnect logic. In CCA only few cores in multi core processor are modified to be cannibalized. The rest of the cores are regarded as non cannibalizable (NC) and can only borrow stages from a cannibalizable core (CC) to repair their faults. This approach limits the flexibility of reconfiguration compared to StageNet. In their paper a number of solutions are presented to handle performance penalties due to stage sharing.

In one solution clock period is decreased to accommodate higher delay while in another solution pipeline registers are used to compromise in IPC so that fixed clock frequency can be used in all cores. Their design also uses input and output buffering in each stage to handle control hazards due to pipelining. Moreover in order to lower this overhead in performance, borrowing of stages is also limited to only single stage per faulty core. This restriction further limits the reconfigurability and as a result the defect tolerance capabilities of this architecture.

The VIPER [16] architecture (fig 2.4) offers a distributed service oriented design that consists of large number of hardware clusters connected in a mesh fashion. These hardware clusters can provide multiple services such as instruction fetch and decode etc. and together form a virtual out-of-order pipeline. This makes this design far more flexible than previously discussed CCA and StageNet. The instructions in VIPER architecture are issued in bundles to the clusters forming a virtual pipeline and these clusters are controlled by distributed control logic called BSU (Bundle Scheduling Units).

Their approach to distribute the control logic has improved the reliability of their design as compare to StageNet in which control logic is a single point of failure. Further multiple instance of same thread can be executed on independent virtual pipelines to obtain higher fault tolerance. The COBRA [17] architecture addresses the performance limitations of VIPER due to its distributed nature and offers multiple fault detection techniques to suit performance and fault tolerance requirements of the user. Their proposed architecture provides higher scalability with graceful performance degradation.

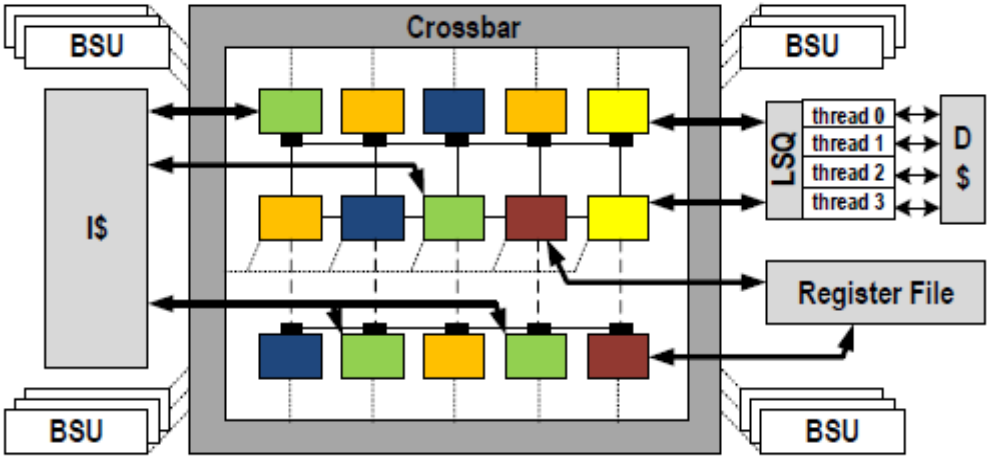


Fig 2.4 VIPER architecture [16]

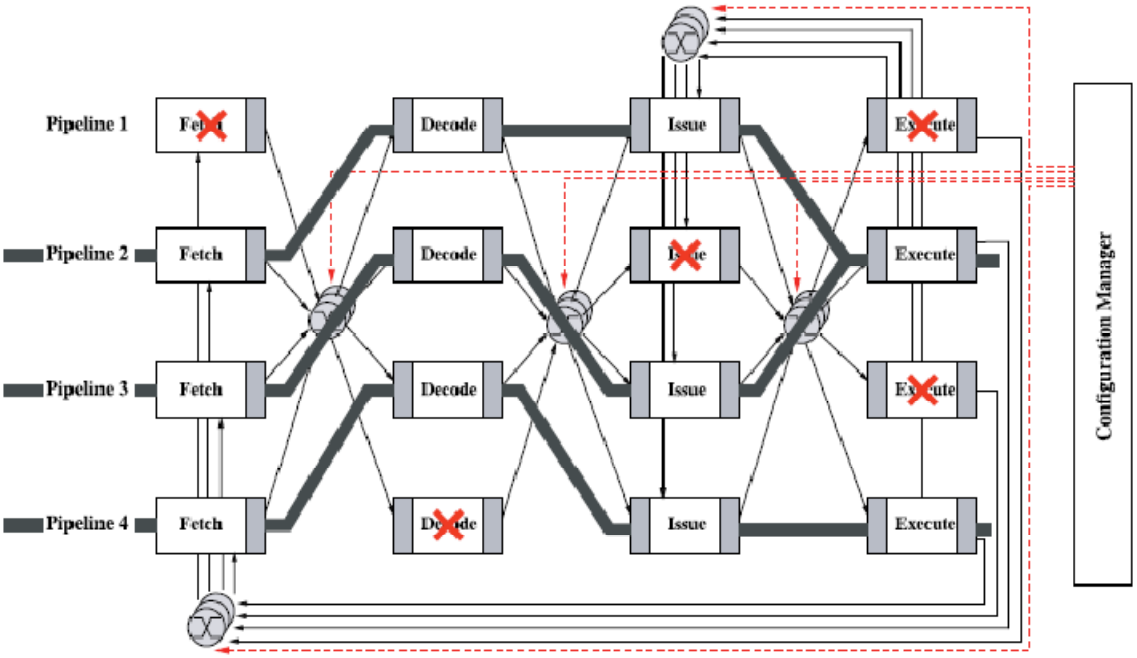


Fig 2.5 StageNet Multiprocessor Array [14]

2.3 Coarse Grain Reconfigurable Multiprocessor Array

Now that we have seen some earlier work on reconfigurable processor arrays Lets develop some understanding of the defect tolerant multiprocessor array this thesis is based on. This architecture is proposed by George Smaragdos in his Master's Thesis [2]. As it is mentioned in sec 1.2, this defect tolerant multiprocessor array is one of the sub components in DeSyRe project. In this section we will see an introduction of DeSyRe project followed by a brief summary of this architecture.

2.3.1 DeSyRe (On-Demand System Reliability)

DeSyRe is a research project sponsored by Euporean Commission Seventh Framework program to built on-demand and reliable system-on-chip (SoCs) [3]. The main aim is to produce reliable and energy efficient devices from unreliable components. In order to achieve this task a System-on-chip framework based on reconfigurable hardware substrate is proposed. This SoCs consists of multiple design levels from software to all the way down to technology substrate. These levels can be broadly classified in to physical and logical domain.

Physical abstraction deals with hardware components and is divided in to fault-free (FF) and fault-prone (FP) sections. Main functionality of the chip is place in FP region which consists of regular inexpensive components.

While the reconfiguration, communication and fault tolerance logic uses expensive FF region. Hence area of FF region is kept to minimum in order to reduce manufacturing cost. DeSyRe Soc uses a mix of fine (logic cell level) and coarse (Core level) grain hardware to provide fault tolerance at component level without compromising much on performance and power efficiency.

Logical domain organizes DeDyRe SOC in three layers. These layers are termed as:

- Component
- Middleware
- Runtime System

Component layer provides actual functionality of the SOC. It consists of multiple components that lie under FP region. This layer is also responsible for fault tolerance in individual components and uses self-checking & correcting schemes to deal with faults at low level. On top of component layer lays Middleware layer that facilitate reconfiguration of hardware and treats components as black boxes. It communicates with Runtime system in order to provide functional hardware. At the top most, Runtime system layer exists that deals with scheduling tasks and provide fault tolerance by selecting best possible task distribution that satisfies application requirements. The Runtime System is implemented in FF section of DeSyRe SOC.

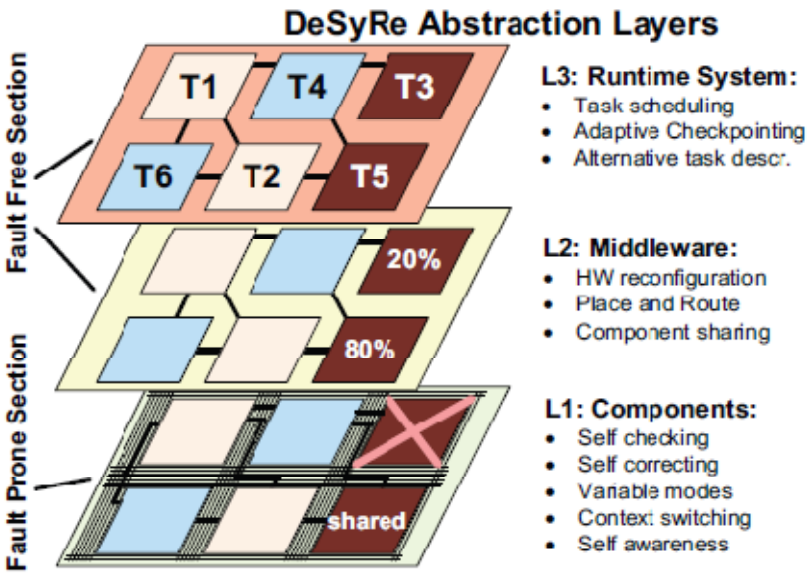


Fig 2.6 Three Abstraction Layers of DeSyRe SoC [3]

2.3.2 Array Architecture

The architecture of this multiprocessor array is similar to previously discussed StageNet and CCA. However here a reconfigurable interconnect is used to facilitate stage sharing. This approach provides higher flexibility in reconfiguration and also supports graceful performance degradation depending on number of defects.

The array consists of 4 customized 32-bit RISC processors. Each of these processors has 4 stage single issue pipeline: Instruction Fetch (IF), Decode (DEC), Execute (EX) and Memory (MEM) with Write Back (WB) combined. The Instruction set is 32-bits and can be divided in to 3 basic types: Integer arithmetic operations, memory operations and control-flow operations. The register file (RF) includes 16 32-bit registers and resides in DEC stage. The existing design has 8-kB instruction and 32-kB data memories which are part of IF and MEM stages respectively. The array is designed such that in the event of any defect the interconnect will reconfigures the array and forms new processor cores from available pipeline stages. In order to reduce wiring delay that is caused by long wires between distant cores, pipelining is used between cores. As a result of this pipelining, there are variable numbers of empty (Bubble) stages between the cores. The number of these bubble stages depends on the distance the data needs to travel from one core to another.

The design supports following features, which are required for proper functioning:

1- **Binary Compatibility:**
   Reconfigured cores can execute same binaries without recompilation.

2- **No Global Signals:**
   No global signals are used for pipeline data hazard resolution.

3- **Reconfiguration only via interconnect:**
   Reconfiguration is solely implemented by means of interconnect switches hence no architectural changes required.

In order to handle pipeline data hazards in the absence of global signals a conflict table is used in EX and MEM stages. This table keeps record of all uncommitted instructions processed by each stage. The record is stored in a FIFO fashion and contains information about instruction type and result if there is any. In this way any subsequent instruction that requires an uncommitted value can get it from this conflict table without the need of any forwarding path. Uncommitted values in MEM stage can also be accessed by EX stage via MEM/EX feedback connection that is incorporated in the interconnect. IF the required value is not available in conflict table a pipeline data hazard occurs and all such hazards are resolved by flush & reload approach.

Flush & reload mechanism works by resetting the pipeline register of EX stage and reloading the same instruction in the IF stage. This effectively provides extra clock cycles to pipeline stages for processing uncommitted instructions and it is anticipated that when the

instruction is executed again the hazard would already be resolved. Hence this mechanism works here as a substitute of global stall. The same approach is also employed for control hazard resolution. In addition a single ID bit is added in the instruction stream to indicate the validity of any instruction. In normal instruction flow this bit carries same value in both IF and EX stages. When a branch or function call occurs the EX stage flips the ID bit and triggers the same flush and reload mechanism but this time with the address of the next instruction in program flow. The IF stage then also flips this bit in new instructions while all other instructions with different ID bit are flushed upon arrival at the EX stage.

The interconnect in this architecture consists of bi-directional tri-state switches and registers as shown in fig 2.8. Each switch has seven different configurations which allows signals to flow both vertically and horizontally in the interconnect matrix. Similarly interconnect registers can also pass signals vertically in either direction.

SWITCH	
Control Signal Bit Pattern	**Function**
0001	In => North , Others Not Used
0010	In => South , North => Out
0011	North => Out , Others Not Used
0100	North => South , In => Out
0101	In => North , South => Out
0110	South = > North , In => Out
All Other	In => Out , Others Not Used
REGISTER	
0	North => South
1	South => North

Table 2.1 Different configuration settings of bi-directional switch and register

As mentioned earlier the EX and MEM stages also have a feedback output for IF and EX stages respectively. Hence each of these stages has two switches at their output, as shown in fig 2.7. The destination of each of these signals can be different from other and entirely depends on the location of respective stages in the array forming a particular core.

This coarse grain reconfigurable multiprocessor array provides fault tolerance at the granularity of processor pipeline stages and it's interconnect is more flexible and comparatively simpler to implement. This array can provide a working core as long as at least one instance of each of the four pipeline stages is available. In this thesis it is used as a four core multiprocessor array.

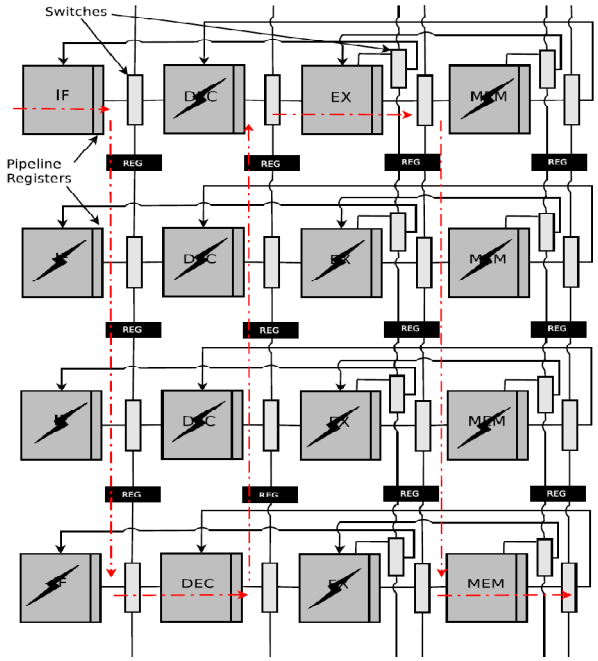


Fig 2.7 Coarse Grain Reconfigurable Multiprocessor Array [2]

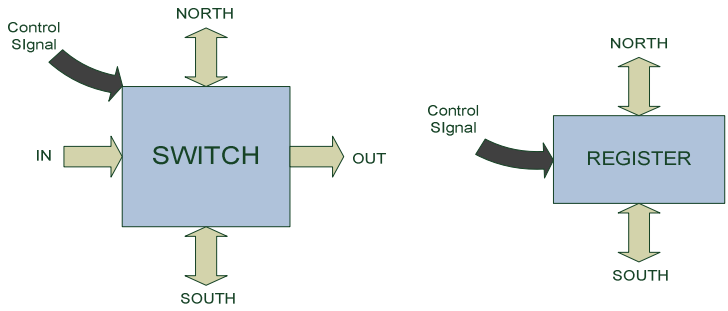


Fig 2.8 Bi-directional switch and register used in the interconnect

2.4 Summary

In this chapter essential background information was presented that will help the reader to follow coming chapters. In the beginning of this chapter some previous approaches that are related to the topic of this thesis are discussed. The differences in core level, fine-grain and coarse grain reconfiguration were presented. Later on the architecture of reconfigurable coarse grain multiprocessor array was discussed along with a short introduction of DeSyRe project. In the next chapter the design and implementation of a fine-grain wild-card for this multiprocessor array will be presented.

Mixed Grain Reconfigurable Multicore Array 3

In this chapter the architecture of multicore array from section 2.3 is discussed with the addition of mixed grain reconfigurability. The array already supports coarse grain reconfigurability via its interchangeable pipeline stages. Now we will also include fine-grain reconfiguration with this array in order to improve its defect tolerance. This fine-grain section will act as a wild card in this array and provides extra pipeline stages to form a working core. In sec 3.1 design approach for instantiating pipeline stages within fine-grain block is presented. After that in sec 3.2 we will see how the objective of improving the performance of fine-grain block by 5X is achieved. At the end in sec 3.3 solution for generating configuration bits from the output of heuristic search algorithm is presented.

3.1 Implementation of Fine-Grain Block

As it is mentioned earlier the fine-grain block is required to replace any of the four pipeline stages in the reconfigurable multiprocessor array. This level of programmability requires an approach similar to Field Programmable Gate Array (FPGA). As we know that this processor array is part of a customized SoC [3], hence actual implementation will most likely consist of a hybrid FPGA-ASIC design. However in this thesis as a proof of concept and for evaluation, Xilinx Virtex-5 FPGA is used to implement this fine-grain reconfigurable block.

For proper integration with reconfigurable array, following are the design-constraints this fine-grain block needs to achieve:

1- Existing interconnect should be used to connect the fine-grain block.

2- Implemented stages should work as any other stage in the array. Hence no modifications should be made in the existing array and DT-core architecture.

3- Instances of all four pipeline stages should be synthesizable with an FPGA, so that simulations and overhead estimation can be performed.

4- Fine-Grain area should be not more than twice the area of single DT-Core.

For the first constraint, the interconnect switch array is extended from one end so that a fine-grain block can be connected with the existing 4-core array as an extra processor core. The fine-grain block can be connected either at the top or bottom face of the array or in the middle (between two cores). The second option has a performance advantage over the first

one because the distance to the furthest core is reduced to half when fine-grain block is placed between the DT-array. Interconnect switches for the fine-grain block are configured along with the required pipeline stage externally when required. Implementation details of each pipeline stage are as follows:

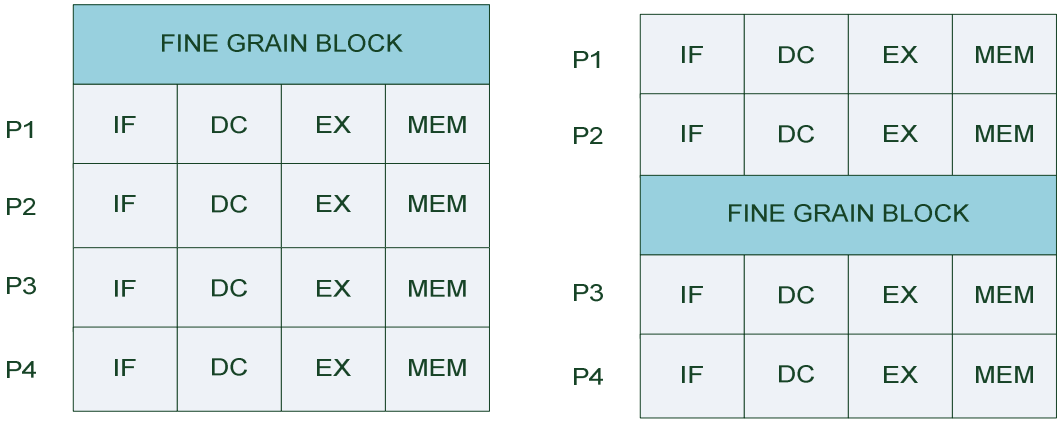


Fig 3.1 Two possible arrangements for 4-core Reconfigurable Multi-core array with Fine-Grain Block

3.1.1 IF (Instruction Fetch) Stage

The Instruction Fetch stage mainly consists of program counter (PC) and instruction memory with its supporting logic. In the original defect tolerant pipeline, registers and memory elements resides outside of detachable pipeline stages. Hence these components always remain local to the particular core. For fine-grain implementation all related logic including registers and memory elements are also needed to be present in or near fine-grain along with respective stages. In any other arrangement extra wires will be required in the interconnect which would violate our main design constraint.

While keeping these factors under consideration, The IF (Instruction Fetch) stage for fine-grain block includes:

1- IF stage of DT-pipeline
2- Program Counter
3- Instruction ID register
4- 8kB Instruction Memory
5- Pipeline Register & Interconnect Switches

Among these five components, instruction memory requires special consideration as memory implementation usually varies with technology and tool vendors. Since our design will compose of both ASIC and FPGA portions therefore memory implementation will be different in each of these sections. In order to reduce area overhead of fine-grain block it is decided to place some heavy logic components in ASIC while remaining logic stays in FPGA substrate (Virtex-5). These components include memory blocks and registers from different stages. In current implementation a 2K x 32 bit single port ROM is used as instruction memory.

For fine-grain we had two options for memory implementation. Fine-grain block can either have its own set of instruction and data memories or it can borrow them from nearest faulty stage. Since there are other ways available to protect memory from defects, it is undesirable to have an extra set of them in the defect tolerant array. Hence it is decided to incorporate extra logic in the array that will allow the fine-grain block to access instruction and data memories of adjacent defect tolerant core. Although this arrangement limits configuration options for the array but it is sufficient enough o allow fine-grain bock to replace any of the four pipeline stages of adjacent cores.

3.1.2 DC (Decode) Stage

The Decode stage receives inputs from IF (Instruction Fetch) stage and outputs to EX (Execution) stage for further processing. It is mainly composed of instruction and address decoding logic along with register file for data storage. This register file in our case includes 16 32-bit registers as mentioned in sec 2.3.2. The list of components included in the fine-grain implementation of DC stage is as follows:

1- DC stage of DT-Core
2- Register File
3- Pipeline Register
4- Interconnect Switches

In order to minimize area overhead only the DC stage from DT-core is implemented in fine-grain area. This includes all main functions of decoding stage:

1- Instruction Decoding
2- Branch address generation
3- Operand selection

All other logic including register file and pipeline register is placed in ASIC section. This logic distribution allows the DC-stage to remain under our area limits.

3.1.3 EX (Execution) Stage

The Execution stage performs the main processing. It receives decoded instructions and their operands from DC (Decoding) stage and outputs the result to MEM (Memory) stage for data saving and write-back. In our DT-core EX-stage also execute branch instructions by

feeding back the new program counter value to IF (Instruction Fetch) stage. The fine-grain implementation of EX-stage consists of following components:

1- ALU (Arithmetic & Logic Unit)
2- Branch Execution Logic
3- Conflict Table
4- Pipeline Register
5- Instruction ID register
6- Interconnect Switches

Out of these six components the conflict table, pipeline register and interconnect switches are placed in ASIC as part of area reduction approach. In order to deal with data hazards the EX-stage of DT-Core can access uncommitted instruction results from two sources, the memory feedback and conflict table. As it is stated in sec 2.3.2 the conflict table is a FIFO register that keeps record of uncommitted instructions. The size of this register depends on the maximum number of uncommitted instructions that can exist in the pipeline in any given configuration. Since the number of bubble stages varies with each configuration therefore a table that can service worst-case pipeline configuration has been implemented in the DT-array. The minimum required size for conflict table can be determined by Eq. 3.1:

$$\textit{Minimum No. of fields in Conflict Table} = 2N - 1 \qquad (3.1)$$

Where,

N = Number of processors we want to reach.

The instruction ID register in EX-stage functions as explained in sec 2.3.2. Its purpose is to differentiate between valid and invalid instructions so that the results of invalid instructions can be discarded. Instructions present between IF and MEM stages of pipeline become invalid when a branch operation is taken. In this case EX-stage feeds-back the new program counter value to IF stage via interconnect switches. This same mechanism (Fetch & Reload) is also used to handle data hazards that cannot be resolved by conflict table and memory feedback. Hence the EX-stage in reconfigurable block works the same way as it does in DT-Core (Sec 2.3.2).

3.1.4 MEM (Memory) Stage

Memory stage of DT-Core performs both data-memory access and write-back operations. It receives input from EX-stage via interconnect and it outputs to DC & EX stages as part of write-back and EX-MEM feedback operation respectively. In fine-grain version, MEM-stage includes:

1- MEM-Stage of DT-Core
2- 32kB Data Memory
3- Conflict Table
4- Pipeline Register
5- Interconnect Switches

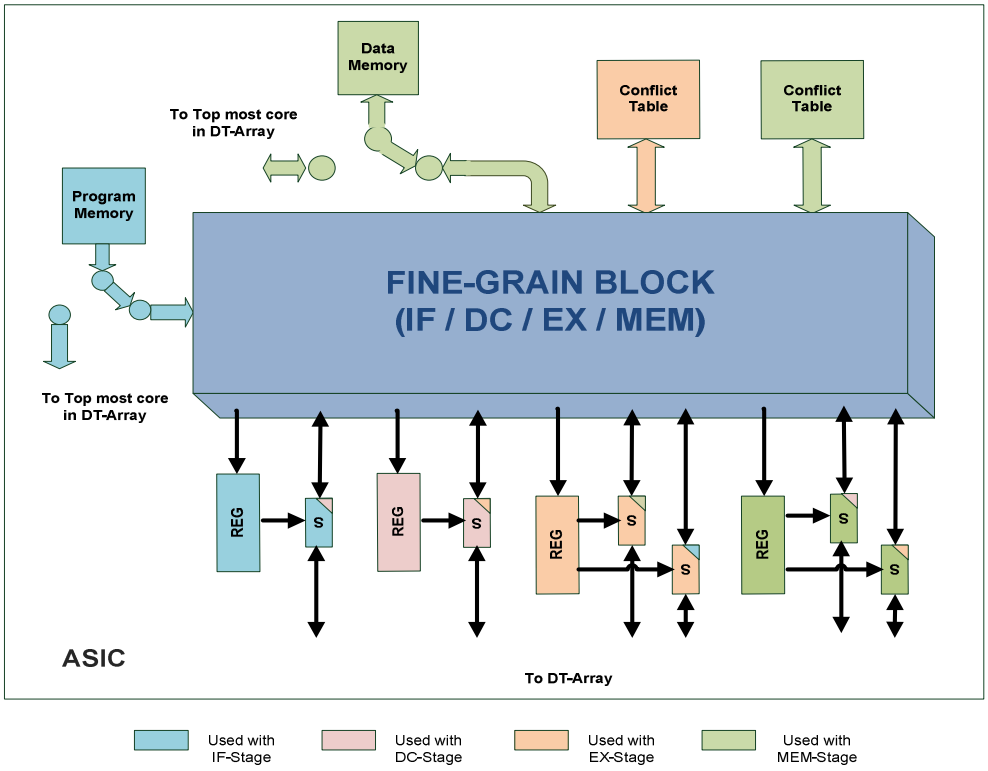


Fig 3.2 Fine-Grain Block with related components in ASIC

Data memory is the main component in MEM-stage. In current implementation it is an 8K x 32 bit single port byte-addressable RAM. Like instruction memory the data memory is also borrowed from nearest MEM-stage in the DT-Array. MEM-stage also includes a conflict table to store produced values. These values are feedback to EX-stage in order to resolve data hazards due to Load/Store instructions. The size of both conflict tables (in EX & MEM stages) is kept the same and its value is calculated from Eq.3 as shown earlier. Other than the MEM-stage from DT-core (that includes logic for Load/Store operations) the remaining components are placed in ASIC section of the SoC.

3.2 Bridging the performance gap between Coarse & Fine-Grain Blocks

While reconfigurable logic brought much more flexibility and defect tolerance in the design its main drawbacks include higher logic footprint and extensive performance degradation. Both of these factors are highly technology dependent and also vary with CLB architecture

in FPGA chips. In our prototype, highest capacity Virtex-5 FPGA is used for fine-grain block implementation. This FPGA is based on 65nm technology and can work at a clock frequency of up to 500MHz.

One of the main objectives of this thesis is to improve the fine-grain block performance by a factor of 5 or more so that the performance overhead associated with the usage of fine-grain reconfiguration can be reduced. This requires some micro-architectural modifications in the fine-grain versions of pipeline stages. From the synthesis of pipeline stages in their original form (on Virtex-5 FPGA) it is determined that a 5X performance improvement would require the critical path delay to be reduced from 30ns (EX-stage) to at least 6ns in all five instances of pipeline stages.

Pipeline Stage	Critical Path Delay in Fine-Grain (65nm)
	(ns)
IF	3.76
DC	2.00
EX	29.07
MEM	12.33

Table 3.1 Critical Path delay of unmodified pipeline stages

According to table 3.1 MEM stage has the second highest delay at 12.3ns, while IF & DC stages are already far below the 6ns constraint. Due to this reason it is decided that only the architecture of EX & MEM stages will be analyzed in order to determine possible design modifications necessary for achieving 6ns or lower clock period in fine-grain block. Rest of this section will address issues related to performance improvement in fine-grain and present proposed solution for EX & MEM pipeline stages.

3.2.1 Performance Improvement for MEM (Memory) Stage

The MEM-stage requires at least 50% reduction in its critical path delay so that it can meet the clock period requirement of 6ns. After detailed analysis of its architecture the critical path is determined with the help of place & route delay reports. As it is stated in sec 3.1.4 the MEM stage includes a FIFO buffer called Conflict Table. This table registers uncommitted instructions which can be searched within a single cycle in order to resolve data hazards. It is observed that the critical path in MEM stage passes through this combinational search logic. Further it is also determined that the logic depth of search logic is proportional to the size of conflict table. Hence the impact of conflict table size on critical path delay is determined for different values of 'N' in Eq. 3.1.

Stage	N = 4	N = 3	N = 2	N = 1
	(ns)	(ns)	(ns)	(ns)
MEM	12.33	10.20	7.42	7.32

Table 3.2 Original critical path delay of MEM-stage for different values of 'N'

It is apparent from table 3.2 that the minimum critical path delay achievable with this approach is 7.32 ns. Besides this, a table that can hold at least three records is required to support data hazard resolution between adjacent cores of DT-array. Due to this constraint it is decided to limit the conflict table size to three fields (N = 2) and then apply logic level reduction techniques to further decrease this critical path delay as much as possible. There are two commonly used techniques for this purpose:

1- **Pipelining**

2- **Parallelism**

Pipelining approach is not the best choice in our case since each new pipeline level would add another bubble stage in the instruction flow, hence raises the need of larger conflict table to cover for one extra clock cycle. Therefore parallelism approach is adapted to decrease logic levels in conflict table. The conflict table search logic works by matching the required register name with the previously stored destination register names in the table. The search starts from most recent record and proceed downwards until either a match is occurred or the table ends. In hardware this functionality is produced by comparators and priority encoders.

The conflict table includes three pieces of information in each record, out of which two (Reg. Name & Instruction Type) are used to select particular record in the table. This results in cascaded comparator & priority encoder logic for each record entry. Final output of this logic is used as 'select' input of a multiplexer which selects stored register value from the third column of record. These cascaded priority encoders are generated during the synthesis process due to nested IF-ELSE conditions in the RTL. It is discovered that these cascaded priority encoders are the main cause of higher number of logic levels in conflict table which essentially makes it a critical path in MEM-stage.

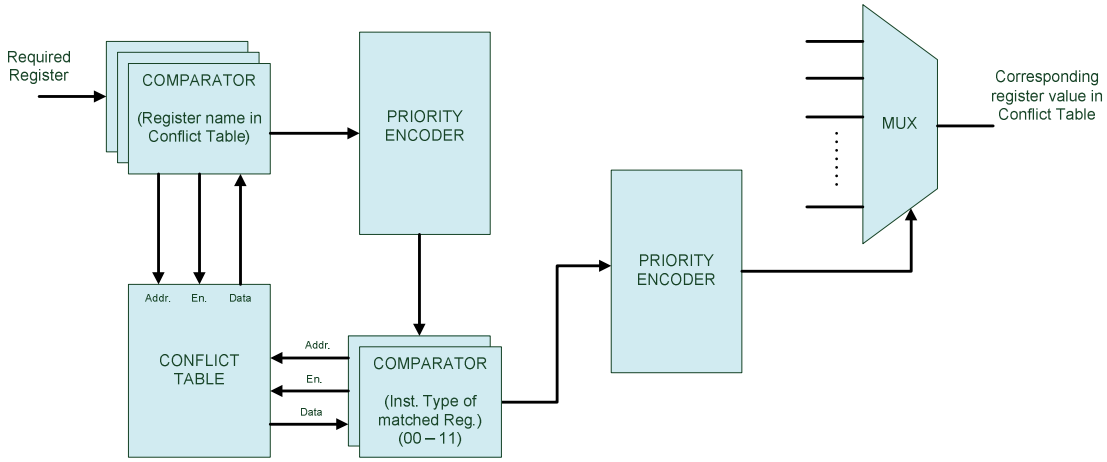


Fig 3.3 Original search logic of Conflict Table

In order to reduce this critical path, cascaded priority encoders are replaced by decoder priority encoder pairs. This approach uses “when-else” statements in VHDL code of conflict table. It is a concurrent statement contrary to IF-ELSE which is sequential. This modification results in 30% reduction of critical path delay according to synthesis reports from Xilinx ISE. Moreover delay in conflict table search logic is further reduced by eliminating address and enable signals used to access each record from other components. Instead the whole conflict table is routed to each component that accesses it. This approach results in a higher logic area due to greater number of fan-outs but the overall access time of conflict table is improved.

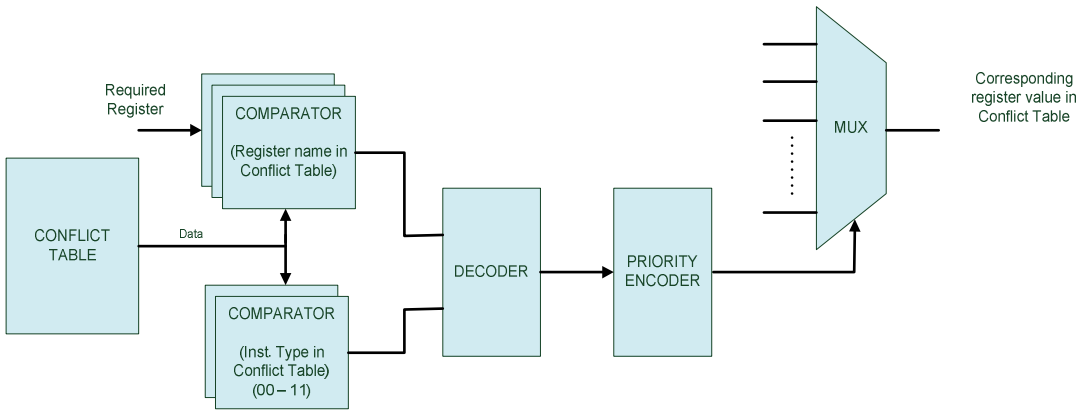


Fig 3.4 Modified search logic of Conflict Table

With all these measures the critical path delay of MEM-stage has been successfully reduced to 5ns, which is well under the original 6ns constraint. The synthesis tool (Xilinx ISE) also plays a part in achieving this value by providing timing optimizations during synthesis process.

3.2.2 Performance Improvement for EX (Execute) Stage

The EX stage exhibits highest critical path delay among the four pipeline stages as shown in table 3.1. In order for the fine-grain block to work at 200 MHz (5ns Clk period) the critical path delay in EX stage needs to be reduced by at least 83%. As mentioned in sec 3.1.3 the EX-stage also includes a conflict table which is exactly the same as in MEM-stage. Therefore same performance improvement measures are implemented for this conflict table that are used in MEM-stage as mentioned in sec 3.2.1.

Stage	**N = 4**	**N = 3**	**N = 2**	**N = 1**
	(ns)	**(ns)**	**(ns)**	**(ns)**
EX	29.07	22.78	16.50	9.85

Table 3.3 Original critical path delay of EX-stage for different values of 'N'

Since we already decided to limit conflict table size to 3 fields in previous section, therefore here we will take this as a constraint for EX-stage and try to achieve 5ns clock period. After similar modifications in conflict table search logic we are able to achieve clock period of 10.85 ns (for N =2). At this point from place & route delay reports it is determined that new critical path passes through ALU block in EX-stage.

Detailed study of EX-stage architecture revealed that parallelism in micro-architectural level cannot further improve the critical path delay. The ripple carry adder in ALU uses fast carry chains available in Virtex-5 FPGA, hence it is already faster and smaller than any carry tree adder at 32-bits [18]. Further the 32-bit shifter in ALU (VHDL 'sll' & 'slr' operators) already uses multiplexers to perform shifting operations, hence cannot be optimized further. Due to these reasons we decided to attempt following two approaches:

1- Pipeline EX-stage by two levels to decrease the critical path delay by 50%.

2- Double the number of execution stages to increase throughput by 50%.

By Two Parallel Execution Stages

As stated earlier in sec 2.3.2 the defect tolerant pipeline we are using is single issue only therefore in order to obtain benefit of having two execution units we have to include a demultiplexer at the input and a multiplexer at the output of dual EX-stage setup. Instructions from decoding stage are then feed to each of the execution stage on alternate clock cycles. In this way theoretically the throughput of EX-stage that is running at half the clock frequency of other stages can be doubled to meet our performance goal (5ns clock period).

However in order for this setup to work properly some modifications in EX-stage are required so that both stages can share uncommitted instruction values via a shared conflict table. This conflict table sharing is necessary since the result of instruction processed in one EX-stage may be required in the second one on subsequent cycles due to alternating instruction issue scheme, as shown in fig 3.5.

Moreover in case of stalls & branch operations the clock cycle penalty will be similar to a pipelined design. This is due to the usage of "flush & reload" mechanism to handle not only branch operations but also stalls (Sec 2.2.3). With all these issues there seems to be no apparent advantage in this approach over pipelining. Instead the area utilization will be more than doubled which not only increases the power consumption but also exacerbate the defect probability.

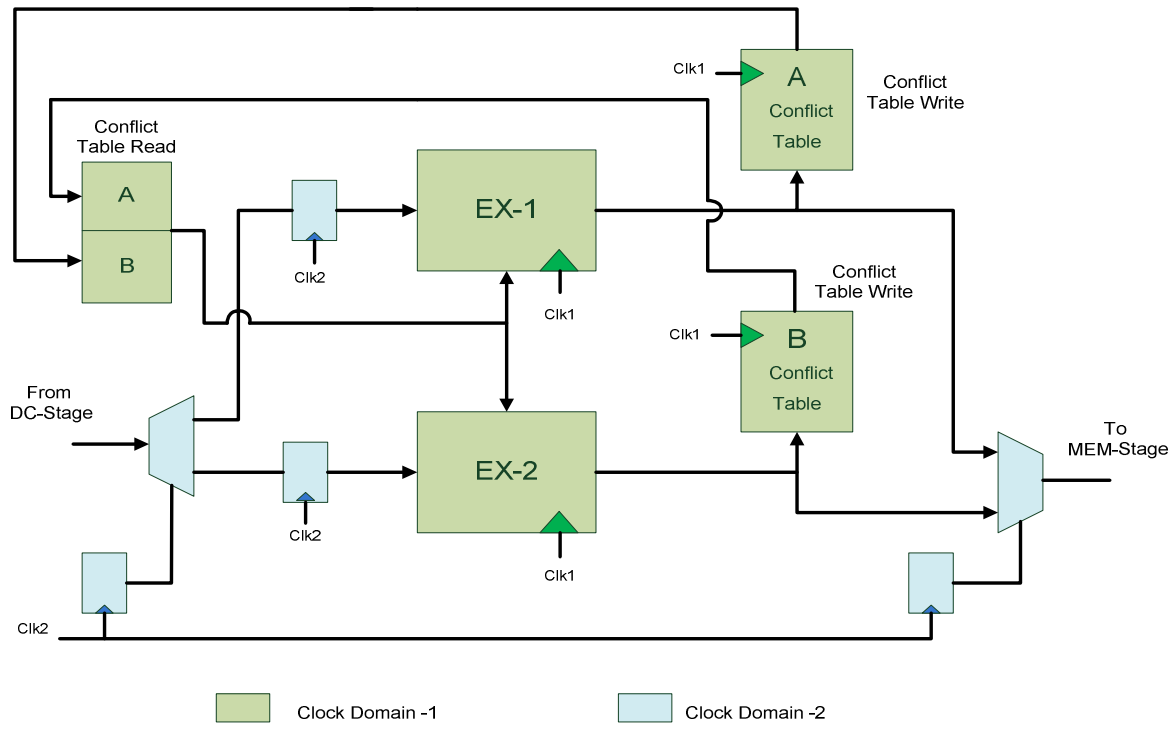


Fig 3.5 Performance improvement of EX-stage by parallelism

By Pipelining the Execution Stage

Another option to decrease critical path delay is by pipelining the design. Since we only require reducing the critical path delay of EX-Stage by 50%, therefore two-level pipelining is sufficient to achieve our target of 5ns. In the following text we will describe how EX-stage is pipelined and how new data & control hazards due to pipelined EX-stage are resolved.

The process of pipelining a design involves addition of registers in the design hierarchy so that critical path delay can be broken down in to a number of levels hence improves maximum clock frequency requirement. The first step in this process is to draw a data flow graph (DFG) of the design. A DFG shows the flow of data between input and output passing through different components in the design. Fig 3.6 shows DFG of EX-stage with critical path market in red color. As we can see there are both feedforward and feedback paths in our design. Since the critical path is on feed-forward direction therefore we will use feedforward cut-set technique to determine the location of registers.

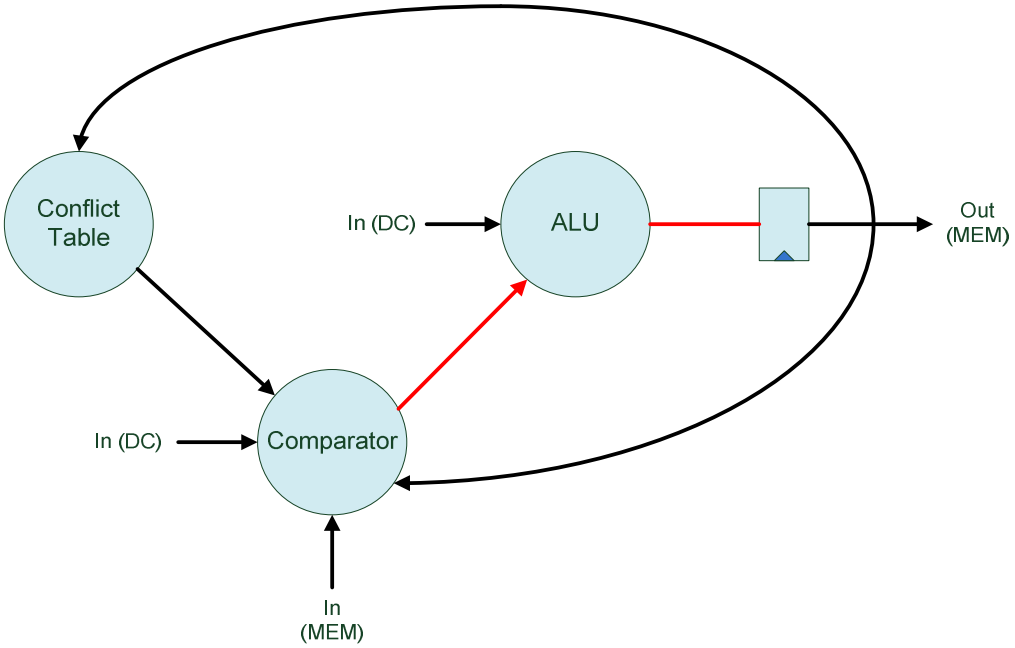


Fig 3.6 Data Flow Graph of EX-Stage-Critical Path in Red

In the second step the ALU node is broken down into two nodes. Ideally these two nodes should have exactly same latencies in order to achieve 2X improvement in clock frequency. However in reality achieving such accurate delay distribution is sometimes too complicated and may require more registers. Therefore designers usually prefer convenience over optimality when pipelining any design [19 , 20].

In our case the ALU is partitioned in to Opcode decoder and Adder/Shifter sections, which is easier to implement as compare to partitioning the adder or shifter from a specific point.

After that a feedforward cut-set line is drawn on DFG. A feedforward cut-set line is defined as:

A cut-set line that joins the edges of the graph on which data flows in forward direction such that if those edges are removed the graph becomes disjoint [19].

As we can see in Fig 3.7 the ALU is divided in to two ALU components ALU1 & ALU2 and then a feedforward cut-set line is drawn between ALU1-ALU2 & Comparator-ALU2 edges on the DFG. These are the only two forward paths in EX-stage which if removed will completely disconnect all inputs from the output node. Therefore these are the points where we will add one register each in order to make EX-stage two-level pipelined. By adding pipeline registers with this method the data coherency is guaranteed which requires that in a pipelined design all forward paths should contain equal number of registers.

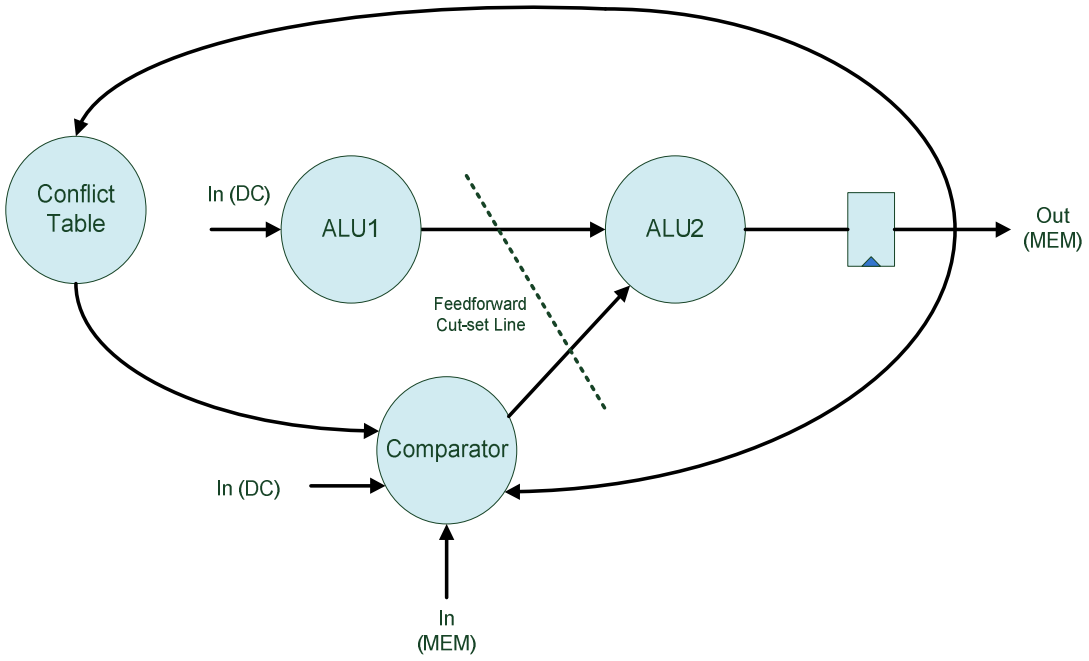


Fig 3.7 DFG of EX-Stage with split ALU & Feedforward Cut-set line

The exact location of these registers in the hardware is determined by register balancing. In our case it is automatically resolved by FPGA synthesis tool (Xilinx-ISE). Now in the third & last step registers are added at the selected locations in RTL of the design (EX-stage). It is worth mentioning here that the process of adding registers in the RTL requires careful study of component hierarchy in the RTL as well, which may be slightly different then DFG. Therefore this process may require modifications in RTL at multiple locations.

The EX-stage in our design also includes branch determination and execution logic along with the ALU. The architecture of branch logic is very similar to ALU and some of the logic is shared between these two components. Due to this reason the branch logic is also pipelined

in a similar fashion as we have seen for ALU. However unlike ALU the output of branch operations are feed to IF-stage via EX-feedback line as described in sec 3.1.3.

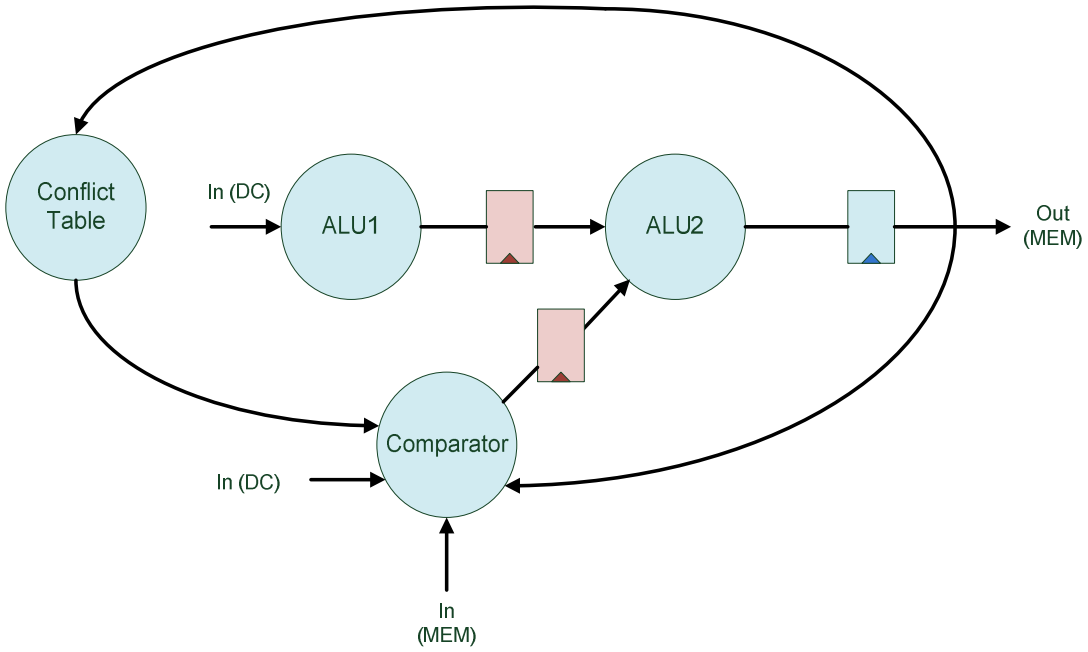


Fig 3.8 DFG of Pipelined EX-Stage

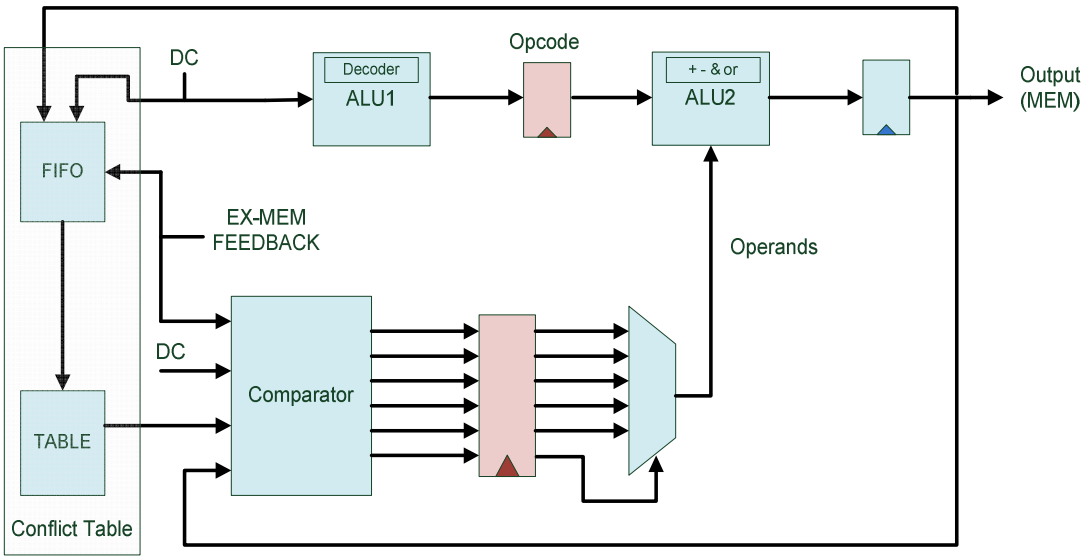


Fig 3.9 Block Diagram of Pipelined EX-Stage

3.2.3 Data Hazard Resolution in Pipelined EX-Stage

Pipelining even though improves the throughput of a design but it comes with a cost in terms of latency. Moreover in a processor based design (Instruction Pipeline) performance cost is not just limited to higher latency caused by pipelining. The addition of pipeline registers also increases data hazards as there are now more uncommitted instructions in the pipeline then before. These data hazards are also a major concern in our pipelined EX-stage. In order to deal with this issue extra hazard detection logic is incorporated at the input of EX-stage.

The original design (DT- Pipeline) doesn't have the provision to handle new data hazards due to extra pipeline stage latency. The existing stall detection logic assumes that all required values should be either available in the register file or can be found in conflict table and memory feedback line. The stall signal is asserted if required value is not available in any of these locations. This can only happen with instructions that require memory access (load/store), hence hazards caused by these operations are already taken care of in the EX-stage.

The additional hazard detection logic works as follows:

1- All incoming instructions, their operands and states of program counter & Instruction ID register are hold for one cycle using a register.

2- Potential hazards are checked only if EX-stage is not already executing a branch or initiated flush & reload operation due to previous stall.

3- Type of current and previous instruction is checked for dependency, i.e. if a RAW or WAR hazard can be caused by them.

4- Hazard Flag is set when source register of current instruction is same as the destination register of previous instruction.

As stated earlier, in our design all pipeline hazards are resolved by "flush & reload" mechanism. Hence detection of any RAW or WAR hazard will be followed by flushing of the pipeline registers and re-fetching the same instruction in the next cycle. Since until the re-fetched instruction reaches the EX-stage there will be un-executed instructions in the pipeline that were previously fetched. All such instructions needed to be flushed as well in order to avoid incorrect program flow. For this purpose a mechanism based on Instruction ID matching is already part of DT-pipeline and discussed in sec 2.3.2.

Besides the addition of hazard detection logic to handle data hazards in pipelined EX-stage, two more issues are identified in the pipelined design. One issue is that in an un-pipelined design when the branch or stall (flush & reload) operation initiates, the pipeline register at the output of EX-stage is flushed and the logic will keep it in reset state until a valid instruction arrives at the input of EX-stage. Now in a pipelined version when a branch instruction is resolved there will be an under-process instruction in the second stage of the pipeline. Hence if pipelined is flushed in next cycle the result of this instruction will be lost. This issue is solved by delaying the flushing signal for one cycle so that the result of last instruction can be saved in the conflict table before pipeline flushing occurs.

The second issue is caused by the hazard detection logic that we have implemented at the input of EX-stage. In a scenario when a hazard is detected by our additional hazard detection logic there will be an under-process instruction in the second stage of pipeline. The result of this instruction will be available at the output of EX-Stage pipeline register in the next cycle while in the same cycle flush & reload operation is also initiated. Since during flush & reload operation the contents of conflict table are not updated in original design therefore the result of last instruction will not be saved in the conflict table, instead it is expected that the register file will get updated before a valid instruction arrives at the input of DC-stage. In order for that to happen we have to delay the EX-feedback signal by one cycle so that new data can arrive from write-back path in DC stage before the required instruction is fetched.

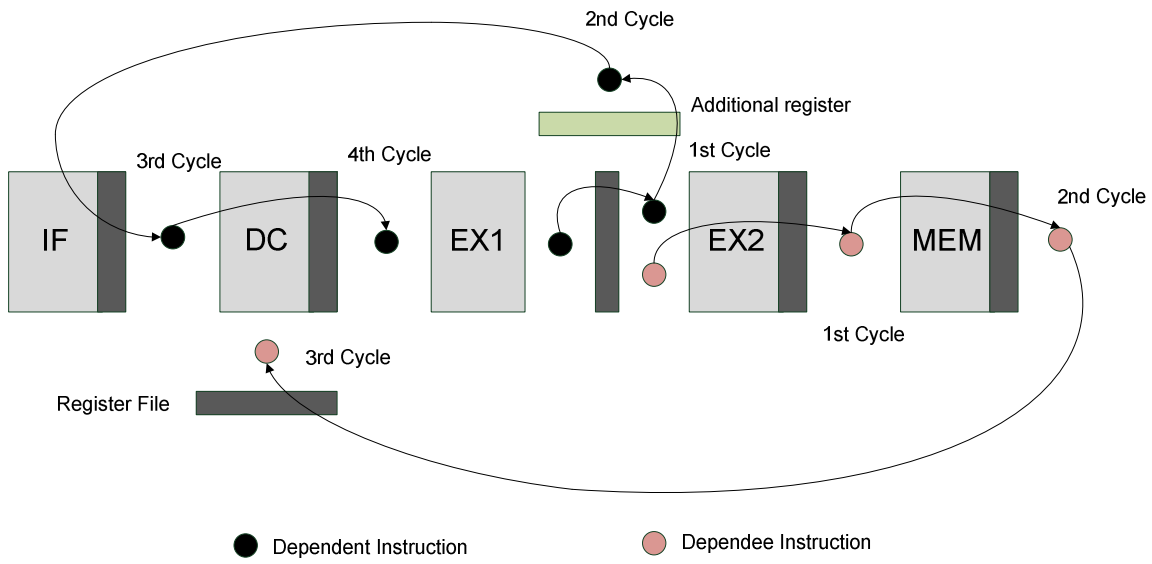


Fig 3.10 Data Hazard Resolution by delaying Re-fetch by one cycle

With this approach we would experience minimum penalty of 4 cycles whenever our additional logic detects a hazard. This penalty can be reduced by one cycle with a slightly different approach. Instead of adding a register at the output of EX-feedback line so that the DC-stage can have enough time to update register file we can store the result in the conflict table by just allowing it to shift values once during the flush & reload operation in EX-stage.

In this way even though register file is not yet updated when a new instruction arrives at DC stage, the updated value will be found in EX-stage through the conflict table and in an ideal scenario data hazards can be resolved in three cycles instead of four. We also need to flush EX-feedback output for invalid instructions like we are doing for EX-stage pipeline register. This functionality is already available in DT-pipeline and it doesn't allow invalid branch instructions to use EX-feedback line. It is part of flush & reload mechanism and its details can be found in [2].

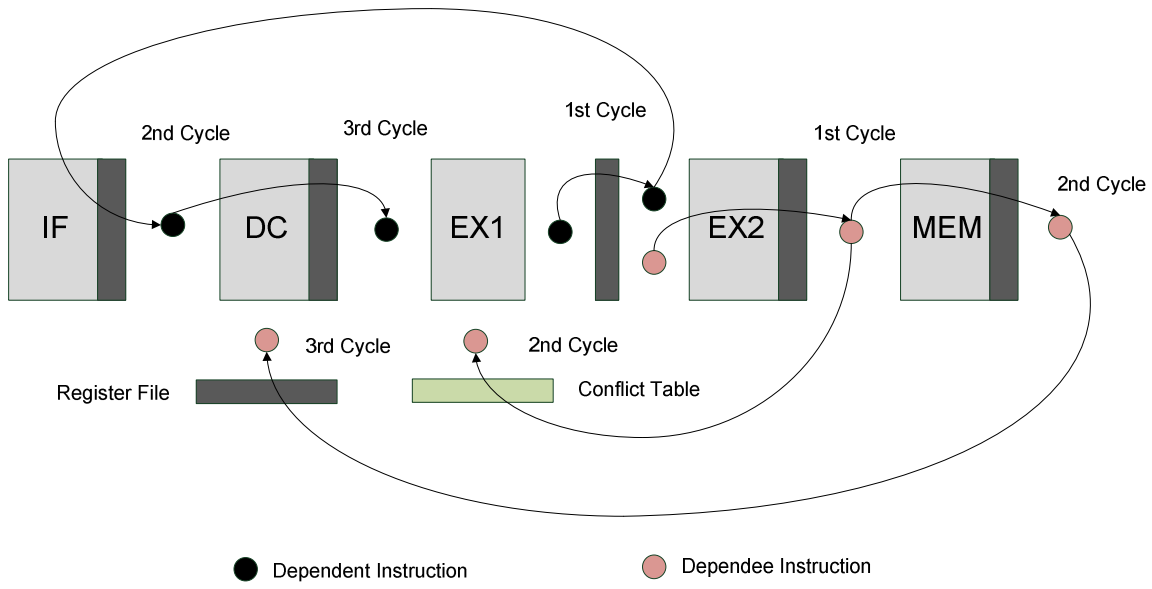


Fig 3.11 Data Hazard Resolution by allowing conflict table shifting during flush & reload operation

With these modifications the pipelined EX-stage is able to process all types of instructions available in our ISA with some performance loss. The effect on performance would get even worse as we add bubble stages in the pipeline to form new cores. This is expected as pipelining causes more data hazards and the sole remedy EX-stage has for all types of pipeline hazards is to re-fetch the instruction and flush the pipeline. Hence more flush & reload operations are expected in the pipelined version as compare to an un-pipelined EX-stage.

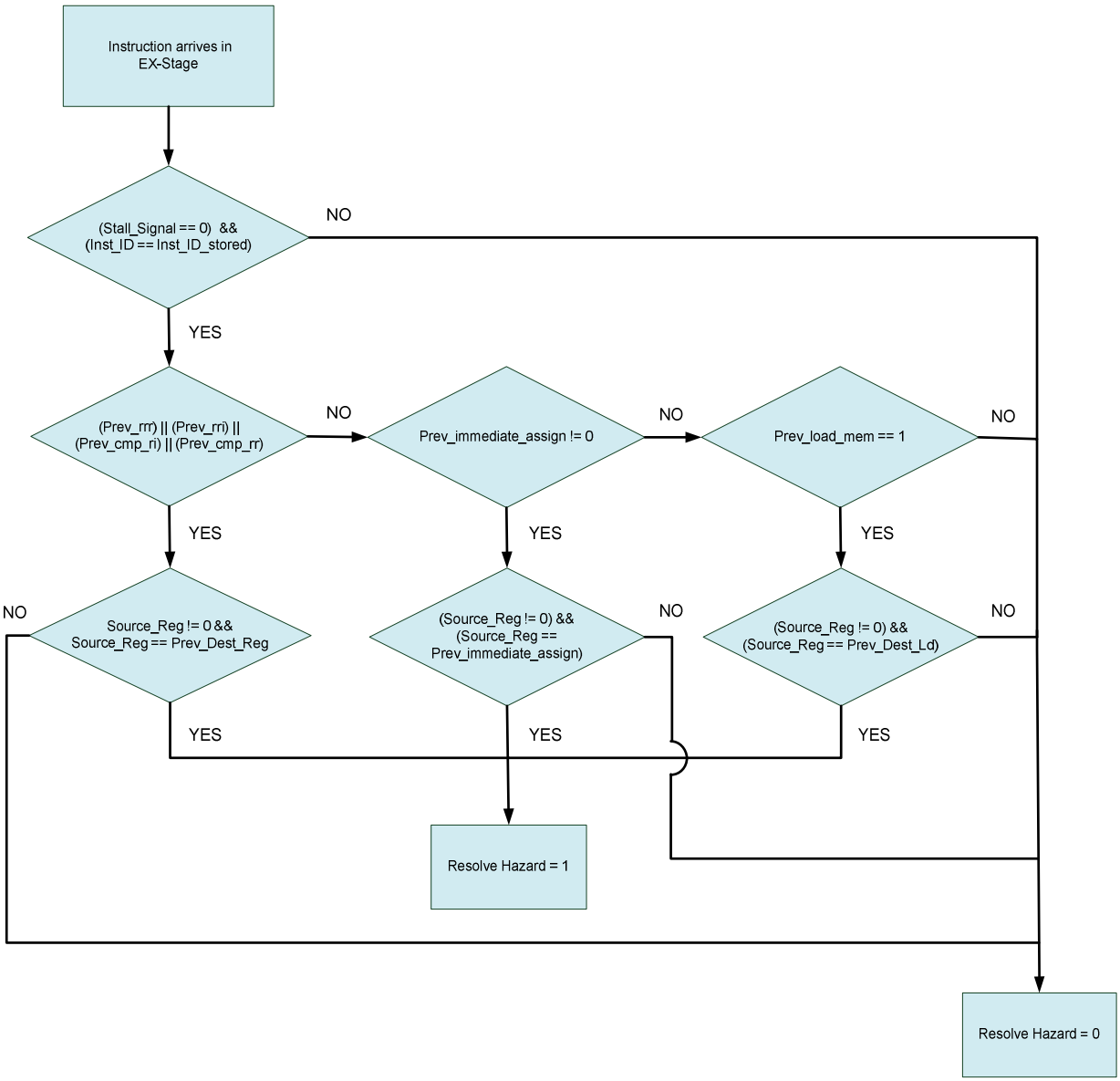


Fig 3.12 Flow Chart of Additional Hazard Detection Logic

3.3 Configuration bitstream generation for 4-core Mixed-Grain DT-Array

Up till now we have discussed how Fine-grain bock is used as a wild card to provide extra pipeline stage in DT-Array. As mentioned before the DT-array consist of 4-DT cores that supports pipeline stage sharing via interconnect switches. These switches are configured by a higher level application (Middleware layer in DeSyRe) according to the results of a heuristic search algorithm called "Greedy" [21 , 22]. This algorithm receives the location of defective stages at regular intervals and based on this information it dynamically reconfigures the DT-Array so that it can provide maximum number of working cores at any given time.

The existing implementation of "Greedy" algorithm doesn't consider fine-grain block when determining new configurations. Therefore before configuration bits for switches can be generated we have to include support for our fine-grain block in this algorithm along with the constraints mentioned in previous sections, such as:

1- Fine-grain block is implemented in the middle of 4-core DT-Array.

2- IF & MEM stages in fine-grain can only replace corresponding stages in adjacent DT-cores.

3- Fine-grain block should only be used to increase the number of working cores, hence needs to be avoided whenever possible.

4- When fine-grain block is not required, it should be bypassed in order to obtain maximum performance.

The Greedy algorithm works by starting from first available pipeline stage (from the top) in the bottleneck column and then searching for nearest available stages one after another in both left and right directions in the array until a four stage pipeline is completed. After that the same process is repeated for second working core starting from the next available stage in bottleneck column and so on. The bottleneck column is the one which contains least number of working stages. When there is no unique bottleneck column exists the algorithm considers the first discovered bottleneck column as the starting point.

Since our fine-grain block can only provide one stage at a time, it can be included with the array matrix as an extra 4-stage pipeline with only one functional stage. In this way the stage implemented in fine-grain block can be used by existing algorithm to form new cores without any major changes. In order to decide which stage should be implemented on fine-grain block if any, following approach is adapted:

1- Search for unique bottleneck column.

2- If a unique bottleneck column exists, fine-grain block will be used.

3- Fine-grain block implements the stage represented by that unique bottleneck column.

The condition of unique bottleneck column ensures that the fine-grain block will only be used when a working core can be made with its help. Since if there exist more than one bottleneck column, replacing one of the defective stages with fine-grain version won't help in building a new working core. Once the decision about fine-grain block is taken we will proceed with greedy algorithm to produce combinations for working cores based on the input DT-Array matrix. If we decided to use the fine-grain block, it will be included in the middle of array matrix as an extra core with only one working stage.

Sine in the original implementation of greedy algorithm the search for working stages always starts by looking downwards in the array; it is observed that the greedy algorithm fails to take advantage of fine-grain block in those cases when the defective stage exists on the top of fine-grain block (first two cores) in the DT-array matrix. This problem is solved by distributing 4 DT-cores in the array in to top and bottom sections. The bottleneck column of top section is determined separately. If this column is found to be same as the bottleneck column of the entire array then the greedy algorithm will start its search by first looking upwards in the array, otherwise it will look downwards as usual.

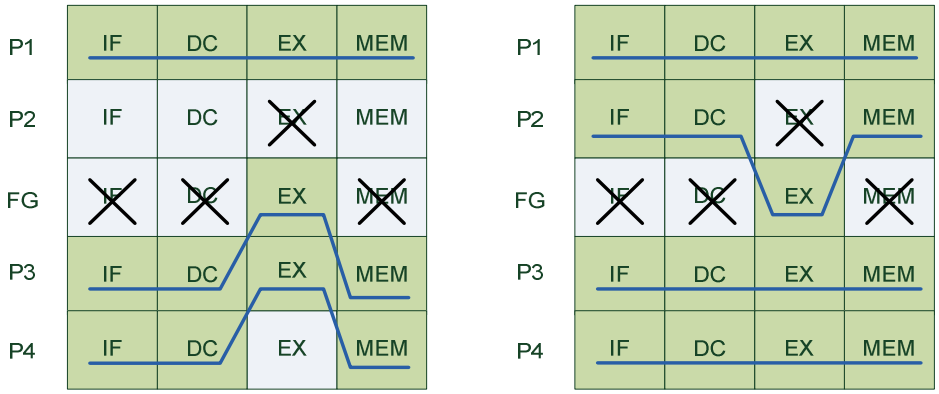


Fig 3.13 Formation of cores by Greedy Algorithm
Left-Original vs. Right-Improved

After obtaining number of new cores and their configuration in DT-Array matrix from the output of greedy algorithm we can now generate interconnect switch settings which will implement that configuration in the hardware. As it is mentioned earlier in sec 2.3.2 the interconnect in our DT-array consists of bidirectional switches and registers. The switches can be configured in seven different ways while registers have only two modes as shown in table 2.1.

In order to produce required switch settings for the array each set of pipeline stages that will form a working core are arranged in a sequence. In this sequence stages appear in pairs with stage that receives data always on the right side and the stage that outputs data on the left. As a result for any core configuration we will get a ten stage sequence, as shown in fig 3.14.

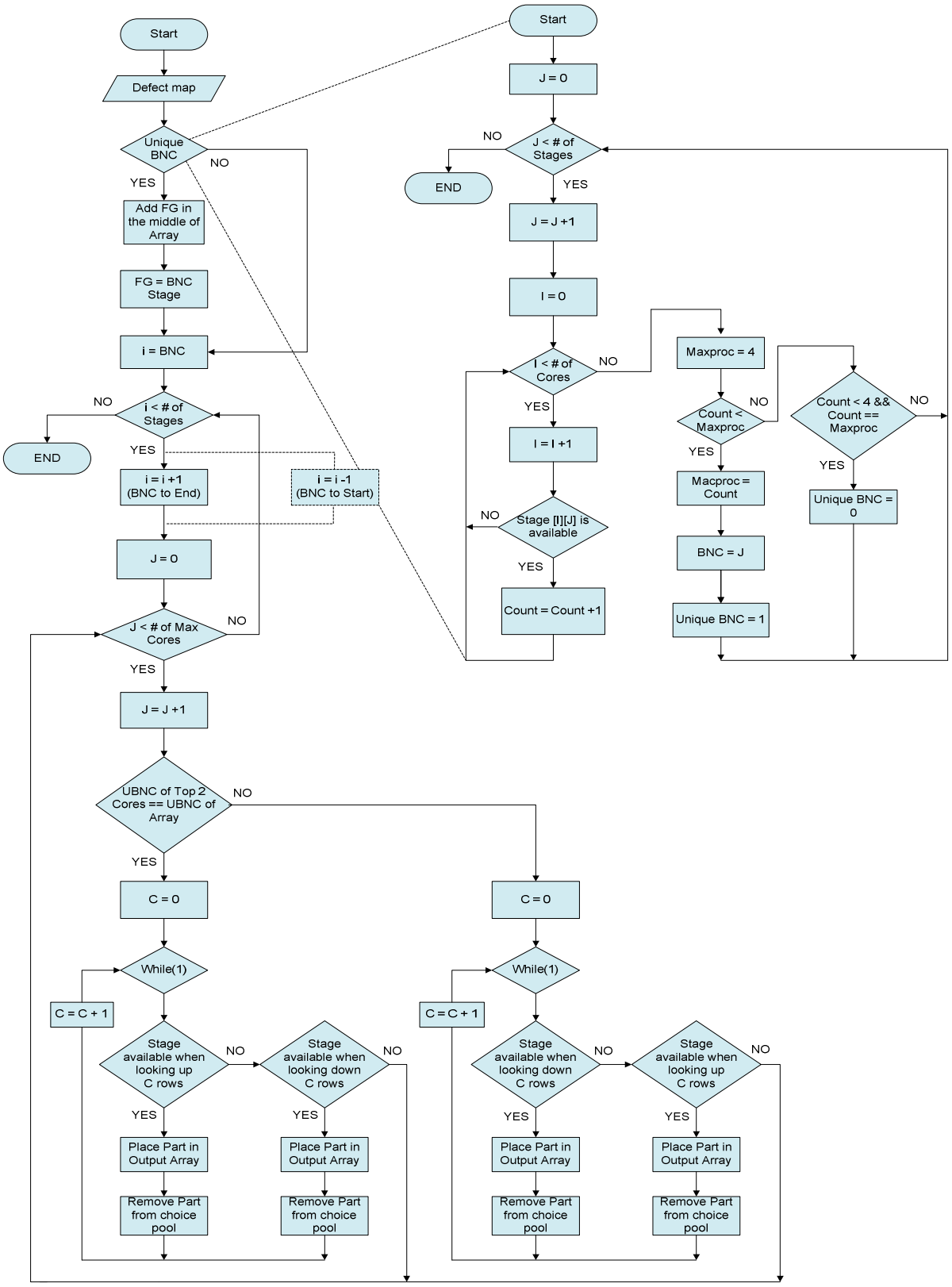


Fig 3.14 Flow chart of Greedy Algorithm

In a C-language code it is implemented as a 1-D array of size ten where each location represents a particular pipeline stage and the value stored in that location represents the core that stage belongs to.

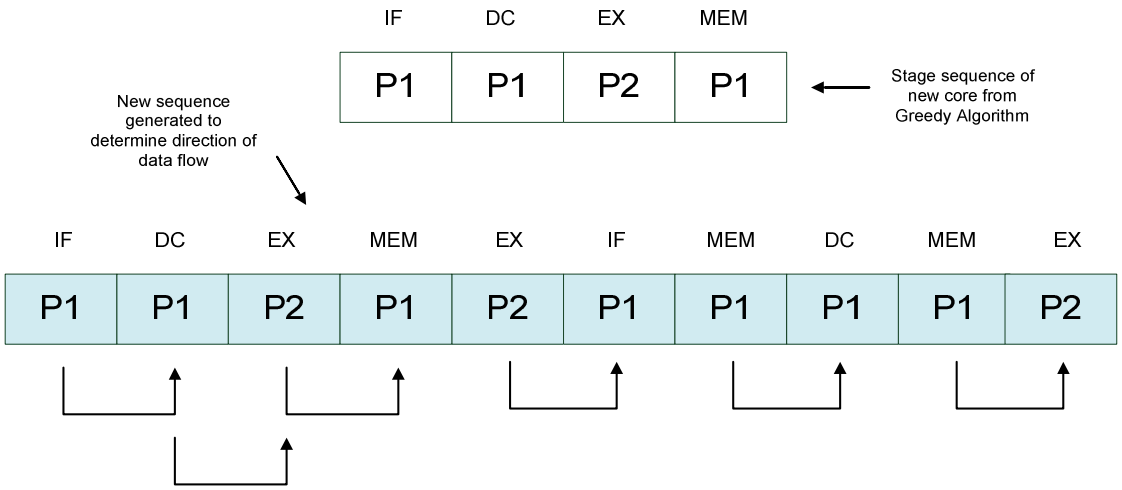


Fig 3.15 Stage sequence used for the generation of array switch settings

This sequence is then used as a true core configuration which clearly specifies the direction of data flow between stages. In the next step switches of each stage in the sequence are assigned a particular setting depending on their location in the DT-array. If a stage that outputs data is located at a higher point in the array as compare to its receiving counterpart the relevant switches are configured to allow data flow from north to south and vice versa. This whole process is repeated for every core configuration produced by the greedy algorithm along with the settings for pipeline registers in the interconnect. The assignment procedure for registers is similar to the switches and takes place simultaneously in the code.

The constraint check for IF & MEM-stage replacement is included in the beginning of the code where the decision to including fine-grain block is taken. Hence if the bottleneck column represents IF or MEM stage and the faulty stage is found to be from distant core then the fine-grain block will not be included in the array matrix. As a result the greedy algorithm will process only a 4-core DT-array and generate possible core combinations based on that.

The software to generate array switch settings is programmed in C language. It is then integrated with the existing program that takes a text file containing defect layout of the 4 core DT-array as input and produces new core combinations for it. Later on the program is updated to include the provision of fine-grain block in the algorithm and display the entire switch and register settings of 4core DT-Array along with the name of the stage that fine-grain block implements when in use.

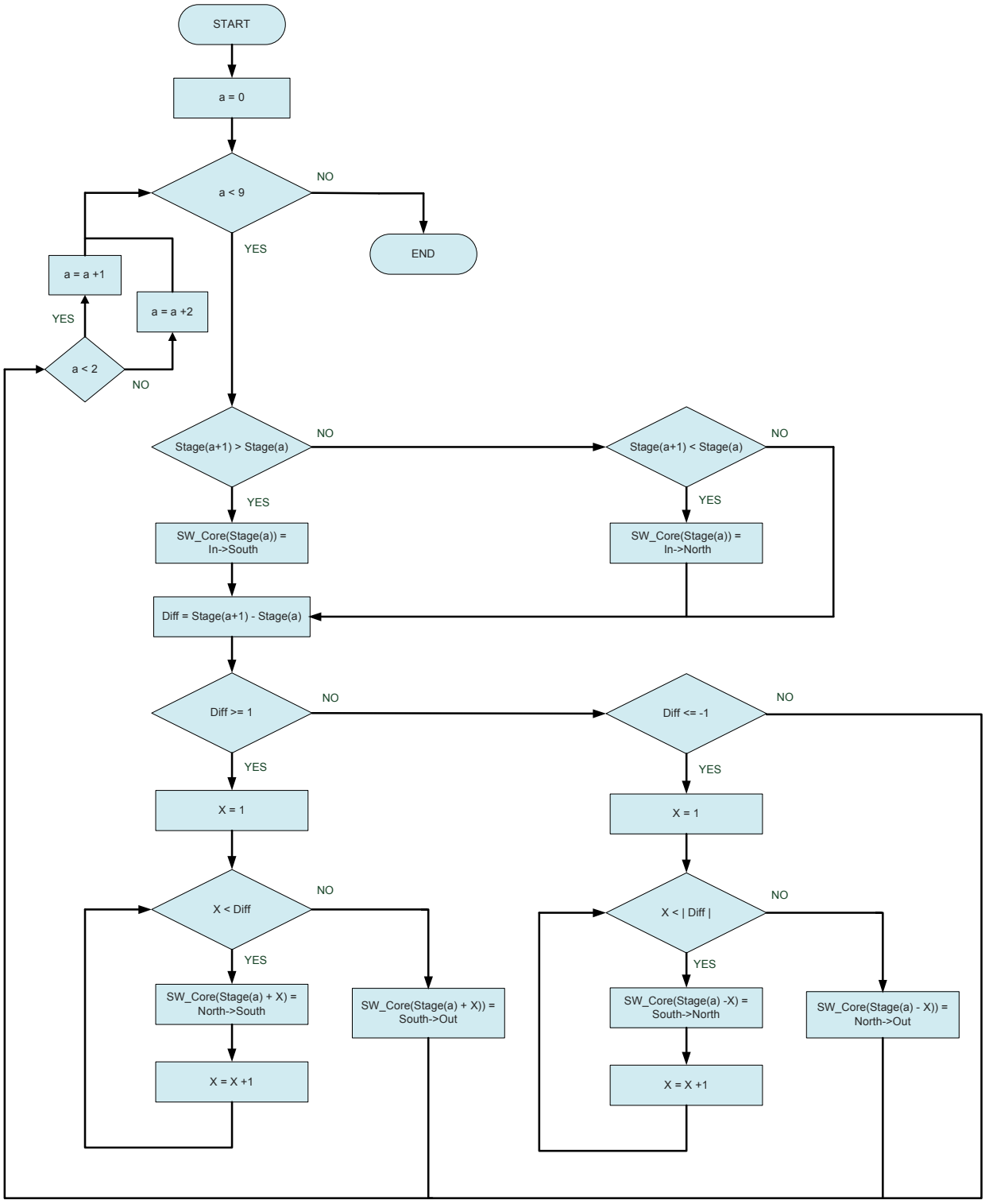


Fig 3.16 Flow chart for generation of DT-Array switch configuration

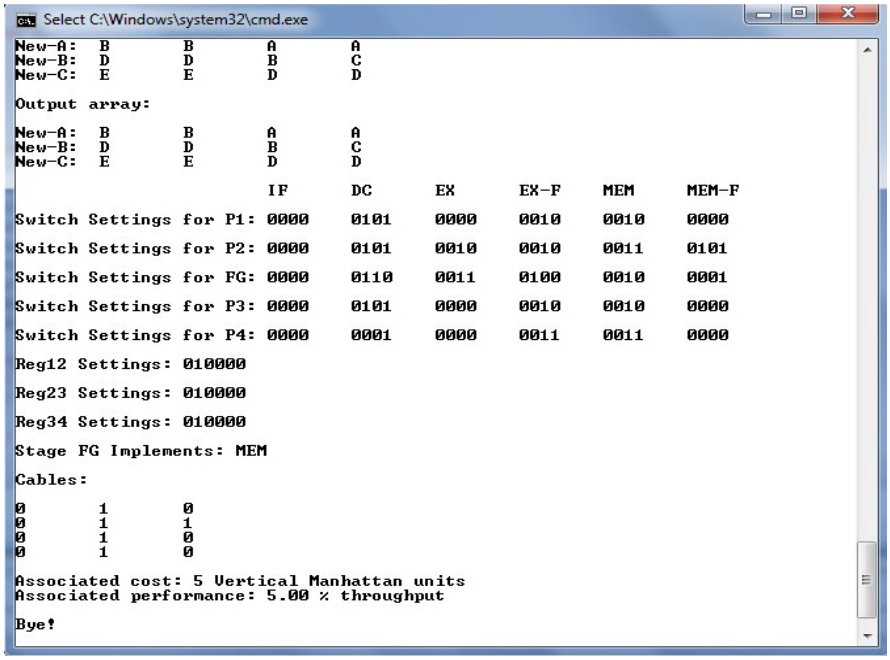


Fig 3.17 Illustration of configuration generation program

3.4 Summary

In this chapter the implementation of fine-grain block for our mixed grain DT array was discussed in detail. All related aspects regarding the placement of fine-grain block in the array and the instantiation of individual pipeline stages were covered. We have seen the realization of two different performance improvement methods, pipelining and parallelism and their impact on our design constraints. An improvement on conflict table logic was proposed to further reduce critical path delay in EX-stage. Then a solution for handling data hazards caused by pipelining was presented in detail. It was shown that with some minor modifications the hazard resolution penalty can be reduced in pipelined version. In the end the process of determining working cores in the DT-array was discussed along with the inclusion of fine-grain block and the generation of switch settings for any given array configuration. The next chapter will present our design evaluation approach and its results for the 4 core DT-array with fine-grain reconfigurability.

4 Evaluation & Results

In chapter 3 we have seen the implementation of fine-grain block for 4 core DT-Array. Now this chapter will describe the evaluation procedure adapted in this thesis to measure and verify different attributes of this fine-grain block when it is used with the 4-core DT array. The fine-grain block is evaluated for area utilization both in ASIC and FPGA , power consumption, its best and worst case performance and the improvement in availability of working cores by the inclusion of fine-grain block. In sec 4.1 we will see how the initial RTL code and C-compiler is obtained for fine-grain implementation. Sec 4.2 will talk about synthesis procedure and measurements for fine-grain area. Evaluation for performance and power consumption is presented in sec 4.3. In the end sec 4.4 includes dependability analysis.

4.1 Acquisition of initial RTL Code and C-Compiler

The original DT-Core was developed from a demo 5-stage RISC processor model in Synopsys Processor Designer suit [2]. The demo processor with some modifications was used as a baseline core in previous work and in this thesis we have also considered it as a baseline for our analysis and comparison. The need to regenerate RTL files for fine-grain version arises due to our reduction in conflict table size for performance gains as discussed in Chapter-3. The Synopsys processor designer suit uses a processor description language called LISA (Language for Instruction Set Architecture) to develop processor models which can be converted to synthesizable RTL when required. Moreover it also generates simulation files and a C-compiler for the given processor design using its compiler designer tool.

In order to change the conflict table size the LISA model of DT-core is revised in PD suit and new VHDL description of the DT-core is obtained. This VHDL code is then used to manually develop fine-grain versions of pipeline stages. The C-compiler of DT-core is also an extension of the C-compiler from baseline design. For DT-core this compiler was modified in previous work so that the delay slots in assembly code will not be filled by NOPs instructions [2]. In our analysis we decided to use the same compiler even though the compiler produced by newer version of PD (PD-2011) generates more optimized binaries. The old compiler is chosen because new 2011 version of PD lacks in the support for old processor models, due to which new C-compiler for baseline processor cannot be obtained. Hence in order to have a fair comparison we have no other choice but to use compilers generated by older version of PD suit with both baseline and DT-core.

4.2 Synthesis and Area Utilization

The actual implementation of fine-grain block is distributed among FPGA and ASIC sections as shown in fig 3.2. Therefore in order to obtain a realistic estimate of their area utilization the RTL of fine grain versions of all four pipeline stages are synthesized with both FPGA and ASIC technology using same feature size of 65nm. The area reports of both implementations are then compared to compile the actual area utilization for each stage. For FPGA synthesis Xilinx Virtex-5 FPGA is used and logic utilization is obtained from Xilinx ISE's map report in terms of Registers and LUTs, while for ASIC synthesis Cadence RTL compiler is used and logic area is obtained from its area reports in micrometer squares. The RTL compiler obtains area values from technology library files. In our case 65nm standard cell and memory libraries from ST microelectronics are used for ASIC synthesis.

In order to compare both ASIC and FPGA area reports we need to convert the logic utilization reported by Xilinx from number of registers and LUTs to a more comprehendible unit such as micrometer squares. For this purpose we exploited a research conducted on the area utilization of ASIC and ALTER FPGAs at 65nm scale [23]. The paper provides us area of a single ALTERA LUT (ALUT) in millimeter squares. From there the architecture of LUTs in both Xilinx Virtex-5 and ALTERA Startex III FPGA are compared and it is found that Xilinx Virtex-5 LUT-Register pair only takes half of the Startex III ALUT area and it is found to be about 0.55 micrometer squares. In this way we are able to compare the area utilization of our design in both ASIC and FPGA and by distributing logic components between ASIC and FPGA according to our implementation a realistic estimate regarding the area utilization for each pipeline stage has been made.

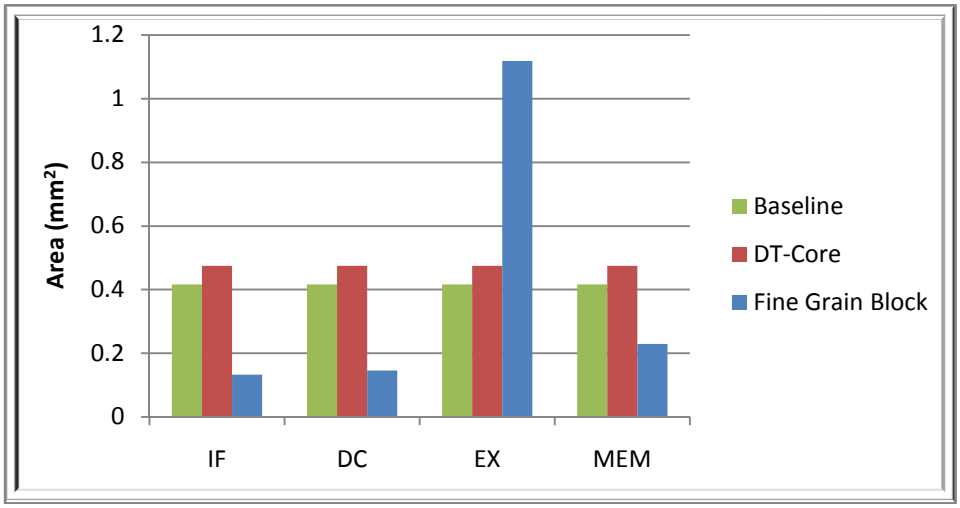


Fig 4.1 Area utilization of each reconfigurable stage in relation to Baseline and DT-core area

In figure 4.1 we can see that the EX-stage occupies more than twice the area of a complete DT-core. The area reports show that 60 % of this area is used by ALU in EX-stage alone and hence it is the main reason for having such a high area value. Since the fine-grain block is designed to replace any of the 4 pipeline stages therefore its size in the DT-array should be at least equal to the size of an EX-stage. i.e. 1.1 mm^2 approx.

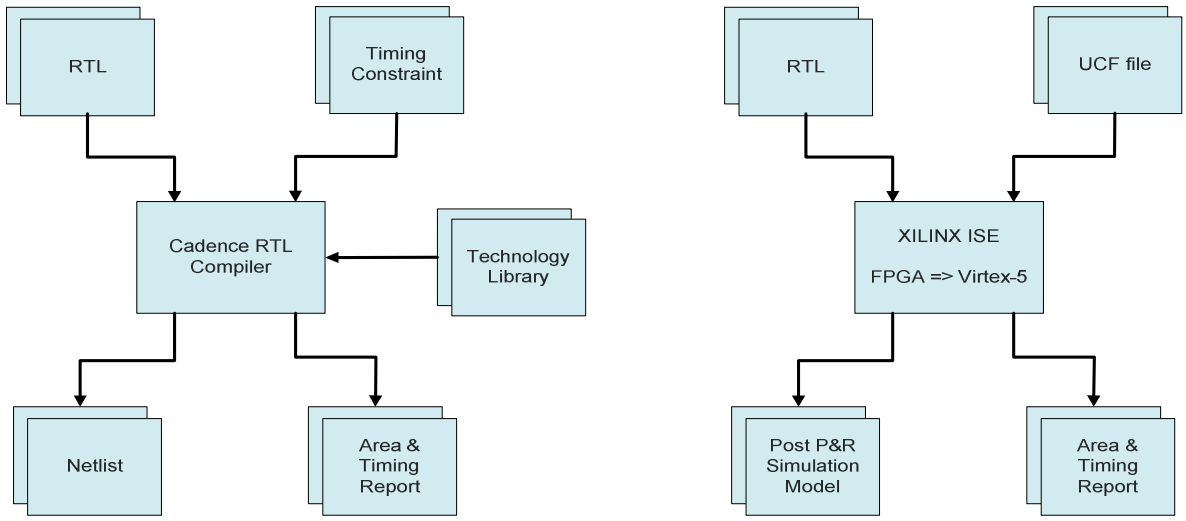


Fig 4.2 Synthesis flow in ASIC & FPGA environment

4.3 Performance and Power Consumption

In this section we will present the method and tools used to assess the performance and power consumption of 4-core DT-Array with Fine-grain block. At first we will discuss benchmark applications that are used in this analysis, and then we will describe the test setup used to acquire performance and power values. Next details of adapted procedure are presented and at the end we will see the results of this analysis.

4.3.1 Benchmarks

In order to evaluate performance and power consumption we required a set of benchmark applications to run with our design. In previous work four such benchmarks were used to assess the performance and power of 4-Core DT-array [2]. We decided to use the same benchmarks as they are equally suitable for providing reasonable workload to pipelined version of EX-stage in Fine-grain. A short description of these benchmarks is as follows:

1- **Function Argument Heavy Code:**
   This code contains function calls having a high number of arguments. It is designed to simulate data hazards due to memory load / store operations. These instructions will cause flush & reloads even in normal DT-pipeline but the penalty will be high when Fine-grain EX-stage and/or bubble stages are involved.

2- **Heavy Read After Write Conflict Code:**
   This code generates high number of RAW hazards. In normal DT-pipeline they are covered by feedback (forwarding) but with Fine-grain version of EX-stage and/or when bubble stages are involved they would cause pipeline to stall.

3- **Heavy Branch Non-Taken Code:**
   It is designed to assess branch miss-prediction handling. Like with previous benchmark the normal DT-pipeline handles these dependencies via feedback (forwarding) but with Fine-grain implementing EX-stage and/or with bubble stages higher number of flush and reload operations occurs.

4- **Normal C For-Loop Code:**
   This code includes a for-loop without body. Other three benchmarks are consisting of similar for-loops with extra instructions in their body to generate respective pipeline hazards. This benchmark represents a piece of code that is commonly found in any application. As before the inclusion of bubble stages and/or Fine-grain implementation of EX-stage is likely to stall the pipeline more often.

4.3.2 Test Setup

The fine-grain block is designed to provide pipeline stage replacement for up to two nearest processors only. Therefore we have placed it in the middle of the 4-core DT-array as shown earlier so that it can reach all four cores. Hence In order to test its functionality and obtain performance and power results we just need to simulate cases that includes half of the 4-core DT array since the values from other half will be exactly the same.

The setup will include one fine-grain block along with two adjacent DT-cores where one of the DT-Core will be directly connected to the Fine-grain block (without any intermediate pipeline registers) while the other core links to the fine-grain block via a single layer of pipeline registers as in the actual implementation of our 4-core mixed-grain DT-Array. There will be two test cases for each fine-grain implementation of pipeline stages. One core configuration with lowest overhead in performance (Best case) and the other with highest overhead (worst case). Fig 4.3 shows these cases, where we can see that the performance overhead is caused by pipeline registers between the two cores. Hence the more we switch stages between the two cores the more bubble stages will be added in the pipeline.

Fig 4.3 Test cases for performance and power evaluation - One Best Case and one Worst-Case configuration for each Fine-grain implementation of pipeline stage

4.3.3 Procedure

Here we describe the procedure used to acquire performance and power results for test cases mentioned in previous section. We have already acquired ASIC netlist and FPGA post P&R simulation model for each fine-grain version of stages from the synthesis process described in sec 4.2. From synthesis process we also obtained the values of minimum clock periods possible along with the area reports. Initially a realistic target clock constraint is provided to the tool and by running the synthesis engine on the design it is determined if that constraint is achievable. If tool fails to achieve the required constraint we increases the clock period a little and thus through trial and error determines the maximum clock frequency of the design.

Now the synthesized RTL files are simulated in QuestaSim simulation suit with all four benchmark applications in order to record execution time and switching activity (VCD files) for each case. The simulations are performed at maximum clock frequency of each component. For fine-grain the maximum clock frequency is 200MHz which corresponds to 5ns clock period as discussed in chapter-3. It is assumed that the DT-core runs at its own maximum clock frequency in normal situation and when fine-grain block is used to replace any stage the whole core will switch to 200 MHz clock.

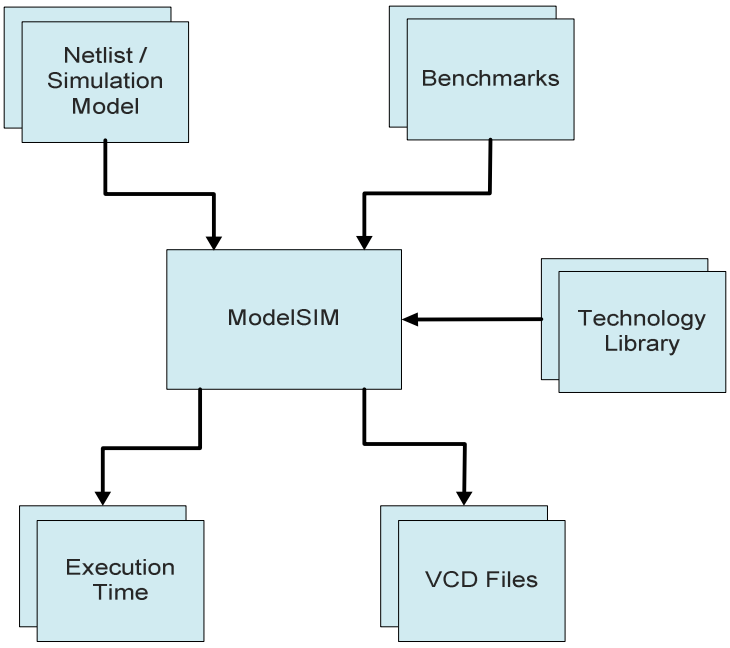


Fig 4.4 Flow chart for determining execution time and generating VCD files

For execution time measurements only the FPGA post P&R model is used along with the behavioral description of 4-core DT-Array, while for Power measurements both ASIC & FPGA versions are simulated in order to acquire two sets of VCD files for each benchmark. During simulation the VCD files are recorded for first 1000ns in order keep their size small while still capturing sufficient switching activity for power estimation. The power values obtained from these two versions of VCD files are later compared to compile a more realistic power estimate similar to the way it is done in area calculation. In order to determine execution time for each benchmark accurately the assembly versions of benchmark codes are slightly modified to indicate program termination by setting a bit in a register or memory location. These bits are then checked during the simulation by a *.do file script to record the execution time of each benchmark in a text file.

The process of obtaining power reports from these VCD files is entirely different for FPGA and ASIC versions. In case of FPGA Xilinx X-Power tools is used, which takes design's *.ncd ,*.pcf files and post P&R simulation model as input and provides static and dynamic power consumption in both text and graphical format. In ASIC version Cadence RTL compiler is used to generate power reports. For this purpose we provide synthesized netlist, clock constraints, technology library and a VCD file as input to the tool and it will generate a text report containing static and dynamic power consumption values for each component in the design. The process of generating these power reports can be automized with the help of script files, while the comparison of these reports is completely done manually in order to obtain final power results.

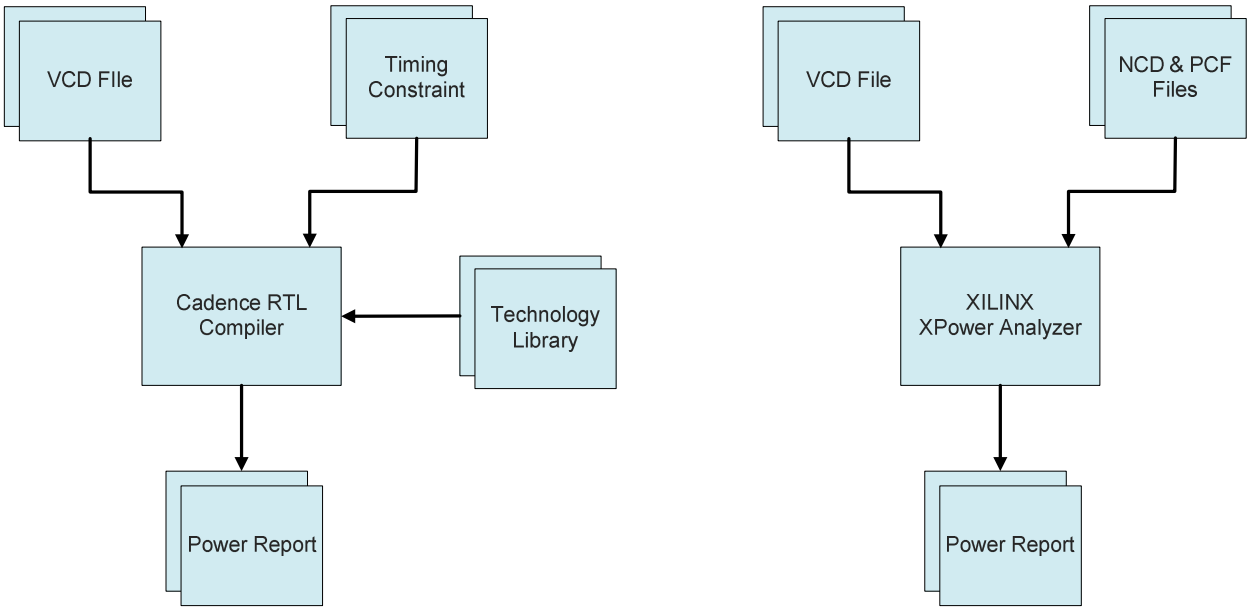


Fig 4.5 Flow charts for obtaining power reports of ASIC and FPGA pipeline stage versions

4.3.4 Performance & Power Results

In this section performance and power measurements for 4-core mixed grain DT-Array are presented. Fig 4.6 displays maximum achievable clock frequencies for fine-grain implementation of each pipeline stage. These values are compared with maximum operating frequencies of Baseline and DT-Core. We can see that Baseline core can operate at up to 700 MHz and DT-core at a slightly lower clock of 550 MHz, while in fine-grain the operating frequency varies between stages, however since the lowest clock frequency in fine-grain is the limiting factor therefore in actual implementation all stages in fine-grain are likely to operate at the lowest clock frequency. i.e. 200 MHz.

The performance of fine-grain block when running our benchmark applications is presented in fig 4.6 & 4.7. In fig 4.6 the number of clock cycles spend by each case of pipeline stages are shown along with DT-core and baseline for comparison, while in fig 4.7 the result is shown in terms of execution time. The values are normalized to baseline so that the overhead in each case can be seen clearly. We can see that due to higher operating frequency the Baseline and DT-core executes benchmarks faster even though they spend same number of clock cycles as most of the best case versions of fine-grain implementations.

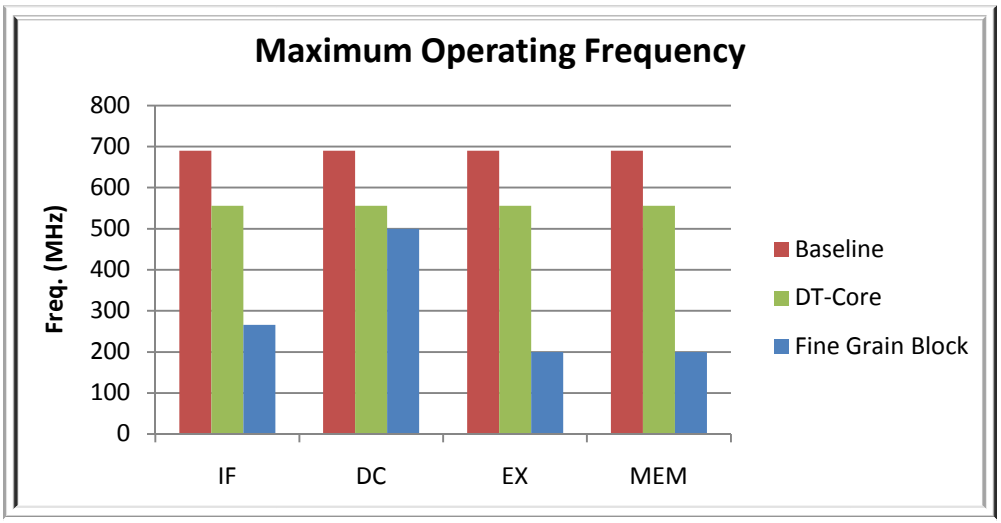


Fig 4.6 Comparison of maximum operating frequencies for Baseline, DT-Core and Fine-grain block with each pipeline stage

Further it is also apparent that the fine-grain version of EX-stage is the slowest configuration among others and takes about twice as much cycles in best case configuration and about 3 times in worst case configuration when running same benchmarks. The highest overhead of 3 cycles (10X in execution time) is received from worst case configuration of EX-stage when it is running high RAW conflict based benchmark application.

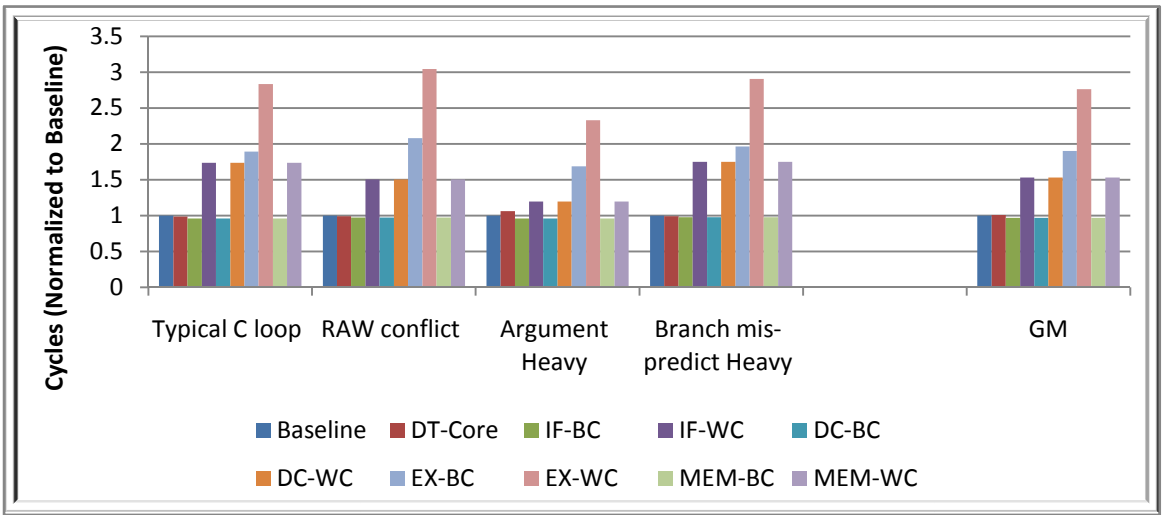


Fig 4.7 Clock cycles spend on running each type of benchmark by Baseline, DT-core and Best & Worst cases of all four fine-grain implementations of pipeline Stages - Normalized to Baseline

It is followed by Heavy branch miss-prediction benchmark with a slightly lower value. Since the fine-grain version of EX-stage is pipelined and we are dealing with all data hazards by flushing and reloading the instruction therefore in pipelined version of EX-stage both RAW hazards and branch miss-predictions are treated in a similar way which results in similar clock cycle penalty in both benchmarks.

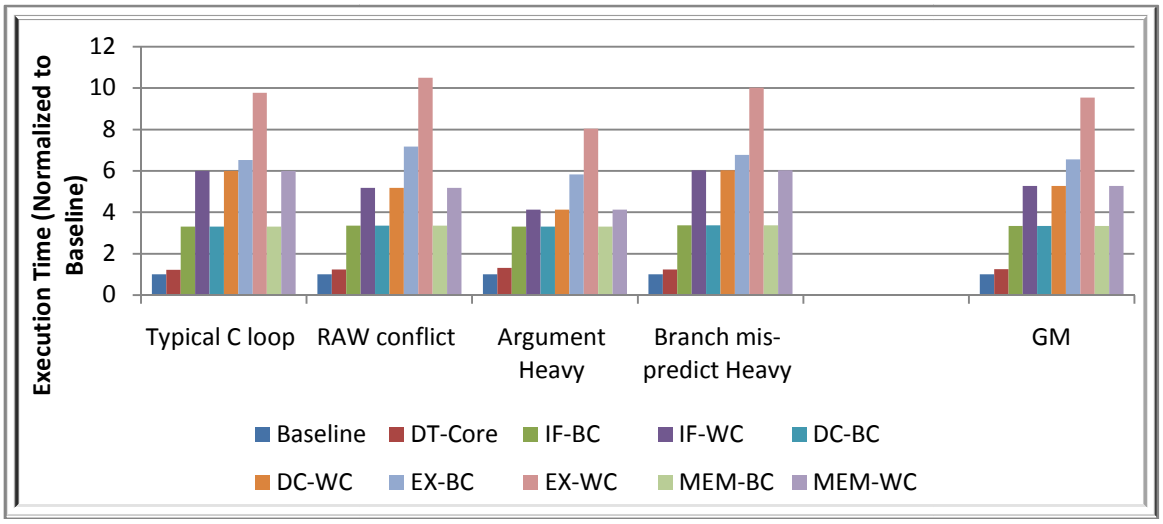


Fig 4.8 Execution time of each type of benchmark in Baseline, DT-core and Best & Worst cases of all four fine-grain implementations of pipeline Stages - Normalized to Baseline

It is also worth noting that the overhead in best and worst case configurations of fine-grain pipeline stages with the exception of EX-stage is nearly equal and only caused by lower operating frequency which is 3.5X slower than the Baseline clock. Moreover the charts shows that best and worst case configurations of each fine-grain stage exhibits lowest overhead difference when running Argument heavy benchmark code. The main reason for this is that the Argument heavy code causes data hazards due to memory read & write operations and these hazards are not covered by conflict table and feedback lines due to which both best and worst configurations stalls at similar rate.

Fig 4.9 shows the comparison in terms of IPS between different test cases when running our benchmarks. The Baseline and DT-core have quite high IPS values as expected due to their higher clock frequencies. The chart shows that on average IPS value of fine-grain stages is 5 X lower than the baseline mainly due to their lower clock frequency as before. Among the fine-grain version of stages the EX-stage has lowest values of IPS for all benchmarks.

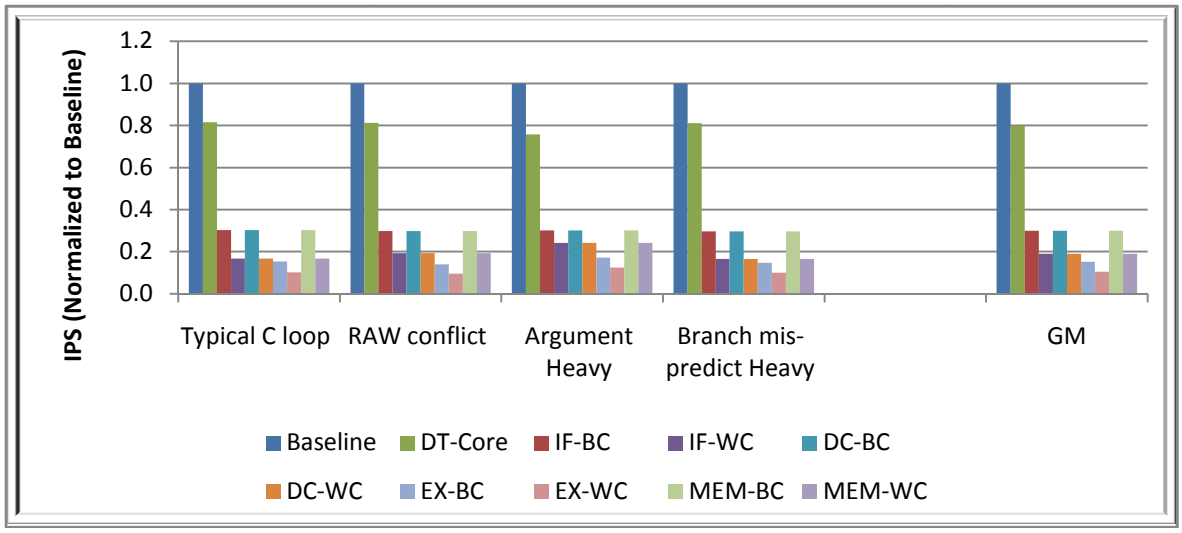


Fig 4.9 Comparison of Instructions Per Second (IPS) when running each type of benchmark on Baseline, DT-core and Best & Worst cases of all four fine-grain implementations of pipeline Stages - Normalized to Baseline

This is due to higher number of stalls (flush & reload) in EX-stage that are generated by our additional hazard detection logic. The instructions that doesn't cause pipeline to stall in other fine-grain stages would create data hazards in our pipelined EX-stage and hence overall it takes longer to process same number of instructions in a pipelined design.

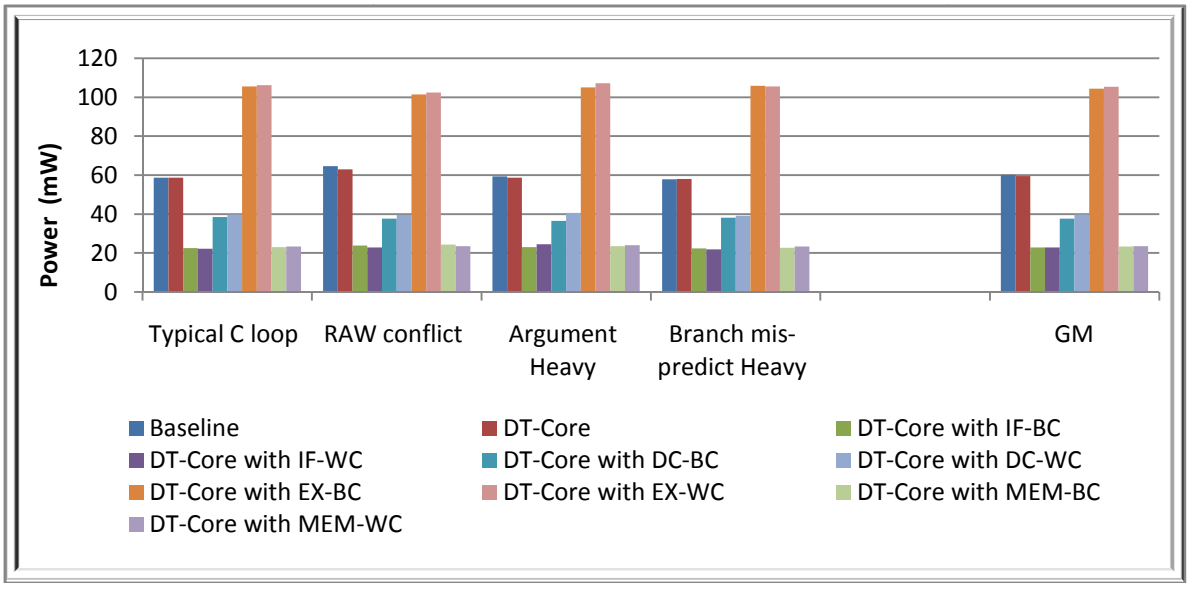


Fig 4.10 Comparison of power consumption when running each type of benchmark on Baseline, DT-core and Best & Worst cases of all four fine-grain implementations of pipeline Stages

Power consumption of reference designs (Baseline & DT-core) and fine-grain versions can be seen in fig 4.10 for all four benchmarks. With exception of EX-stage the fine-grain versions have lower power values as compare to baseline and DT-core mainly due to their lower operating frequency (200MHz) and secondly also due to their relatively smaller fine-grain area, as their logic is distributed between ASIC & FPGA sections in the SoC. While the EX-stage demonstrates highest power consumption, about 1.75X of baseline mainly due to its largest (2X) overall area utilization as compare to other three pipeline stages. Further it is evident from fig 4.10 & 4.11 that there is no considerable difference between the power consumptions of best & worst cases of the same stage due to similar switching rates in both configurations.

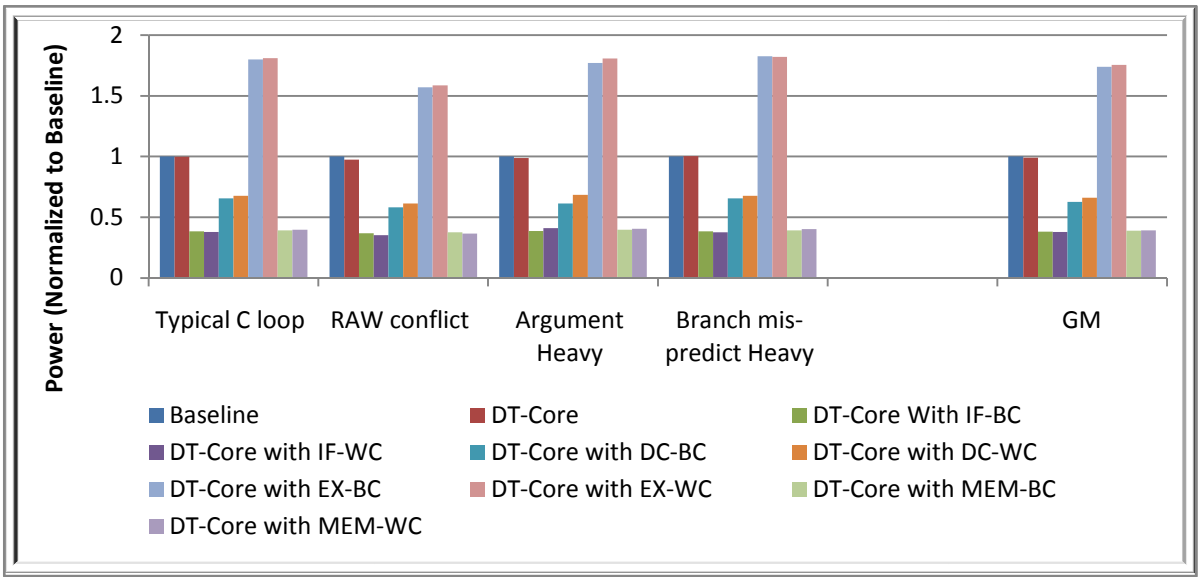


Fig 4.11 Power consumption normalized to Baseline

It can be observed in the power charts that worst case configurations of fine-grain stages have slightly higher power utilization than their best case counterparts; this is due to the addition of more registers in the pipeline which would have their own static as well as dynamic power values.

4.4 Dependability Analysis

After the performance and power measurements now we will evaluate the defect – tolerance of our 4-core mixed grain DT-array. Since the DT-array now also includes a wild card (Fine-Grain Block) along with 4 DT-cores therefore it is expected that the reliability of the previous design will improve. In this section we will first present the assumptions made for this dependability analysis, then the actual procedure will be described followed by the presentation of results.

4.4.1 Considerations

This dependability analysis includes measurements of availability in terms of average number of functional cores and the probability of having at least one functional core for a given defect rate. In order to compute these probabilities we would need a defect rate or defect probability for our design. Since the actual numbers are not available we will compute these probabilities for a range of defect rates starting from 0.0 to 2.8, similar to way it is done in previous work [2]. We assume that these are the defect rates for a Baseline core and further that in the DT-Core all stages have same area coverage and thus would have same defect rates. Here it is assumed that defect rate of '1' corresponds to a single defect in one core and '2' refers to two defects at a time in one core and so on. Further it is assumed that the array interconnect and fine-grain implementation of pipeline stages remains defect free at all time i.e. for all defect rates.

We are interested in the dependability comparison of core redundancy and the reconfigurable multicore array of both coarse and mixed grain types. To achieve this we have determined the configurations of Baseline cores, Coarse grain DT-Array and Mixed grain DT-Array that could occupy nearly equal chip area. With this approach we can demonstrate the possible advantage of having mixed grain reconfigurability over coarse grain reconfigurability and core redundancy in terms of defect tolerance.

Baseline	Coarse Grain DT-Array	Mixed Grain DT-Array
7 redundant cores	6 CG Cores in 4 cores & 2 Cores Clusters	4 Coarse Grain cores with 1 Fine-Grain Block

Table 4.1 Three cases for dependability comparison

Based on these assumptions we can drive the defect rates for 4-Core DT-Array. Since now we know the ratio between numbers of cores in these cases occupying same area, we can say:

$$7 \times Psd_{Baseline} = 6 \times K\,(Psd_{CG\_Stages})$$

Where,

Psd = Defect Rate.
K = number of stages in each CG core. i.e. 4 in our case.

From here the defect rate of each coarse grain stage is calculated to be:

$$Psd_{CG\_Stages} = (7/24)\,(Psd_{Baseline})$$

The defect rate of mixed grain DT-Array should be the same as coarse grain since in this case the fine-grain block is occupying the area of 2 coarse grain cores therefore the defect

probability still remains the same since we assumes that defects in fine-grain block would not cause failures. Therefore:

$$Psd_{MG\_stages} = Psd_{CG\_Stages}$$

We have also computed the relation between IPS (Instruction per second) of each test case and different defect rates. For this purpose the best case and worst case IPS values of each architecture (Baseline, 4-core CG Array and 4 Core CG with 1 Fine-Grain) are used as given in sec 4.3.4. It is assumed that minimum overall IPS is observed when in the coarse and mixed grain array only one of the core will be running at slowest IPS due to worst case configuration overhead while remaining cores will be working at best case IPS values. This assumption is quite realistic for coarse grain and mixed grain arrays as the worst case configurations are only implemented when higher number of pipeline stages in the array becomes defective. For maximum overall IPS it is assumed that all cores in the array will be running at their maximum IPS values.

4.4.2 Procedure

The method of calculating probability distribution and availability for baseline and coarse grain array is already available to us from the previous work [2]. We used MATLAB to perform calculations and plotting results. For Baseline the probability of having at least a specific number of functional cores is calculated as:

$$P(N,M) = \sum_{M \geq i \leq N} \frac{N!}{i! * (N-i)!} * Psd^{N-i} * (1-Psd)^{i} \qquad (4.1)$$

Where,

N = Total Number of cores
M = Minimum number of functional cores
Psd = Defect rate

By using equation 4.1 we calculate probabilities of having minimum one to maximum N number of cores for all values of defect rates. The probability for having at least one working core at different defect rates is also called reliability and it will be plotted separately for comparison. For Coarse grain array the probability calculation will be slightly different due to large number of combinations resulting from interchangeable pipeline stages. We again use equation 4.1 but with 'Psd' values of coarse grain array this time:

$$P(N,M)_{Coarse\ Grain} = P(N,M)^{K} \qquad (4.2)$$

Where,

N = Total number of cores

M = Minimum number of functional cores
K = Number of stages in each core. i.e. 4 in our case.

Now for mixed grain DT-array we have to consider those case as well in which fine-grain can replace any stage and repair the coarse grain array. Hence effectively increases the overall probability of having at least particular number of cores. Further we also have take in to account the limitations of our fine-grain block when it repair stages, such as the IF & MEM stages can only be repaired / replaced in two out of four DT-cores. This is achieved by generating all possible stage combinations in a 4-Core DT-array and then using a verifying scheme to check if a particular combination can form at least N working cores. i.e. Have N stages of each type. Where N varies from 1 to 4 in our case.

When a particular combination cannot form a working core we repair one of the defective stages in the combination to depict that it is replaced by our fine-grain block. Then the same combination is once again verified to see if a working core can be formed now. During this process all working combinations are stored in a text file and later we calculate the probability of all these combinations which will be our final result. Although this method is quite computational extensive but the advantage we get from it is that we can include desired conditions in the verifying scheme and therefore can obtain probability of working cores at a greater accuracy and according to the actual implementation of our design. In the end the average number of functional cores for each defect rate is computed by summing all the probabilities. i.e. probability of having at least 1 to 4 working cores:

$$Avg_{Workgin\ Cores} = \sum_{1 \geq i \leq N} P(N, i) \qquad (4.3)$$

Where,

N = Total number of Cores

The IPS values in IPS vs. Defect rate analysis are computed as follows for coarse & mixed grain arrays at each defect rate:

$$IPS_{\min} = \left(Avg_{Working\_Cores} - 1\right) * IPS_{Best\_Case} + IPS_{Worst\_Case} \qquad (4.4)$$

$$IPS_{\max} = Avg_{Working\_Cores} * IPS_{Best\_Case} \qquad (4.5)$$

$$IPS_{\text{Avg}} = \frac{\left(IPS_{\min} + IPS_{\max} \right)}{2} \qquad (4.6)$$

4.4.3 Dependability Results

Now we will look at the dependability results of our mixed grain DT-array. Fig 4.12 shows average number of available cores for each test case we mentioned in table 4.1. We can see that baseline version which is exploiting core-level redundancy fails very rapidly and for defect rates greater than one there will be no working core available. In coarse grain case there will be at least one working core available up to 2X defect rate while the mixed grain version even though initially have lower number of total cores, will out-perform the coarse grain array in terms of availability at defect rates higher than 1.3X. Hence at 2X defect rate the mixed grain version can still provide one working core while the coarse grain array would have no working core available.

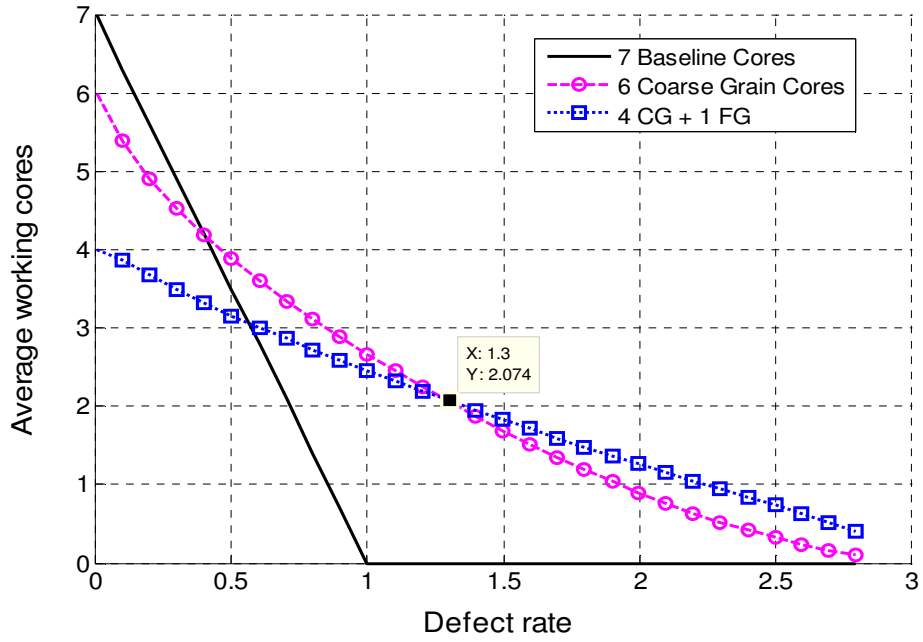


Fig 4.12 Number of average working cores at different defect rates

The reliability plots for the three test cases are presented in fig 4.13. It is apparent that the mixed grain version of DT-array offers much better reliability especially at higher defect rates. We can see that at defect rate of 2.5X the reliability of coarse grain version drops to 30% while the mixed grain array still holds a reliability value close to 70%.

In this dependability analysis another interesting comparison can be made for amount of instructions that can be executed per second in each test case at different defect rate values. This measurement also incorporates performance of each test case in the dependability analysis.

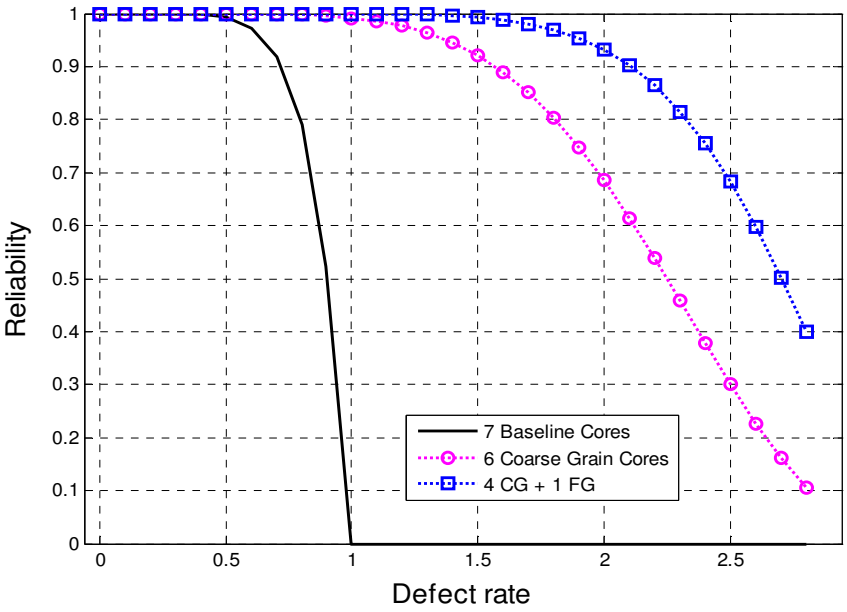


Fig 4.13 Reliability comparison in three test cases

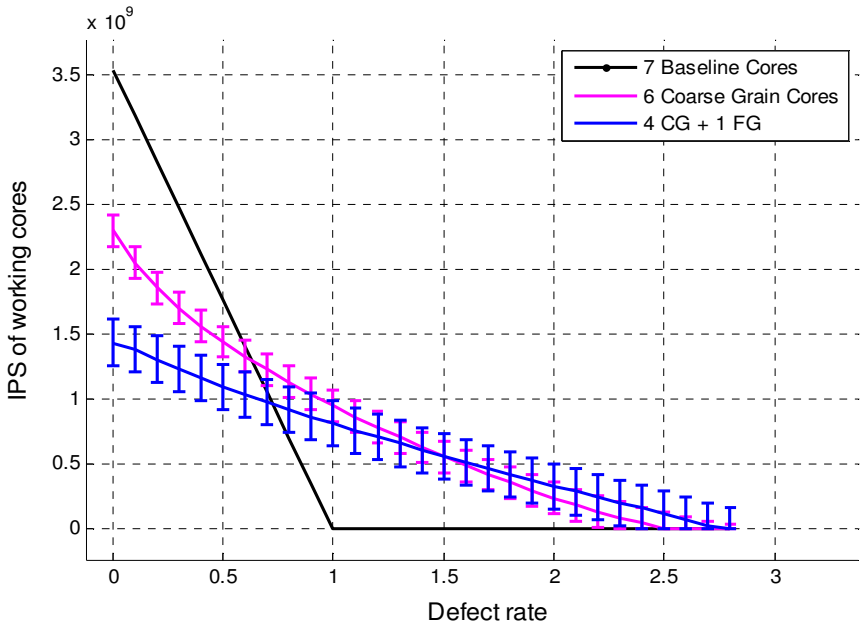


Fig 4.14 Comparison of Instructions per second (IPS) vs. defect rate for three test cases – indicating max & min IPS with error bars

We can see in fig 4.14 the IPS of mixed grain array is about 40% lower than coarse grain version at very low defect rates. While the difference gradually decreases as the defect rate increases and at defect rates above 1.5X the IPS of Mixed grain becomes higher than the coarse grain array.

Another point to note in this plot is that the difference in minimum and maximum IPS values in mixed grain version is higher due to slower clock rate of fine-grain block while in coarse grain version this difference is low. This also indicate lower performance overhead of worst case configuration in coarse grain array as compare to the mixed grain version.

4.5 Summary

In this chapter we have seen the evaluation procedure used in this work and the results of this evaluation. In the beginning we mentioned the source of design RTL files and the corresponding C-compiler. Then we talked about synthesis process and the method we used to determine area utilization in FPGA and ASIC sections of our design. The benchmarks and test cases adapted for performance and power measurements were discussed along with the description of main procedure and tools. Then we have seen the results of performance and power analysis for different core configurations. At the end the procedure used to measure availability and reliability of our design was presented and these values were compared with two other scenarios, one using core-level redundancy and other coarse grain reconfiguration. It was shown that the mixed grain version can out-perform coarse grain version at higher defect rates. In the next chapter we will present our conclusions and the outcomes of this thesis along with future recommendations and suggestions.

5 Conclusions & Future Work

In this thesis work we took coarse grain implementation of defect-tolerant array and tried to include a fine-grain wild-card like block which could replace any one of the defective pipeline stages in the array when required. Thus would improve the availability of the multiprocessor array at higher defect rates. During this endeavor we identified many design challenges and proposed solutions for them. We presented the implementation and evaluation process adapted and then discussed the results of this evaluation at the end. In this chapter we will present the conclusions we drawn from this work and the goals that we achieved along with some suggestions for future work in this direction.

5.1 Conclusions

We have presented the implementation and evaluation details of a fine-grain wild-card block that can instantiate and replace any one of the pipeline stages in the coarse grain defect-tolerant multiprocessor array. From the evaluation results of this design we can say that the major drawback of this approach is the area overhead of fine-grain implementation which is about 58% (2.4X) of the complete defect tolerant core. Although some area optimization techniques have been employed such as reduction in logic size and distributing it between FPGA and ASIC portions, however the ratio is still inadequate to provide reasonable advantage at lower defect rates as compare to coarse grain only approach. The large area of fine-grain is caused by execution stage which includes highest logic density (5X) as compare to other stages. Further the fine-grain implementation also includes other supporting logic such as conflict table which is not part of execution stage in the coarse grain version.

In terms of performance the main limiting factor came from lower operating frequencies of pipeline stages in fine-grain block. The stages in fine-grain can work at up to 200 MHz while the baseline and defect-tolerant core can attain clock frequencies of up to 556 MHz and 690MHz respectively. Hence it is already about 64 to 70% slower without taking in to account the pipeline overhead of EX-stage. In worst case configuration the performance overhead can reach up to 10X of the baseline in terms of execution time with the inclusion of pipeline latencies and re-fetch delays caused by data hazards. The same trend is observed in instruction per second values however in terms of clock cycles the overhead would be just 3X of the baseline in a worst case scenario. This is about 25% lower than the overhead of worst case configuration in a 4-core coarse grain DT-array. Therefore we can conclude that at lower clock frequencies the performance of mixed-grain DT-array will be similar to the coarse grain only version and hence would be much viable approach to use due to its higher defect tolerance capabilities. Further by using a better compiler such as the one available in latest version of Processor Designer suit the performance of fine-grain block can

be improved considerably as the main cause of stalls in this case is due to the inefficient use of available registers in the assembly code.

The major portion of power consumption in fine-grain block comes from static power of EX-stage due to its larger logic area. Therefore the power overhead is also about 1.8X as compare to baseline in a worst case scenario. While in the case of other three stages the power consumption is just 40 to 60% of the baseline core. This lower power is a result of lower clock frequencies the concerned core is running on when utilizing fine-grain block. Overall due to the inclusion of more pipeline registers and hazard detection logic in the fine-grain version of EX-stage the power consumption of a core using a fine-grain block would be in worst case 60% higher than a coarse grain DT-core when run at same clock frequencies.

In terms of dependability due to much greater fine-grain area the mixed grain array only exhibit advantage over coarse grain version at very high defect rates. In our dependability analysis we divided the coarse grain array in to clusters of 4 & 2 cores and thus avoid those array configurations that are not supported by the hardware implementation. This results in a more realistic comparison between two variants. From the results it can be concluded that core redundancy achieved by clustering of baseline cores is in most cases sufficient to provide defect tolerance at lower defect rates. The coarse grain version on the other hand offers much better defect tolerance at a higher performance cost and can only be outperformed by an equivalent mixed grain version when defect rate would be as high as one fault per core at the least.

5.2 Thesis Contributions

In this thesis work we have successfully:

- Designed and implemented a fine-grain reconfigurable block that works as a wild-card in the coarse grain DT-array and provides replacement for defective pipeline stages.

- Implemented performance improvement measures in the fine-grain versions of pipeline stages and reduced the performance gap between ASIC and reconfigurable sections with the help of pipelining and logic optimization.

- Developed a C-program to generate reconfiguration bits for mixed grain DT-array and determines which stage the fine-grain block instantiates when in use in order to dynamically produce maximum number of working cores in a defective array.

- Evaluated and measured the performance , area and power overhead of the fine-grain block in the mixed grain DT-array and determined availability improvements the 4-core mixed grain DT-array provides as compare to coarse grain array of the same area utilization.

5.3 Future Work

In continuation of the work presented in this thesis, we like to suggest following improvements that can be explored in future:

- Data hazard handling can be improved by incorporating a local stall mechanism in pipelined EX-stage. This can be achieved by using a FIFO buffer at the input of EX-stage which can act as a temporary delay slot when a RAW hazard is detected and would be bypassed in normal pipeline execution. Such approach can reduce the number of flush & reloads in a typical program by up to 50%.

- The EX-stage can be split in to 2 to 3 smaller stages in order to reduce the fine-grain area. With a lower fine-grain area more coarse grain cores can be included in the array which would ultimately improves the overall availability of the mixed-grain DT-Array. In order to achieve this we would have to decide how the logic will be distributed in different sub sections and how it will impact the performance and interconnect area of the array.

Bibliography

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation”," *Micro, IEEE 25*, pp. no. 6, 10 - 16., 2005.

[2] G.Smaragdos, "An Adaptive Defect-Tolerant Multiprocessor Array Architecture," Master’s Thesis, Computer Engineering Department , Faculty Of Electrical Engineering, Mathematics And Computer Science Delft University Of Technology, Netherlands, 2012.

[3] Ioannis Sourdis, et al, "The DeSyRe Project: On-Demand System Reliability," in *Digital System Design (DSD),2012 15th Euromicro Conference on. IEEE*, 2012.

[4] Scott Hauck, Andre' dehon (editors), *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.*: Morgan Kaufmann/Elsevier, 2008, pp. 831-832.

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

[6] LaFrieda, Christopher, et al, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on. IEEE* , 2007.

[7] Powell, Michael D., et al "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ACM SIGARCH Computer Architecture News. Vol. 37. No. 3. ACM*, 2009.

[8] Sylvester, Dennis, David Blaauw, and Eric Karl, "Elastic: An adaptive self-healing architecture for unpredictable silicon," in *Design & Test of Computers, IEEE 23.6 (2006)*, pp. 484-490.

[9] Wagner, Ilya, Valeria Bertacco and Todd Austin, "Shielding against design flaws with Field repairable control logic," in *Proceedings of the 43rd annual Design Automation Conference. ACM*, 2006.

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

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

[12] Emmert, John, et al, "Dynamic fault tolerance in FPGAs via partial reconfiguration," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on. IEEE*, 2000

[13] Davide Quarta, Cristiana, and Marco D. Santambrogio Bolchini, "SEU mitigation for RAM-based FPGAs through dynamic partial reconfiguration.," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI. ACM*, 2007.

[14] Gupta, Shantanu, et al. "Stagenet: A reconfigurable fabric for constructing dependable cmps," in *Computers, IEEE Transactions on 60.1 (2011): 5-19*.

[15] Romanescu, Bogdan F., and Daniel J. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults.," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM*, 2008.

[16] Pellegrini, Andrea, Joseph L. Greathouse, and Valeria Bertacco, "Viper: virtual pipelines for enhanced reliability.," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on. IEEE*, 2012.

[17] Pellegrini, Andrea, Joseph L. Greathouse, and Valeria Bertacco, " Cobra: A Comprehensive Bundle Reliable Architecture." 2013 [Online]. Available: http://web.eecs.umich.edu/~valeria/research/publications/SAMOS13Cobra.pdf. [Access -ed Sep. 15, 2013].

[18] R., and V. Bagyaveereswaran Anitha, "High Performance Parallel Prefix Adders With Fast Carry Chain Logic," *International Journal*, 2012.

[19] Khan Shoaib Ahmed, *Digital Design of Signal Processing Systems*.: Wiley, 2011, pp. 302-305.

[20] G. Shiva Sajjan, *Advance Computer Architecture*.: CRC Press, 2006, pp. 73-105.

[21] Vasileios Vasilikos, "Heuristic Search For Defect Tolerant Adaptive Multiprocessor Arrays," Master's Thesis, Computer Engineering Department , Faculty Of Electrical Engineering, Mathematics And Computer Science Delft University Of Technology, Netherlands, 2011.

[22] Vasileios, et al Vasilikos, "Heuristic search for adaptive, defect-tolerant multiprocessor arrays," in *ACM Transactions on Embedded Computing Systems (TECS) 12.1s (2013): 44.*

[23] Henry, Vaughn Betz, and Jonathan Rose Wong, "Comparing FPGA vs. custom cmos and the impact on processor microarchitecture," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays.* ACM, 2011.