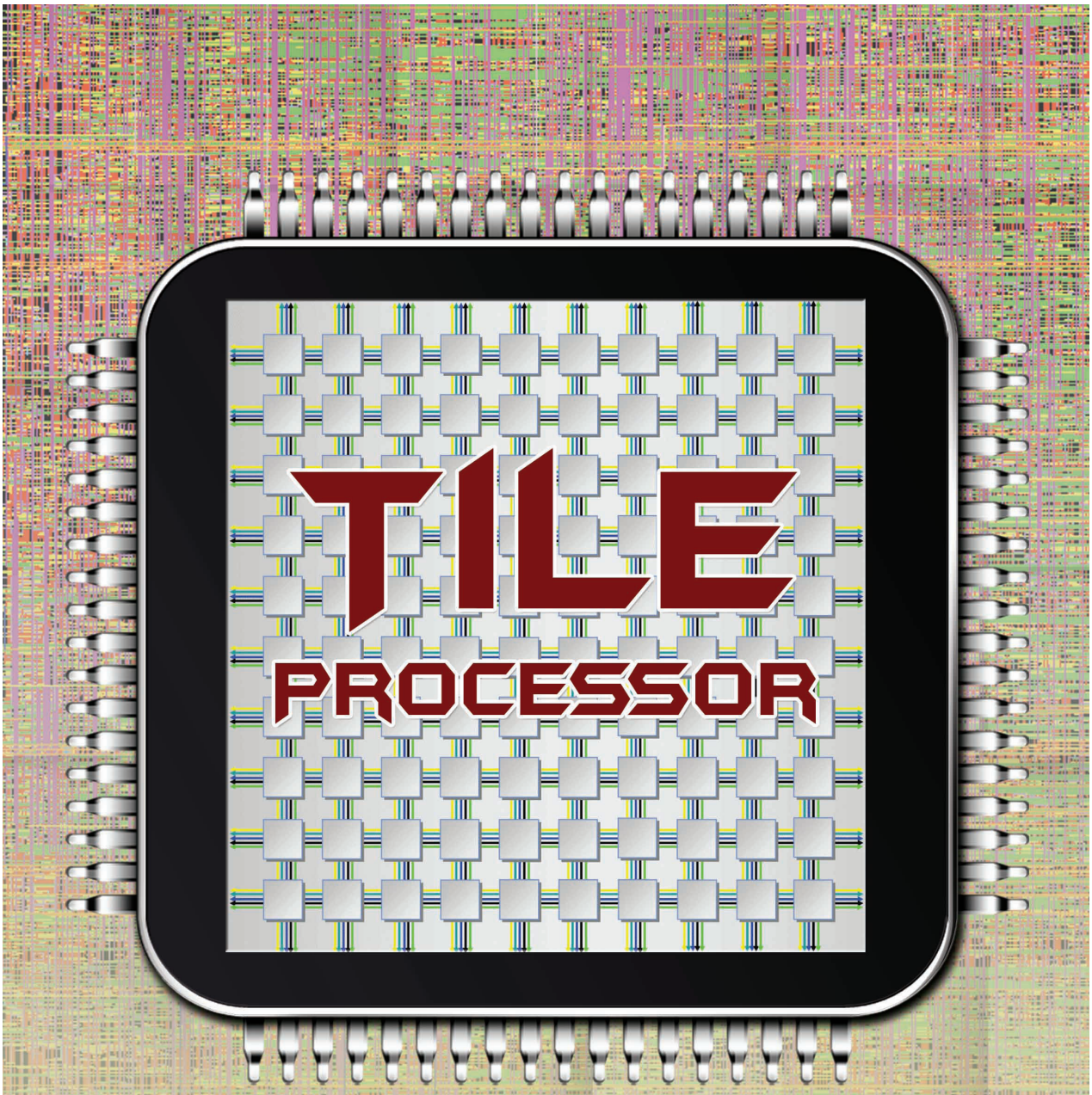


NISC 2D PROCESSOR ARRAYS FOR STATICALLY SCHEDULED PROBLEMS IN DSP APPLICATIONS

ALEXANDRIA UNIVERSITY

Egypt Innovate
Technology Innovation and Entrepreneurship Center (TIEC)

VLSI Egypt

NISC 2D Processor Arrays for Statically Scheduled Problems in DSP Applications

BSc. Graduation Project
June 2013

Dr. Yasser Hanafy, AAST University

Prof. Dr. Mohamed Rizk, Alexandria University

AlexandriaUniversity, Faculty of Engineering, Communication and Electronics Department

This page is intentionally left blank.

Table of Contents

Abstract 1
Acknowlegement 2
1. Introduction 5
2. Background 7
3. Related Work 15
4. Proposed Design and Implementation Details 19
5. Realization 82
6. Verification 87
7. Conclusion and Future Work 89

Abstract

With the recent advances of Technology, the number of transistors on a single chip has increased dramatically, but we have almost reached an asymptotic value of operating frequency. Performance increase is now moving horizontally where we have to explore parallelism and architectural innovations. Processor arrays started to develop significantly, as a general purpose devices. General purpose processors usually compromise performance to accommodate the largest number of applications. On the other end of the spectrum, special purpose hardware is tuned for a target application. Outside the scope of the target application, special purpose hardware performs poorly or does not even perform at all. Reconfigurable architecture, such as FPGA's have shown versatility in the last decade, but it's performance still depends on the application.

In this project we are focused on the design and implementation of reconfigurable processors arrays, to fill the gap between general purpose processors (Multicores), special purpose hardware (GPU), and general purpose reconfigurable architectures (FPGA). Our architecture serves a class of problems which are known as statically scheduled problems. Examples are matrix operations, DSP applications … etc.

The objective of the project is to design and implement a simple processor array targeting statically scheduled problems. The selection of this a class of problems simplifies the design of the processor significantly, yet, it preserves a large spectrum of applications in high performance computing. The target processor array is divided into two major components; the processing element, and the interconnection network. The implementation achieved an operating frequency greater than 1 GHz, which we believe can be increased by further optimizations. A complete implementation of the chip is carried out using Synopsys EDA tool chain.

In this book we present the design and VLSI implementation of a single tile (processing element) of the 2D array architecture discussed.

Acknowledgement

The success and final outcome of this project was not to be accomplished if not for the help of guidance of many people along the past year. All what we have done was only due to such guidance and assistance.

We would like to thank our dear advisor Dr. *Yasser Hanafy*, AAST University, for giving us the opportunity to work on such a project with him. We would like to express our appreciation and gratitude for his support, guidance and following up with the project in every step. His advice was valuable both in the project and in real life.

We respect and thank Prof. Dr. *Mohamed Rizk*, Alexandria University, for helping us and allowing us to execute our project in **Virginia Tech - MENA** labs.

We owe our profound gratitude to the Technology, Innovation and Entrepreneurship Center, **TIEC**, represented by Dr. *Maged Ghoniema*, for sponsoring and supporting this project, whether through the VLSI Academy Training for team members during the summer of 2012, or through providing us in VT-MENA Labs with the full digital flow Synopsys tool chain so we can execute our project.

We would not forget to remember Eng. *Mohamed Wasfy*, AAST University, for providing us with the information we needed on the architectural level of the system. We would like to thank him for helping us during the implementation of the Network-on-Chip Router, as well.

Finally, we are grateful for all the encouragement and support we got whether from our teaching staff or our colleagues.

Team Members

Ahmad Bassam Sayed Ayoub
01278523499
bassam.ayoub@ieee.org

Amany Eweida Ahmed Eweida
01223475121
amany.eweida@gmail.com

Lokas Fikry Naguib Asaad
01273774683
lokasfikry@hotmail.com

Marwan Mohamed Adel Abd El-Azzim
01002255446
marwan.m.adel@gmail.com

Marawan Moustafa Moustafa Mohamed Ahmed
01009502087
marawanmmma@gmail.com

Mohamed Osama Abdelaziz Abdellatif Elmoghany
01092705041
elmoghany@ieee.org

Mustafa Mahmoud Mohamed Khairallah
01224806242
khairallah@ieee.org

Noha Ahmed Mohamed El Shaarawy
01062878176
noha_sharawy@ymail.com

Yasmine Ashraf Mohamed Abgad
01227161080
yasmineashraf91@gmail.com

This page is intentionally left blank.

1. Introduction

Statically scheduled problems represent a large spectrum of scientific computing problems where the execution steps of the algorithms are well defined at the run time [**1**]. Most of these problems require high performance floating point computations. Among these problems, solution of large linear systems is the most dominant. It includes performing floating point computations on massive amounts of data. Linear systems solution algorithms include various matrix operations, such as: LU decomposition, Gauss-Jordan Elimination, Matrix Multiplication, Matrix Inversion and other matrix operations.

Such problems appear in a wide range of scientific problems, from spice circuit simulations to the simulation of blood circulation inside human brain. Some research institutes and researchers spend a lot of money on buying super-computers to perform such complex simulations, due to the performance shortcomings of the other approaches that are discussed below. In the past, general purpose processors, with the aid of floating point co-processors, were used to solve such systems. However, they introduced a lot of limitations and inefficiencies that affected the overall performance badly [**2**].

To overcome the drawbacks of using general purpose processors, another approach appeared, which is to use massively parallel GPUs for such algorithms, such as in [**3**][**4**]. The performance on GPUs was better than the performance on general purpose processors, as GPUs were designed to do operations that are similar to linear systems solution. However, the performance achieved was much lower than the peak performance that the hardware could achieve. In [**4**] the performance achieved was around 12 GFLOPs, while the NVidia 6800 GPU has a peak performance of 48 GFLOPs.

Another approach is to use FPGAs as hardware accelerators as in [**5**]. The author instantiated six NIOS soft IP processors to operate in parallel. The two drawbacks of this designs were that NIOS used a single precision floating point unit, which is not accurate enough and that the architecture was based on shared memory, which will be a bottleneck for the performance at large matrix sizes.

In [**1**], the author introduced a new architecture of a 2D array of reconfigurable NISC processing elements connected using a reconfigurable network on chip, NoC. The architecture was simulated

on the LU decomposition algorithm using 64 processing elements and resulted in 177 GFLOPs. The author also made suggestions for an optimized data path. We have been working with some of the authors inspired by this design in [**1**], to develop a VLSI implementation for the previously mentioned architecture in 65 nm CMOS technology. Such architecture was initially inspired by the mesh network, distributed memory based architecture presented by the authors of [**6**], which was later adopted commercially by Tilera.

Our design includes the design of a high speed floating point unit, a scalable NoC and a micro programmed NISC control unit.

1.1 Motivation and Problem Statement

A need is rising for a high performance reconfigurable computing platform to perform heavy scientific computations. It has to be characterized by high accuracy and low cost in order to be able to provide a reasonable replacement for super-computers in some applications. This need arises from the shortcomings present in the other platforms currently used, such as GPUs and FPGAs, which were discussed in the previous section, and the high cost of super-computers.

1.2 Research Goals

The objective set out for this project is implement the architecture in [**1**] in the 65 nm CMOS technology. Some of the minor goals achieved in the project are:

• Design a high speed double precision floating point unit with a fused-multiply-add unit and a high speed floating point divider.

• Design a scalable NoC router that can be used to interface any number of cores smoothly without producing performance issues when the number of cores increases.

• Design a micro programmed NISC control unit that achieves a high level of hardware parallelism.

1.3 Book Overview

Chapter 2 introduces the reader to the basic principles of multi-core architectures, NISC architectures, microprogramming, network-based architectures and floating point number representations. It alsopresents the architecture upon which our VLSI implementation was based.

Chapter 3 presents a short overview of the work in literature related to our project and available computing platforms.

Chapter 4 discusses the implementation details of our processing element on both the top and block level. It also includes the synthesis and placement and routing results.

Chapter 5 discusses the realization of our design on Synopsys tool chain.

Chapter 6 discusses the tests performed on the design.

Chapter 7 concludes the book discusses some of the drawbacks and bottlenecks of our design and how we plan to solve them.

2. Background

2.1 Multi-core Architectures

Since the earliest processors came onto the market, the semiconductor industry has depended on Moore's Law[1] to deliver consistent application performance gains through the multiplicative effects of increased transistor counts and higher clock frequencies. However, power dissipation and design complexity issues have emerged as dominant design constraints that severely restrict the ability to increase clock frequency or utilize more complex out-of-order mechanisms to improve performance. Exponential growth in transistor counts still remains intact and a powerful tool to improve performance. This trend has led major microprocessor companies to put multiple processors onto a single chip [**8**].

As an application of the previous observation and as personal computers have become more prevalent and more applications have been designed for them, the end-user has seen the need for a faster, more capable system to keep up. Speedup has been achieved by increasing clock speeds and, more recently, adding multiple processing cores to the same chip. Although chip speed has increased exponentially over the years, that time is ending and manufacturers have shifted toward multi-core processing. However, by increasing the number of cores on a single chip challenges arise with memory and cache coherence as well as communication between the cores. Coherence protocols and interconnection networks have resolved some issues, but until programmers learn to write parallel applications, the full benefit and efficiency of multi-core processors will not be attained [**9**]. However, multi-core systems have managed to become the dominant mainstream computing platform, despite the fact that one of the biggest challenges going forward is how to efficiently utilize the ever increasing computational power provided by multi-core systems. Multi-core systems increase throughput and efficiency by utilizing multiple cores to perform computation in parallel and complete a larger volume of work in a shorter period of time.

1 In 1965, Intel co-founder Gordon E. Moore predicted that the number of transistor that can be fit in the same area of the integrated circuit will double every 18 months [**7**]

2.1.1 Background

As transistor density increases, power dissipation and on-chip wire latency become two major challenges for processor designers. Consequently, increasing single core performance becomes costly in terms of power and may cause bad effects on the extra transistors used. Another way to utilize the extra transistors is to place multiple cores on the same die to provide more computing power and throughput on a single chip. **Figure 2.1** shows a typical implementation of multi-processor chips available on the market. Two cores are packaged in a chip in this example. Each core contains private level 1 instruction and data cache, execution pipeline and maintains separate architectural states such as register files. The level 2 cache and main memory are shared between the two cores. The two cores access a shared memory address space. Multiple threads communicate through memory, the producer of data writes to the memory and the consumer load from the same memory address. The communication latency between cores on this chip multiprocessor is much lower than that on traditional multiprocessor systems because the cores are on the same die and they share a level 2 cache. For example, in the Intel Core 2 Duo processor, the communication latency between cores can be as low as 14 core clock cycles depending on the location of the data in the cache hierarchy. In the architecture shown in **Figure 2.1**, two cores can execute two tasks or threads at any given time. This architecture provides benefit for users that run multiple simultaneous tasks or explicitly threaded applications.

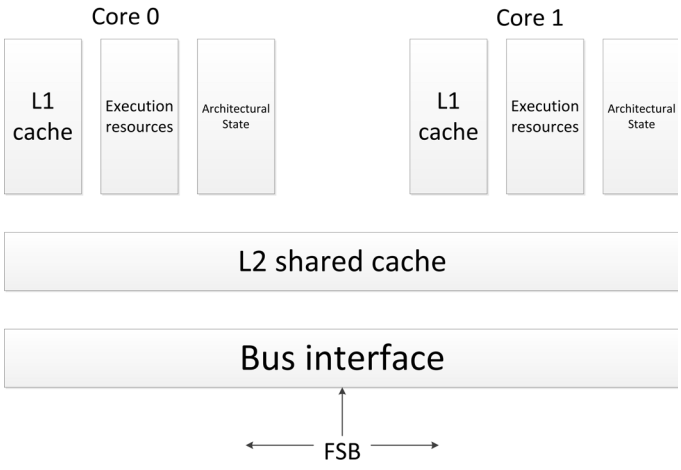


Figure 2.1: Multicore processor architecture

2.1.2 Shared Memory Model vs. Distributed Memory Model

Two different approaches to implement multi-core architectures are shown in **Figure 2.2.** If we set two cores side-by-side, one can see that a method of communication between the cores, and to main memory, is necessary. This is usually accomplished either using a single communication bus or an interconnection network. The bus approach is used with a shared memory model, whereas the inter-connection network approach is used with a distributed memory model. After approximately 32 cores the bus is overloaded with the amount of processing, communication, and competition, which leads to diminished performance; therefore, a communication bus has a limited scalability. On the other hand, using inter-connection network allows more cores to be placed on the same die, it is highly scalable [**9**].

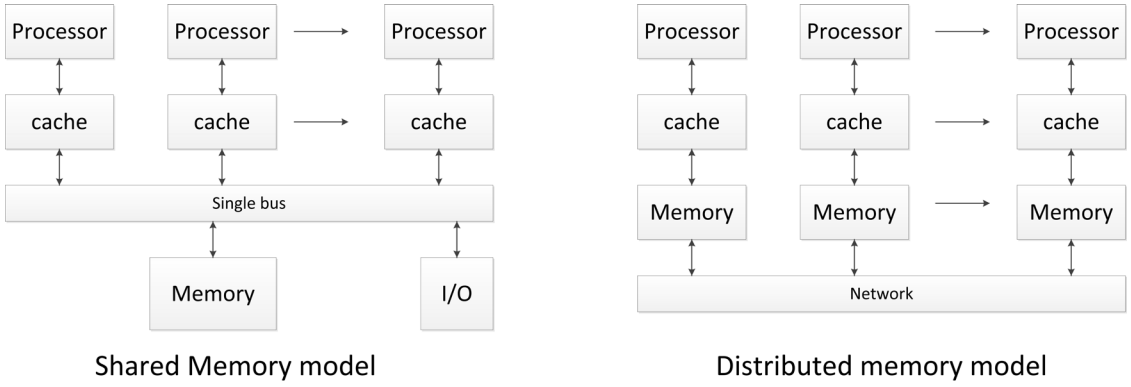


Figure 2.2: Shared Memory Model vs. Distributed Memory Model

2.2 Microprogrammed Control Unit

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of micro-operations. A hardwired control requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

2.2.1 Microprogramming

Microprogramming is an alternative to what is referred to as hardwired approach for designing the control part of a system. The key benefit of microprogramming is its flexibility in making design changes without having to redesign or rewire the system. This makes it extremely useful during the design and development phase as frequent modifications are inevitable at this stage.

The control part and the data path are the two major components of any system. These parts interact via two sets of signals; the status signals from the data to the control and the control signals from the control to the data part. The data part contains the resources for storing and performing operations on data values whereas the control part sequences these operations/data transfers in a specific order to implement the algorithm. The control part controls the sequence of operations by issuing the signals in a specific order.

The control words are stored in the control memory. The microinstruction contains a control word that specifies one or more micro-operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the micro-operations are being executed, the next address is computed in the next address generator circuit to read the next microinstruction. Thus a microinstruction contains bits for initiating micro-operations in the data processor part and bits that determine the address sequence for the control memory.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs.

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is some-times called a pipeline register. It allows the execu-

tion of the micro-operations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register. The control word and next-address information are taken directly from the control memory. It must be realized that the control memory operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in the memory remains in the output wires as long as its address value remains in the address register. Each clock pulse will execute the micro-operations specified by the control word and also transfer a new address to the control address register. [**10**]

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

But it is worth mentioning that the hardwired control unit is much faster and more optimized than the microprogrammed control unit.

2.3 No Instruction Set Computer (NISC)

In the recent years, Increasing Productivity and Reducing time to introduce the product in the market are the most two targets needed to be reached by the designers. Consequently, they are searching for a new alternative approach that can handle them.

One of the probable solutions that come to mind is increasing levels of abstraction so as not to waste times in the small details that have been covered before. In fact the progression of the processors architecture has passed through 3 historical stages:

1. During 1970s, CISC (Complex Instruction Set Computer) has been introduced where single instructions execute several low level operations (such as a load from memory, an arithmetic operation, and a memory store) where, the designers try to improve the performance by using complex instructions stored in a very sluggish Program memory (PM). Each complex instruction takes several

clock cycles to be executed. The datapath control words of each clock cycle is stored in micro program memory (mPM) which is faster than the original PM as shown in **Figure 2.3**. Here, compilers have not to be very complex as only few small instructions are used.

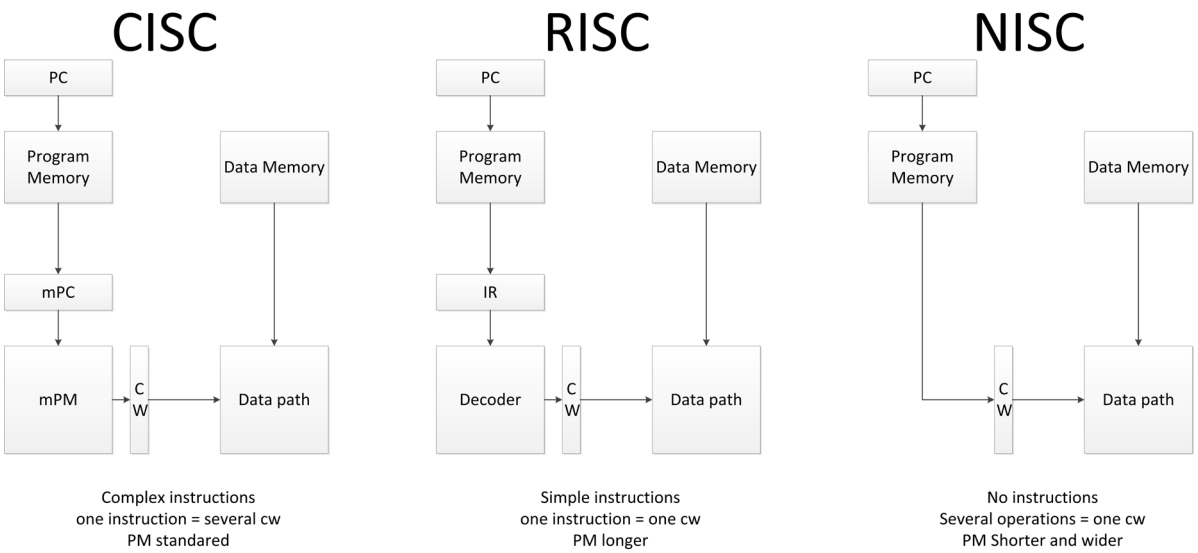


Figure 2.3: Layout of CISC, RISC and NISC [**NISC** 2]

2. During late 1980s, RISC (Reduced Instruction Set Architecture) becomes popular. RISC introduces the idea of breaking down the complex instruction into reduced ones where all the instructions are simple and executed in one clock cycle. Here, the mPM was replaced with decoding stage that followed the instruction fetch from PM. Given that instructions are simpler, a RISC wants approximately two instructions for each complex instruction and, consequently, the size of the PM is doubled. Nevertheless, the Fetch-Decode-Execute-Store pipeline of the whole processor improved the execution speed several times in comparison to the CISC [**11**]. The RISC structural design involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor could be :

1. Relatively few instructions.
2. Relatively few addressing modes.
3. Memory access limited to load and store instructions.
4. All operations done within the registers of the CPU.

5. Fixed length, easily decoded instruction format.

6. Single cycle instruction execution

7. Hardwired rather than micro-programmed control. [**12**]

Finally, the concept of NISC offers an entirely new approach for design of custom processor. It completely deletes the decode stage and stores the control word in the PM. In other words its control is very simple that only needs control words provided by the compiler to be stored within the PM. Consequently, the NISC architecture is the fastest of the 3 mentioned architecture to be introduced in the market. Additionally, NISC is reconfigurable, which allows for very fine tuning to any application and performance. It eradicates instructions to facilitate faster execution and better customization of process. NISC compiler, without instructions, has full control of all the components and connections in the data-path which permits it to achieve better resource utilization.

The major benefits of the NISC technology could be:

1. High productivity gain through RTL code generation i.e. very simple control unit through RTL stage.
2. Low Power, performance and area optimization.
3. Easy system Integration.
4. Providing ultimate reuse and re-reconfigurability.
5. Excellent market penetration [**11**][**13**].

2.4 Floating Point Number Representation

Floating point and fixed point numbers are the two formats used to store and manipulate numbers within the devices. Fixed-point numbers are the real data type for numbers. Floating point numbers describe a method of representing an approximation of a real number in a way that can support a wide range of values.

Most processor manufacturers such as IBM, HP, DEC and Cray used to have their own floating-point format. leading to a lot of inconsistency as to how the same program behaved on different machines. Today, almost all floating-point arithmetic is based on the IEEE-754 standard for floating-point arithmetic. [**14**]

Floating-point numbers are represented in scientific notation of fraction (F) and exponent (E) with a radix of 2, in the form of F×2^E. There are mainly two representation schemes: 32-bit single-precision and 64-bit double-precision. **Figure 2.4**

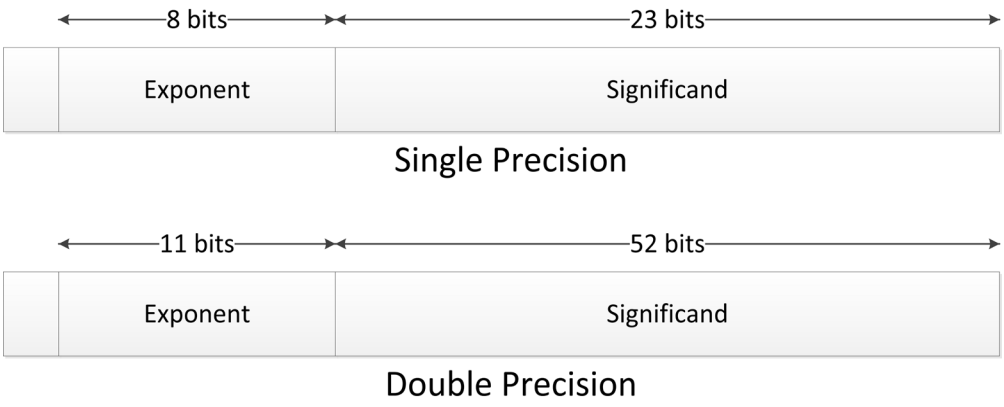


Figure 2.4: Floating point number representation

Double-precision binary floating-point is a commonly used format on PCs, due to its wider range over single-precision floating point, in spite of its performance and cost. As with single-precision floating-point format, it lacks precision on integer numbers when compared with an integer format of the same size.

3. Related Work

3.1 NOA'S-Arc: NISC based, Optimized Array Scalable Architecture

In [**1**], the authors proposed a new architecture of an array of reconfigurable NISC (No Instruction Set Computer) processing elements connected by a reconfigurable NoC to target statically scheduled scientific computing problems.

A single PE contains two floating point units (Divider, FMA), reconfigurable control unit and-16KB X 64 data cache unit. The data flows in the data path performing the floating point operations in parallel where the FMA is a specialized floating point unit that can operate on three operands at once for multiply add/sub instructions, and the divider is another floating point unit which can perform the division instruction with low latency.

Their proposed design has a reconfigurable control unit which optimizes the flow of data and minimizes the control processing time. Also a reconfigurable NoC that moves data between different processing cores was designed due to the high communication required between different cores and which provides good scalability of the design. Both the reconfigurable control unit and NoC allow scalability and customization of the hardware according to the problem.

For the control technique, they used the NISC approach where a reconfigurable control memory controls the data path and optimized the execution of the specified application. The statically scheduled problems were advantageous to them as they statically scheduled the whole operation which allowed them to make any modification needed in the operations of the micro program counter which saved a significant processing time.

The architecture is simulated using a tile of 64 PEs to run LU decomposition algorithm of a dense matrix, and the results show a performance of 177 GFLOP/s which accounts to 92.6% of the peak performance of this design and this performance was achieved due to three reasons; First, the reco figurable NoC was responsible for handling all the processing elements' communication to free it for execution making it efficient in data movement both at the initial processor state and inter processors

data movement during the operation execution. Second, the optimized data path and reconfigurable control which when programmed allowed fast parallel execution of the matrix update phase and reduced the control processing time significantly. Finally the employed load balancing technique distributes the operations among the cores before the execution phase starts, reducing significantly the need for data exchange between the PEs for load balancing reasons [**1**].

3.2 An Example of Multi-core Based Implementations For Solving Statically Scheduled Problems

In [**15**], the authors used multicore solution to optimize LU factorization where this paper to speed up the process of LU decomposition that is used in linear system solving on a multicore using OpenMP interface it discussed three parallel strategies: a proposed pipelined model, row block and row cyclic. The row block and row cyclic techniques use static scheduling approach with patterns of different matrix storage which is trivial parallelizing techniques.

The proposed algorithm based on the pipeline technique; the authors proposed a simple and fast model to predict the performance of the LU decomposition method with the pipelining techniques.

The experimental results on a multicore CPU showed that the proposed pipelined techniques when implemented achieved good overall performance compared to the two trivial methods.

3.3 An Example of GPU Based Implementations For Solving Statically-Scheduled Problems

Another solution is using GPU as a co-processor for solving linear algebra matrices by programming GPU. In [**4**] the author proposed an algorithm to reduce LU factorization problem to a series of rasterization problems that are mapped for GPU so as to solve them. The problem data is represented in 2D formats which should match GPU grid. It introduced delay but with high utilization of GPU pipeline is done by parallel data transfer. Also, performance was tested using full and partial pivot scheme.

Mapping LU algorithm GPU core is not easy to do as it depends on massive memory which will not fit on the GPU memory as it's small in size increasing delay.

3.4 An Example of an FPGA-Accelerated Implementation For Solving Statically Scheduled Problems

In [**5**], the authors used the new technologies of Field Programmable Gate Array (FPGA) to implement a parallel machine for LU factorization that will enhance the performance of solving linear systems on a system on a programmable chip (SOPC) board which provides many opportunities of configurable computing.

They mapped a highly parallel Block-Diagonal-Bordered algorithm for LU factorization to their Non-Uniform Memory Access (NUMA) shared memory multi-processor. To implement this multiprocessor they used a commercially available NIOS Altera soft IP RISC processor which is a fully configurable soft IP that offers over 125 MHz in the Stratix FPGA.

The communication network between processors and peripherals in an IP-based multiprocessor design is a critical element for good system performance as the communication overhead has always been a bottleneck for current parallel architectures. There are some disadvantages for this design; one of them is that a single precision FPU was implemented in their design which is not as accurate as the double precision FPU. Another one is the use of a shared memory will lead to a memory bottle neck in case of large matrices.

3.5 Raw Processor, The Birth of Tile Processors

In 1997, the raw architecture was first introduced by the authors of [**6**]. Their approach was to build an architecture based on multiple identical tiles where each has its own instructions and data memory. The raw processor consists of a number of interconnected tiles, arithmetic logic unit (ALU), registers, configurable logic and a programmable switch for dynamic and static routing.

Each tile has a RISC-like pipeline and is connected to other tiles with a pipelined point-to-point network. The presence of many distributed registers allows a greater degree of instruction level parallelism (IPL). The raw processor uses switches instead of buses as interconnects, where it supports single cycle message send and receive operations. Any signal in a raw processor only travels a single tile in one clock cycle.

The use of distributed memory (Static RAM) across the tiles will eliminate the memory bottle neck and consequently shorter latencies. The control unit of the raw processor consists of two sets of control logic and instruction memory. One of them controls the operations of the processing element of each tile and the other one is responsible for the routing instructions and that reduces the latency as the routing of independent messages passing through a switch is not disturbed by the PE operations.

Among the raw architecture advantages there is the inter-tile communication with short latencies (Close to those of the register access) because of the tightly integrated, synchronous network interface, but the static scheduling enables the operands whenever needed which means there is no need for explicit synchronization. Moreover, the programmer can configure the logic in each tile to construct specific operations that suit the needed application. These advantages guarantees high switching speeds and simplified hardware design.

3.6 Tile Processor, The Raw Processors Delivered Commercially

After several years, a start-up named Tilera, owned by some of the fathers of the raw processor, presented to the market processors based on the highly scalable tile processor architecture. Tile Processor Architecture [**16**] is a two-dimensional array of processing elements (each referred to as a tile), connected via multiple two-dimensional mesh networks.

The architecture is scalable and provides high bandwidth and extremely low latency communication among tiles. Each tile is a powerful, full-featured computing system that can independently run an entire operating system. Each tile implements a 32-bit integer processor engine utilizing Very Long Instruction Word (VLIW) architecture. An individual tile is capable of executing up to three operations per cycle. It also has additional instructions to accelerate DSP applications that require saturating arithmetic and unaligned memory accesses.

4. Proposed Design and Implementation Details

This chapter provides a full functional description of a single processing element in our proposed architecture. It is worth mentioning that in order to have multi-cores, the architecture of the single core will only be duplicated to obtain the desired number of cores. These identical cores are connected together via the NoC router in order to maintain communication and data exchange between cores.

The different components of the single processor will be explained later in this chapter in details.

4.1 Overall Architecture

The single core architecture is shown in **Figure 4.1.** In the proposed architecture we are focused on the design and implementation of reconfigurable processors arrays to serve a wide class of problems which are known as statically scheduled problems. Examples are matrix operations and DSP applications.

The objective of the project is to design and implement a simple processor array targeting statically scheduled problems. It preserves a large spectrum of applications in high performance computing. The implementation is targeting an operating frequency of 1 GHz.

It is worth mentioning that this is a multipurpose architecture not a general purpose one. The multipurpose architecture is faster than general purpose, and also it's different from the special purpose as we can add other applications with only minor changes. Below, we will discuss briefly the different components of the processor architecture, full description and function of each block will be discussed in details.

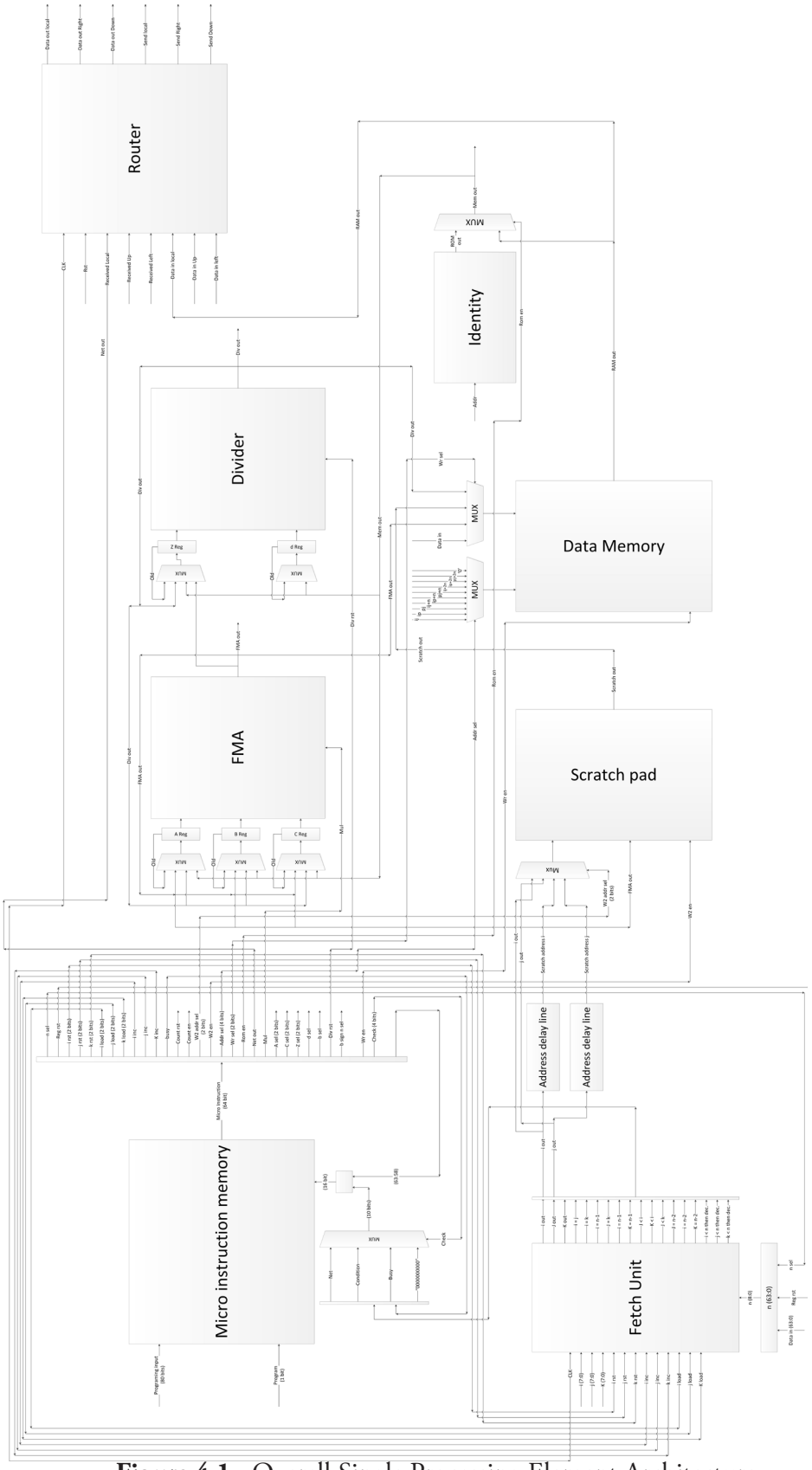


Figure 4.1.: Overall Single Processing Element Architecture

4.1.1 Floating Point Fused Multiply Add Unit (FMA)

The fused multiply-add differs from the traditional floating-point data paths that spatially separated each operation. A multiplication was performed on multiplication hardware, additions on addition/subtraction hardware and division on division hardware. Such floating-point units can actually be thought of as a collection of data paths, one for each operation. Our FMA unit implementation entirely replaces the per operation structure. Any combination of multiplication and addition/subtraction can easily be derived from the FMA.

By using the constants 1 and 0, multiplication, addition and subtraction are easily realized on FMA units. It takes three input operands each of 64 bits, the output is also 64 bits which is the result of the multiply and accumulate operation (A×B + C).

The FMA unit has seven stages pipelines in order to increase the throughput of the unit and allow parallel operation. The result of the FMA unit is obtained after 7 clock cycles.

4.1.2 Floating Point Divider

The floating point divider is one of the most challenging and complicated logic blocks in arithmetic intensive circuits [**17**]. It is the most complex of the four basic arithmetic operations and the hardest one to speed up [**18**]. For achieving high performance divider the implemented divider is based on SRT Radix-4 algorithm.

The divider unit has two input operands each of 64 bits. It has only one output which is the quotient and it is also 64 bits.

It is worth mentioning that the divider proposed in this book has much higher performance than other related work. We have optimized our design in order to achieve higher performance as will be explained in details later in this chapter. The operation of the implemented floating point divider is sequential. Therefore the divider keeps performing many iterations till finally obtain a valid result. The true output of the divider unit is obtained after 30 clock cycles.

4.1.3 Micro-Instruction Memory (Re-configurable Control Unit)

One way to think about the control of any sequential circuit is to try to implement this using finite state machine. Finite state machine is a better way to represent a sequential system whenever it is difficult to implement it using ordinary design techniques. However, using this approach will result in a hardwired control unit. That's why in our design the control unit is not implemented as a FSM but as a block of memory.

In our design, a micro-instruction is considered a vector of data containing the current state and a key to the next state. Our micro-Instruction block was designed using Content Addressable Memory (CAM) which has a different way of operation from other memory architectures. The general idea of Content Addressable Memory is that it compares the input data against the stored data in parallel and outputs the address of the matched data. Based on the way the CAM operates, we implemented our Micro-Instruction Unit; first, the CAM is programmed with all the possible states in our design, almost 128 states. It worth saying that the CAM width is 80 bits, distributed to two parts; 16 bits for the current state and 64 bits output. Then, the 16-bit input is applied to the CAM which, by default, compares the input with the current state 16-bit field and when find matching it generate the 64-bit micro-Instruction as output. The 64-bit output represent the various control signals in the core. Six bits out of those 64 bits are used as a key for the next –state, where they are combined with 10-bit generate from the fetch unit, resulting in the 16-bit input which is used in generating the output of the control word when comparing with the current state field.

4.1.4 Fetch Unit

In order to obtain the output micro-instruction, the input addresses are combined with the bits represents the current-state of the control unit. Unfortunately, using this approach will result in a very large control word width. In order to overcome this, the fetch unit is optimized to decrease the number of those bits. Mainly, the fetch unit consists of number of counters that generate the addresses of the row and column where the element being processed occurs in the matrix. It also generates the iteration index for applications that need many iterations on a specific row of the matrix. As mentioned before, in order to decrease the number of the output bits, comparators where added to the fetch unit where the addresses are being compared and the output of the fetch unit will only be the status of that comparison i.e. i greater then j, i greater than k, j equals n, etc. where i is the row address, j is the column address, k is the number of iteration and n is the size of the square matrix. The output of this unit is 10 bits which when added with the current state of the control unit can indicate the next state of operation.

4.1.5 Data Memory

The data memory used in our architecture is a single port memory generated using Artisan 65 nm single port T memory compiler. The total size of the Data memory unit is 2K×64 bit.

4.1.6 Identity Unit

The identity unit is used in some applications which required the involvement of square identity matrix such as the matrix inverse operation. If the row and column addresses are the same i.e. (i=j) the identity unit output is 1. Otherwise, the output is high impedance.

4.1.7 Scratch Pad Memory

The Scratch Pad memory is used as a temporary storage memory for data of size 128×64 bit. It was found that in some applications such as memory inverse, we needed another memory for data to save the results before saving it in the main data memory. As to obtain the actual correct value of each element in the matrix, we need to perform more than one iteration on the whole raw. Therefore, we can't write back the value obtained after the first iteration in the main data memory as the result won't be accurate yet as we will need to use the values located in the main data memory to perform the next iterations.

After finishing all manipulations on a specific raw and saving it in the scratch pad, it can then be copied back to the main data memory as these results are the final correct results. In the rest of the chapter, we will discuss some of the previously mentioned blocks in further details.

4.1.8 NoC Router

When dealing with multi-core processors, then you are talking about many cores on single network in what is called "Network On Chip" or its abbreviation "NOC". The network is responsible for forwarding a message "packet" from a certain source to a certain destination based on the header of the message. These cores will need to connect with each other according to the problem the processor is facing; the main component upon which the cores are connected is the router. The router is considered the cornerstone by which the cores connect with each other. According to the header sent inside the packet, the router tries to forward the message; either to itself or to another core. During the design of NOC, a lot of constraints should be taken into consideration like security, maximum delay, and number of wires inside the network. Another point that differs from design to another is the way the router responds if many messages arrive at the same time, and what kind of arbitration will be used whether it was fixed or round-robin arbitration.

4.2 Fused Multiply-Add Unit

FPU is one of the most important custom applications needed in most hardware designs as it increases its accuracy and performance.

Recently, the floating-point units of several commercial processors have included a floating-point fused multiply add FMA unit to execute the double-precision fused multiply add operation (A×B)+C as an indivisible operation, with no intermediate rounding.

FMA operation is very important in many scientific and engineering applications like DSP. It can be used instead of floating-point addition by setting B=1 and floating-point multiplication by setting C=0.

4.2.1 Related Work

We investigate in this section some designs of floating- point fused multiply add (FP-FMA) units. We will mention and compare other FPU designs stating their architecture and performance.

Fused-Multiply-Add

Fused Multiply-Add Floating Point and Integer Data-path [19]

The author combines double precision Floating point's and integer's logic and arithmetic operations. Due to the difference between logic and arithmetic operations of both integer & floating point, it was not easy to achieve. The result of frequency of this FMA was 1.2 GHz on 65 nm technology.

Binary Floating Point Fused Multiply Add [20]

The proposed architecture is synthesized for Cyclone II FPGA device. For the proposed architecture, the overall delay is 70.985 ns. It occupies 13,930 from 68,416 total logic elements which are corresponding to occupying 20% of total logic elements. Also, the proposed architecture achieves a delay improvement about 25.5% as compared to the basic architecture. The increase of area in the proposed architecture is about 6.2%.

In [**21**][**22**][**23**], authors present several efforts to design fused-multiply-add operations. In [**23**], in a design that is very similar to ours, the authors presented a performance of 1.81 GHz on 45 nm technology. The total dynamic power reached 95 mW and the design used an area of 0.05 mm2.

Scaling these results to the 65 nm technology we are implementing on, we can see that our results that will be presented later are much better. The previously mentioned design has a pipeline of 6 stages depth, while ours has a depth of 7 stages. Although that design has lower latency, we believe that the gain in the clock speed compensates that defect in our design and is more important.

Floating Point Units

Cell Processor [24]

The first generation of Cell Processor runs at 3.2GHz; achieving theoretical throughput of 6.4 GFLOPS per single processing element. It included one double-precision and 4 single-precision arithmetic cores.

UltraSparc T2

The UltraSparc T2 Processor has eight cores and supports eight threads per core. Each core is equipped with one floating-point unit which is fully IEEE-754 compliant and implements double and single precision floating-point operations. The pipeline consists of six stages for the execution of addition, subtraction, multiplication and non-arithmetic operations.

Intel Itanium

Intel Itanium Supports the standard IEEE-754 single and double precision formats as well as several custom floating-point formats. Multiply-add is basically the only operation that has been implemented directly in hardware. All other instruction are derived from, or implemented based on multiply-add operations.

Dual-path Adders

The idea of a dual-path adder came from the observation that a full-length alignment shift and a full length normalization shift are mutually exclusive. i.e., if alignment requires a large shift then normalization does not and vice versa. The dual-path adder splits the data-path for addition into two separate paths. The Closet path that is chosen when the difference between the input exponents is zero or one, and the far path that is chosen when the difference is greater than one. The dual-path approach has recently been investigated for application in FMA. Although the performance results are promis-

ing, the penalty is that the area increases by approximately 45%.

4.2.2 Design Choices

Before we start describing the micro-architecture of the FMA, we need to re-call the floating point double precision number representation.

Number Representation

We use a double precision number representation which is one of the formats described in IEEE-754 standard that represents any number in the form of 64 bits where these bits are divided into 52 bits as significant, 11 bit exponent and 1 sign bit with (+1023) as maximum value and a minimum value equals to (-1022).

Supported Operations

There are two basic operations that the processor performs; addition/subtraction and multiplication. Both are implemented natively in hardware for floating-point to ensure maximum performance. Another interesting feature in the offered arithmetic operations is the "multiply and accumulate" operation (multiply-add). Besides improving the quality of floating-point operations, because only one round-off error is made, this operation eliminates load and store overhead of two separate instructions.

A summary of the supported operations I presented in **Table 4.1**.

Instruction	Input	Operation
Multiply-Add	A;B;C: Floating-point(signed)	C + A × B
Multiply-Subtract	A;B;C: Floating-point(signed)	C - A × B
Multiply	A;B: Floating-point (signed)	A×B
Add	A;C: Floating-point(signed)	C + A
Subtract	A;C: Floating-point(signed)	C - A

Table 4.1: Supported operations by our FMA

4.2.3 Architecture

Inputs and Outputs

In **Figure 4.2**, you can find an illustration of the interfaces of our FMA.

Figure 4.2: FMA I/O Interfaces

Data-flow

Figure 4.3 represents the data-flow inside the FMA block.

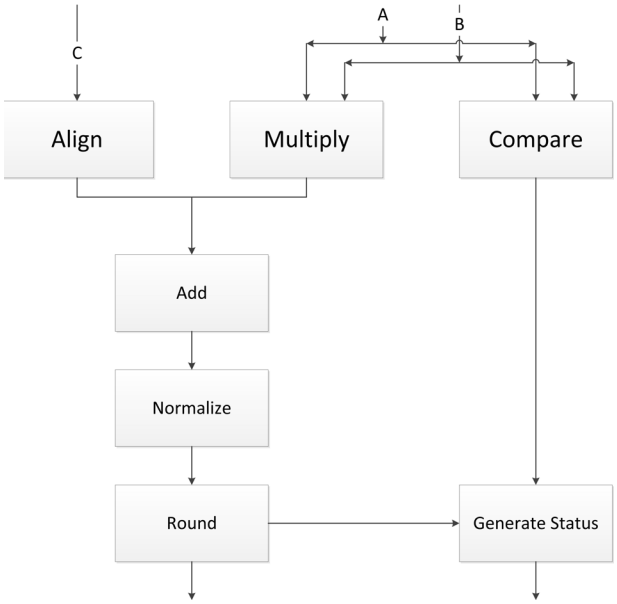


Figure 4.3: Simplified FMA Data-flow

Overall Architecture

Our architecture, presented in **Figure 4.4**, modifies the simplified data-flow presented in figure 4.3. It consists of 7 pipeline stages as illustrated. In the following section we will cut this design down and describe its detailed behavior.

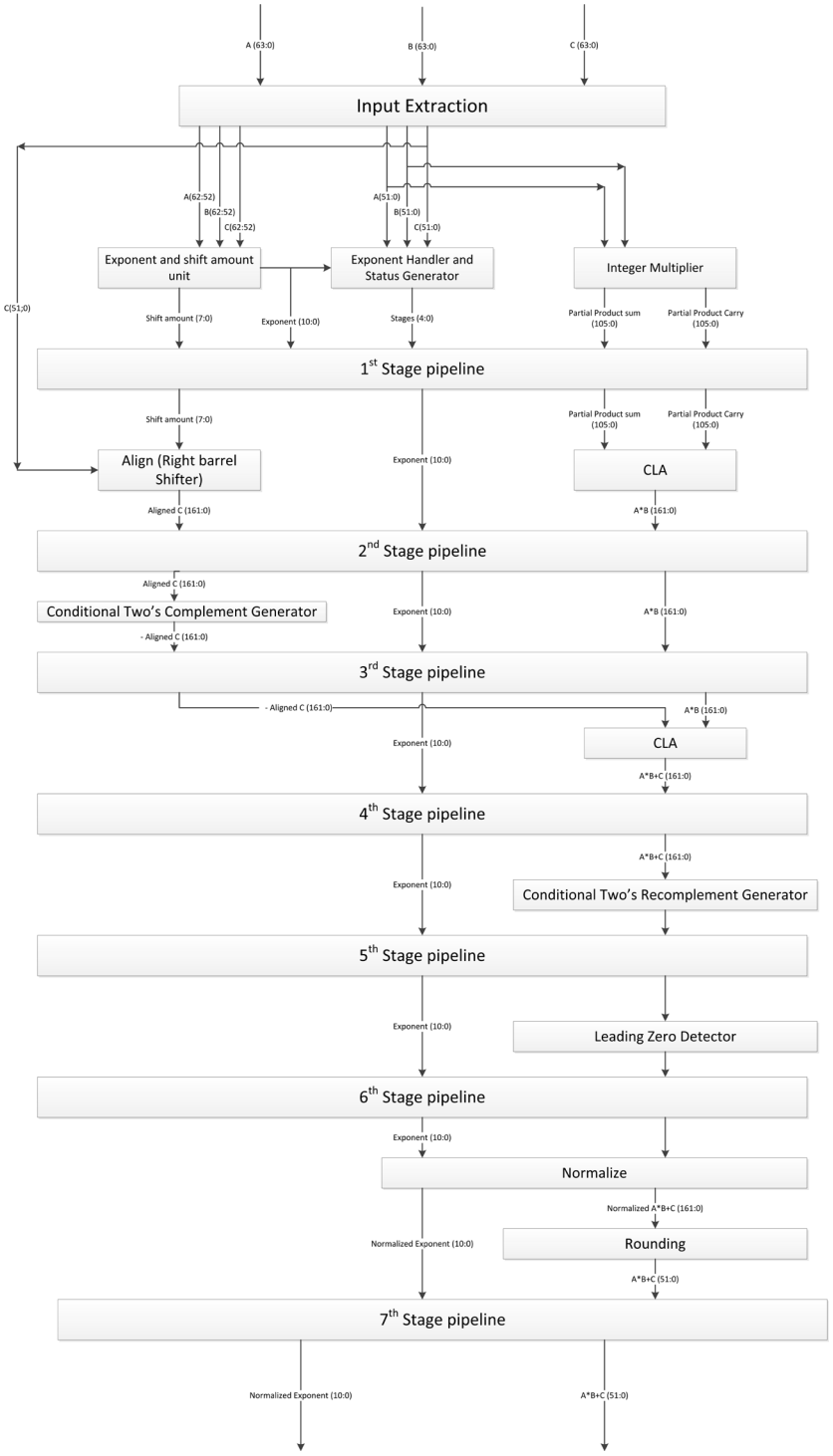


Figure 4.4: Overall FMA Architecture

Operand Alignment

It is a constraint that the added operands have the same alignment. To align the exponents, the absolute difference α must be found by subtracting the exponents. In FMA, the exponents of A and B are first added causing the bias to accumulate. So, the bias must be subtracted again. Once α is

found, the significant of the smallest operand is shifted α positions to the right and its exponent increased by α as shown in **Figure 4.5**

Alignment needs adders, subtracters, a comparator and a logical right shifter. It is rather straight forward to implement the alignment by shifting the smaller operand of the two addition operands.

However, for FMA it is costly to shift the smallest operand. Before this operand can be designated, the multiplication of A and B must be completed, leading to a massive delay. There are ways to carry out alignment more efficiently for multiply-add. The basic idea is to always shift the C operand which is immediately available at the start of the operation. **Figure 4.6.** presents this idea.

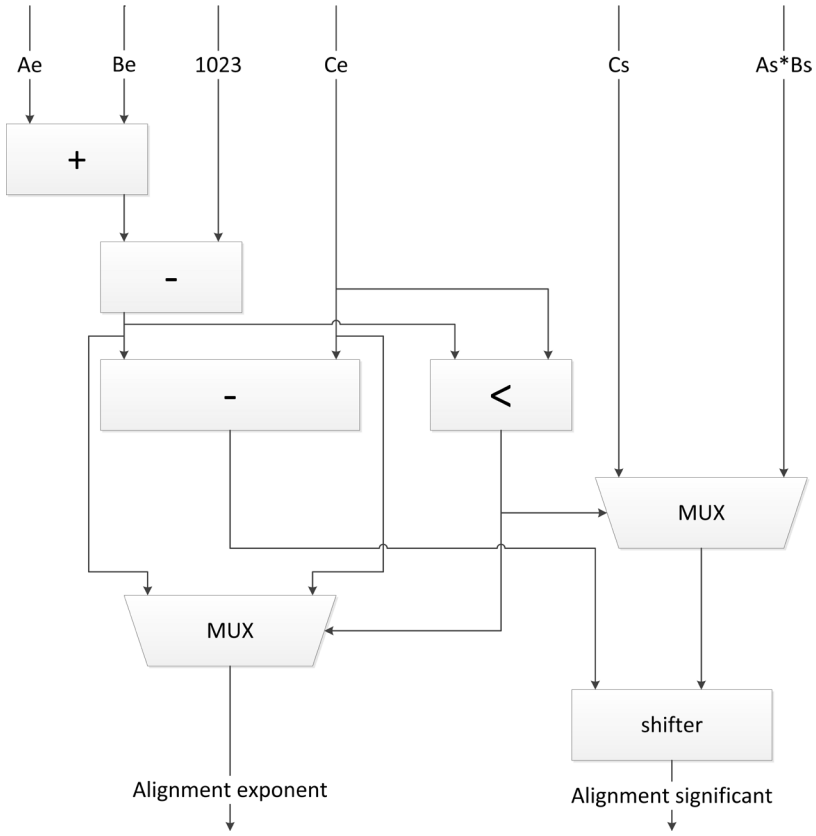


Figure 4.5: Operand Alignment

The operand C is aligned with the product, while operands A and B are multiplied to form the product itself. Alignment of two floating-point numbers consists of two actions: adjusting the exponent of one operand to match the exponent of the other and shifting the significant of that same operand to match the new exponent. Adjustment of the exponents is often not discussed in floating-point

arithmetic' related publications. This is mostly because it is assumed to be trivial. This is because that the multiplication is different from addition/subtraction, where the exponent of the result is simply the exponent of the operand that was not shifted. However, in multiplication you should put in your consideration the overflow, underflow, and non-regular input conditions. Especially in the case of FMA, exponent adjustment is anything but trivial.

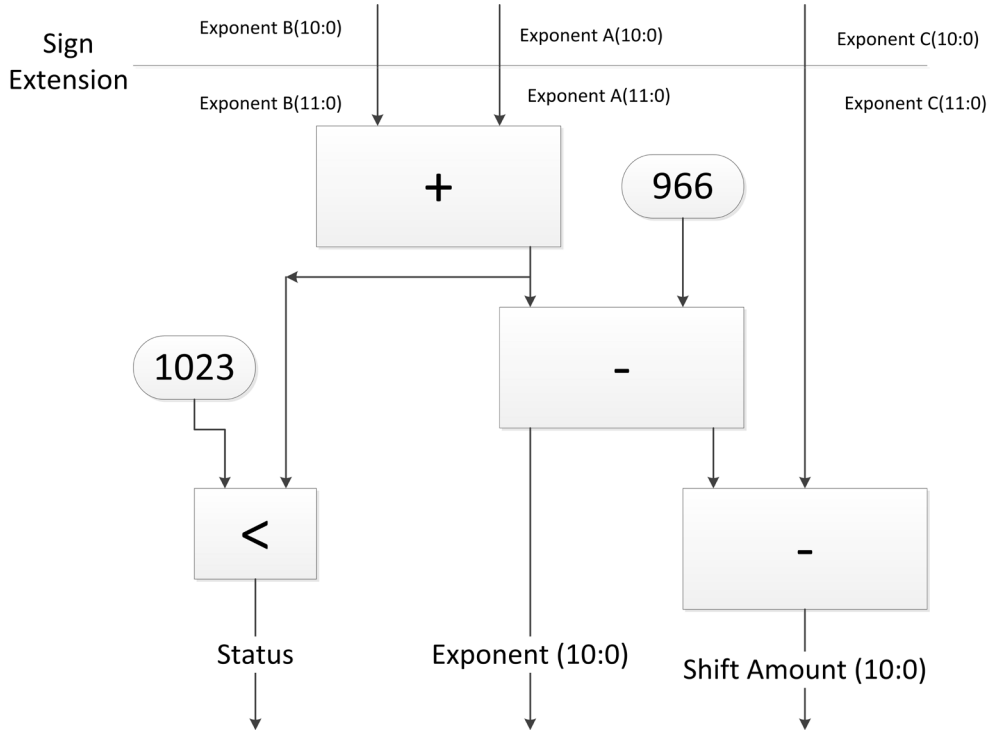


Figure 4.6 Exponent Adjustment

Shift Count = Ae + Be - Ce - Bias + (Significant width+ 2 Gaurd bits+ 2 Multiply compensation) (**4.1**)

Exponent = Ae + Be - Bias + (Significant width+ 2 Gaurd bits+ 2 Multiply compensation) (**4.2**)

Exponent = Ae + Be - 1023 + 57 = Ae + Be - 966 (**4.3**)

This flow is based on **Equation 4.2.**, the equation states that after adding the exponents of A and B, the bias must be subtracted and an implementation of specific offset must be added. The bias and the offset are both constant. So, **Equation 4.2** can be simplified to one addition and one subtraction. The bias for an 11-bit exponent is 1023 and the offset for a significant of 52 bits with 2 guard bits is 57 (53 significant bits + 2 guard-bits + 2 multiply compensation). **Equation 4.3** is implemented by adding the exponent of A and B in a 12-bit adder and subtracting the constant in a 12-bit subtractor

as shown in Figure 4.6. The result is also used to drive the shifter that aligns the significant.

Later, Operand C can efficiently be aligned with the product by placing C entirely in front of A&B. **Equation 4.1.** was derived to find a right-shift count for C. This equation, that actually describes the absolute difference between the exponents of "A, B" and C, only slightly differs from the one used to find the exponent. By subtracting Ce, we find the shift count needed to drive a shifter that aligns the significant. Because the exponent of A&B is already known at this point, the required shift count is straightforward to implement. The exponent of C is however not a constant. So, another subtractor is needed, as shown in **Figure 4.6.**

Multiplication

Together with addition, multiplication is the main resource to optimize for better utilization in floating-point instructions. Many highly optimized multiplier designs exist that can be used to facilitate the multiplication of A and B. Fast multiplication is needed for high performance. Fast multipliers are trees or arrays of adders. They are large components making them high priority candidates for area optimization. In [**18**], the reader can find a detailed explanation of arithmetic multiplication algorithms discussed below.

The most efficient multiplier is a partial product multiplier based on modified radix-4 Booth encoding and a Wallace tree structure as presented in **Figure 4.7**

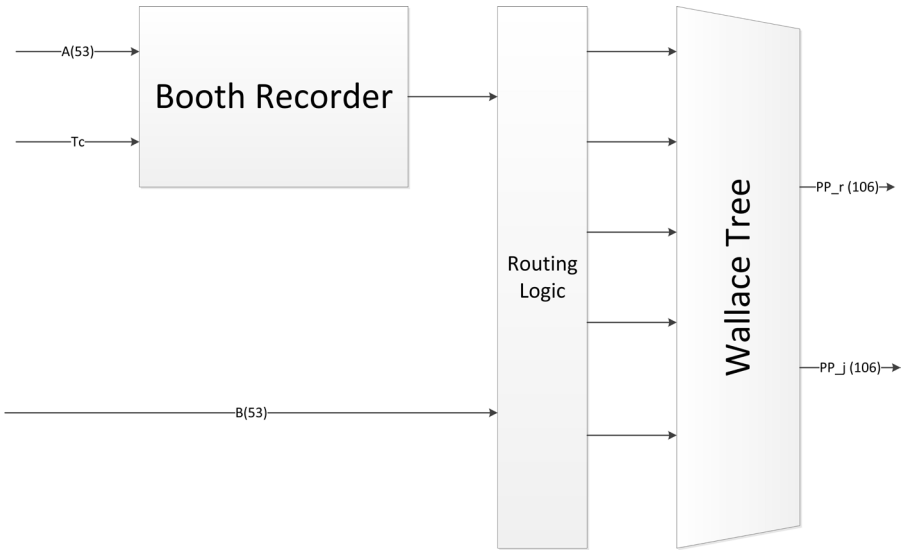


Figure 4.7: Integer Multiplier

Booth Encoding

Booths' algorithm serves two purposes; first, it enables us to multiply signed (two's complement) numbers, this is not directly relevant for IEEE floating-point multiplication, second, it helps reduce the number of partial products, this can increase the speed of the multiplier and reduce its area.

Wallace Tree

The final product is obtained by summing up all the partial products to together. Based on the radix, the number of partial products can be determined. If we would just perform one addition after another, the latency of the multiplier will render it useless in terms of performance. The role of Wallace tree appears here, where the partial products are summed in a tree-like structure. This reduces the delay from n additions to log2(n) additions. This idea was first presented in [**25**]; a special adder is used to eliminate the carry propagation delay, this is needed so as to match the latency of alignment such that the latency of the critical path in the first stage is minimal. The Wallace tree used in our design is presented in **Figure 4.8.** However, in order to fully understand the operation of Wallace trees, one needs to understand the carry-save adders topology which will be discussed in the next section.

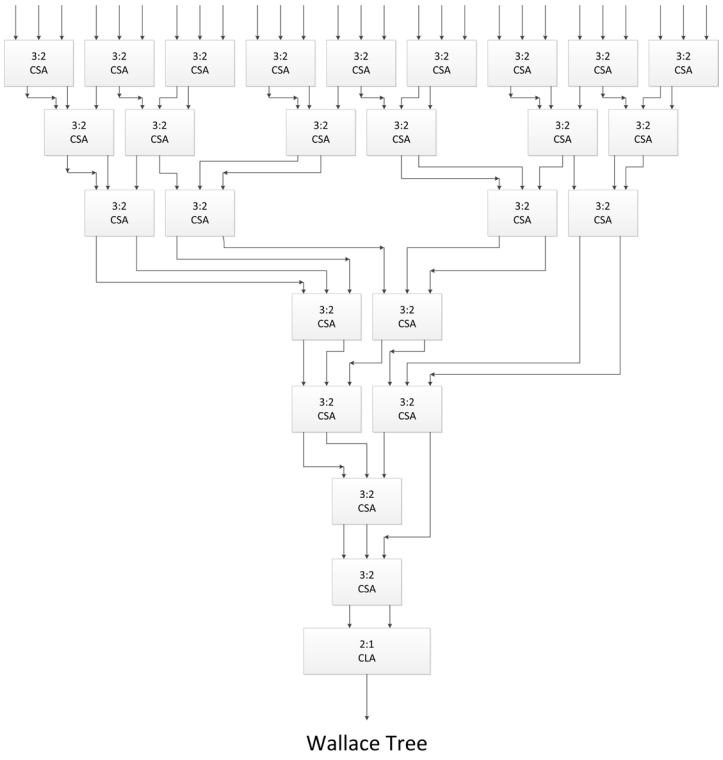


Figure 4.8: Wallace Tree Adder

Addition

First of all, we present a comparison between the most dominant three adders topologies[**26**].

Ripple Carry Adder (RCA)

The ripple carry adder is constructed by cascaded full adders (FA) blocks; where the carryout of one stage is fed directly to the carry-in of the next stage. One of the most serious drawbacks of this adder is that the delay increases linearly with the bit length. However, the advantages of the RCA are lower power consumption as well as compact layout giving smaller chip area.

Carry Look-Ahead Adder

Carry look-ahead adder is designed to overcome the latency introduced by the rippling effect of the carry bits. This can be eliminated by carry look-ahead adder. This adder is based on the principle of looking at the lower order bits of the augends and addend in case a higher order carry is generated. This adder reduces the carry delay by reducing the number of gates through which a carry signal must propagate.

Carry-Save Adder

The carry-save adder reduces the addition of 3 numbers to the addition of 2 numbers. The propagation delay is 3 gates regardless of the number of bits, just like CLA. The carry-save unit consists of n full adders, each of which computes a single sum and carry. Based on the corresponding bits of the three input numbers, the output carry is generated.

The main application of carry save algorithm is for efficient CMOS implementation of much wider variety of algorithms for high speed digital signal processing. Carry propagation can be speed up in the partial product line whenever CSA is applied.

Having investigated different adder topologies, we are no ready to explain the fused multiplication addition process. The multiply-add structure is one of the basics structure in FPU; this component is used for both multiplication and addition. The conditional complementing and re-complementing unit also have to be included in order to support subtraction. Such a module is presented in **Figure 4.9**.

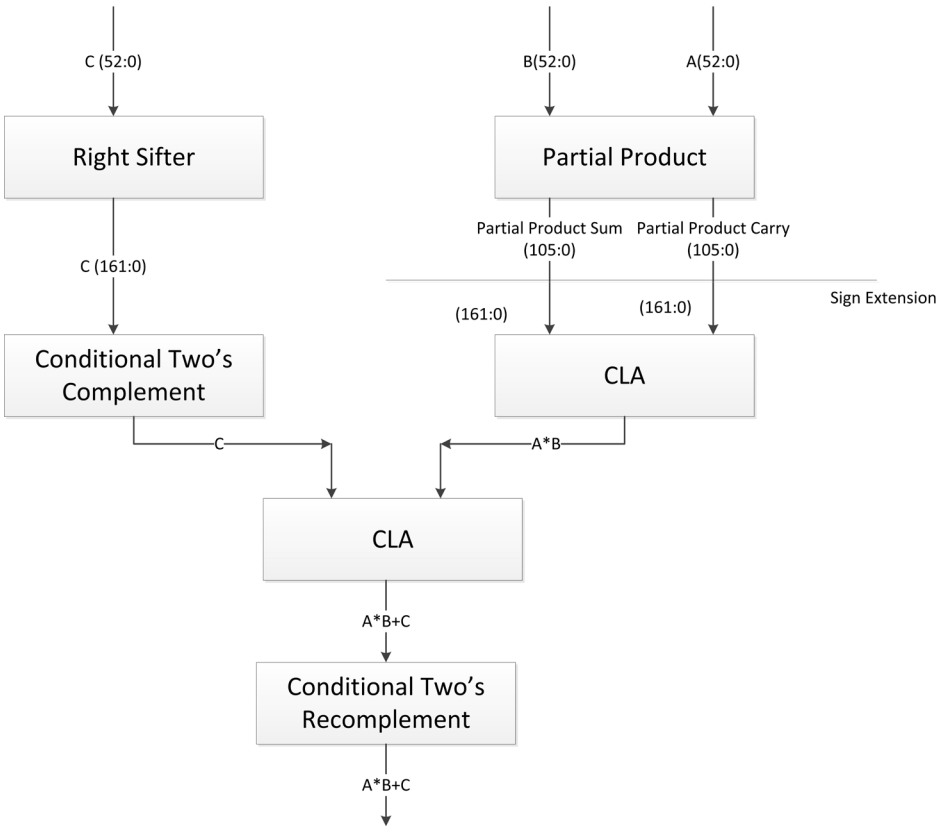


Figure 4.9: Fused-Multiply-Add

After being aligned, the 164-bit addend C may need to be complemented in case of effective subtraction. Fortunately, the effective operation is very easy to determine. A simple XOR gate can be used to indicate whether or not the effective operation is a subtraction. This is done normally except if any of the operands is zero, in this case the addend should never be complemented. If A or B is zero, the C operand is simply forwarded to the output (i.e., A×0+C=C or 0×B+C=C). If C itself is zero, it should not influence the result (A×B+0=A×B), hence C should never be complemented if any of the operands is zero. The effective operation is therefore combined with the logical OR of the zero detects signals from the instruction decoder. Based on the evaluation of the resulting control signal, the addend C is inverted to obtain its complement.

The (complemented) addend is also extended by one bit. Because the partial product obtained from the multiplier is always signed due to Booth encoding, a false carry out may occur. Some of these false carry out cases could be eliminated by adding a sign bit explicitly. Although including the sign-bit is not sufficient to solve the entire problem, we mention it here because the entire data-path needs to be arranged for 162-bit wide data.

In order to keep the end-around carry addition fast enough, a carry look-Ahead adder is used as a final adder. Another constraint is that the end-around carry addition requires that the final addition is incremented in case the product was smaller than the addend besides having subtraction as an effective operation.

The logical value of (((A sign XOR B sign) XOR C sign) ^ Cout) is fed to the secondary input of the XOR gates. If the carry-out is 0 and the addend was complemented for effective subtraction, the result must be re-complemented.

Normalization

The sum obtained from the multiply-add core may have to be normalized. It consists of 2 stages; the first is "Leading Zero Detection" (LZD) which detect the number of zeros before first '1' comes from the left ,while in the last stage a shifter actually performs the left shift and the according exponent adjustment based on the prediction of the second stage. Together the leading zero detector and shifter form the Normalize sector of the data-path as shown in **Figure 4.10.**

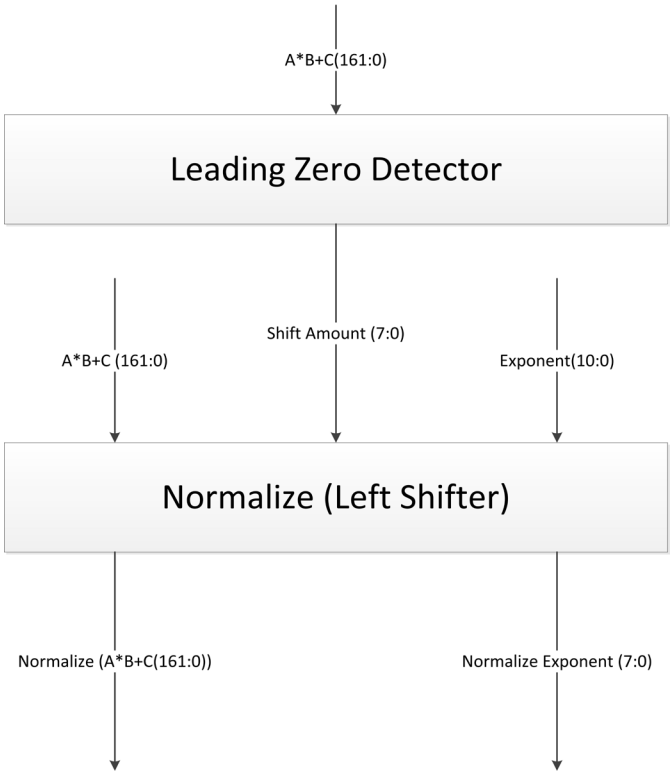


Figure 4.10: Normalization

The LZD circuit counts the number of zeros until the first 1 comes and then shift the significant with this number of zeros so the MSB is 1. Generally, it is often that the left shifter is used for normalization than right shifter. Theoretically, there is no difference between both arithmetic shifters and logic shifters, where zeros are inserted in both. A common 161-bit shifter suffices for both purposes.

Rounding

In our FMA, we rounded the normalized result of 161-bit wide by truncating it to the first 52 MSBs.

4.2.4 Results

The design has been simulated and verified on a very large set of test cases, including but not restricted to the special cases when one or two of the operands are 0 or infinity. It has been synthesized on TSMC low power 65 nm technology using Synopsys Design Compiler, in order to be used in the final chip placement and routing. Here are the synthesis results in **Table 4.2.**

Critical Path Length	0.6 ns
Maximum Frequency	1.66 GHz
Latency	7 cycles
Cell Count	16200 cells
Total Cell Area	0.05 mm^2
Total Dynamic Power	132 mW
Number of Nets	16605 nets

Table 4.2 : Synthesis results of the FMA

Scaling the results presented in [**23**], we get 1.26 GHz maximum frequency, 146 mW total dynamic power and 0.083 mm^2 so it is clear that despite the extra pipeline stage, our results are much better.

4.3 Floating Point Divider

The floating point divider is one of the most challenging and complicated logic blocks in arithmetic intensive circuits [**17**]. It is the most complex of the four basic arithmetic operations and the hardest one to speed up [**18**]. Thus, dividers are more expensive and/or slower than multipliers.

Fortunately, division operations are less common than multiplications [**18**]. Actually, the divider usually ends up as the critical path of many designs [**17**]. There are three types of dividers [**27**]:

1. Combinational (Array) Dividers.
2. Convergence Dividers.
3. Sequential Dividers.

Combinational dividers are useful for smaller bit-widths and lower frequency designs. For area centric and high frequency designs sequential dividers are most suitable [**17**]. However, it is not the scope of this book to describe the difference between these two types of dividers.

4.3.1 Sequential Divider

In chapter 13 of [**18**], the division algorithms are discussed thoroughly from the arithmetic point of view. The basic block diagram of sequential divider is shown in **Figure 4.11**.

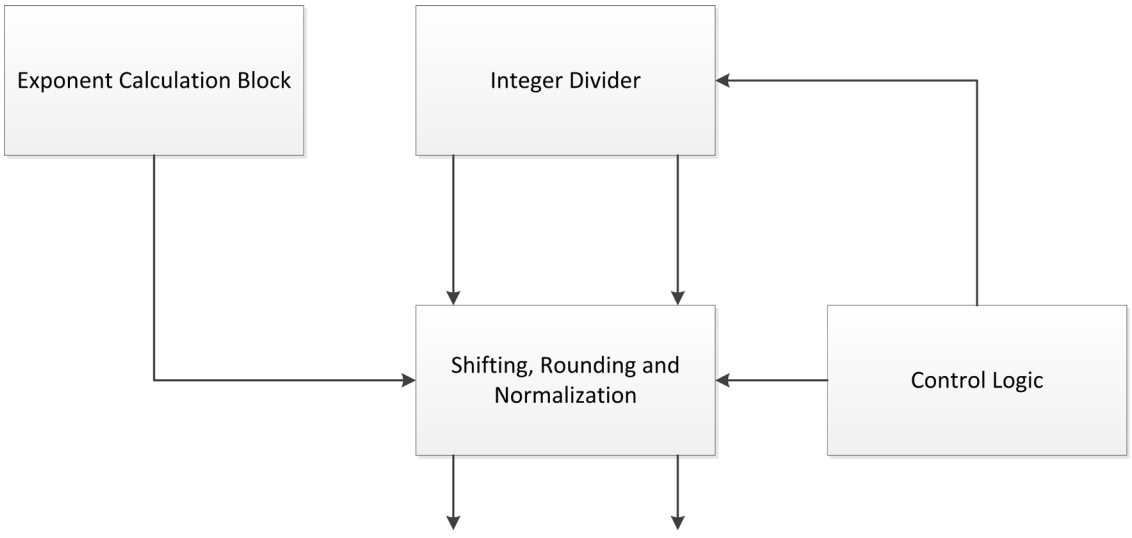


Figure 4.11.: Basic Sequential Divider

4.3.2 Design Choices

The division operation is reduced to successive subtraction operations. To implement a straight-forward fractional division algorithm, we need a 55-bit subtracting circuit, which can be either implemented using a carry-propagate adder which induces a large delay reducing the speed, or using a carry-look ahead adder, which has large area with a little advantage in terms of delay over carry-propagate adder if compared with the loss in area.

Another problem that we need to optimize while choosing the architecture is that, unlike the multiplication algorithms as we discussed in the background section, the partial remainders depend on each other and their generation cannot be done in parallel. This leaves us with either choosing to implement a large-delay cascaded divider sacrificing the speed of the circuit or implementing a sequential divider, which has proved as we mentioned earlier in the related work section to be the most reliable implementation. However, this implementation can only generate one digit of the quotient per cycle i.e. a normal binary double precision floating point divider needs at least latency of 54 cycles to get the final quotient.

After we cleared out the parameters according to which we will choose the optimum architecture for our needs, we started investigating the available architectures [**27**][**28**] for implementing the divider:

1. The straight-forward paper-and-pen divider algorithm: it was clear that this architecture is not suitable for any of the previously mentioned problems. It needs a slow 55-bits subtracting circuit, the full 54-cycles latency and large area.

2. Digit recurrence algorithm: This algorithm optimized the previous in terms of operations needed, but we still need to implement nearly the same hardware, so it doesn't optimize any of the speed or the area. However, some of its implementations had the capability of shifting over zeros, so it reduced the latency for some values of the operands.

3. SRT Algorithm [**29**][**30**]: As we have already explained the algorithm thoroughly in previous sections, the SRT algorithm uses a redundant format for saving the partial remainder; it eliminated

the need for the 55-bit subtractor, replacing it by a very fast 55-bit carry-save adder and, therefore, increasing the speed. Also, it doesn't have much advantages in terms of area due to the added look up table, and the circuit that generates the multiples of the divisor to be selected by the divider. Moreover, the normal radix-2 SRT algorithm has the same latency as the previous algorithm [**31**][**32**]. However, the SRT Radix-4 algorithm cuts the latency to half this value. In fact, we could use even a radix-8 or radix-16 algorithm that have low latency, but it would add much area to the look up table, and it will produce a new problem considering the generation of the multiples of the divisor. As we will see mentioned in the background section, it used to be a problem to generate the 3d multiple of the divisor, and of course the case is worse for the -3d multiple [**31**][**33**], and for long time, the designer's choice was to use an SRT Radix-4 algorithm that uses only the multiples: -2d, -d, 0, d, 2d [**34**][**35**][**31**][**36**] [**37**][**38**][**39**][**40**][**41**] [**42**], although this needs a large, slow, complex look up table. As the technology of VLSI industry advances, the generation of the 3d multiple became a strong option to consider [**42**] due to the low area and faster look up table it requires. Also, we proposed here as new idea to implement such circuit in a faster way. Yet, still it would be slow to implement the 7d multiple needed in the radix-8 algorithm.

4.3.3 Approach

After investigating all the previous architectures thoroughly, and regarding our need to design a fast high performance divider that copes with the other parts of the processor at a frequency greater than 1 GHz, it became clear that the best option was the SRT Radix-4 algorithms with the multiples -3d:3d.

We started implementing this architecture on the circuit level, investigating all possible optimizations possible to enhance its performance, as will be discussed in the implementation details section.

Our divider was synthesized as a stand alone core reached a frequency of 1.28 GHz and latency of 30 cycles on TSMC low power 65nm design kit.

4.3.4 Radix-4 SRT Division

The theory of SRT algorithms is well described in [**18**]. we will use the same notation here where p means partial remainder, d divisor, z means dividend and q means quotient. The p-d graph corresponding the division algorithm we used is represented in **Figure 4.12.**

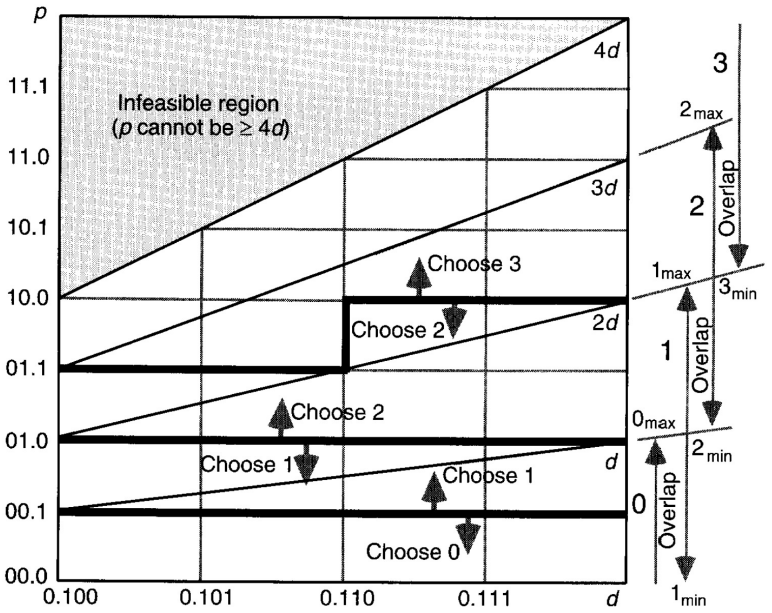


Figure 4.12.: Folded SRT Radix-4 Maximally Redundant p-d Graph [**18**]

4.3.5 Related Work

We investigate in this section some designs of SRT Radix-4 dividers from the literature. We will not discuss Radix-2 designs as it is not the scope of our project, but we have pointed to some of them in the previous section in case anyone is interested.

In the past, the area of the chip was the most crucial design aspect, so the usage of large high speed adders was avoided. To do so, most of the designs used only the minimally redundant digit set. This choice always led to a complicated slow look up table. Not only did it introduce a large delay overhead, but it was also complicated to implement. In [**43**] Intel explains a flaw that has been discovered in the floating point divider of the Pentium processor. A bug was discovered in the look up table, that was believed to be redundant and the results would fix themselves eventually while applying the SRT algorithm. However, some researchers found that extremely wrong results were calculated by the

divider module. Intel had to replace all the malfunctioning processors. This barely noticeable defect cost the company about $475,000,000.

In [**42**], the inventors pointed to the fact that increasing the redundancy leads to a much simpler quotient digit look up table. However, to further simplify the look up table, they introduced an over redundant digit set, and used pre-scaling of the operands. So, they did not need to implement the full look up table. However, we tried this approach and found that the pre-scaling circuit adds an additional delay to the critical path of the design, which is larger than the optimization acquired from the look up table optimization proposed.

In the same year, in [**33**], the authors also pointed to the same issue; that in deep sub-micron technologies, it is attractive to reconsider using maximally redundant digit set, leading to a faster and simpler look up table. We used the architecture presented there as a basis for our design then we optimized it as it will be shown in the following sections. It is worth mentioning that our design has much higher performance than this one, which proposes a critical path of 2 ns on 65 nm technology, while ours proposes only 0.7 ns, but this design is targeted towards energy efficiency so it consumes about one third of the power consumed in our design.

4.3.6 Overall Architecture

Our architecture, represented in **Figure 4.13**, includes the following parts:

1. **Inputs and outputs:** the circuit has 3 inputs; the two operands and a reset signal to initiate the division operation. It has only the quotient as an output.

2. **Control circuit:** as a typical implementation of a sequential circuit; it is divided to a data path and a control circuit to control the data processing blocks. It is simplified into a simple 5-bit counter, that is reset to 0 and stalls when it reaches 27 until the next reset.

3. **Multiples of the divisor generation circuit:** the circuits that generate the multiples of the divisor -3d, -2d, -d, d, 2d and 3d once to be used by the circuit in all iterations.

4. **55-bits Carry Save Adder:** to calculate the partial remainders in the carry-save redundant format.

5. **6-bit adder:** to calculate the range of partial remainder.

6. **Look-up table:** to select the following quotient digit in each iteration.

7. **Registers:** to save Partial remainder sum and carry, next quotient digit and the intermediate values of Q+ and Q-.

8. **55-bit subtractor:** to calculate the actual value of the quotient in a non-redundant formant.

9. **1-bit left shifter:** to normalize the quotient in case the dividend is greater than the divisor

10. **Sign unit:** to calculate the sign of the quotient and it is reduced to a single XOR gate.

11. **Exponent unit:** to calculate the correct exponent of the quotient; it subtracts, the exponents of both the dividend and the divisor, it, also, corrects this value according to whether or not normalization has occurred.

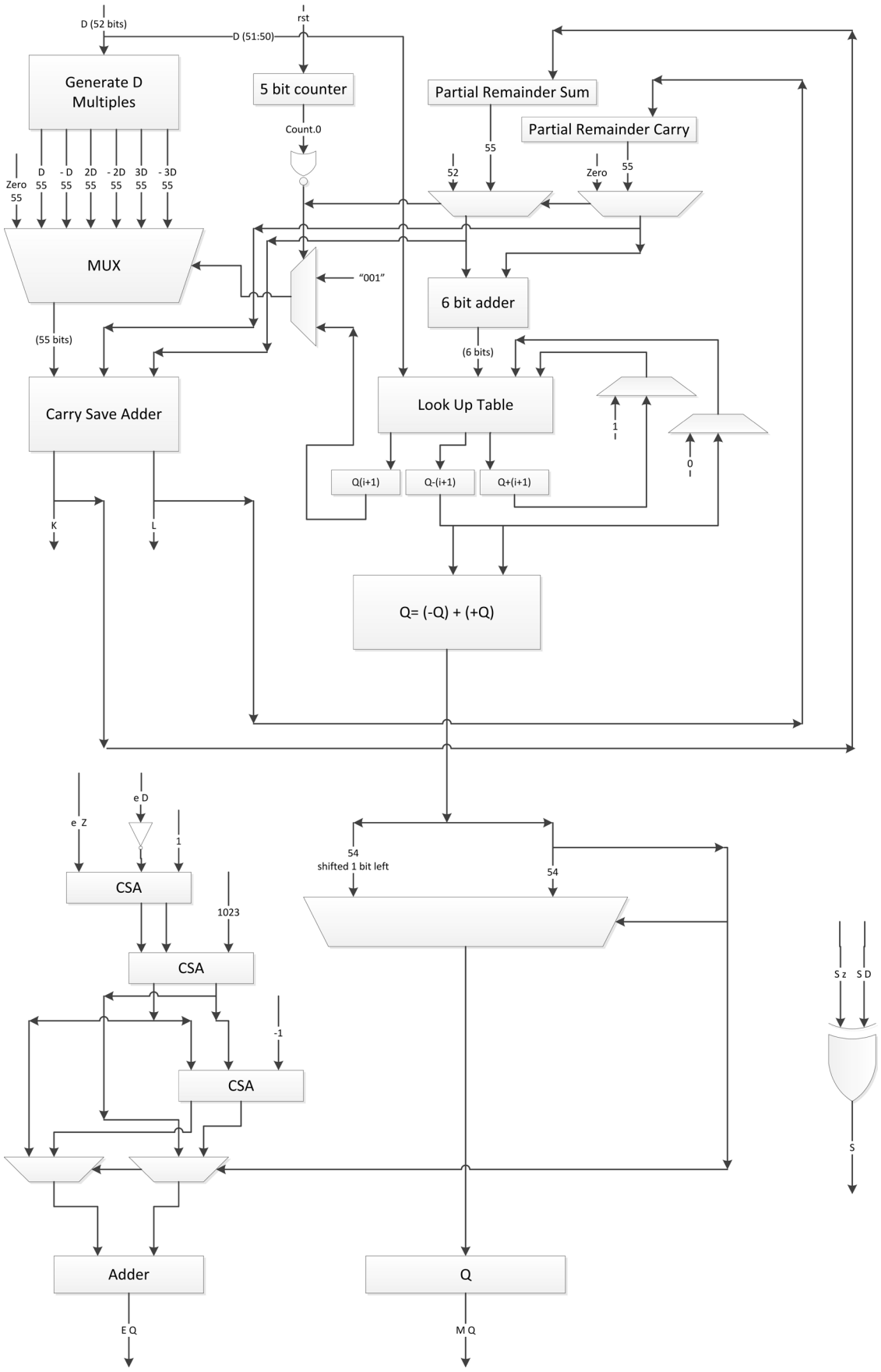


Figure 4.13: Our Proposed SRT Radix-4 FP Divider Architecture

4.3.7 Implementation Details

Control Logic / Counter

The counter is a simple binary counter; it uses the reset signal to initialize at 0. Using this condition, the circuit chooses to use the initial input values instead of the partial remainder. Also it initializes the quotient digit with +1, Q+ with 1 and Q- with zero. As the circuit iterates, the counter operates as a normal binary counter. Also Q+ and Q- get shifted and updated in each cycle.

When the counter reaches 27, the final values of the redundant quotient Q+ and Q- are available and they have to be maintained so that the final quotient can be calculated. So the counter stalls at this value and controls the registers to maintain their values until another reset signal arrives.

Multiples of the divisor generation circuit

This circuit takes the mantissa of the divisor as an input, concatenates the hidden bit to it and generates the six required multiples:

- d : its generation is straight forward.
- 2d : its generation is simply a 1-bit left shift operation.
- 3d : requires the addition of the two previous multiples so it needs a 55-bits along its path.

In fact, this adder does not add any additional delay over the architecture that uses only (-2d : 2d) as the negation process includes an adder along its path also as shown in the circuit diagram below.

- · -d, -2d : negation of the corresponding positive multiples.
- -3d is the most critical value to be calculated, as when it is calculated by negating the 3d multiple, we can easily see that it will have two adders along its path, which have large delay. So we proposed a new idea for calculating it.

Instead of modeling -3d as the two's complement of 3d (not(3d) +1) as in **Figure 4.14**, we proposed a new idea to model it as (-d) + (-2d). Yet, each of these two values is calculated using and adder. So, adding them doesn't solve the problem. The solution is to model 3d as (not(d)+1) + (not(2d)+1) or not(d) + not(2d) + 2. These three numbers can be added using a carry-save adder which is much faster, and its results are added using a carry-look ahead adder, as in **Figure 4.15**. So the advantage of

the architecture that uses -2d:2d multiples over this architecture, in terms of delay, is only the delay of the carry save adder.

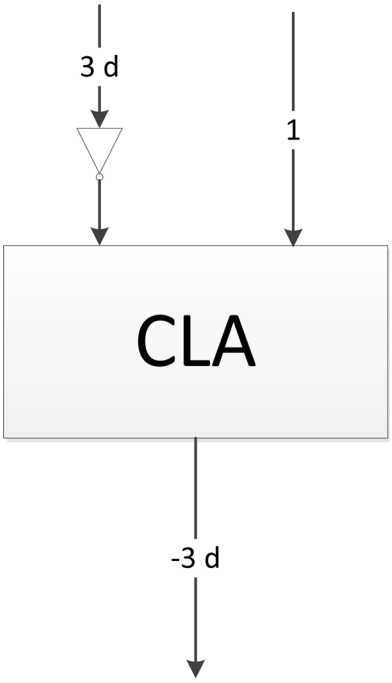


Figure 4.14: Conventional (-3d) calculation

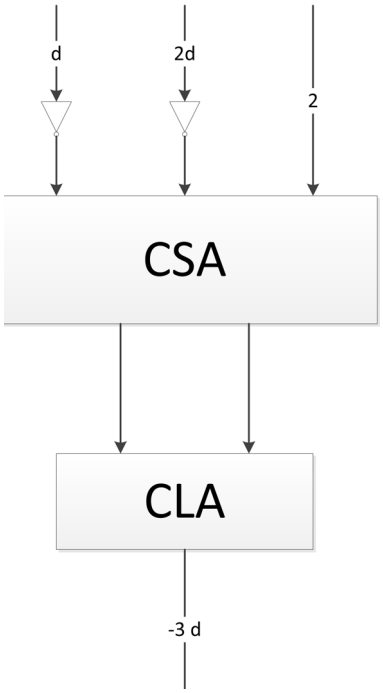


Figure 4.15 : Proposed (-3d) calculation

Look-Up Table

The look-up Table is a basic ROM design for the SRT look-up table. However, we made it faster as it updates the values of Q+ and Q- in parallel with selecting the next quotient digit, and using the same hardware that it would use if the values were updated based on this. To understand our contribution in this area, you can check **tables 4.3 and 4.4**. In the traditional approach, Q+ and Q- were updated based on the value of Q extracted form the look-up table.

Partial Remainder Range	Q

Table 4.3: Traditional look-up table

Partial Range Remainder	Q	Q+	Q-

Table 4.4: Our proposed look-up table

In fact, the table only contains in each entry only two additional bits for each of Q+ and Q-, which are used for updating their values.

Normalization

According to the IEEE STD 754 for floating point arithmetic [**14**], the mantissa of normalized floating point numbers lie within the range]-2,-1] U [1,2[. Therefore, the mantissa of the quotient calculated in the integer divider block shown in **Figure 4.13.** lies with the range]-2, -0.5] U [0.5, 2[.

According to the previous conclusion, there are only two cases that can occur from the point of view of normalization:

- The result is in the range]-2, -1] U [1, 2[so no normalization is needed.
- The result is in the range]-1, -0.5] U [0.5, 1[so a one-bit left-shift is needed.

It is noticeable that this is not the case in the multiplication or addition operations, where normalization is harder to predict. Due to this simplicity, we calculate the two values of exponent for both cases and choose one, according to the MSB of the mantissa. This is shown in **Figure 4.13.**

Sign

The sign of the quotient is just an XOR function between the signs of the operands. Again, this is much simpler than the sign calculation in the Fused-Multiply Add Module.

4.3.8 Contributions

- A new fast way for calculating the (-3d) multiple.
- Modified look up table for delay optimization.
- Fast exponent Calculation/Correction unit.

4.3.9 Results

The design has been simulated and verified on a very large set of test cases, including, but not restricted to, the special cases when one or two of the operands are 0 or infinity. It has been synthesized on TSMC low power 65 nm technology using Synopsys Design Compiler, in order to be used in the final chip placement and routing. Here are the synthesis results in **Table 4.5.**

Critical Path Length	0.78 ns
Maximum Frequency	1.28 GHz
Latency	30 cycles
Cell Count	4307 cells
Total Cell Area	0.01 mm^2
Total Dynamic Power	13.8099 mW
Number of Nets	3782 nets

Table 4.5: Synthesis results of the divider

As mentioned earlier, our results are in the same range with the results in [**42**] as their design targets energy efficiency and ours targets high performance.

4.4 Control Unit

4.4.1 Background

The control unit coordinates the components of a computer system. It fetches the code of all of the instructions in the program. It directs the operation of the other units by providing timing and control signals. All computer resources are managed by the CU. It directs the flow of data between the Central Processing Unit (CPU) and other devices. The control unit receives external instructions or commands which are converted to a sequence of control signals that the control unit applies to the data path to implement a sequence of register-transfer level operations. The control unit implements the instruction set of the CPU [**43**]. It performs the tasks of fetching, decoding, managing execution and then storing results. It may manage the translation of instructions (not data) to micro-instructions and manage the scheduling of micro-instruction between various execution units.

NISC control unit is the simplest compared to other architectures, where it just consists of the program memory(micro-instruction memory) which contains all the micro-instructions needed to control the data-path. NISC compiler will generate control words for each type of Data-path.

Therefore, NISC is reconfigurable; which allows for very fine tuning to any application.

Hardwired Control Unit

Hardwired control units are implemented through use of sequential logic units, featuring a finite number of gates that can generate specific results based on the instructions that were used to invoke those responses. Hardwired control units are generally faster than micro programmed designs. Their design uses a fixed architecture . It requires changes in the wiring if the instruction set is modified or changed. This architecture is preferred in reduced instruction set computers (RISC) as they use a simpler instruction set.A controller that uses this approach can operate at high speed. However, it has little flexibility, and the complexity of instruction set, it can implement, is limited. The hardwired approach has become less popular as computers have evolved. Control units for CPUs were ad-hoc logic, and they were difficult to design.

Micro Programmed Control Unit

Micro programs are organized as a sequence of micro-instructions, and are stored in a special control memory.

The algorithm for the micro programmed control unit is usually specified by flowchart description [**59**]. The main advantage of the micro programmed control unit is the simplicity of its structure. Outputs of the controller are organized in microinstructions and they can be easily replaced. Execution of a machine language instruction could then be caused by fetching the proper sequence of microinstructions from the control memory and sending them out to the data path section of the computer.

4.4.2 Approach

An ordinary controller consists of a state register (SR), Next state logic and Output logic. SR stores the present state of the processor which is equal to the present state of the FSM model describing the operation of the Controller. The Next-state logic (fetch unit) computes the next state to be loaded into the SR, while the Output logic generates the Control signals and the Control outputs. The Next-state and Output logic are combinatorial circuits that are implemented with gates. A block diagram of such a basic control is shown in **Figure 4.16**

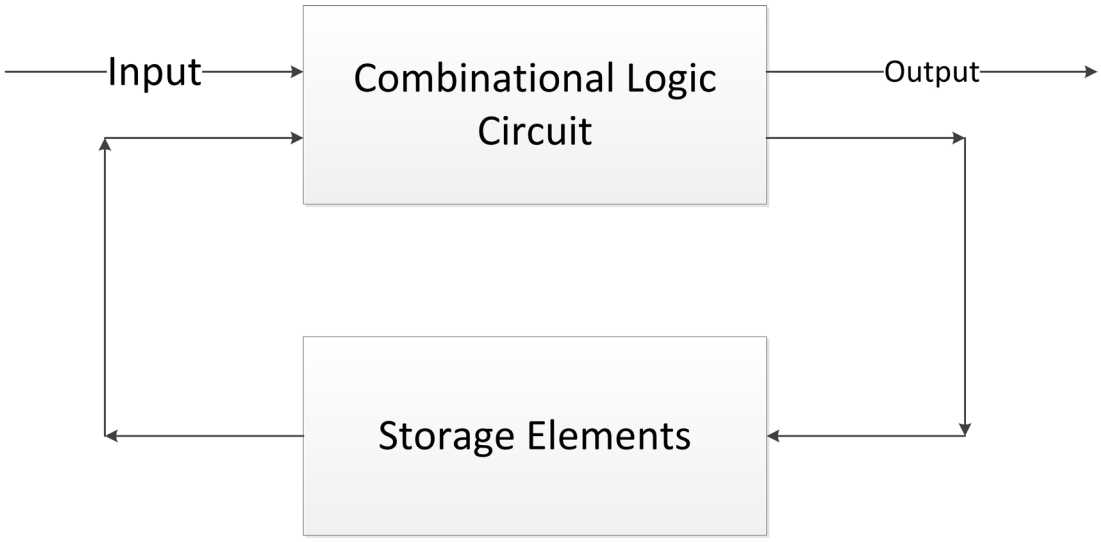


Figure 4.16: Basic FSM Controller

In order to make a hardware implementation of such controller, state tables are used, such as **Table 4.5.**

Current State	Input Variables	Next State	Output Variables

Table 4.5.: State Table

Our approach to implement the reconfigurable micro-programed control unit, was to find a way to implement the state table as a programmable hardware module.

The straightforward approach was to implement it using a content addressable memory, CAM.

CAM compares input data against the stored data in parallel and outputs the address of the matched data. CAM is an outgrowth of the RAM technology but the searching function is quite opposite for both of them. RAM needs an address to output the required data. Whereas to CAM, contents are provided and matching address is received at the output.

However, such an implementation requires a powerful CAM IP or a custom design for such one, which we could not afford during the project. In order to overcome that, we present two other alternatives:

1. In [**45**][**46**], each of the authors presented some approaches about how to design an SRAM-based CAM; we used these ideas to design one version of our control unit. SRAM-based CAMs achieves advantages like lower power consumption, lower cost, and comparable search performance. the design consists of 2 SRAMs; one of them is 43-bit wide used to find the corresponding next state microinstruction based on the current state and the inputs, the other SRAM is 64-bit wide contains the micro instruction of the next state. The two memories are filled by the compiler depending on the problem being solved. However, it had some serious performance complications. So, we went on to the other approach.

2. In order to prove the concept on the targeted performance, we emulated the function of the ordinary CAM using a content addressable register file.

4.4.3 Design

The register file consists of 128 registers as the one in **Figure 4.17.** The outputs of such registers drive an OR gate. Since, only, one comparator asserts a match signal, the OR gate will only have one correct input and zeros at all the other 127 inputs.

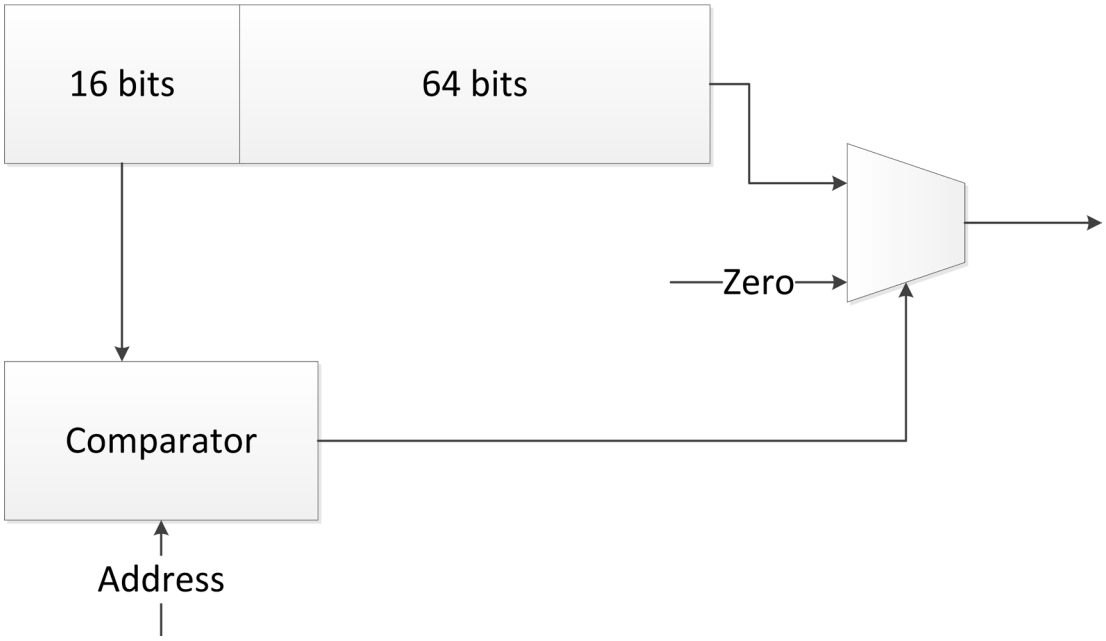


Figure 4.17: One cell of the content addressable register file

The 80-bit word consists of 16 bits input signal and the remaining 64-bits are the micro instruction which consists of 6-bits for the next state and the remaining are the control signals. Each set of control signals issued causes one basic operation (micro-operation) to occur within the data path. Microinstructions are supposed to be generated by the Compiler. In instruction-set-based processors, the controller is very complex mostly because of the instruction decoder. Changing the controller, and hence changing the architecture, is very difficult in these processors. NISC does not have any instructions or instruction decoder. Instead, the NISC compiler generates the control words for each clock cycle. As an input to the NISC compiler, the NISC architecture model is represented by a simple controller plus the netlists of data path components. Therefore, the architectural modifications are much easier in NISC than in traditional processors.

The microinstruction being saved is calculated using Finite State Machine (FSM) where each

microinstruction depends on the inputs and the current state to find the next state control signals.

Control unit has 2 main blocks:

• *Microinstruction memory,* Containing all the states being considered within the application and this memory is filled by the compiler just like the conventional state table where the next state depends on the current state and the next state logic, as shown in **Figure 4.18.**

• *Fetch Unit,* is the next state logic or micro-sequencer Which generates a sequence of bits which is used to fetch the next microinstruction taking into account the status signals originating from the data part, as shown in **Figure 4.19.**

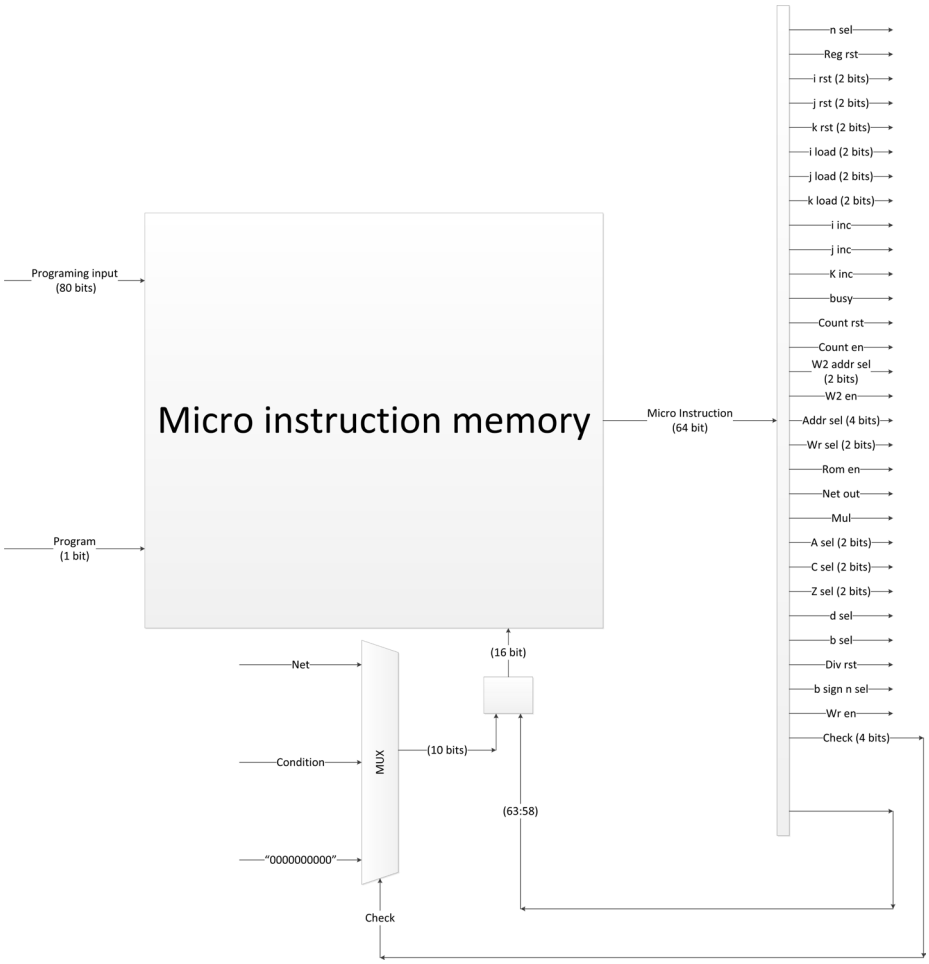


Figure 4.18.: Microinstruction Memory

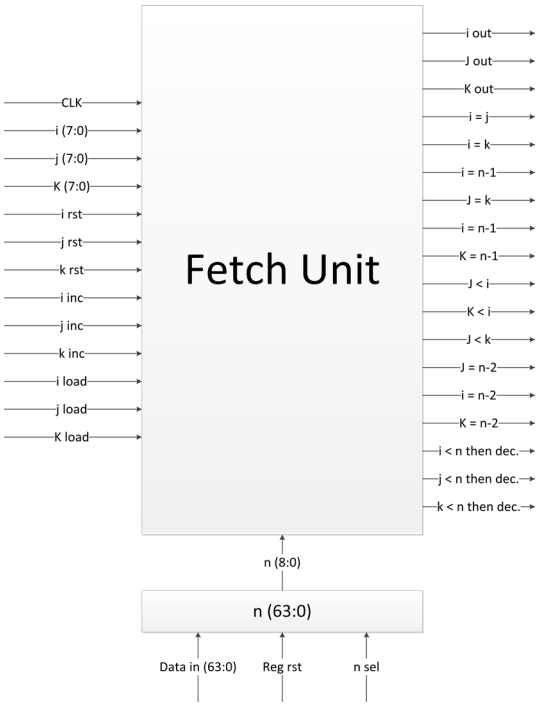


Figure 4.19.: Fetch Unit

4.4.4 Contribution

The main advantages of the proposed Control Unit

• Structured and re-configurable: The design is extremely flexible, as any change requires only a modification in the contents of the microinstruction memory. The control part hardware structure is fixed as opposed to a random logic design.

• Testing sequences can be easily incorporated: Testing a system both for faults in implementation as well as for faults developing over time due to component failure require considerable effort and cost. In micro-programmed systems as all the components are being controlled from the control memory, it is relatively easy to develop test sequences that activate various components in a controlled manner to test the complete system components functioning.

4.5 NoC Router

4.5.1 Introduction

In Multi-Processor Systems-On-Chip (MPSoCs), consisting of complex components, communication became a very important factor. Movement of data is the factor limiting cost, performance, size and power consumption in these systems.

Networks-On-Chip (NoCs) have been proposed as a promising replacement to eliminate many of the overheads of buses and MPSoCs connected by means of general-purpose communication architectures.

The Intercommunication requirements of MPSoCs, made of many cores, will not be practical using a single bus shared between them or a hierarchy of buses due to bad scalability with the size of the system, also, their shared bandwidth between all cores attached and the requirements energy efficiency of final chip. So as to have a good design for scalability and complexity, Networks-On-Chip (NoCs) is proposed as a better replacement to remove buses overheads, and as a result, MPSoCs will be linked as general-purpose communication design and architecture.[**47**]

The NoC-based SoC design utilizes two major concepts, different from those of the usual bus-based SoC architecture. First, it is a packet transaction rather than a circuit transaction, and, secondly, it is a distributed network structure, rather than a conventional globally shared bus or a centralized matrix. .[**48**]

The Network on Chips solution has many advantages including:

1. High Bandwidth
2. Low latency communication
3. Minimum cost and energy
4. More scalability
5. Efficient link utilization

Comparison between NoC-Based and Bus-Based Design Methodologies **[48]**

	NoC Design	Bus Design
Bandwidth and speed	+Non-blocked switching to make multiple concurrent transactions. Pipelined links with better throughput and clock speed.	- In a shared bus a transaction blocks other transactions. Every unit adds more parasitic capacitance. Performance gets lower with growth.
Resource utilization	+Packet transactions link resources in a statistic multiplexing manner.	-A single master shared bus during its transaction.
Reliability	+Short switch to switch link, more error-free signaling.	-more complex control in end to end with more penalty
Arbitration	+smaller and faster Distributed arbiters. -arbiter uses only local information.	-arbiter is bigger and slower as each master require single arbiter. +better decision with central arbitration.
Transaction energy	+minimum energy consumption due to Point-to-point connection.	-More energy consumption due to broadcast transaction.
Modularity and complexity	+less design time as switch/link design is re-instantiated.	-not reusable as bus design is specific.
Scalability	+bandwidth scales with network size.	-less scalable and design becomes larger and slower
Latency	-Packet latency due to Internal network contention. - Cumulative latency due to Repeated arbitration on each switch. - Additional latency due to Packetizing, synchronizing, and interfacing.	+wire speed increase with Bus latency once a master has a grant from an arbiter.
Overheads	-Additional area and power with routers/switches and buffers.	+Less area and fewer buffers are used.

Table 4.6 : Comparison between NoC-based and Bus-based Methodologies

4.5.2 OSI Seven-Layer NoC Model

The correspondence of the building blocks of NoC to the OSI seven layer model is presented as shown in **Figure 4.20.**

In the Hardware layers:

• Physical layer: Responsible of signaling(BW, frequency,…)

• Data link layer: Point-to-point communication (Buffer size, flow control)

• Network layer: Switching(Topology, Switch scheduler, routing)

• Transport layer: End-to-end communication: packet ordering [Networks,3]

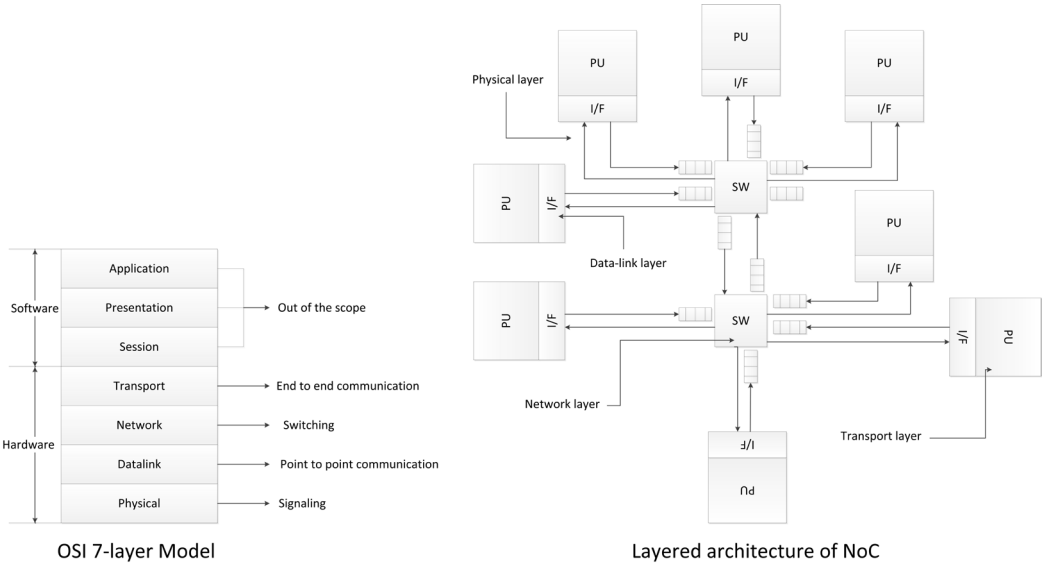


Figure 4.20.: OSI seven layer model

The Noc is defined by 2 factors:

1. *Topology*
2. *Protocol*

4.5.3 NoC Topologies

Network topology refers to the static arrangement of nodes in an interconnection network, i.e. the road map over which packets travel.

Selecting the network topology is the first step in designing a network because the routing strategy and flow-control method depend heavily on the topology. [**49**]

The right topology is chosen by exploiting the available packet technology to maximize the performance: Bandwidth and latency.

To Maximize the bandwidth we should saturate the BW across the mid point of the system which is called the bisection bandwidth.

Measures of a good topology:

1. Load balance between the network and processing unit among tiles
2. Topology regularity
3. Spatial locality
4. Base measures of a network's quality: latency, throughput and path diversity **[48]**[**49**].

In **Figures 4.21 to 4.25**, we present the different network topologies.

Topologies examples [**49**]

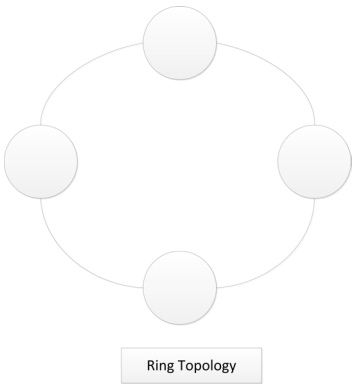


Figure 4.5.2: Ring Topology

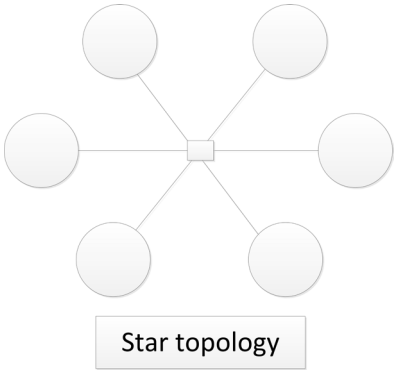


Figure 4.5.3: Star Topology

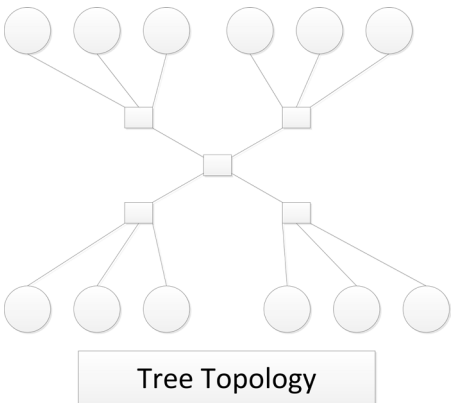


Figure 4.5.4 :Tree Topology

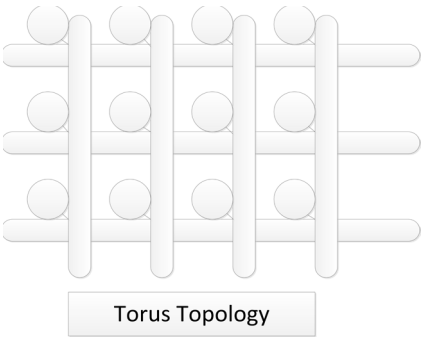


Figure 4.5.5: Torus Topology

Why mesh topology?

We chose to implement the mesh topology for several reasons, among which, that it can withstand high traffic that the data can be transmitted simultaneously from different devices. Moreover, there are always alternative paths between nodes if one path fails to transmit the data then data transfer is never affected because in the mesh topology all of the network nodes are interconnected with each other, and the node does not only send its signals but also relays data between other nodes.

One of the main reasons we decided to use the mesh topology is that its expansion and modification can be done without disrupting other nodes.

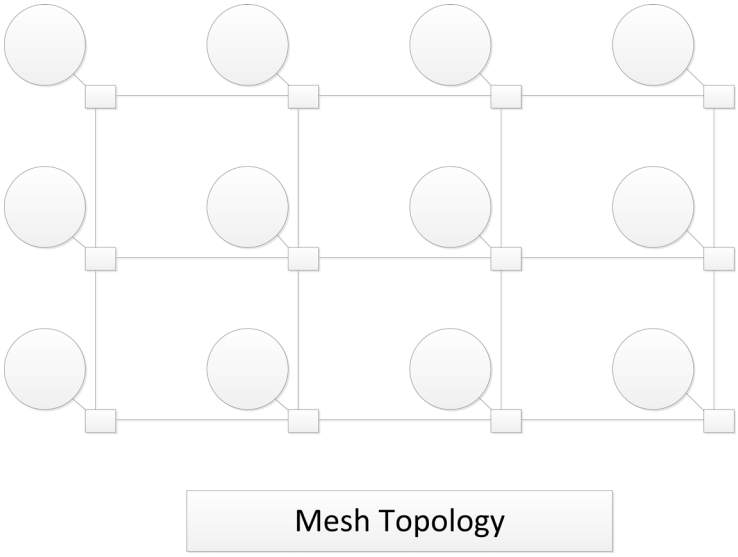


Figure 4.25: Mesh Topology

4.5.4 NoC Protocol

The protocol concerns the strategy of moving data through the NoC. The protocol is mainly described by:

• **Routing,** which determines the path selected by a packet

• **Switching,** which determines how packets flow (are forwarded) through the selected path. in our case: wormhole

• **Flow-control,** which is the mechanism that determines the packet movement along the network path, to assure correct operation of the network

A. Routing

In networks on chip (NoC), routing is done to connect different components together in the most efficient manner. For example, selecting a short path may be important in one situation, while creating an appropriate traffic load balancing may be of greater importance in other situation. [**51**]

The routing determines the path taken by the packet. Some networks take single route from source to destination. Others, such as torus topology -networks, have many possible paths.

A good Routing algorithm is critical for several reasons:

1. It balances load across network channels even in the case of non-uniform traffic
2. It is the one that keeps path length as short as possible, reducing the number of hops
3. The ability to work in the presence of faults in the networks

These algorithms are often classified in two groups.

• Deterministic algorithms

• Adaptive algorithms

In our case, we used a simple algorithm called XY-routing algorithm which is one kind of distributed deterministic routing algorithms. XY routing is one of the types of Dimension order routing (DOR), which is more suitable for networks using mesh or torus topology. XY routing algorithm routes packets first in x-direction (or horizontal direction) to the correct column and then in y-direction (or vertical direction) to the destination. [**51**]

In XY routing, the addresses of the routers are their xy- coordinates. One of the advantages of XY routing is that it never runs into deadlock or livelock.

B. Switching

There are several methods for switching:

1- **Store and Forward:** The whole incoming packet is stored in a buffer for switching and forwarding. This method requires a buffer with large capacity

2- **Wormhole:** The incoming packet is forwarded right after the packet header is identified and the complete packet follows the header without any discontinuity. In this method, the path

used by packet, through the switch, is blocked against the access of any other packet.

And this is the one used in our design.

3- **Virtual cut through:**

The path is determined by a wormhole routing, but in the case of an occupied path, the next hop packet tail is stored in a local buffer waiting for clearing the path. In this method, the buffer size is smaller than the size mentioned in the first method mentioned before. But, if the packet size is large, many local buffers would be occupied resulting in delaying the switch throughput.

Packet Format

The packet used in this network is the simple type which consists only of Header bits and data bits:. The packet header contains the address of the destination PE, Function ID that designates the operations to be performed on the packet, and additional control information. The Function ID enables flexibility and context switching to support simultaneous operation so that centralized control is not required for most of the data processing. [**48**]

C. Flow control

If we made a small analogy for the network on chip concept, it will be like: the topology determines the roadmap, the routing method steers the car, and the flow control controls the traffic lights, determining when a car can advance over the next stretch of road (channels) or when it must pull off into a parking lot (buffer) to allow other cars to pass.[**49**]

Flow control determines how the network's resources – channel bandwidth, buffer capacity and control state- are allocated to messages, transmitted through the network, and how these resources are shared between all the messages inside the network.

A good flow control protocol allocates these resources in an efficient way so that the latency experienced by the messages at low loads is lowered by imposing high overhead in resource allocation, and increases the network throughout by enabling effective sharing of buffers and links across messages.

On the other hand, a poor flow-control protocol wastes bandwidth by leaving resources idle and

doing nonproductive work with other resources.

a. Resources and Allocation Units

To transmit data through a network, the messages must be of allocated resources: channel bandwidth, buffer capacity, and control state. **Figure 4.26.** shows these resources in a single node of a network.

When a packet arrives at a node, it must first be allocated some control state and it is temporarily held in buffer while waiting for the channel bandwidth. All flow control methods include allocation of control state and channel bandwidth. However, some methods do not allocate buffers, i.e. bufferless.

To improve the efficiency of the resource allocation, the message is divided into packets for the allocation of control state. Then, packets are divided into flow control digits (flits) which are used for the allocation of channel bandwidth and buffer capacity. Flits are divided into physical transfer digits (phits). The phit is the unit of information transferred across a channel in one clock cycle.

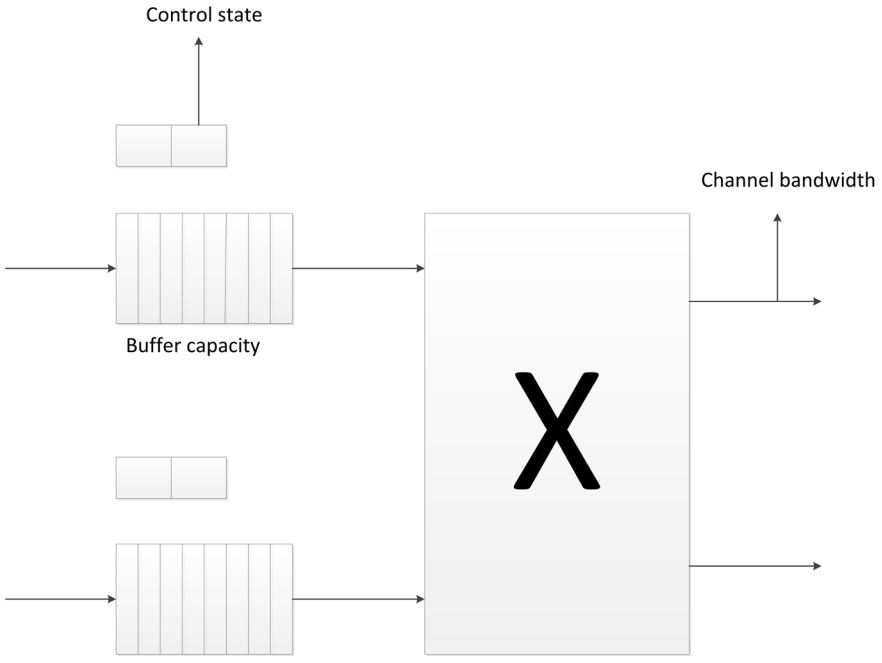


Figure 4.26.: Network resources

Why are packets divided into flits?

There are several policies that suffer from conflicting constraints when choosing the packet size, because we want the packet to be large to pay off the overhead of routing and sequencing. At the same

time, we would like to make packets very small to assure efficient resource allocation and minimize blocking latency; this is done through the flits. We can achieve low overhead by making packets large enough and, also, achieve efficient resource utilization by making flits very small.

b. Bufferless Flow Control

This form is considered one of the simplest forms of flow control. This is due to its ability to use no buffering besides its ability to allocate channel state and bandwidth to competing packets. The bufferless flow-control method must perform an arbitration operation to decide which packet gets the grant to be transmitted. After arbitration, the packet chosen is transmitted through the channel. The disadvantage of the bufferless flow control is that the losing packets are either dropped or misrouted.

Circuit switching is a type of bufferless flow control that allocates channels to form a circuit from source node to destination node. Then, it sends one more packet till there are no other packets to be sent, then the circuit is deallocated.

Circuit switching and dropping flow control are different in that if the request flit is blocked, it is held in place instead of dropping it.

c. Buffered Flow Control

Adding a buffer prevents the waste of the channel bandwidth caused by dropping or misrouting packets or the idle time in circuit switching. Therefore, we can reach 100% channel utilization with buffered flow control which is much more efficient than the bufferless flow control.

1. Packet-Buffer Flow Control

Buffers and channel bandwidth can be allocated to either flits or packets. There are two types of packet-buffer flow control: store-and-forward and cut-through.

In case of store-and-forward flow control, each node waits until a packet is completely received and then moves the packet to the next node.

The major drawback of store-and-forward flow control is that it has very high serialization latency at each hop.

Cut-through flow control overcomes the latency of store-and-forward flow control by forward-

ing a packet as soon as the header is received and resources are acquired as it doesn't have to wait for the entire packet to be received. Cut-through flow control allocates buffers and channel bandwidth in units of packets.

The packet-based method has two disadvantages. First, when allocating buffers in units of packets, it makes very inefficient use of buffer storage. Second, by allocating channels in units of packets, the latency is increased. For example, when a high-priority packet collides with a low-priority packet, it has to wait for the whole low-priority packet to pass through the channel before it can be transmitted.

2. Flit-Buffer Flow Control

1. Wormhole Flow Control

Wormhole flow control is similar to cut-through, but with channel and buffers allocated to flits not packets. When the head flit of a packet arrives at a node, it acquires three resources before it can be forwarded to the next node. The resources are: channel state, one flit buffer, and one flit of channel bandwidth.

Wormhole flow control efficiently uses the buffer space if compared to the cut-through flow control as only a small number of flit buffers are required in a virtual channel in contrast to the cut-through flow control. On the other side, this reduces the throughput.

2. Virtual-Channel Flow Control

Virtual-channel flow control overcomes the blocking problems of wormhole flow control because it allows other packets to use the channel bandwidth. Otherwise, when packets are blocked, it would be left idle. When a head flit arrives at a node it must allocate a virtual channel, flit buffer, and channel bandwidth to advance.

Virtual channels allow packets to pass blocked packets, making use of otherwise idle channel bandwidth, thus improving the network performance.

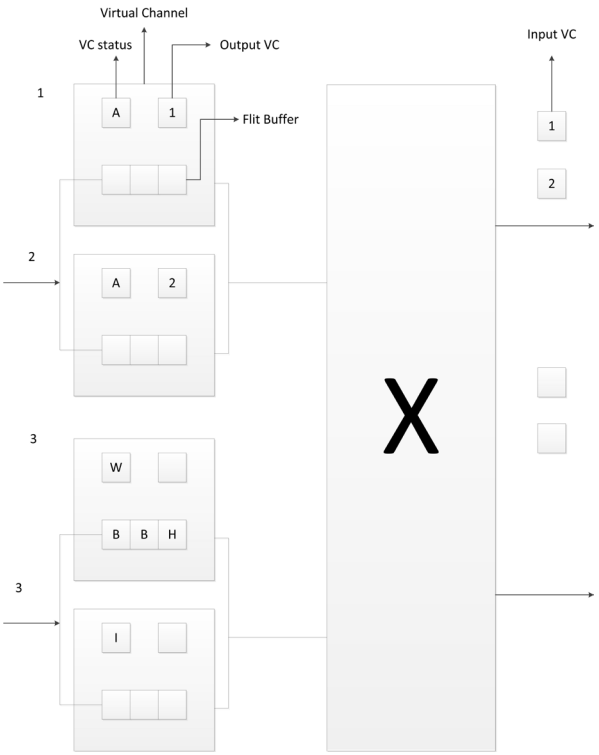


Figure 4.27.: Virtual channel

The **Figure 4.27**. shows the block diagram of a virtual channel router showing the state associated with each virtual channel. Each input virtual channel includes the channel status, the output virtual channel allocated to the current packet, and a flit buffer. Each output virtual channel identifies the input virtual channel to which it is allocated.

Input virtual channel 1 is in the active state forwarding a packet to output virtual channel 1. Similarly, input virtual channel 2 is forwarding a packet to output virtual channel 2. Input virtual channel 3, associated with the lower physical channel, is waiting to be allocated an output virtual channel. Input virtual channel 4 is idle, waiting for the arrival of a packet [**49**].

Table 4.7. summarizes the previous section [**48**].

	Links	**Buffers**	**Comments**
Circuit-Switching	Messages	Bufferless	Requires setup & acknowledgment
Store and forward	Packet	Packet	Head flit must wait for entire packet before proceeding on next link
Virtual cut-through	Packet	Packet	Head can begin next link traversal before tail arrives at current node
Wormhole	Packet	Flit	Head of line blocking reduces efficiency of link bandwidth

	Links	Buffers	Comments
Virtual channel	Flit	Flit	Can interleave flits of different packets on links

Table 4.7 : Summary for flow control

4.5.5 Router Microacrchitecture

A router is composed of registers, switches, function units, and control logic that implement the routing and flow control functions needed to buffer and transmit flits to their destinations.

The virtual channel router block diagram is showed in **Figure 4.28**.

Blocks of router are partitioned to:

1. Data path

It is the part responsible for storage and movement of packet payload. Also, it's responsible for input and output buffers and switch: an input unit for each input port, an output unit for each output port, a crossbar switch to connect between the input and output ports.

2. Control plane:

It's responsible for coordinating the movement of packets. It consists of a router to determine an output port for each packet, a virtual channel allocator to determine an output virtual channel and a switch allocator to assign a switch time slot for each flit.

When a flit of a packet arrives at an input unit of the router, it is held in input unit flit buffers until it's forwarded. Then, routing is performed to determine the output port to which the packet will be forwarded. After that, an output virtual channel is requested from the virtual channel allocator. Then, the packet flit is transmitted through the virtual channel by allocating a time slot on switch and output channel using a switch allocator. Finally, the output unit forwards the flit to the next output in the packet's path.

Buffer Organization

Buffer organization highly affects the network throughput, as it influences how efficiently packets share link bandwidth. Buffers are used to store packets or flits when they cannot be forwarded at once.

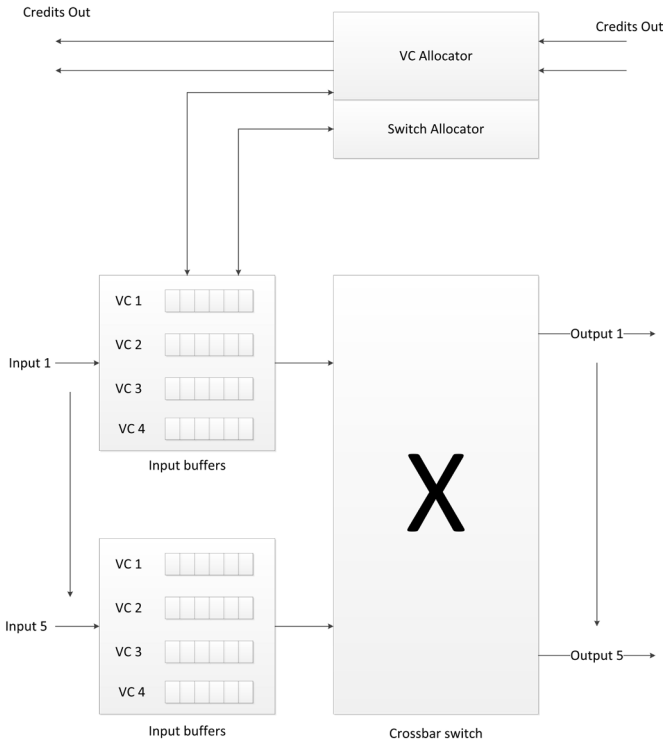


Figure 4.28.: Virtual channel router block diagram

Flits can be buffered on the input ports and/or on the output ports depending on the rate of the switch and whether it's greater than the rate of the channel or not.

Mainly, Input port buffers are distributed to: [**48**]

• Single fixed-length queue: a single queue in each input port

• Multiple fixed-length queue: multiple queues at each input port

• Multiple variable-length queue: each VC queue can be of variable-length

Switch Desgin

The crossbar switch is the main part of the router data path as it switches bits from the input to the output ports. When the packets arrive at the input port, the crossbar switch scheduler gets the destination information from the input packets. [**50**]

If every packet arrives at a different input port and wants to leave from another output portand there are no input and output conflicts, the scheduler connect the cross junctions so as to connect the packets at the input ports to their output ports. If conflicts occur, the scheduler should resolve them by the predefined algorithm. Buffers are important to store the packet data temporarily for congestion

control.

Allocators and Arbiters

An allocator matches a number of requests to a number of resources, as well as the arbiter.

The allocator/arbiter that delivers high matching probability results in a higher network throughput. Also, allocators and arbiters must be fast so they can work under high clock frequencies. [**48**]

There are 4 types of arbiters/ allocator:

• **Round-Robin arbiter:** where the last request to be grant will have the lowest priority in the next round of arbitration.

• **Matrix arbiter:** where the least recently served request has the highest priority

• **Separable allocator:** each input port has separate request lines for each output.

• **Wavefront allocator:** arbitrates among requests for inputs and outputs simultaneously.

4.5.6 Conclusions

As working with our 2D processor array, shifting from single-core processors to multi-core systems presents new problems and challenges to be solved. One such problem is the design of the router micro-architecture to cope with ever increasing bandwidth requirements, while minimizing the area overhead and power consumption.

As we present in the next section, the results are not in an acceptable range, so we are working on another optimized design that has not been integrated yet, but we provide its early results in the next section.

4.5.7 The implemented design

The main idea of a router in a PE, as explained before, is to handle the operation of sending or receiving data from other cores “PE”. According to the design we worked on, the top module of the NOC code includes many inputs and outputs; send signals which play the role of a flag to enable sending data whether to the right or down, receive signals which act as flag to permit receiving the data, data-in & data-out signals.

When a router of a certain PE wants to send a packet it activates a send signal, whether sending

to the right or downwards. At the mean time, the router of the receiving PE activates a receive signal, whether receive from the left; i.e. in the same row or from upwards; i.e. to the next row.

The router first sends the header bits which contain the destination address, the number of elements being sent, and some control signals; start & end pointers. If the receiving router finds that the address coming in the header matches its address, then the data is transmitted afterwards, if not then it calculates the intended address of the PE with respect to its own position and it relays the header bits to the following router of the next PE.

This architecture allows high degree of flexibility regarding receiving the data, where the router can receive data from two sources at a time without any conflict.

Figure 4.29., shows the architecture of the proposed NoC.

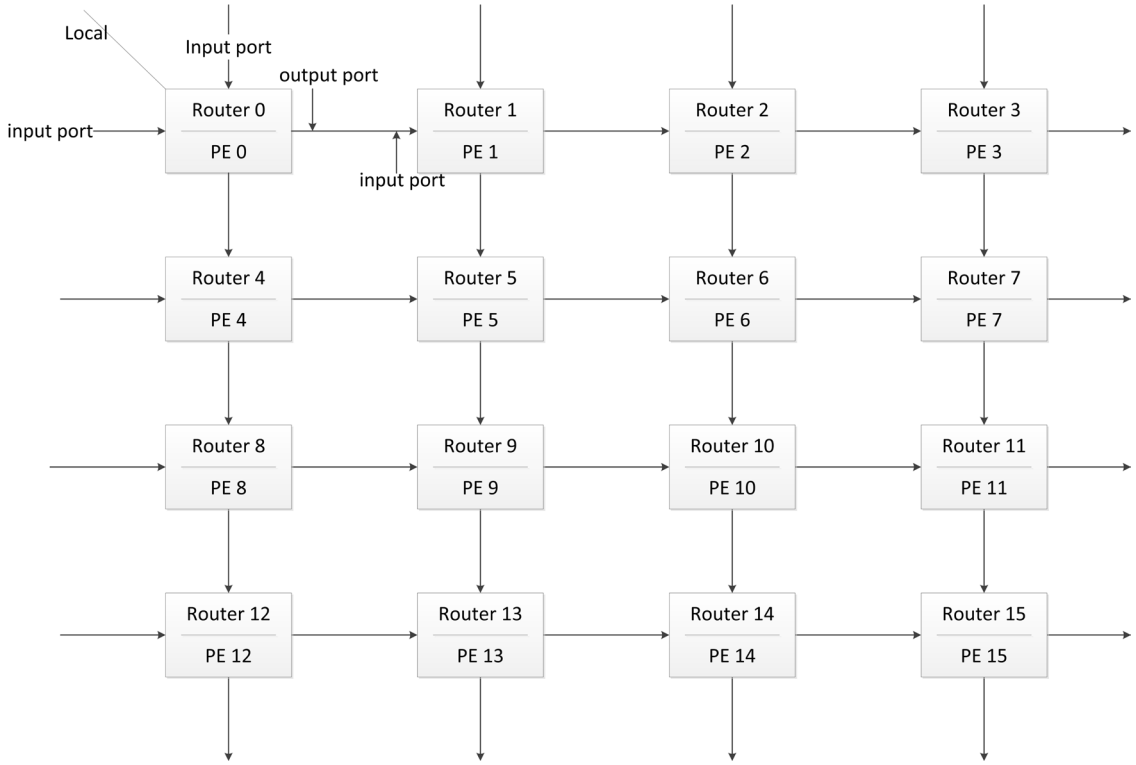


Figure 4.29: Proposed NoC Architecture

4.5.8 Our newly proposed design

Each input consists of: 64 bit data + $\log_n$(bits) going through register based FIFO with depth 8 to hold 2 packets of data, where the packet consists of data(a,b, and c) and in order to avoid congestion and deadlocks of data during heavy traffic. When the FIFO becomes half full (packet arrived) or full (two packets arrived) , it sends a), it sends a request to the round-robin arbiter. Then, it replies back with agrant for only 1 FIFO to read data from it. So, the output should be the data read by FIFO which got grant and other FIFO outputs is zero putting them on OR gate to get the output. Only one OR gate is enabled using the conditions stated below in **Figure 4.30.**

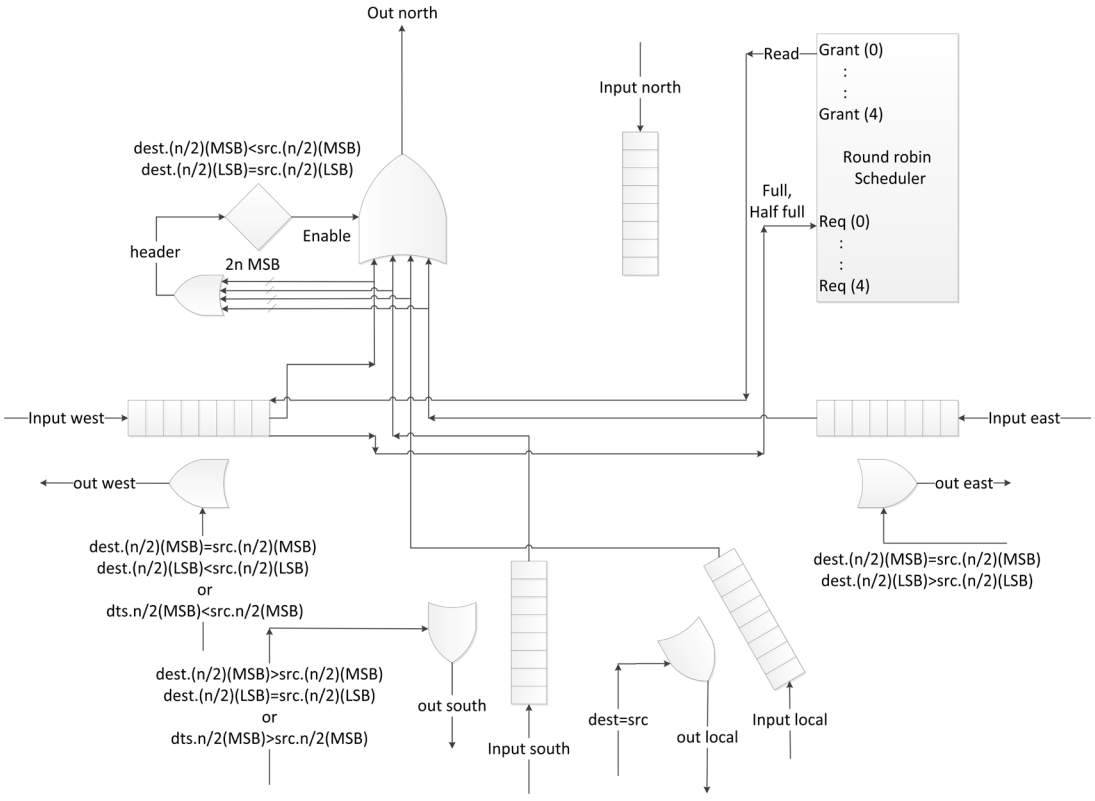


Figure 4.30: The newly proposed router architecture

Register file

The design is based on synchronous first in first out (FIFO) queue, so in our design the latency of FIFO is 8 clock cycles so as to cross the 8 registers of FIFO as shown in **Figure 4.31.** Our design implementation is a Register Based FIFO as in [**53**].

Figure 4.31: FIFO Buffer

The register file is a single port register file as shown in Figure 4.30. When wr_addr is at specificaddress, the decoder enables only one 64-bit register. Then, the data (w_data) is written into this address of register file and it can be read by selecting the mux controller for the required readingaddress (r_add).

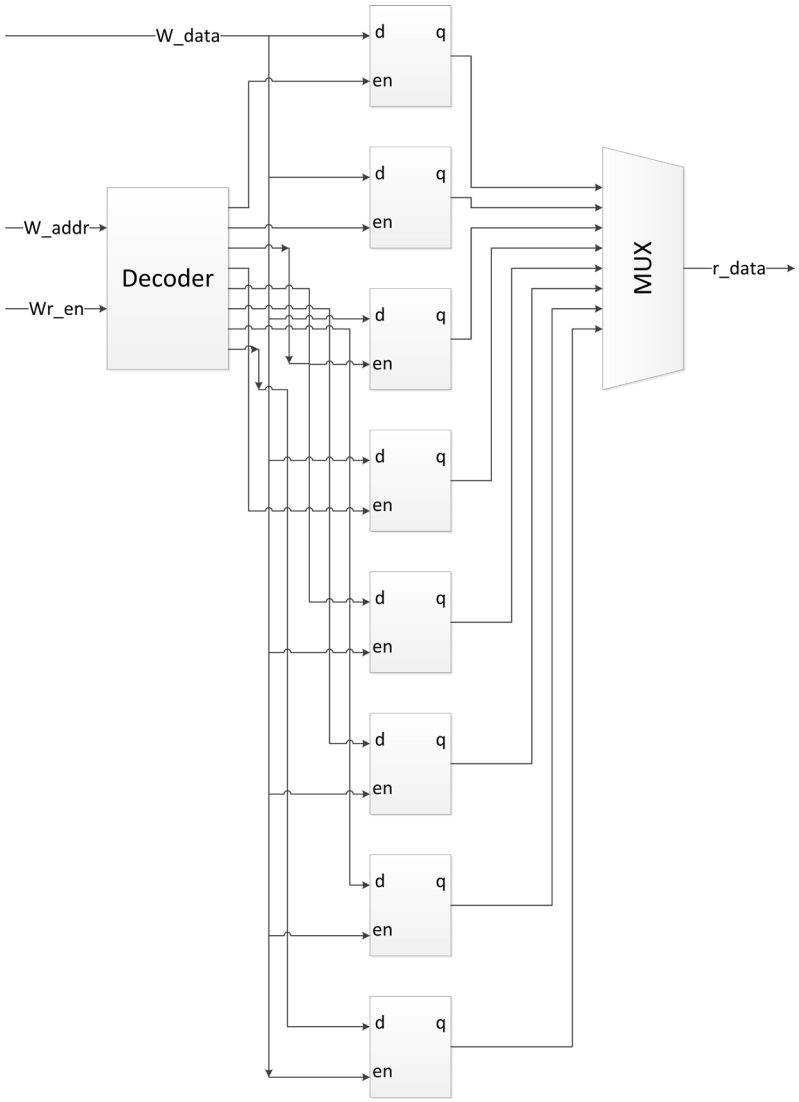


Figure 4.32: Register File

FIFO controller

It can be represented as shown in **Figure 4.33** as a ring FIFO with two pointers, one for writing address and one for reading address. When writing, its pointer counter increases with each write. When reading, its pointer also increases with each read. If both pointers are equal then FIFO is empty, if not; then the FIFO is either full or there is filled spaces inside. The full flag can be determined through making the pointers – in our case for 8 depth FIFO – by adding 1 bit for the pointer to be 4bits instead of 3 bits, when the MSBs aren't equal and the LSB of both pointers are equal then the-FIFO is full as shown in **Figure 4.33.**

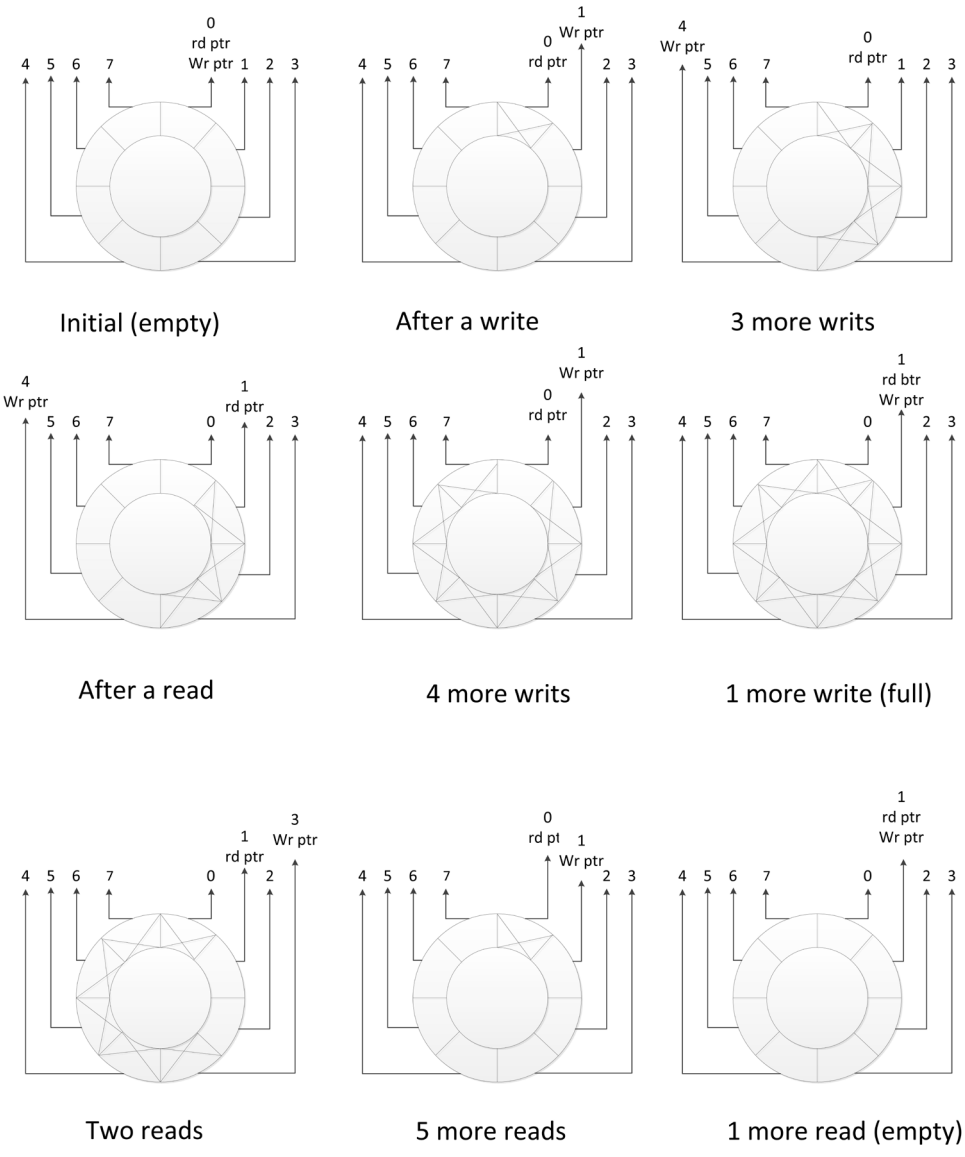


Figure 4.33 FIFO controller operation

Integrating the register file and the FIFO controller, as shown in **Figure 4.34**, results inthe requiredfunction of FIFO.

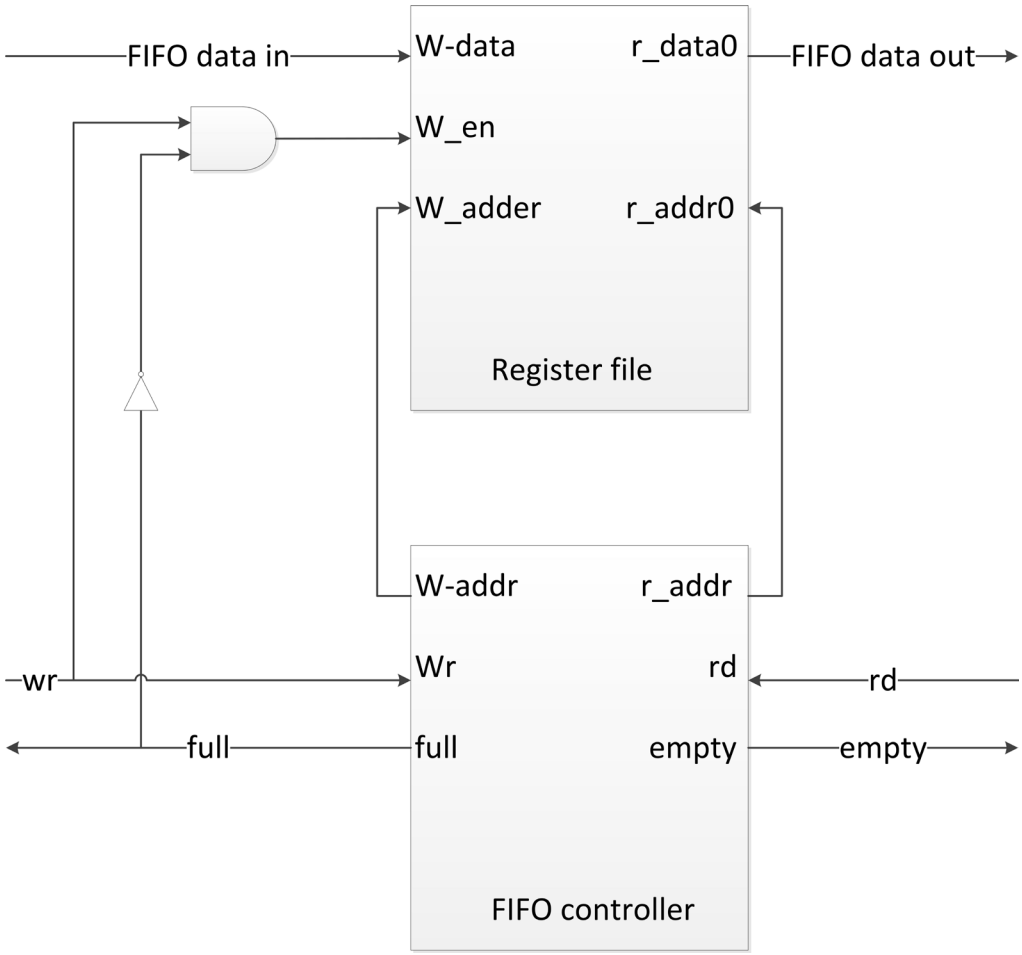


Figure 4.34 FIFO Block Diagram

Round-Robin

The idea of round-robin is to provide fairness between all destinations of the router datapath so that each direction would work – in our case – for only 1 packet to be sent which is 4 clk cycles per each grant.

As described in [**52**], when a request is sent, the fixed priority logic gives a grant only if the ring counter selected the same priority logic, i.e. request(0) is sent and priority logic 0 is enabled by ring counter, so, the grant(0) is activated by the priority logic 0, while rest of priority logic gives zero outputs as they are not enabled. While if request(1) is sent, also, it won't be given a grant(1) for the first

four clock cyclesbut it will receive a grant - when ring counter enables it - in the nextpacket to be sent as priority logic 1 gives priority for request(1). Output is OR gate ORing all grant(0), grant(1)….etc together as only 1 priority logic will be enabled.

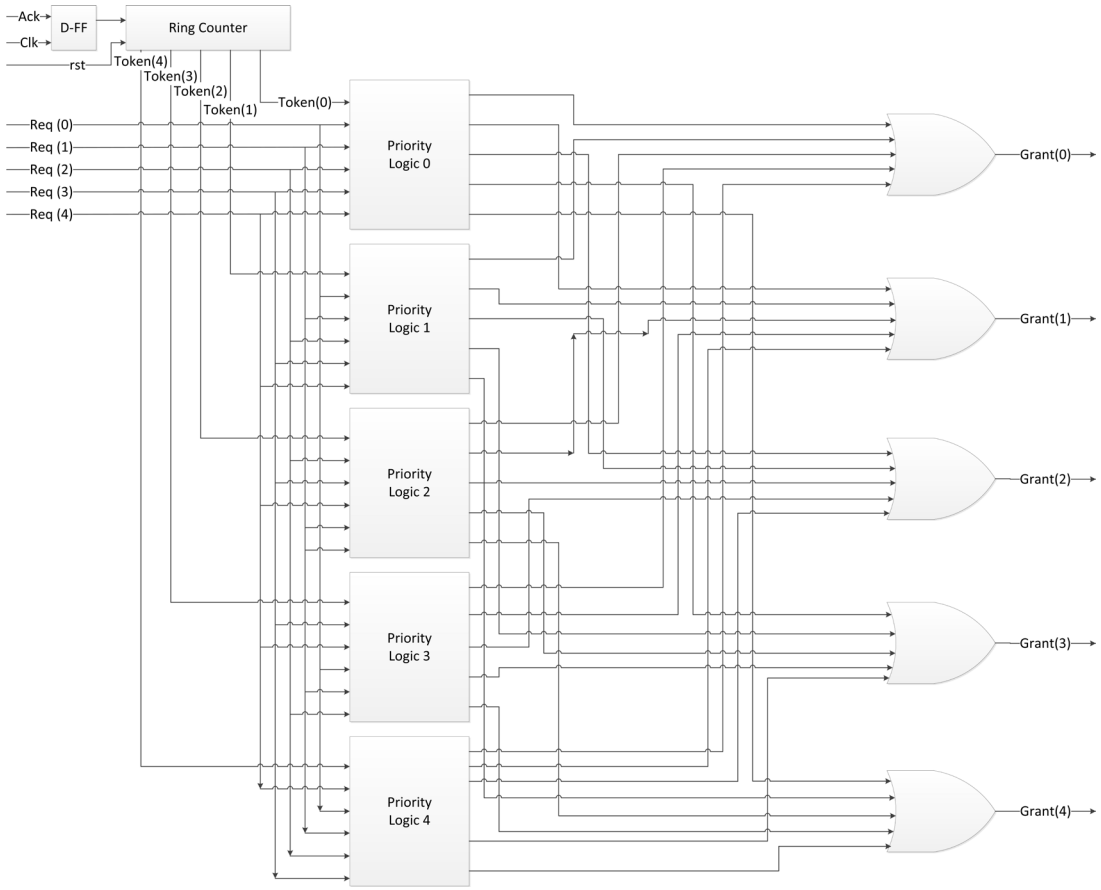


Figure: 4.35: Round Robin Arbiter

4.5.9 Results

The design has been synthesized on TSMC low power 65 nm technology using Synopsys Design Compiler, in order to be used in the final chip placement and routing.

Here are the synthesis results in **Table 4.7.**

Critical Path Length	0.96 ns
Maximum Frequency	1.04 GHz
Cell Count	24179 cells
Total Cell Area	0.085 mm^2
Total Dynamic Power	114.3113 mW
Number of Nets	22464 nets

Table 4.7 Synthesis results of the NoC router

In **Table 4.8** we provide the results of the new router we are currently working on. The block has been finished and here are its synthesis results.

Critical Path Length	0.5 ns
Maximum Frequency	2 GHz
Cell Count	8105 cells
Total Cell Area	0.04 mm^2
Total Dynamic Power	78 mW
Number of Nets	7765 nets

Table 4.8 : Synthesis results of the new NoC router

4.6 Data Memory

4.6.1 Design Constraints

Generally, memory can be described by four entities; size, number of ports, latency, and bandwidth. Latency is known to be the access delay time to a certain data element, while bandwidth can be explained as the data rate by which the memory interface with other element in a certain system "i.e. Processor, I/O Devices"[**54**]. Latency is a dependent factor of the size, it is directly proportional to it, and it is considered to be due to the following factors; Delay caused by the row and column decoders, delay due to the word line of the memory until it is activated, bit line sensing delay which is the time needed by the sense amplifier to read the content of a certain cell, output driving delay which the time needed by the data to propagate from the sense amplifier to the output ports, a fifth type of delay is the RC delay which is represented in the last three types as all these lines can be represented as a capacitances and resistances "i.e. These lines contain a lot of transistors in their paths".

The bandwidth is somewhat dependent on the specifications and requirements stated by the end user, it also can be represented as a function of the number of ports or output pads of the memory block, as the number of ports increases the bandwidth increases. It is worth to be said that latency and bandwidth play large roles in system performance and data manipulation [**55**].

4.6.2 Memory Architecture and Building Blocks

The best architecture of memory is to make the aspect ratio to be unity if possible; this will lead to the smallest delay to be reached if possible. This approach can be done using row and column address decoders, the address will be divided into two parts, row address and column address, the row address will activate a certain word line, while the column address will activate a certain bit line, this approach is considered to be the best approach from delay and timing point of view. Different memory architectures are shown in **Figure 4.36**. A better optimization could be done if we partitioned the memory into many memory blocks; this will lead to power savings since only one block will be activated on a time.

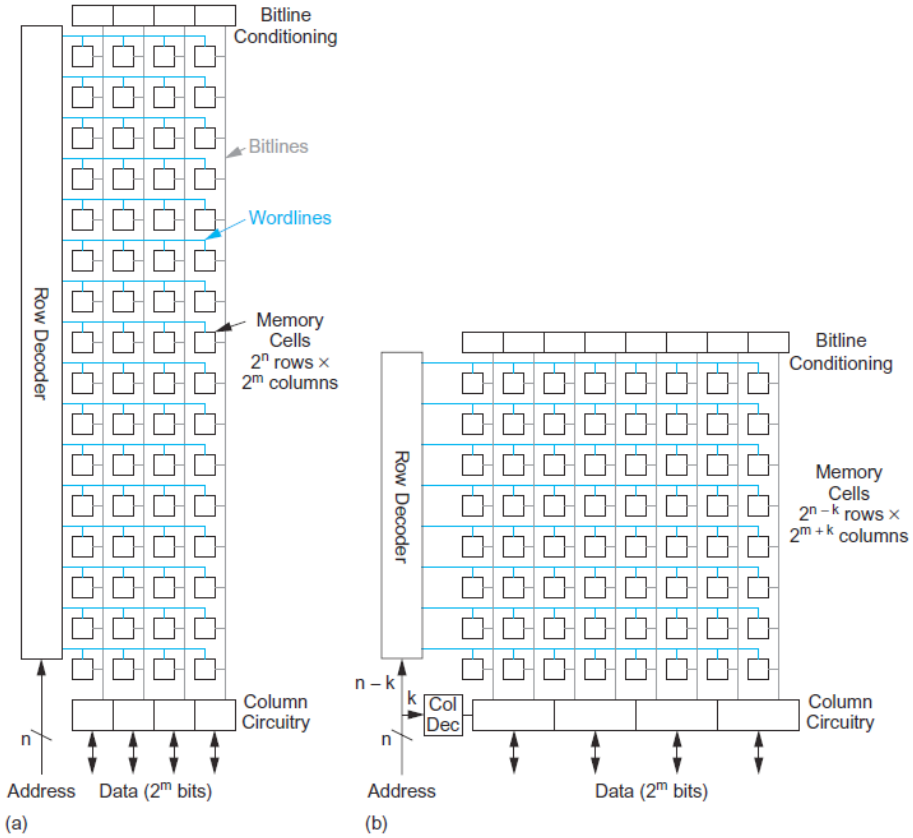


Figure 4.36: Different Memory Architectures

Memory Peripherals

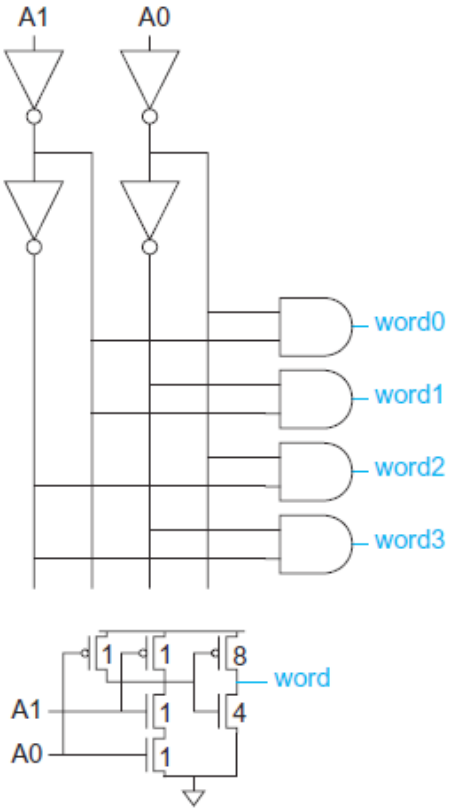
Memory design relies mainly on the peripheral circuits, surrounding the memory core, to remedy the problem of speed and electrical integrity [**56**] and while the design of the memory core is out of hand of the designer and only depends on the available technologies, the design of the peripheral circuits depend on the designer and whether his design has a remarkable impact on the overall design or not.

Row Decoder

Many techniques can be used in the design of the Row address decoder of the memory block. The first technique, or the straight forward technique, is the ordinary NAND or NOR gates decoder with the input of the gates representing the Address of the row and the output connected to the word lines of the memory block, as shown in **Figure 4.37**. It was proved by experiments that NAND gates are faster than NOR gates. So, we will concern NAND gates only. Inverter sizing should be taken into consideration during the design so that it could drive the heavy word line capacitances. This technique

is working efficiently for small addresses "i.e. small number of input bits to the NAND gate.

Figure 4.37: NAND Gate Decoder



However, for large number of inputs, or large range of addresses, we will face the problem of large Fan-in which will affect the delay and the performance of the system. This problem could be recovered by the usage of pre-decoding technique; this technique depends on splitting the logic gate, a complex one, into two or more logic layers, this could produce both a faster and a cheaper implementation, but with a trade-off with the power consumption, it should be noticed that the pre-decoder doesn't improve the path effort but it do improve the area needed, as shown in **Figure 4.38.**

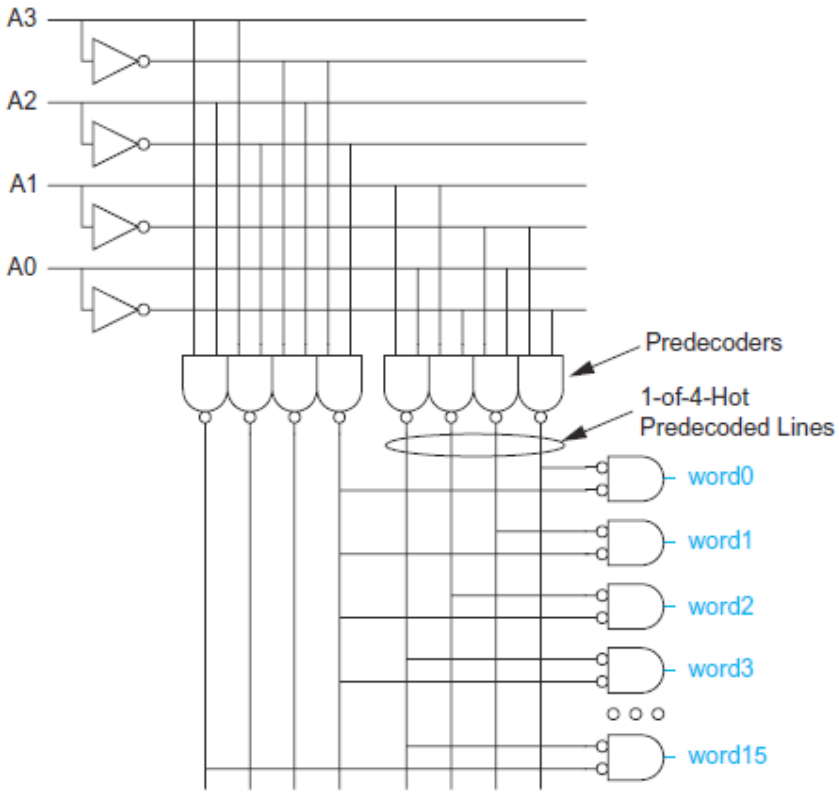


Figure 4.38: Pre-decoding Algorithm

Column Decoder

A column decoder, mainly, performs the same function as the row decoder; the difference is that it should match the bit lines of the memory block. When performing a read operation, the decoder should provide a discharge path from the precharged bit lines to the sense amplifier. During the write operation, the decoder should be able to drive the bit line high or low to write a 1 or 0 in the memory cell, respectively [**54**]. Two implementations are used in the design of the column decoder; the choosing of one of them depends on area, performance, and architectural considerations.

Mainly, there are two architectures in designing column decoders like pass-transistor based decoder and tree based decoder.

Sensing Amplifier

Generally, there are two algorithms of sensing amplifier circuits; large signal-sensing and small signal-sensing. In large signal-sensing, sometimes called single-ended sensing, the sensing circuit has to wait for a full bit line swing between zero and VDD. The advantage of this technique is the high accu-

racy of the sensed signal. However, it takes a long time due to the full swing of the signal being sensed.

The small signal-sensing, called differential sensing, has an advantage over the large signal-sensing, it sense the small difference in the signal, so it saves time and energy. However, this technique needs a timing circuit to state when the sense amplifier should operate, if the time was too short, a wrong answer may be detected [**56**].

Multi-port SRAM

Data caches in super-scalar microprocessors often require multiple ports to handle multiple simultaneous loads and stores. In a multi-ported SRAM, each additional single-ended read port can be provided at the cost of a read word line, a read bit line, and two read transistors. Each additional write port can be provided at the cost of a write word line, true and complementary write bit lines, and two access transistors [**56**].

4.6.3 Approach

It is beneficial for our architecture to use Dual- or Triple-port SRAM. However, due to the time constraints and the lack of advanced memory design tools, in this version of the project we use a single port SRAM. Also, is is not a good practice to implement the memory in RTL as it gets synthesized into register files.

We used Artisan memory compiler to generate our data memory **Figure 4.39.**

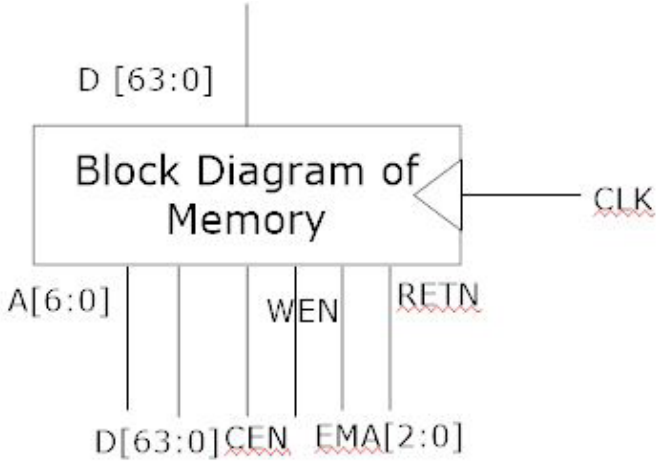


Figure 4.39. Block Diagram of Memory

4.6.4 Results

In **Table 4.9**, you will find the results of our data memory implementation.

Critical Path Length	0.84 ns
Maximum Frequency	1.19 GHz
Total Cell Area	0.2 mm^2
Total Dynamic Power	22.7 mW
Number of Nets	1850 nets

Table 4.9: Synthesis results of the new data memory

5. Realization

5.1 ASIC Design Flow

Generally, any design begins with the specifications to be written first. In general, architects write the specifications which describe the overall architecture, interfaces and blocks without caring about how to implement that on hardware. After writing the specifications, it is the turn of the digital designers to understand the architecture very well and then begin to implement the hardware using HDL languages [**58**].

Our design was implemented using VHDL language. while doing so, we took care of using generically synthesis code that is understood by all synthesis tools. After finishing the RTL model and performing the tests discussed in chapter 6, our design was ready for logic synthesis. In general, Logic synthesis tools convert the RTL description to a gate-level netlist. A gate-level netlist is a description of the circuit in terms of gates and connections between them. Logic synthesis tools ensure that the gate-level netlist meets timing, area, and power specifications. In fact, logic synthesis is what makes digital VLSI design different from software programming. In an HDL like Verilog or VHDL, not everything that can be simulated can be synthesized. Some codes contain some syntax that can't be synthesizable by synthesis tools like for loop instructions. There is a difference between simulation and synthesis semantics. Simulation semantics are based on sequential execution of the program with some notion of concurrent synchronous processes, while synthesis works on converting the code to gate-level netlist. Not all such programs can be synthesized. It is not easy to specify the synthesizable subset of an HDL. It depends on the designers experience to figure out what the tool can or cannot synthesize.

We use Synopsys Design Compiler for logic synthesis, which is the de-facto standard for logic synthesis. It takes the standard cell library and the RTL code as inputs. it is controlled through a TCL script that includes the compiler instructions and the design constraints. Throughout the book, we present the synthesis results for each module. Due to the large design we are dealing with, we preferred to take the bottom up approach, due to its speed. Each block is synthesized separately. The memory

modules are generated using Artisan Memory Compiler. It is then compiled using Synopsys Library Compiler in a Design Compiler friendly format, i.e. the .lib files are converted into .db format. As a final step at this stage we synthesize the core's top module, using the previously synthesized netlists of the blocks.

After logic synthesis, pre-layout static timing analysis is performed on the design to check forset-up and hold time violations before moving on to the back-end stage of the design.

The first step in this stage is to adapt the memory cells to be compliant with the Milky way library format adopt by Synopsys IC Compiler that we use for placement and routing. We add the (.vclef) file that holds the physical dimensions and ports of the memory cells to our Milky way library, to make a FRAM view of such cells.

Afterwards, we import the previously synthesized netlists and the technology files in the same milky way library and we start placement and routing process in the following steps:

1. The initial floor-plan is made, declaring the space of which the standard cells will be placed and the space of IO pins and pads

2. Rectangular power rings and power straps are made in a way that power with VDD and GND are distributed over all the chip, so that it does not take long routes to reach any cell in the design, which make the power degrades with increasing the wire length due to wire resistance. It also decreases routing wires to the power pins.

3. Preroute is done to perform RC extraction, RC estimation, delay calculations. At this step, the routing topology is estimated.

4. This step is optional but recommended. It is called power planning.

5. Placement and placement optimization are performed. The area that any block should be placed in and its margin are defined, so as not to overlap with other blocks.

6. Clock tree synthesis is performed. In our design we had only one main clock tree.

7. Routing is done in two steps:

1. Global routing: during this step, no connections are made, they are just planned. Typically, the whole chip is globally routed before detailed routing.

2. Detailed routing: the step at which specific routing tracks, wires and vias are assigned. After that, post-layout static timing analysis is performed. Also, DRC and LVS checks are made before the final chip tape-out.

5.2 Overall Results

In the tables from **5.1**, **5.2** and **5.3** summarization of the overall results of the project.

Block	Area
Overall Area (after placement and routing)	0.77 mm^2
FMA	0.06 mm^2
Divider	0.01 mm^2
Control Unit	0.08 mm^2
Memory	0.2 mm^2
Router	0.085 mm^2
Scratch Pad Memory	0.01 mm^2
Fetch unit	0.01 mm^2

Table 5.1: Area Results

We can clearly see from **Table 5.1** that we could only achieve a core utilization of around 65%. That is due to time limitations and our poor experience in that part. As it will be mentioned in chapter 7, we expect to reach better ratios soon. It is also noticeable that the area will be optimized upon using the new NoC router we are currently working on.

Block	Total Dynamic Power
FMA	132 mW
FDIV	13 mW
Memory	22 mW
Router	114 mW
Control Ubit	6 mW
Scratch Pad	7 mW
Others	0.5 mW
Overall power consumption	294.5 mW

Table 5.2: Power Results

We can see from **Table 5.2** that the most two power consuming block are the FMA and the router. Already, we are on our way to optimize the router's power as well as other parameters. Concern-

ing the FMA, we will try to optimize its power without affecting its performance. However, we have shown in chapter 4 that its results are better than other results published before.

Critical path with old router	0.96 ns (router)	Maximum Frequency	1.04 GHz
Critical path with new router	0.84 ns (data memory/scratch pad memory)	Maximum Frequency	1.19 GHz

Table 5.3: Timing Results

As we are on our way to optimize the router, **Table 5.3** tells us that the critical point concerning performance is the memory technology we possess.

In **Figures 5.1** and **5.2**, snapshots of our core's layout are presented.

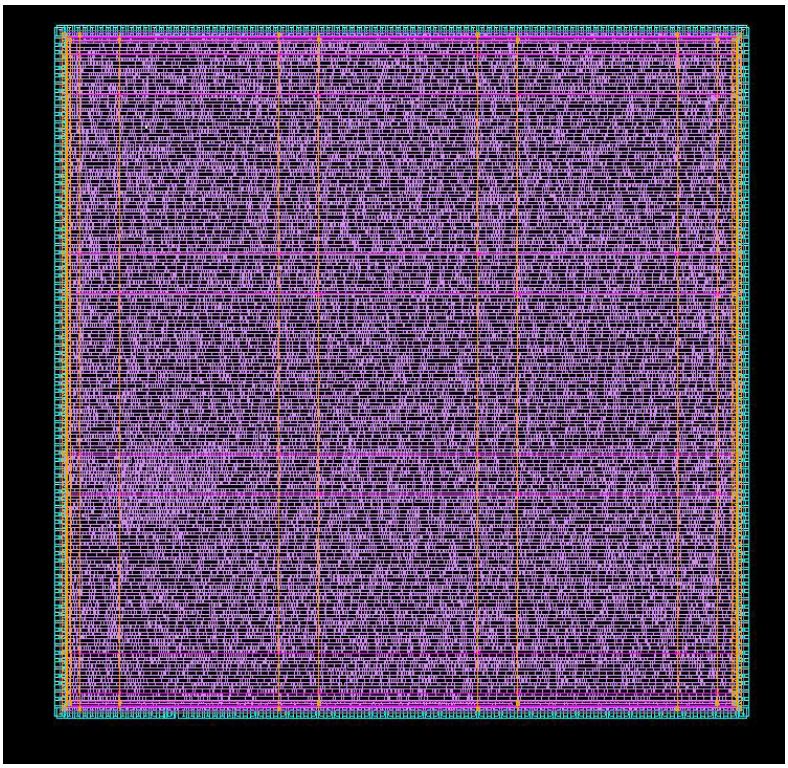


Figure 5.1: Layout

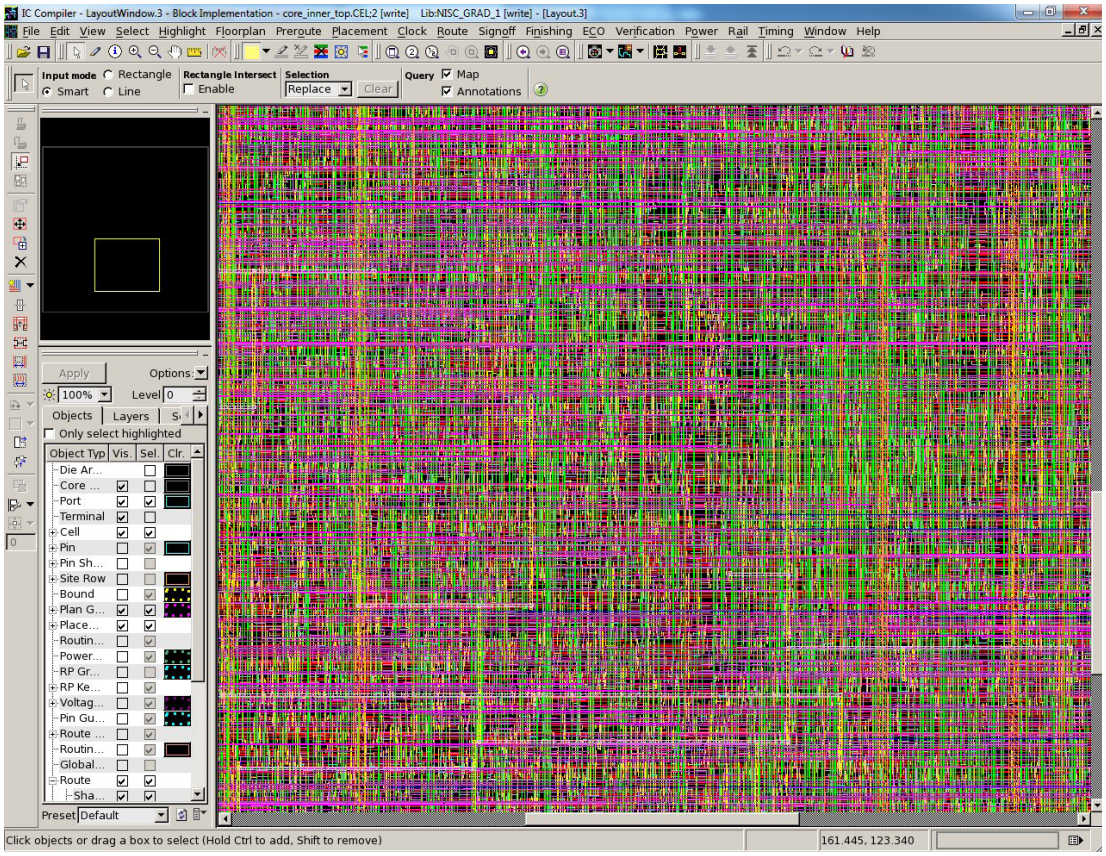


Figure 5.2: Layout (zoomed in)

6. Verification

Without performing the proper tests on hardware designs, they are likely to have errors. Any error has to be discovered and fixed before the chip tape-out. Unfortunately, you can find and fix a bug in the design, but you cannot prove its absence. In this chapter, we present some of the tests we have made so far, in order to verify the correctness of our design.

6.1 Block Level Testing

Each block in the design has been thoroughly tested by a large number of test cases:

6.1.1 Arithmetic Blocks

The most important parameter in testing the arithmetic blocks such as the Fused-Multiply-Add Unit or the Divider is the test set used. The test set we prepared covered a wide range of cases including a wide range of numbers. A large portion of the test set included cases that dealt with zero and infinity whether at the input or the output.

6.1.2 Router

In order to properly test the router, a scenario was built where a controller reads data filled in a register file, sends it from one port of the router, and receives it at another port. The controller tested cases where the router is flooded with data.

6.2 Single Tile Integration Testing

Before we implemented the microprogrammed control unit, we needed to test the performance of all blocks working together, so, we designed two hardwired controllers to do so:

1. The first controller received an n x n matrix and performed LU decomposition on it.

2. The second controller received an n x n matrix and calculated its inverse.

6.3 Core Testing

After the implementation of the microprogrammed control unit, that was the hardest testing task so far.

To test the core at that level, given we have not designed a compiler for it yet, we had to calculate the microprogram that would be saved in the control unit for different applications. We have worked

out three types of applications so far:

1. LU decomposition of an n x n matrix.

2. Matrix Inversion.

3. Matrix Multiplication.

Yet, a full verification has to be tested against a trustful full verification plan.

7. Conclusion and Future Work

This project has implemented the processing element of scalable tiled-based NISC 2D processor arrays that are used for solving statically scheduled scientific problems that include intensive scientific computations. Such problems cover a wide range of scientific problems such as circuit SPICE simulation or some simulations in the biological or medical field. Such problems are solved using supercomputers, among other approaches that also used parallel GPUs or FPGA-based accelerators.Due to the cost represented in the case of super-computers or low performance represented in the other two cases, a need is rising for a high performance reconfigurable computing platform to perform these heavy scientific computations that has to be characterized by high accuracy and low cost in order to be able to provide a reasonable replacement.

The purpose of this project was to solve the previously mentioned problem by implementing the architecture in [**1**] in the 65 nm CMOS technology. As a consequence of this primary goal, some minor goals were set:

• Design a high speed double precision floating point unit with fused-multiply-add unit and a high speed floating point divider.

• Design a scalable NoC router that can be used to interface any cores smoothly without producing performance problems when the number of cores increases.

• Design a micro programmed NISC control unit that achieves a high level of hardware parallelization.

Currently we have implemented successfully one tile of the array. Throughout the book, we have proven the possibility of implementing such an architecture in VLSI 65 nm technology at an operating frequency higher than 1 GHz and with further work this frequency can be raised to 1.5 GHz. In chapter 5, you will find a summarization of the results achieved for each block and the tile as a whole. Among the contributions achieved in the project:

• Proposing and designing a micro-programmed NISC control unit that emulates the function

of ordinary FSM controller in a reconfigurable way and provides a high level of parallelism and utilization of available hardware.

• Designing a fast double precision floating point fused-multiply-add pipeline.

• Modifying the design of hardware SRT radix-4 divider and designing a fast floating point divider based on this modifications.

• Designing of NOC router that is characterized by its high speed, scalability, ease of interface and operating frequency higher than 2 GHz with optimizations made to its crossbar switch.

• An optimized core organization for matrix operations.

Among the limitations for current version of the core:

• Data Memory low performance.

• The lack of proper CAM technology for the control unit.

Despite the positive results achieved in this project or in [**1**], it can be only considered as the beginning of a long journey of research and work before we can get satisfied by the computing platform we are introducing to market. As this work started as a Bachelor of Science graduation project at Alexandria University, our time is limited by the faculty delivery date. Some working has yet to be done to be completely satisfied by this work. Afterwards, we need to develop this project into a commercial product. Among the work needed to do so:

7.1 NoC Router Improvement

As the first router designed is one of the critical blocks and bottlenecks for the performance of the whole systems, we designed a higher performance mesh router in order to speed up the whole design. In section 4.5, you will find early results of this block.

7.2 Floating Point Unit

We will try to further optimize the floating point unit to reach a performance higher than 1.5 GHz. Also, we will try to make our floating point unit support integer operations as well as floating point operations so it would be more generalized.

7.3 Data Memory

As discussed in section 4.6, using a single port memory is crucial to the overall performance specially in multiplication-based operations, as it cuts down the performance of the fused-multiply-add module to one third of its peak performance. We will try to get a better memory compiler that is capable of generating a triple- or dual-port memories and that has a higher clock speed. In case this approach failed, we will have to perform some custom design efforts to produce such a memory. With the work that will be done on both the NoC and the Data Memory, we believe that the performance can be increased to cope with the speed of the floating point unit.

7.4 Micro Instruction Memory

As mentioned in section 4.4, the two approaches we used for the content addressable memory were due to the lack of a content addressable memory available. We will try to fix this problem, or we will have to do some custom design efforts for that too.

7.5 ASIC Implementation

As we discussed in chapter 5, our experience in the back-end design techniques have yet to be improved. We plan to optimize the placement and routing of the chip to achieve a better core utilization ratio. We also have to do more timing, signal integrity and power analysis on the design.

7.6 Multi-core Performance Issues

We are planning to interface a 4 x 4 array of processing elements as well as testing some performance related issues on them

7.7 Verification

Although the core was tested against some algorithms, we still need to make and execute a full trustful verification plan for our design.

7.8 Compiler Design

We plan to design a compiler to calculate the microprograms to be saved in the control unit for each application.

7.9 Chip Implementation

We plan to get some funds to be able to tape-out our final chip.

References

[1] Mohamed W. Hassan; Ahmed A. Abouel Farag; Yasser Hanafy, NOA'S-Arc: NISC based Optimized Array Scalable Architecture.

[2] Rehan Hameed, et.al., "Understanding Sources of Inefficiency in General-Purpose Chips",Commun. ACM,vol 54(10),pp. 85-93, october 2011.

[3] "Yulu Jiaa; Piotr Luszczeka; Jack Dongarra, ""Multi-GPU Implementation of LU Factorization"", Scivwerse ScienceDirect, Iternational Conference on Computational Science, pp. 106-115, November 2012. of LU Factorization"", Scivwerse ScienceDirect, Iternational Conference on Computational Science, pp. 106-115, November 2012. "

[4] Galoppo, N.; Govindaraju, N.K.; Henson, M.; Manocha, D., "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, "Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference , vol., no., pp.3,3, 12-18 Nov. 2005

[5] "Xiaofang Wang; Ziavras, S.G., ""Parallel direct solution of linear equations on FPGA-based machines,"" /Parallel and Distributed Processing Symposium, 2003. Proceedings, pp. 22-26 April 2003. "

[6] Baring It All to Software: Raw Machines, Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua Jonathan Babb Saman Amarasinghe Anant Agarwal. Massachusetts Institute of Technology, Laboratory for Computer Science

[7] "Gordon E. Moore, Cramming more components onto integrated circuits."

[8] Hongtao Zhong, architectural and compiler mechanisms for accelerating single thread applications on multicore processors, University of Michigan, 2008.

[9] Discovery Guide,Multicore Processors – A Necessity,Bryan Schauer, ProQuest, September 2008.

[10] M. Balakrishnan, CHAPTER 6: Microprogrammed Control Unit Design.

[11] NISC Technology website: http://www.cecs.uci.edu/~nisc/

[12] Paul Chow, RISC-(reduced instruction set computers), IEEE Potentials, volume 10, Issue 3, pp.28-31, October 1991.

[13] M. Reshadi; D. Gajski, NISC Modeling and Compilation, Center for Embedded Computer Systems, TR 04-33, pp 2-7, December 2004.

[14] IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Std 754-1985 , vol., no., pp.0_1,, 1985

[15] Michailidis, P.D.; Margaritis, K.G., "Implementing Parallel LU Factorization with Pipelining on a MultiCore Using OpenMP," Computational Science and Engineering (CSE), 2010 IEEE 13th International Conference on , vol., no., pp.253,260, 11-13 Dec. 2010

[16] Tile Processor Architecture Overview for the TILEPro Series RELEASE 1.2, Tilera Corporation, February 2013.

[17] A. Syed, Tradeoffs Between Combinational and Sequential Dividers, Sr. Corporate Applications Engineer, DesignWare IP.

[18] B. Parhami, Algorithms and Hardware Designs, Computer Arithmetic, Oxford University Press, 2000.

[19] Tom M. Bruintjes, Design of a Fused Multiply-Add Floating-Point and Integer Datapath, Master's Thesis, University of Twente, May 2011.

[20] W. Ibrahim, Binary Floating Point Fused Multiply Add Unit, Master's Thesis, Ain Shams University, 2012.

[21] A 1.45 GHz 52-to-162 GFLOPS/W Variable-Precision Floating-Point Fused Multiply-Add Unit with Certainty Tracking in 32nm CMOS, Himashu Kaul, Mark Andres, Sanu Mathew, Steven Hsu, Amit Agarwal, Farhana Sheikh, Ram Krishnamurty, Shekar Borkarr, ISSCC 2012

[22] Vangal, S.R.; Hoskote, Y.V.; Borkar, N.Y.; Alvandpour, A., "A 6.2-GFlops Floating-Point Multiply-Accumulator With Conditional Normalization," Solid-State Circuits, IEEE Journal of , vol.41, no.10, pp.2314,2323, Oct. 2006

[23] Galal, S.; Horowitz, M., "Energy-Efficient Floating-Point Unit Design," Computers, IEEE Transactions on , vol.60, no.7, pp.913,922, July 2011

[24] Mueller, S.M.; Jacobi, C.; Hwa-Joon Oh; Tran, K.D.; Cottier, S.R.; Michael, B.W.; Nishikawa, H.; Totsuka, Y.; Namatame, T.; Yano, N.; Machida, T.; Sang H.Dhong, "The vector floating-point unit in a synergistic processor element of a CELL processor," Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on , vol., no., pp.59,67, 27-29 June 2005

[25] Wallace, C. S., "A Suggestion for a Fast Multiplier," Electronic Computers, IEEE Transactions on , vol.EC-13, no.1, pp.14,17, Feb. 1964

[26] R. Uma; V. Vijayan; M. Mohanapriya; S. Paul, Area, Delay and Power Comparison of Adder Topologies, International Journal of VLSI Design& Communication Systems, Vol. 3, No. 1, February 2012.

[27] Hooman Nikmehr, Architectures for Floating Point Division, Doctor of Philosophy Thesis, University of Adelaide, Australia, August 2005.

[28] S. Oberman, Design Issues in High Performance Floating Point Arithmetic Units, December 1996.

[29] Harris, D.L.; Oberman, S.F.; Horowitz, M.A., "SRT division architectures and implementations," Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on , vol., no., pp.18,25, 6-9 Jul 1997

[30] T. Williams; M. Horowitz, SRT Division Diagrams and their Usage in Designing Custom Integrated Circuits for Division, Technical Report, November 1986.

[31] Amey P. Phadke, Study of Recursive Divide Architectures Implementation for Division and Multiplication, Master's Thesis, University of Pune, India, 2007.

[32] Quotient Digit Selection Logic for Floating Point Division Square Root, US Patent 5954789

[33] Pham, T.N.; Swartzlander, E.E., "Design of Radix-4 SRT Dividers in 65 Nanometer CMOS Technology," Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on , vol., no., pp.105,108, Sept. 2006

[34] Jan Fandrianto, Algorithm for High Speed Shared Radix 4 Division and Radix 4 Square-Root, IEEE 1987

[35] Atkins, D.E., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," Computers, IEEE Transactions on , vol.C-17, no.10, pp.925,934, Oct. 1968

[36] Nikmehr, H.; Phillips, B.; Cheng-Chew Lim, "Fast Decimal Floating-Point Division," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.14, no.9, pp.951,961, Sept. 2006

[37] Joengmin Ha; Sooyeon Lim, The Implementation of the SRT Division Algorithm, Undergraduate Thesis, yonsei University, 2006.

[38] Floating Point Remainder Generator for Maths Processor, US Patent 5357455

[39] Wey, C.-L.; Wang, C.-P., "Design of a fast radix-4 SRT divider and its VLSI implementation," Computers and Digital Techniques, IEEE Proceedings - , vol.146, no.4, pp.205,210, Jul 1999

[40] S. Oberman; N. Quach; M. Flynn, The Design and Implementation of a High Performance Floating Point Divider, Technical Report, January 1994.

[41] David Goldberg, Computer Architecture- A Quantitative Approach, Computer Arithmetic - Appendix I

[42] Method and System of a Microprocessor Subtraction-Division Floating Point Divider, US Patent 7127483 B2

[43] Statistical Analysis of Floating Point Flaw: Intel White Paper, 2004

[44] Alexander Barkalov; Larysa Titarenko, Logic synthesis for FSM based control units.

[45] Low Power SRAM Based Content Addressable Memory, US Patent 2012/0127772 A1

[46] Z. Ullah; S. Baeg, Vertically Partitioned SRAM-Based Ternary Content Addressable Memory, IACSIT International Journal of Engineering and technology, Vol. 4, No. 6, December 2012.

[47] D. Atienza; F. Angiolini; S. Murali; A. Pullini; L. Benini; G. De Micheli, Network-on-Chip design and synthesis outlook, INTEGRATION, the VLSI journal 41, 2008.

[48] H. Yoo, K. Lee; J. Kim, Low-Power NoC for High-Performance SoC Design, 2008.

[49] W. Dally;B. Towles, Principles and Practices of Interconnection Networks, 2003.

[50] Imran Shamim, Energy Efficient Links and Routers for Multi-Processor Computer Systems.

[51] Shubhangi D Chawade; M. Gaikwad; R. Patrikar, Design XY Routing Algorithm for Network-On-Chip Architecture.

[52] Shin, E.S.; Mooney, V.J.; Riley, G.F., "Round-robin Arbiter Design and Generation," System Synthesis, 2002. 15th International Symposium on , vol., no., pp.243,248, 2-4 Oct. 2002

[53] P. Chu, RTL Hardware Design Using VHDL, 2006.

[54] J. Deschamps; Gery J. Bioul; G.Sutter, Synthesis of Arithmetic Circuits.

[55] Joshua Ruggiero, Measuring Cache and Memory Latency and CPU to Memory Bandwidth For use with Intel® Architecture.

[56] J. RABAEY, Digital Integrated Circuits : A Design Perspective.

[57] N. Weste; D. Harris, CMOS VLSI Design: A Circuits and Systems perspective.

[58] Samir Palnitkar, Verilog HDL - A Guide to Digital Design and Synthesis, Second Edition.

[59] "Sadhna.K. Mishra et. al., 1729-1736 Processor Architecture Design Practices, / International Journal of Engineering Science and Technology Vol. 2(6), 2010"