Chapter 1

INTRODUCTION

This research has been carried out to implement a flexible and efficient and generic processor to perform digital signal processing tasks on the field programmable gate array (FPGA). The architecture of the proposed processor is designed for various digital signal processing algorithms. FPGA is a concurrent device and keeping in mind that there is bulk of data that needs to be processed in any digital signal processing (DSP) task to get the results. For example, in Fourier Transform (FT), if we want to get FT of a signal then we need to process thousands of samples (input data) to get the result. So almost all signal processing tasks are very computational intensive. So, to carry out these tasks efficiently, a very efficient and powerful processor was required. The proposed solution is entirely generic and can perform any of the digital signal processing tasks.

A question arises that why a processor was needed even though we have DSP kits available to perform DSP tasks? The answer is that, those DSP kits are sequential in nature i.e. they perform every task sequentially and there is no concept of parallel processing in these devices and we cannot perform parallel/concurrent processing in these devices so during this research work it has been observed that much of operations other than actual processing such as data input/output put extra burden on the processing if they are not done in parallel to the processing of data.

The VLSI design cycle has three main components namely: designing, implementation and verification. This project emphasizes on the design of a special family of circuits called synchronous circuits. Most of the designs nowadays implemented belong to this family. Synchronous digital circuits are those where a digital clock clicks and makes all components synchronously operate to implement the design functionality. These circuits can be implemented easily in both Verilog and VHDL by any proficient hardware programmer but resource management of the target device and efficient utilization of clock ticks is where engineering comes in. Not every engineer is

capable enough and makes the successful design in market easily. Both a stroke of genius and sense of creativity are required, especially when design is complex for which verification requires testing millions of gates. This is indeed the most challenging part of the DSP design project. Project managers of the DSP design project first have to finalize the requirements and specifications (R&S) which along with the functionality of the product, includes the power consumptions, input sampling rate, noise tolerance and other application specific parameters, for example bit error rate (BER) and baud rate in communication system.

The R&S are explored are explored to try to get many solutions and best among them is chosen. After testing of algorithm on high level languages, it comes the task of implementation. The implementation is first tested on FPGA before fabrication.

All circuits, implemented in hardware are designed in Register Transfer Language (RTL) logic which consists of register arrays separated by computational clouds. These clouds basically implement arithmetic operations involved in the logic design. Ninety percent of the critical path delay is caused by combinational cloud. Rest is due to the reading from and writing into the registers. The longest path between two consecutive register arrays is called the critical path, draws limitation on the maximum clock frequency.

1.1 PROBLEM STATEMENT

"To design a generic processor that facilitates digital signal processing applications in a fast and efficient way"

1.2 RESEARCH AIM

To facilitate the DSP designers to analyze and decompose 1-D and 2-D signals to get finer details, in terms of high frequency and low frequency components etc., to be used in more complex applications of digital signal processing with higher throughput.

1.3 PROPOSED SOLUTION

To achieve the goal of this research work, a very flexible methodology has been adopted to accomplish the task. A Generic processor has been implemented based on the Reduced Instruction Set Computer (RISC) architecture. The architecture is fully pipelined and hence gives higher throughput. An instruction set is designed to facilitate the programmer to use the system easily. Based on the instruction set, all operations of the proposed processor are carried out. Due to pipelined design, the achieved cycle per instruction (CPI) is 1 that has a very significant impact on the throughput of overall system.

1.4 MOTIVATION

The field of digital signal processing (DSP) has been always a very challenging and innovative field for engineering applications. Also digital design of signal processing systems gives a new way of thinking to implement digital signal processing algorithms embedded devices such as FPGAs and DSPs. To reduce the cost of end user devices, the system design in double precision floating point format are converted to its fixed point equivalent. The fixed point implementation is them mapped on fixed-point DSPs, FPGAs and ASIC.

Taking into account the need to develop the wavelet processing system onto off the shelf components such as FPGA the proposed system has been developed to facilitate the DSP designers.

1.5 REPORT STRUCTURE

This document throws light on all the aspects of the project, including the techniques implemented and details of the developed system. This document is divided into following chapters:

- Chapter 2: Literature Review and background. This chapter contains a glimpse of all the literature gone through for the completion of the project.
- Chapter 3: Methodology. This chapter explains the approach used for development of the algorithm, system architecture and design of the project.
- Chapter 4: Results and Discussions. This chapter includes the steps followed and the final results obtained.
- Chapter 5: Conclusions and Recommendations. This chapter summarizes the whole project report and contains proposed methodologies, which could further affect improvements in the project and the research carried out during the execution of this project, and finally formulates recommendations for future work.

Chapter 2

Literature Review and Background

The literature review has been carried out to figure out the perfect understanding of different aspects of the work and also to understand the working of tools that are to be used to complete the project. Appraisal of the literature is categorized in modules as under.

1) Digital System Design Process
2) Digital System Design Considerations
3) Design Based on Finite State Machines
4) Design of Data path Controllers
5) Digital Signal Processing Review

2.1 DIGITAL SYSTEM DESIGN PROCESS

Main points of the digital system design process that are necessary of the understanding of the subject are discussed in this section. Following shows the complete digital system design process:

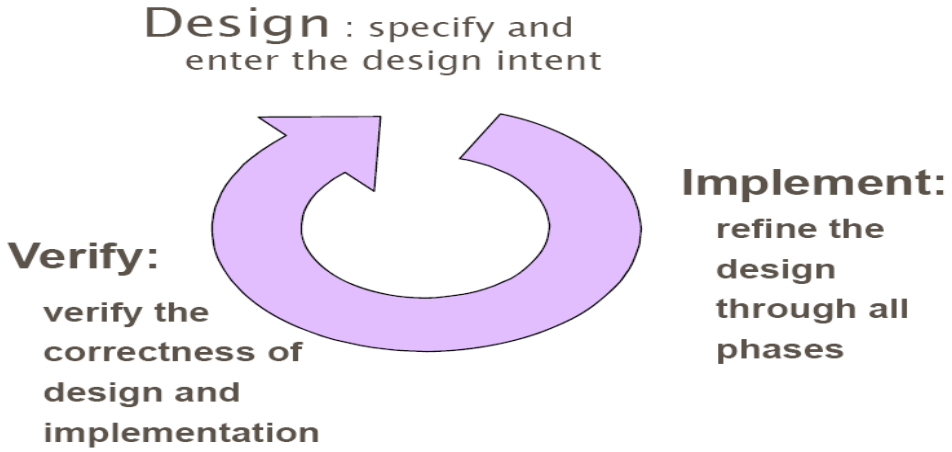


Figure 2.1: Digital System Design Process [3]

The cycle starts with the requirements specification, followed by the design of an algorithm using tools like MATLAB. To facilitate partitioning of the algorithm into hardware (HW) and software (SW), and its subsequent mapping on different platforms, algorithm design and coding techniques in MATLAB are described. The MATLAB code has to be structured so that the algorithm developers, SW designers and HW engineers can correlate various components and can seamlessly integrate, test and verify the design and can return to the original MATLAB implementation if there are any discrepancies in the results.

2.1.1 System Design

It is a level of abstraction where the digital designer specifies all the registers and elaborates how data will flow through these registers. The combinational logic between two sets of registers is usually described using high level mathematical operations, and is drawn as a cloud.

The design process is done at behavioral modeling level of abstraction which is the highest level, and then mapped into gate level net list for implementation. The design typically means description and flow of data through the registers of the architecture.

2.1.2 System Implementation

After describing the entire architectural design in RTL logic, it is implemented in some hardware description language (HDL), Verilog in our case, which in general is a straight forward translation. The translated code is tested and synthesized to be programmed on a Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC).

2.1.3 System Verification

Usually any digital design is tested and verified by checking its output for all possible combinations of inputs. But as number of gates on single silicon device is increasing, this makes the job of testing and verification job a very critical and challenging one.

2.2 DIGITAL SYSTEM DESIGN CONSIDERATIONS

A digital designer is always confronted with finding the best design options in area-power time tradeoffs. Following are some design objectives [3]:

1) Area of the design
2) Critical path delay of the design
3) Testability of the design
4) Power dissipation of the circuit

By considering all the four mentioned factors the efficiency of a product can be measured.

Power is increasingly becoming the key limitation in processor performance. In the embedded market, where many processors go into environments that rely solely on passive cooling or on battery power, power consumption is often a constraint that is as important as performance and cost.

No doubt, many readers will have encountered power limitations when using their laptops. Indeed, between the challenges of removing excess heat and the limitations of battery life, power consumption has become a critical factor in the design of processors of laptops. Battery capacity has improved only slightly over time, with the major improvements coming from new materials. Hence, the ability of the processor to operator efficiently and conserve power is crucial. To save power, techniques ranging from putting parts of the computer to sleep, to reducing clock rate and voltage, have all been used. In fact, power consumption is so important that Intel has designed a line of processor, the Pentium M series, specifically for mobile, battery-powered applications.

For CMOS technology, we can reduce power by reducing frequency. Hence, recent processors intended for laptop use all have the ability to adapt frequency to reduce power consumption, simultaneously, of course, reducing performance. Thus, adequately evaluating the energy efficiency of a processor requires examining its performance at maximum power, at an intermediate level that conserves battery life, and at a level that maximizes battery life. In the Intel Mobile Pentium and Pentium M lines, there are two

available clock rates: maximum and a reduced clock rate. The best performance is obtained by running at maximum speed, the best battery life by running always at the reduced rate.

Figure 2.2 shows the performance of three Intel Pentium processors designed for use in mobile applications. As we can see the newest processor, the Pentium M, has the best performance when run a full clock speed, as well as with the adaptive clock rate mode. The Pentium M's 600 MHz clock makes it slower when run in minimum power mode than the Pentium 4-M, but still faste than the older Pentium III-M design.

Figure 2.2 shows the relative energy efficiency or these processors running the SPEC2000 benchmarks. In all three modes, it has a significant advantage in energy that the Pentium 4-M has only a slight efficiency advantage over the Pentium III-M. this data clearly shows the advantage of a processor like the Pentium M, which is designed for reduced power usage from the start, as opposed to a design like the Pentium III-M or Pentium-M, which are modified versions of the standard processors, of course, adequately measuring energy efficiency also requires the use of additional benchmarks designed t reflect how users employ battery-powered computers. Both PC review magazines and Intel's technical journal regularly undertake such studies.

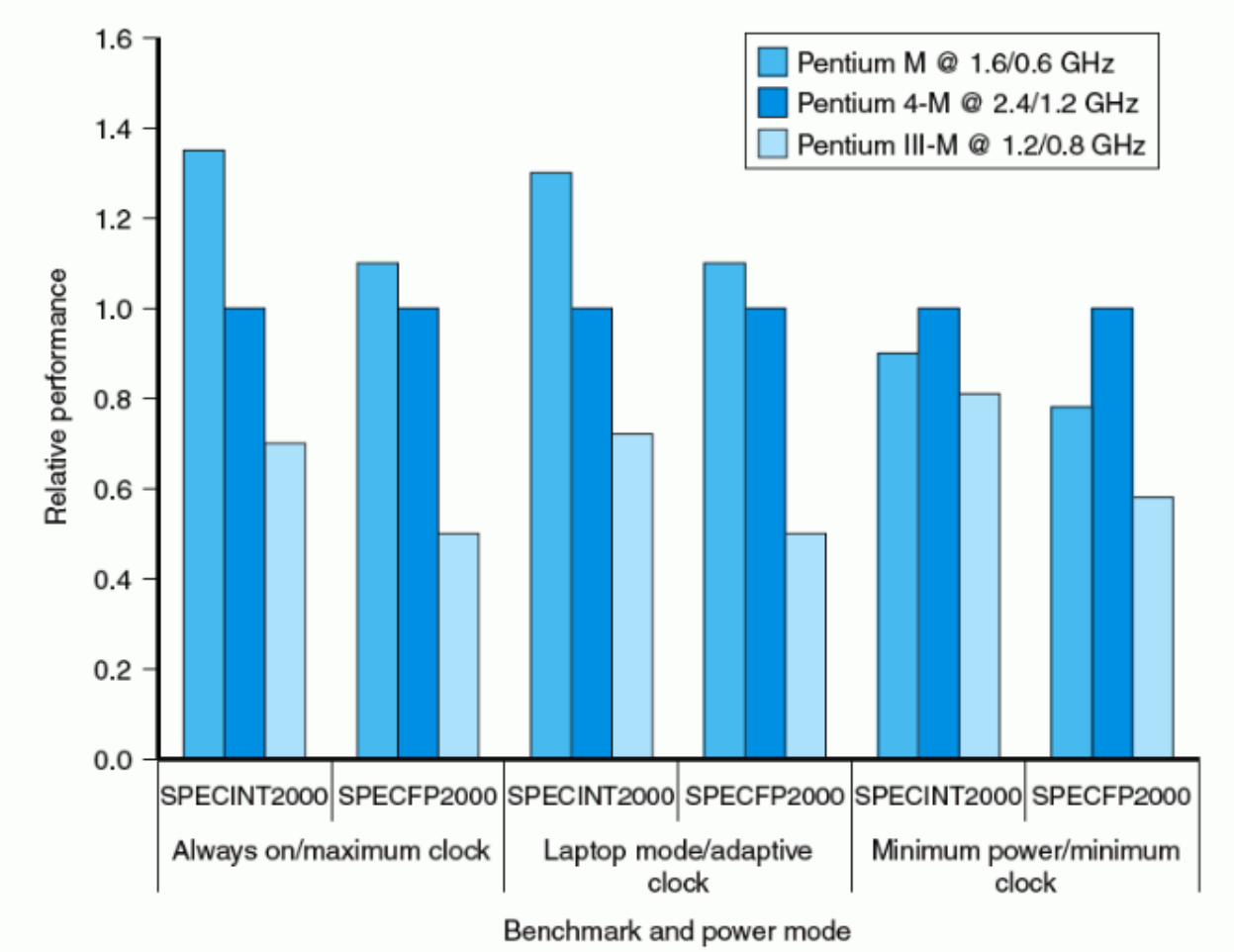


Figure 2.2 : Relative performance of Intel processors

2.3 DESIGN BASED ON FINITE STATE MACHINES

Partitioning a sequential machine into a data path and a controller clarifies the architecture and simplifies the design of the system. The sequence of steps in an application-driven design process is shown in Figure 2.3. If the architecture of the data path unit has been selected to support the instruction set of an application, sequences of operations (control states) that support the instruction set can be identified. The control states are used to schedule assertions of the signals that control the movement and manipulation of data as the machine executes instructions.

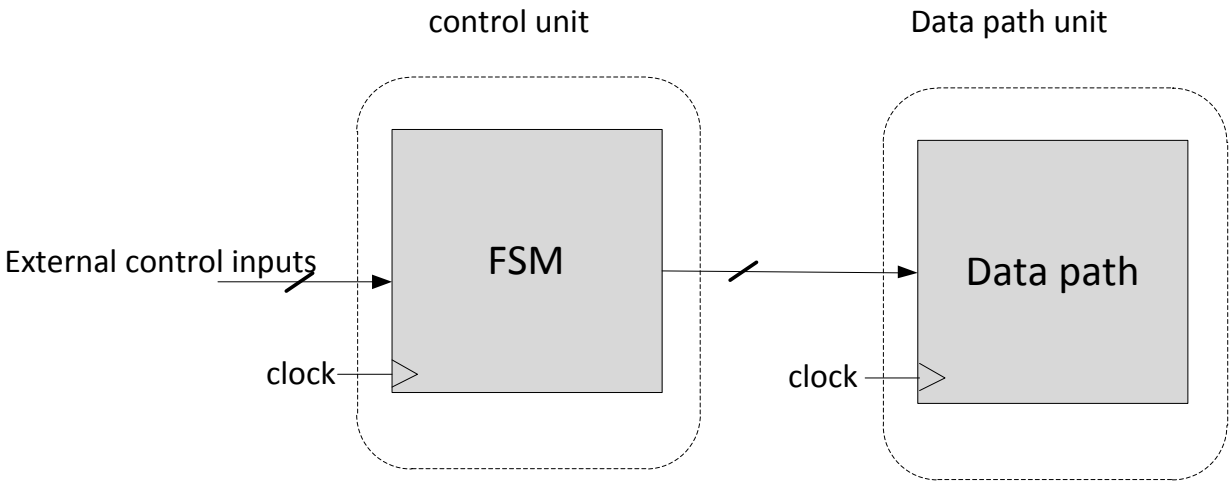


Figure 2.3: State machine controller for a datapath

Then an FSM can be designed to generate the control signals. In this section we will illustrate the design of datapath controllers for some simple functional units, to prepare for the design of a stored-program reduced instruction-set computer in the next section. In synchronous machines, a common clock synchronizes the activities of the controller and data path functional units.

Note that the control unit in Figure 2.3 is implemented as an FSM, and is itself controlled by external input signals and by status signals from the data path unit. The FSM produces the signals that control the operation of the data path unit. Data path units are commonly described by dataflow graphs; control units are commonly modeled by state transition graphs and/or algorithmic-state machine (ASM) charts for FSMs. Partitioned sequential machines can be modeled by an FSM and datapath (FSMD), a combined control-dataflow graph, which expresses datapath operations in the context of a state-transition graph (STG). We favor using an ASM and datapath (ASMD) chart, which likewise links an ASM chart for a control unit to the operations of the data path that it controls.

2.4 DESIGN OF DATAPATH CONTROLLERS

Digital systems range from those that are control-dominated to those that are data-

dominated. Sequential machines are commonly classified and partitioned into data path units and control units.

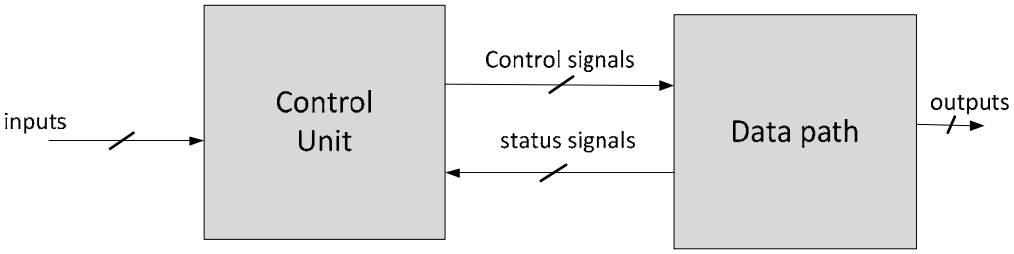


Figure 2.4: Components of time shared architectures [3]

In general, a time shared architecture consists of a datapath and a control unit. The data path is the computational engine and consists of registers, multiplexers, de multiplexers, ALUs, multipliers, shifters, combinational circuits and buses. These HW resources are shared across different computations of the algorithm. This sharing requires a controller to schedule operations on sets of operands. The controller generates control signals for the selection of these operands in a predefined sequence. The sequence is determined by the dataflow graph or flow of the algorithm. Some of the operations in the sequence may depend on results from earlier computations, so status signals are fed back to the control unit. The sequence of operations may also depend on input signals from other modules in the system.

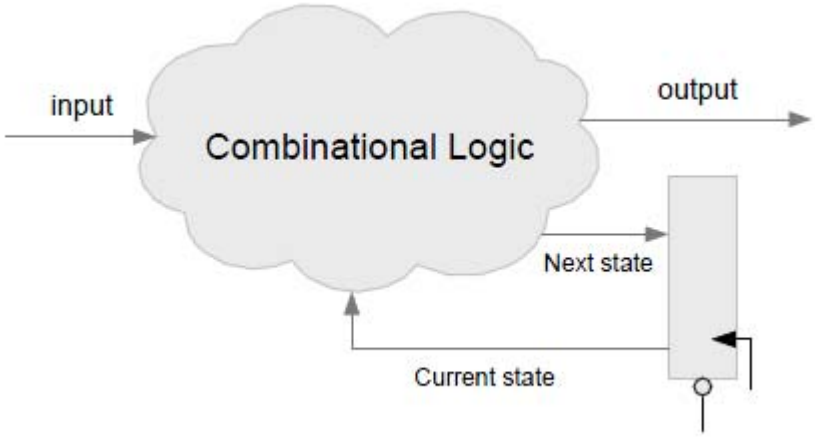


Figure 2.5: Combinational and sequential components of FSM [3]

Most data paths include arithmetic units, such as arithmetic and logic units (ALUs), adders, multipliers, shifters, and digital signal processors, but some do not, such as graphics coprocessors. The data path unit is controlled by a finite-state machine (FSM) that coordinates the execution of instructions that perform operations on the data path. Architectures that are dominated by control units will generally have a significant amount of random (irregular) logic, together with some regular structures, like multiplexers for steering signals, and comparators[10].

2.5 DIGITAL SIGNAL PROCESSING REVIEW

As the proposed processor has been designed specifically for digital signal processing applications, so a comprehensive study of signal processing algorithms has been carried out. Following is the brief description of the some digital signal processing transform and their comparison.

2.5.1 Fourier Transform

Fourier Transform (FT) is used to compute which frequency components exist in the signal. So Fourier Transform gives the frequency spectrum of the signals. Frequency spectrum shows the frequency components that are also called spectral components of the signal. As we know that frequency is rate of change something. If something changes abruptly then its frequency is high and if something changes slowly then its frequency is low. If there is no change in any signal then its frequency is zero.
The mathematical formulations of the Fourier Transform are as under:

$$F(\omega) = \int [f(t)\exp(-j\omega t)dt]$$

$$f(t) = 1/2\pi \int F(\omega)\exp(j\omega t)d\omega$$

Mostly in our daily life the signals we encounter are time domain signals. There are two types of signals

- Stationary signals.
- Non-stationary signals.

The stationary signals are those that have fixed frequency over all times as shown in figure 2.6 below shows a signals that has four

frequency components and those four frequency components exist in this signal at all

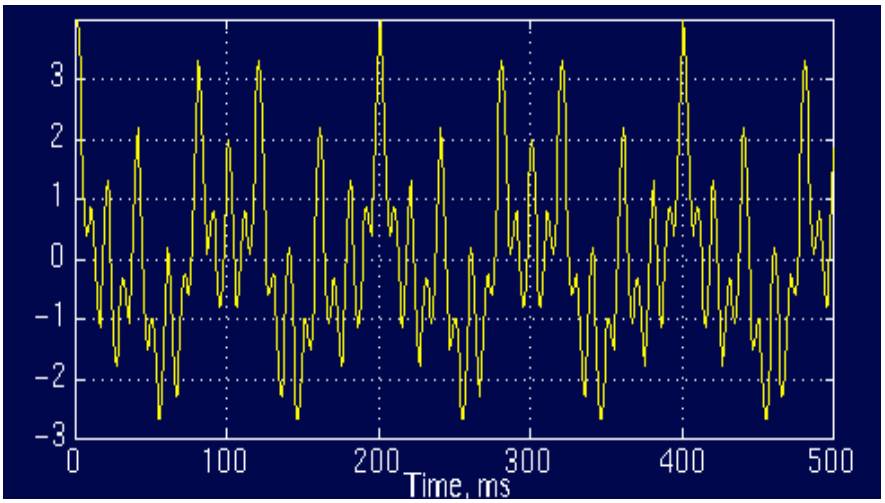


Figure 2.6: A time domain stationary signal

times that's why this signal is called stationary signal.

Figure 2.7 shows a non-stationary signal i.e. it has different frequencies and different at different time. By looking at the figure below it can be seen that the signal has different frequencies at 200, 400, 600 intervals and so on.

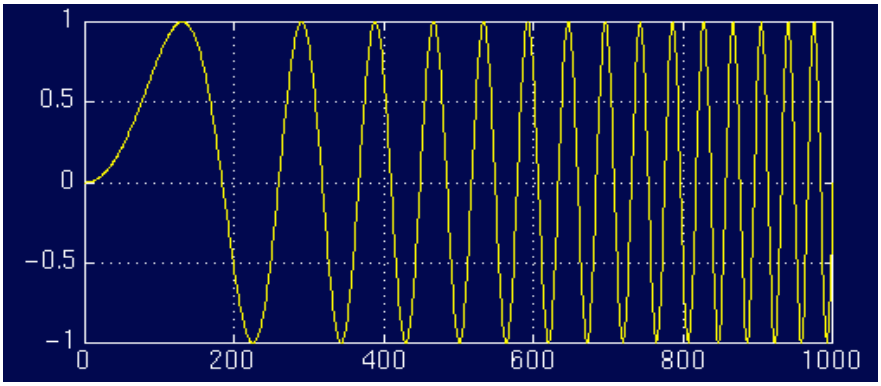


Figure 2.7: Non-Stationary signal

The Fourier Transform of the signals only gives information about the frequency components that exist in the signal. So, for those application Fourier Transform is not a

suitable then people sit together and come with a transform that gives both time and frequency information simultaneously i.e.

2.5.2 Wavelet Transform

Here is the description of the working of the Wavelet Transform. Suppose we have a signal that has frequency components from 0 to 100. First we will break the into low frequency and high frequency components i.e. the two portions of the signal, one containing the frequency components from 0 to 50 and the other from 51 to hundred. Now we have to different signals that have different frequency components but they both belong to the same signal. If we want to further split the signals then we will take any one of them, normally low pass portion is taken again because most signal information is in lower frequency components. So we take the signal with frequencies from 0 to 50, again break this signal into low and high frequency components i.e. from 0-25 and from 26-50. And in the same way keeping on breaking any portion into low and high frequency components and reach the desired level of decomposition. If we plot these bunch of signal on the same 3-D graph i.e. one axis giving time information, 2nd axis giving frequency information and the 3rd axis giving amplitude then this graph will look like figure 2.8. Note that frequency axis is labeled as scale and scale is inverse of frequency.

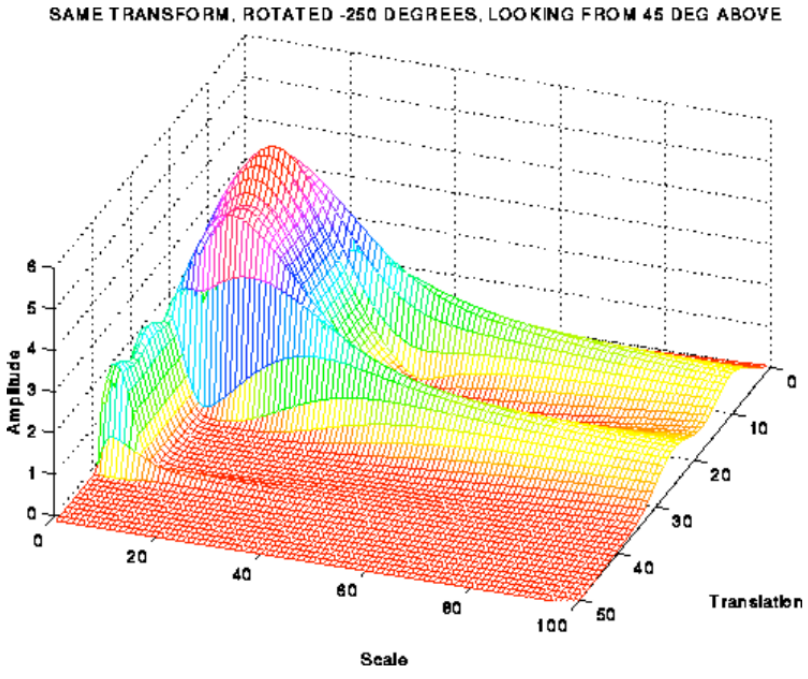


Figure 2.8: Wavelet Transform

There are two types of Wavelet Transform:

- Discrete Wavelet Transform
- Continuous Wavelet Transform

Our focus will mainly be on Discrete Wavelet Transform.

2.5.3 LIFTING SCHEME

Lifting scheme is a very efficient approach to perform the discrete wavelet transform. Lifting schemes reduces the number of computation of the wavelet transform to almost half. Figure 2.12 shows how the lifting scheme works.

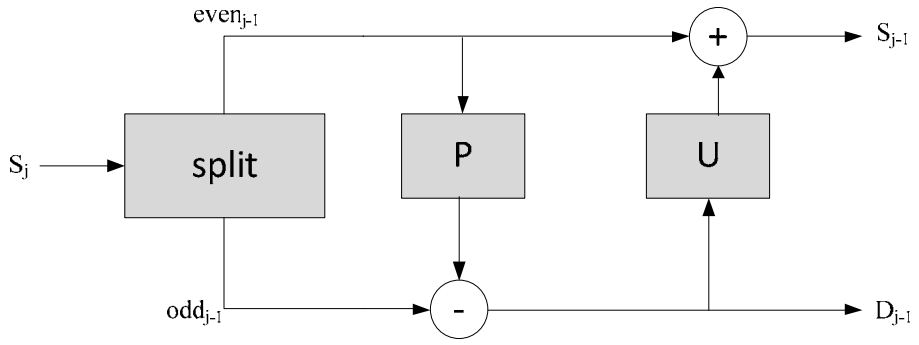


Figure 2.9: Lifting Scheme Forward Transform

In lifting scheme three steps are performed i.e.

- Split
- Predict
- Update

After performing these steps the input signal is split into a high pass and a low pass component. S corresponds to high-pass and D corresponds to low pass component.

In the same way as in forward transform, the inverse transform using lifting scheme also exists. Figure 2.13 shows the inverse transform using lifting scheme. After performing the inverse transform we get the original signal back.

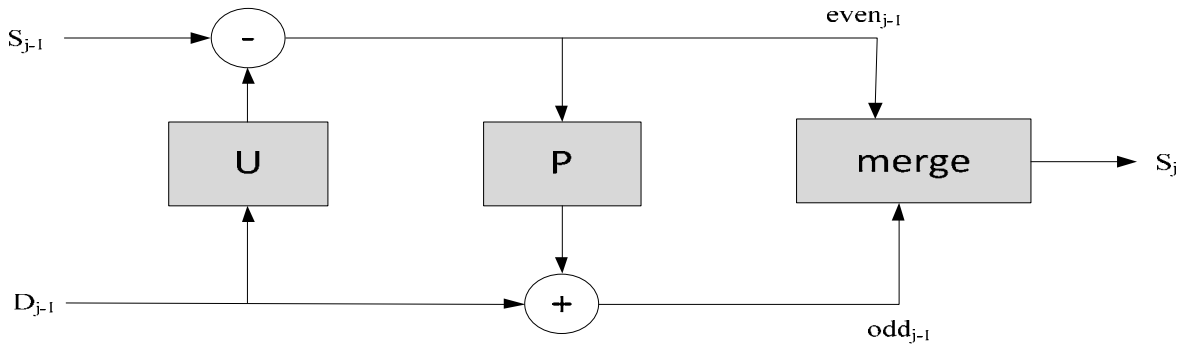


Figure 2.10: Lifting Scheme Inverse Transform

2.5.4 FAST LIFTING WAVELET TRANSFORM FILTERS

As noted earlier the lifting scheme presents an efficient way to perform Wavelet Transform. There are number of fast lifting wavelet transform filters that performs the wavelet transform by using lifting scheme. Some of them are listed in the following diagram:

Filter	Difference Equation
5/3	d[n] = d0[n] − [1/2*(s0[n+1)+s0[n])]
	S[n] = s0[n] + [1/4*(d[n] + d[n-1]) + 1/2]
2/6	d1[n] = d0[n] − s0[n]
	S[n] = s0[n] + ½ * d1[n]
	d[n] = d1[n] + [¼ * (-s[n+1] + s[n-1] + 1/2) + ½]
5/11-C	d1[n] = d0[n] − [1/2 * s0[n+1] + s0[n]]
	S[n] = s0[n] + [¼ * (d1[n] + d1[n-1]) + 1/2]
	d[n] = d1[n] + [1/16 * (s1[n+2] − s1[n+1] − s1[n] + s1[n-1]) + 1/2]
9/7-F	d1[n] = d0[n] + [203/128 * (-s0[n+1] − s0[n]) + 1/2]
	S1[n] = s0[n] + [217/4096 * (-d1[n] − d1[n-1]) + 1/2]
	d[n] = d1[n] + [113/128 * (s1[n+1] + s1[n]) + 1/2]
	s[n] = s1[n] + [1817/4096 * (d1[n] + d1[n-1]) + 1/2]

Figure 2.11: Difference Equations of the various Fast Lifting Wavelet Transform Filters

The working of all of these filters is pity simple and the following shows the data flow in the 5/3 filter.

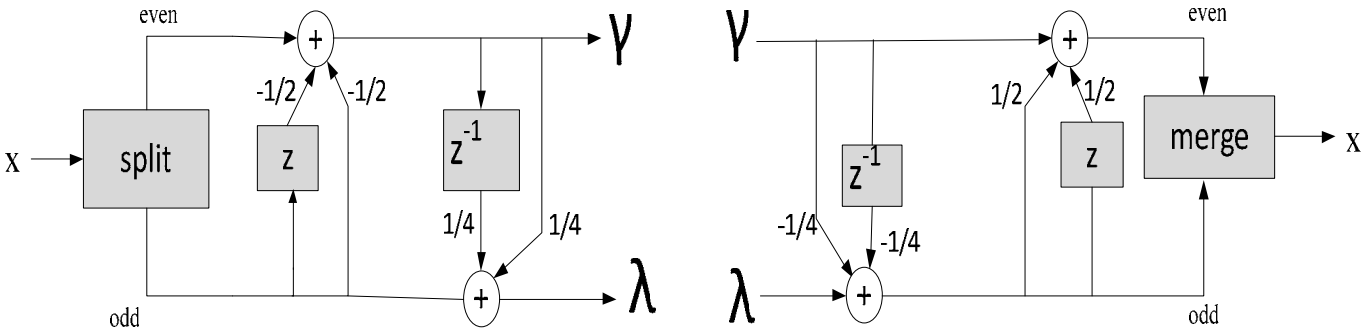


Figure 2.12: Signal Flow diagram of 5/3 filter [2]

The working of this is as an input signal x is first splitted into two data sets i.e. even indexed elements and odd indexed elements, then current element of odd data set is added to product of -1/2 and sum of current value of even and one advanced value of the even dataset which results into a high frequency component i.e. shown as gamma in the diagram. The same procedure is repeatedly until we reach the end of data sets. On the other side the operation is same but now the coefficient is 1/4 instead of -1/2 and previous value of odd data set is used rather than the advanced one. The result of this operation gives the low frequencies values that are shown here as lambda.

Chapter 3

Methodology

3.1 INTRODUCTION

Normally, while designing a general purpose processor an instruction set is designed. This instruction set consists of a different class of instructions, by class I mean different instruction types, which include:

- The memory reference instructions
- The arithmetic-logical instructions
- The branch or jump instruction
- Compound instruction

Many of the key design principles are introduced by looking at the implementation. The instruction cycle of every instruction includes the following phases:

- Instruction fetch
- Instruction decode
- Instruction execute

In addition, more concepts used to implement the Reduced Instruction Set Architecture (RISC) are discussed. A very special instruction called compound instruction is also added in the architecture whose instructions cycle is same as the other instructions but it has the capability of performing multiple arithmetic operations only in one cycle.

3.2 System Design Flow

Figure 3.1 shows a design diagram. This section only highlights that a signal processing application is usually divided into software and hardware components. The hardware design is implemented in Verilog. The design is then mapped either on custom ASICs or FPGAs. This design needs to work with the rest of the software application.

There are usually standard interfaces that enable the SW and HW components to transfer data and messages. Architecture is designed to implement the hardware part of the application. The design contains all the requisite interfaces for communicating with the part implemented in software. The HW design and the interfaces are coded in Verilog. This chapter focuses on RTL coding of the design and its verification for correct functionality. The verified design is synthesized on a target technology. The designer, while synthesizing the design, also constrains the synthesis tool either for timing or area. The tool generates a gate level netlist of the design. The tool also reports if there are paths that are not meeting the timing constraints defined by the designer for running the HW at the desired clock speed. If that happens, the designer either makes the tool meet the timing by trying different synthesis options, or transforms the design by techniques described in this book. The modified design is re coded in RTL and the process of synthesis is repeated until the design meets the defined timings. The gate level netlist is then sent for a physical layout, and for custom ASICs the design is then 'taped out' for fabrication. The field programmable gate array tools provide an integrated environment for synthesis, layout and implementation of a bit stream to FPGA[3].

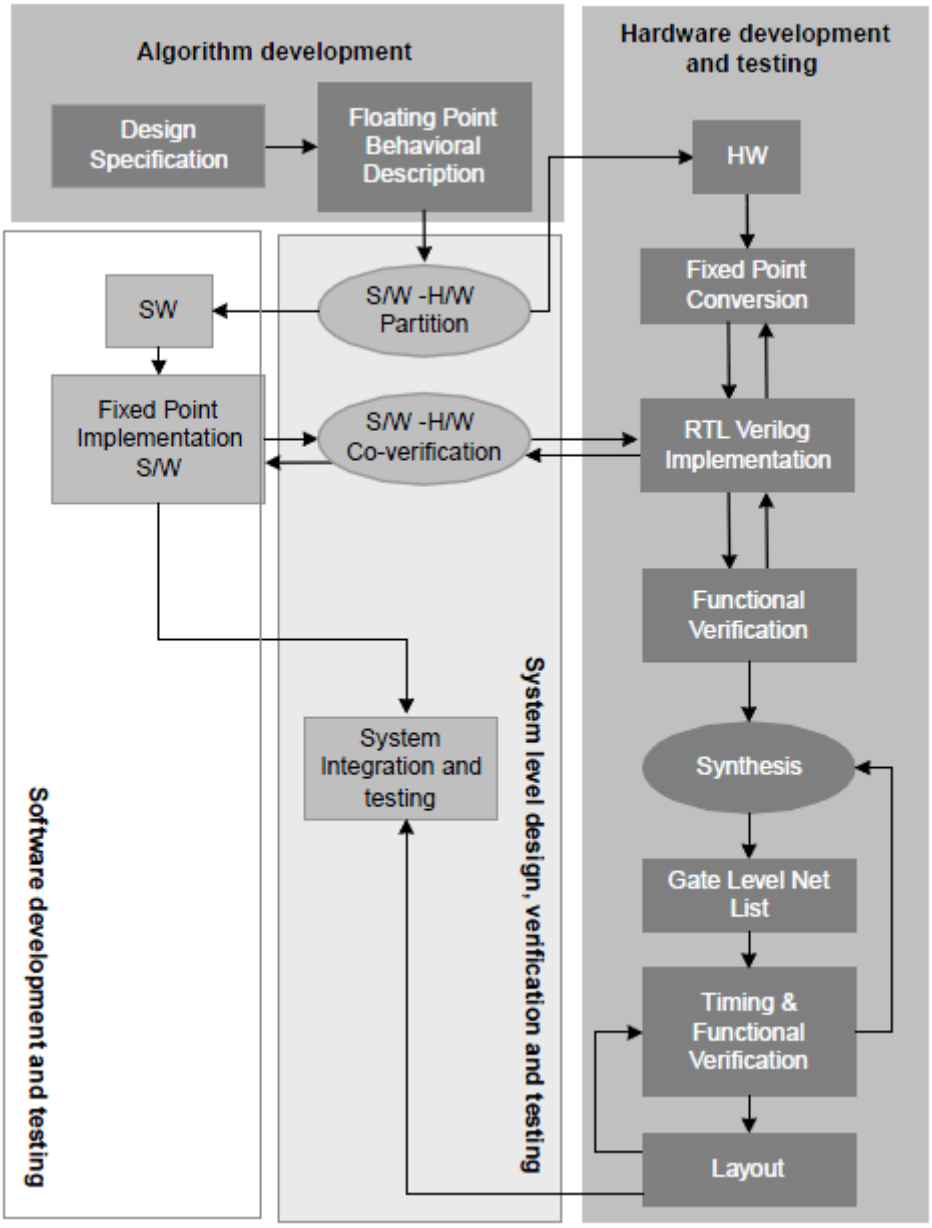


Figure 3.1: System level design components[3]

3.3 AN OVERVIEW OF DESIGN

In this chapter we looked at the core instructions require to build and RISC architecture including arithmetic-logical instructions, the memory-reference instructions, and the branch instructions.

As highlighted earlier the compound is basically the arithmetic instructions that perform multiple arithmetic operations simultaneously in one clock cycle. Because of the requirement that any signal processing task mainly consists of basic operations such as addition, subtraction, multiplication and division so it has become very easy to include such an instruction in the instruction set.

As we know all the signal processing tasks are very computational intensive i.e. much of processing needs to be done to complete signal processing tasks. For example, when Fourier Transform (FT) of a signal is to be taken then bulk of data samples needs to be processed to get the desired transformed signal. Because every sample of input signal needs to be processed to get the results so by looking at the equation of FT it can be seen that we need to do three to four basic arithmetic operation to get the output sample. If each basic operation is done in one clock cycle then it will take four clock cycles to produce an output sample in each iteration. This will not harm if the data to be processed contain few samples but unfortunately in all signal processing applications the input data is usually of thousands of thousands of samples so by considering only the Fourier Transform, the execution time of signal will be multiplied by four with the number of input samples of the signal. This is a huge difference. So a careful deliberation is done to come up with a solution that each output sample should come is each clock cycle. This will increase the throughput of the system quite remarkably. So, the compound instruction has done the job for us. To perform a bulk of data processing this compound instruction generates control signals for the processing that needs to be performed to produce the output sample after every clock cycle.

Even across different instruction classes there are some instruction classes. For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading operands from the register file. As we can see, the simplicity and regularity of the instruction set makes the execution of many instruction classes similar.

3.3.1 Clocking Methodology

Figure 3.2 shows the tow state elements surrounding a block of combinational logic, which operates in a single clock cycle: All signals must propagate from state element 1 through the combinational logic, and to state element 2 in the time of on clock cycle. The time necessary for the signals to reach state element 2 defines the length of clock cycle.

For simplicity, we do not show a write control signal when a state element is written on every active clock edge. Both the clock signal and the write control signal are inputs.

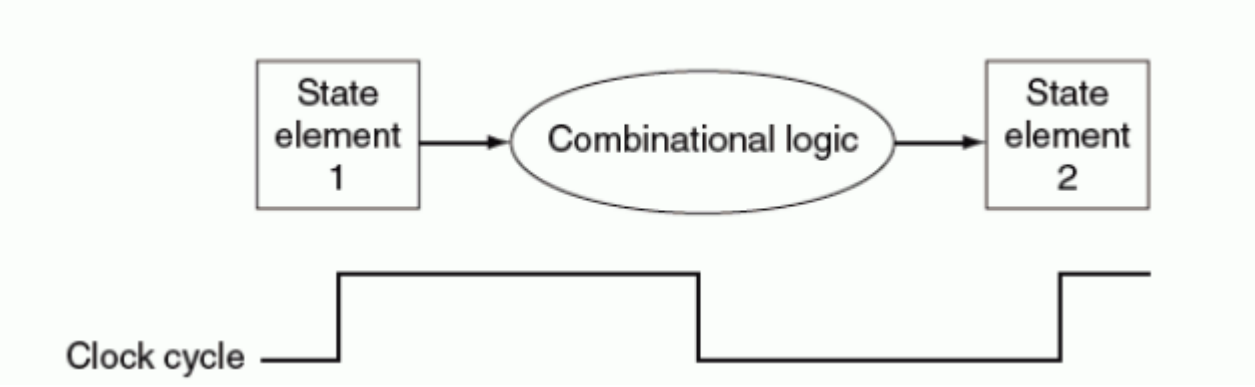


Figure 3.2 : Combinational logic, state elements and clock are closely related [10]

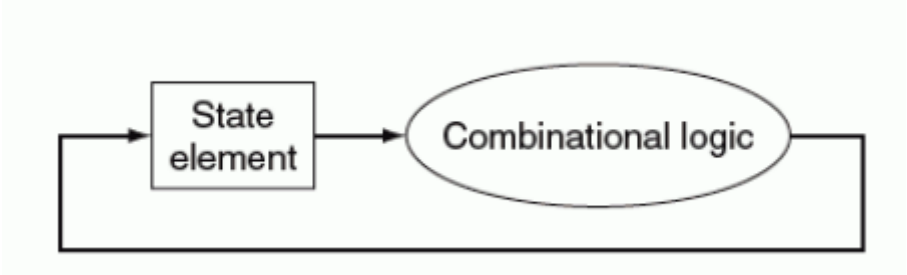


Figure 3.3: Edge triggered methodology eliminates race condition that can cause intermediate data values [10]

The figures will indicate buses, which are signals wider than 1 bit, with thicker lines. At times we will want to combine several buses to form a wider bus, for example, we may want to obtain a 32-bit bus by combining two 16-bit buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, color indicates the control signal as opposed to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

3.4 IMPLEMENTATION

3.4.1 Overview

Reduced Instruction Set Computers (RISC) are designed to have a small set of instructions. In this section we will model a simple RISC based architecture. This architecture also serves as a starting point for developing architectural variants and a more robust instruction set.

Designers make high-level tradeoffs in selecting an architecture that serves an application. Once architecture has been selected, a circuit that has sufficient performance (speed) must be synthesized. Hardware description languages (HDLs) play a key role in this process by modeling the system and serving as descriptive medium that can be used by a synthesis tool.

Figure 3.5 shows the block diagram of the proposed design. The whole design is pipelined to achieve the higher throughput. Pipelining is described in the next section.

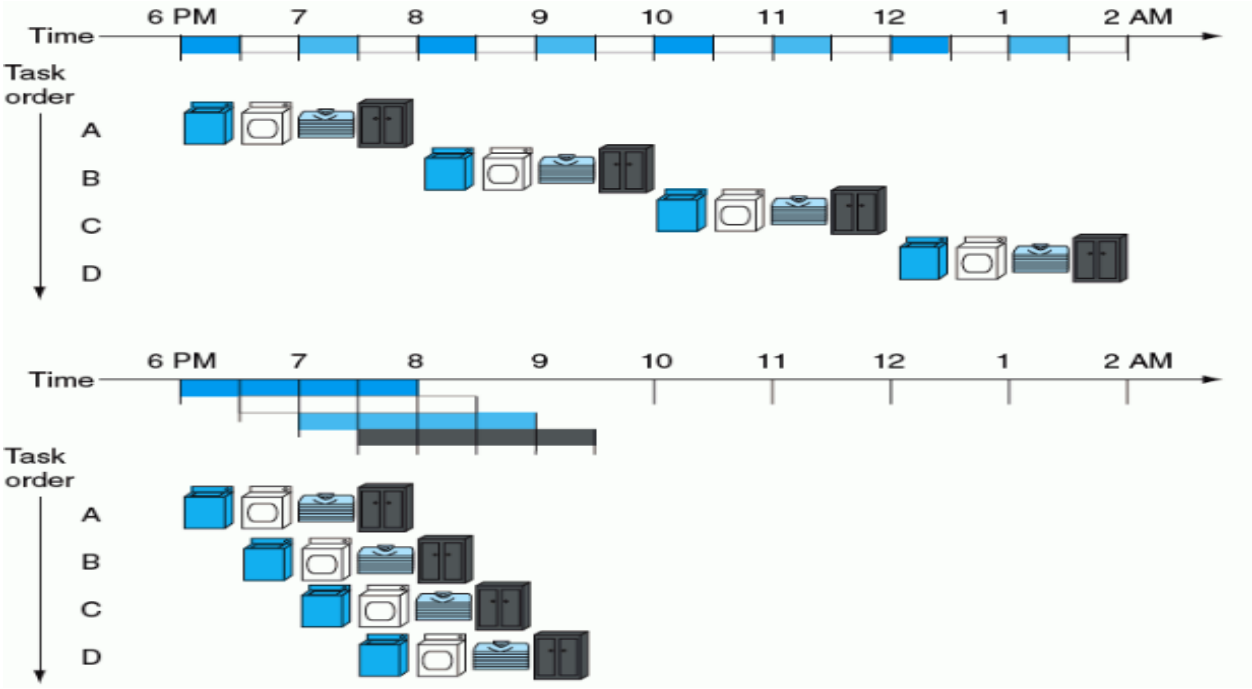
3.4.2 Pipelining

This section relies heavily on one analogy to give an overview of the pipelining terms and issues. If you are interested in just the big picture, you should concentrate on this section.

The pipelined approach takes much less time, as figure 3.4 shows. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into washer. At this point all steps called stages in pipelining, are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.

Pipelining improves throughput of our laundry system without improving the time to complete one load of laundry.

Figure 3.4: Laundry analogy to pipelining [10]



Pipelined laundry is potentially four times faster than non-pipelined, 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It's only 2.3 times faster than in above figure because we only show 4 loads. Notice that at the beginning and end of the work load in the pipelined version in above figure, the pipeline is not completely full, this starts up and wind down affects performance when the number of tasks is not large compared to the loads is much larger than 4, then the stages will be full most of the time and the throughput will increase with a factor of 4.

The same principles apply to processors where we pipeline instruction execution. The instructions of proposed processor take five steps:

1. Fetch instruction from memory
2. Decode the instruction
3. Read operands from the register file
4. Execute the instruction
5. Write the result back to register file

These are the five pipeline stages of the proposed processor.

Following is the pipeline speed up formula

$$\text{Time between instruction}_{\text{pipelined}} = \frac{\text{time between instructions}_{\text{non-pipelined}}}{\text{Number of pipeline stages}}$$

The formula suggests that a five-stage pipeline should offer nearly a five fold improvement over 800 ps non-pipelined time, or a 160 ps clock cycle. The example shows, however, that the stages can be imperfectly balanced. In addition, there is some overhead involved in pipelining, the source of which will be clearer shortly. Thus, the time per instruction in the pipelined processor will exceed the minimum possible.

Of course, this is because the number of instructions is not large what would happen if we increased the number of instructions? We could extend the previous figures to much more instructions.

3.4.3 Designing Instruction Sets for Pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the instruction set which is designed for pipelined execution.

Third, memory operands only appear in load and store instructions. This restriction means we can use the execute stage t calculate the memory address and then access memory in the following stage.

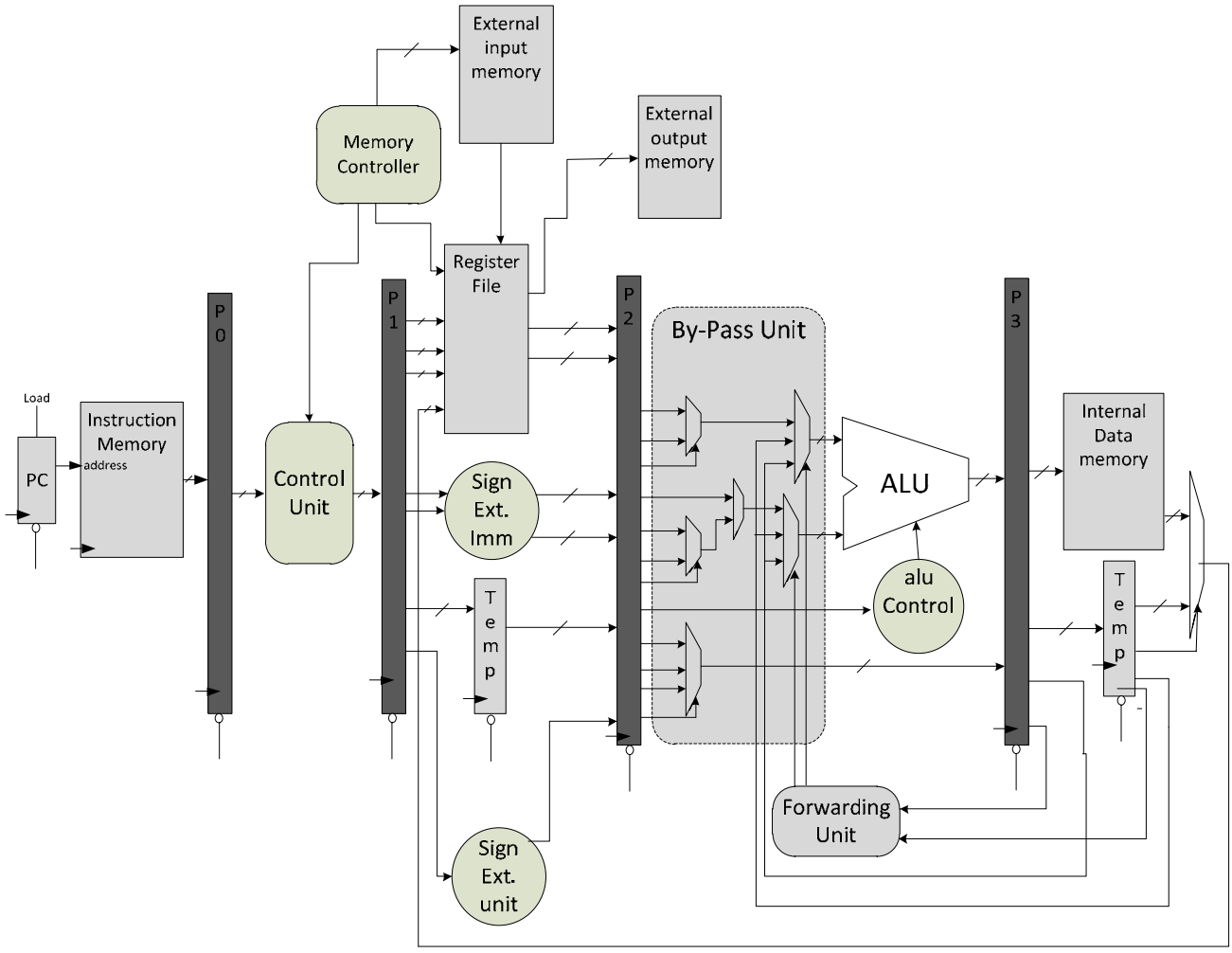


Figure 3.5: Block Diagram of Proposed Processor

3.4.4 Building a Data Path

Let's start by looking at which data path elements each instruction needs. When we show the data path elements, we will also show their control signals.

Figure 3.6 shows the adder, which is combinational in nature, is not shown explicitly in the above block diagram of the design but shown implicitly that PC is incremented whenever *inc_PC* signal is high.

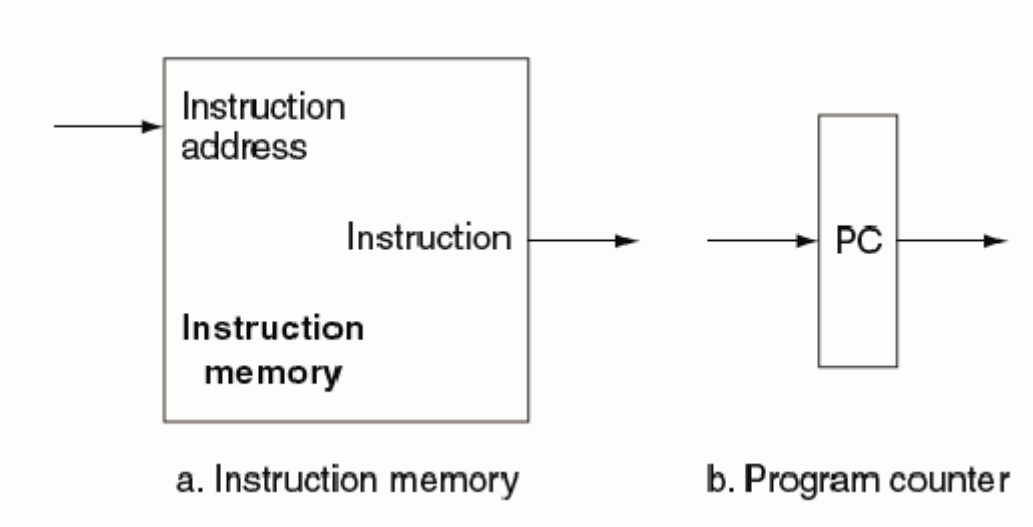


Figure 3.6: Two elements of data path: instruction memory and PC

For each data word to be read from the register, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.

Figure 3.7 shows the ALU, which takes two 32-bit inputs in case of normal instructions and three 32-bits inputs in case of compound instruction and produces a 32-bit result along with the status signals i.e. zero, carry out, negative etc. The ALU control signal is described later in this chapter.

Thus we need both the register file and ALU (for address calculation) from figure 3.7.

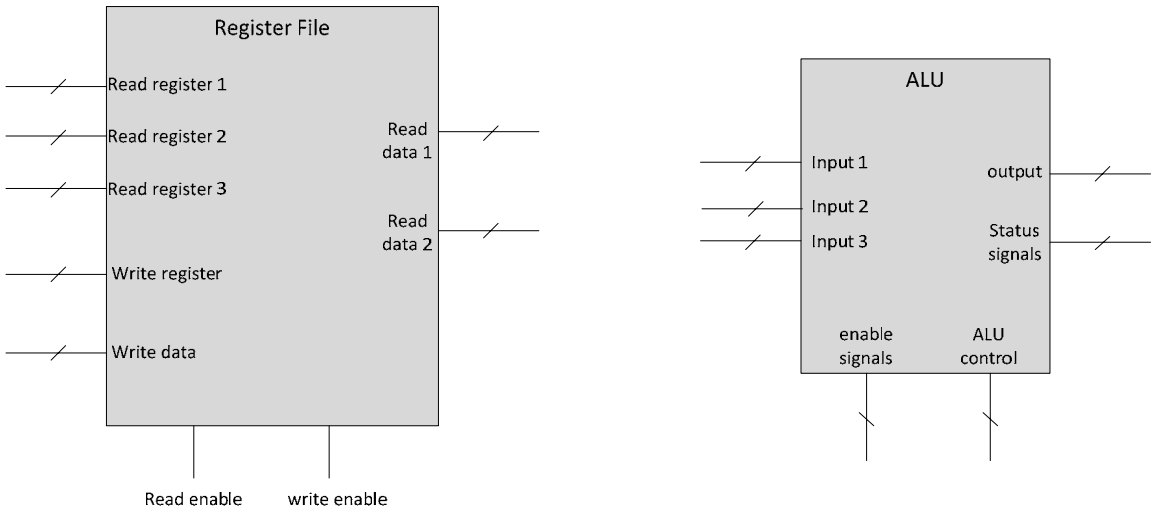


Figure 3.7 : Register file and ALU of data path

3.4.5 Components of Data Path

Data path is composed of following components

- Program Counter
- Instruction Memory
- Control Unit
- Memory controller
- Register File
- Sign Extension Unit
- ALU
- ALU control
- Forwarding Unit
- Internal data Memory
- Bypass Unit
- Input Memory
- Output Memory

Following is detailed description of each of these components.

3.4.5.1 Program Counter (PC):

Following is block diagram of program counter.

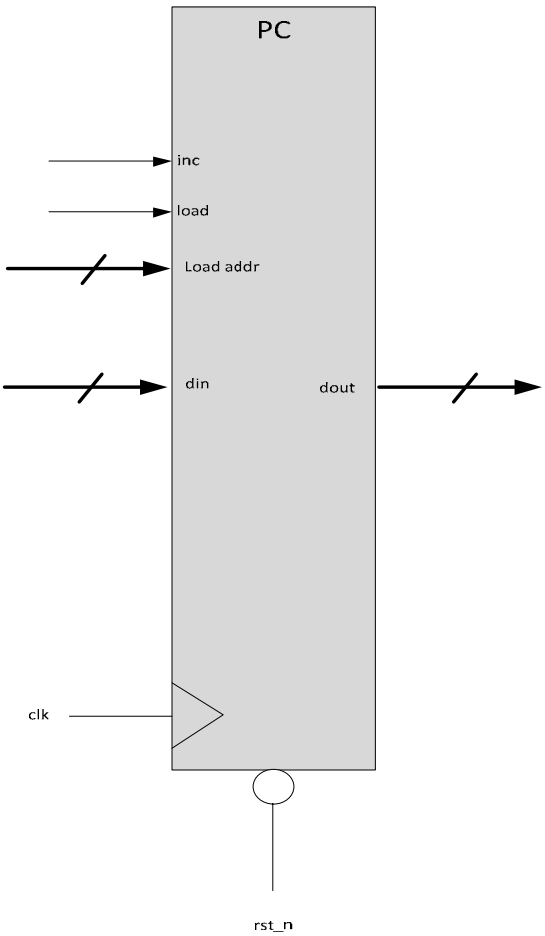


Figure 3.8: Program Counter (PC)

The program counter (PC), also known as the instruction pointer (IP) and sometimes called the instruction address register is a processor register that indicates where a computer is in its program sequence. For the purposes of proposed processor it has it has 5 inputs and one output. 4 of the total inputs are the control inputs i.e. load, increment, clock and reset. Because the reset is active low resent so the PC will be reset if the value of reset i.e. rst_n is 1 on active clock edge. If the load signal is asserted, it means that the execution sequence has to change and now the PC will be loaded to some other address indicated by "load addr" into the PC on the next active clock edge. If there is no jump to be taken, then by asserting the signal inc the PC will increment on every active clock edge. Clk denotes the

clock signal and din and dout simply denotes the input and output to the PC. The length of the PC is customizable as the parameter passed to the PC module will decide the length of the PC.

3.4.5.2 Instruction Memory:

Instruction Memory stores the instructions that need to be executed to complete a task. Following diagram shows the block level diagram of Instruction Memory of the proposed processor.

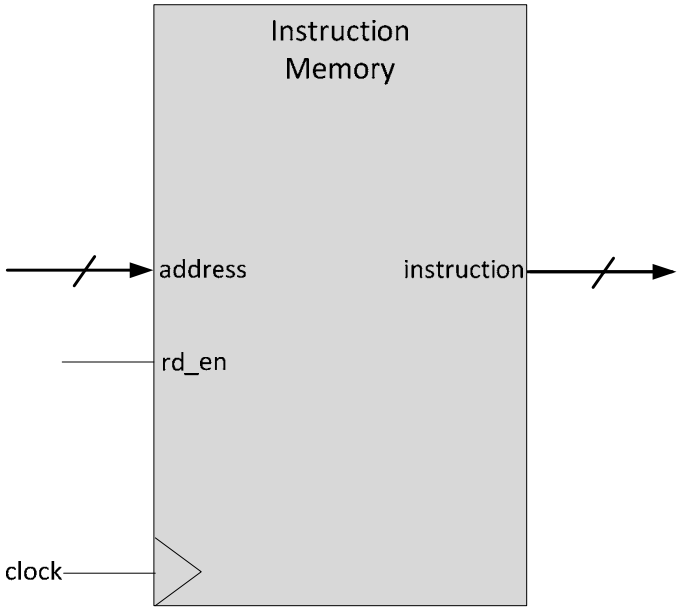


Figure 3.9: Instruction Memory

It has two control inputs i.e rd_en (read enable) and clock and one data input and one output. The address input of the instruction memory is connected to the output of the program counter (PC) described above.

3.4.5.3 Control Unit:

The control unit manages all the activities of the processor. In a way, it controls everything happening in the processor.

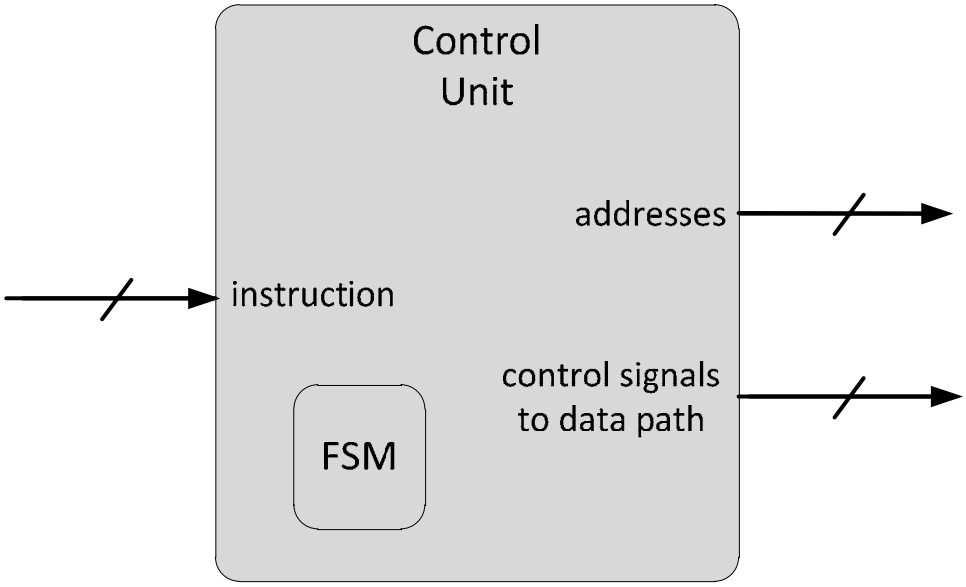


Figure 3.10: Control Unit

The control unit of the proposed processor is micro-program based. Part of the instruction called op-code (operation code) decides what signals to generate for the data path. Based on the instruction op-code all the control signals of the rest of the data path are generated. There is also a hardwire FSM that only communicates with the memory controller described later in this chapter. This FSM is shown in the following diagram:

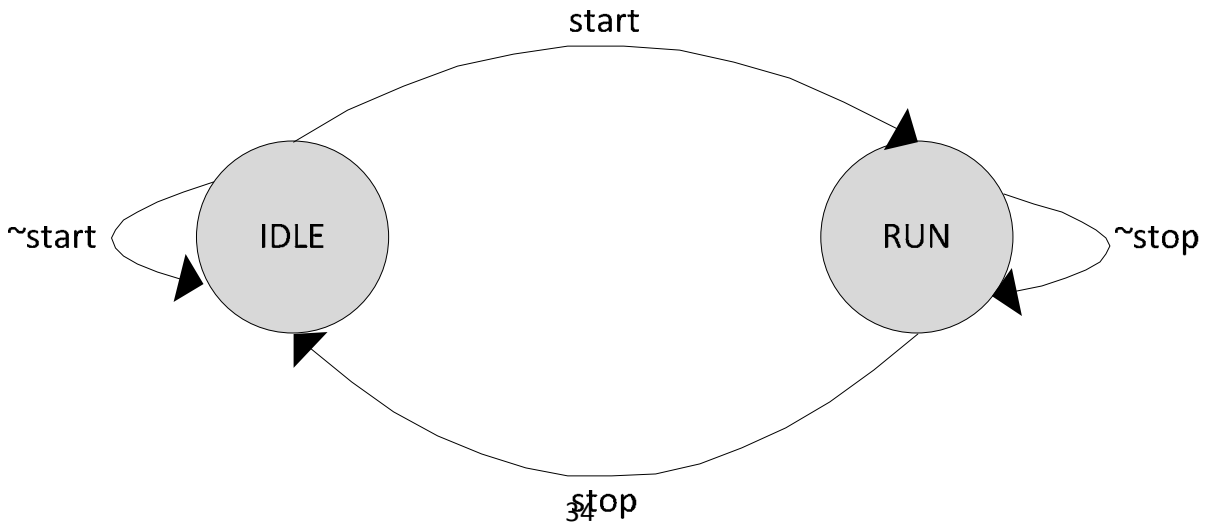


Figure 3.11: Control Unit FSM

The state machine shown above consists of two state IDLE state and RUN state. This state machine works in a way that the control unit waits for the start signal from memory controller. As soon as that memory controller sends this signal the control unit switches to the RUN state and control unit starts the required processing by fetching and executing instructions. This execution stops as soon as the stop signal asserted and then state machine to IDLE state.

It is useful to review the formats of all the instruction classes' i.e.

- The memory reference instructions
- The arithmetic-logical instructions
- The branch or jump instruction
- Compound instruction

Figure 3.12 shows these formats.

Each instruction is of 32 bits

- **R-type:**

Op code	Rs	Rt	Rd	Shift	func
31:26	25:21	20:16	15:11	10:6	5:0

- **Load and store instructions:**

Op code	Rs	Rt	address
31:26	25:21	20:16	15:0

- **Compound Instruction:**

Op code	Rs	Rt	Rd	Shift	Rs2

- **Branch instruction:**

Op code	Rs	Rt	address
31:26	25:21	20:16	15:0

- **Instructions having immediate operands**

Op code	Immediate1	Immediate2	rd	func
31:26	25:18	17:11	10:6	5:0

Op code	Immediate1	rs	rd	func
31:26	25:18	17:11	10:6	5:0

Op code	rs	Immediate2	rd	func
31:26	25:18	17:11	10:6	5:0

3

Figure 3.12: Instruction Formats

Some of the observation regarding these formats are as under:

- The op field, also called the opcode, is always contained in bits 31:26.
- The 16-bit offset for branch, load and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus we will need to add a multiplexer to select which field of the instruction is used to indicate the register number to be written.

Table 3.1 shows 12 control signals. It is useful what these 12 control signals do informally before we determine how to set these control signals during instruction execution.

Signal Name	Function
regdst	The register file is to be written with the rd field of instruction
alusrc	The 2nd ALU operand is the sign extended lower 16 bits of instruction

memtoreg	Indicates data will move from memory to register file
Regwrite	The register on the Write register input is written with the value on write data input
Memread	Data will be read from memory
Memwrite	Internal data memory contents designated by the address input are replaced by the value on the write data input of internal data memory
memtoReg	The value fed to the register write data input comes from the internal data memory.
Branch	Indicates when the branch is to be taken
Aluop	Specifies the which operation ALU should perform
regFile_rd_en	Indicates now register can be read when asserted
mux_sel_mov_instr	Selects line of the MUX for move instruction
imm_MUX1_sel	Select line for the MUX when first ALU operand is immediate operand
imm_MUX2_sel	Select line for the MUX when second ALU operand is immediate operand

Table 3.1: Control signals generate by control unit

Now that we have looked at the function of each of the control signals, we can look at how to set them. With the information contained in table x we can design the control unit logic, but before we do that.

Following are the steps that how an R-type instruction is executed, the pipeline stage at which these steps occur are also indicated.

- The instruction is fetched and PC is incremented (Instruction Fetch stage)
- The address for register file operands and all the required control signals are generated during this step (instruction decode stage).
- Operands are read from the register file (Operands fetch Stage).

lw R1, R2 + offset (load contents of data memory at location (R2 +offset) into R1)

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of the following table x.

Instruction type	Regdst	Alusrc	memtoReg	Regwrite	Memread	Mem write	Branch	aluOp 0	aluOp 1
R-format	1	0	0	1	0	0	0	0	1
Lw	0	1	1	1	1	0	0	0	0
Sw	0	1	X	0	0	1	0	0	0
Branch	X	0	X	0	0	0	1	1	0
compound	1	0	0	1	0	0	0	0	1
Move	x	x	x	1	x	x	x	x	x

regFile_rd_en	mux_sel_mov_instr	imm_MUX1_sel	imm_MUX2_sel
1	X	0	1
1	X	0	1
1	X	0	1
1	X	0	1
1	X	0	1
1	1	x	x

Table 3.2: Value of control signal depends entirely on the instruction opcode

3.4.5.4 Memory Controller:

Another very important component of the processor is the Memory controller. It allows direct access of memory to CPU registers without intervention of the main CPU. By the inclusion of this controller into the design the efficiency of the processor have increase increased many time because without memory controller the processor has to first fetch data operands from external memory to register file and then process these operands and return back the result when the processing completes. So in that way much of the CPU time was wasted by fetching data operands from external memory to register file and writing the results back from register file to external memory. So as a solution of the extra work the CPU had to do to retrieve operand and write back results, now the memory controller takes care of the fetching of operand and storing of results to the external memory and it fully controls the communication between external memory and register file of the processor. After the inclusion of memory controller in the design the CPU only does the actual processing i.e. to execute instruction the extra work of transfer of data is now transferred to memory controller. Start address and stop addresses are sent as parameters to the memory controller. Start address specifies where to start reading the memory and stop address specifies where to stop reading the memory. All the memory locations between start and stop address are read and written to the register file.

Figure 3.13 shows the block diagram of memory controller

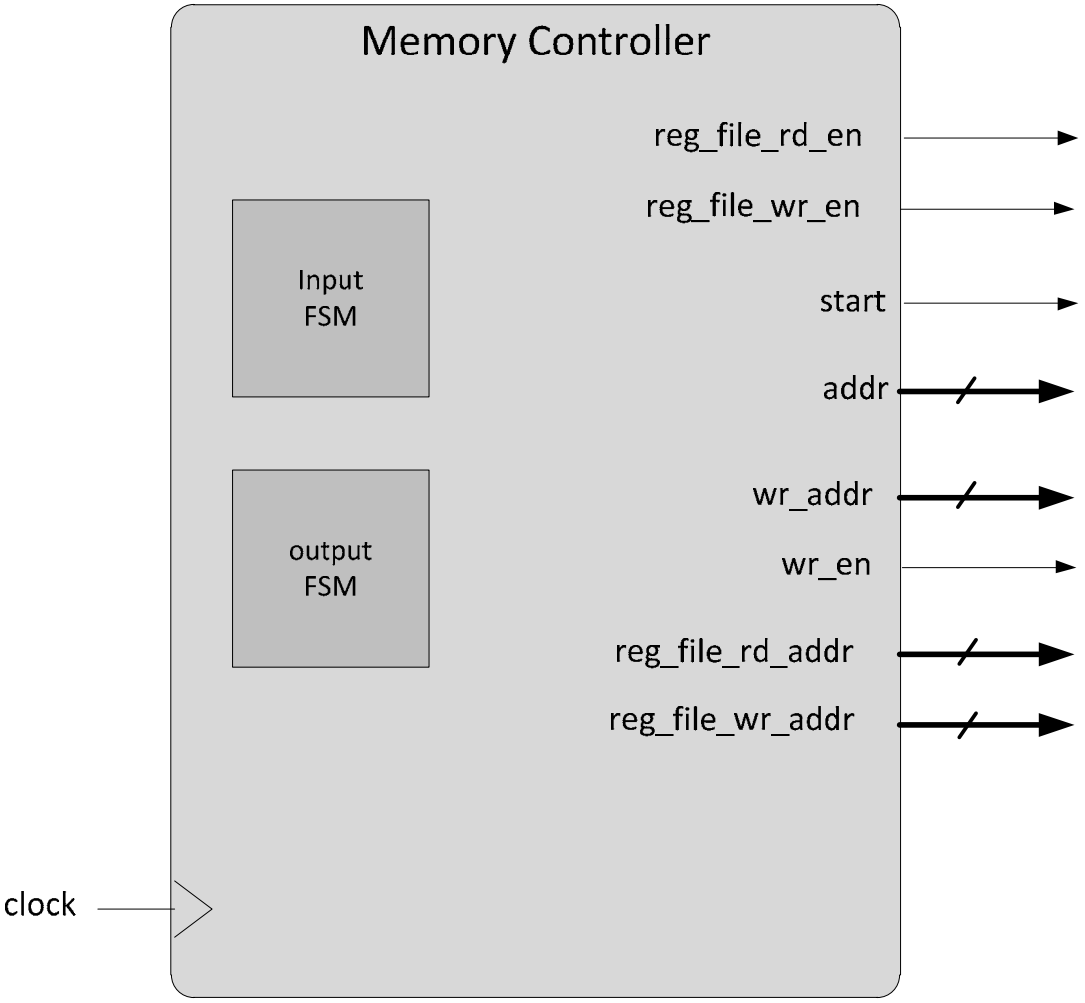


Figure 3.13: Memory Controller

The memory controller consists of two state machines

- Input State Machine
- Output State Machine

Input state machine controls the flow of data from external memory to the register file. Figure 3.14 shows the input state machine of memory controller

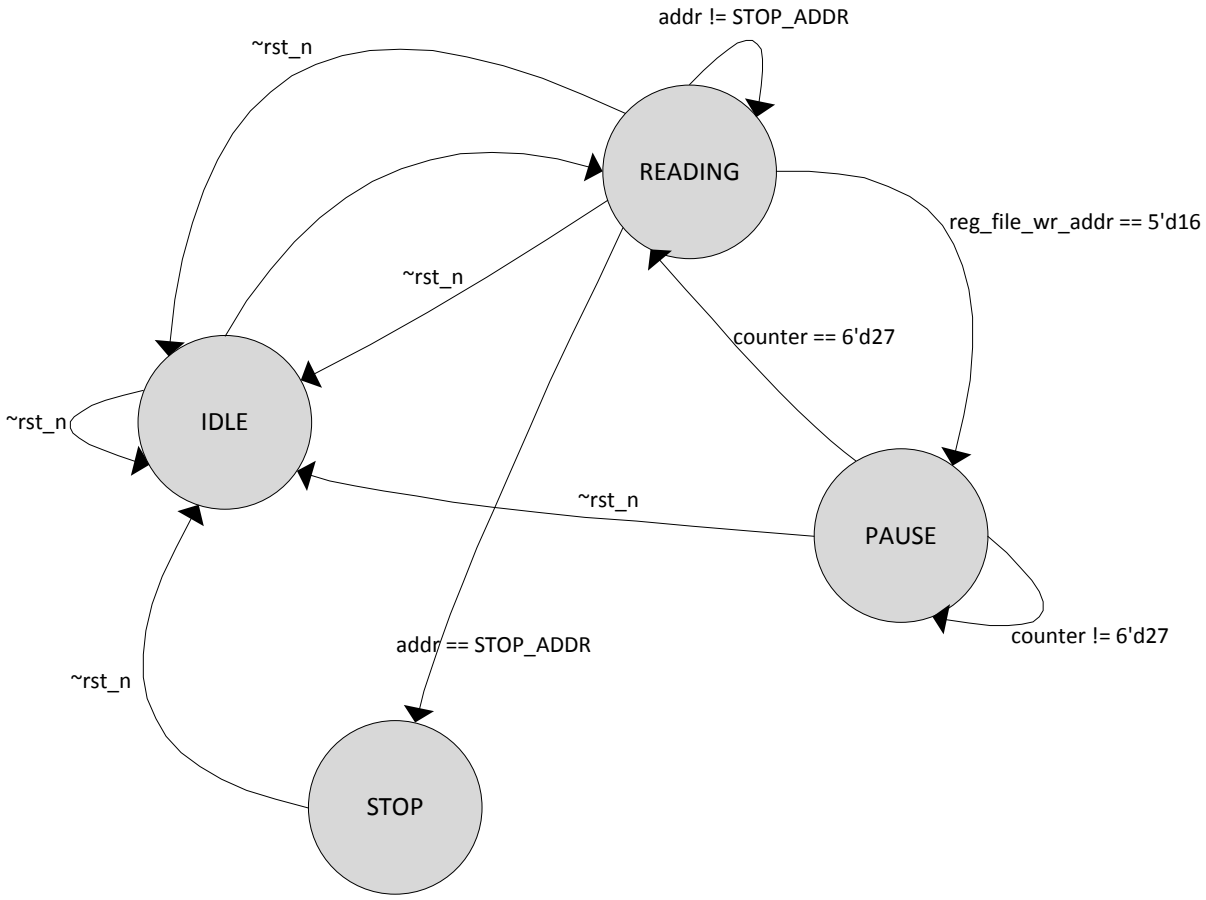


Figure 3.14: Input FSM of memory controller

On the first active clock edge after the reset signal is de-asserted; the machine will go into the READING state unconditionally. The FSM will remain in the READING state until the condition addr != STOP_ADDR remains true, as soon as this condition

becomes false the FSM switches to PAUSE state. STOP_ADDR is a parameter passed to the DMA module. Similarly, the FSM will remain in the PAUSE state until the condition counter != 27 remains true and switches to READING state as soon as this condition becomes false. There is no transition of FSM from PAUSE state to STOP state. The FSM can only come in the STOP state through the READING state. It is understood that whenever reset signal asserted, the FSM will switch to IDLE state from any state.

Output state machine controls the flow of data from register file to the external memory. Figure 3.15 shows the output state machine of DMA controller

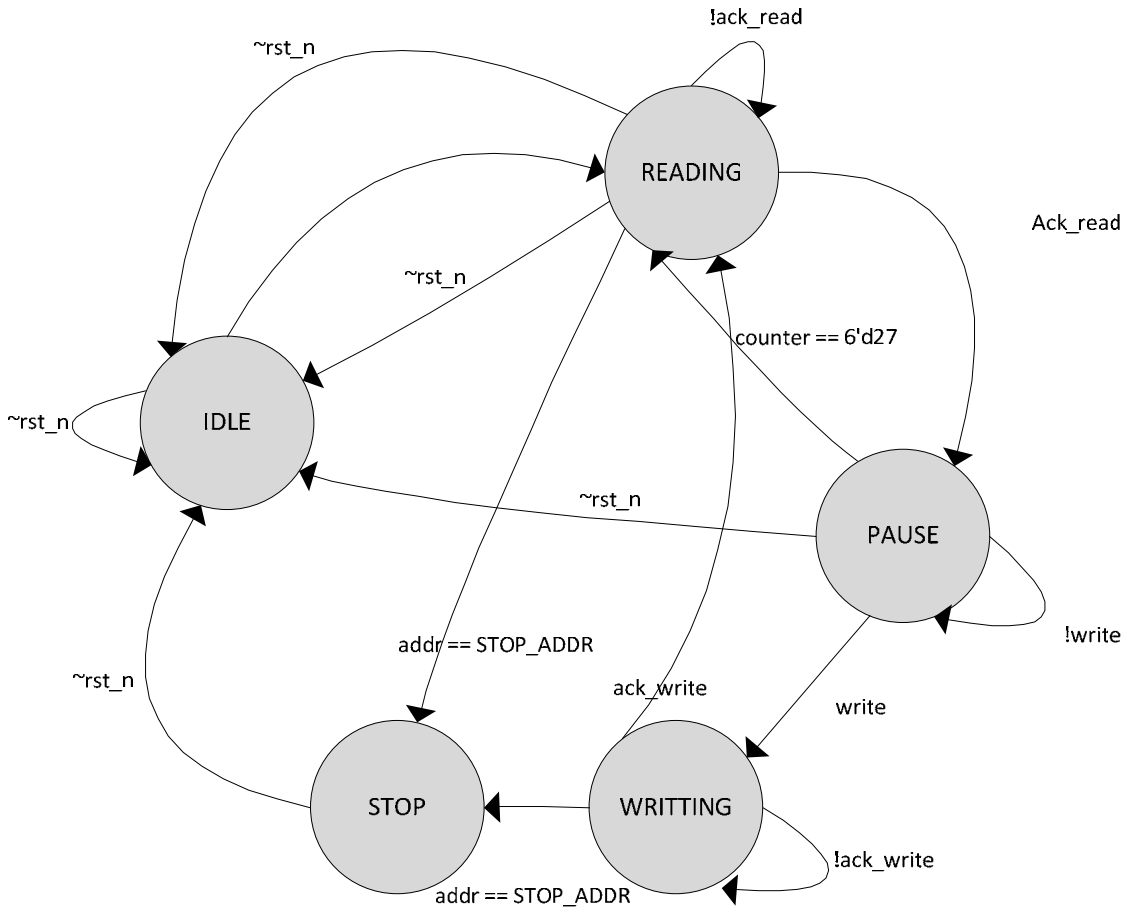


Figure 3.15: Complete FSM of Memory Controller

On the first active clock edge after the reset signal is de-asserted; the FSM will switch to the WAITING state because FSM will wait for the results computation, as soon

as the results are available, FSM is asked to switch to the WRITING state and write back the results into the external memory. While in waiting state, the FSM waits for the value of counter signal to be 27, as soon as the value of counter reaches 27 the FSM switches to WRITING and starts writing results into the external memory. The FSM keep on writing until the value of the rd_addr signal reaches value 7 and then the machine again switches to WAITING state. Meanwhile, whenever the reset signal is asserted the FSM will switch to IDLE state.

3.4.5.5 Register File:

In RISC based architectures, the role of register file is very important in a way that the CPU on access the data operands from the Register File. There is no direct access of the register file CPU to memory. So, every operand that needs to be processed by CPU must reach it through the register file because the CPU can only reach the register file only, nothing else. The design of register file is also a very important perspective of the overall design. So a very careful deliberation has been made while designing the register file of the proposed processor.

Figure 3.5 shows the register file, here is detailed description of the design of the register file is presented. The register file module basically consists of 3 actual register file i.e. register file 1, register file 2 and register file 3. Register file 1 is of size 32x32 and remaining two register files are of size 16x32. Register file 1 is dual ported i.e. it has one read port and one write port, and it can be read and written on the same active clock edge. From the remaining two register files, one register file is used for reading and the other is used for writing purposes.

At the start of processing, the register file is populated with data from the external memory. During the same time the register 2 is also populated with the first 16 samples of the data as only 16 samples of data from external memory are input to the register file 1 and register file 2. The results of the processed samples are stored into the register file 3 and the DMA controller starts sending data from register file 2 to the external output memory. Once reading and writing of data is complete, the read and write enable signals of all the register files are de-asserted.

Figure 3.16 shows the block diagram of register file module.

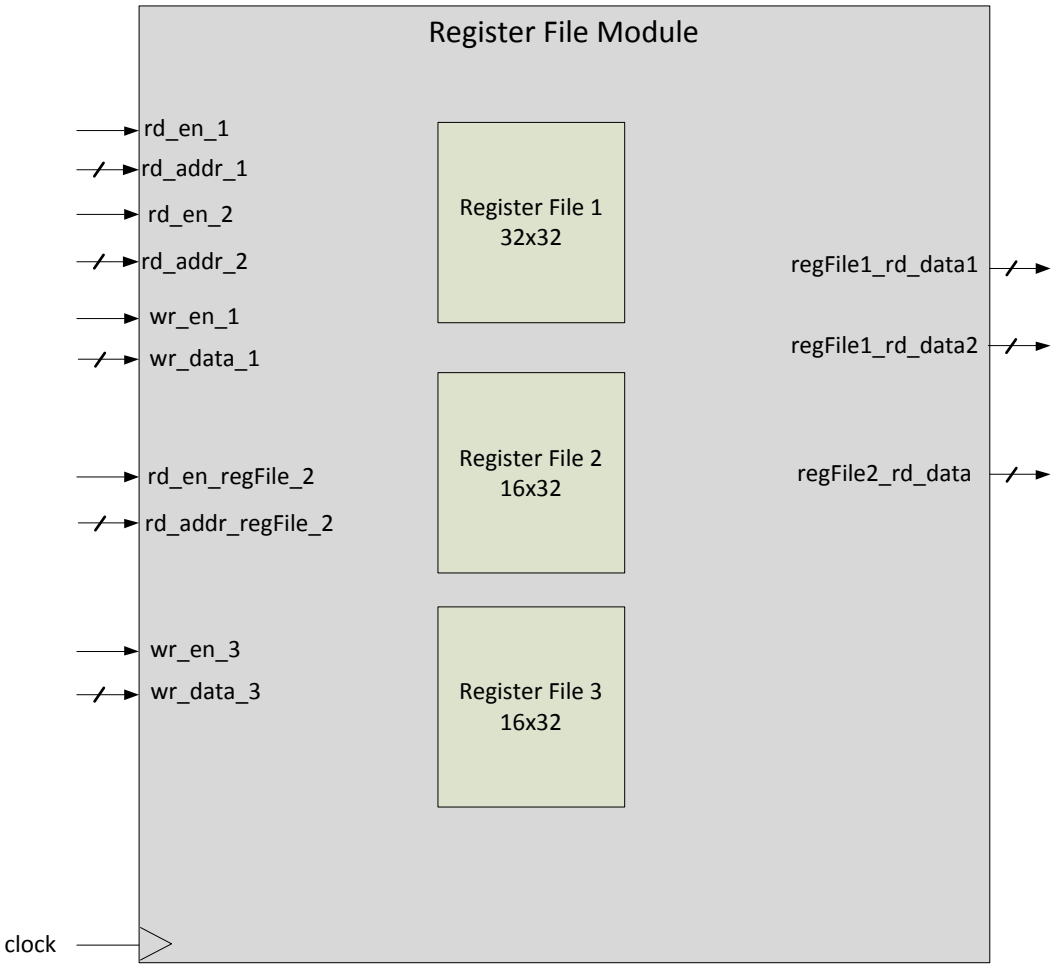


Figure 3.16: Register File Module

As seen from the above diagram the register file module consists of three register files, the multiple register files are used to increase the efficiency of the processor. With the single register this efficiency could not have been achieved.

3.4.5.6 Sign Extension Unit:

The Sign extension unit is needed to implement the load and store instruction. It generates the offset that is added with the one register operand indicated by the instruction field. Following show the sign extension unit

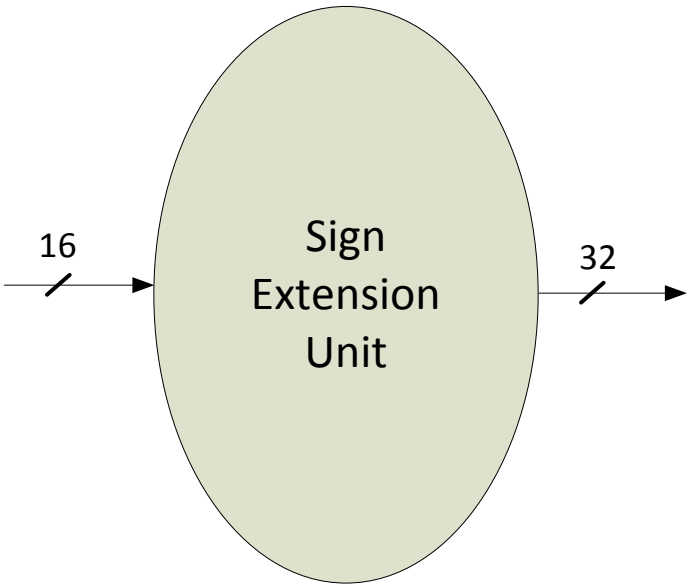


Figure 3.17: Sign Extension Unit

Sign extension unit takes 16-bit input and generate a 32-bit decimal equivalent of the input by replicating the sign bit of the input 16 times.

3.4.5.7 Arithmetic and Logic Unit (ALU):

Now let's have discussion on the most important part of the processor called Arithmetic and Logic Unit (ALU). This is a part of the processor that performs all the arithmetic and logical computations. This is surely the brain of the processor as execution of any instruction is not possible without ALU. The ALU carries out all the computations of the processor. There are four major blocks in the ALU i.e. adder, subtracter, multiplier and shifter. As the length of the operands of these components is customizable and parameterized but for our purposes I have kept it 32-bits wide. That is, it takes 32-bit inputs and generates an output of 32 bits. For multiplication two 32-bit inputs produces a result of 64 bits whose 32 most significant bits are truncated to make it of 32 bits. Figure 3.18 shows the block diagram of ALU.

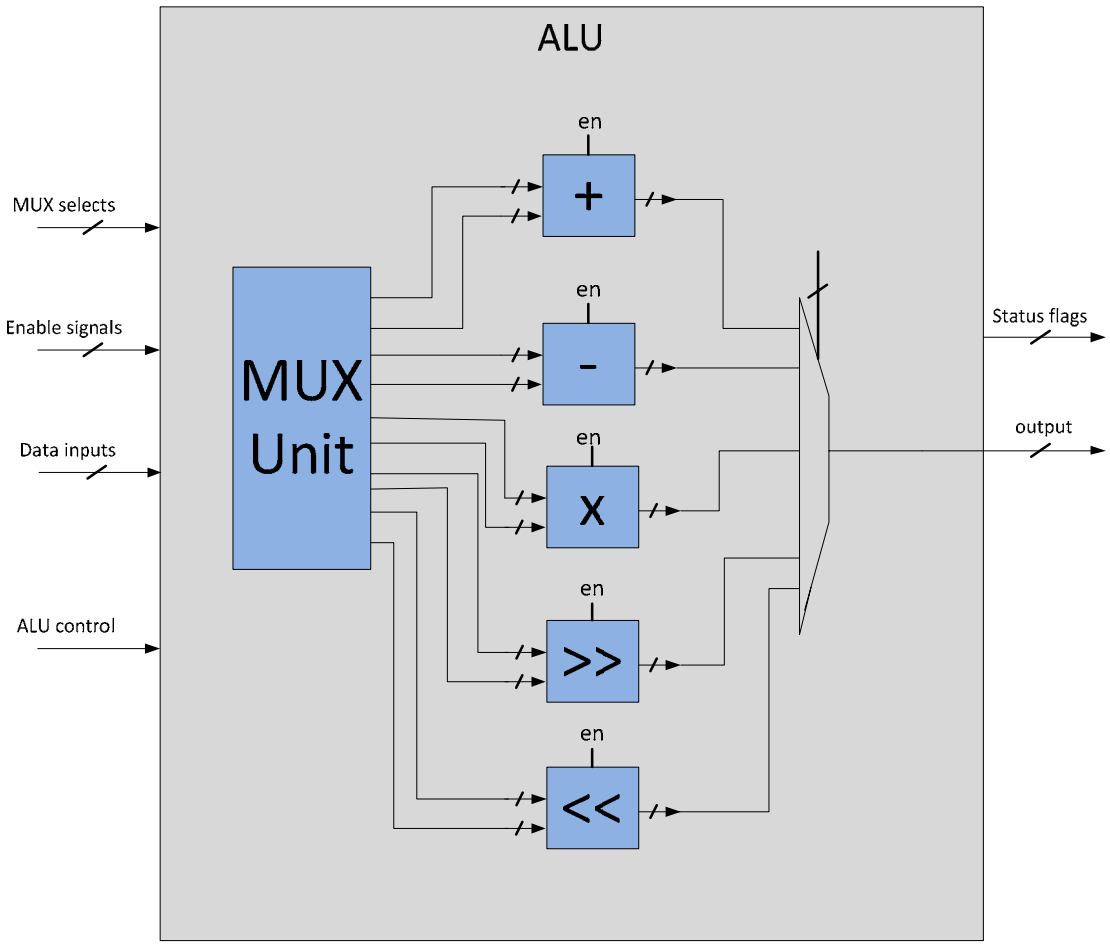


Figure 3.18: Arithmetic and Logic Unit (ALU)

Above figure shows the micro components of the ALU. Following is the brief description of each of them.

Computational block i.e. adder, subtracter, multiplier and shifter are shown in the above diagram along with their enable signals. For normal instruction, only of of these components will be active during each clock cycle and for compound instruction multiple block will be active during each clock cycle and the output multiplexer will decide the final output.

MUX unit shows a bunch of multiplexers that generates the required inputs to the computational blocks. Based on the multiplexer selects, each input of the computational block is forwarded to it through the MUX unit. For each input to each computational block there is a multiplexer. Also the output of one computational block (adder, subtracter, multiplier, shifter) could be input to the other block so the outputs of these computational blocks are also the inputs of the multiplexers.

Furthermore, the output of the ALU comprises of data output and status flags. Status flags include carry out, zero flag and negative flag. Based on these flags several actions could be taken and hence these are very important to show and cannot be eliminated from the design. For the purposes of the clarity of the diagram the data inputs are not explicitly shown, the data inputs signal includes the data operands and the shift number which specifies if a shifter is to be used then how much it will shift the desired number.

ALU control signal specifies what operation the ALU has to perform in case of normal instruction. Depending upon the value of lower 6-bits of the instruction.

3.4.5.8 ALU Control:

Table x shows the operation of the ALU in response to the alu control lines in case of normal instruction mode.

ALU Control Line	Function
000	ADD
001	SHIFT LEFT
010	SUB
011	MULTIPLY
100	AND
101	SHIFT RIGHT
110	OR
111	Ex-OR

Table 3.3: ALU control lines and their function

ALU control signal specifies what operation the ALU has to perform in case of normal instruction. For branch instruction, the ALU must perform subtraction. Figure 3.19 show the diagram of ALU control, the input to the alu control unit is the concatenation of lower 6 bits of instruction and the ALUOp field generated by the main control unit depending upon the class of the instruction.

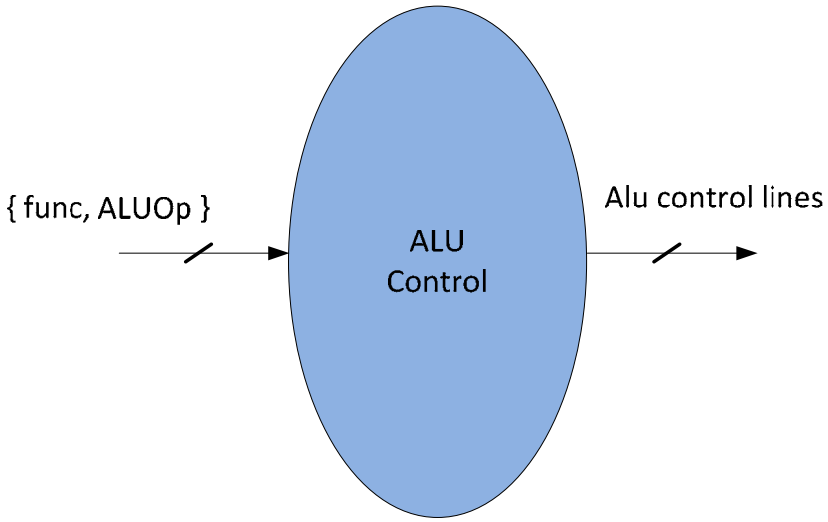


Figure 3.19: ALU control unit

For completeness, the relationship between the ALUOp bits and the instruction opcode is also shown.

Instruction opcode	ALUOp	Instruction Operation	Func Field	Desired ALU action	ALU control input
LW	00	Load word	xxxxxx	Add	000
SW	00	Store word	xxxxxx	Add	000
Branch	10	Branch	xxxxxx	Subtract	010
R-type	01	Add	100000	Add	000
R-type	01	Subtract	110000	Subtract	010
R-type	01	Multiply	111000	Multiply	011
R-type	01	AND	101000	And	101
R-type	01	OR	110001	OR	110

Table 3.4: Setting of ALU control line depending upon ALUOp and func field

Because in many instances we do not care about the values of some of the inputs and to keep the tables compact, we also include don't care terms. A don't care term is represented by x, indicates that the output doesn't depend on the value of input corresponding to that bit position.

3.4.5.9 Forwarding Unit:

This unit is incorporated to overcome data hazards in result of the pipelining. Although the pipelining improves the performance of the system significantly, but if not handled properly, it can cause significant malfunctioning of the system. If all the instructions of a program are independent i.e. the no instruction uses the result of any immediate previous instruction then the pipelining will work fine for you without any extra effort, but if the subsequent instructions of a program a dependent upon previous ones then this can lead to wrong result as in pipelined design next instruction enters the pipeline on every clock edge and results of each instruction are written back few cycles

later. So this forwarding unit caters such situation that what if some instruction requires the result of the previous instruction but that result is not written on to the specified location when the next instruction reads it as an operand. Figure 3.20 shown the block diagram of the forwarding unit:

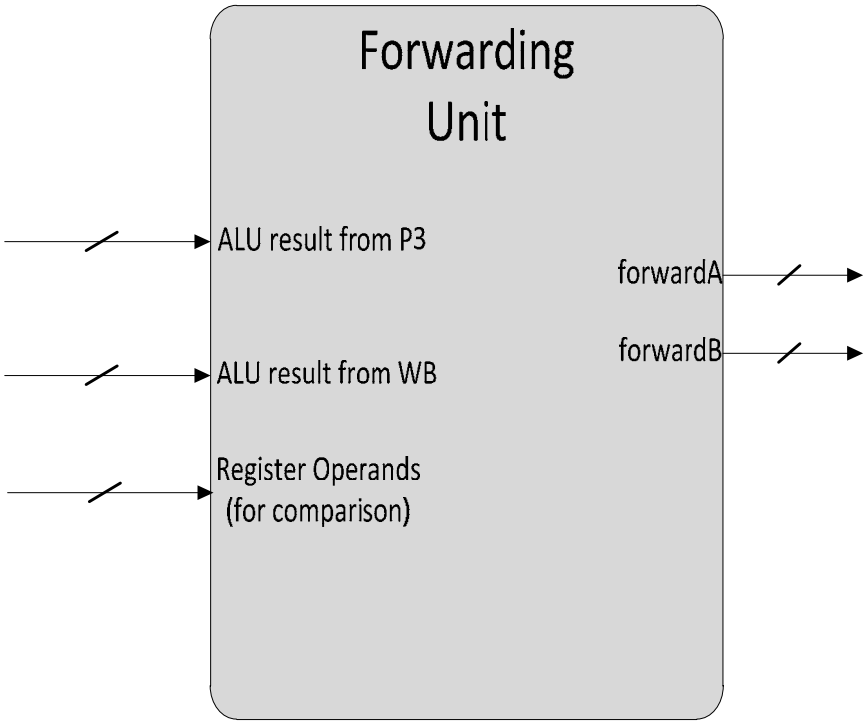


Figure 3.20: forwarding unit

Let's look at a sequence of instructions which are dependent upon each other i.e.

```
sub   R2, R1, R3
and   R12, R2, R5
or    R13, R6, R2
add   R14, R2, R2
sw    R15, 100(R2)
```

The last four instructions are all dependent on the result of register R2 of the first instruction. If register R2 had the value 10 before the subtract instruction and -20 after wards, the programmer intends that -20 will be used in the following instructions that refer to register R2.

How would this sequence perform with the pipeline of the proposed processor? Figure 3.20 illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of figure 3.20 shows the value of register R2, which changes during the middle of clock cycle 5, when the sub instruction writes its result.

Figure 3.20 shows that the values read for register R2 would not be the result of the sub instruction unless the read occurred during clock cycle 5 or later. Thus the instructions that would get the correct value of -20 are add and sw, the and and or instructions would get the incorrect value 10! Using this style of drawing such problems become apparent when a dependence line goes backwards in time.

But look carefully at figure 3.20: when is the data from the sub instruction actually produced? The result is available at the end of the Execution stage or clock cycle

3. When is the data actually needed by the and and or instructions? At the beginning of the execution stage, or clock cycles 4 and 5, respectively. Thus, we can execute this segment without stalls if we simply forward the data as soon as it is available to any units that need it before it is available to read from the register file.

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the execution stage, which may be either an ALU operation or an effective address calculation.

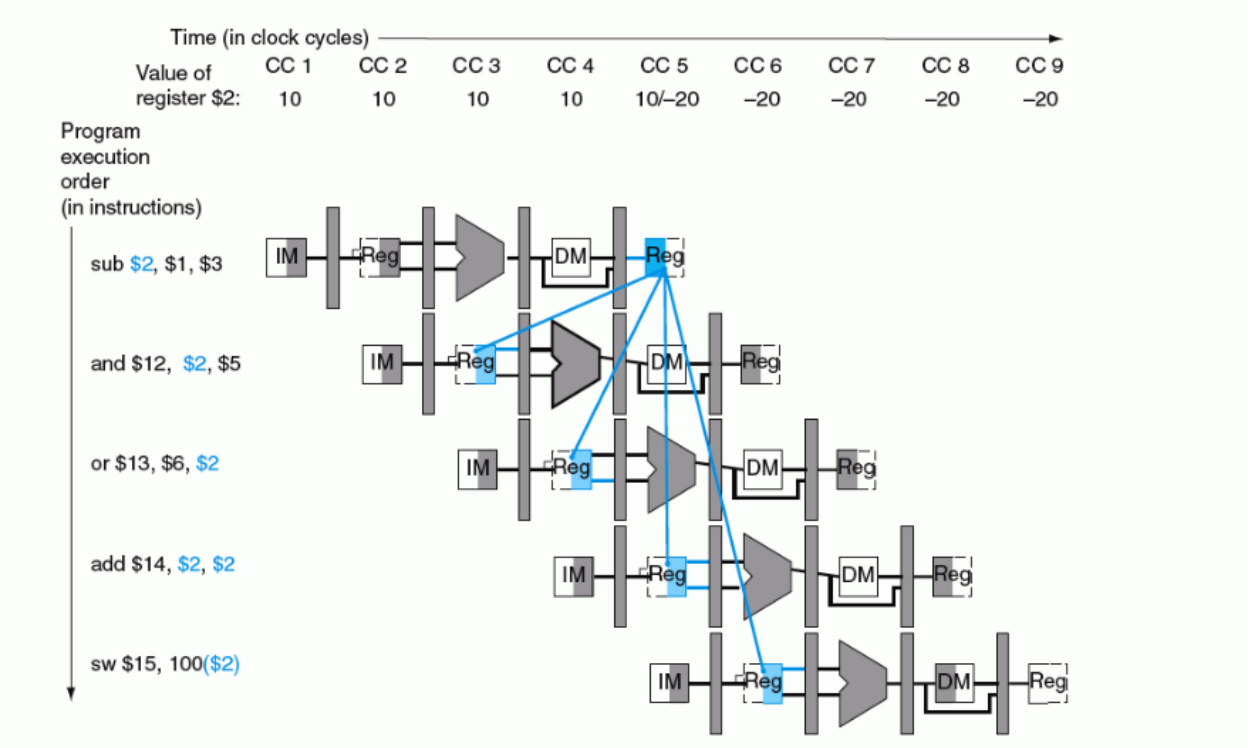


Figure 3.21: Pipelined dependencies in a five-instruction sequence using simplified data path[10]

The value of register field rs between instruction decode stage and operand read stage is denoted by OR/EX.registerRs refers to the number of one register whose value is found in the pipeline register after next to register file. Using this notation, the two pairs of hazard conditions are.

a. OR/EX.regsiterRd = ID/OR.registerRs

b. OR/EX.registerRd = ID/OR.registerRt
c. EX/WB.registerRd = ID/OR.registerRs
d. EX/WB.registerRd = ID/OR.registerRt

The first hazard in the sequence on the instruction sequence explained above is on register R2, between the result of sub R2, R1, R3 and the first read operand of and R12, R2, R5. This hazard can be detected when the and instruction is in the execution stage and te prior instruction is in the WB stage, so this the hazard a indicated above:

OR/EX.regsiterRd = ID/OR.registerRs = R2

Now what we can detect hazards, half of the problem is resolved, but we must still forward the proper data.

Thus the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

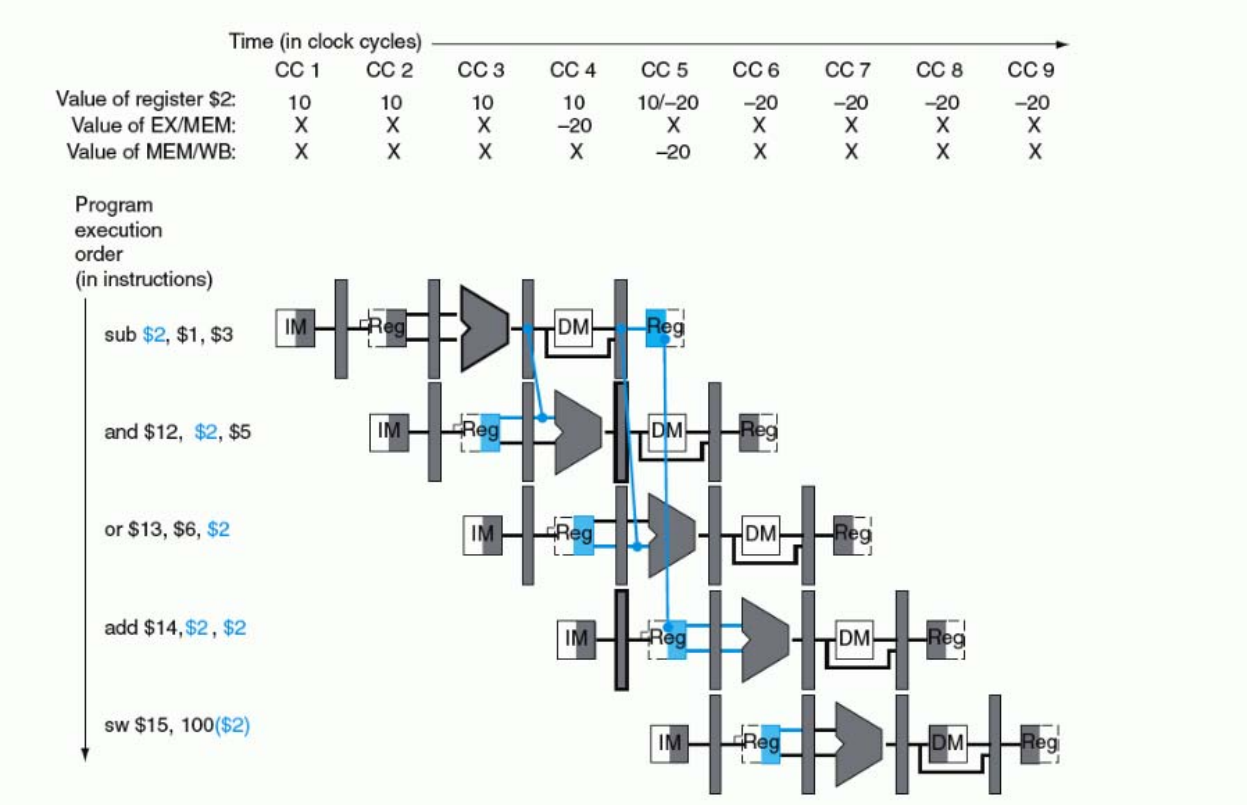


Figure 3.22: Dependences between pipeline registers move forward in time, so it is possible to supply the inputs needed by the ALU[10]

For now, we will assume the only instruction we need to forward the four R-format instructions: add, sub, and, and or. The forwarding control will be in the execution stage because the ALU forwarding multiplexers are fond in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/OR pipeline register to determine whether to forward values. We already have the rt field of instruction (bits 20:16). Before forwarding, the ID/OR register had no need to include space to hold the rs field. Hence, rs (bits 25-21) is added to ID/OR. Let's now write both the conditions for detecting hazards and the control signals to resolve them

i.e.

if ((OR/EX.regWrite) && (OR/EX.registerRd == ID/OP.registerRs))

forwardA = 01

and,

if ((OR/EX.regWrite) && (OR/EX.registerRd == ID/OP.registerRt))

forwardB = 01

This case forwards the results from the previous instruction to either input of the ALU, the steer the multiplexer to pick the value instead from the pipeline register OR/EX. forwardA and forwardB shows the select line of the multiplexers of the forwarding unit.

As mentioned, there is no hazard in the WB stage because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is the potential For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

add R1, R1, R2

add R1, R1, R3

add R1, R1, R4

If (EX/WB,regWrite && (OR/EX.registerRd ≠ ID/OR.registerRs) && (EX/WB.registerRd == ID/OR.registerRs))

ForwardA = 10;

And,

If (EX/WB,regWrite && (OR/EX.registerRd ≠ ID/OR.registerRt) && (EX/WB.registerRd == ID/OR.registerRt))

ForwardB = 10;

Conclusion: Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the WB stage, forwarding is easy. But consider loads immediately followed by stores. We need to add more forwarding hardware to make memory-to-memory copies run faster.

3.4.5.10 Internal data Memory:

The internal data memory acts as cache memory for the processor. The role of data memory comes into play for load and store instructions. For load instruction, some data memory is put into the register file and for store instruction some data from the register file is stored into the data memory. The address of the data memory is calculated by the ALU.

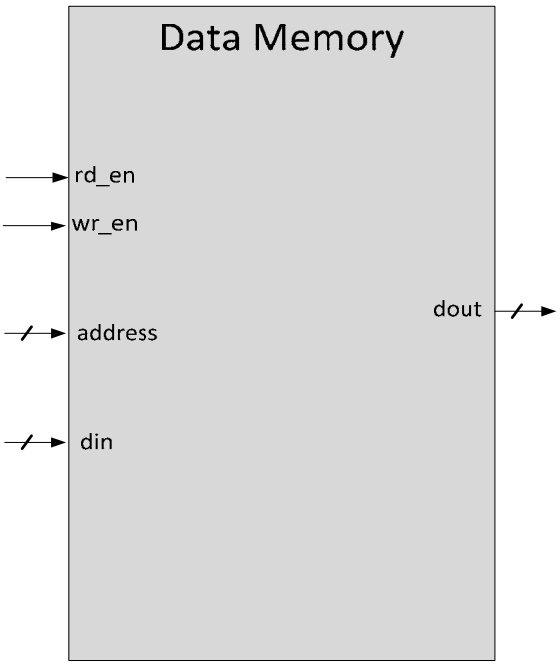


Figure 3.23: Data Memory

The size of data memory is customizable as it can be set through the parameter. Data can only be read from the data memory when the rd_en signal is asserted and same is the case with writing data in to the data memory as to write data the wr_en signal must be written asserted.

3.4.5.11 By-Pass Unit:

By pass unit supplies actual and appropriate inputs to the ALU. It consists of a series of multiplexors that forwards the correct operands to the ALU for the processing. The by-pass unit along with the forwarding unit discussed above plays a very significant

role in the efficiency of the overall system. Figure 3.24 shows the block diagram of the by-pass unit.

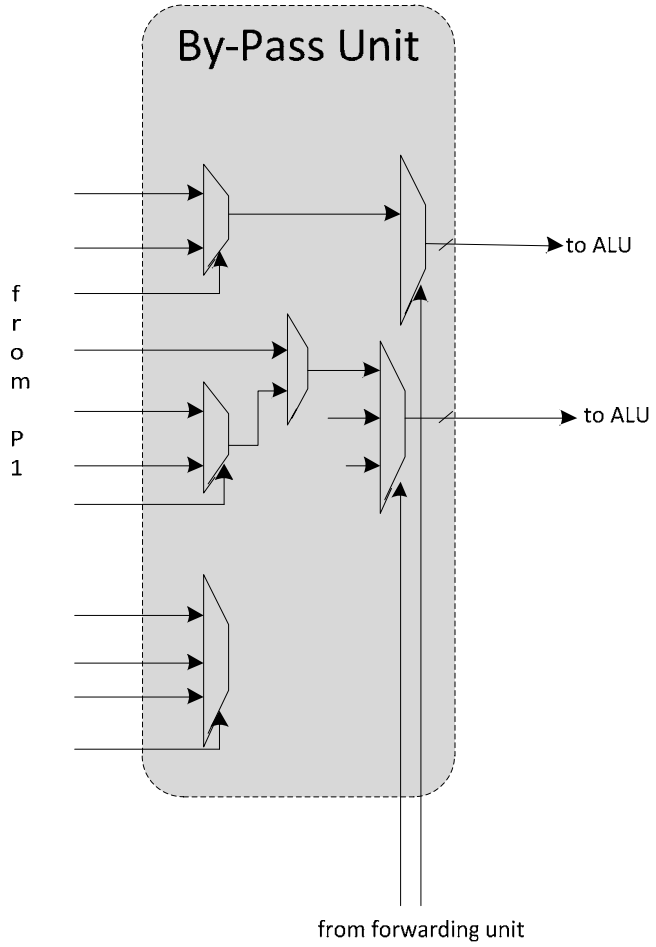


Figure 3.24: By-pass unit

By pass unit takes inputs from two blocks i.e. from forwarding unit and from P1 and generates operands for the ALU.

3.4.5.12 Input/output Memory:

The proposed processor takes data for processing from input memory and generates output to be stored in the output memory.

3.5 SYNTHESIS

Synthesis means the transformation of high level description into gate level net list. The proposed processor has been synthesized and successfully tested on the Xilinx virtex-5 FPGA. Figure 3.25 shows further details of the device.

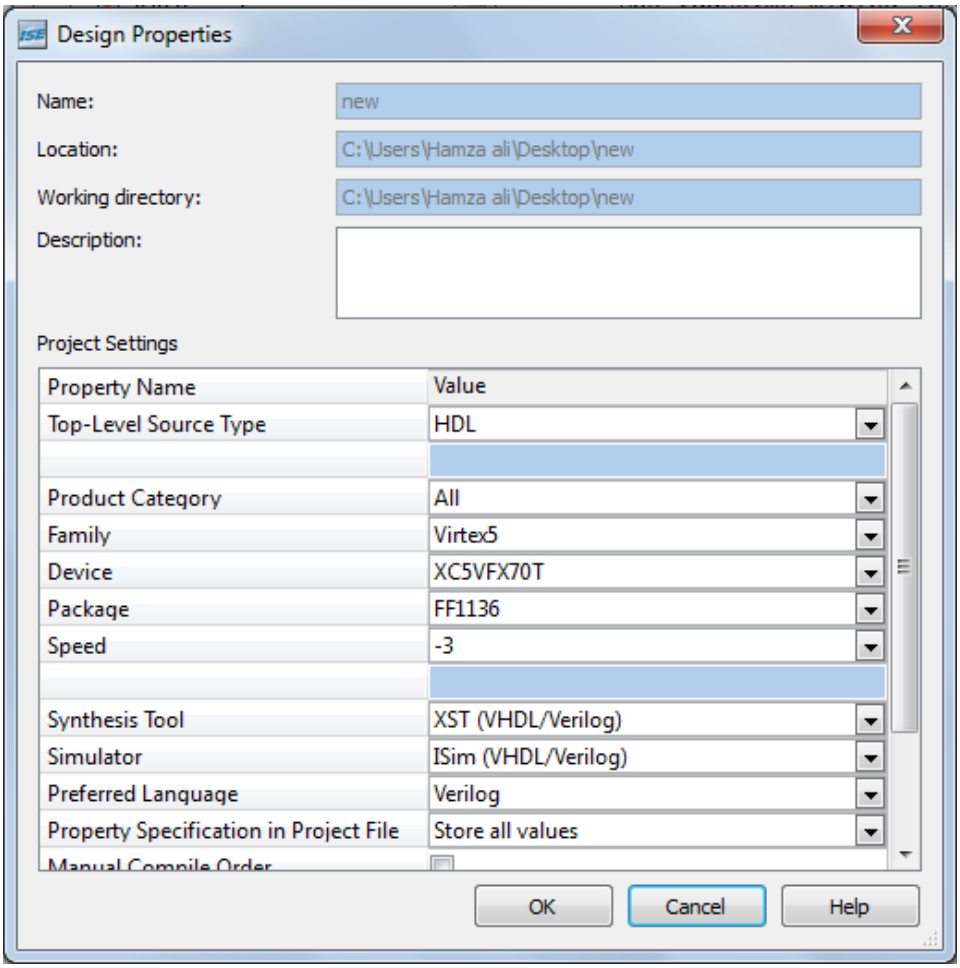


Figure 3.25: Synthesis Design Properties

Following diagram shows some points of the synthesis report

```
Selected Device : 5vfx70tff1136-3


Slice Logic Utilization:
 Number of Slice Registers:             133  out of  44800     0%
 Number of Slice LUTs:                  135  out of  44800     0%
    Number used as Logic:               127  out of  44800     0%
    Number used as Memory:                8  out of  13120     0%
       Number used as RAM:                8

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    179
   Number with an unused Flip Flop:      46  out of    179    25%
   Number with an unused LUT:            44  out of    179    24%
   Number of fully used LUT-FF pairs:    89  out of    179    49%
   Number of unique control sets:        14

IO Utilization:
 Number of IOs:                         122
 Number of bonded IOBs:                  70  out of    640    10%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:               1  out of     32     3%
```

Figure 3.26: Device Utilization Details

Following diagram shows the timing summary of the design after synthesis.

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 2.859ns (Maximum Frequency: 349.822MHz)
   Minimum input arrival time before clock: 2.287ns
   Maximum output required time after clock: 2.809ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)
```

Figure 3.27: Timing Summary of Design

3.5.1 Chip Scope Testing

Chip Scope pro is the built in Xilinx debugging utility that provides flexibility of debugging the design meanwhile the code is running on the FPGA device. Chip Scope Pro is very powerful debugging tool and used widely. Figure 3.28 shows the chip scope result.

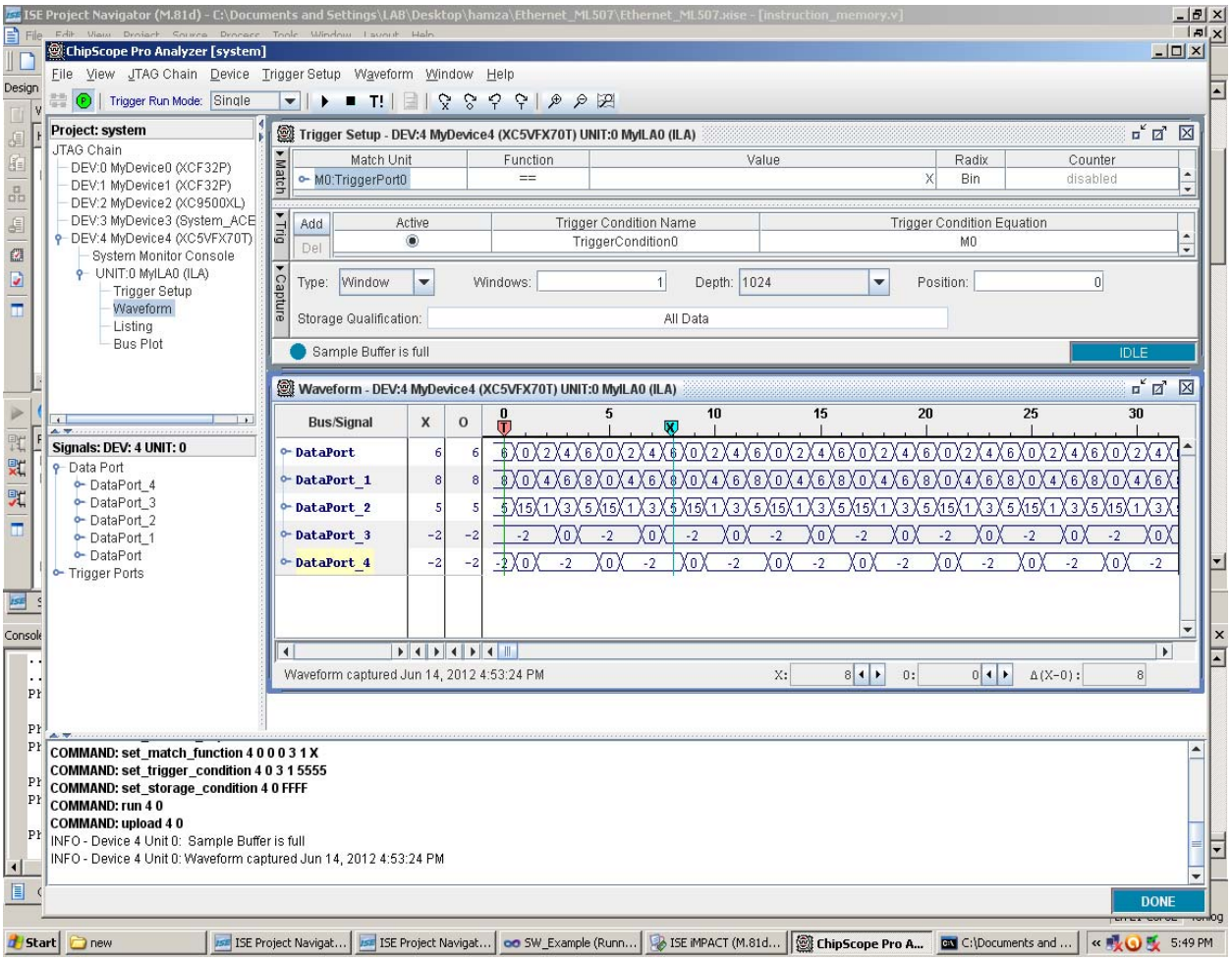


Figure 3.28: Chip Scope Pro Tool

3.6 INTERFACING

Interfacing must be done to get data from the PC into the FPGA and vice versa. So, for this purpose the Microsoft's Simple Interface for Reconfigurable Computing (SIRC) interface is used. This interface uses ether net for the communication between PC and FPGA. Figure 3.28 shows block diagram of this interface.

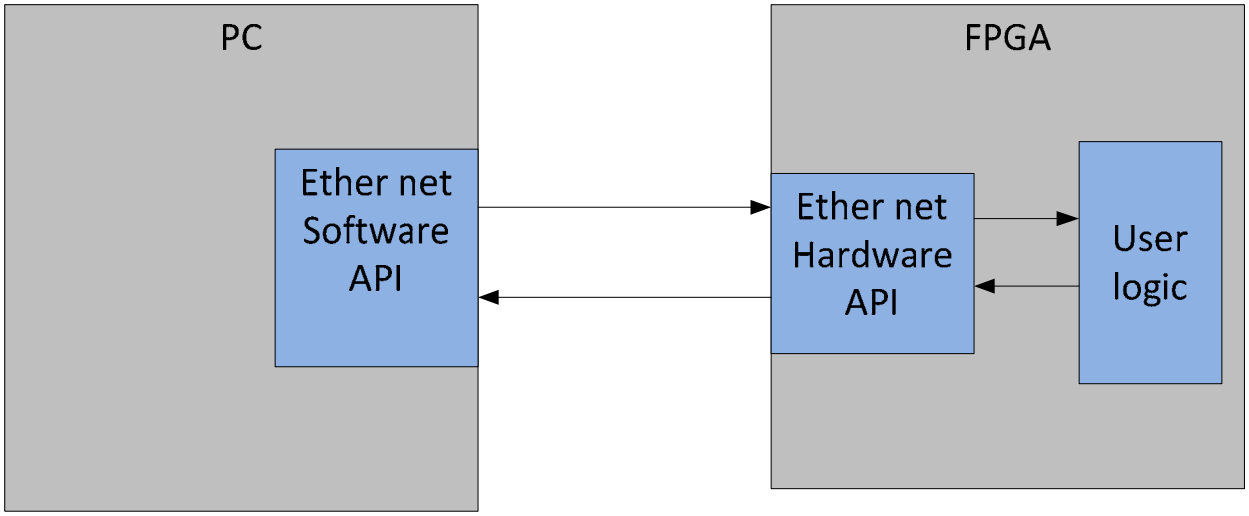


Figure 3.29: Interfacing between PC and FPGA

By using this interface the communication between PC and FPGA is done though two important components i.e. software API and hardware API. Software API runs on the PC side (in visual studio 2010) and the hardware API runs on FPGA side. PC sends data to FPGA through software API and FPGA receives data for processing through hardware API. Code for the software API is written in C++ and for the hardware API is written in Verilog HDL. Complete description of this interface is available at "http://www.drsparrows.com/wp-content/uploads/pdfs/SIRC_README_v1.0.1.pdf".

Figure 3.29 shows diagram of the state machine of the logic that how data is retrieved for processing and send back result back to PC.

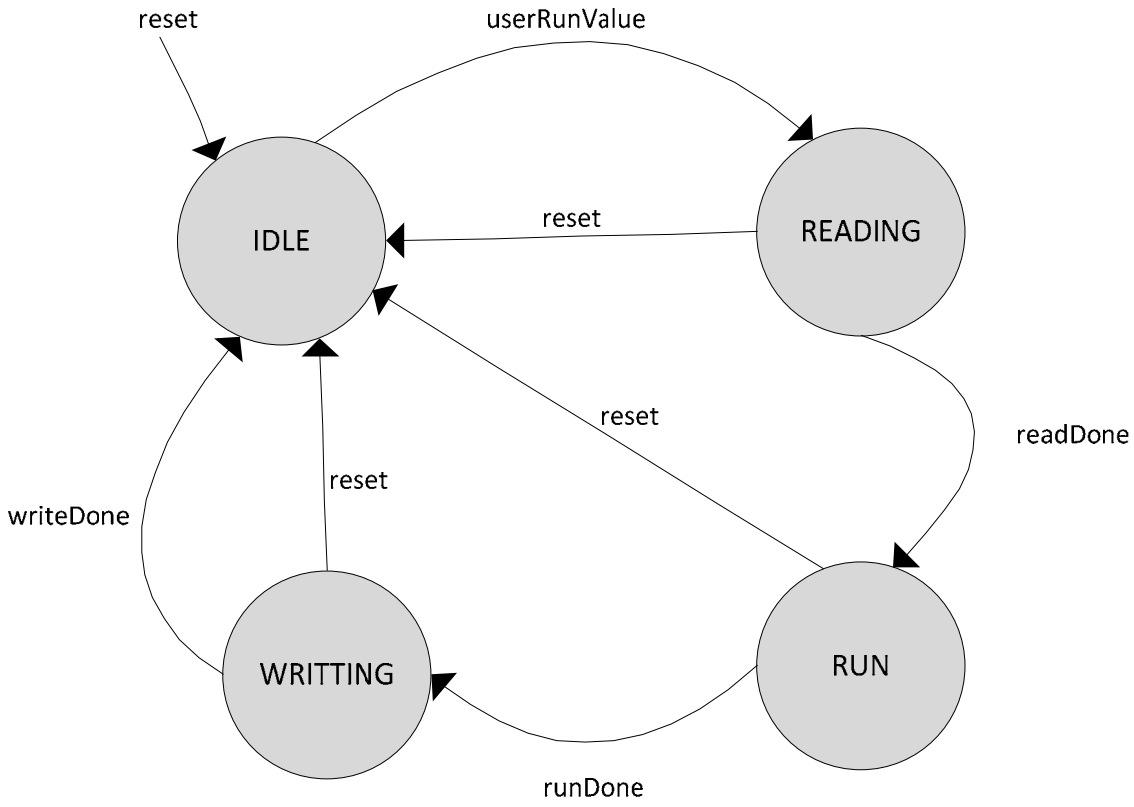


Figure 3.30: SIRC user logic state machine

The machine goes into the IDLE state straight away when reset is asserted. It keeps on waiting in IDLE state for the userRunValue signal to be asserted. As soon as this signal is asserted the state machine switches to READING state and start reading data from input buffer into the local memory. After all reading is done the readDone signal is asserted and the state machine switches to RUN state. Then all the processing of the proposed processor is done and results are written to local output memory. As soon as all the required processing is done the runDone signal isasserted and state machine switches to WRITTIN state. In this state the data (results) are written from local output memory to the output buffer from where the interface will send data to PC.

3.7 ASSEMBLY CODE

Following is the assembly code for the computation of 5/3 fast lifting wavelet transform algorithm

```
Label
        load    R1, Ri(base)
        load    R2, Ri(base)
                .
                .
                .
        load    R16, Ri(base)
        cmp  R1, R2, R17            add, shift, subtract
        cmp  R3, R4, R18
        cmp  R5, R6, R19
        cmp  R7, R8, R20
        cmp  R9, R10, R21
        cmp  R11, R12, R22
        cmp  R13, R14, R23
        cmp  R15, R16, R24
        store   R17, Ri(base)
        store   R18, Ri(base)

                    .
                    .
        store   R24, Ri(base)
Jump label
```

This assembly code runs basically in a loop. When it reaches the jump label instruction it takes jump to the start i.e. on Label and runs in the same fashion until loop breaks i.e. when address to read the memory equals stop address. In the same way the assembly code for all the fast lifting wavelet transform filters indicated in chapter 3 or any other signal processing algorithms can be written

Chapter 4

RESULTS AND DISCUSSIONS

1.1 OVERVIEW

Findings and results of the project "Design and Implementation of an Efficient Generic RISC Processor on FPGA for Digital Signal Processing Applications" are highlighted in this chapter. As proposed processor is meant for the digital signal processing applications, so different signals processing algorithms are run on the processor and following are results of these algorithms. The results of the proposed system has been retrieved and matched with the results of the matlab simulations to make comparison between the two. The comparison shows that there no dissimilarity between the results of the proposed processor when running the same algorithm as running in the matlab. Different signal processing algorithms are implemented for 1-D and 2-D (image) signals and following results are obtained.

As explained in chapter 3, the proposed processor has two modes of instructions i.e. normal mode and compound mode. In compound mode all ALU components (adder, shifters, multiplier and subtracter) can be used in the same clock cycle which improves the performance of the system significantly. Figure 4.1 and Figure 4.2 show comparison between normal mode instruction and compound mode instruction,if the 5/3 algorithm is run using normal mode then it takes 22 clock cycles to compute 4 output samples shown in figure 4.2 and if same task is done through compound instruction then it will take only 4 clock cycles to computer these 4 output samples of low frequency components shown in figure 4.1. It means by only considering 4 output samples we can drop the consumption of clock cycles by 18 in this case. As there are millions of input samples that need to be processed so by using compound instruction we can perform the computation intensive tasks quite efficiently.

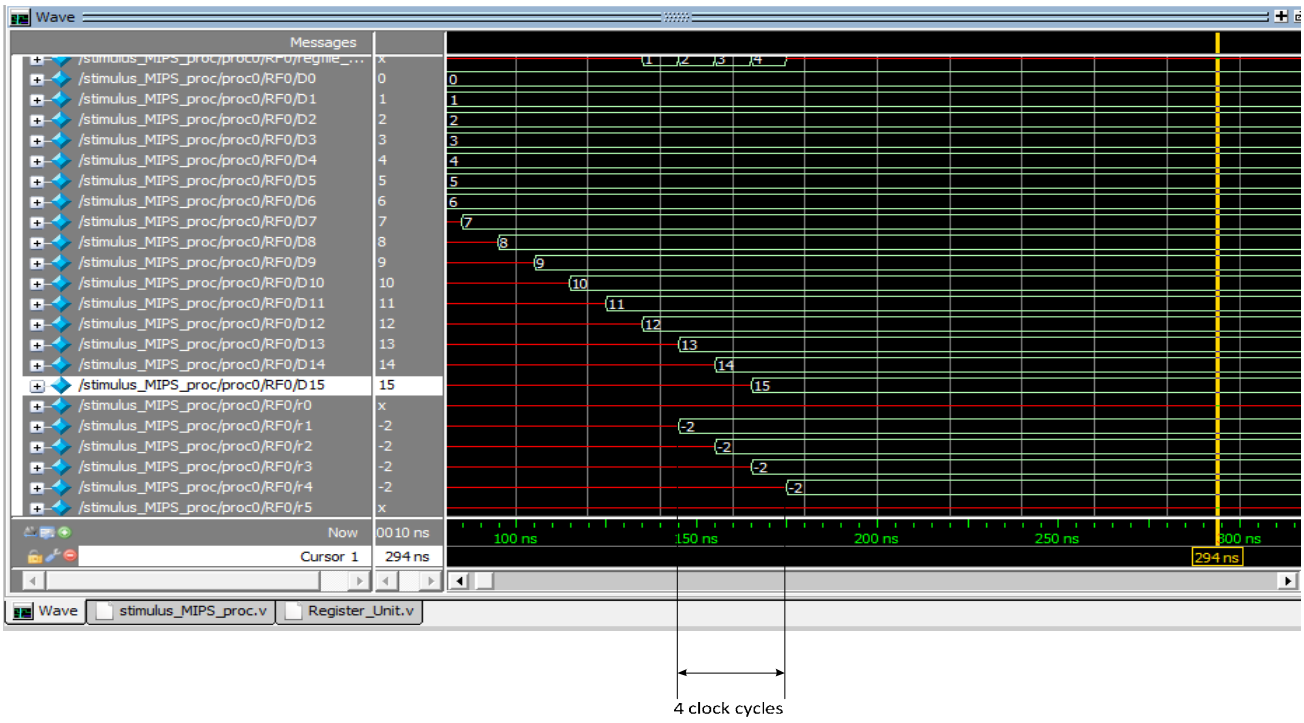


Figure 4.1: Compound Instruction waveform for 5/3 filter

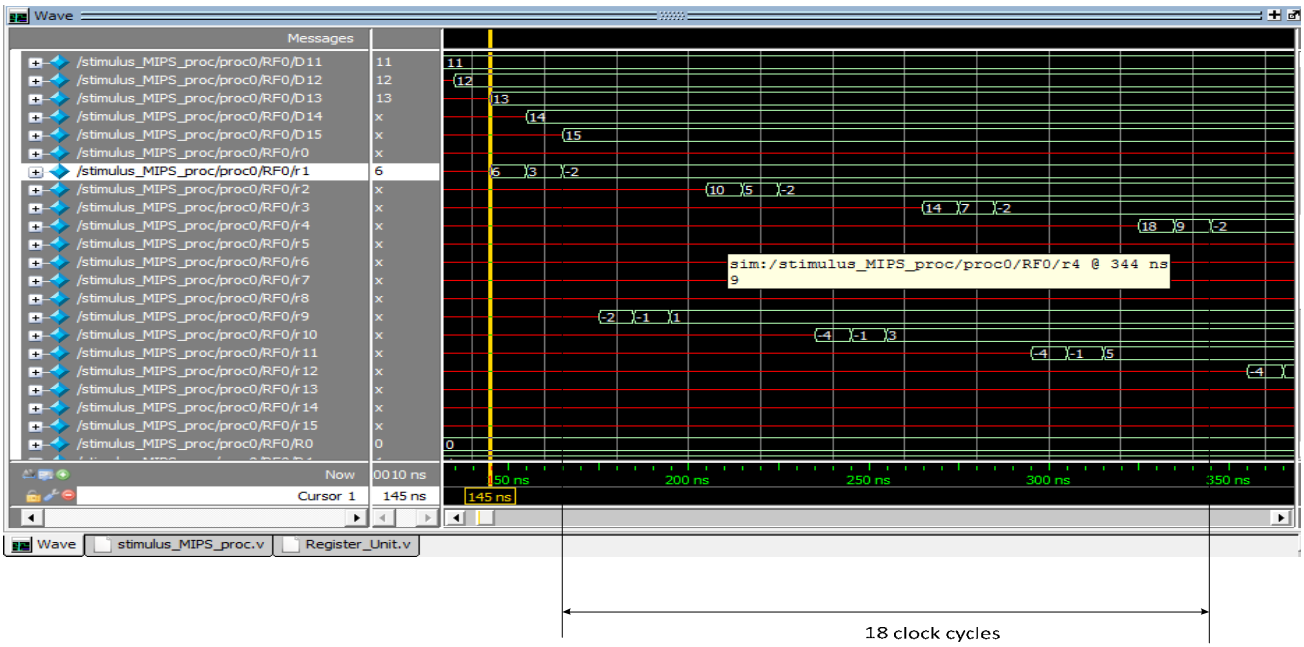


Figure 4.2: Normal Mode Instruction waveform for 5/3 filter

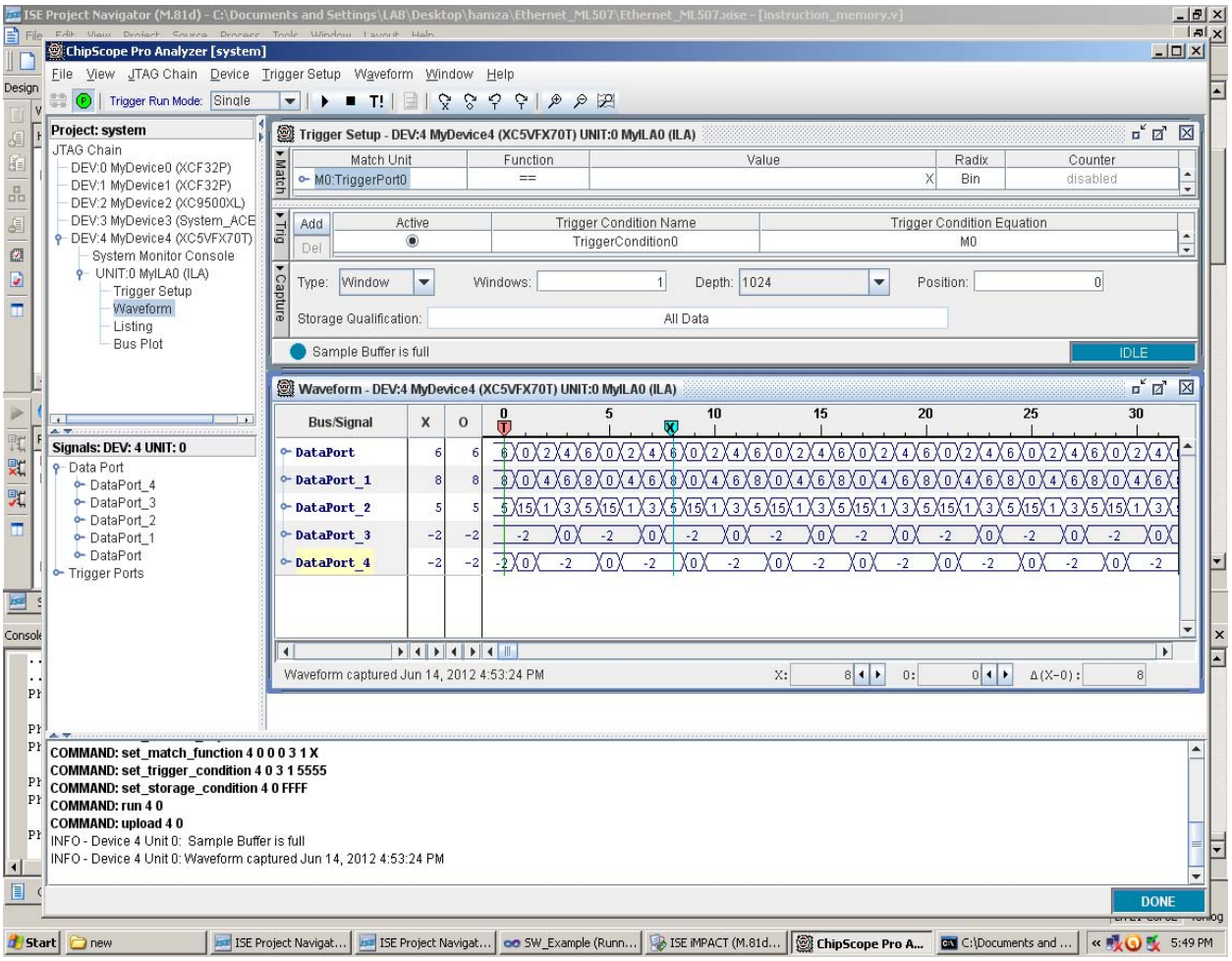


Figure 4.3: Compound Instruction results using chip scope pro for 5/3 filter

Both figure 4.1 and figure 4.3 show the results of compound instruction. Figure 4.1 shows simulation result of Model Sim while figure 4.3 show hardware result while code is running on the FPGA.

1.2 CIRCUIT PERFORMANCE RESULTS

Table 4.1 and table 4.2 show the results regarding to the efficiency of the circuit of the proposed processor.

Table 4.1 show the comparison of the proposed architecture with the existing architectures i.e.

Features	Liu's architecture	Chen's architecture	Wu's architecture	Lee & Lim's architecture	The proposed architecture
Frame size	32x32	256x256	1024x1024	1024x1024	1024x1024
Wavelet filter type	9/7	2 to 20 taps	Programmable	Programmable	Programmable
Clock frequency	25 MHz	50 MHz	100 MHz	200 MHz	350 MHz

Table 4.1: Performance comparison with existing architectures

The description of existing architectures is given in [2].

Image Size	Filter Type	Execution Time with compound instruction mode in micro seconds	Execution Time with normal instruction mode in micro seconds	No. of PE
256x256	5/3	93.622	280.866	1
256x256	5/3	46.811	140.433	2
512x512	5/3	374.491	1123.473	1
512x512	5/3	187.245	561.735	2
480x360	5/3	246.844	740.532	1
480x360	5/3	123.42	370.253	2
256x256	2/6	93.622	280.866	1
256x256	2/6	46.8	140.433	2
512x512	2/6	374.49	1123.473	1
512x512	2/6	187.245	561.735	2
480x360	2/6	246.844	740.532	1
480x360	2/6	123.42	370.253	2
256x256	9/7-f	187.244	1217.086	1
256x256	9/7-f	93.622	608.543	2
512x512	9/7-f	748.982	4868.38	1
512x512	9/7-f	374.491	2434.19	2
480x360	9/7-f	493.688	3708.972	1
480x360	9/7-f	246.84	1854.486	2

Table 4.2: Execution time of various wavelet transform algorithms on 2-D signals

Figure 4.4 shows the performance comparison between existing architectures and the proposed architecture in graphical form

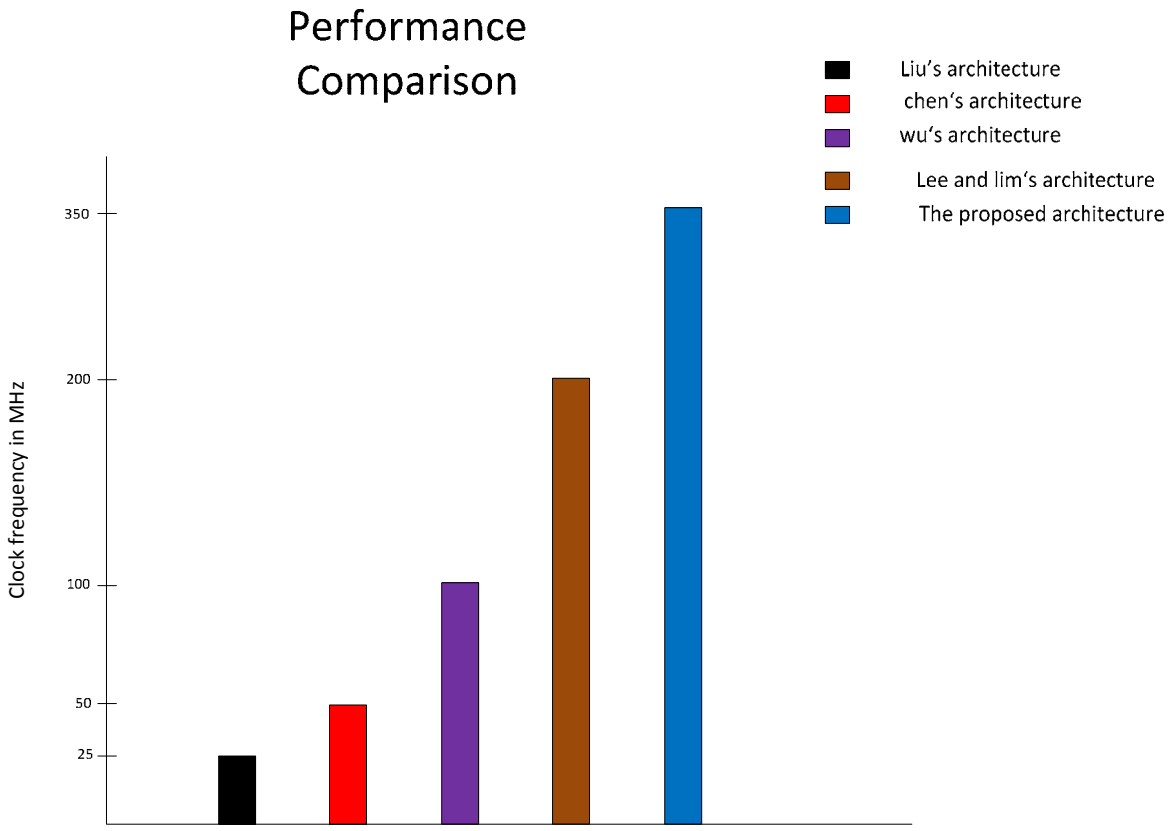


Figure 4.4: Performance comparison of existing architectures with proposed architecture

4.3 OUTPUT RESULTS

Following are the results of the design when different images of different sizes given to the design:

Following image of a texture when input to the algorithm gives the low frequency and high frequency components of the input image

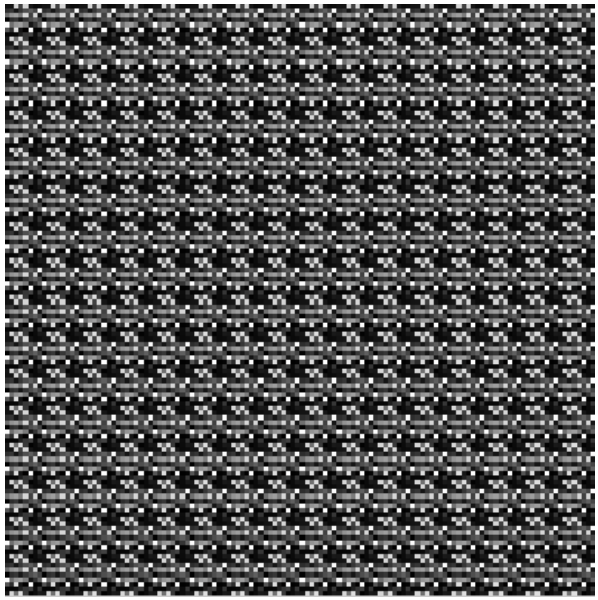


Figure 4.5: input image to 5/3 wavelet filter

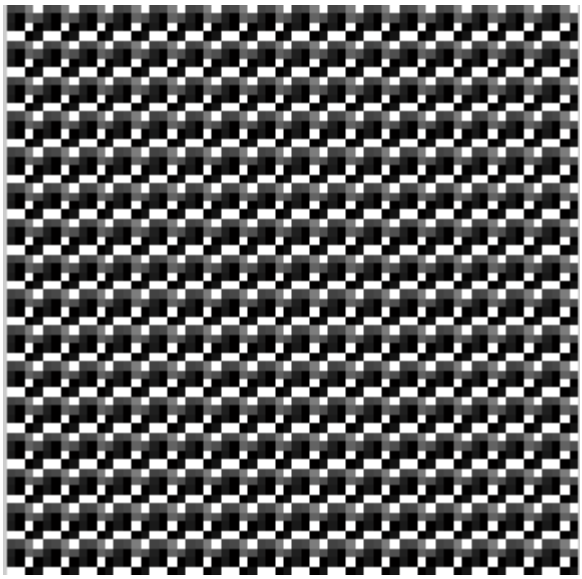


Figure 4.6: Low frequency component of input image

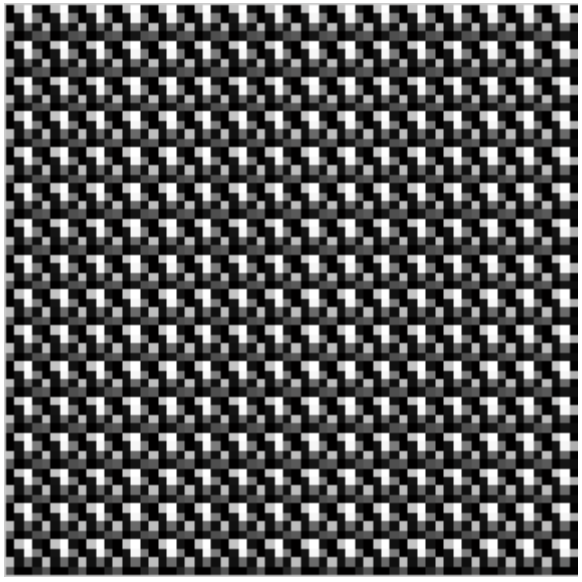


Figure 4.7: High frequency component of input image

Following results are of different wavelet transform algorithms.

Figure 4.8: input image to 5/3 wavelet filter

Figure 4.9: Level 1 decomposition of 5/3 wavelet filter

Figure 4.10: Level 2 decomposition of 5/3 wavelet filter

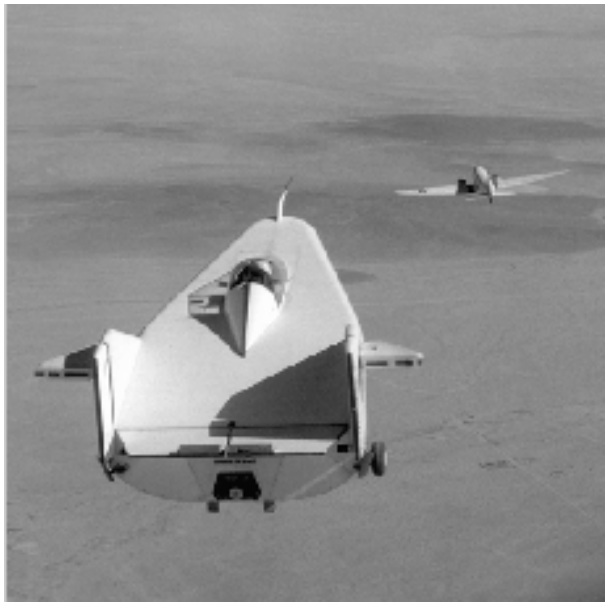


Figure 4.11: Input image to 2/6 wavelet filter

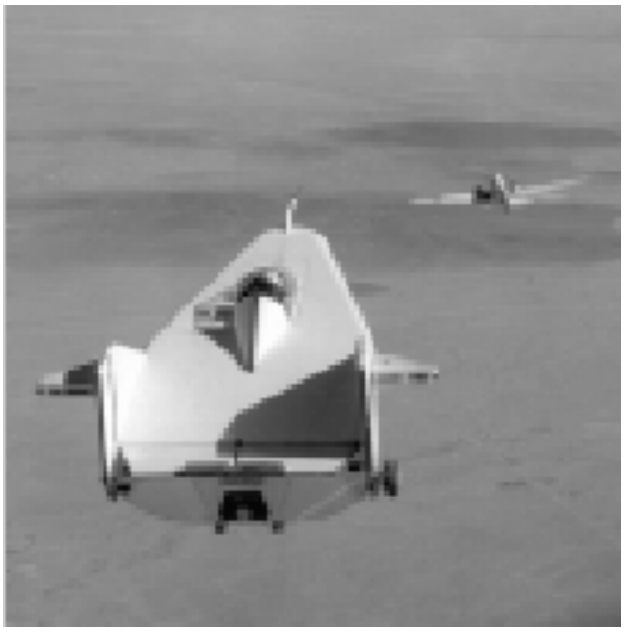


Figure 4.12: Level 1 decomposition of 2/6 wavelet filter

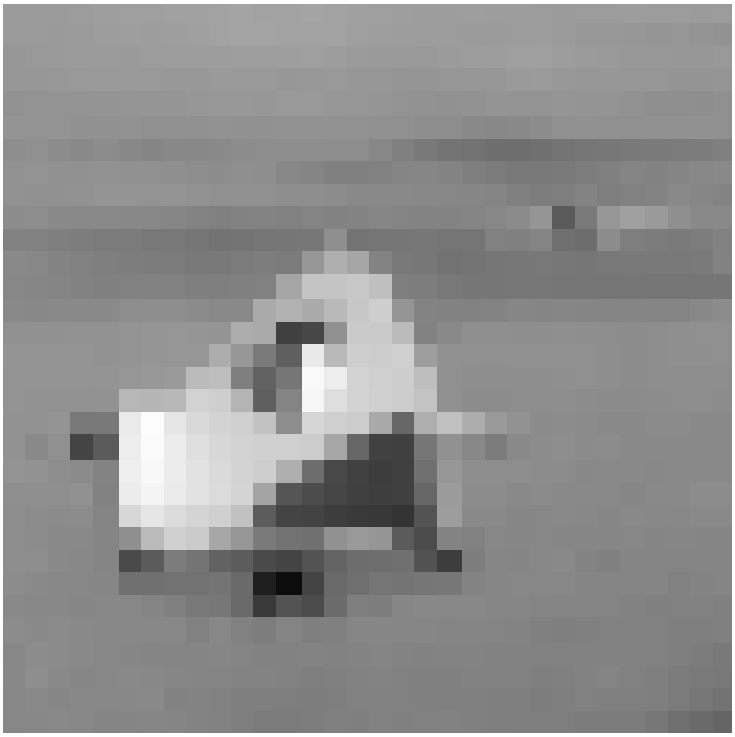


Figure 4.13: Level 3 decomposition of 2/6 wavelet filter

Chapter 5

CONCLUSIONS AND RECOMMENDATIONS

5.1 DESIGN SUMMARY

The proposed processor is primarily based upon on RISC architecture. Reduced Instruction Set Computers (RISC) are designed to have a small set of instructions that execute in short clock cycles, with a small number of cycles per instruction. RISC machines are optimized to achieve efficient pipelining of their instruction streams. The proposed processor is basically a model of RISC based architecture. This architecture also serves as a starting point for developing architectural variants and a more robust instruction set.

To achieve the goal of this research work, a very flexible methodology has been adopted to accomplish the research work. A Generic processor has been implemented based on the Reduced Instruction Set Computer (RISC) architecture. The architecture is fully pipelined and hence gives higher throughput. An instruction set is designed to facilitate the programmer to use the system easily. Based on the instruction set, all operations of the proposed processor are carried out. Due to pipelined design, the achieved cycle per instruction (CPI) is 1 that has a very significant impact on the throughput of overall system.

Having discussed all the design parameters, architecture and the results it is concluded that a highly flexible, scalable and efficient processing engines as per the requirements of the application at hands gives very high design performance. High through put of the system is achieved through pipelining. Each instruction of the design can be executed in 1 clock cycle.

As almost all signal processing tasks are very computational intensive i.e. much of processing is done on very large number of data samples so to keeping in mind this

parameter a compound instruction has been designed that utilized multiple computational blocks (adder, subtractor, shifter, and multiplier) to be used in the single clock cycle to increase efficiency of the overall system.

5.2 CONCLUSIONS

The conclusions drawn from the developments and findings of this research work are enumerated below:

1) Algorithm preserves the logic.
2) Hardware results are well comparable with those reported by simulations.
3) High throughput is achieved by keeping the cycle per instruction (CPI) equal to 1.
4) Fully pipelined design makes the execution faster and in an efficient way.
5) There is a significant level of improvement in circuit timing, while area remains almost the same, and in some cases it was observed to even increase.
6) Each signal processing algorithm can be implemented using the same instruction set of the proposed processor.

5.3 RECOMMENDATIONS

Fast Lifting Scheme is a novel innovation and is need of the hour, both in developed and in developing countries. The future is guiding the mankind toward faster possible means of performing all the Digital Signal Processing operations in real times.

In this scenario it is inevitable that consistent and sustainable efforts remain on the right track to implement a continuous process of improvement till an optimum level of excellence is achieved in this field.

Since the completion of the project needed extra efforts to unearth some basic facts, many new problems cropped up which are considered to be dealt separately for their proper treatment so that they are resolved for the benefit of the whole mankind.

With this background, following recommendations have been formulated for future search, research, studies and projects:

1) The design of the ALU can be improved.
2) The external memories and the internal data memory can be merged together to have common storage for all instructions.
3) The ALU block can be reduced in way that subtracted can be omitted from the ALU by having subtraction from the adder through 2's compliment technique