Report Titled

Design and Implementation of Thumb Processor on FPGA

Submitted in partial fulfillment of the requirements of the degree of

Bachelor Of Technology (Electronics And Telecommunication Engineering)

By

Apurva Tayade 111091970

Chinmay Gadgil 101090035

Janaki Thakkar 101091012

Mohnish Bhatia 111090971

Sagar Bachwani 111090973

Saurabh Nair 101090010

2013-14

Under the guidance of Dr. R. D. Daruwala

Electrical Engineering Department

Veermata Jijabai Tehcnological Institute

Mumbai 400 019

2013-14

DEPARTMENT OF ELECTRICAL ENGINEERING

VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE

(AUTONOMOUS INSTITUTE AFFILIATED TO UNIVERSITY OF MUMBAI)

MATUNGA, MUMBAI - 400019

CERTIFICATE

This is to certify that **Apurva Tayade**, **Chinmay Gadgil, Janaki Thakkar, Mohnish Bhatia**, **Sagar Bachwani** and **Saurabh Nair** students of Bachelor of Technology (Electronics and Telecommunication Engineering), have completed the thesis / dissertation / report entitled, **"Design and Implementation of Thumb Processor on FPGA"** to our satisfaction.

Dr. R.D.Daruwala — **Guide / Supervisor**

Dr. M. S. Panse — **Head (Electrical Department)**

Declaration Of The Students

We declare that this written submission represents our ideas in our own words and where others' ideas/words have been included, we have adequately cited and referenced the original sources.

We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented/falsified any data/fact/idea/source in our submission.

We understand that any violation of the above will be cause for disciplinary action by the institute and can also invoke penal action from the sources which have not been properly cited or from whom proper permission has not been taken where needed.

Signatures Of The Students

Ms. Apurva Tayade

Mr. Chinmay Gadgil

Mr. Janaki Thakkar

Mr. Mohnish Bhatia

Mr. Sagar Bachwani

Mr. Saurabh Nair

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

As a part of our Final Year B. Tech Project (2013-14), we decided to work on Design and Implementation of Thumb Processor on FPGA under the guidance of the experienced faculty Dr. Robin Daruwala, VJTI.

We humbly express our gratitude towards our guide Dr. Daruwala for his guidance and patience towards building a project that required expert knowledge, skill and hard work.

We also express our thankfulness towards the Head of the Department of Electrical Engineering, Mrs. Panse for her motivation and encouragement and allowing us to undertake such a unique project.

And last but not the least, we would like to thank our seniors and other staff members of the Department of Electrical engineering, VJTI for lending a helping hand in times of challenges during making of this project and also the library staff for lending us the reference materials and journals necessary for the successful completion of this project.

CHAPTER 1

INTRODUCTION

The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.
—Edsger W. Dijkstra

1.1 What is a Processor?

A general-purpose processor is a finite-state automaton that executes instructions held in a memory. The state of the system is defined by the values held in the memory locations together with the values held in certain registers within the processor itself . Each instruction defines a particular way the total state should change and it also defines which instruction should be executed next. The stored program digital computer keeps its instruction and data in the same memory system, allowing the instructions to be treated as data whenever necessary. This enables the processor itself to generate instructions which it can subsequently execute. Although programs that do this at a fine granularity (self-modifying code) are generally considered bad form these days since they are very difficult to debug, use at a coarser granularity is fundamental to the way most computers operate. Whenever a computer loads in a new program from disk (overwriting an old program) and then executes it, the computer is employing this ability to change its own program. Because of its programmability a stored-program digital computer is universal, which means that it can undertake any task that can be described by a suitable algorithm. Sometimes this is reflected by its configuration as a desktop machine where the user runs different programs at different times, but sometimes it is reflected by the same processor being used in a range of different applications, each with a fixed program. Such applications are characteristically embedded into products such as mobile telephones, automotive engine-management systems, and so on.

The above block diagram is that of a general microprocessor. A micro processor has many support devices like Read only memory, Read-Write memory, Serial interface, Timer, Input/Output ports etc. All these support devices are interfaced to microprocessor via a system bus. So one point is clear now, all support devices in a

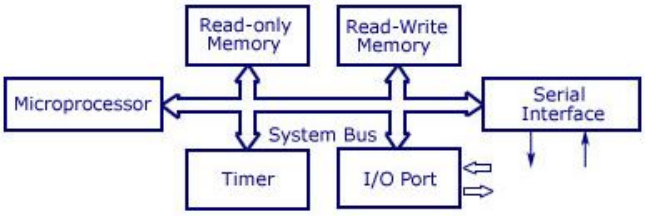

Figure 1.1 General Micro Processor Block Diagram

microprocessor based system are external. The system bus is composed of an address bus, data bus and control bus.

1.2 Evolution Of Processors

All modern general-purpose computers employ the principles of the stored-program digital computer. The stored-program concept originated from the Princeton Institute of Advanced Studies in the 1940s and was first implemented in the 'Baby' machine which first ran in June 1948 at the University of Manchester in England. Fifty years of development have resulted in a spectacular increase in the performance of processors and an equally spectacular reduction in their cost. Over this period of relentless progress in the cost-effectiveness of computers, the principles of operation have changed remarkably little. Most of the improvements have resulted from advances in the technology of electronics, moving from valves (vacuum tubes) to individual transistors, to integrated circuits (ICs) incorporating several bipolar transistors and then through generations of IC technology leading to today's very large scale integrated (VLSI) circuits delivering millions of field-effect transistors on a single chip. As transistors get smaller they get cheaper, faster, and consume less power. This win-win scenario has carried the computer industry forward for the past three decades, and will continue to do so at least for the next few years. However, not all of the progress over the past 50 years has come from advances in electronics technology. There have also been occasions when a new insight into the way that technology is employed has made a significant contribution. Following are the interpretations of these terms:

1. Computer architecture describes the user's view of the computer. The instruction set, visible registers, memory management table structures and exception handling model are all part of the architecture
2. Computer organization describes the user-invisible implementation of the architecture. The pipeline structure, transparent cache, table-walking hardware and translation look-aside buffer are all aspects of the organization.

Amongst the advances in these aspects of the design of computers, the introduction of virtual memory in the early 1960s, of transparent cache memories, of pipelining and so on, have all been milestones in the evolution of computers. The RISC idea ranks amongst these advances, offering a significant shift in the balance of forces which determines the cost-effectiveness of computer technology.

The following table gives a brief description of all the significant processors which finally led to ARM architecture processor:

Table 1.1 History Of Processors

Number	Name	Year	Features
1	Intel 4004	1971	Number of instructions-46, designed clock speed-1 MHz,740 kHz was achieved
2	Intel 8008	1972	Clock speed-0.5 MHz, Number of instructions-48
3	Intel 8080	1974	Clock speed increased to 2 MHz,Separation of address and data bus,Supported 256 I/Os
4	Motorola 6800	1974	Memory-mapped I/O
5	Intel 8085	1977	Von-Neuman architecture, 256 instructions
6	Intel 8086	1979	Clockspeed 10 MHz, Onboard clock
7	Sparc	1987	Clock Speed 40 Mhz, One of the fastest computers, 256 I/O pins
8	AM386	1991	Clock Speed 40Mhz, 32 bit data bus
9	ARM7TDMI	1998	32-bit RISC Processor, 16-bit Thumb mode
10	ARM9e	2004	DSP-enhanced 32-bit RISC processor
11	ARM11	2007	Low power, performance range from 350 Mhz to 1 Ghz
12	Cortex A8	2009	Superscalar dual-issue micro-architecture
13	Cortex A53	2012	32-64 bit ,8 stage pipelne

CHAPTER 2

DESIGN OBJECTIVE

Design is Intelligence made Visible.

—Alina Wheeler

2.1 ARM

ARM chips are RISC processors that were created by Acorn and the design is now owned by ARM Limited. They license the design out to manufactures to add on to their own chips to create systems on a chip that contain many feature on one chip, and with the ARM design are able to include the CPU as well.

It has the following features:

1. 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified)
2. Conditional execution of every instruction
3. The inclusion of very powerful load and store multiple register instructions
4. The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle
5. A very dense 16-bit compressed representation of the instruction set in the Thumb architecture
6. Pipelining : Instruction pipelining is one of the key features which reduces execution time
7. A high clock rate with single-cycle execution

2.1.1 ARM as a reference design

For designing a processor, we have taken ARM instruction set as our reference design, more specifically ARM processor, used in 16-bit Thumb mode, which will be explained in the further sections. The key features of the ARM have already been covered in the previous sections. Taking that as a reference, the architecture, instruction set, and the organization of the processor designed as a part of this project has been done.

2.1.2 Why ARM

The sales pitch goes something like this, "The ARM architecture has the best MIPS to Watts ratio as well as best MIPS to dollars ratio in the industry; the smallest CPU die size; all the necessary computing capability coupled with low power consumption of which a highly flexible and customizable set of processors are available with options to choose from, all at a low cost."

Multiple thoughts have gone into ariving on a strong reference design for the project, and ARM won the battle because of the following reasons:

1. ARM is better than Intel chips at decoding instructions
2. ARM can make CPU cores about 20 percent smaller and more power efficient because of RISC
3. Since a simplified instruction set allows for a pipelined, superscalar design RISC processors often achieve 2 to 4 times the performance of CISC processors
4. Because the instruction set of a RISC processor is so simple, it uses up much less chip space; extra functions, such as memory management units or floating point arithmetic units, can also be placed on the same chip
5. A shorter development time : A simple processor should take less design effort and therefore have a lower design cost and be better matched to the process technology when it is launched (since process technology developments need be predicted over a shorter development period
6. The combination of the simple hardware with an instruction set that is grounded in RISC ideas but retains a few key CISC features, and thereby achieves a significantly better code density than a pure RISC, has given the ARM its power-efficiency and its small core size

Infact, the small size, low cost, and low power usage leads to one of the most common uses for an ARM processor today, embedded applications. Embedded environments like cell phones or PDAs (Personal Digital Assistants) require those benefits that this architecture provides. Sure, there has to be a trade-off between performance, cost, and size. But, the ARM fits into this category nicely. It has very small die size, its performance, although not on the cutting edge, is more than adequate for the tasks at hand, and most importantly, it is cheap and low in power consumption.

2.1.3 Why Thumb in ARM

To those readers familiar with modern RISC instruction sets, the ARM instruction set may appear to have rather more formats than other commercial RISC processors. While this is certainly the case and it does lead to more complex instruction decoding, it also leads to higher code density. For the small embedded systems that most ARM processors are used in, this code density advantage outweighs the small performance penalty incurred by the decode complexity. Thumb code extends this advantage to give ARM better code density than most CISC processors. The Thumb instruction set may be viewed as a compressed form of a subset of the ARM instruction set. Thumb instructions map onto ARM instructions, and the Thumb programmer's model maps onto the ARM programmer's model. Implementations of Thumb use dynamic decompression in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor.

Thumb instructions are 16 bits long and encode the functionality of an ARM instruction in half the number of bits, but since a Thumb instruction typically has less semantic content than an ARM instruction, a particular program will require more Thumb instructions than it would have needed ARM instructions. The ratio will vary from program to program, but in a typical example Thumb code may require 70% of the space of ARM code. Therefore if we compare the Thumb solution with the pure ARM code solution, the following characteristics emerge:

1. The Thumb code requires 70% of the space of the ARM code
2. The Thumb code uses 40% more instructions than the ARM code
3. With 32-bit memory, the ARM code is 40% faster than the Thumb code
4. With 16-bit memory, the Thumb code is 45% faster than the ARM code
5. Thumb code uses 30% less external memory power than ARM code content

Most Thumb instructions are executed unconditionally. (All ARM instructions are executed conditionally). Many Thumb data processing instructions use a 2-address format (the destination register is the same as one of the source registers). (ARM data processing instructions, with the exception of the 64-bit multiplies, use a 3-address format). Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.

Examples of Thumb system:

1. A high-end 32-bit ARM system may use Thumb code for certain non-critical routines to save power or memory requirements
2. A low-end 16-bit system may have a small amount of on-chip 32-bit RAM for critical routines running ARM code, but use off-chip Thumb code for all non-critical routines.

The second of these examples is perhaps closer to the sort of application for which Thumb was developed. Mobile telephone and pager applications incorporate real-time digital signal processing (DSP) functions that may require the full power of the ARM, but these are tightly coded routines that can fit in a small amount of on-chip memory.

The more complex and much larger code that controls the user interface, battery management system, and so on, is less time-critical, and the use of Thumb code will enable off-chip ROMs to give good performance on an 8- or 16-bit bus, saving cost and improving battery life.

2.2 Design Tasks

As a part of this project, following tasks of designing and implementation were undertaken:

1. Designing of block diagram of the processor
2. Building a datapath
3. Designing an Instruction Set for the processor
4. Synthesizing of the various components of the datapath in VHDL with the help of Xilinx ISE
5. RTL simulation of the components of the datapath
6. Generating VHDL testbench waveforms of the RTL models
7. Simulation and Testing of individual models using ISim Simulator

8. Incorporation of each model in the common top model which is our actual prototype
9. Prototype Simulation and Testing
10. Verification of the entire design
11. Programming the FPGA device

CHAPTER 3

SYSTEM REQUIREMENTS

There are finer fish in the sea, than have ever been caught.

—Irish Proverb

3.1 FPGA

3.1.1 What is FPGA

A field programmable gate array (FPGA) is a logic device that contains a two-dimensional array of generic logic cells and programmable switches. The conceptual structure of an FPGA device is shown in the figure. A logic cell can be configured (i.e., programmed) to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic cells. A custom design can be implemented by specifying the function of each logic cell and selectively setting the connection of each programmable switch. Once the design and synthesis is completed, we can use a simple adaptor cable to download the desired logic cell and switch configuration to the FPGA device and obtain the custom circuit. Since this process can be done ”in the field” rather than ”in a fabrication facility (fab),” the device is known as field programmable.

3.1.2 Overview of Diligent Spartan 3E Board

The Nexys-2 is a powerful digital system design platform built around a Xilinx Spartan 3E FPGA. With 16 Mb of fast SDRAM and 16Mb of Flash ROM, the Nexys-2 is ideally suited to embedded processors like Xilinx's 32-bit RISC Microblaze. The on-board high-speed USB2 port, together with a collection of I/O devices, data ports, and expansion connectors, allow a wide range of designs to be completed without the need for any additional components.

Following are the on-board components and features of this board:

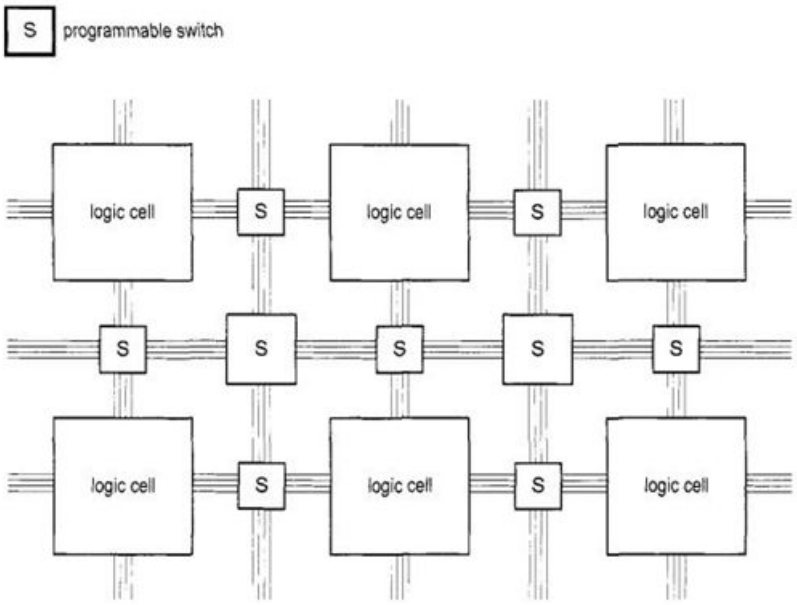

Figure 3.1 FPGA(Field Programmable Gate Array)

1. Xilinx Spartan-3E FPGA, 500K or 1200K gate
2. USB2 port providing board power, device configuration, and high-speed data transfers
3. Works with ISE/Webpack and EDK
4. 16MB fast Micron PSDRAM
5. 16MB Intel StrataFlash Flash R
6. Xilinx Platform Flash ROM
7. High-efficiency switching power supplies (good for battery-powered applications
8. 50MHz oscillator, plus a socket for a second oscillator
9. 75 FPGA I/Os routed to expansion connectors (one high-speed Hirose FX2 connector with 43 signals and four 2x6 Pmod connectors)
10. All I/O signals are ESD and short-circuit protected, ensuring a long operating life in any environment.
11. On-board I/O includes eight LEDs, four-digit seven-segment display, four pushbuttons, eight slide switches

3.2 Overview of Xilinx ISE

Xilinx ISE (Integrated Software Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize (”compile”) their designs, perform timing analysis, examine RTL diagrams, simulate a design’s reaction to different stimuli, and configure the target device with the programmer.

Xilinx ISE 14.2 has been used for the project. It supports FPGA Spartan Nexys 2 board.

Xilinx Partial Reconfiguration technology allowed us to change functionality on the fly, eliminating the need to fully reconfigure and re-establish links, dramatically enhancing the flexibility that FPGAs offer.

3.2.1 Creating Design project and HDL codes

Xilinx ISE (integrated software environment) controls all aspects of the development flow. Project Navigator is a graphical interface for users to access software tools and relevant files associated with the project.

An ISE project contains basic information of a design, which includes the source files and a target device. There are three tasks in this step-Create a project, Add or create HDL files, Check the HDL syntax. The ISE project fileis a file that contains all source-relevant data for the project as follows:

1. ISE software version information
2. List of source files contained in the project
3. Source settings, including design and process properties

The ISE project file includes the following characteristics, which are compatible with source control environments:

1. Contains all of the necessary source settings and input data for the project.
2. Can be opened in Project Navigator in a read-only state.
3. Only updated or modified if a source-level change is made to the project.

3.2.2 Creating TestBench Waveform and performing RTL simulation

To simulate your design, both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench are needed. A test bench is HDL code that provides a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing. The testbench functions as a virtual lab bench. It consists of the HDL module to be tested and a code segment to generate the stimulus. The test bench instantiates the unit under test (UUT), and stimulus is applied from the top-level test bench to the lower-level design or portion of the design being tested. The same test bench can be used for both functional and timing simulation.

The RTL simulation verifies operation of the HDL module in the host computer. After the HDL synthesis phase of the synthesis process, you can display a schematic representation of your synthesized source file. This schematic shows a representation of the pre-optimized design in terms of generic symbols, such as adders, multipliers, counters, AND gates, and OR gates, that are independent of the targeted Xilinx device. Viewing this schematic may help in discovering design issues early in the design process.

3.2.3 Adding constraint files and synthesizing and implementing codes

Constraints are certain conditions imposed on the synthesis and implementation processes. For our purposes, the main type of constraint is the pin assignment of a top-level I/O port and the minimal clock rate. During the implementation process, an IiO signal of the top-level module must be mapped to a physical pin of the FPGA device. Since the peripherals' I/O signals are already permanently connected to the designated FPGA's pins on the prototyping board, we must ensure that the signals are mapped to the corresponding pins. The other type of constraint is about timing, which specifies the minimal clock frequency to facilitate the oscillator of the board. The constraint information is stored in a text file with an extension of .ucf (for the user constraint file). There are several ISE tools to specify and generate the constraint file. Since all of our experiments are done in the same prototyping board, the constraints (i.e., pin assignment and clock frequency) remain the same.

During synthesis, the synthesis engine compiles the design to transform HDL sources into an architecture-specific design netlist. The ISE software supports the use of Xilinx Synthesis Technology (XST). After synthesis, one can run design implementation, which converts the logical design into a physical file format that can

be downloaded to the selected target device. Using the Project Navigator Design Goals and Strategies, one can modify process properties to control the implementation and optimization of the design.

Invoking the synthesis and implementation procedure is very simple. The module to be synthesized is assigned as top level before synthesis. Even after checking the syntax, the code may contain constructs that cannot be synthesized or may lead to poor implementation (such as a combinational loop). The error and warning messages are displayed in the console tab of the transcript window. The problems are corrected and the simulation and synthesis processes are repeated if needed.

3.2.4 Generating and downloading connfiguration file to the FPGA

In computing, configuration files, or config files configure the initial settings for some computer programs. They are used for user applications, server processes and operating system settings. There are several types of configuration files. A Bitstream file (*.bit) is used to configure an FPGA. After initializing a chain or adding a device, a configuration file is prompted. This is the file that is used to program the device.

Digilent Adept is a powerful application which allows for configuration and data transfer with Xilinx logic devices. Downloading the configuration file to FPGA is done using Digilent Adept. When using Digilents Adept software for programming, a dialog box available from the Config tab can be used to select a configuraiton file. Pressing the Program button will program the FPGA with the selected file. This may result in a warning that the configuration file was built for an unknown device; however, this warning can be ignored. A .bit file can be associated with the FPGA by right-clicking on the FPGA icon, selecting the desired .bit file, and clicking Program.

An alternative way to configure the FPGA is to download the configuration file to a PROM and load the configuration file from the PROM.

3.3 Overview of VHDL

The VHSIC(Very High Speed Integrated Circuit) Hardware Description Language is an industry standard language used to describe hardware from the abstract to the concrete level. VHDL is a powerful language with numerous language constructs that are capable of describing very complex behavior. VHDL Descriptions consist of primary design units and secondary design units. The primary design units are the Entity and the Package. The secondary design units are the Architecture and the Package Body. Secondary design units are always related to a primary design unit. Libraries are collections of primary and secondary design units. A typical design usually contains one or more libraries of design units.

All designs are expressed in terms of entities. An entity is the most basic building block in a design. The uppermost level of the design is the top-level entity. All entities that can be simulated have an architecture description. The architecture describes the behavior of the entity. A single entity can have multiple architectures. One architecture might be behavioral while another might be a structural description of the design. A configuration statement is used to bind a component instance to an entity-architecture pair. A configuration can be considered like a parts list for a design. A process is the basic unit of execution in VHDL. All operations that are performed in a simulation of a VHDL description are broken into single or multiple processes.

3.4 Overview of ISIM Simulator

ISim is an abbreviation for ISE Simulator, an integrated HDL simulator used to simulate Xilinx FPGA and CPLD designs. ISim provides a complete, full-featured HDL simulator integrated within ISE. HDL simulation now can be an even more fundamental step within your design flow with the tight integration of the ISim within your design environment. After compiling the testbench and corresponding files, we can perform the simulation and examine the resulting waveform. This corresponds to running the circuit in a virtual lab bench and checking

the waveform in a virtual logic analyzer. This simulator that lets you perform functional (behavioral) and timing simulations for VHDL.

The Xilinx ISE Design Suite provides an integrated flow with ISim that lets you launch simulations directly from the ISE Project Navigator. Simulation commands that prepare the ISim simulation are generated, and automatically run in the background when simulating a design.

ISIM has many key features like Standalone Waveform viewing capabilities, debug capabilities, waveform tracing, waveform viewing, HDL source debugging.

CHAPTER 4

THUMB PROCESSOR

Everything should be made as simple as possible, not simpler.

—Albert Einstein

The Thumb processor derives its name from the 16 bit compressed thumb instruction set of the ARM processors. All the internal registers are 16 bit long as against the thumb mode in ARM where the same registers as those used for ARM were used. An important distinction to make here is that though inspired from the thumb mode in ARM the Thumb processor is implemented as an independent 16 bit processor.

4.1 Block Diagram

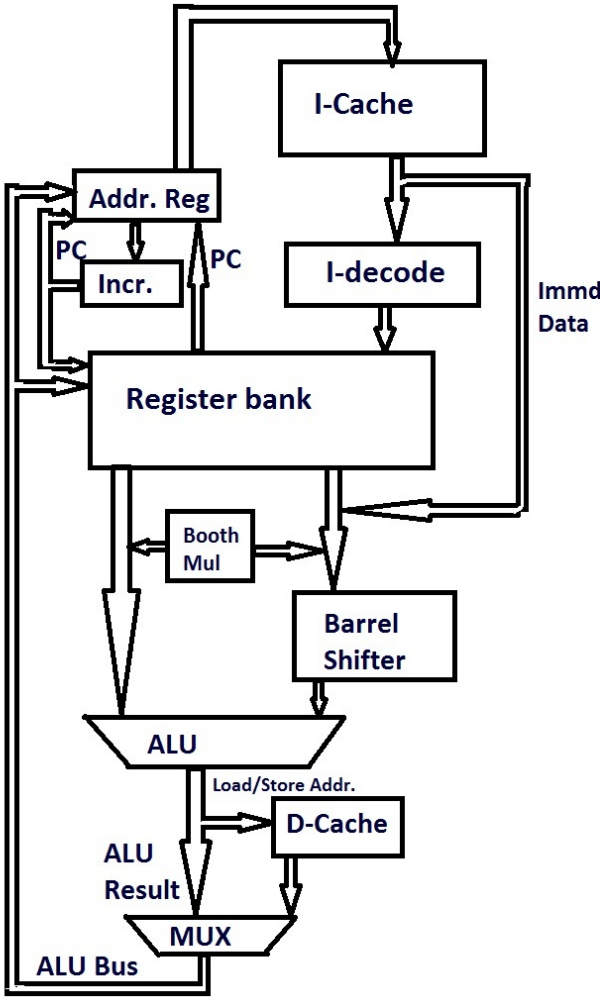

Figure 4.1 Block Diagram Of Thumb Processor

4.2 Thumb Features

The Thumb Processor has notably distinct features making it stand out from the parent ARM processor. Some of them are:

1. 16 bit RISC processor
2. Load and Store Architecture
3. It has 16 bit address bus and hence can access upto 65536 memory locations
4. It consists of 16 general purpose registers each of 16 bit
5. It has 256 x 16 bits of RAM
6. It has 16 bit Program Counter, Stack Pointer and Link Register which are a part of general purpose registers
7. Simple but powerful and very dense 16 bit Instruction Set
8. High code density
9. Tested at a clock frequency of 50 MHz
10. Multi cycle execution of all instructions along with 5 stage pipeline architecture.

4.3 Thumb Architecture

Reduced instruction set computing, or **RISC** , is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. It is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures. A computer based on this strategy is a *reduced instruction set computer*, also called *RISC*. The opposing architecture is called complex instruction set computing, i.e. CISC.

The Thumb processors instruction set consists of 16-bit RISC instructions. The Thumb instruction set provides most of the functionality required in a typical application. Arithmetic and logical operations, load/store data movements, and conditional and unconditional branches are supported.

4.3.1 Harvard Architecture

Harvard Architecture has some of these distinguishing features:

1. Physically separates storage and signal pathway for instructions and data.
2. Generally, the number of Instruction bits is greater than the number of data bits.
3. For some computers, the Instruction memory is read-only.
4. In cases without caches, the Harvard Architecture is more efficient than von-Neumann.

4.3.2 Modified Harvard Architecture

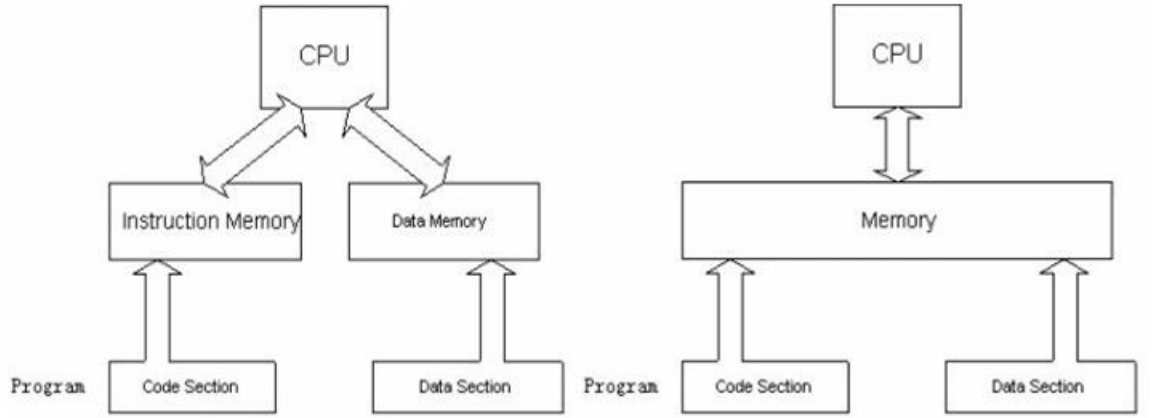

Figure 4.2 Harvard Architecture, Modified Harvard Architecture(from left to right)

A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The most common modification includes separate instruction and data caches backed by a common address space. While the CPU executes from cache, it acts as a pure Harvard machine. When accessing backing memory, it acts like a von Neumann machine (where code can be moved around like data, a powerful technique). This modification is widespread in modern processors such as the ARM architecture andX86 processors. It is sometimes loosely called a Harvard architecture, overlooking the fact that it is actually ”modified”.

CHAPTER 5

BUILDING A DATAPATH

To understand a program, you must become both the machine and the program.

—A. Perlis

The simplest datapath might attempt to execute all instructions in one clock cycle. This means that any element can be used only once per instruction. So these elements have to be duplicated.
If possible datapath elements can be shared by different instruction flows. Therefore multiple connections to the input must be realised. This is commonly done by a multiplexer.

5.1 Major Components

At first we look at the elements required to execute the instructions and their connection. The first element needed is a place to store the instructions fetched from the program memory. This Instruction queue is used to hold and supply instructions for execution.

The address must be kept in the Program Counter (PC), and in order to increment the PC to the address of the next instruction, we also need an incrementer.

After fetching one instruction from the instruction memory, the program counter has to be incremented so that it points to the address of the next instruction.

This is realised by the datapath shown in figure. The program counter value is incremented by 1 and this new value goes to the address register to fetch the next instruction.

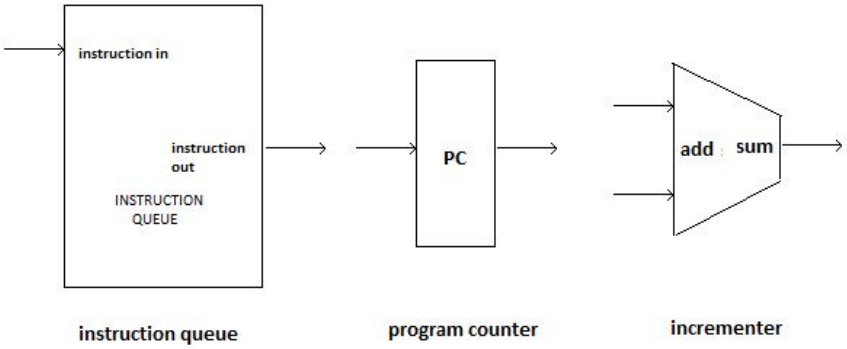

Figure 5.1 Instruction Queue, Program Counter and Implementer

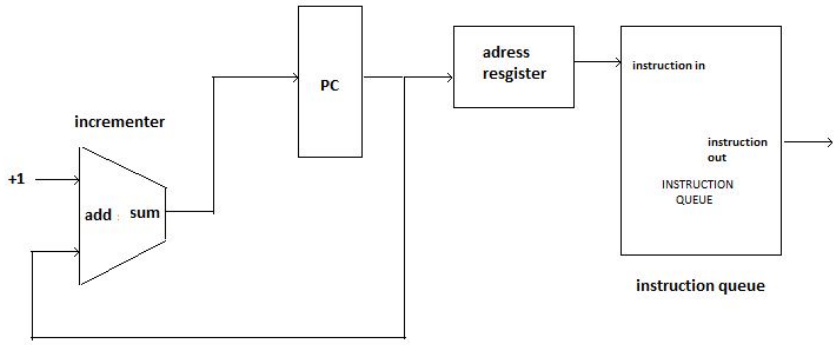

Figure 5.2 Portion of Datapath for Fetching Instruction and Incrementing PC

5.2 Operation of Datapath

The design which we have developed assumes that each instruction starts when it has arrived in the instruction queue.The processor must start in a known stage. Usually requires reset input to cause it to start executing instructions from a known address.The instruction executes in the following stages:

1. The instruction is fetched and the PC is incremented.

2. The instruction to be executed is fetched from the queue. The PC increments by one and points to the next instruction to be executed.

3. The required operands are fetched. The two operands are read from the register file if the instruction uses register addressing mode and if immediate addressing mode is used one immediate operand is read and one register operand is read.

4. Perform the desired operation. The alu operates on the two operands depending upon the aluselect(3:0) and generates the result

5. The result is stored. The alu result generated in the previous stage is written back into the destination register in the register file.

5.3 RTL design

The next step is to determine exactly the control signals that are required to cause the datapath to carry out the full set of operations. We assume that all the registers change state on the rising edge of the input clock, and where necessary have control signals that may be used to prevent them from changing on a particular clock edge.

A suitable register organization along with datapath is shown in the fig. This shows the enable register on all registers, function select lines to alu, the mode control lines to barrel shifter, the select control lines for multiplexers, the control line for the data memory and the memory read and write control lines.

5.4 Design of control unit

A clocking methodology defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes, because if a signal is written the same time it is read the value of read could correspond to the old value, the newly written value, or even some mix of the two! Computer designs cannot operate on such unpredictability. A clocking methodology is designed to ensure predictability.

The control unit provides this clocking methodology. A rising edge triggered clocking scheme is used which means that any values stored in the sequential logic element are updated only on the rising clock edge. Because only state elements can store a date value, any collection of combinational logic must have its inputs come from a set of state elements and outputs written into a set of state elements. The input elements are the values written in the previous clock cycle, while the outputs are the values that can be used in a following clock cycle.

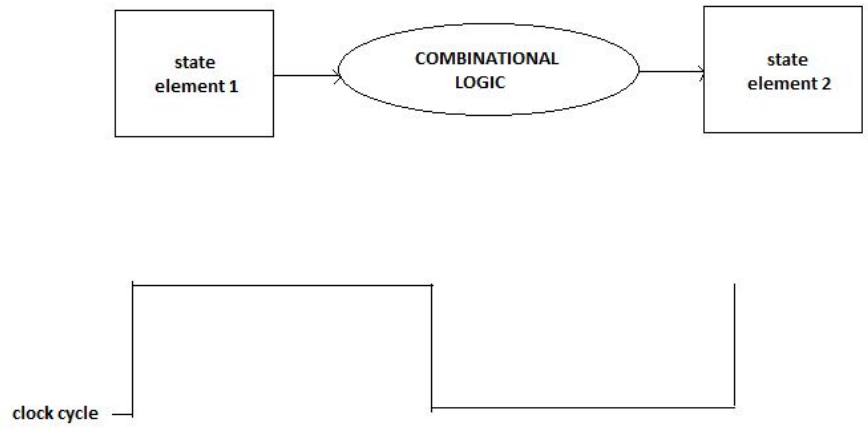

Figure 5.3 Relation between Combinational Logic State Elements and Clock

The control logic simply has to decode the current instruction and generate the appropriate levels on the datapath control signals, using the control inputs from the datapath where necessary.

5.5 Muticycle Implementation

In modern designs a single cycle implementation of a processor is not used, because it is inefficient. A clock cycle must have the same length for every instruction and therefore it is determined by the longest possible path. Almost this is the path of the load word instruction which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory and the register file again. However, a single cycle implementation can be used for a small instruction set. But if the machine gets more powerful there can be used

thousands of functional units and then the longest path causes the cycle time.

To avoid the disadvantages of the single cycle implementation , a multicycle implementation is used.This technique divides each instruction into steps and each step is executed in one clock cycle. The multicycle implementation allows a functional unit to be used more than once in a instruction, so that the number of functional units can be reduced.The major advantage of a multicycle design is the ability to share functional units within an execution.This sharing can can help reduce the amount of hardware required. The ability to allow instructions to take different numbers of clock cycle and the ability to share functional units within the execution of a single instruction are major advantages of multicycle implementation. Further we have used a multicycle implementation along with pipeline which improves the performance by increasing the instruction throughput.

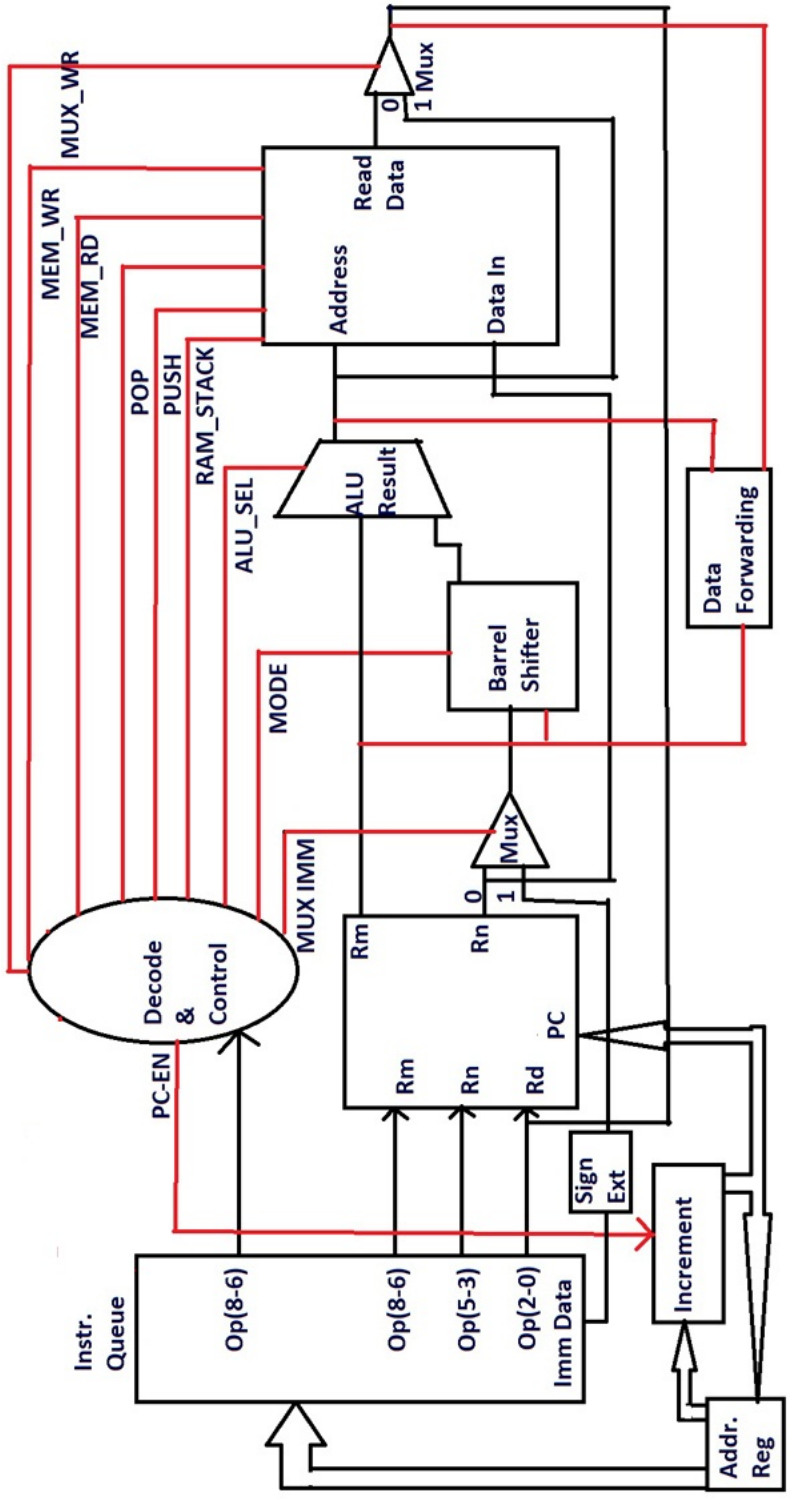

Figure 5.4 Multicycle Implementation of the Datapath

CHAPTER 6

MODULE SPECIFICATIONS

God lies in the details.

—Anonymous

6.1 Instruction queue

6.1.1 Functional Description

Instruction Queue allows the next instructions or data to be fetched from memory while the processor is executing the current instruction. The memory interface is usually much slower than the processor execution time, so this decouples the memory cycle time from the execution time. Instruction queue is used to prefetch the next instructions in a separate buffer while the processor is executing the current instruction.With a five stage pipeline, the rate at which instructions are executed is four times that of sequential execution.

The processor usually has two separate units for fetching the instructions and for executing the instructions.The implementation of a pipeline architecture is possible only if the bus interface unit and the execution unit are independent. While the execution unit is decoding or executing an instruction which does not require the use of the data and address buses, the bus interface unit fetches instruction opcodes from the memory.This process is much faster than sending out an address, reading the opcode and then decoding and executing it

6.1.2 RTL Schematic

The RTL Schematic for Instruction Queueis shown in figure 6.1.

- **Instr_in**: PC points to this address
- **Instr_ptr**: Points to the location of the Instruction Queue

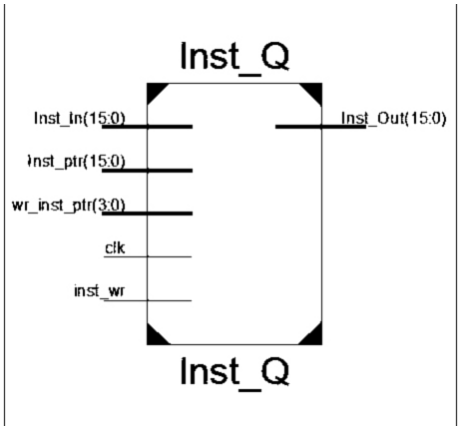

Figure 6.1 Instruction Queue

- **wr_instr_ptr**: Points to the last Instruction(length) of the queue
- **clk**: for synchronisation
- **instr_wr**: Enable signal used for updating the Queue
- **instr_out**: Instruction sent from queue to the decoder

6.1.3 Simulation Results

The simulation waveform of Instruction Queue is shown in Fig. 6.2.

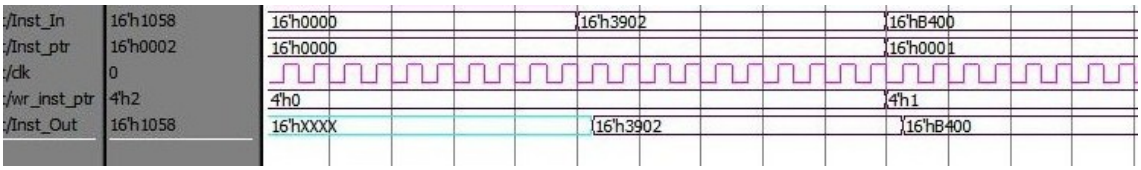

Figure 6.2 Instruction Queue Simulation

6.2 Decoder

6.2.1 Functional Description

In processor design, the instruction decoder is the part of the processor that converts the bits stored in the instruction register or, in processors that have microcode, the microinstruction into the control signals that control the other parts of the processor.

Even the incredibly simple microprocessor will have a fairly large set of instructions that it can perform. The collection of instructions is implemented as bit patterns, called op-codes, each one of which has a different meaning when loaded into the instruction register. The op-code decoder is the middle state of our processor pipeline. Therefore its inputs are defined by the outputs of the previous stage and its outputs are defined by the inputs of the next stage. Every instruction can be broken down as a set of sequenced operations that manipulate the components of the microprocessor in the proper order. Some instructions, might take two or three clock cycles, others might take five or six clock cycles. The Instruction Decoder reads the next instruction in from the

instruction queue, and sends the component pieces of that instruction to the necessary destinations.

The 16 bit instruction from the instruction queue is given as an input to the decoder, on the basis of which it generates the necessary timing and control signals required to perform the operation specified in the instructions. The flags from PSR also form the decoder inputs. The inputs and outputs of the decoder are explained in detail.

6.2.2 RTL Schematic

The RTL Schematic of decoder is shown in Fig. 6.3.

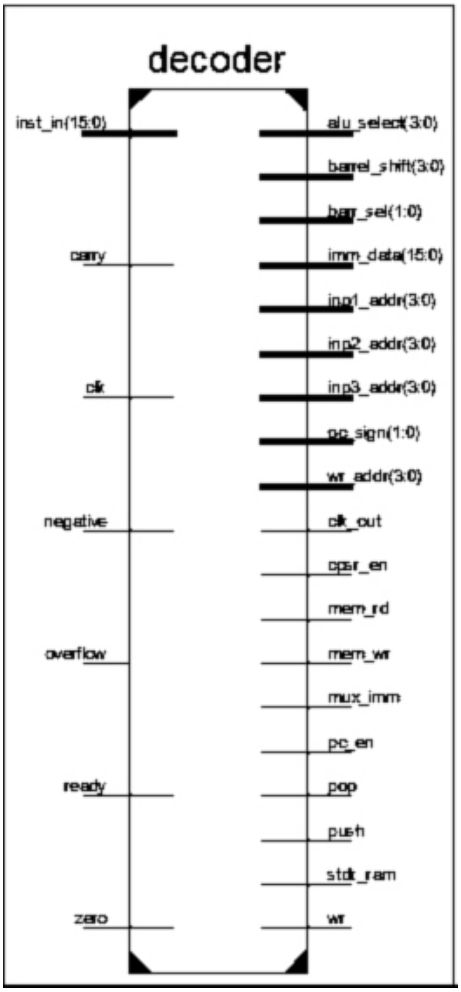

Figure 6.3 Decoder

- **inst_in(15:0)**: is the instruction being decoded
- **clk** is the clock signal. The opcode decoder is a pure pipeline stage so that no internal state is kept between clock cycles. The output of the opcode decoder is a pure function of its inputs.
- **carry,negative,overflow,zero**: the flag inputs from PSR(program status register) are given as inputs to the decoder to be used in making decisions regarding conditional branch instructions

- **ready**:This signal is given to the decoder by ram controller when the instruction queue is full. It is an indication to decoder to start decoding the instructions which are present in the instruction queue.
- **alu_select(3:0)** is a set of 4 lines to decide which ALU operation has to be performed out of the 16 available options, including addition, subtraction, logical AND, logical OR, etc.
- **barrel_shift(3:0)** is a set of 4 lines giving a range of 0 to 15, this is the number of places by which the contents of data input to barrel shifter has to shifted.
- **barr_sel(1:0)** are the two inputs which go to the barrel shifter, one for selecting the direction of rotation and other to select the type of rotation- arithmetic or logical.
- **imm_data(15:0)** are 16 lines used to pass direct data from user to register bank/stack/ALU.
- **inp1_addr(3:0)** are four output lines from decoder to the register bank. They are used to select a register from the set of 16 registers in them for read operation.
- **inp2_addr(3:0)** are four output lines from decoder to register bank. They are used to select the second register from the set of 16 registers in them for read operation.
- **inp3_addr(3:0)** are four output lines from decoder to register bank. They are used to select the third register from the set of 16 registers in them for read operation.
- **wr_addr(3:0)** are a set of four lines used to select a register from a set of 16 registers from register bank for write operation.
- **cpsr_en**: is used to enable the cpsr
- **mem_wr**: is used to enable memory write
- **mem_rd**:is used to enable memory read
- **mux_imm** is an input line to the barrel shifter used for selecting data input type, i.e. addressing mode, immediate or register.
- **pc_en** is used to enable the program counter, to transfer its contents or to increment
- **pop** is the input line to the stack, used to pop i.e. remove and read data from the top of the stack.
- **push** line is the input line to the stack used for writing data on the top of the stack.
- **stck_ram** is used to access the RAM as stack, i.e. enable stack opearations
- **wr** is used to enable writing data in register and stack.
- **clk_out** is used for synchronizing the modules which follow the decoder in the datapath

6.2.3 Simulation Results

The simulation waveform of Decoder is shown in Fig. 6.4.

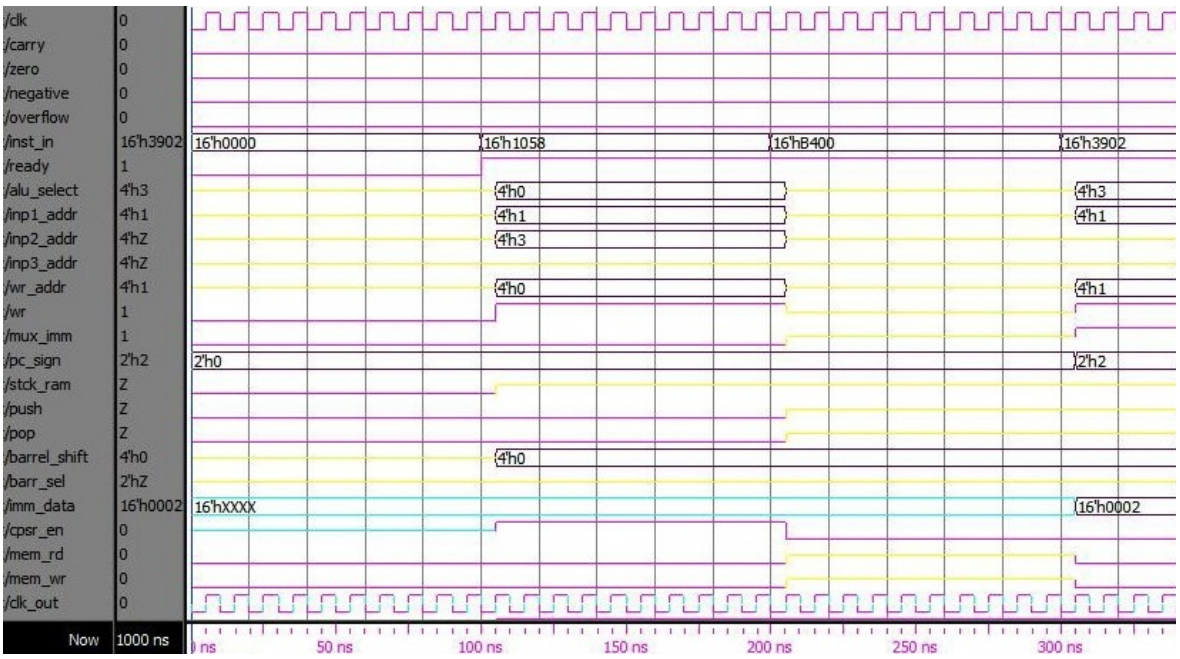

Figure 6.4 Decoder Simulation

6.3 Control Unit

6.3.1 Functional Description

The control unit is a component of a computer's central processing unit (CPU) which directs operation of the processor. It controls communication and co-ordination between input/output devices. It reads and interprets instructions and determines the sequence for processing the data. It directs the operation of the other units by providing timing and control signals. All computer resources are managed by the CU (Control Unit). It directs the flow of data between the Central Processing Unit (CPU) and the other devices.

The control unit implements the instruction set of the CPU. It performs the tasks of fetching, decoding, managing execution and, finally, storing results. The control unit may manage the translation of instructions (not data) to micro-instructions and manage scheduling the micro-instructions between the various execution units. On some processors, the control unit may be further broken down into other units, such as an instruction unit or scheduling unit to handle scheduling, or a retirement unit to deal with results coming from the instruction pipeline. The control unit handles the main functions of the CPU. The main functions are to carry out instructions in the software and to direct the flow of data through the computer.

6.3.2 State Diagram

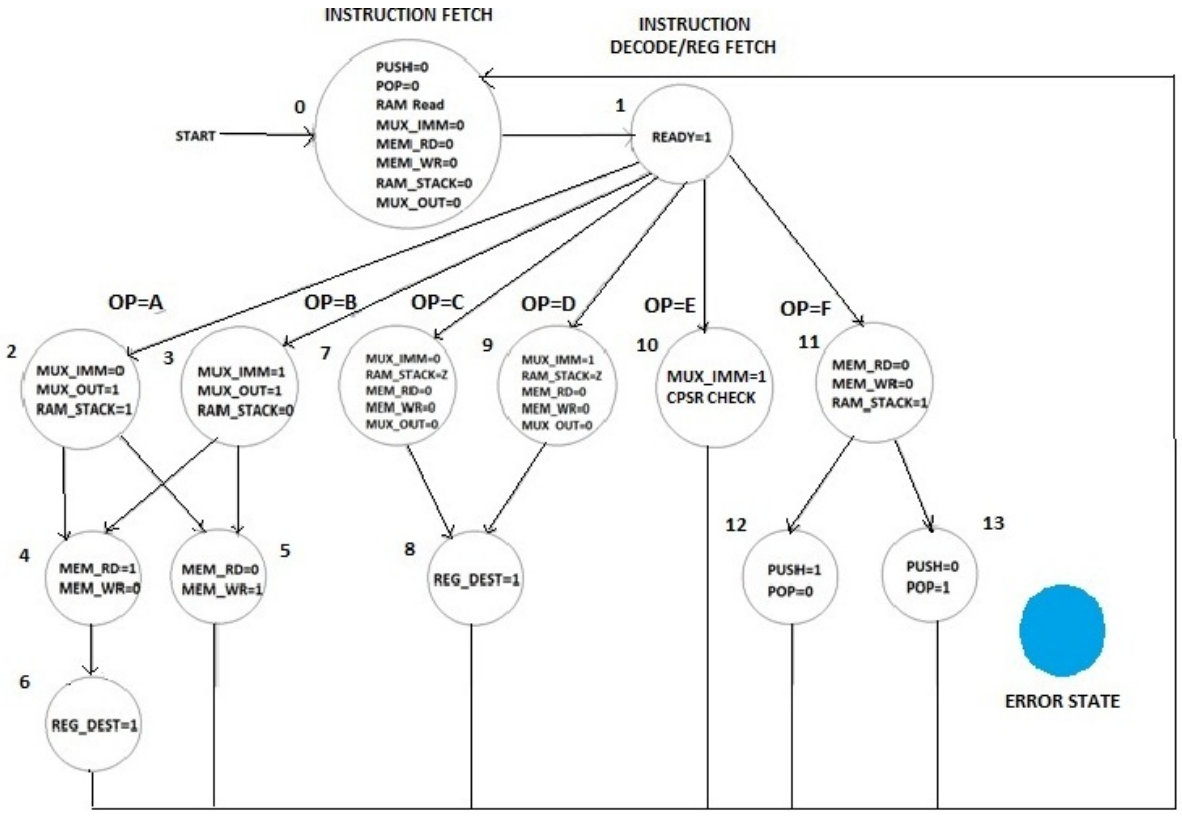

Figure 6.5 Control Unit State Diagram

Opcodes From The State Diagram:

- A- Load Or Store In Register Mode
- B-Load Or Store In Immediate Mode
- C- Data Processing Or Move In Register Mode
- D- Data Processing Or Move In Immediate Mode
- E-Branch Operations
- F-Stack Operations

States:

- 0- Instruction Fetch
- 1-Instruction Decode/Register Fetch
- 2- Memory Address Calculation For Register Mode
- 3- Memory Address Calculation For Immediate Mode
- 4- Memory Access For Load
- 5- Memory Access For Store
- 6- Memory Read Completion
- 7- Execution For Register Mode
- 8-Execution For Immediate Mode
- 9- Execution Complete
- 10- Branch Completion
- 11- Stack Selection
- 12- Push Completion
- 13-Pop Completion

6.3.3 RTL Schematic

The RTL Schematic of Control Unit is shown in Fig.6.6. It synchronizes all the input signals and sends the

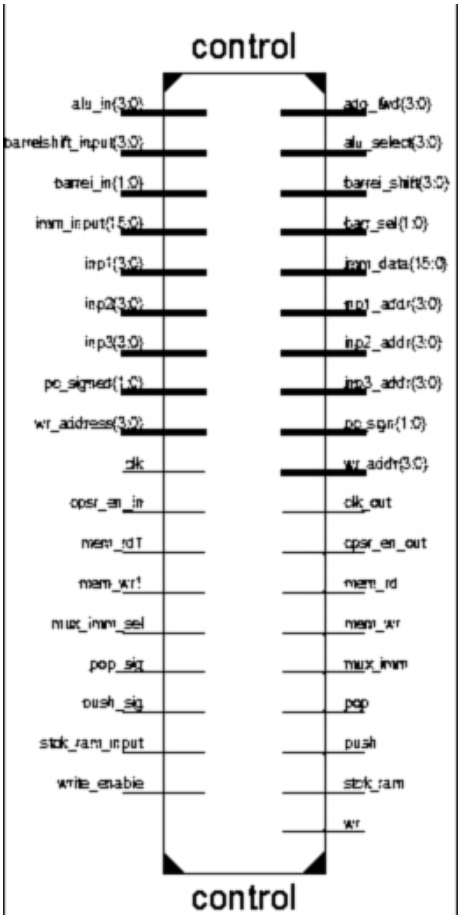

Figure 6.6 Control Unit

synchronized signals as outputs.

6.3.4 Simulation Results

The simulation of Control Unit is shown in Fig. 6.7.

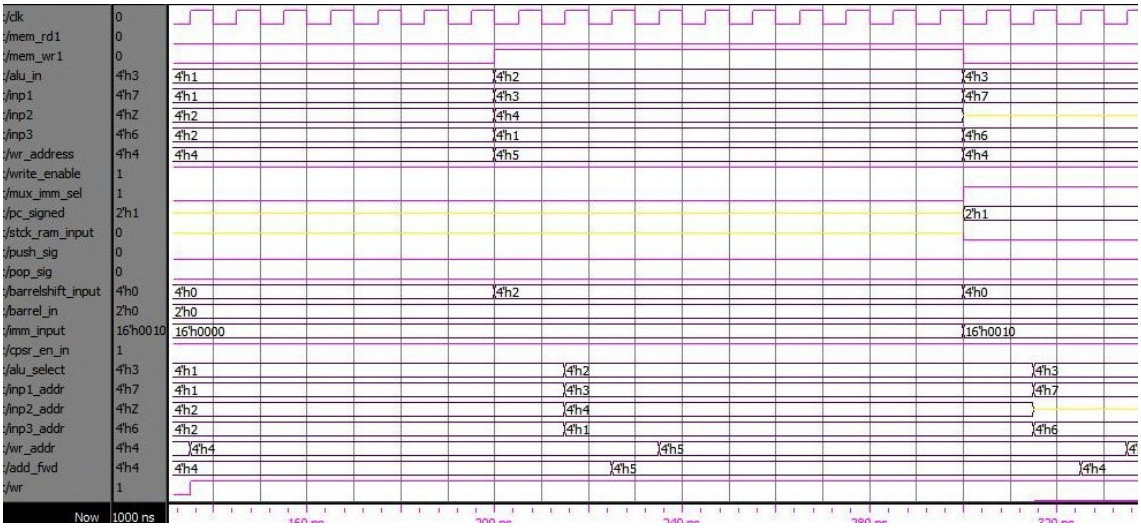

Figure 6.7 Control Unit Simulation

6.4 Register File

6.4.1 Functional Description

A register file is an array of processor registers which are used to stage data between memory and the functional units on the chip. The register file is part of the architecture and visible to the programmer, as opposed to the concept of transparent caches. Register files may be clubbed together as banks. A simple register file has a minimum of two addresses, two data inputs and two outputs, but this is extended to include the destination register as well as the read and write inputs.The processor register file has 16 registers- r0 to r15 as shown in Fig. each of which is 16 bits wide.

1. **General purpose registers (r0 to r12)** - 13 of these 16 registers r0 to 12 are general purpose and can be used to store and hold the result/operands of ALU operations temporarily.
2. **Link register (r13)** - Link register is a special purpose register which holds the address to return to when a function call completes. r13 is used as the subroutine link register and receives a copy of r15 when a Branch and Link instruction is executed. Register r13 receives the return address when a Branch with Link (BL or BLX) instruction is executed. It may be treated as a general purpose register at all other times.
3. **Stack pointer (r14)** - A stack is usually implemented as a linear data structure which grows up (an ascending stack) or down (a descending stack) memory as data is added to it and shrinks back as data is removed. A stack pointer holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a full stack), or by pointing to the vacant slot where the next data item will be placed (an empty stack). Therefore the default value of r14 is 255.
4. **Program counter (r15)**- The PC is incremented after fetching an instruction, and holds the memory address of (points to) the next instruction that would be executed. Instructions are usually fetched sequentially from memory, but control transfer instructions change the sequence by placing a new value in the PC. These include branches (sometimes called jumps), subroutine calls, and returns. The Program Counter (PC) is accessed as PC (or r15). It is incremented by the size of the instruction executed which is 2 bytes in thumb mode. Branch instructions load the destination address into PC.

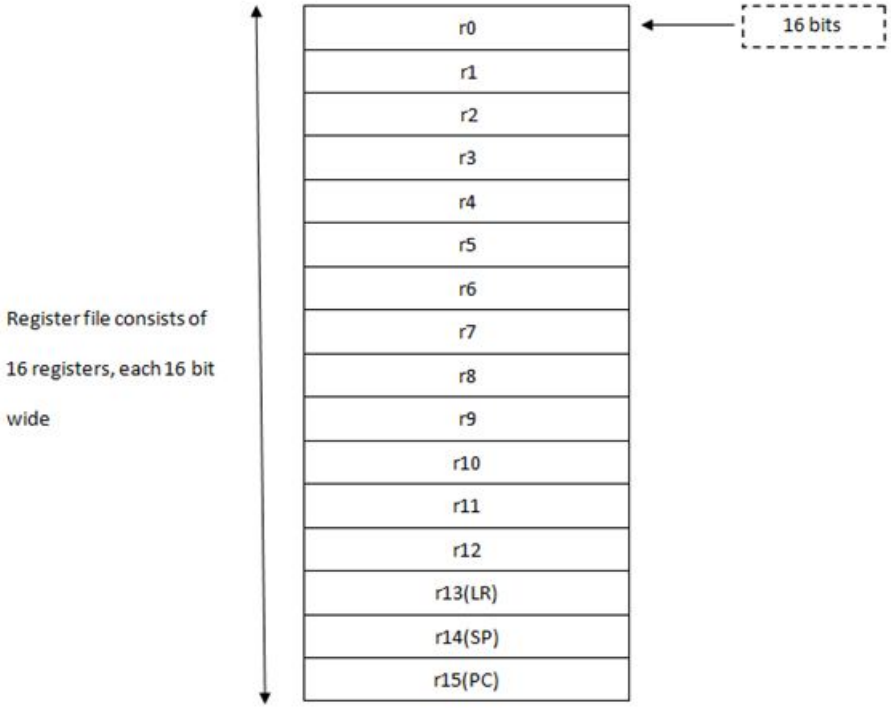

Figure 6.8 Register Bank

6.4.2 RTL Schematic

The RTL Schematic of Register File is shown in Fig.6.9.

- **link_in(15:0)**: used to give input to the link register, of the address to return to when a function call completes
- **r_addr1(3:0)**: defines the first source register to hold one of the operands.
- **r_addr2(3:0)**: defines the second source register to hold the other operand.
- **r_addr3(3:0)**: defines the third operand destination register
- **stack_in(6:0)**: takes in data input for stack
- **w_addr(3:0)**: defines the destination register for storing the result of an operation from the output of the ALU.
- **w_data(15:0)**: these 16 lines hold the result of operation from the ALU which is to be written into the register specified by w_addr(3:0)
- **clk**:to synchronize the operations of the other modules with the register bank operations.
- **reset**: is used to reset all the register values to zero.
- **wr_en**: this pin goes high when data is to be written in any of the registers
- **pcin(15:0)**: these input lines carry the 16 bit value to be loaded in the program counter(r15)
- **r_data1(15:0)**: carries the first 16 bit operand output to the ALU input

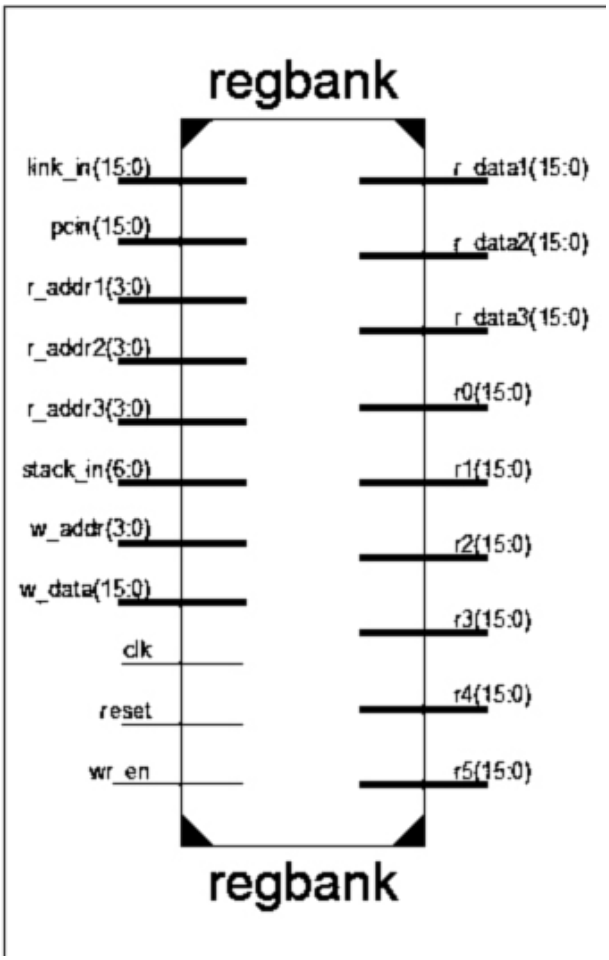

Figure 6.9 Register File

- **r_data2(15:0)**: carries the second 16 bit operand output to the other ALU input
- **r_data3(15:0)**: carries the third 16 bit opearand output to the ALU input
- **r0,r2,r3,r4,r5(15:0)**:used to access the contents of the 6 registers

6.4.3 Simulation Results

The simulation waveform of Register File is shown in Fig.6.10.

Figure 6.10 Register File Simulation

6.5 Sign Extension Block

6.5.1 Functional Description

Sign extension is the operation, in computer arithmetic, of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value. This is done by appending digits to the most significant side of the number, following a procedure dependent on the particular signed number representation used.It is useful in carrying out backward jumps and for performing signed operations.

For example, if six bits are used to represent the number ”00 1010” (decimal positive 10) and the sign extend operation increases the word length to 16 bits, then the new representation is simply ”0000 0000 0000 1010”. Thus, both the value and the fact that the value was positive are maintained.

6.5.2 RTL Schematic

The RTL Schematic of Sign Extension Unit is shown in Fig. 6.11.

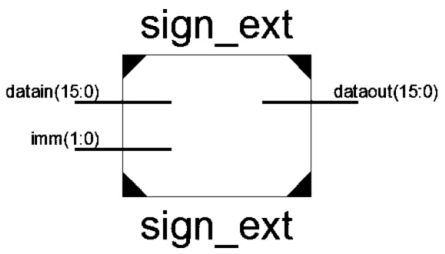

Figure 6.11 Sign Extension Unit

6.5.3 Simulation Results

The simulation waveform of Sign Extension Unit is shown in Fig. 6.12.

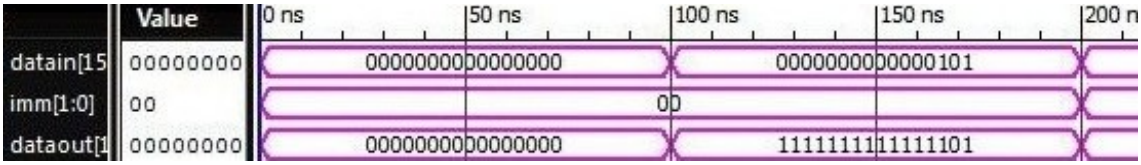

Figure 6.12 Sign Extension Simulation

- **datain(15:0)**:takes data input for extending sign of smaller numbers
- **imm(1:0)**: tells the length of the immediate value
- **dataout(15:0)**: gives out the sign-extended data

6.6 Barrel Shifter

6.6.1 Functional Description

A barrel shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle. It can implement two types of shifting, arithmetic and logical. It can be implemented as a sequence of multiplexers (mux.), and in such an implementation the output of one mux is connected to the input of the next mux in a way that depends on the shift distance. It can typically specify the direction(left or right), the type of shift(circular or logical) and the amount of shift 0 to n-1 bits).

A common usage of a barrel shifter is in the hardware implementation of floating-point arithmetic. For a floating-point add or subtract operation, the significands of the two numbers must be aligned, which requires shifting the smaller number to the right, increasing its exponent, until it matches the exponent of the larger number. This is done by subtracting the exponents, and using the barrel shifter to shift the smaller number to the right by the difference, in one cycle. If a simple shifter were used, shifting by n bit positions would require n clock cycles. The length of the barrel shifter is 16 bits i.e. a maximum shift of 16 bits is possible.

6.6.2 RTL Schematic

The RTL Schematic of Barrel Shifter is shown in Fig. 6.13.

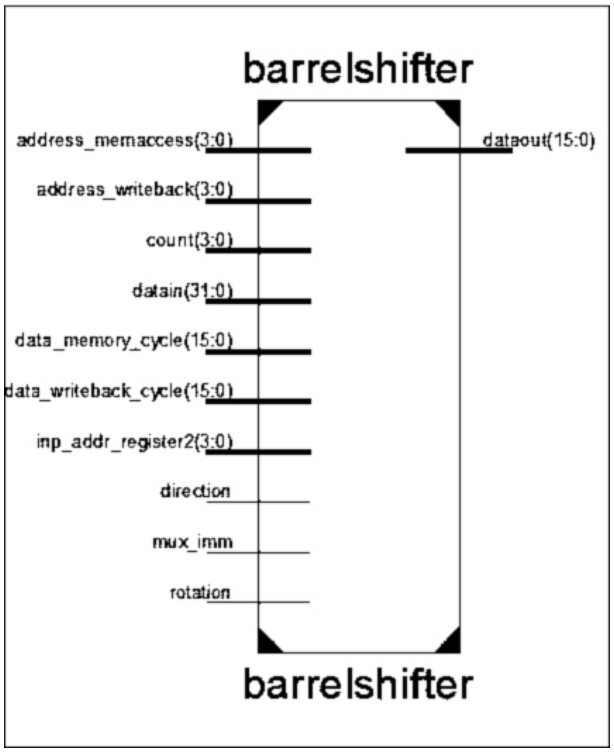

Figure 6.13 Barrel Shifter

- **address_memaccess(3:0)**: connected to control and timing unit wr_address of forward memory for memory access
- **address_writeback(3:0)**:connected to control unit's wr_address for memory access
- **count(3:0)**: are 4 input lines used to give the number of places by which the content in the barrel shifter has to be shifted.
- **datain(15:0)**: is the 16-bit data input to the barrel shifter.
- **data_mem_cycle(15:0)**: connected to the ALU's output
- **data_writeback_cycle**: connected to the memory access' data output
- **inp_addr_register(3:0)**:selects the register to be shifted
- **direction**: is used to decide the direction of shifting data, i.e. left or right.
- **mux_imm**: used to select imm/reg operand
- **rotation**: is used to select the type of shifting, i.e. arithmetic or logical.
- **dataout(15:0)**:is used to output the shifted data.

6.6.3 Simulation Results

The simulation waveform of Barrel Shifter:

Figure 6.14 Barrel Shifter simulation

6.7 ALU

6.7.1 Functional Description

An arithmetic logic unit (ALU) is a digital circuit that performs integer arithmetic and logical operations. The ALU is a fundamental building block of the central processing unit of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. An ALU processes numbers using the same formats as the rest of the digital circuit.

The inputs to the ALU are the data to be operated on (called operands) and a code from the control unit indicating which operation to perform. Its output is the result of the computation. The ALU also takes or generates inputs or outputs a set of condition codes from or to the CPSR (explained in the further sections). These codes are used to indicate cases such as carry-in or carry-out, overflow, divide-by-zero, etc.

6.7.2 RTL Schematic

The RTL Schematic of ALU is shown in Fig. 6.15.

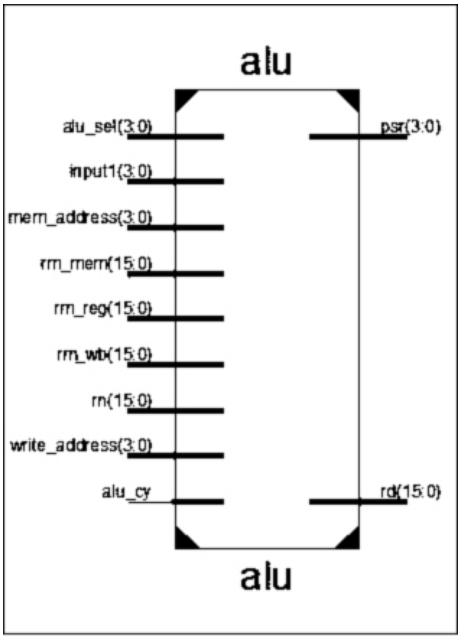

Figure 6.15 ALU RTL Schematic

- **alu_sel(3:0)**-The 4-bit input, alu_sel" viz. ALU SELECT decides which arithmetic or logical operation is to be performed by the ALU
- **input1(3:0)**:
- **mem_addr**: connected to the wr_address of the memory
- **rm_mem(15:0)**: If input addres is equal to memory address, then rm_mem is stored in Rm
- **rm_wb(15:0)**:If input address is equal to write address, then rm_wb is stored in Rm
- **rm_reg(15:0)**: If none of the above happens then this case is executed
- **alu_cy**-is the one bit carry(or borrow, in case of subtraction)input to the ALU
- **rd(15:0)**- the result of the ALU operation is output on these lines.
- **psr(3:0)**- Depending upon the result, the status of the flags is indicated in the PSR (Program Status Register).

Following is the table showing the combinations of alu_sel lines and the operation performed by the ALU for that combination:

Table 6.1 ALU Operations

ALU_SEL	Operation Performed
0000	ADD(Addition)
0001	SUB(Subtraction)
0010	CMP(Compare)
0011	MOV(Move)
0100	AND
0101	EXOR
0110	OR
0111	Complement
1000	MUL(Multiply)
1001	ADC(Add With Carry)
1010	SBC(Subtract With Borrow)

6.7.3 Simulation Results

The Simulation Waveform of ALU is shown in Fig. 6.16.

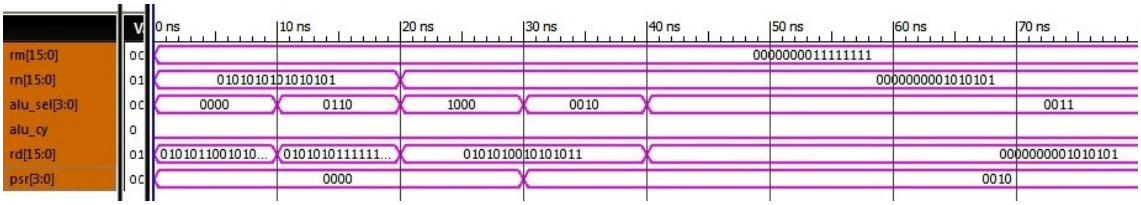

Figure 6.16 ALU Testbench Waveform

6.8 CPSR

The CPSR is used in user-level programs to store the condition code bits. These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken. The user-level programmer need not usually be concerned with how this register is configured.

15	14	13	12	11-0
N	Z	C	V	unused

Figure 6.17 CPSR Bit Structure

The condition code flags are in the top four bits of the register and have the following meanings:

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 16-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 16-bit result was zero).

- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: Overflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

6.8.1 Functional Description

6.8.2 RTL Schematic

The RTL Schematic of CPSR is shown in Fig. 6.18.

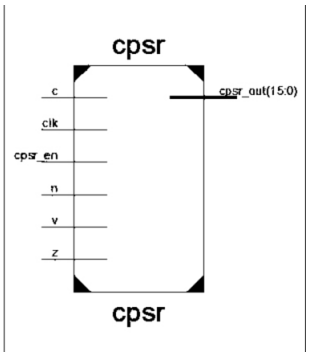

Figure 6.18 CPSR

- **c,n,z,v**: these are the flag bits explained above
- **clk**: this is used for synchronization
- **cpsr_en**: used to enable cpsr
- **cpsr_out(15:0)** used to store cpsr contents

6.8.3 Simulation Results

The simulation of CPSR is shown in Fig. 6.19.

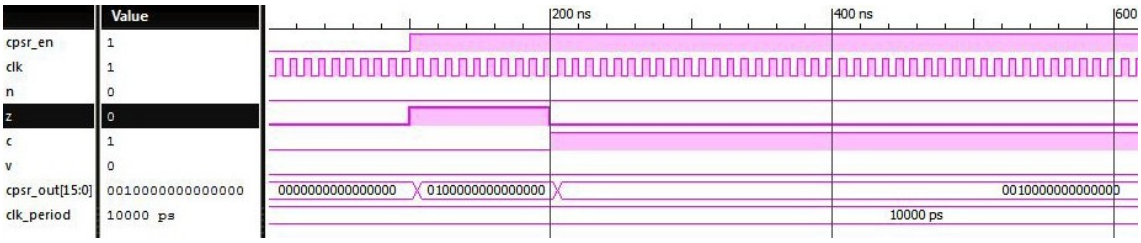

Figure 6.19 CPSR Simulation

6.9 Data Memory

6.9.1 Functional Description

The memory is organized into two segments-data segment and stack segment. The memory consists of 256 locations of 2 Bytes each. The memory is divided into 2 segments - data and stack each of 128 locations. 0 to 127 comprises of the data segment and 128 to 255 is the stack segment.

Data is synchronously written to or read from the memory with a data bus width of 16 bit. It is in chunks of 2 Bytes since one memory location is of 2 Bytes. The data area contains global and static variables used by the program that are explicitly initialized with a non-zero (or non-NULL) value.
The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. A ”stack pointer” register tracks the top of the stack; it is adjusted each time a value is ”pushed” onto the stack.

6.9.2 RTL Schematic

The RTL Schematic of Data Memory is shown in Fig. 6.20.

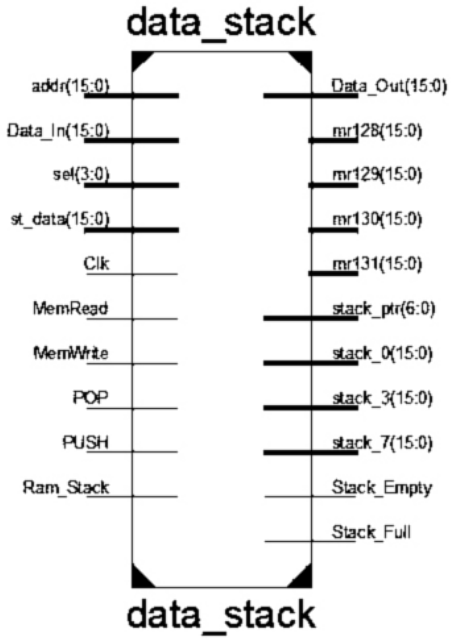

Figure 6.20 Data Memory

1. **addr(15:0)**: The 16 bit address bus input to holds the address of the data location to be accessed.
2. **data_in(15:0)**: Data to be pushed/write to memory
3. **sel(3:0)**: it is used to check the value of the stack pointer; to map its value from the from the register bank.
4. **st_data(15:0)**: it holds the value to be stored in the location given by the address.
5. **clk**: this signal is used as clock for the stack.
6. **MemRead**: enable to read data from data memory, enable for load

7. **MemWrite**: enable to write data into data memory, enable for store
8. **Pop**: it is used to pop data from the stack
9. **Push**: it is used to push data on to the stack
10. **Ram_Stack**: used to enable or disable RAM
11. **Data_out**: data to be popped/read from memory
12. **mr128(15:0)**: these lines are used to read from the memory location 128
13. **mr129(15:0)**: these lines are used to read from the memory location 129
14. **mr130(15:0)**: these lines are used to read from the memory location 130
15. **mr131(15:0)**: these lines are used to read from the memory location 131
16. **stack_ptr(6:0)**: it points to the top of stack
17. **stack_0(15:0)**: the content of data location 120 is moved on these lines
18. **stack_3(15:0)**: the content of data location 123 is moved on these lines
19. **stack_7(15:0)**: the content of data location 127 is moved on these lines
20. **stack_empty**: goes high when the stack is empty
21. **stack_full**: this output goes high when the stack is full

6.9.3 Simulation Results

The simulation of Data Memory is shown in Fig. 6.21.

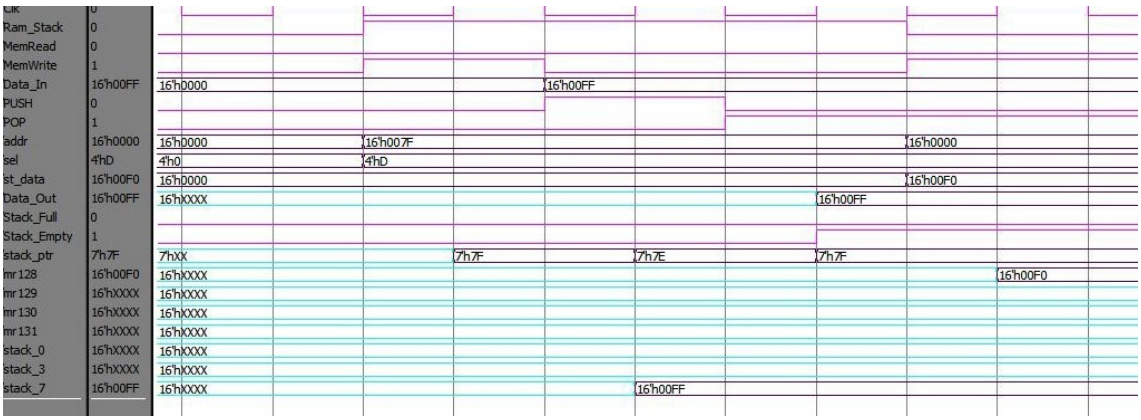

Figure 6.21 Data Memory Simulation

6.10 Program Counter

6.10.1 Functional Description

The program counter, or PC is a processor register that indicates where the computer is in its instruction sequence. Depending on the details of the particular computer, the PC holds either the address of the instruction being executed, or the address of the next instruction to be executed.

This instruction pointer is incremented automatically after fetching a program instruction, so that instructions are normally retrieved sequentially from memory, with certain instructions, such as branches, jumps and subroutine calls and returns, interrupting the sequence by placing a new value in the program counter. The program counter is incremented by one byte for fetching the next instruction from the memory using a separate adder called incrementer.

6.10.2 RTL Schematic

The RTL Schematic of Program Counter is shown in Fig. 6.22.

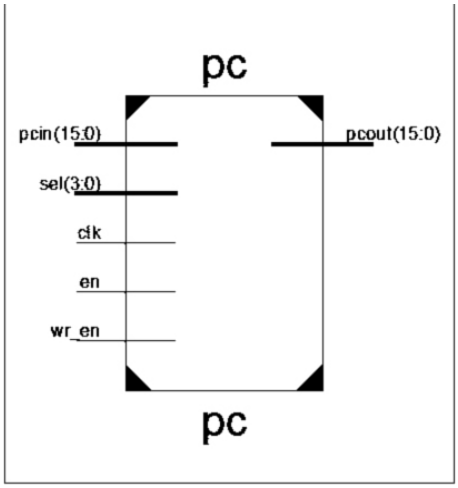

Figure 6.22 Program Counter Block Diagram

6.10.3 Simulation Results

The simulation waveform of Program Counter is shown in Fig. 6.23.

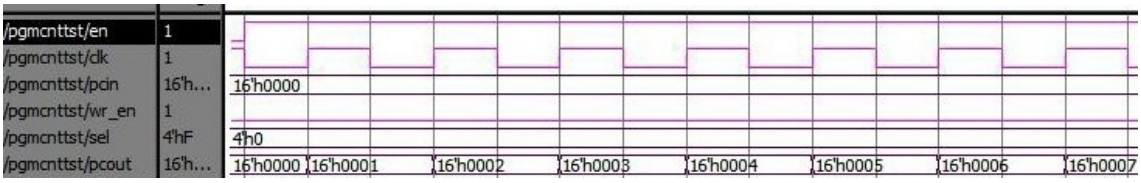

Figure 6.23 Program Counter Simulation

6.11 Input Output Module

6.11.1 Functional Description

It shows all the connections between the the registers of processor and FPGA switches and the connections to LCD on the FPGA. It is basically used for demonstration purposes on FPGA.

6.11.2 RTL Schematic

The RTL schematic of Input Output Module is shown in Fig. 6.24.

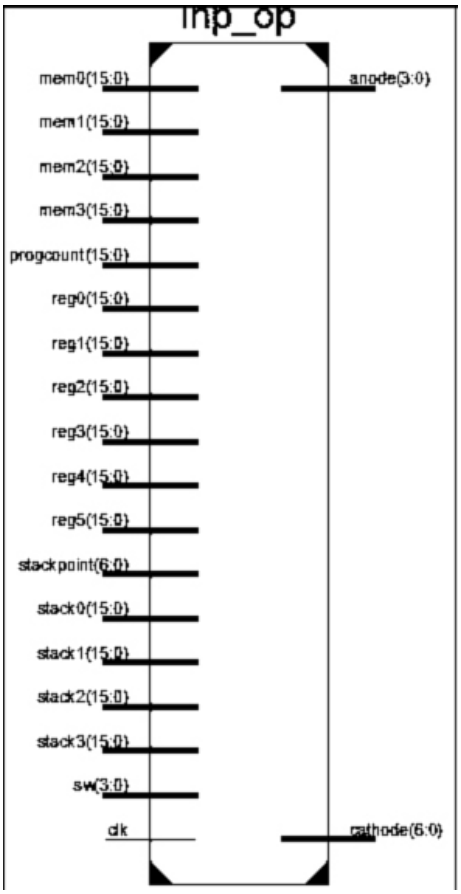

Figure 6.24 Input Output Module Block Diagram

6.12 Ram Controller

6.12.1 Functional Description

The Nexys 2 board contains onboard RAM that is external to the FPGA. The first is a 16 Mbyte RAM organized as 8Mega x 16 bit words. The RAM has a 23 bit address bus (22:0) and 16 bit bi-directional data bus DQ(15:0) as well as write enable and output enable signals which are active low signals.

6.12.2 RTL Schematic

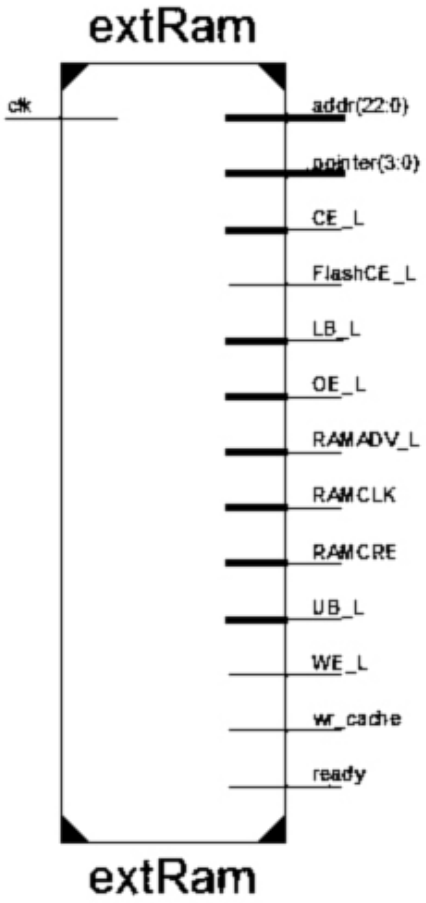

Figure 6.25 Ram Controller Block Diagram

- **Clk**: It receives the clock from the on board FPGA clock
- **addr**: It is the 23 bit address which points to the contents of the memory location of the RAM (where the opcode is stored) to be fetched.
- **CE_L** : This is a chip enable signal which is made low to select the RAM
- **UB_L**: This signal is made low in order to read from or write into the upper byte
- **LB_L**: This signal is made low in order to read from or write into the lower byte DQ(7:0)
- **FLASHCE_L**: In order to select RAM, this pin should always be made high in order to make sure that flash memory is disabled.
- **OE_L**: While reading from the RAM location, this has to be kept low. While writing, it should be kept high. The memory access time is 70 ns.
- **WE_L**: While writing into the RAM location, this is to be kept low for at least 45 ns. While reading, it should be kept high.

- **RAMCLK**: It is ignored and generally kept low while accessing RAM
- **RAMADV_L**: It is ignored and generally kept low while accessing RAM
- **RAMCRE**: It is ignored and generally kept low while accessing RAM
- **Ready**: This is given from the RAM controller to decoder and the instruction queue when the queue is being filled with instructions.
- **wr_cache**: This is made low when the instruction queue is full. pointer: It points to the location of the instruction queue.

6.13 Top Module

6.13.1 Functional Description

Our design is a hierarchial model. This means that the top module basically contains two or more submodules and connects all the submodules to each other with the help of port map in VHDL. So all the submodules in the design: PC, instruction queue, decoder, control unit, register file, ALU, Barrel shifter, sign extension and data memory are given proper connections to each other in the top module.

CHAPTER 7

5-STAGE PIPELINE

Many hands make light work.

—Anonymous

7.1 What is a Pipeline

The processor that we have implemented uses a 5-stage pipeline and therefore is much efficient compared to 8085, 8086 and 8051 processors/controllers. An instruction pipeline is a technique used in the design of computers to increase their instruction throughput(the number of instructions that can be executed in a unit of time). The basic instruction cycle is broken up into a series called a pipeline. Rather than processing each instruction sequentially (one at a time, finishing one instruction before starting the next), each instruction is split up into a sequence of steps different steps can be executed concurrently (by different circuitry), and indeed in parallel (at the same time).

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 7.1 Basic five-stage pipeline in a RISC machine

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Not all instructions will require every step, but most instructions will require most of the steps required in the Fig.7.1. These steps tend to use different hardware functions, for instance the ALU is probably only used in step 4. Therefore, if an instruction does not start before its predecessor has finished, only a small proportion of the processor hardware will be in use in any step. An obvious way to improve the utilization of the hardware resources, and also the processor throughput, would be to start the next instruction before the current one has finished. This technique is employed in pipelining, and is a very effective way of exploiting concurrency in a general-purpose processor. Taking the above sequence of operations, the processor is organized so that as soon as one instruction has completed step 1 and moved on to step 2, the next instruction begins step 1. This is illustrated in Fig. 7.1. In principle such a pipeline should deliver a five times speed-up compared with non-overlapped instruction execution; in practice things do not work out quite so well for reasons we will see below.

7.2 Advantages of Pipeline

Pipelining of instructions in CPU architecture gives major benefits. A few of them are:

1. The cycle time of the processor is reduced; increasing the instruction throughput. Pipelining doesn't reduce the time it takes to complete an instruction; instead it increases the number of instructions that can be processed simultaneously ("at once") and reduces the delay between completed instructions (called 'throughput').

2. The more pipeline stages a processor has, the more instructions it can process "at once" and the less of a delay there is between completed instructions. Every predominant general purpose microprocessor manufactured today uses at least 2 stages of pipeline up to 30 or 40 stages.

3. If pipelining is used, the CPU Arithmetic logic unit can be designed faster, but more complex.

4. Pipelining in theory increases performance over an un-pipelined core by a factor of the number of stages (assuming the clock frequency also increases by the same factor) and the code is ideal for pipeline execution.

5. Pipelined CPUs generally work at a higher clock frequency than the RAM clock frequency, (as of 2008 technologies, RAMs work at a low frequencies compared to CPUs frequencies) increasing computers overall performance.

However, these advantages come at the cost of latency.Piepelining does not reduce instruction latency (the time to complete a single instruction from start to finish) as it still must go through all steps. Indeed, it may increase latency due to additional overhead from breaking the computation into separate steps and worse, the pipeline may stall (or even need to be flushed), further increasing latency. Pipelining thus increases throughput at the cost of latency, and is frequently used in CPUs, but avoided in realtime systems, where latency is a hard constraint.

The advantages of pipelined architecture greatly outnumber the disadvantages, and hence a 5-stage instruction pipeline is employed in this processor.

7.3 Pipeline Hazards

It is relatively frequent in typical computer programs that the result from one instruction is used as an operand by the next instruction. When this occurs the pipeline operation breaks down, since the result of instruction 1 is not available at the time that instruction 2 collects its operands. Instruction 2 must therefore stall until the result is available, giving the behaviour shown below. This is a read-after-write pipeline hazard as shown in Fig.7.2.

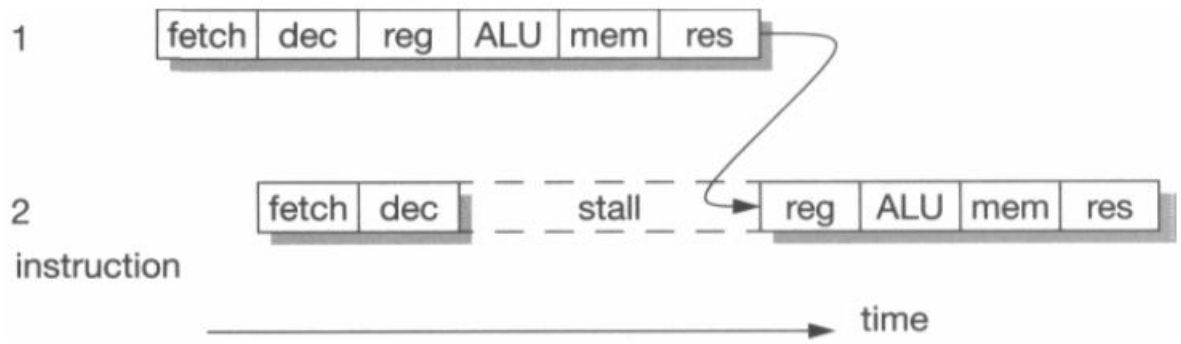

Figure 7.2 Read-after-write Pipeline Hazard

Branch instructions result in even worse pipeline behaviour since the fetch step of the following instruction is affected by the branch target computation and must therefore be deferred. Unfortunately, subsequent fetches will be taking place while the branch is being decoded and before it has been recognized as a branch, so the fetched instructions may have to be discarded. If, for example, the branch target calculation is performed in the ALU stage of the pipeline in, three instructions will have been fetched from the old stream before the branch target is available. It is better to compute the branch target earlier in the pipeline if possible, even though this will probably require dedicated hardware. If branch instructions have a fixed format, the target may be computed speculatively (that is, before it has been determined that the instruction is a branch) during the decode stage, thereby reducing the branch latency to a single cycle, though note that in this pipeline there may still be hazards on a conditional branch due to dependencies on the condition code result of the instruction preceding the branch.

Though there are techniques which reduce the impact of these pipeline problems, they cannot remove the difficulties altogether. The deeper the pipeline (that is, the more pipeline stages there are), the worse the problems get. For reasonably simple processors, there are significant benefits in introducing pipelines from three to five stages long, but beyond this the law of diminishing returns begins to apply and the added costs and complexity outweigh the benefits. Pipelines clearly benefit from all instructions going through a similar sequence of steps. Processors with very complex instructions where every instruction behaves differently from the next are hard to pipeline. In 1980 the complex instruction set microprocessor of the day was not pipelined due to the limited silicon resource, the limited design resource and the high complexity of designing a pipeline for a complex instruction set.

7.4 Solution to Pipeline Hazards

The problem with data hazards, introduced by this sequence of instructions can be solved with a simple hardware technique called forwarding.

	1	2	3	4	5	6	7
ADD R1, R2, R3	IF	ID	EX	MEM	WB		
SUB R4, R5, R1		IF	IDsub	EX	MEM	WB	
AND R6, R1, R7			IF	IDand	EX	MEM	WB

Figure 7.3 Ideal Pipeline

If the result can be moved from where the ADD produces it (EX/MEM register), to where the SUB needs it (ALU input latch), then the need for a stall can be avoided. Using this observation , forwarding works as follows:
The ALU result from the EX/MEM register is always fed back to the ALU input latches. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.
Forwarding of results to the ALU requires the additional of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.
The paths correspond to a forwarding of:

1. The ALU output at the end of EX
2. The ALU output at the end of MEM
3. The memory output at the end of MEM

Without forwarding our example will execute correctly with stalls:

	1	2	3	4	5	6	7	8	9
ADD R1, R2, R3	IF	ID	EX	MEM	WB				
SUB R4, R5, R1		IF	stall	stall	IDsub	EX	MEM	WB	
AND R6, R1, R7			stall	stall	IF	IDand	EX	MEM	WB

Figure 7.4 Stalling in Pipeline without data forwarding

As our example shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started three cycles earlier. Forwarding can be arranged from MEM/WB latch to ALU input also. Using those forwarding paths the code sequence can be executed without stalls:

	1	2	3	4	5	6	7
ADD R1, R2, R3	IF	ID	EXadd	MEMadd	WB		
SUB R4, R5, R1		IF	ID	EXsub	MEM	WB	
AND R6, R1, R7			IF	IDand	EXand	MEM	WB

Figure 7.5 Pipeline with Data Forwarding

The first forwarding is for value of R1 from EXadd to EXsub . The second forwarding is also for value of R1 from MEMadd to EXand.
This code now can be executed without stalls.

Forwarding can be generalized to include passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

CHAPTER 8

THUMB INSTRUCTION SET AND ADDRESSING MODES

I speak Spanish to God, Italian to women, French to men, and German to my horse.

—Charles V

8.1 Instruction Set

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor. A complex instruction set computer (CISC) has many specialized instructions, which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by only implementing instructions that are frequently used in programs; unusual operations are implemented as subroutines, where the extra processor execution time is offset by their rare use. Theoretically, important types are the minimal instruction set computer and the one instruction set computer, but these are not implemented in commercial processors. Another variation is the very long instruction word (VLIW) where the processor receives many instructions encoded and retrieved in one instruction word.

A complete instruction set needs to do more than perform arithmetic operations on operands in memory. A general-purpose instruction set can be expected to include instructions in the following categories:

- Data processing instructions such as add, subtract and multiply.
- Data movement instructions that copy data from one place in memory to another, or from memory to the processor's registers, and so on.

- Control flow instructions that switch execution from one part of the program to another, possibly depending on data values.
- Special instructions to control the processor's execution state, for instance to switch into a privileged mode to carry out an operating system function.

Sometimes an instruction will fit into more than one of these categories.

We have implemented 36 instructions on our processor and they can be divided into 5 categories:

1. Data Processing Instructions
2. Data Transfer Instructions
3. Stack Related Instructions
4. Branch Instructions
5. Miscellaneous

These have been explained in detail in the further sections.

8.1.1 Data Processing Instructions

Data Processing instructions consist of:

1. **Arithmetic Instructions**
   (a) **ADD Rm, Rn, Rd**: This instruction adds the contents of the registers Rm and Rd and stores the result in register Rd. This instruction uses register addressing mode.
   (b) **SUB Rm, Rn, Rd**: This instruction subtracts the contents of the register Rd from Rm and stores the result in register Rd. This instruction uses register addressing mode.
   (c) **ADC Rm, Rd**: This instruction adds the contents of Rm and Rd with carry flag and stores the result in Rd. This instruction again uses register addressing mode. The carry flag is reset after using this instruction.
   (d) **SBC Rm, Rd**: This instruction is subtract with borrow (borrow is nothing but carry flag contents). The contents of register Rd and borrow are subtracted from Rm and result is stored in Rd. This is register addressing mode.
   (e) **MUL Rm, Rd**: This is nothing but multiplication instruction which multiplies the contents of specified registers and the result is stored in Rd. The overflow flag is set in case the size of the result exceeds the size of register Rd (16 bit). This instruction uses register addressing mode.
   (f) **ADD #immd3, Rm, Rd**: This instruction adds the contents of Rm, Rd and a 3-bit immediate data specified in the instruction and stores the result in Rd. This instruction is used in immediate addressing mode.
   (g) **ADD Rd/Rn, #immd8**: This instruction adds 8-bit immediate data specified in the instruction with the contents of register Rd/Rn and the result is stored in that register itself. This instruction uses immediate addressing mode.
   (h) **ADD H1 H2 Rm, Rd**: This instruction adds the contents of registers Rm and Rd where one or both of these registers are high registers (H1 for Rm and H2 for Rn). The register could be any from R0 to R15. The result is stored in Rd. No flags affected.
   (i) **SUB Rd/Rn, #immd8**: This instruction subtracts 8-bit immediate data specified in the instruction from the contents of register Rd/Rn and the result is stored in that register itself. This instruction uses immediate addressing mode.

(j) **CMP Rn, #immd8**: This instruction compares the contents of the register Rn with 8 bit immediate data and updates the flags accordingly. This instruction is used in immediate addressing mode.

2. **Logical Instructions**

(a) **AND Rn, Rd**: This instruction bitwise ANDs the contents of the specified registers. This instruction is used in register addressing mode.

(b) **OR Rn, Rd**: This instruction bitwise ORs the contents of the specified registers. This instruction is used in register addressing mode.

(c) **XOR Rn, Rd**: This instruction bitwise XORs the contents of the specified registers. This instruction is used in register addressing mode.

(d) **LSL #immd4, Rn, Rd**: This instruction Logically shifts the contents of register Rn to the left by the value specified by 4-bit immediate data and stores the results in Rd. It provides the value of the contents of a register multiplied by a constant power of two. It inserts zeroes into the bit positions vacated by the shift, and updates the condition code flags, based on the result. Immediate addressing mode is used.

(e) **LSR #immd4 Rn, Rd**: This instruction Logically shifts the contents of register Rn to the right by the value specified by 4-bit immediate data and stores the results in Rd. It provides the unsigned value of a register, divided by a constant power of two. It inserts zeroes into the vacated bit positions. It updates the condition code flags, based on the result. Immediate addressing mode is used.

(f) **ASL #immd4 Rn, Rd**: ASR provides the signed value of the contents of a register multiplied by a variable power of 2.It updates the condition code flags, based on the result. Immediate addressing mode is used.

(g) **ASR #immd4 Rn, Rd**: ASR provides the signed value of the contents of a register divided by a variable power of 2.It updates the condition code flags, based on the result. Immediate addressing mode is used.

8.1.2 Data Transfer Instructions

Following are the Data Transfer Instructions used:

1. **MOV Rn, Rd**: This instruction copies the contents of the register Rn to Rd. This is again register addressing mode. Flags are updates as per the results.

2. **MOV Rd, #immd8**: This instruction copies the 8-bit immediate content to the destination register Rd. This is immediate addressing.

3. **MVN Rm, Rd**: This instruction copies the complemented contents of register Rm to Rd. This mode is register addressing.

4. **LOAD Rm, Rn, Rd**: This instruction loads the contents of the memory whose address is formed by the contents of registers Rm and Rd to the destination register Rd. This instruction is used in indirect addressing mode. The addressing mode is useful for pointer+large offset arithmetic and for accessing a single element of an array.

5. **STORE Rm, Rn, Rd**: This instruction stores/copies the contents of the register Rd to the memory location whose address is formed by the contents of registers Rm+Rd.This instruction is used in indirect addressing mode. The addressing mode is useful for pointer+large offset arithmetic and for accessing a single element of an array.

6. **LOAD #immd5, Rm, Rd**: This instruction copies the contents of the memory location formed by the 5-bit immediate value + the contents of the register Rm to the destination register Rd.

7. **STORE #immd5, Rm, Rd**: This instruction copies the contents of the register Rd to the memory location formed by the 5-bit immediate value + the contents of the register Rm.

8.1.3 Branch Instructions

B¡cond¿ ¡target_address¿ where: ¡cond¿ Is the condition under which the instruction is executed.
The conditions are as follows:

1. **BZ target_address** - BRANCH IF Z=1
2. **BC target_address** - BRANCH IF C=1
3. **BN target_address** - BRANCH IF N=1
4. **BNC target_address** - BRANCH IF C=0

target_address specifies the address to branch to. The branch target address is calculated by:

- Shifting the 11-bit signed offset of the instruction left one bit.
- Sign-extending the result to 16 bits.
- Adding this to the contents of the PC (which contains the address of the branch instruction plus 2).

8.1.4 Stack related Instructions

Following are the Stack-related instructions used in the processor:

1. **PUSH (r0 to r7)** where r0 to r7 is the list of registers to be stored, separated by commas and surrounded by and . The list is encoded in the register_list field of the instruction, by setting bit[i] to 1 if register Ri is included in the list and to 0 otherwise, for each of i=0 to 7. The R bit (bit[8]) is set to 1 if the LR is in the list and to 0 otherwise.
2. **POP (registers)** where r0 to r7 is the list of registers to be loaded from stack, separated by commas and surrounded by and . The list is encoded in the register_list field of the instruction, by setting bit[i] to 1 if register Ri is included in the list and to 0 otherwise, for each of i=0 to 7. The R bit (bit[8]) is set to 1 if the LR is in the list and to 0 otherwise.

8.1.5 Miscellanous

Following are the Miscellaneous operations for special use:

1. **NOP** When the processor encounters this instruction all the register values will be retained from the previous operation but all the other signals go into the high impedance.
2. **BKPT #imm8bit** whenever the processor encounters breakpoint instruction the fetch unit output is disabled i.e. the instructions will not be given as input to the decoder on breakpoint. 8-bit immediate value, which is placed in bits [7:0] of the instruction is ignored by the hardware, but can be used by a debugger to store additional information about the breakpoint.

8.2 Addressing Modes

Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs. The various addressing modes that are defined in a given instruction set architecture define how machine language instructions in that architecture identify the operand (or operands) of each instruction. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

When accessing an operand for a data processing or movement instruction, there are several standard techniques used to specify the desired location. Most processors support several of these addressing modes (though few support all of them):

1. **Immediate addressing**: The desired value is presented as a binary value in the instruction.
2. **Register addressing**: The desired value is in a register, and the instruction contains the register number.
3. **Register indirect addressing**: The instruction contains the number of a register which contains the address of the value in memory.
4. **Base plus offset addressing**: The instruction specifies a register (the base) and a binary offset to be added to the base to form the memory address.
5. **Base plus index addressing**: The instruction specifies a base register and another register (the index) which is added to the base to form the memory address.
6. **Base plus scaled index addressing**: As above, but the index is multiplied by a constant (usually the size of the data item, and usually a power of two) before being added to the base.
7. **Stack addressing**: An implicit or specified register (the stack pointer) points to an area of memory (the stack) where data items are written (pushed) or read (popped) on a last-in-first-out basis.

8.3 List of Opcodes

Table 8.1 Opcode Sheet

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSL	0	0	0	0	0	0	Shift				Rn			Rd		
LSR	0	0	0	0	0	1	Shift				Rn			Rd		
ASL	0	0	0	0	1	0	Shift				Rn			Rd		
ASR	0	0	0	0	1	1	Shift				Rn			Rd		
ADD(Reg Mode)	0	0	0	1	0	0		Rm			Rn			Rd		
SUB(Reg Mode)	0	0	0	1	0	1		Rm			Rn			Rd		
ADD(Imm Mode(3-bit))	0	0	0	1	1	0		#immd3			Rn			Rd		
SUB(Imm Mode(3-bit))	0	0	0	1	1	1		#immd3			Rn			Rd		
ADD(Imm Mode(8-bit))	0	0	1	0	0	Rd/Rm			#immd8							
SUB(Imm Mode(8-bit))	0	0	1	0	1	Rd/Rm			#immd8							
COMPARE(Imm Mode(8-bit))	0	0	1	1	0	Rd/Rm			#immd8							
MOV(Imm Mode(8-bit))	0	0	1	1	1	Rd/Rm			#immd8							
LOAD(Reg as Offset)	0	1	0	1	1	0		Rm			Rn			Rd		
STORE(Reg as Offset)	0	1	0	1	0	0		Rm			Rn			Rd		
ADD(R0-R15)	0	1	0	0	0	1	0	0	H1	H2	Rn			Rd		
MOV(R0-R15)	0	1	0	0	0	1	1	0	H1	H2	Rn			Rd		
AND	0	1	0	0	0	0	0	0	0	0	Rn			Rd		
EXOR	0	1	0	0	0	0	0	0	0	1	Rn			Rd		
OR	0	1	0	0	0	0	1	1	0	0	Rn			Rd		
MVN	0	1	0	0	0	0	1	1	1	1	Rn			Rd		
MUL	0	1	0	0	0	0	1	1	0	1	Rn			Rd		
ADC	0	1	0	0	0	0	0	1	0	1	Rn			Rd		
SBC	0	1	0	0	0	0	0	1	1	0	Rn			Rd		
LOAD(Immd as Offset)	0	1	1	0	1	#immd5					Rn			Rd		
STORE(Immd as Offset)	0	1	1	0	0	#immd5					Rn			Rd		
PUSH	1	0	1	1	1	0			Register List							
POP	1	0	1	1	1	1			Register List							
NOP	1	0	1	1	0	1										
BKPT	1	0	1	0	0	1										
MOV(Reg Mode)	1	0	1	0	1	0					Rn			Rd		
Unconditional Branch	1	1	1	0	0	#immd11										
Branch if Z=1	1	0	1	0	0	0	0	0	#immd8							
Branch if C=1	1	1	0	1	0	0	0	1	#immd8							
Branch if N=1	1	1	0	1	0	0	1	0	#immd8							
Branch if Z=0	1	1	0	1	0	0	1	1	#immd8							

CHAPTER 9

RESULTS OF PROTOTYPE TESTING

It is the result more than the path taken to achieve it that matters.

—Anonymous

9.1 Simulation Results

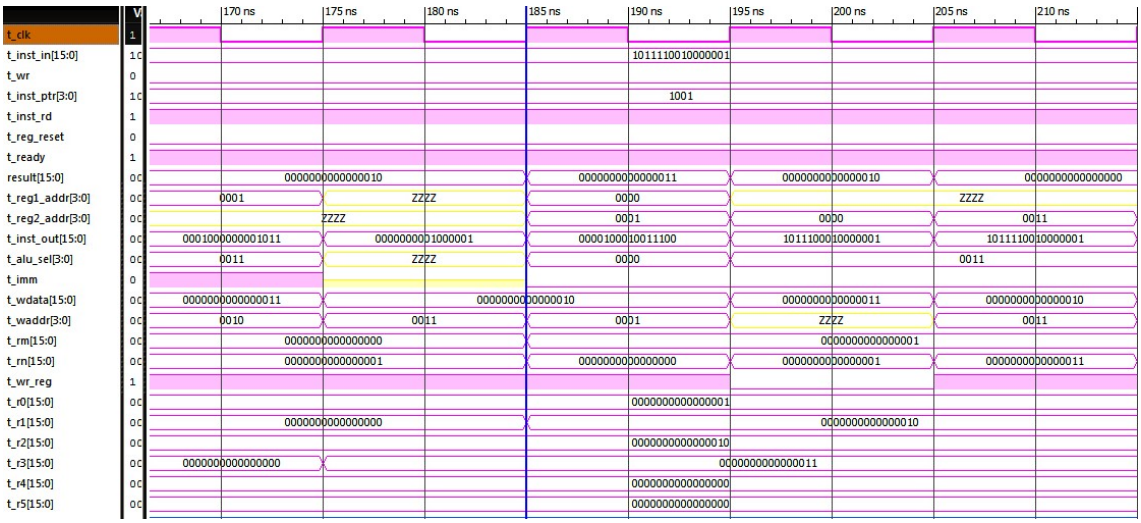

Figure 9.1 Processor Simulation Waveform

9.2 Description

For the first test of the completed processor and the memory a simple set on instructions were tested. Therefore, at first, the memory has to be loaded with the instructions. The instructions written into the memory are:

```
MOV R0, #01
MOV R2, #02
MOV R3, #03
MOV R1, #02
NOP
ADD R0,R1,R2
LSL R0,R1,#01
ASL R3,R4,#02
```

CHAPTER 10

CONCLUSION

> Computing is not about computers anymore. It's about living.
>
> —N. Negroponte

10.1 Conclusion and Experiences

The accomplishment of the project helped in gaining knowledge about VHDL language, design principles for a general purpose processor architecture along with pipelining fundamentals.
Nexys-2 Spartan 3E FPGA board was used to test the modules such as implementing the RAM controller and displaying the fetched instructions on seven segment display.
The tests helped in understanding external RAM protocols and timing constraints in accessing it.
In doing so, it was discovered that the on board RAM is Big Endian so the instructions had to be fed accordingly.
Towards the end, a pipelining hazard turned up which was dealt by implementing a simple data forwarding logic.
Furthermore, a lot of experience was gained in using the simulation and synthesis tools.
So the overall theme of the project to study Thumb mode of ARM architecture and implement it using a 16 bit processor was successfully achieved.

10.2 Future Scope

Although we achieved most of the goals we intended for there are still some areas where there is scope for future improvement. The processor functionality can be extended further by including the following units.

1. **FLOATING POINT UNIT**: ARM floating point architecture can support for floating point operations in half-, single- and double-precision floating point architecture. With the inclusion of this feature, imaging applications, body control applications, FFT and filtering in graphics can be supported.
2. **BRANCH PREDICTION UNIT**: In processors without branch predict unit, the target of the branch is not known until the end of the Execute unit. At the Execute stage it is known whether or not the branch is taken. Branch prediction attempts to guess whether a conditional jump will be taken or not. The longer the pipeline the greater the need for a good branch predictor

CHAPTER 11

APPENDIX

11.1 Flowchart And Algortihm

Algorithm:

1. Fill 16 bytes of Instruction Queue from external RAM
2. Send Ready signal to decoder when the Queue is full
3. Auto Increment Program Counter whose output is used as an index for Instruction Queue.
4. Read Instruction corresponding to the index of program counter from Queue into the decoder.
5. Segment the Instruction into different parts and store them into different variables inside decoder.
6. Analyze each part sequentially to identify the Instruction type and accordingly generate signals.
7. Pass these signals to Control Unit.
8. Control Unit passes these signals at correct stage of the pipeline to the following Units.
   (a) Sign Extension- Identifies the amount of bits to be appended,if there is an immediate data.
   (b) Barrel Shifter- Identifies whether the data from sign extension is to be shifted.
   (c) Register Bank- The number of registers to be read at most two and their addresses.
   (d) Data Memory & Stack- The write enable signal is enabled if data memeory is to be accessed.
9. ALU receives the alu select signals which tells the ALU the operation to be performed. E.g alusel=0000 for Add Instruction, alusel=0011 for MOV Instruction.
10. Alu produces result and passes it to the input of data memory.

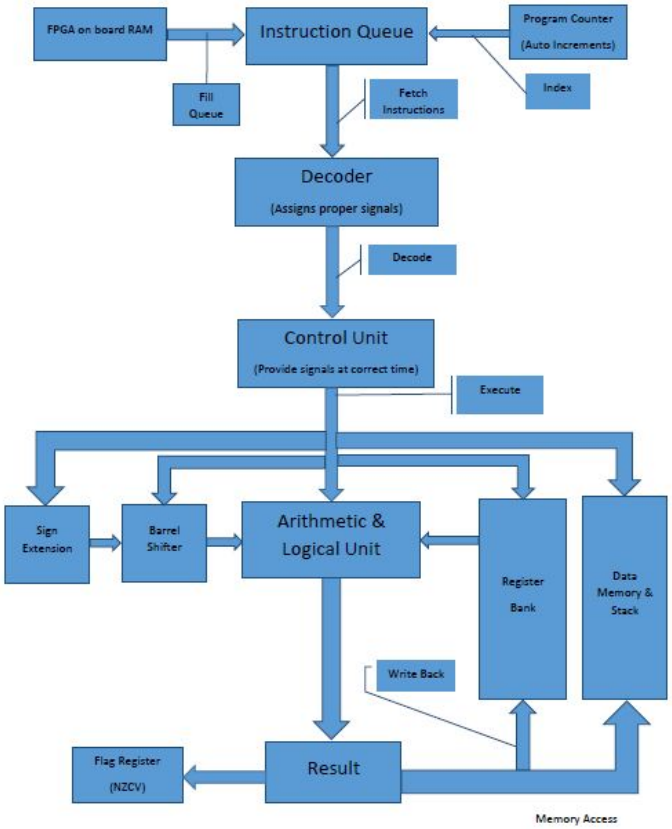

Figure 11.1 Program Flow

11. If data memory is to be accessed then alu result is written or used as an address (e.g in Store Instruction).
12. If Instruction is not related to data memory then the ALu result is passed to next stage that is Write back and the result is stored in the destination register of Register Bank.
13. Steps 3 to 12 are repeated till all instructions are fetched from the queue which forms a 5 stage pipeline.

11.2 Research Paper Approval

Approval Letter

Dear Authors,
Warm Greetings from ISRT.
Thank you for valuable contribution in the research work. Your research paper titled **Design of 16 bit ARM based Instruction Set Architecture** has been accepted and it will be published in Indian Streams Research Journal in June Issue.
Regards,
ISRT Team

11.3 Research paper

Design of 16-bit ARM-based Instruction Set Architecture

Apurva Tayade, Chinmay Gadgil, Janaki Thakkar
Mohnish Bhatia, Sagar Bachwani, Saurabh Nair
B.Tech(Electronics and Telecommunication)
Veermata Jijabai Technological Institute(VJTI), Mumbai, India

Abstract: The sales pitch goes something like this, The ARM architecture has the best MIPS to Watts ratio as well as best MIPS to dollars ratio in the industry; the smallest CPU die size; all the necessary computing capability coupled with low power consumption , all at a low cost. This success of ARM has made it the best choice for Embedded Low-Power Applications. An important factor that contributes to making this claim true is the 16 bit compressed instruction set of ARM called Thumb Instruction Set due to which programs can be coded much more densely thus cutting down the hardware size. Our paper aims to make use of this optimized subset of ARM and also strike a difference between 16 bit ARM and 16 bit MIPS in terms of their speed and performance.

Index Terms: ARM, ISA, RISC, Code density

I. Introduction: ARM is a family of instruction set architectures for computer processors based on a reduced instruction set[1] computing (RISC) architecture developed by British company ARM Holdings. The first ARM processor was developed at Acorn Computers Limited, of Cambridge, England, between October 1983 and April 1985. At that time, and until the formation of Advanced RISC Machines Limited(which later was renamed simply ARM Limited) in 1990, ARM stood for Acorn RISC Machine. Later, after a judicious modification of the acronym expansion to Advanced RISC Machine, it lent its name to the company formed to broaden its market beyond Acorn's product range. Despite the change of name, the architecture still remains close to the original Acorn design.

II. 16 bit ARM Instruction Set Description

A. Motivation Even though, RISC processors provide the advantages of a smaller die size, low cost, shorter development cycle and a higher performance, they also suffer from the drawback of having a poor code density as compared to CISC processors.[6][9] The poor code density is the result of having a fixed length instruction set. ARM ,being made on the RISC principles, also has poor code density. This disadvantage is overcome by using the 16 bit compressed subset of ARM instruction set called the Thumb Instruction Set. Thumb Instructions are 16 bit long and encode the functionality of ARM in-

struction in half the number of bits. The implementation of such an instruction set would take up less space on FPGA and will also serve as a means of reducing the die size and hardware thus leading to simple and small design and also reducing the power consumption.[5][7]

B. The 16 bit ARM programmers model

The Thumb instruction set is a condensed state of the ARM instruction set with total flexibility in accessing eight Lo general purpose registers r0 to r7. r13 to r15 are special purpose registers. r13 is used as a stack pointer, r14 is used as link register and r15 as program counter. r8 to r12 have restricted access. [3]

Table 11.1 Register Bank

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
SP(r13)
LR(r14)
PC(r15)

C. 16 bit ARM and 32 bit ARM differences

16-bit ARM	**32-bit ARM**
Instructions are executed unconditionally.	Instructions are executed conditionally.
Instructions use 2-address format.	Instructions use 3-address format.
16-bit ARM code uses 40% more instructions than 32-bit ARM code.	It uses fewer instructions than 16-bit ARM.
With 16-bit memory, 16-bit ARM code is 45% faster than 32-bit ARM code.	With 32-bit memory, 32-bit ARM code is 40% faster than 16-bit ARM code.
16-bit instruction formats are less regular than 32-bit instruction format.	32-bit instruction formats are more regular.
16-bit ARM code uses 30% less external memory power than 32-bit ARM code.	32-bit ARM code consumes more external power.

D. Difference between 16 bit MIPS and 16 bit ARM

16-bit MIPS	**16-bit ARM**
ALU is simpler as it doesnt support operations with bulky logic such as multiplication and division.	ALU is very complicated as it supports operations like multiplication and division.
It does not consist of the Barrel Shifter unit.	It consists of Barrel Shifter which provides shifting operations.
Pipelining is not essential and its implementation depends on applications.	Pipelining is the main characteristic of 16-bit ARM processor.
MIPS is not power efficient.	ARM is power efficient.

E. List of Instructions

The 16 bit ARM which we have designed supports 33 instructions which can be divided into 5 categories:

1. Data Processing Instruction: It includes all the arithmetic ,logical and shifting instructions

2. Data Transfer Instructions: It includes move, load and store instructions

3. Branch Instructions: It includes conditional and unconditional branch instructions

4. Stack Related Instructions: It includes push and pop instructions

5. Miscellanous: It includes instructions like breakpoint, halt and no operation instructions

6. A complete list of 33 instructions is provided in the table with a short description of each.

Sr.No	Mnemonic	Instruction Format	Description
1	ADD	ADD Rm,Rn,Rd	Adds the contents of Rm and Rn and stores the result in Rd
2	SUB	SUB Rm,Rn,Rd	Subtracts the contents Rn from Rm and stores the result in Rd
3	ADC	ADC Rn,Rd	Add Rm and Rn with carry flag and store the result in Rd
4	SBC	SBC Rn,Rd	Subtracts Rn and borrow from Rm and result is stored in Rd
5	MOV	MOV Rn,Rd	Copies the contents of the register Rn to Rd
6	MUL	MUL Rn,Rd	Multiplies the contents of Rn and Rd and the result is stored in Rd
7	AND	AND Rn,Rd	Bitwise ANDs the contents of Rn and Rd and stores the result in Rd
8	OR	OR Rn,Rd	Bitwise ORs the contents of Rn and Rd and stores the result in Rd
9	XOR	XOR Rn,Rd	Bitwise XORs the contents of Rn and Rd and stores the result in Rd
10	MVN	MVN Rn,Rd	Copies the complemented contents of register Rn to Rd
11	LDR	LDR [Rm,Rn],Rd	Loads Rd with contents of memory address formed by Rm+Rn
12	STR	STR [Rm,Rn],Rd	Store contents of Rd to the memory address formed by Rm+Rn
13	ADI	ADI #immd_8,Rd	Adds Rd and a 8-bit immediate data and stores the result in Rd
14	SBI	SBI #immd_8,Rd	Subtracts 8-bit immediate data from Rd and result is stored in Rd
15	MVI	MVI #immd_8,Rd	Copies the 8-bit immediate content to the destination register Rd
16	CMP	CMP #immd_8,Rd	Compares the contents of Rd with 8 bit immediate data
17	ADDH	ADDH Rn,Rd	Adds Rn and Rd where one or both of these are high registers
18	MOVH	MOVH Rn,Rd	Copies Rn to Rd where one or both of these are high registers
19	LSL	LSL #immd_4,Rn,Rd	Logically shifts Rn to left by 4-bit data and stores result in Rd
20	LSR	LSR #immd_4,Rn,Rd	Logically shifts Rn to right by 4-bit data and stores result in Rd
21	ASL	ASL #immd_4,Rn,Rd	Provides the signed value of Rn multiplied by variable power of 2
22	ASR	ASR #immd_4,Rn,Rd	Provides the signed value of Rn divided by a variable power of 2
23	PUSH	PUSH registers	Used to store contents of the mentioned registers on the stack

24	POP	POP¡registers¿	Used to load contents of the mentioned registers from the stack
25	BL	BL ¡target addr¿	Provides an unconditional subroutine call to another routine
26	BUC	BUC,target adress	Branch unconditionally to the target address
27	BC	BC,target adress	Branch to the target address when carry is set
28	BNC	BNC,target adress	Branch to the target address when carry is not set
29	BZ	BZ,target adress	Branch to the target address when zero flag is set
30	BNZ	BNZ,target adress	Branch to the target address when result of operation is non zero
31	BKPT	BKPT	Causes a software breakpoint to occur
32	NOP	NOP No operation	An idle machine cycle is executed
33	HLT	HLT	Halts the processor

III. Pipeline Implementation

In a pipeline, instruction cycle is broken up into fixed steps. Rather than processing each instruction sequentially just like a single cycle execution design, each instruction is split up into a sequence of steps which are executed with steps of several instructions being executed parallel. This makes a productive design allowing the processor deal with several instructions at the same time increasing efficiency. With many more than one instruction being executed in parallel, we can achieve more through-put proportional to number of stages in the pipeline. The latency remains same compared to non-pipelined systems but maybe even more due to stalls and pipeline hazards. But with separate systems working together, we have the liberty to increase the frequency of the system making it faster.

In our model, we used the classic RISC pipeline with 5 stages.[2] The 5 stages were fetch, decode, execute, memory operation and write back.

1. **Fetch**: The fetch block is the program counter and incrementer itself. This block keeps incrementer the program counter register at every rising edge of the clock. Overwriting the register for a new value supplies the method for branching instructions. Fetching keeps the instruction queue filled.

2. **Decode**: This block of code takes in the instruction from the instruction queue and assigns correct signals at the next clock cycle. The decoder doesnt deal with timing of all the signals. This cycle decides the next state of all the blocks in the system.

3. **Execute**: The signals from the decoder pass select signals for the execute stage for e.g. add select lines for ALU.

4. **Memory read/write**: This stage is especially designed for load store instructions for the memory data transactions. The ALU result is passed to this stage as address or value to be written back.

5. **Write back**: This stage is the final stage of the pipeline where the final result is stored back into the register file. Implementation of states: The decoder simply analyses the whole opcode and decides on all the pertaining signals to all units. But just assigning signals directly will lead to a single cycle execution only. Hence to design a pipeline out of these signals, we designed a control unit with different delays to synchronize the timing for all the 5 stages. The control unit is a 5 process design which is 5 states which work simultaneously making the pipeline possible.

IV. Conclusion: With the reviews of various type of controllers and processors in the

market, we thus have come to a conclusion that it is time to have a 16 bit efficient controller with higher processing capabilities like arm v9 processors. Hence to keep the costs low and size small, we have designed and simulated the 16 bit arm design with 33 instructions successfully on VHDL and iSim simulator respectively. The design isnt using all features of Thumb mode and arm but has the powerful instruction set architecture that ARM possesses.

V. References

1. Patterson & Hennessy, Computer Organization and Design (ARM Edition)
2. Steve Furber, ARM system On-chip architecture
3. ARM v7-M Architecture, Application Level Reference Manual
4. vhdlguru.blogspot.com
5. Pong Chu, FPGA prototyping by VHDL examples
6. ARM architecture reference manual
7. John F Wakerly, Digital design
8. MIPS Technologies, MIPS32 Architecture for Programmers Volume IV-a

CHAPTER 12

BIBLIOGRAPHY

1. Patterson & Hennessy,*Computer Organization And Design (ARM Edition,4th edition)*, Morgan Kaufmann Publishers[2006]
2. Steve Furber, *ARM system On-chip architecture(2nd edition)*, Pearson Education[2009]
3. Richard E, Haskell & Darrin M. Hanna, *Learning by Example using VHDL Advanced Digital Design with a Nexys 2 FPGA Board(2nd edition)*, LBE Books[2008]
4. Pong Chu, *FPGA prototyping by VHDL examples(1st edition)*, Wiley Interscience[2007]
5. John F Wakerly, *Digital design principles and practices, 3rd Editon*, Pearson Education[2010]
6. Douglas Perry, *VHDL Programming by example(4th edition)*, McGraw Hill[2002]
7. *vhdlguru.blogspot.com*
8. *www.stackoverflow.com*
9. *www.stackexchange.com*