CALIFORNIA STATE UNIVERSITY NORTHRIDGE

Design of a 5-stage pipelined MIPS Processor

A graduate project submitted in fulfillment of the requirements

For the degree of Masters of Science

in Electrical Engineering

By

Abhirup Jannu

DECEMBER 2013

The graduate project of Abhirup Jannu is approved:

Mirzaei, Shahnam, Ph.D.	Date
Amini, Ali, Ph.D.	Date
Roosta, Ramin, Ph.D., Chair	Date

CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Acknowledgements

First and foremost, I would like to thank Dr. Ramin Roosta, without whom, this project would not have been possible. He has been the guiding force for me, by providing me excellent inputs and support whenever I was in a fix. The presentations that he asked me to give towards the end, were worthy of an invaluable experience. I am grateful to have been able to work under him.

Also, I sincerely want to extend my regards to my other committee members, Dr. Ali Amini and Dr. Shahnam Mirzaei for their constant support and for taking time to review my project report and giving required suggestions. Not only did they advise about my work through emails and talks with them have evoked a good interest in work.

Preface

This report mainly focuses upon the implementation of 5-stage pipelined MIPS CPU architecture. It explains all the basics required, right from RISC and CISC, Single and Multi-cycle processor and then goes on to explain the complete MIPS architecture. The simulation results and deductions based on the synthesis also shall be discussed.

I have tried my best to keep the report simple yet technically correct. I hope I succeeded in my attempt.

Abhirup Jannu

Table of Contents

List of Figures

ABSTRACT

Design of 5-stage pipelined MIPS processor

By

Abhirup Jannu

Master of Science in Electrical Engineering

The aim of this project is to design a 5-stage pipelined MIPS processor, using Verilog HDL. The 5 stages being used are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB). The instruction set being used is of 32-bits. The various modules being used are Instruction Memory, Data Memory, ALU, Registers etc. I also have implemented a Forwarding Unit and a hazard detection unit for the detection of Data hazards.

The main goal is to do the complete ASIC flow (RTL to GDS II), using Synopsys design tools. VCS is used for simulation, Synopsys DC Compiler for Synthesis (timing and area are optimized in this step) and Synopsys IC Compiler for Clock tree Synthesis and Place and Route.

This report focuses upon, basics of RISC and CISC, MIPS Processor, Hazard detection in MIPS processor and also the simulation and synthesis based results and deductions.

Chapter1: Introduction

The aim of my grad project is to design a 5-stage pipelined MIPS processor, using Verilog HDL and to synthesize it using Synopsys DC compiler. The 5 stages being used are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB). The instruction set being used is of 32-bits. The focus is to understand the processor design, pipelining and instruction set of MIPS.

MIPS is basically a Microprocessor without Interlocked Pipelined Stages. By designing MIPS, we are basically designing a RISC based CPU (Central Processing Unit).

Let me start with the basics of CPU first. CPU mainly is of 2 types RISC and CISC.

1.1 Basics of RISC and CISC

CISC (Complex Instruction Set Computer), is based on the idea of integrating Computer Architecture with Computer Science. It was initiated by IBM360 system, which adopted a control called Microcoded control, which made the use of complex instructions. CISC is basically designed to reduce semantic gap between machine language and high-level language of processor. CISC CPU's are such that they make use of instructions, which are executed with the help of Microcode. Just by changing the microcode, it enables us to change the hardware, yet maintain the compatibility of instructions with earlier computers.

Compiler's function was to convert the high-level language to assembly language. Assembler was used to convert assembly language to machine code. CISC had its own problems. It became complicated as well as time consuming, when more complex instructions are used, which also made them more expensive. It was feasible to simplify the instruction set, as the high level language posed some problems to compilers. The better compiler technology and cheaper memory devices led to the evolvement of RISC based CPU's.

RISC (Reduced Instruction Set Computer) CPU, made use of simpler instructions, larger no. of registers, simpler pipelined processor, a small transistor count. Because of this, it becomes cheaper and easy to design at high clk rate. Single-cycle based instructions became easier because of RISC.

1.1.1 CISC

CISC are basically chips which are easily programmable and also make efficient and better memory usage. It was developed to simplify the compiler development. For example, a CISC processor has a built in capability to implement complex instructions, instead of making compiler write lengthy machine instructions. Pentium is an example of CISC processor.

CISC instructions have a few common characteristics based on its design constraints:

- They follow a two operand format basically that of source and destination. Source and destinations can either between register and register (or) register and memory (or) memory and register.

- The length of the instructions can vary. It basically varies with respect to addressing mode.
- Instructions which need multi-clocks also can be executed.

CISC architectures also have many common characteristics:

- Single instruction can support many addressing modes. Hence complex instruction decoding logic can be used.
- Instruction can make use of memory directly. Limited chip space is not dedicated for ID, IE & microcode storage. Hence number of general-purpose registers is used.
- Special purpose registers are used for interrupt handling, stack pointer etc. Here hardware is simplified, by making the instruction set complex.
- Error conditions can be checked by using a 'Condition code' register, which shows if the result of the operation is greater than or less than or equal to and tells us if there is any error.

To optimize performance, CISC machines make use of some of the available technologies:

- Microprogramming – easy and less expensive.
- Microcoding – Computer has a superset of instructions which are in earlier computers. Hence, can also run same programs as that of earlier computers.

- The given task can be implemented by using less number of instructions. Slow memory is used to make it more efficient.
- Compiler is less complicated as microprogramming instructions are written to match high level constructs.

CISC has its own disadvantages:

- Each newer generation of instructions for a computer have a subset of previous version instructions. Hence it becomes more complex with every latest generation PC's.
- Instructions can be of any length. Hence each instruction will take different number of clock cycles to execute. Hence the overall performance is slowed down.
- As there can be any number of instructions, just about 20% of instructions are usually used.
- Condition code registers are used. Users must check those bits before the next instruction completes execution as it changes them. This further makes the process slower.

1.1.2 RISC

RISC is a microprocessor type architecture which uses small and highly optimized set of instructions, than a special instruction set found in other instructions.

RISC processors have few design features:

- Each instruction on a CPU is optimized. Therefore each instruction uses a single clock cycle to execute. CPI = 1 CLK cycle.
- RISC makes use of PIPELINING (this is what we basically focus upon in this project). Pipelining allows simultaneous execution of instructions, making the process more efficient.
- To avoid huge number of interactions with the memory, RISC makes use of large no. of registers.

Let us talk about pipelining. RISC processor may have any number of steps. But, usually a RISC processor comprises of 5 basic instructions:

- Fetch – Instructions are fetched from the memory
- Decode – Decodes the instructions
- Execute – Executes the instructions
- Memory – Accesses data memory
- Write Back – Writes the instructions back into registers.

These instructions are further discussed later.

One of the disadvantages of RISC is that, by simplifying the hardware, there is a greater burden because of RISC architecture on software. But one wonders if it is worth it, because microprocessors are anyhow becoming faster and cheaper.

1.1.3 RISC vs CISC

CISC	RISC
More emphasis on HW	More emphasis on SW
Has multi clk complex instructions	Has Single clk simpler instructions
Memory to Memory : LOAD and STORE are incorporated in the instructions	Register to Register : LOAD and STORE are separate instructions
Large cycles/sec, Small codes	Small cycles/sec, Large codes
Instructions are stored in transistors	Transistors arespent on memory registers

Chapter2: Single and Multi Cycle CPU

A CPU can be implemented in 2 ways

- Single cycle
- Multi cycle

2.1 Single-Cycle Implementation

A single cycle Implementation means that all the operations take equal amount of time. There are many instructions in a CPU. Each instruction might take different amount of time. But in a Single Cycle, all the operations take equal amount of time which is completed in on clock cycle. So the question arises about how the clock cycle is determined? There might be 'n' number of instructions. But the clock cycle is determined by the time taken by the slowest instruction. Usually the slowest instruction is load word. Other instructions might be executed before the clock tick. In that case the instruction will just have to wait for the next clock, to start executing.

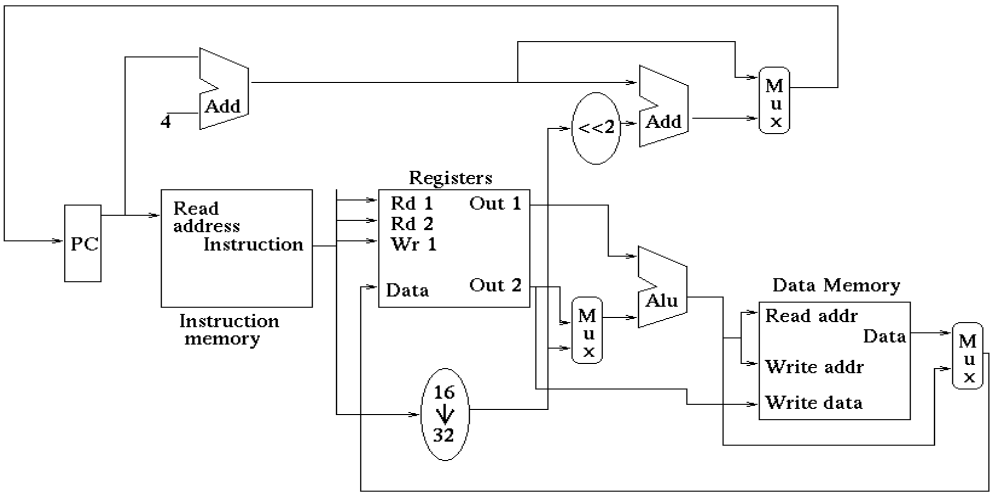

Figure 1: Single-Cycle CPU [8]

The above figure depicts the data path for single-cycle implementation of CPU. It uses I-Type branch instruction. The read address for the Instruction Memory is generated from program counter. The output of Instruction Memory is divided and a part of it is sent to registers module, to determine the data which needed to be accessed and the other part is sent to sign extend which extends the 16-bit data to 32-bit data. The output of the registers module, which is the data selected based on output from Instruction Memory, is sent to ALU, where all the arithmetic operations are performed based on the op-code. The result from the ALU is sent to Data Memory, which writes the result to the memory.

Coming to PC, it depends on the comparison of the operands. If operands are eq, then PC is incremented by 4 and also the value of offset*4 is added to it. Otherwise just the PC is incremented by 4 and the offset is not taken

The detailed operation and all the blocks Instruction Memory, ALU etc will be briefly in the later sections. There is a further discussion for the single cycle, where we have to add a control module which will generates the op-codes etc for our implementation. That will also be eventually be discussed.

2.2 Types of Instructions

So, let me start off with the types of instructions that are used for a CPU. There are 4 types of instructions:

- R-Type
- I-Type
- J-Type
- Jump

My instructions are of 32-bits. These instructions are selected in Instruction Memory. The 1st 6-bits are always op-code. From the rest of the bits, part of the instruction is always the address. The address bits are determined by the type of instruction. Instruction control bits are usually determined by this. In my implementation each is of 6-bits. Let me take each of them into consideration.

R-Type:

They are nothing but Register type instructions.

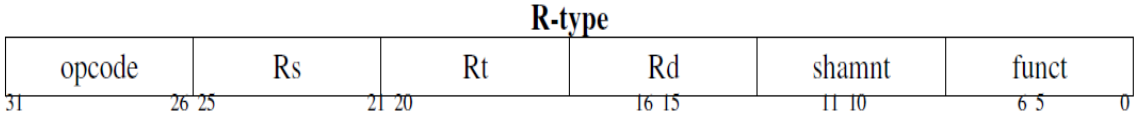

Figure 2: R-type instructions

The above address bar depicts R-Type instructions. We can clearly see that:

- The last 6bits that is I[31:26] is nothing but op-code.
- I[25:21], I[20:16] and I[15:11] are 2 registers Rs, Rt and Rd respectively, on which operations are performed. Rs, Rt are source registers and Rd is the destination registers respectively.
- I[10:6] is nothing but shift amount. It is of 5-bits. It points to the number of bits to be shifted.
- I[5:0] is a 6-bit function field. Points to the function that needs to be performed on the registers.

The data path for R-Type instruction can be depicted as follows. Used mainly for Add, Sub, And, Or

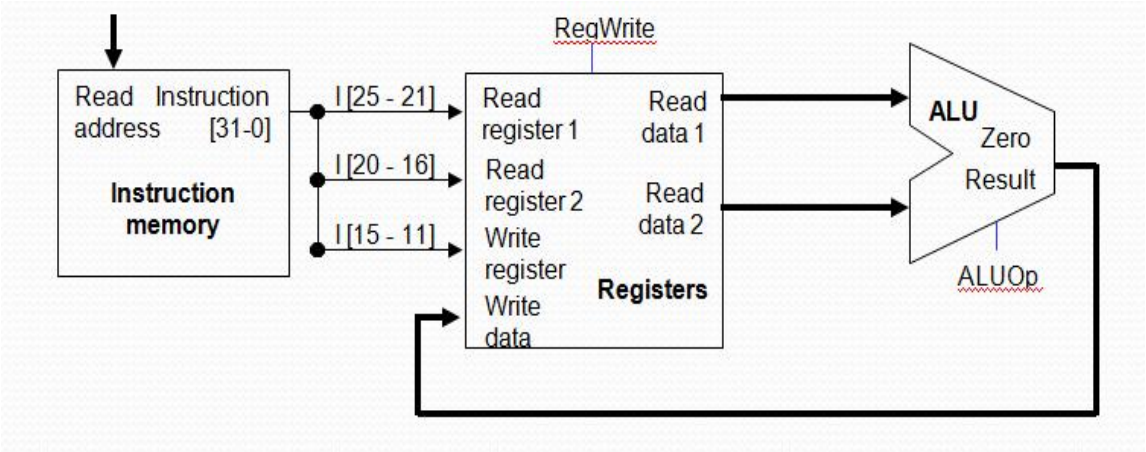

Figure 3: Data Path for R-Type instructions [5]

The above figure depicts all the three registers Rs, Rt and Rd as I[25:21], I[20-16] and I[15-11] respectively. Operation is done in ALU.

For example:

add $Rd, $Rs, $Rt

Here signed addition contents of ($Rs) $_+$ ($Rt) is saved into address $Rd

I-Type:

These are Immediate Type instructions. Used mainly for load and store.

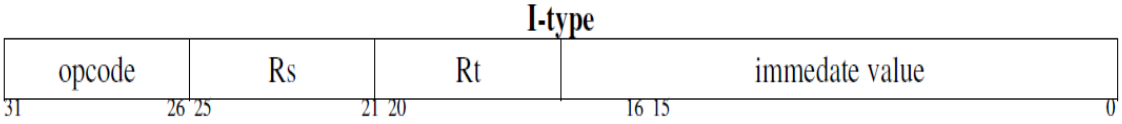

Figure 4: I-type Instructions

From the above address bar, we can clearly see that:

- I[31:26] is op-code
- I[25:21] and I[20:16] are 2 5-bit register addresses Rs and Rt. Note that there are only 2 registers as compares to 3 for R-Type. Here Rs is source register and also note that Rt is source register for store and destination register load.
- I[15:0] is nothing but immediate value. It is of 16-bits. It is a part of instruction but not part of memory. Easier to access.

The data path for I-Type instruction can be depicted as follows:

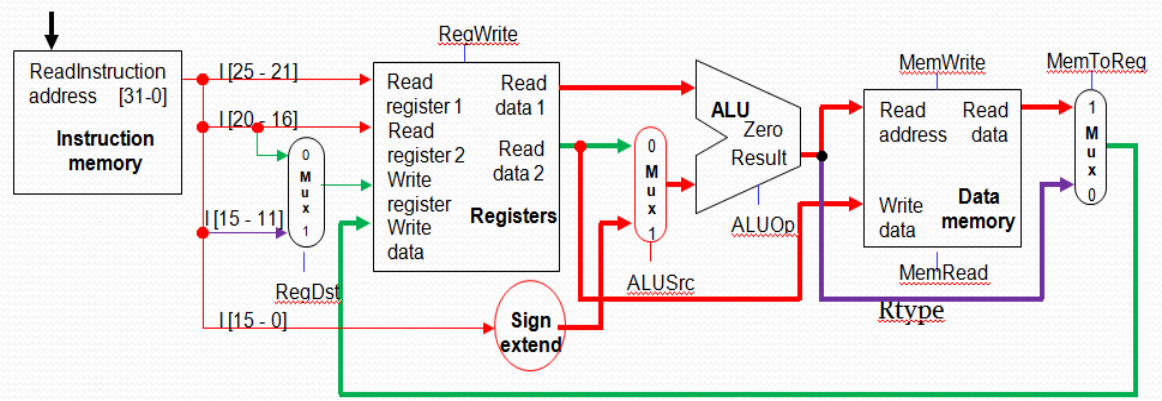

Figure 5: Data Path fo I-Type instructions [5]

In the above figure, we can clearly observe that Rt I[20:16] can be used both as source and destination accordingly. He I[15:11] is also shown, if we want to use it as R-Type. I[15:0] is the immediate value sent to sign extend and then to ALU, for performing the required operation. For example:

addi $Rt, $Rs, 5

Here ($Rs) + 5 is stored in the destination address $Rt. 5 is immediate value.

Branch instruction:

The description and the figure for branch instruction is as described for Single-Cycle CPU in the section 2.1.

J-Type:

These are Jump instructions.

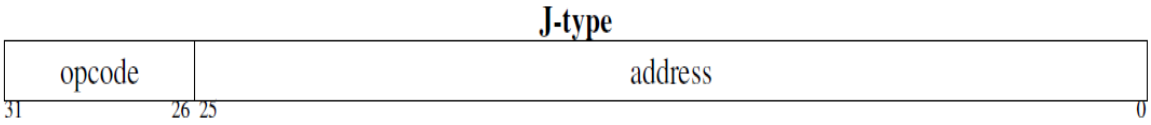

Figure 6: J-type Instructions

The above address bar clearly shows that there are only opcode I[31:26] and address I[25:0].

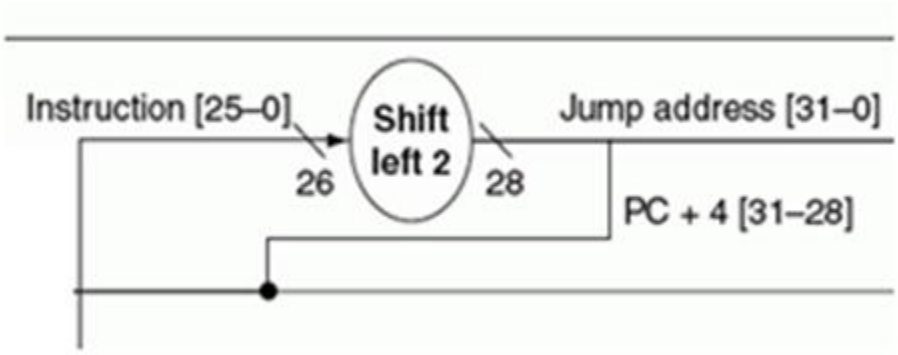

Figure 7: Data path for J-type instructions [5]

Above figure shows it's functioning. We can see that the last four bits of PC+4[31:28] are appended to the shift left by 2 value of the 26-bit I[25:0] instruction, taken from Instruction Memory, to get the 32-bit jump address.

For example:

j trgt

j is jump and trgt is target. It skips the other instructions and jumps to the target.

We'll eventually talk about ALU and control, which is also needed to be added to both single-cycle and multi-cycle data path.

2.3 Multi-Cycle CPU

The CPI for Multi-Cycle CPU is more than 1 as the instructions can take more than 1 clock cycle for execution. For example load instruction takes 5 clock cycles to execute, where as branch instruction might take only 3 clock cycles. Using this process, complexity of instructions can be increased. A control unit must be used in this case as well, but the difference between single and multi is that, multi uses FSM's for the control signals, where as single uses multiplexers.

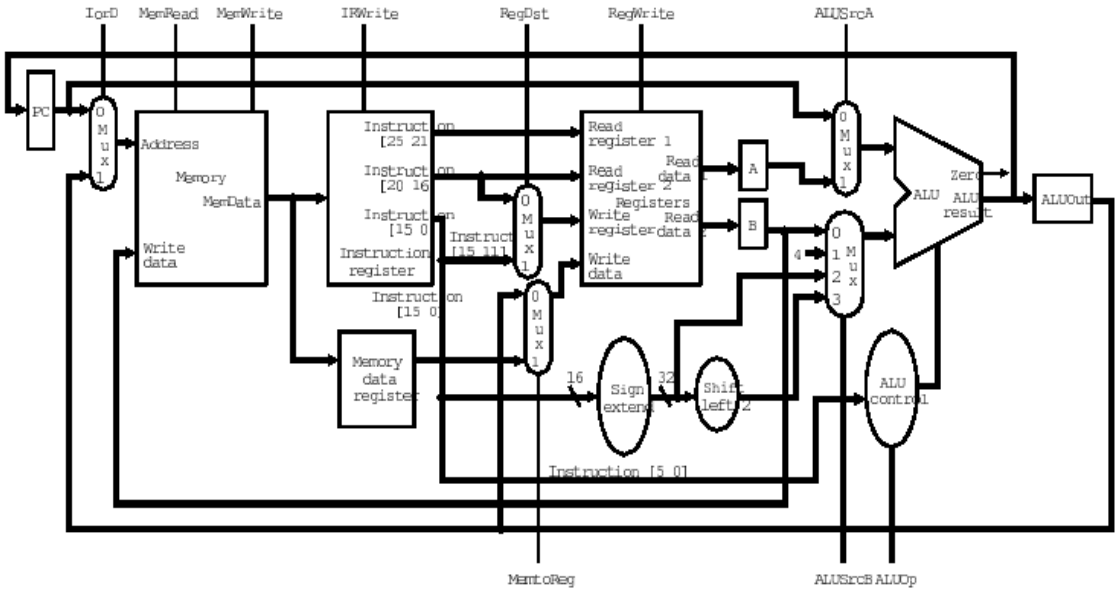

Figure 8: Multi-cycle data path [8]

The above figure depicts the multi cycle data path. After looking at the datapath of multi-cycle CPU, we can notice few differences with the single-cycle, the most prominent one is the addition of extra registers A, B etc. There is only one ALU, which is also used for incrementing PC by 4 followed by ALUout register, as compared to single, which has an adder as well. PC+4 is executed in the first clock cycle. Then the ALU is used again for another operation. But we may need PC+4 at a later stage, hence to keep ALU from loosing the information, we make use of these registers. The various stages followed here are:

- Instruction Fetch: PC is incremented by 4 and loaded to PC. And Instruction Register is loaded with instruction at PC.
- Instruction Decode: The values from register module are loaded into A and B and also ALUout is loaded with the target address.
- Execute: ALU operation takes place and it is loaded into ALUout. Again there are 2 types of instructions in this. The regular ALU operations and Branch equal to (Beq) operations. During arithmetic operations, the operation is done and the result is loaded to ALUout. In case of Beq, the data at A and B are subtracted. If the result is 0, the the value which is at ALUout is loaded into PC. In this case, process is done and we return again to Instruction Fetch.
- Memory: Here again there can be 3 steps load, store and arithmetic. If load is going on, the data at the address of ALUout, is loaded into Memory Data Register. If store operation is going on, the dta at register B is loaded into memory at ALUout address. If arithmetic operation is going on, the value in ALUout is

written into register module.In the store and arithmetic cases, we return ti 1st step Instruction Fetch.

- WriteBack: Here load instruction takes place. The value in memory data register is written into register. The process is completed here and we return to 1st step Instruction Fetch again

Chapter3: MIPS

MIPS ("Microprocessor without Interlocked Pipelined Stages") is basically a RISC based architecture. I implemented a 32-bit MIPS processor. So all the bit sizes, widths etc referred in this report are taken with respect to this.

3.1 Pipelining

Before knowing what MIPS is, we need to focus upon the concept of Pipelining. Pipelining is nothing but doing more than one operation, in a single data path. A multi-cycle CPU consists of many processes. For example load might take up to 5 clock cycles, but beq takes only 3 clock cycles. So if one process is taking place, instead of waiting for the process to complete, we can simultaneously start a new process in the same data path, without disturbing the previous process.

For this to happen, the each part of the process is divided into various pipelined stages. So after every clock, the process is stored into next pipelined stage, enabling another operation to start in that stage without disturbing the previous process. Hence all the stages in the path can be used simultaneously. This in turn can increase the throughput of your design.

Our particular MIPS processor is divided into 5 stages as shown below:

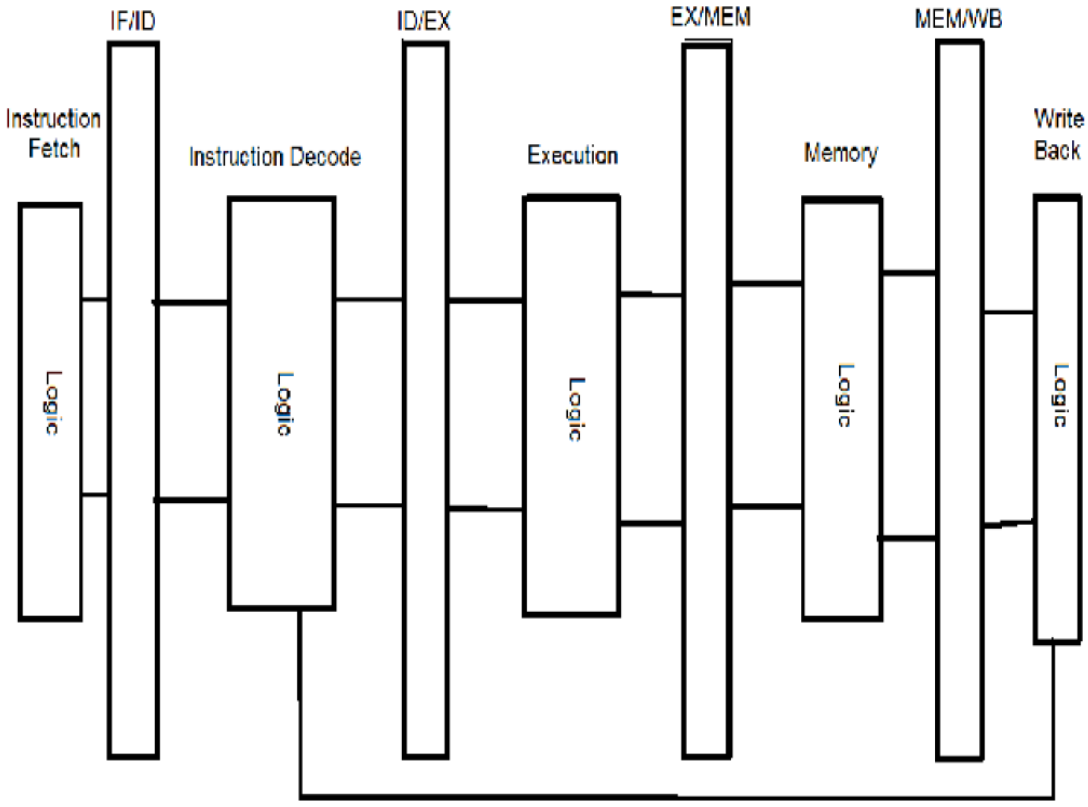

Figure 9: Stages of MIPS Processor

From the above figure, we can say that there are 5-stages in our MIPS. Same as the ones we have seen before:

- Instruction Fetch
- Instruction Decode
- Execute
- Memory
- Write Back

Each of these stages is separated by pipelined stages in between, namely IF/ID, ID/EX, EX/MEM and MEM/WB.

The following is the basic data path of a pipelined MIPS

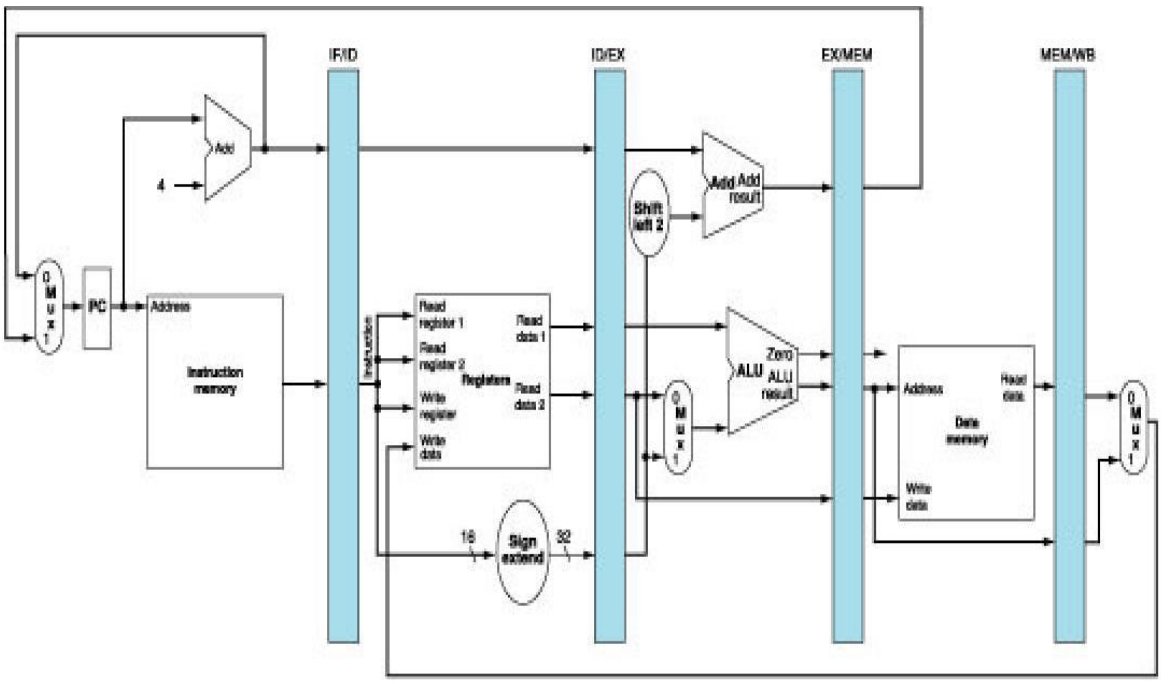

Figure 10: Pipelined MIPS Data Path [1]

Let us start with the 5-stages of our MIPS:

Instruction Fetch:

PC is used to fetch the instruction from the Instruction Memory and is stored in the Instruction Register (IF/ID) at the next +ve clock. This stage has various modules like Instruction Memory, which holds the instructions needed. PC holds the address of the current instruction, which is used as address to the Instruction Memory. The instructions

read out from the Instruction memory are stored in the Instruction Register, which is a part of IF/ID stage. The following is the data path for Instruction Fetch.

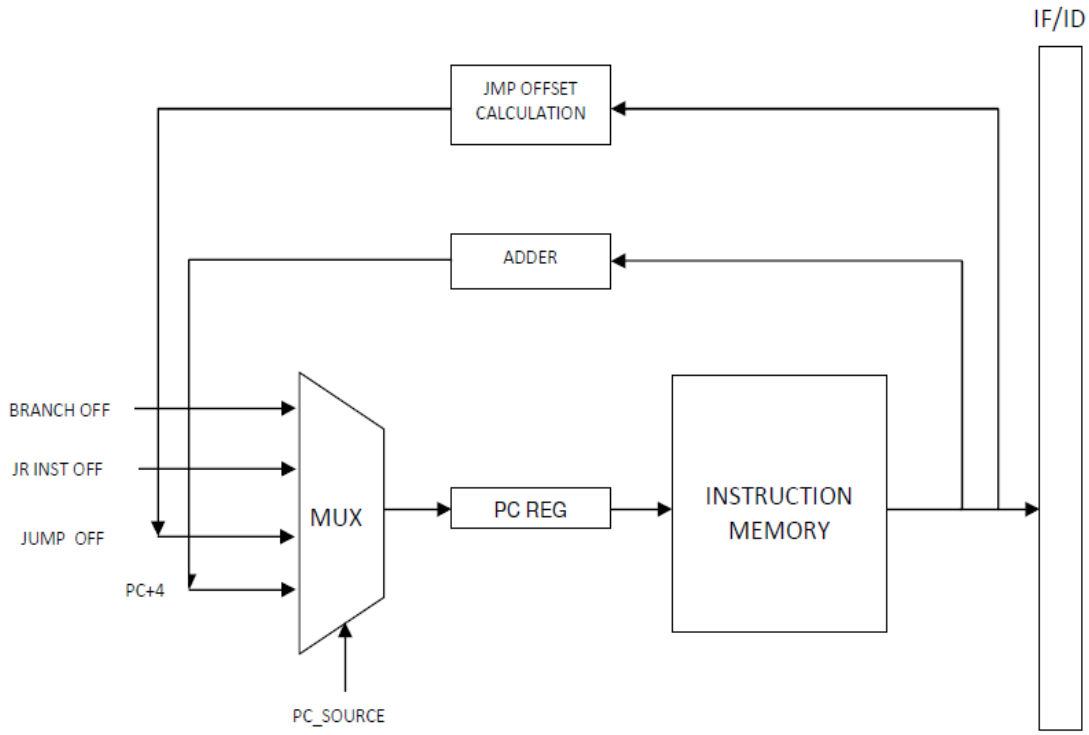

Figure 11: Instruction Fetch Data path

Instruction Decode:

Decodes the instructions sent from Instruction register. Based on the instructions, it reads the operands required for register file. Out of 32- bits, 16 go to sign extend, where those 16 bits are extended to 32-bits. The register module gives out the value of 2 registers register A and register B, which are sent to ALU through ID/EX stage. The following is the data path representation od Instruction Decode

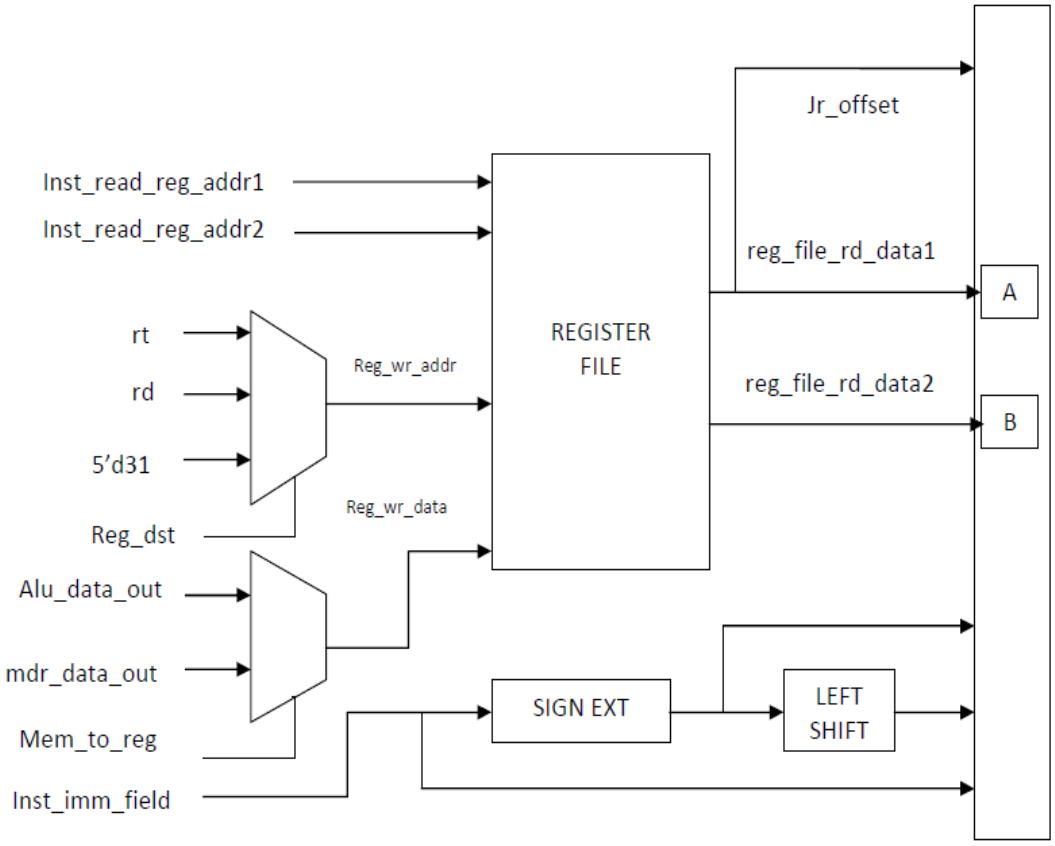

Figure 12: Instruction decode Data Path

Execute:

All the instructions are executed in this stage. All ALU operations like functional and logical operations, take place in this stage. It performs operations on the data sent from ID/EX stage. This stage also has left shift by2 and an adder, for beq operation. The result from ALU is sent to ALUout register which is in EX/MEM stage.

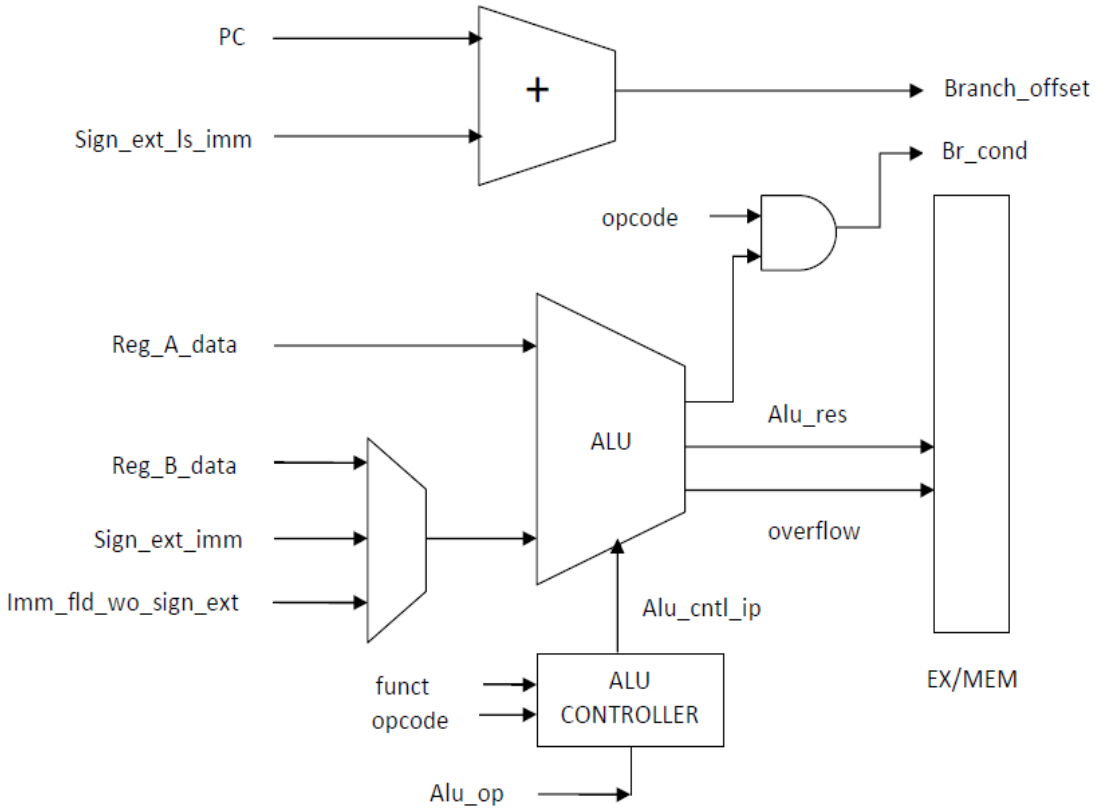

Figure 13: Execute Stage Data Path

Memory:

Memory basically has a Data Memory. Data memory has 1 read port and 1 write port. It is 64- words deep. The data read is stored in Memory data out register, which is of 32-bits. This register is in MEM/WB stage.

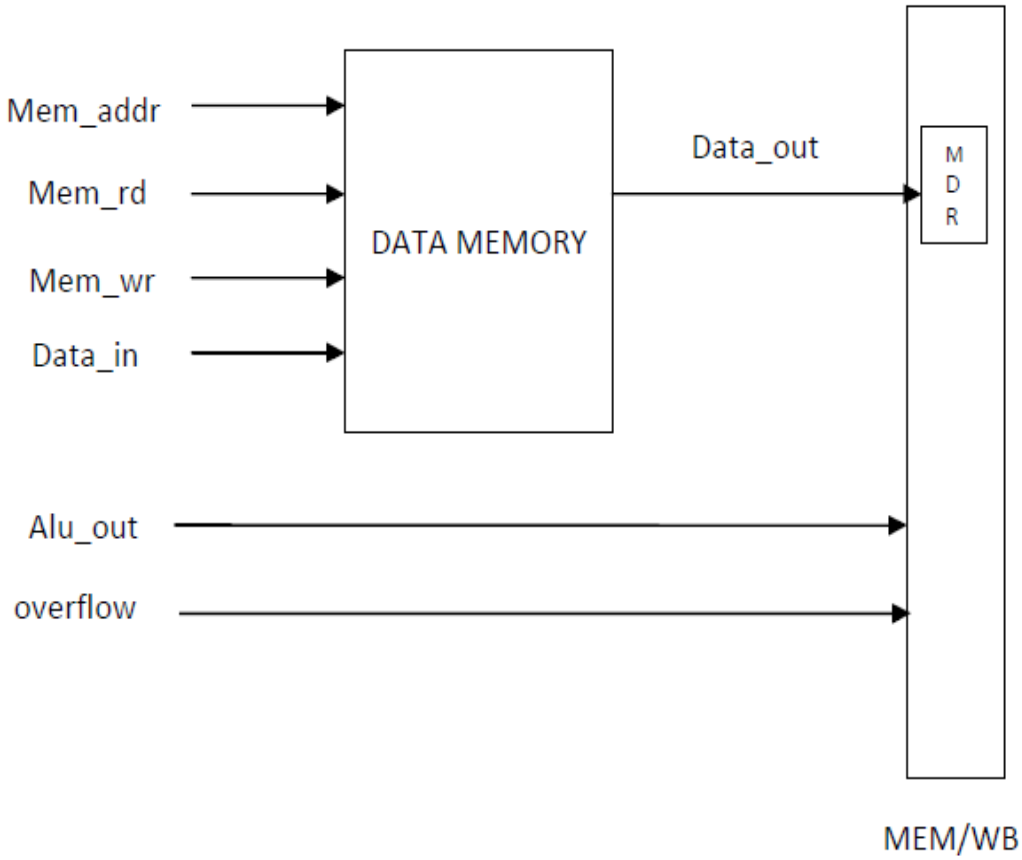

Figure 14: Memory Stage Data path

Writeback:

It basically writes back the result into register file.

The other prominent modules which need some discussion are:

ALU: It performs all the operations in the execute stage. The operations are selected with the help of opcode which is generated from ALUControl module. ALU control which gets the instruction opcodes, ALU_op and function field.

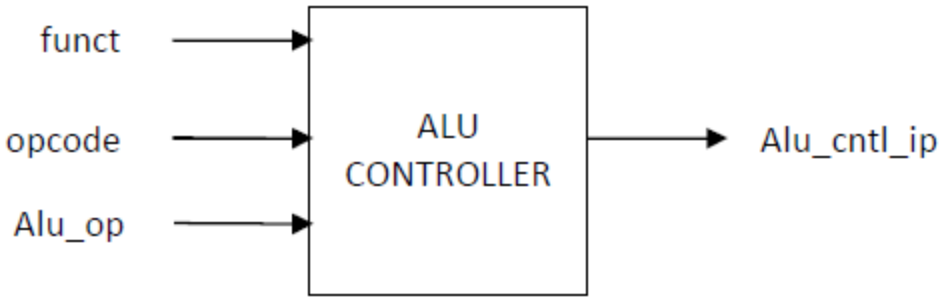

Figure 15: ALU Control

The following is the figure when you add ALUControl to our design.

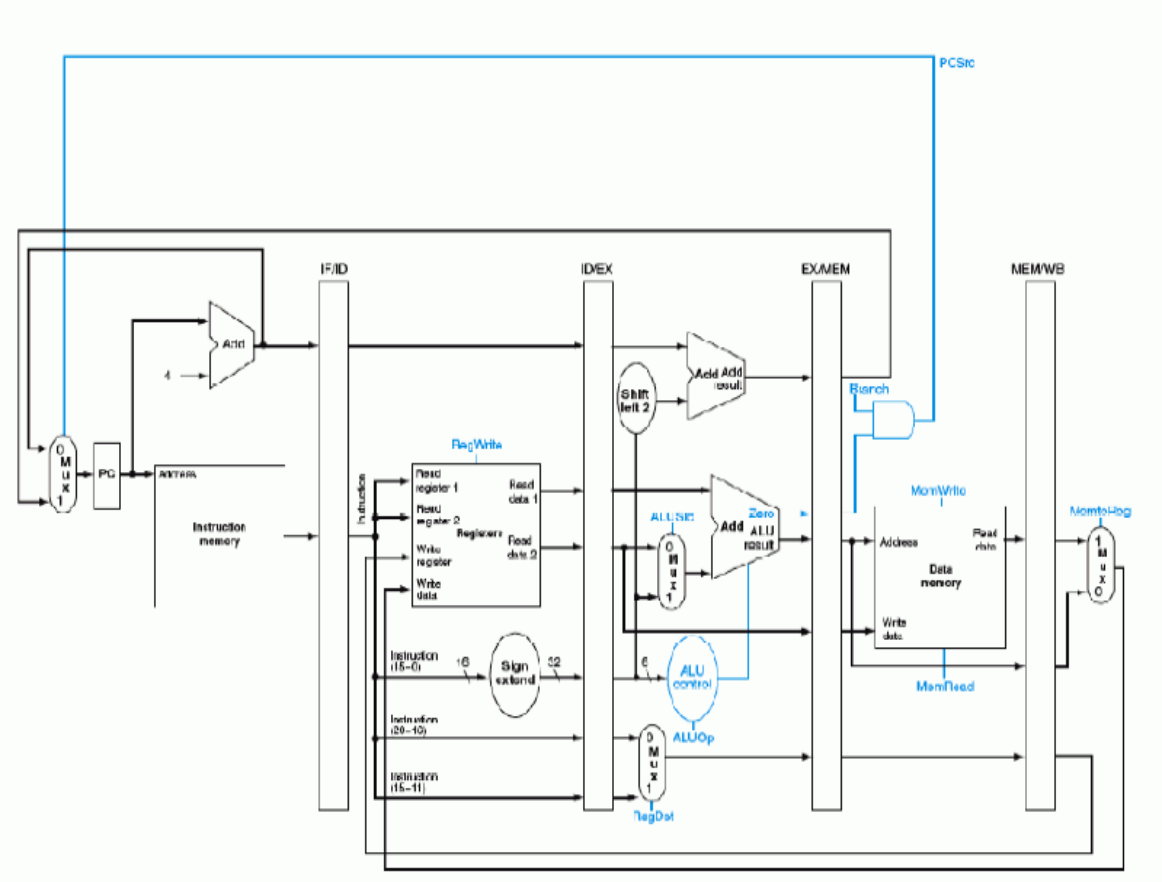

Figure 16: MIPS design after adding ALU Control [1]

Control:

Control is one of the most important modules for the design. It generates different control signals for various modules in the design. It mainly implements control signals which are passed from pipelined registers.

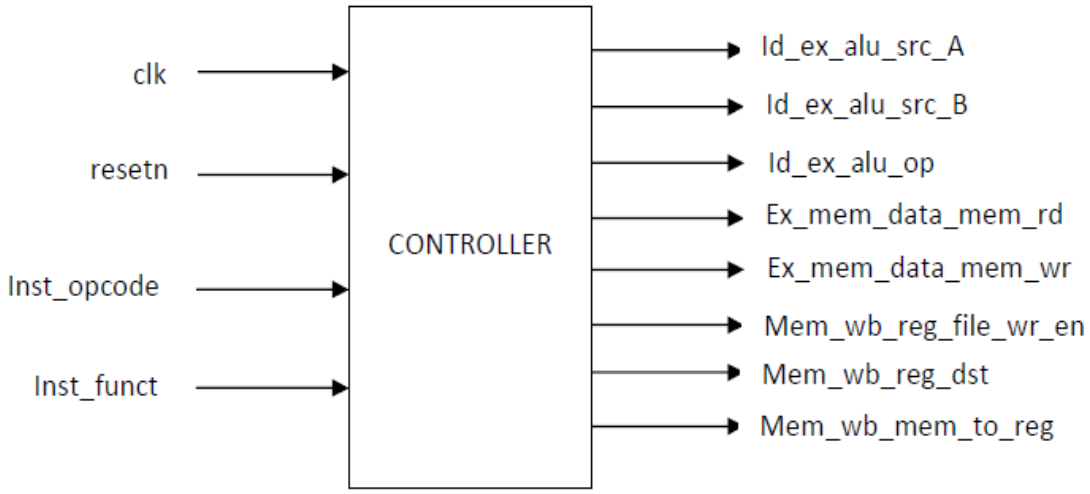

Figure 17: Control Module

	func	10 0000	10 0010	n/a			
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100
		add	**sub**	**ori**	**lw**	**sw**	**beq**
Control Signals	**RegDst**	1	1	0	0	X	X
	ALUSrc	0	0	1	1	1	0
	MemtoReg	0	0	0	1	X	X
	RegWrite	1	1	1	1	0	0
	MemWrite	0	0	0	0	1	0
	nPC_sel	0	0	0	0	0	1
	ExtOp	X	X	0	1	1	X
	ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract

All Supported Instructions

Figure 18: State Diagram of my Control Module

- Op - instruction opcode field from instruction memory
- Func - instruction function field.
- RegDst - sel line to write address mux. This signal is for register writeback.
- ALUSrc - sel line for ALU I/p B sel Mux required in inst_execute stage
- MemtoReg - reg file write enable signal, required for register writeback
- RegWrite – Gives the address for the write back stage to the registers block.
- MemWrite - write signal for data memory. This signal is for memory_acc stage
- nPC_sel – pc_sel signal for branch instruction.
- AluCTR<2:0> - sel line to select the ALU operation. This signal is for inst_execution stage

From the above table, we can get the combinations for the control signals:

r = ~Op[5]&~Op[4]&~Op[3]&~Op[2]&~Op[1]&~Op[0];

lw = Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];

sw = Op[5]&~Op[4]&Op[3]&~Op[2]&Op[1]&Op[0];

beq = ~Op[5]&~Op[4]&~Op[3]&Op[2]&~Op[1]&~Op[0];

bne = ~Op[5]&~Op[4]&~Op[3]&Op[2]&~Op[1]&Op[0];

j = ~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&~Op[0];

andi = ~Op[5]&~Op[4]&Op[3]&Op[2]&~Op[1]&~Op[0];

ori = ~Op[5]&~Op[4]&Op[3]&Op[2]&~Op[1]&Op[0];

addi = ~Op[5]&~Op[4]&Op[3]&~Op[2]&~Op[1]&~Op[0];

imm = andi | ori | addi; //immediate value type

regdst = r;

alusrc = lw | sw | imm;

memtoreg = lw;

regwrite = r | lw | imm;

memread = lw;

memwrite = sw;

nPC_sel = beq

The following figure shows how control block can be included in our design.

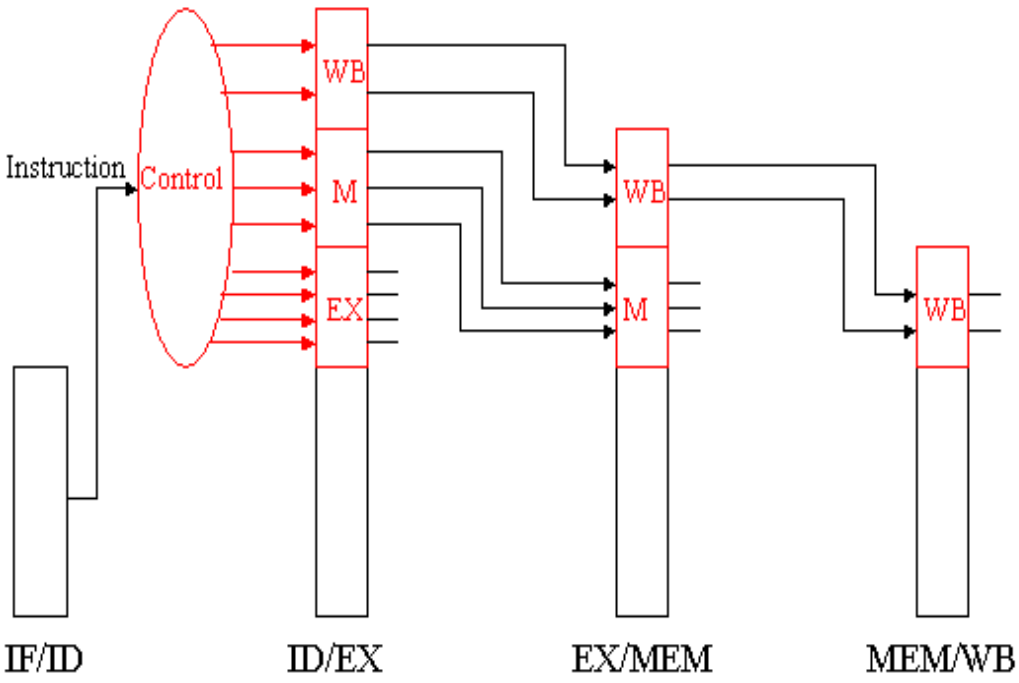

Figure 19: Dipiction of propogation of control signals through MIPS [7]

Now after adding the control block, the whole structure can be simply depicted as shown in the figure below

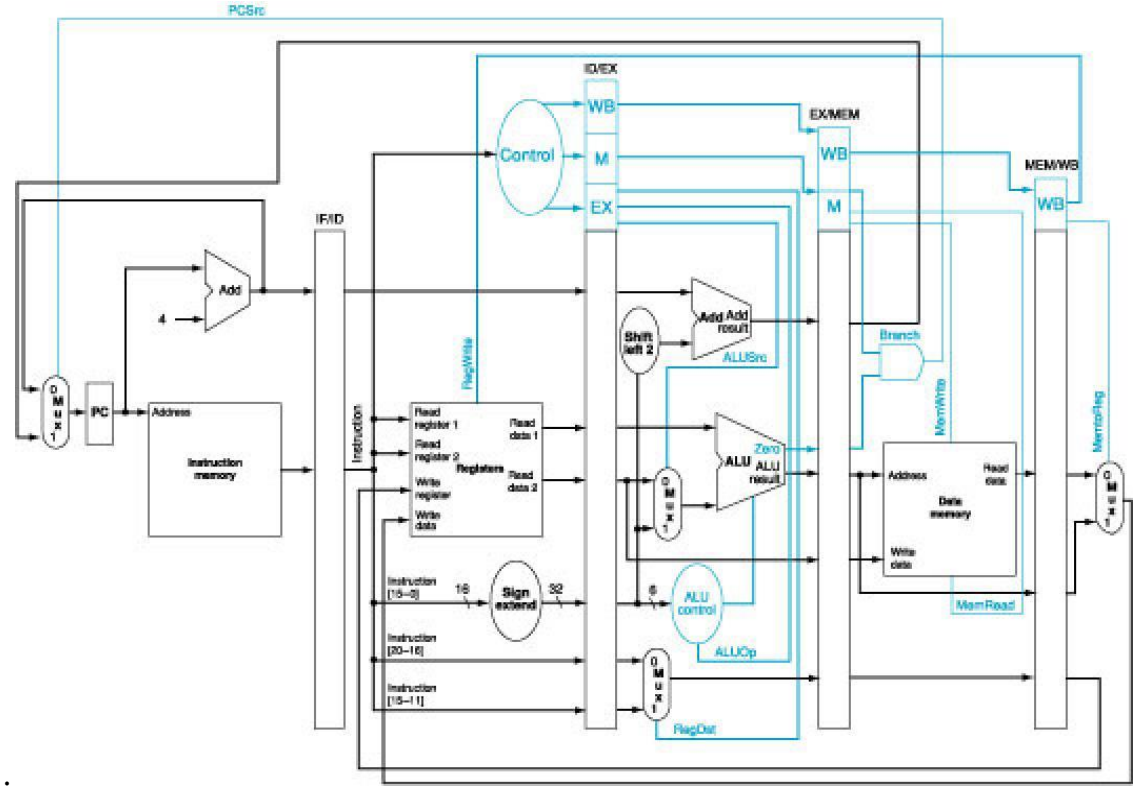

Figure 20: 5-Stage pipelined MIPS processor [1]

Chapter 4: Simulation and Synthesis

For simulating with vcs, I wrote my codes in Verilog. I divided my MIPS processor as follows:

- The top module was divided into 2 modules, control module cntl_path and data module data_path respectively. It is as below:

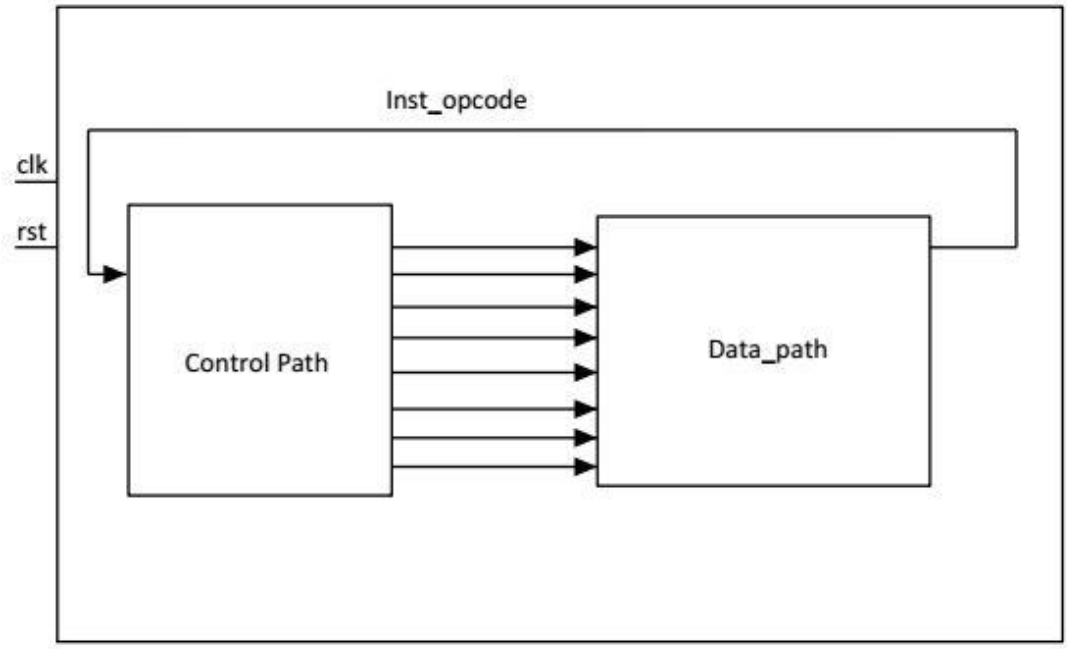

Figure 21: MIPS top module

The cntl_path gives all the control signals, required for datapath.

- The data path contains all the stages of pipelining, Instruction fetch(inst_fetch), instruction decode(inst_decode), instruction execute(inst_execute) and memory access(memory_acc). Writeback stage is directly coded in the data_path stage, ehere all these modules are instantiated. The following is the depiction of data path:

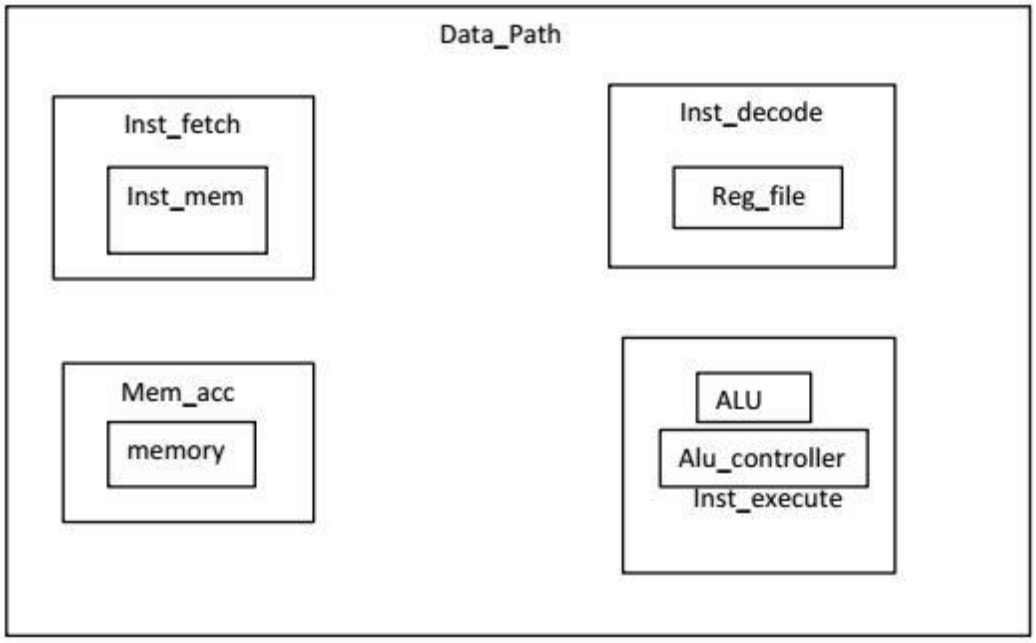

Figure 22: data_path

From the above figure we can see that:

- The inst_fetch moduleconsists of inst_mem instantiation
- Inst_decode block has an instantion of reg_file
- Inst_execute has both alu and alu_controller
- Mem_acc has a memory block in it.

4.1 Simulation

The Simulation of my pipelined MIPS processor was done using VCS simulator and the synthesis was done on Synopsys Design Compiler. The following is the schematic for my design:

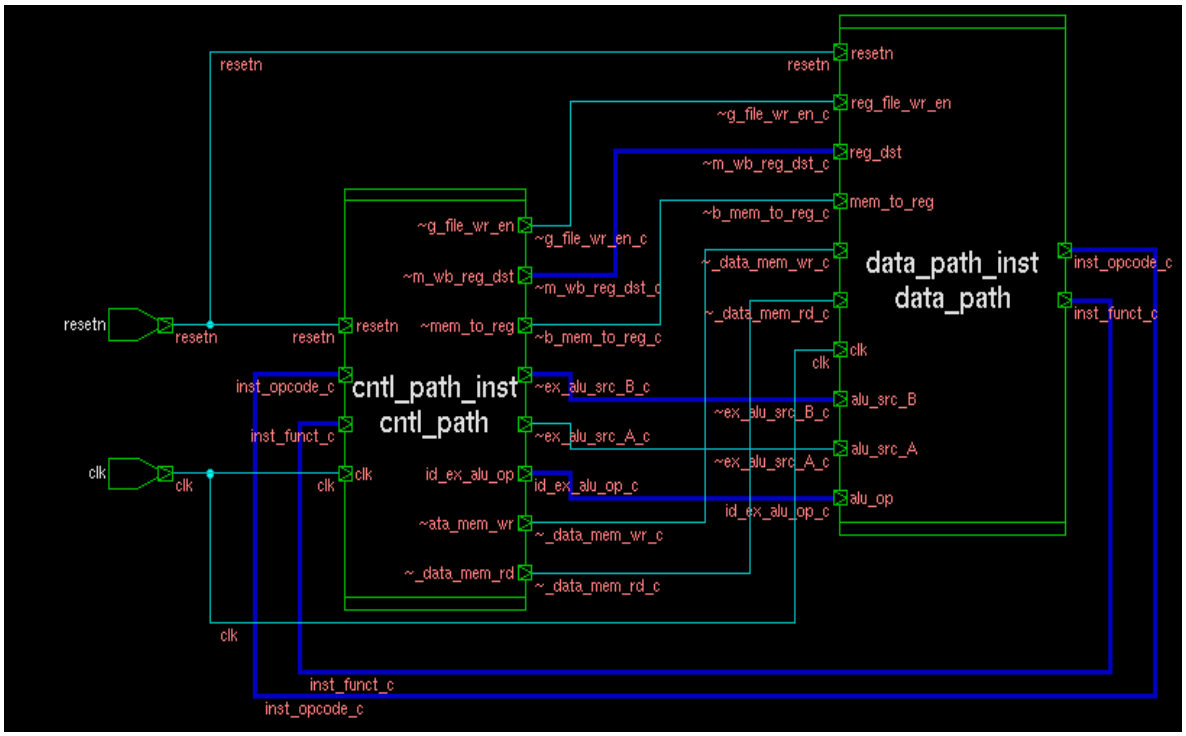

Figure 23: Top module schematic

The above schematic shows the design's top module. It contains a Control Path (cntl_path) and Data Path (data_path).

The following is the schematic for control path (cntl_path):

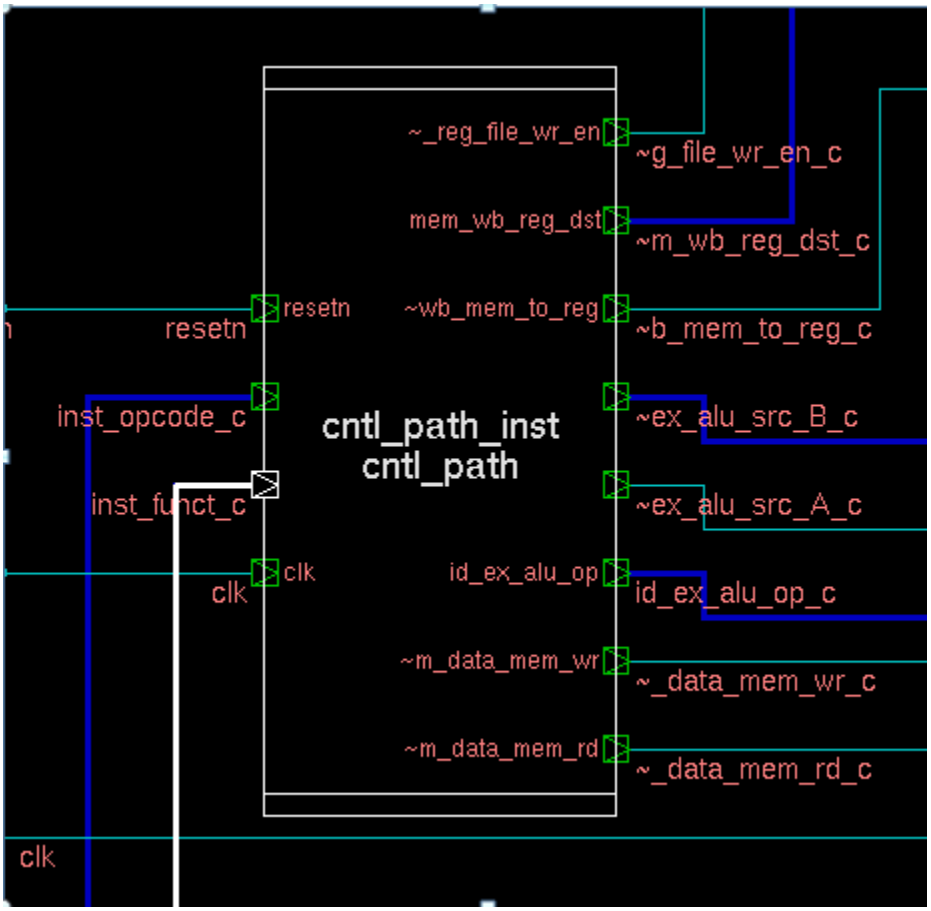

Figure 24: Control schematic

From the above control schematic, we can see that the inputs of control path are inst_opcode and inst_func, which come from data_path. We can also see the output control signals, which go to data_path.

The following is the data_path

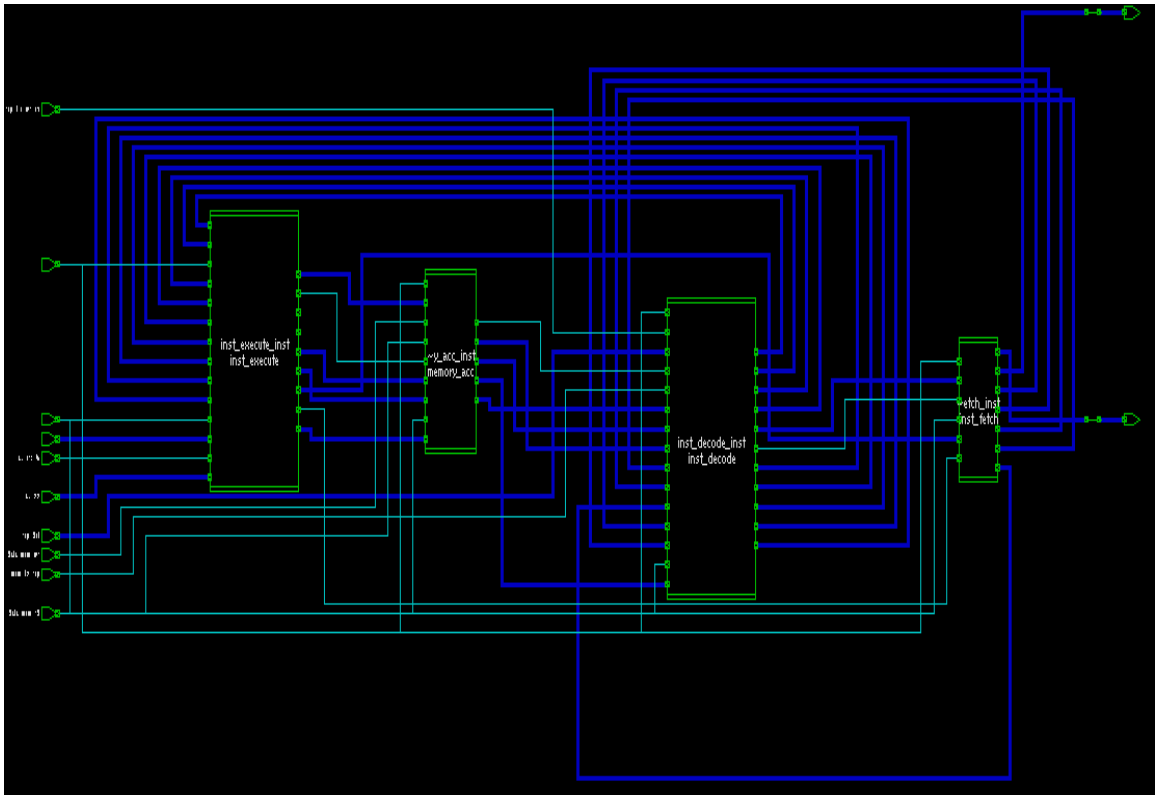

Figure 25: Data_path schematic

We can see that the above data_path has 4 blocks. The blocks are inst_execute, memory_acc, inst_decode and inst_fetch respectively. The detailed schematics of each one of them are shown below.

The following is the schematic for inst_fetch:

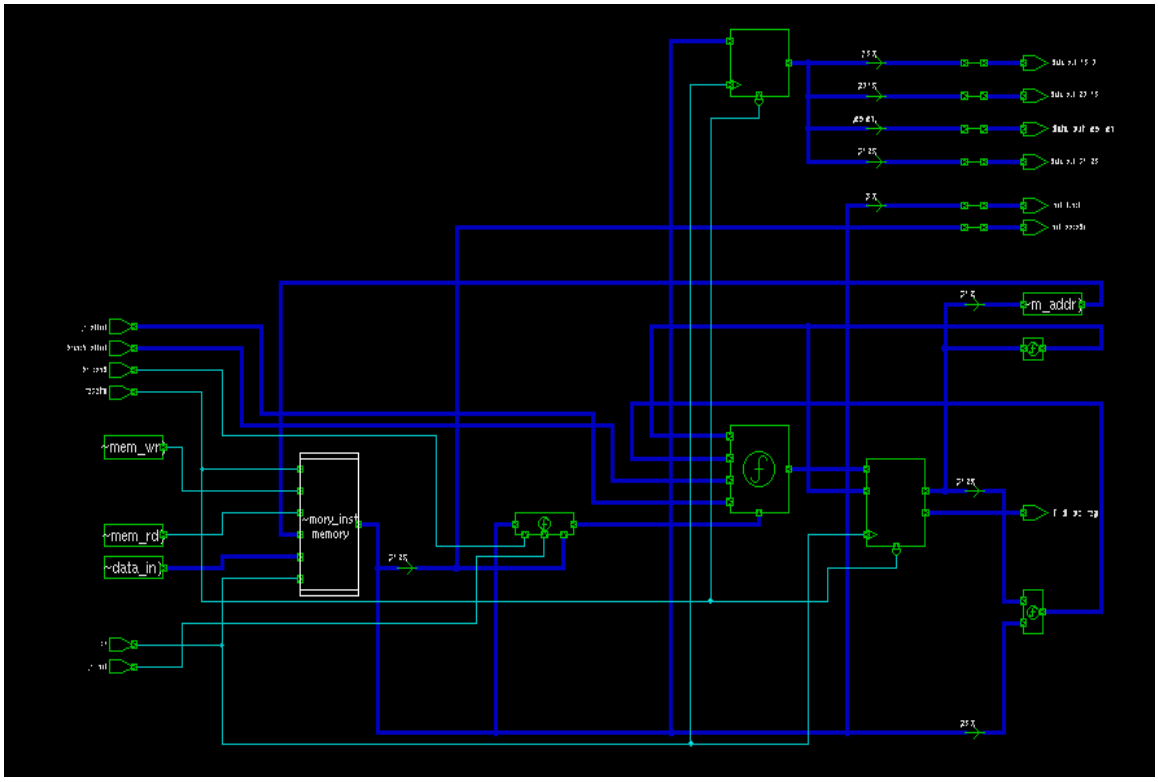

Figure 26: Instruction Fetch schematic

The first block is instruction memory (inst_mem). I used my memory module in the place of instruction_memory. PC is also included in this. The following is the schematic.

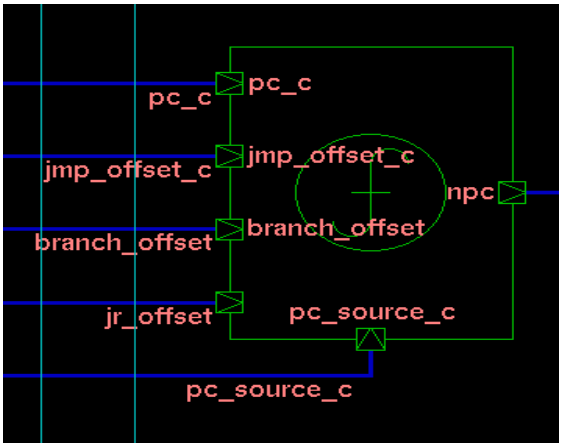

Figure 27: PC Schematic

In the above PC module, pc_c is nothing but PC+4. Others are nothing but jump and branch conditions.

The following is the schematic for inst_decode

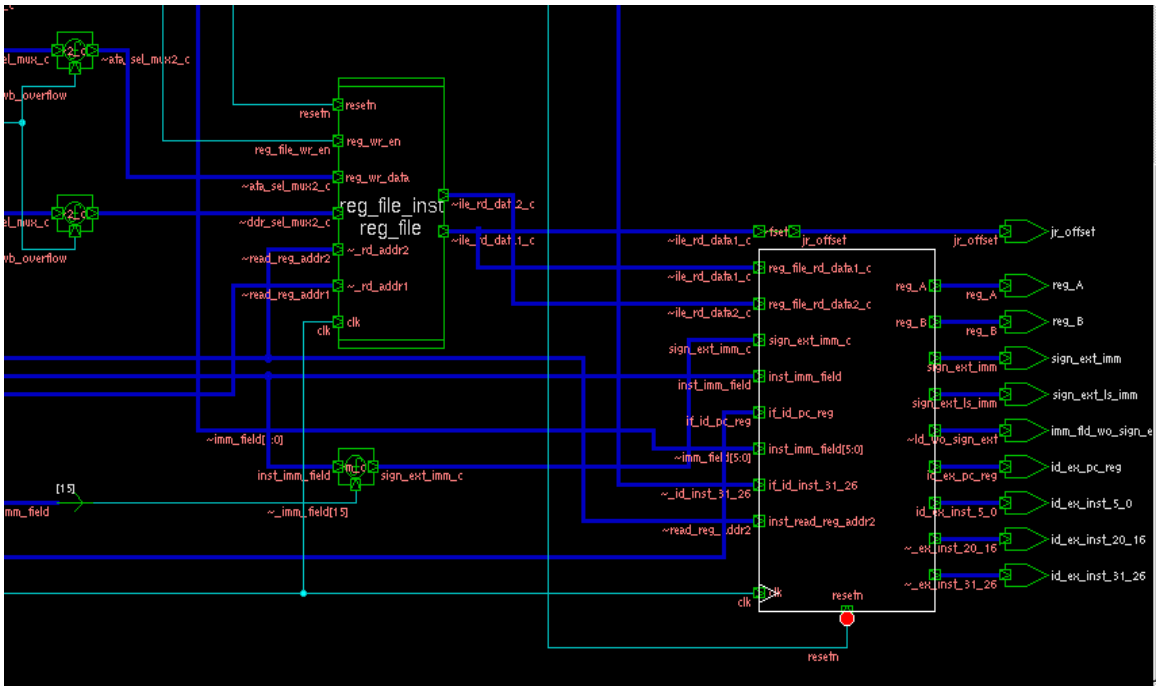

Figure 28: Instruction Decode schematic

The first block is reg_file. And the second one has all the data registers, sign extend etc.

The following is the schematic for reg_file:

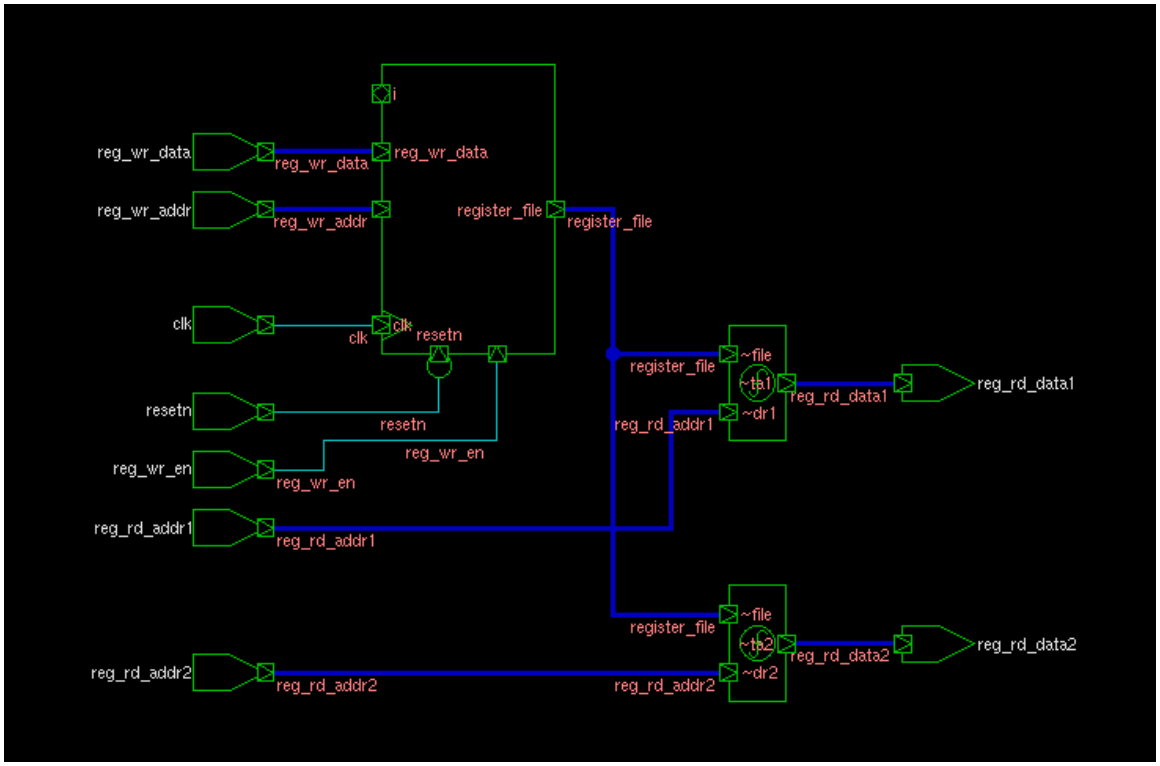

Figure 29: register file schematic

The following is the schematic for inst_execute:

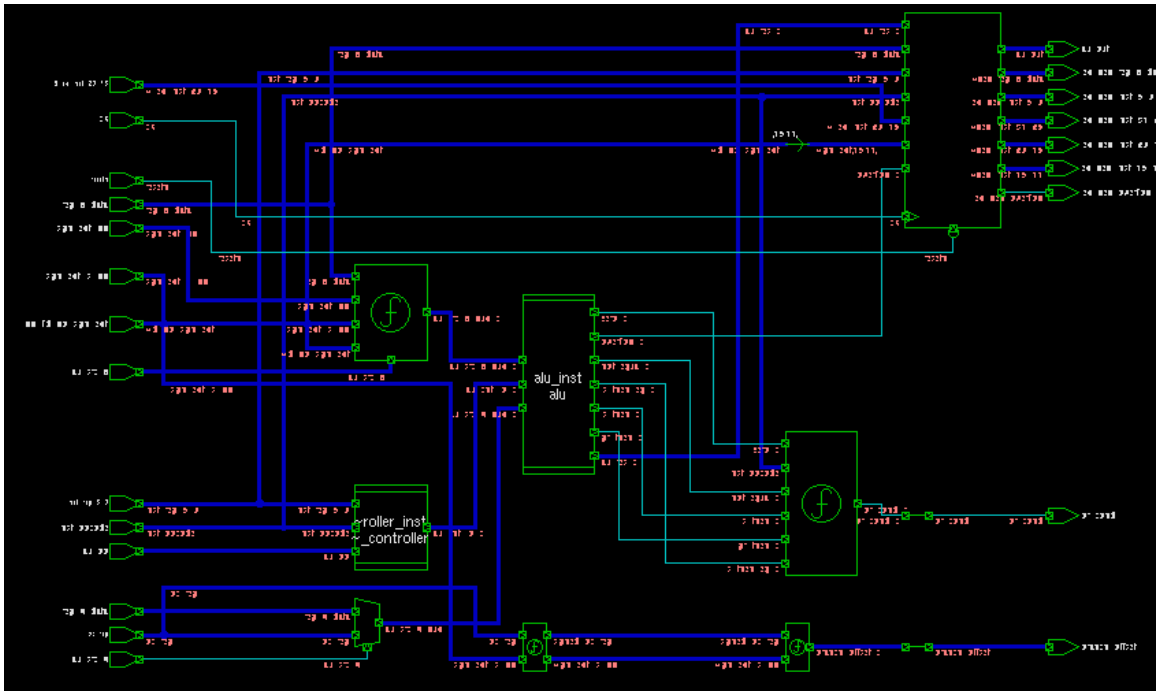

Figure 30: instruction execute schematic

In the above schematic, we can clearly see the alu and alu_controller. The following are the schematics for alu and alu_controller respectively:

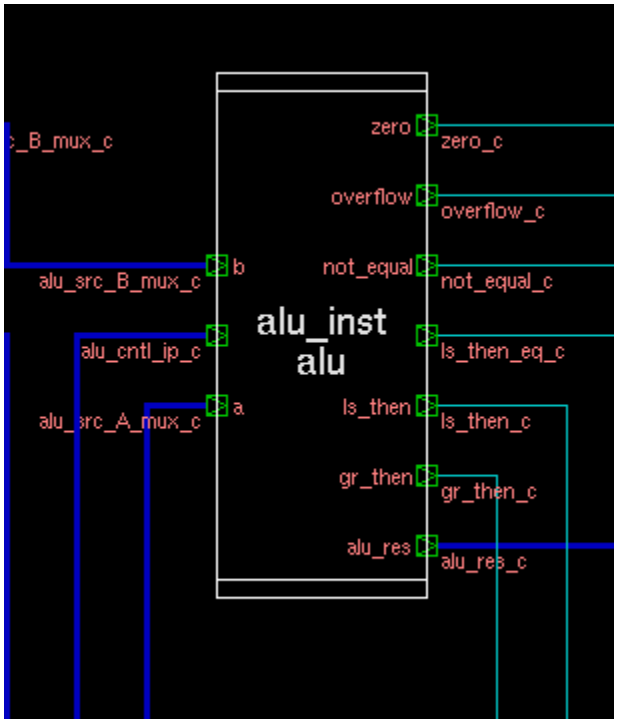

Figure 31: ALU Schematic

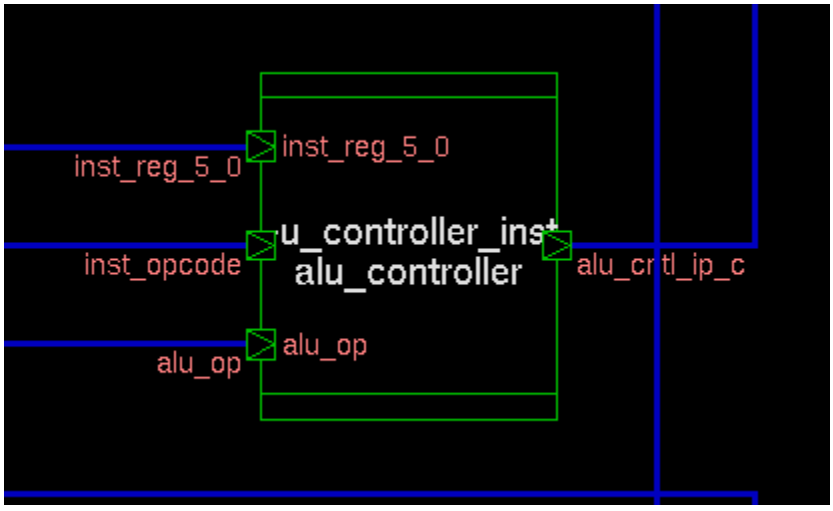

Figure 32:alu control schematic

I wrote a test bench to test my design. I have sent a series of instructions, which were read from a memory to instruction memory. I used monitor statements to check my outputs. The following are the instructions I have sent:

HEX Code	Assembly Code
8C030000	lw R3,0(R1)
8C040001	lw R4,1(R0)
00642820	add R5,R3,R4
00A43022	sub R6,R5,R4
00643824	and R7,R3,R4
00644025	or R8,R3,R4
00644827	nor R9,R3,R4
00C5502A	slt R10,R6,R5
80000008	j startloop
2063FFFF	loop: addi R3,R3,-1
14E3FFFE	startloop: bne R3,R7,-2
01295818	mult R11,R9,R9
0166601A	div R12,R11,R6
34CE0002	ori R14,R6,2
11CC0000	beq R14,R12, next
ADCE0006	sw

The registers R0 and R1 are stored with data 32'd8 and 32'd1 respectively.

The following are some of the observations I have made based on the outputs:

```
# R3:    0 R4:    0 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    4
# R3:    0 R4:    0 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    8
# R3:    0 R4:    0 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    12
# Does WrReg:  0 Data:    0
# R3:    0 R4:    0 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# Skipped writting to PC - nop
# Does WrReg:  0 Data:    0
# Does WrReg:  3 Data:    0
# Does WrReg:  3 Data:    8
# R3:    8 R4:    0 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# R3:    8 R4:    0 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    16
# Does WrReg:  3 Data:    8
# Does WrReg:  4 Data:    8
# Does WrReg:  4 Data:    1
# R3:    8 R4:    1 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# R3:    8 R4:    1 R5    0 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    20
# Does WrReg:  4 Data:    1
# Does WrReg:  5 Data:    1
# Does WrReg:  5 Data:    9
# R3:    8 R4:    1 R5    9 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# R3:    8 R4:    1 R5    9 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    24
# Does WrReg:  5 Data:    9
# R3:    8 R4:    1 R5    9 R6:    0 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    28
# Does WrReg:  5 Data:    9
# Does WrReg:  6 Data:    9
# Does WrReg:  6 Data:    8
# R3:    8 R4:    1 R5    9 R6:    8 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# R3:    8 R4:    1 R5    9 R6:    8 R7:    0 R8    0 R9:    0 R10:    0 R11    0 R12:    0 R13:    0 R14
0
# PC:    32
```

Figure 33: Simulation Output 1

From the above results, we can observe that:

- PC is incrementing by 4 as required.
- First two instructions, lw(load word) can be observed at PC = 12 and PC = 16 respectively. The data of R0 and R1 are stored into R3 and R4 respectively.

- At PC = 20 and PC = 28, we can see that the 3rd and 4th instructions ‘add R5,R3,R4’ and ‘sub R6,R5,R4’ are executed.
- The below result shows the ‘or R8, R3, R4’ operation

```
# Does WrReg:  7 Data:       0
# Does WrReg:  8 Data:       0
# Does WrReg:  8 Data:       9
# R3:       8 R4:       1 R5       9 R6:       8 R7:       0 R8       9 R9:       0 R10:       0 R11       0 R12:       0 R13:       0 R14       0
# R3:       8 R4:       1 R5       9 R6:       8 R7:       0 R8       9 R9:       0 R10:       0 R11       0 R12:       0 R13:       0 R14       0
# PC:       40
```

Figure 34: Simulation Output 2

Similarly all the other instructions can be observed from the complete simulation results, which I have added in the appendix

4.2 Synthesis:

Design compiler was used to synthesize the design. The library used was SAED_EDK90nm. I optimized the design for timing and area. I used frequencies varying from 10MHz to 1000MHz. After verifying various constraints, the following are my observations:

- I found the best optimized design with small area at approximately 25MHz. Here the area = 360033.6, arrival time = 41.89ns, required time = 41.82ns and slack = -0.08ns.
- Then I tried to further optimize it by using set_area_max 100. The area was further reduced to 354870.5 and both the arrival and required time are 41.72ns, thereby making the slack 0.
- If I take into consideration, the fastest design, the fastest design was at the frequency of nearly 100MHz. Here the area is more which is 395974.2, arrival time 37.15 and required time is 9.83. But for this design, the slack has increased, which is -27.32ns.

From the above deductions, we can clearly see that both the area as well as timing cannot be optimized at once. If area decreases, timing increases and vice-versa.

Chapter 5: Conclusion and Future Enhancements

5.1 Conclusion

The project has been successfully implemented. Starting from RISC and CISC architecture, the pipelines MIPS architecture and its advantages over single and multi-cycle processor were clearly understood. I was able to learn synopsis tools like VCS, Design Compiler. TCL scripting was used to write the synthesis scripts, hence few basics about TCL scripting was understood. The main advantage is, understanding of synthesis part of the design. I was able to optimize my MIPS processor both in timing and area.

5.2 Future Enhancements:

The future enhancements which can be considered are detection of various hazards like data hazards, control hazards etc. A forwarding unit and hazard detection unit can be added for the detection of data hazard. CACHE memory can be implemented in place of memory. Also we can do the complete ASIC flow till GDS II, and understand the complete physical design.

References

1. David A. Patterson, John L. Hennessy: Computer Organization and Design – The Hardware/Software Interface. Fourth Edition (2006). Morgan Kaufmann Publisher, Inc.
2. http://nptel.iitm.ac.in/video.php?subjectId=106102062 Computer Architecture Principals video tutorials, Retrieved: August, 2013
3. http://www.iitg.ernet.in/asahu/cs222/ Lectures on Computer Organization and Architecture, Retrieved: August, 2013
4. Prof. Grishman, http://cs.nyu.edu/courses/fall08/V22.0436-001/lecture18.html retrieved: September, 2013
5. MIPS architectur, http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/MIPS.html retrieved: September, 2013.
6. Multicycle processor ppt of EE422 class of CSUN sent to me by Dr. Roosta
7. Synopsys VCS User guide
8. M.S.Schmalz(n.d.), Organization of Computer Systems, http://www.cise.ufl.edu/~mssz/CompOrg/CDA-pipe.html retrieved: August, 2013

Appendix A: Code Listing

Mips_top_tb.v

Mips_top.v

- cntl_path.v
- data_path.v
    - inst_fetch
        - imst_mem.v
    - inst_decode
        - reg_file.v
    - inst_execute
        - alu.v
        - alu_cntl.v
    - mem_acc.v
        - memory.v

Instruction_Mem_Load.dat

Register_File_Load.dat

Data_Mem_Load.dat

Instruction_Mem_Dump.txt

Register_File_Dump.txt

Data_Mem_Dump.txt

Appendix B: Simulation results

Chronologic VCS simulator copyright 1991-2013

Contains Synopsys proprietary information.

Compiler version H-2013.06; Runtime version H-2013.06; Dec 6 02:01 2013

VCD+ Writer H-2013.06 Copyright (c) 1991-2013 by Synopsys Inc.

R3: 0 R4: 0 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 4

R3: 0 R4: 0 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 8

R3: 0 R4: 0 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 12

Does WrReg: 0 Data: 0

R3: 0 R4: 0 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

Skipped writting to PC - nop

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 8

R3: 8 R4: 0 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 0 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 16

Does WrReg: 3 Data: 8

Does WrReg: 4 Data: 8

Does WrReg: 4 Data: 1

R3: 8 R4: 1 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 1 R5 0 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 20

Does WrReg: 4 Data: 1

Does WrReg: 5 Data: 1

Does WrReg: 5 Data: 9

R3: 8 R4: 1 R5 9 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 1 R5 9 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 24

Does WrReg: 5 Data: 9

R3: 8 R4: 1 R5 9 R6: 0 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 28

Does WrReg: 5 Data: 9

Does WrReg: 6 Data: 9

Does WrReg: 6 Data: 8

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 32

Does WrReg: 6 Data: 8

Does WrReg: 7 Data: 8

Does WrReg: 7 Data: 0

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 0 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 7 Data: 0

Does WrReg: 8 Data: 0

Does WrReg: 8 Data: 9

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 0 R10:

0 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 8 Data: 9

Does WrReg: 9 Data: 9

Does WrReg: 9 Data: 4294967286

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 0 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 0 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 9 Data: 4294967286

Does WrReg: 10 Data: 4294967286

Does WrReg: 10 Data: 1

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 10 Data: 1

R3: 8 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 7

R3: 7 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 7 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 7

R3: 7 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 7

Does WrReg: 0 Data: 0

R3: 7 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 6

R3: 6 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 6 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 6

R3: 6 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 6

Does WrReg: 0 Data: 0

R3: 6 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 5

R3: 5 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 5 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 5

R3: 5 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 5

Does WrReg: 0 Data: 0

R3: 5 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 4

R3: 4 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 4 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 4

R3: 4 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 4

Does WrReg: 0 Data: 0

R3: 4 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 3

R3: 3 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 3 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 3

R3: 3 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 3

Does WrReg: 0 Data: 0

R3: 3 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 2

R3: 2 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 2 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 2

R3: 2 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 2

Does WrReg: 0 Data: 0

R3: 2 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 1

R3: 1 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 1 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 1

R3: 1 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 36

Does WrReg: 0 Data: 1

Does WrReg: 0 Data: 0

R3: 1 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 40

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

R3: 0 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

R3: 0 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 44

Does WrReg: 3 Data: 0

R3: 0 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 48

Does WrReg: 0 Data: 0

R3: 0 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286

R10: 1 R11 0 R12: 0 R13: 0 R14 0

PC: 52

Does WrReg: 0 Data: 0

Does WrReg: 3 Data: 0

Does WrReg: 3 Data: 4294967295

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9:

4294967286 R10: 1 R11 0 R12: 0 R13: 0 R14

0

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 0 R12: 0 R13: 0 R14

0

PC: 56

Does WrReg: 3 Data: 4294967295

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 0 R12: 0 R13: 0 R14

0

PC: 60

Does WrReg: 11 Data: 4294967295

Does WrReg: 11 Data: 100

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 0 R13: 0 R14

0

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 0 R13: 0 R14

0

PC: 60

Does WrReg: 11 Data: 100

Does WrReg: 12 Data: 100

Does WrReg: 12 Data: 12

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14 0

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14 0

PC: 64

Does WrReg: 12 Data: 12

Does WrReg: 14 Data: 12

Does WrReg: 14 Data: 10

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14 10

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14 10

PC: 68

Does WrReg: 14 Data: 10

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14 10

PC: 72

Does WrReg: 0 Data: 4294967294

Writing 10 -> Addr: 16

Does WrReg: 0 Data: 0

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14

10

PC: x

Does WrReg: 0 Data: 0

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 0 R14

10

PC: x

Does WrReg: 13 Data: 16

Does WrReg: 13 Data: 10

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 10 R14

10

R3: 4294967295 R4: 1 R5 9 R6: 8 R7: 0 R8 9 R9: 4294967286 R10: 1 R11 100 R12: 12 R13: 10 R14

10

PC: x

Does WrReg: 13 Data: 10

Appendix c: TCL Scripts

Top_down.tcl

Bottom_up.tcl

- cntl_path.tcl
- data_path.tcl
    - Mem_acc.tcl
        - Memory.tcl
    - Inst_fetch.tcl
        - Memory.tcl
    - Inst_decode.tcl
        - Reg_file.tcl
    - Inst_execute.tcl
        - Alu.tcl
        - Alu_controller.tcl

The above TCL scripts also contain characterize and don't_touch attributes in them. I was just editing them as per requirement.

Various area and timing reports are also included.