Spring 2020

EE 275
Midterm Project: I

Last Name :-	Nikola	Patel	Yadav
First Name :-	Pal	Dharm	Rushiraj
ID :-	014512836	014532908	014513031
Date:-	Arpil-16,2020		

1. Introduction

The MIPS processor, designed in 1984 by researchers at Stanford University, is a RISC (Reduced Instruction Set Computer) processor. Compared with their CISC (Complex Instruction Set Computer) counterparts (such as the Intel Pentium processors), RISC processors typically support fewer and much simpler instructions. The premise is, however, that a RISC processor can be made much faster than a CISC processor because of its simpler design. These days, it is generally accepted that RISC processors are more efficient than CISC processors; and even the only popular CISC processor that is still around (Intel Pentium) internally translates the CISC instructions into RISC instructions before they are executed. RISC processors typically have a load-store architecture. This means there are two instructions for accessing memory: a load (l) instruction to load data from memory and a store (s) instruction to write data to memory. It also means that none of the other instructions can access memory directly. So, an instruction like "add this byte from memory to register 1" from a CISC instruction set would need two instructions in a load-store architecture: "load this byte from memory into register 2" and "add register 2 to register 1".

2. Basic Processor Architecture

The execution of an instruction in a processor can be split up into a number of stages. How many stages there are, and the purpose of each stage is different for each processor design. Examples includes 2 stages (Instruction Fetch / Instruction Execute) and 3 stages (Instruction Fetch, Instruction Decode, Instruction Execute). The MIPS processor has 5 stages:

IF:- The Instruction Fetch stage fetches the next instruction from memory using the address in the PC (Program Counter) register and stores this instruction in the IR (Instruction Register)
ID:- The Instruction Decode stage decodes the instruction in the IR, calculates the next PC, and reads any operands required from the register file.
EX:- The Execute stage "executes" the instruction. In fact, all ALU operations are done in this stage. (The ALU is the Arithmetic and Logic Unit and performs operations such as addition, subtraction, shifts left and right, etc.)
MA:- The Memory Access stage performs any memory access required by the current instruction, So, for loads, it would load an operand from memory. For stores, it would store an operand into memory. For all other instructions, it would do nothing.
WB:- For instructions that have a result (a destination register), the Write Back writes this result back to the register file. Note that this includes nearly all instructions, except nops (a nop, no-op or no-operation instruction simply does nothing) and s (stores).

3. Processor Modules

I. Program Counter & Instruction Memory Block

Signals to PC_IM Block	Input variable Name	No. of Bits	Explanation
Inputs	jmp_loc	16	Address of jump location
	pc_mux_sel	1	Selection bit for jmp_loc or pc_out
	stall	1	Selection bit for address control mux
	stall_pm	1	Selection bit for instruction control mux
	reset	1	Reset signal for register, address and instruction
	clk	1	Input clock signal
Outputs	ins	32	Instruction
	current_address	16	Present address value for IM

Table 2.1 Inputs & Outputs of the Block

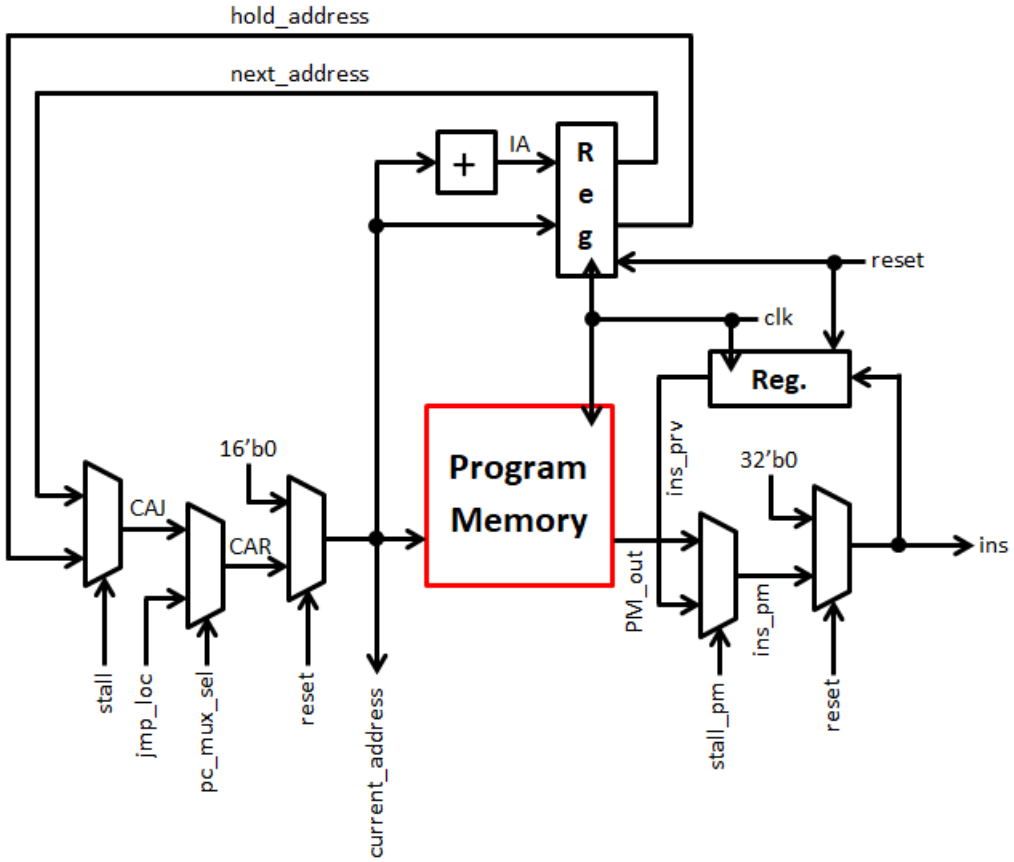

Figure 2.1 Program Counter & Instruction Memory

Block Diagram Description:

The objective of this block is to generate instructions for decode, execution and data memory blocks, and also take care of different controls. There are 3 different 2 x 1 multiplexer having selection line namely ‘stall’, ‘pc_mux_sel’ and ‘stall_pm’ respectively. ‘stall’ generated from ‘stall control’ block (i.e. one of the module of processor) which is used to select assign either hold_address or next_address as CAJ. ‘pc_mux_sel’ decides whether current address will pass to program memory or address provided by ‘jump control’ block. A mux with ‘stall_pm’ selection bit decides whether to pass actual instruction (PM_out) or pass previous instruction (ins_prv) to ‘ins_pm’. Another two multiplexers with ‘reset’ selection bit are used to reset ‘current_address’ and ‘ins’ values. Processor needs to generate new instruction at every positive edge of clock and ‘program counter’ will provide address of the new instruction. Program counter is combination of ‘increment’ (+) and ‘register’ block. ‘Increment’ block increments address and register will pass that value to ‘next_address’ on every positive edge of the clock and when there is reset it will clear the output of ‘register’. ‘reset’ is negative level triggered and clock synchronized. Another important block is program memory which has been designed using Xilinx IP core generator.

Program Memory Specifications:

- Single port ROM
- Data Read width is 32 bits
- Data Read depth is 65536 locations
- Inputs are 16-bit address, clock and reset
- Output is 32-bit instruction

II. Dependency Check & Instruction Decode

Signals to DC Block	Input variable Name	No. of Bits	Explanation
Inputs	ins	32	Instruction from Program Memory
	clk	1	Input clock signal
	reset	1	Reset for registers/FFs
Outputs	imm	16	Immediate number
	op_dec	6	Opcode
	RW_dm	5	Write address
	mux_sel_A	2	Selection bit for Mux A
	mux_sel_B	2	Selection bit for Mux B
	imm_sel	1	Immediate number select
	mem_en_ex	1	Memory enable signal
	mem_rw_ex	1	Memory read/write signal
	mem_mux_sel_dm	1	Memory mux select bit

Table 2.2 Inputs & Outputs of the Block

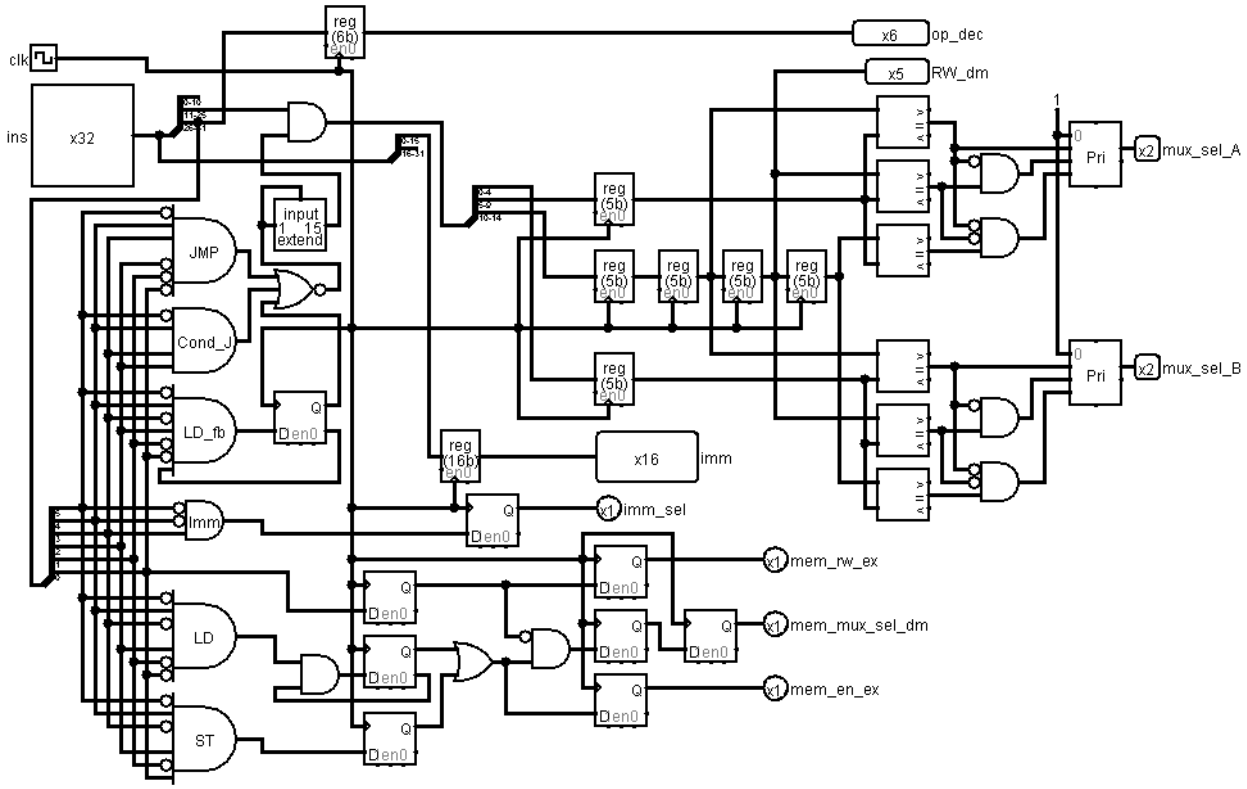

Figure 2.2 Dependency Check & Instruction Decode Block

Block Diagram Description:

Dependency check block handles data hazard in pipeline processor. It can check dependency of current instruction with previous three instructions and enables data forwarding in pipeline by generating 'mux_sel_A' and 'mux_sel_B' signals. This block also generates memory control and immediate control signals. All outputs of this block are generated at the positive edge of clock. Also, the block is responsible for decoding the opcode for execution stage as well as it will decode the addressed for register A, B and the address of the register where data is supposed to be stored.

III. Register Bank

Signals to RB Block	Input variable Name	No. of Bits	Explanation
Inputs	ans_ex	16	Answer from Execution block
	ans_dm	16	Answer from Data Memory block
	ans_wb	16	Answer from Writeback block
	imm	16	Reset signal for register, address and instruction
	RA	5	Register A read address
	RB	5	Register B read address
	RW_dm	5	Write address
	mux_sel_A	2	Mux selection bit for A
	mux_sel_B	2	Mux selection bit for BI
	imm_sel	1	Immediate number select
	clk	1	Input clock signal
Outputs	A	16	First operand for Execution block
	B	16	Second operand for Execution block

Table 2.3 Inputs & Outputs of the Block

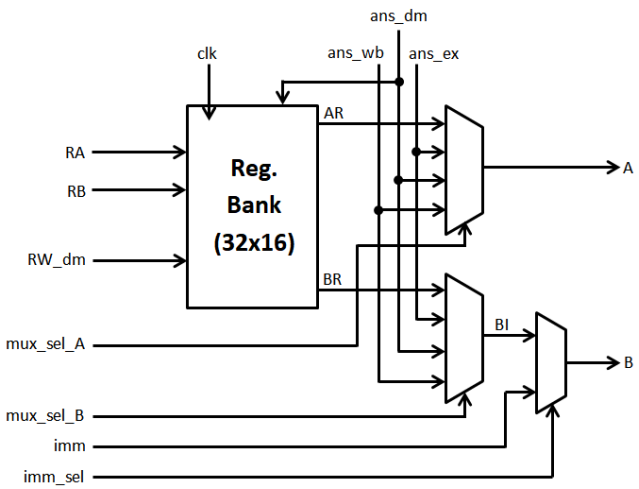

Figure 2.3 Register Bank

Block Diagram Description:

This block generates operands (A and B) for Execution block based on the instruction generated from Instruction Memory block. Input RA is used as first operand (A) address and input RB is used as second operand (B) address. If applied instruction is register type then both the operands fetched from register bank, else one of the operand (A) fetched from register bank and other operand (B) taken as immediate number (imm). This selection is controled by 2x1 Mux (with selection line imm_sel). Data forwarding operations performed by both 4x1 Mux based on the values applied to it's selection line (mux_sel_A and mux_sel_B). Answer from Data Memory block (ans_dm) is written in register bank as per the address indicated by RW_dm.

mux_sel_A	A
00	AR
01	ans_ex
10	ans_dm
11	ans_wb

mux_sel_B	BI
00	BR
01	ans_ex
10	ans_dm
11	ans_wb

imm_sel	B
0	BI
1	imm

Figure 2.3 Mux Functionality

IV. Execution Block

Signals to Execution Block	Input variable Name	No. of Bits	Explanation
Inputs	A	16	First operand for ALU operations
	B	16	Second operand for ALU operations
	data_in	16	Data from external input port
	op_dec	6	Opcode
	clk	1	Clock for execution block
	reset	1	Reset signal to clear register of execution block
Outputs	ans_ex	16	Execution block output
	DM_data	16	DM_data = B (Data for Data Memory block)
	data_out	16	Data to external output port
	flag_ex	2	Status bits :- flag_ex[1] = zero; flag_ex[0] = overflow;

Table 2.4 Inputs & Outputs of the Block

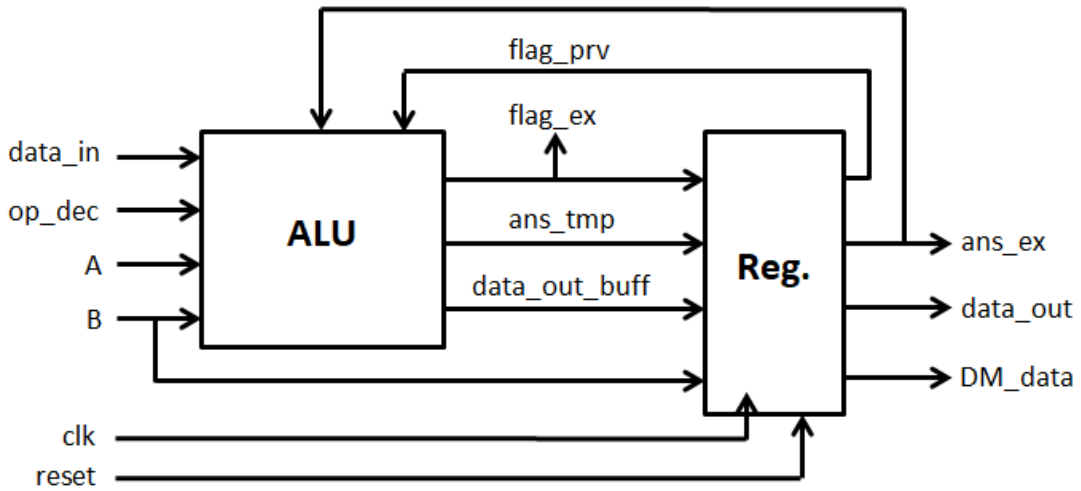

Figure 2.4 Execution Block

Block Diagram Description:

This block will take values A and B from register bank and perform operation based on the opcode. On the based of the operation performed the flags will be generate (zero flag & overflow flag). The result of the operation performed is then passed on to the next stage i.e. Data Memory Block. All the opcodes and respective operations with flag information and instruction set for the ALU is given below.

Sr. No.	Instruction	Opcode	Operation in Execution Block	Flags Affected
1	ADD	000000	ans_ex = A+B	Overflow, Zero
2	SUB	000001	ans_ex = A-B	Overflow, Zero
3	MOV	000010	ans_ex = B	Zero (Reset Overflow flag)
4	MUL	000011	ans_ex = A*B	Zero (Reset Overflow flag)
5	AND	000100	ans_ex = A&B	Zero (Reset Overflow flag)
6	OR	000101	ans_ex = A|B	Zero (Reset Overflow flag)
7	XOR	000110	ans_ex = A^B	Zero (Reset Overflow flag)
8	NOT	000111	ans_ex = ~B	Zero (Reset Overflow flag)
9	ADI	001000	ans_ex = A+B	Overflow, Zero
10	SBI	001001	ans_ex = A-B	Overflow, Zero
11	MVI	001010	ans_ex = B	Zero (Reset Overflow flag)
12	ANI	001100	ans_ex = A&B	Zero (Reset Overflow flag)
13	ORI	001101	ans_ex = A|B	Zero (Reset Overflow flag)
14	XRI	001110	ans_ex = A^B	Zero (Reset Overflow flag)
15	NTI	001111	ans_ex = ~B	Zero (Reset Overflow flag)
16	RET	010000	Hold previous ‘ans_ex’	Reset all flags
17	HLT	010001	Hold previous ‘ans_ex’	Reset all flags
18	LD	010100	ans_ex = A	Reset all flags
19	ST	010101	ans_ex = A	Reset all flags
20	IN	010110	ans_ex=data_in	Zero (Reset Overflow flag)

21	OUT	010111	Hold previous 'ans_ex' data_out = A	Reset all flags
22	JMP	011000	Hold previous 'ans_ex'	Reset all flags
23	LS	011001	ans_ex = A<<B	Zero (Reset Overflow flag)
24	RS	011010	ans_ex = A>>B	Zero (Reset Overflow flag)
25	RSA	011011	ans_ex = A>>>B	Zero (Reset Overflow flag)
26	JV	011100	Hold previous 'ans_ex'	Previous State
27	JNV	011101	Hold previous 'ans_ex'	Previous State
28	JZ	011110	Hold previous 'ans_ex'	Previous State
29	JNZ	011111	Hold previous 'ans_ex'	Previous State

Table 2.4.0 Opcodes & Operations

MOV	Move data from register to register
MVI	Move immediate data to register
LD	Move data from data memory to register
ST	Move data from register to data memory
IN	Move data from input to register
OUT	Move data from register to output

Table 2.4.1 Data Transfer Instructions

ADD	Add data of two registers
SUB	Subtract data of two registers
MUL	Multiply data of two registers
AND	Perform AND operation on the data of two registers
OR	Perform OR operation on the data of two registers
XOR	Perform XOR operation on the data of two registers
NOT	Perform NOT operation on the data of two registers
ADI	Add immediate number with register data
SBI	Subtract immediate number from register data
ANI	Perform AND operation between immediate number with the data register
ORI	Perform OR operation between immediate number with the data register
XRI	Perform XOR operation between immediate number with the data register
NTI	Perform NOT operation between immediate number with the data register
LS	Perform left shift operation on register data
RS	Perform right shift operation on register data
RSA	Perform arithmetic right shift operation on register data

Table 2.4.2 Arithmetic & Logical Instructions

RET	Return to program memory address after completion of interrupt sub-routine
HLT	Stop the execution of program
JMP	Unconditional jump to given program memory address location
JV	Jump to given program memory address location if overflow flag is 1
JNV	Jump to given program memory address location if overflow flag is 0
JZ	Jump to given program memory address location if zero flag is 1
JNZ	Jump to given program memory address location if zero flag is 0

Table 2.4.3 Conditional Instructions

V. Jump Control

Signals to JC Block	Input variable Name	No. of Bits	Explanation
Inputs	jmp_address_pm	16	Jump address from instruction (ins)
	current_address	16	Address of current instruction
	op	6	Opcode
	flag_ex	2	Flag from execution block
	interrupt	1	External interrupt signal (Non maskable)
	clk	1	Input clock signal
	reset	1	Reset for registers/FFs
Outputs	jmp_loc	16	Jump address for PC_IM block
	pc_mux_sel	1	Mux selection bit for PC_IM block

Table 2.5 Inputs & Outputs of the Block

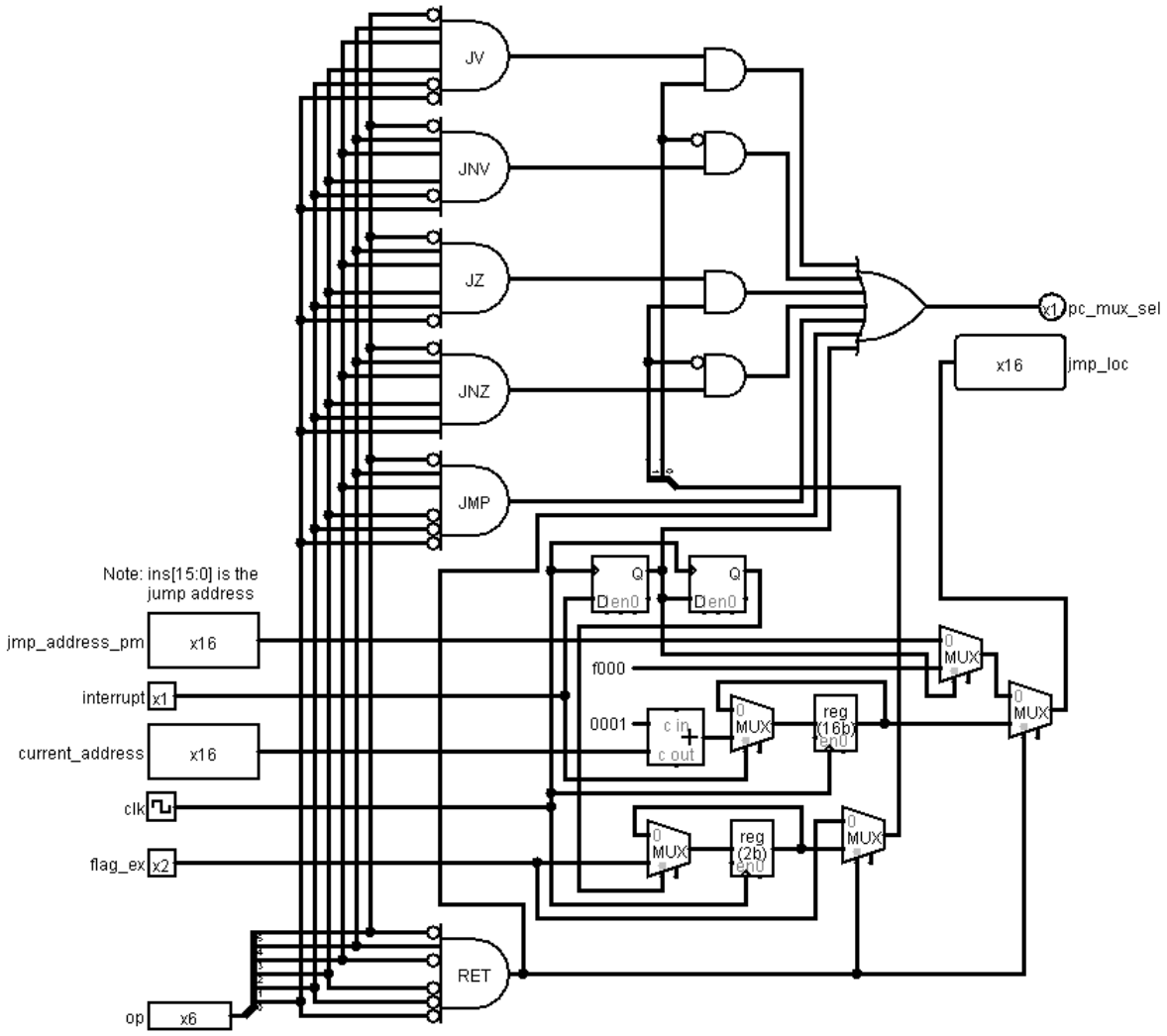

Figure 2.5 Jump Control Block

Block Diagram Description:

Jump control block generates jump location address (jmp_loc) and mux control signal (pc_mux_sel) for program memory block, based on jump conditions and value of interrupt. For conditional jumps (JC, JNC, JZ and JNZ), if condition is true then 'pc_mux_sel' value will be '1', else we check for unconditional jump (JMP), interrupt and return (RET) instructions, if any of these instruction/signal is true then also 'pc_mux_sel' value will be '1', for other cases 'pc_mux_sel' value will be '0'. When interrupt signal value is '1', we stores 'current_address' and 'flag_ex' in the system and program will jump to interrupt subroutine location (16'hF000). When interrupt subroutine is over then we provide RET instruction to continue our main program, for that we need to reproduce 'current_address' and 'flag_ex' which was previously stored in system.

VI. Stall Control

Signals to SC Block	Input variable Name	No. of Bits	Explanation
Inputs	op	6	Opcode
	clk	1	Input clock signal
	reset	1	Reset for FFs
Outputs	stall	1	Stall signal for PC_IM block
	stall_pm	1	Delayed stall signal for PC_IM block

Table 2.6 Inputs & Outputs of the Block

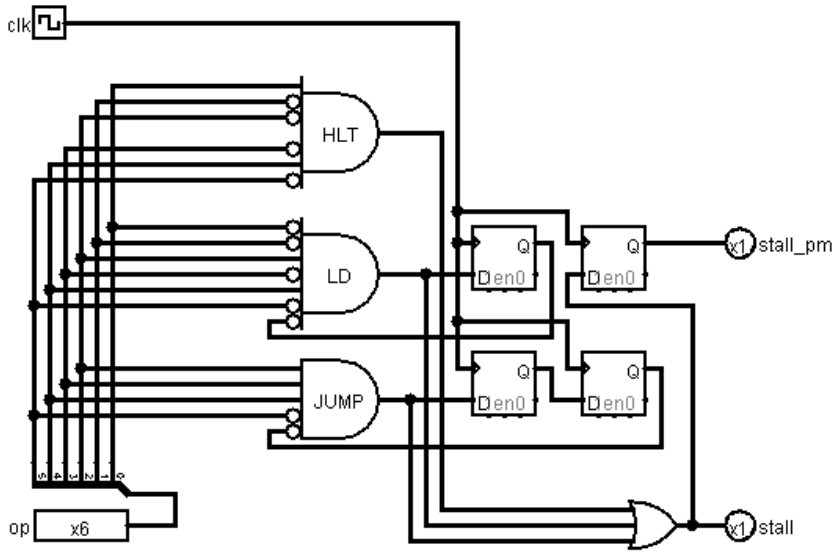

Figure 2.6 Stall Control Block

Block Diagram Description:

Some instructions like 'Load' and 'Conditional Jump' are creating control hazard in pipeline processor. To overcome this problem, we need to stall pipeline for one or two clock cycles. Stall Control block decides when to stall a pipeline. Also, this block will stall pipeline forever when 'HLT' instruction will execute.

VII. Data Memory

Signals to DM Block	Input variable Name	No. of Bits	Explanation
Inputs	ans_ex	16	Answer from Execution block (used as address)
	DM_data	16	Input data for Data Memory
	mem_rw_ex	1	Memory read/write signal
	mem_en_ex	1	Memory enable signal
	mem_mux_sel_dm	1	Mux selection bit for ans_dm
	reset	1	Reset signal for register
	clk	1	Input clock signal
Outputs	ans_dm	16	First operand for Execution block

Table 2.7 Inputs & Outputs of the Block

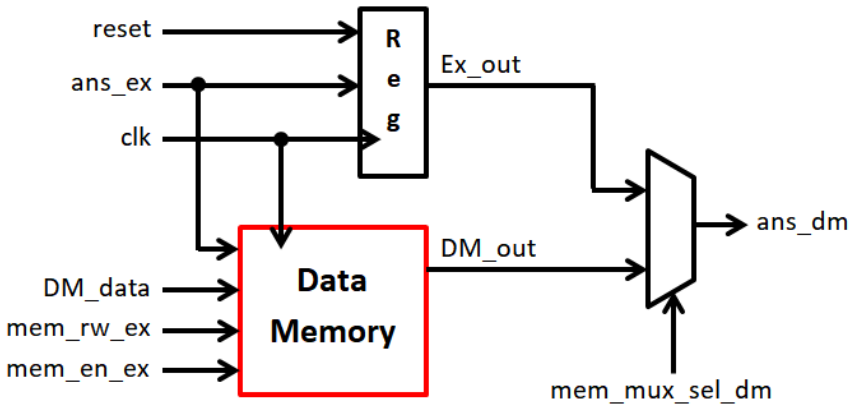

Figure 2.7 Data Memory Block

Block Diagram Description:

Data Memory is designed using IP core. It's a 65536x16 RAM with R/W and enable signal. Data read or write in Data Memory is possible only if 'mem_en_ex' signal is 1. If 'mem_rw_ex' is 0 data read operation will perform else data write. Data written in the Data Memory should be provided to 'DM_data'; and 'ans_ex' is used as an address. 2x1 Mux is used to select between

execution block answer (Ex_out) or Data Memory's output (DM_out). When 'mem_mux_sel_dm' is 1, it will select 'DM_out' as an 'ans_dm' else it will select 'Ex_out' as an 'ans_dm'.

VIII. Write Back

Signals to WB Block	Input variable Name	No. of Bits	Explanation
Inputs	ans_dm	16	Output of Data Memory block
	clk	1	Input clock signal
	reset	1	Reset signal for register
Output	ans_wb	16	Output of Write Back block

Table 2.8 Inputs & Outputs of the Block

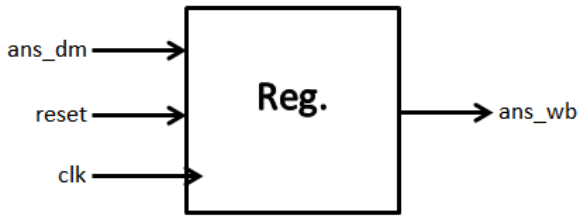

Figure 2.8 Write Back Stage

Block Diagram Description:

Write Back block is used to delay 'ans_dm' for one clock cycle. This block helps to resolve read after write hazard in the IDOF (Instruction Decode and Operand Fetch) stage.

4. Executed Instruction Set

Ins No.	Operation	Load / Store Address	Address of operand A	Address of operand B	Immediate Value / Jump address	Operation Description
1	MOVI	$0	$0	$0	0	Move Immediate 0 to $0
2	MOVI	$3	$0	$0	3	Move Immediate 3 to $3
3	MOVI	$5	$0	$0	5	Move Immediate 5 to $5
4	MOVI	$7	$0	$0	9	Move Immediate 9 to $7
5	ADD	$1	$0	$0	0	Add $0 + $0 -> $1
6	LD	$2	$3	$0	0	Load from $3 -> $2
7	LD	$4	$5	$0	0	Load from $5 -> $2
8	MUL	$2	$2	$4	0	Multiply $2 * $4 -> $4
9	ADD	$1	$1	$2	0	Add $1 + $2 -> $1
10	ADDI	$3	$3	$0	4	Add Immediate $3 + 4 -> $3
11	ADDI	$5	$5	$0	4	Add Immediate $5 + 4 -> $5
12	SUBI	$7	$7	$0	1	Sub Immediate $7 – 1 -> $7
13	JNZ	$0	$0	$0	5	Jump if $7 != 0
14	OUT	$0	$1	$0	0	Output
15	ST	$0	$1	$0	0	Store $1 -> $0
16	HLT	$0	$0	$0	0	Halt

Table 3.1 Instruction Set Architecture for the given Task

The dot product of the two student ID's having 9 is performed using the above instruction set architecture. For that purpose, the program memory has been uploaded with the instruction set and the data memory has been loaded with the two student ID's. Example using the ID's is shown below

Student ID-1 :- 014512836
Student ID-2 :- 014532908

The dot product of the above two ID's is shown below

(0 x 0)+(1 x 1)+(4 x 4)+(5 x 5)+(1 x 3)+(2 x 2)+(8 x 9)+(3 x 0)+(6 x 8) = 169

5. Behavioral Simulation

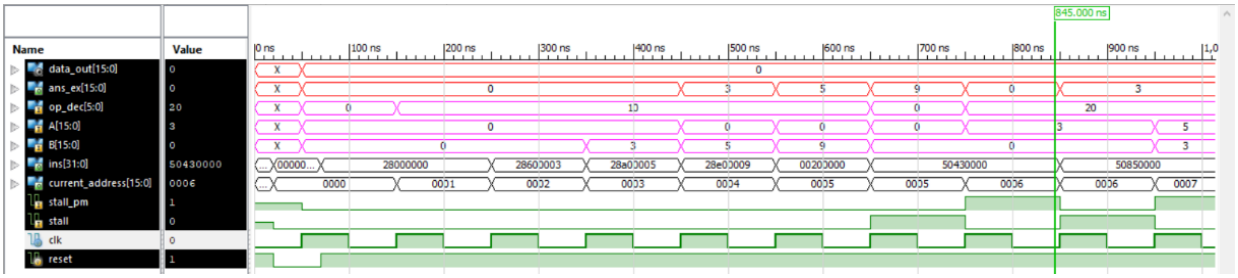

Figure 4.1 Simulation Results

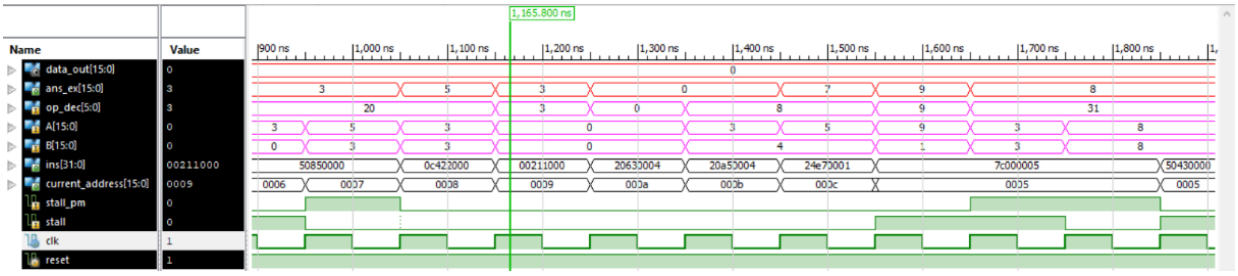

Figure 4.2 Simulation Results

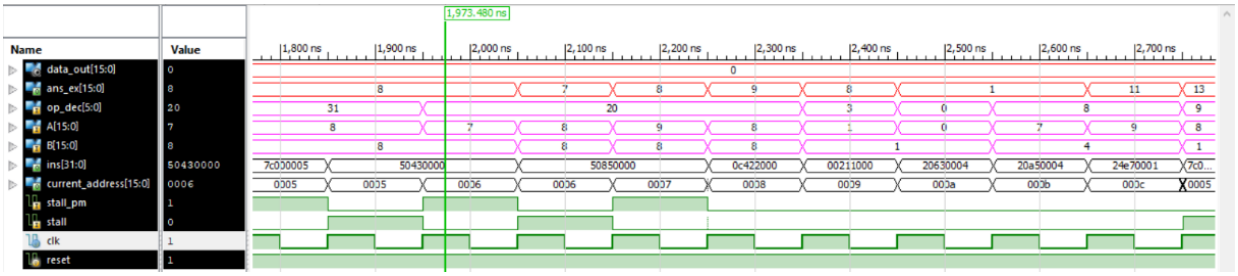

Figure 4.3 Simulation Results

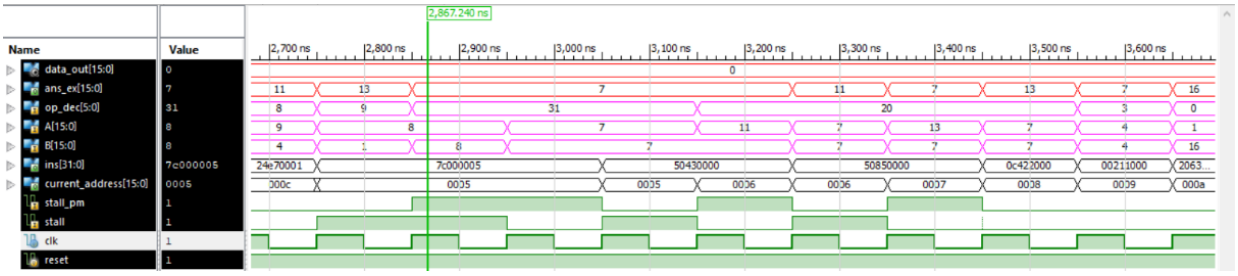

Figure 4.4 Simulation Results

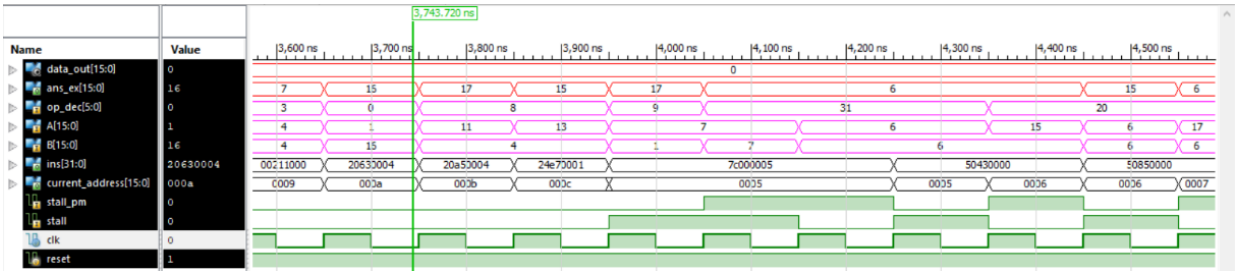

Figure 4.5 Simulation Results

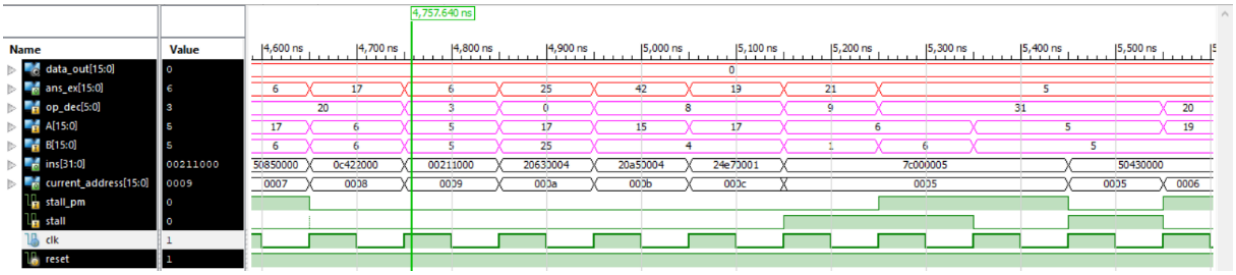

Figure 4.6 Simulation Results

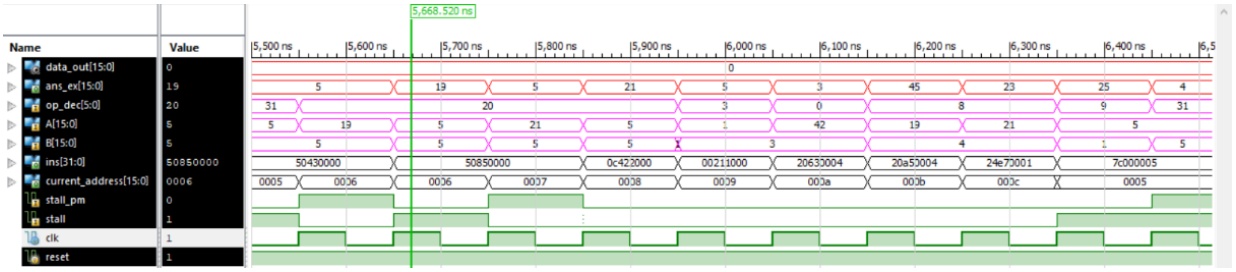

Figure 4.7 Simulation Results

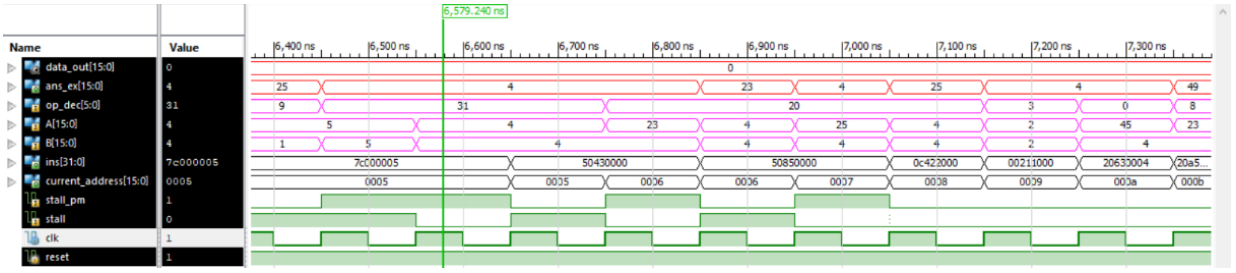

Figure 4.8 Simulation Results

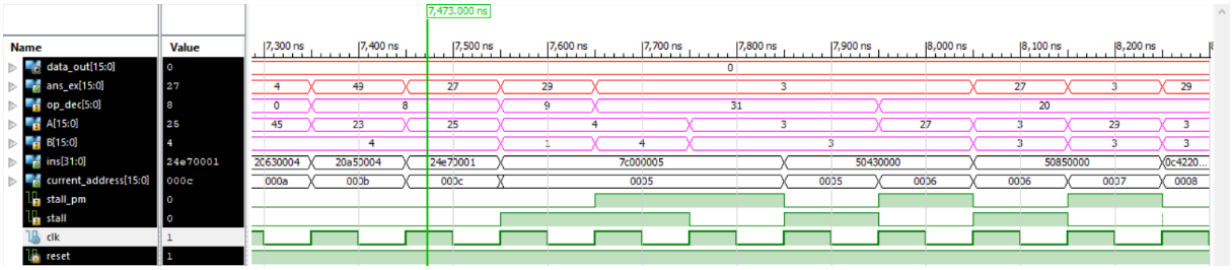

Figure 4.9 Simulation Results

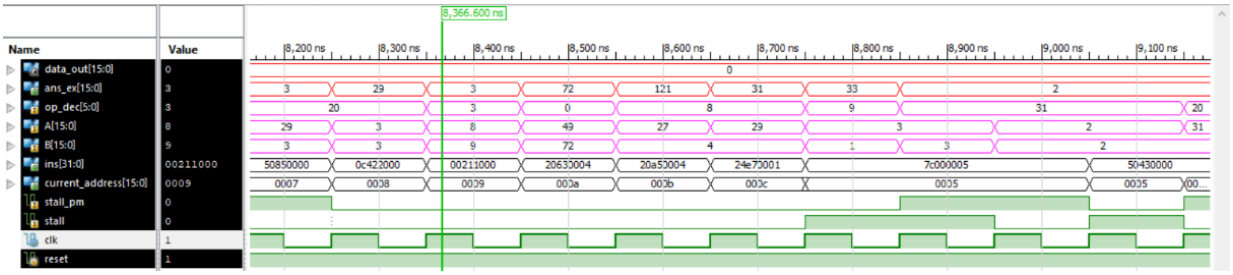

Figure 4.10 Simulation Results

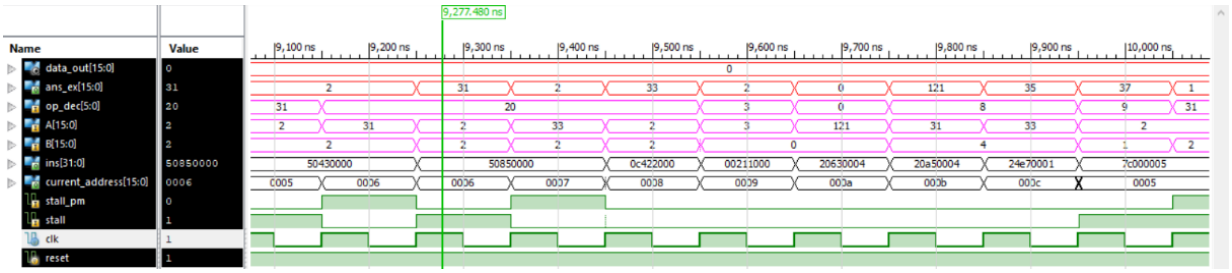

Figure 4.11 Simulation Results

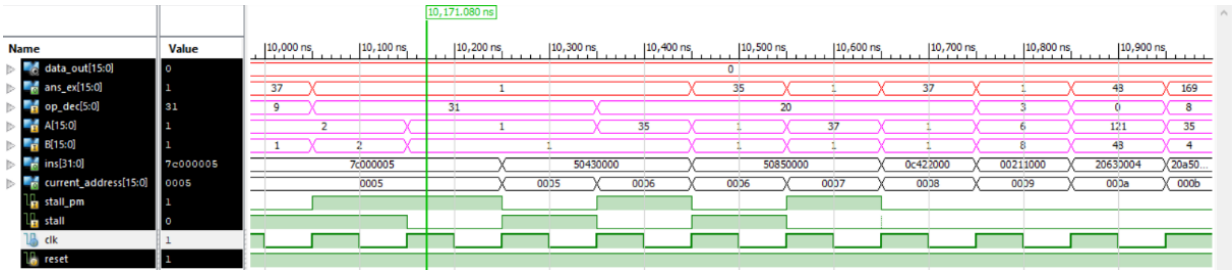

Figure 4.12 Simulation Results

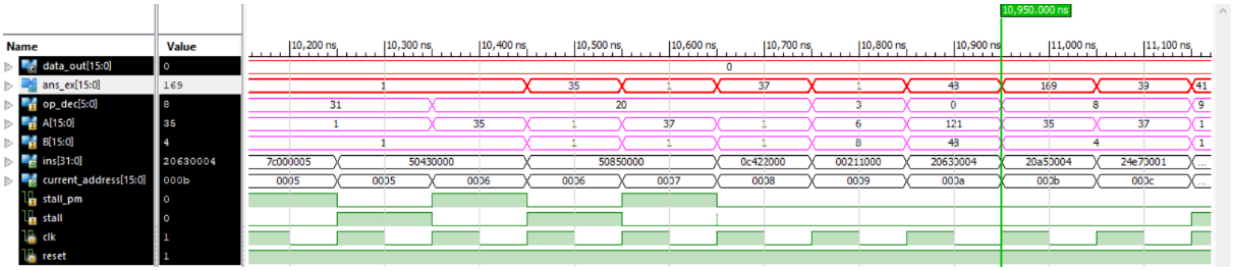

Figure 4.13 Simulation Results

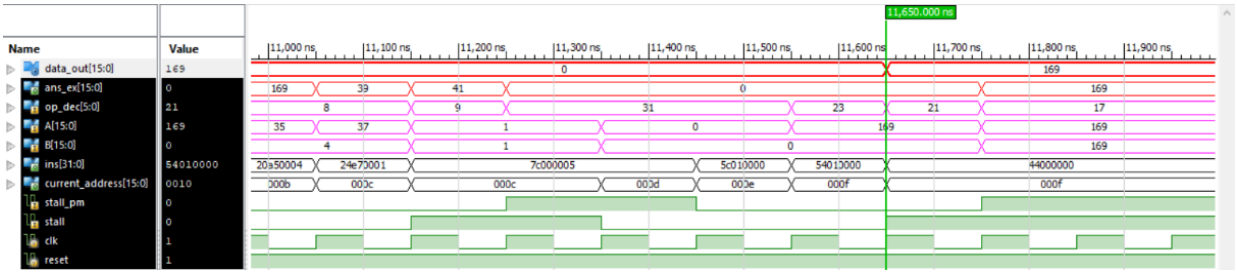

Figure 4.14 Simulation Results

The Figures 4.1 to 4.13 show the complete simulation of the executed instruction mentioned in the above section III for two student ID's (014512836 & 014532908). As shown in the Figure 4.13 the **ans_ex = 169 @ 10950ns i.e. 10.95us** after that the dot product is pushed out to the external world using OUT operation and that change takes place **data_out = 169 @11650ns i.e. 11.65us** as shown in Figure 4.14. In all the above figures the **data_out** is the output of 16-bits from the processor to the external world. Variable **op_dec** is the operational code of 6-bits for ALU. Variable **A** and **B** are the two operands of 16-bits for the ALU. Variable **ins** is 32-bits instruction shown in hex format. Variable **current_address** is the 16-bits address value for the program memory to fetch the instructions. Variables **Stall** and **Stall_pm** are the 1-bit signals generated by stall control block to stall the current_address and ins to avoid control hazards.

6. Synthesis Results

The 5-stage 32-bits MIPS pipelined processor has been coded in Verilog-HDL using Xilinx-ISE 14.7. The target device is nexys4-DDR Artix7 FPGA which has clock frequency of 100Mhz. Show below is the resource utilization done after performing synthesis on the targeted device.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	243	126800	0%
Number of Slice LUTs	714	63400	1%
Number of fully used LUT-FF pairs	129	828	15%
Number of bonded IOBs	35	210	16%
Number of Block RAM/FIFO	88	135	65%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48E1s	1	240	0%

Figure 5.1 Resource Utilization Summary

7. Block Diagram of Processor

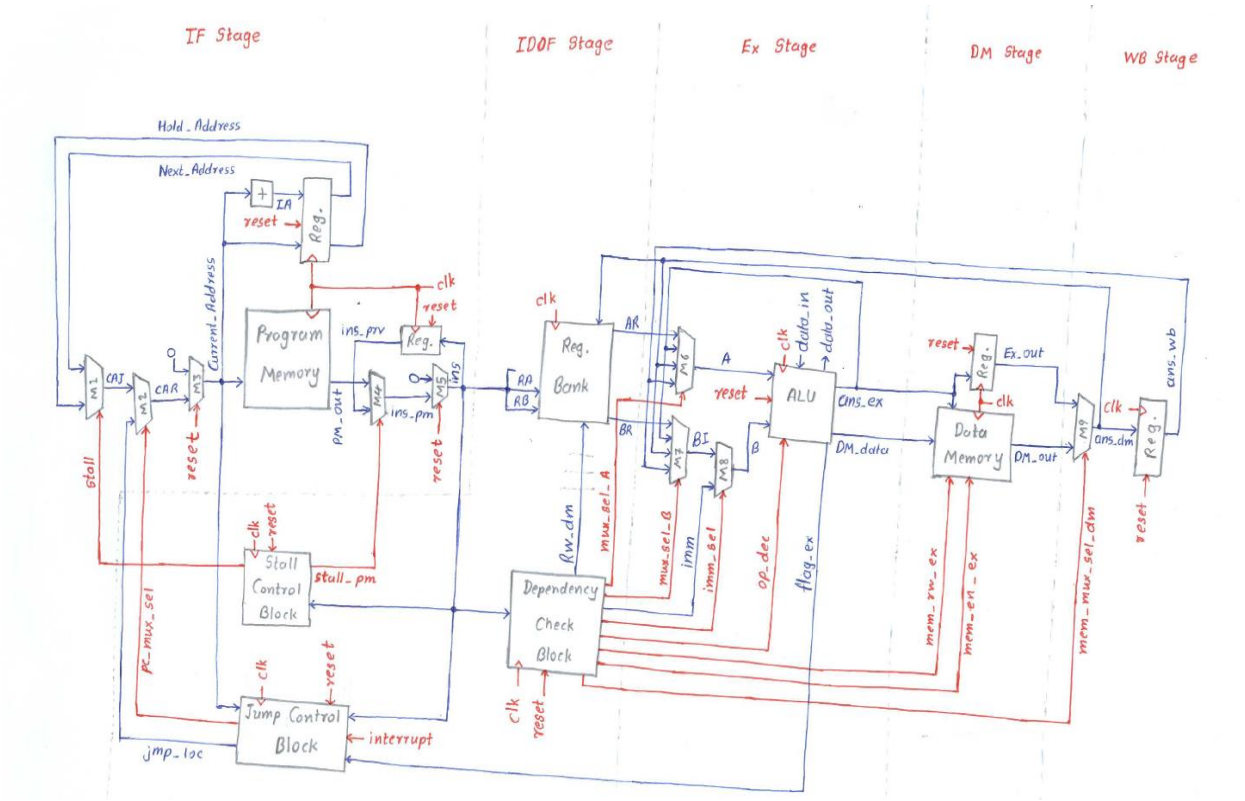

Figure 6.2 5-stage pipeline processor block diagram

8. RTL Schematic of MIPS processor

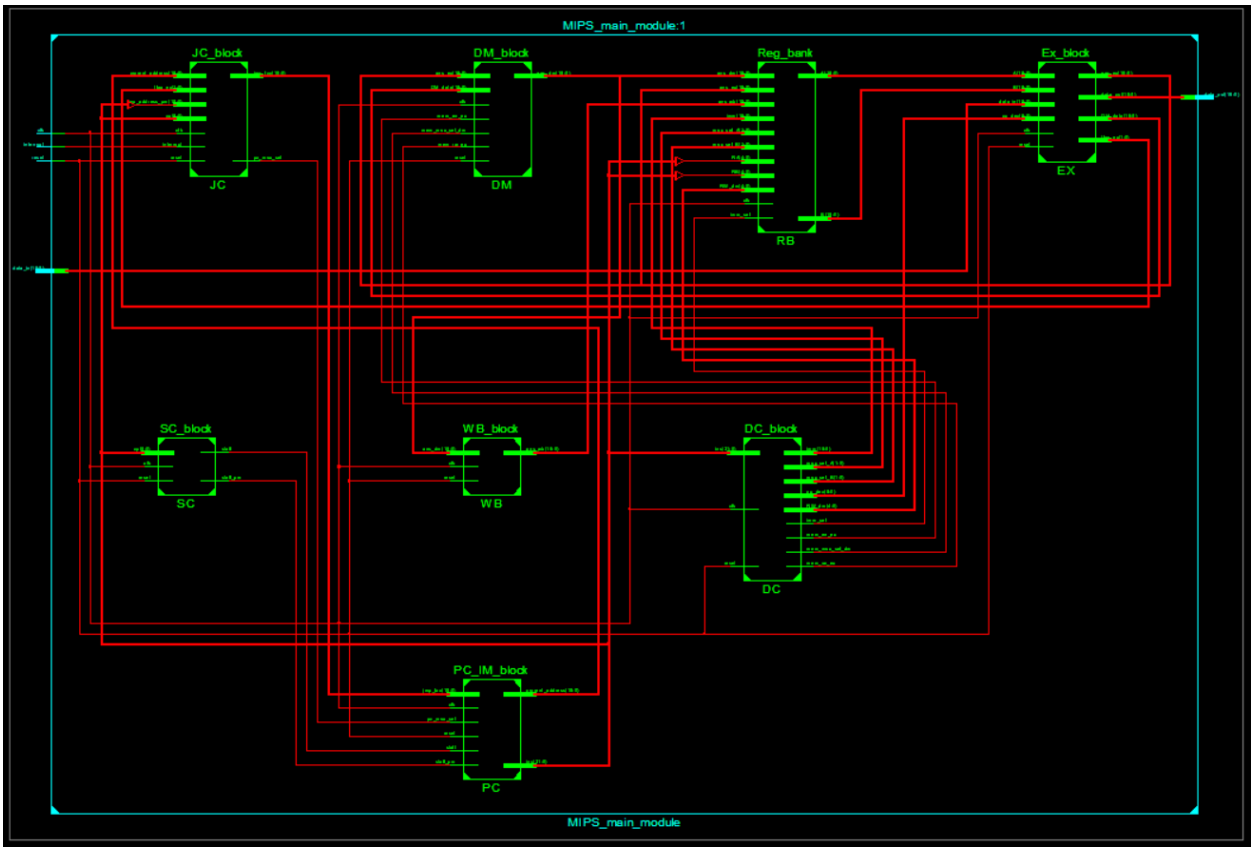

Figure 7.1 RTL schematic generate by Xilinx -ISE 14.7

9. Verilog-Codes & Test Bench

Each and Every Module of the processor has been verified individually using individual testbench. All the Verilog modules of the processor are in the folder "**1. 32-bits MIPS Processor Verilog Codes**". All the test benches are in the folder "2. Test Bench". The instruction set used in the program memory IP core is the folder "**3. Program Memory File**". The data file used to fill data memory with student ID's is in the folder "**4. Data Memory File**".

10. Important Note

We are not following the coding guidelines given by the Professor about adding the last two digits of the student ID to every variable in the code. It's been approved by professor. Because the codes have many variable and combined code is above 500 lines.