ABOUT THE AUTHOR

I am Saiteja and I am from Hyderabad, India. I completed my bachelors in Electronics and Communication Engineering from Gokaraju Rangaraju Institute of Engineering and Technology with a CGPA of 9.97/10 in the year 2019. I received a Gold Medal for the academic excellence from my university. Right after that, I got an opportunity to work for Tata Consultancy Services as a Systems Engineer and have worked for 3 years until July 2022. I speak English, Hindi, and Telugu where the last of these is my first language. My interests include but not limited to Cricket, Table-Tennis and learning new languages. Currently I am pursuing my master's in Computer Engineering at San Jose State University.

TABLE OF CONTENTS

LIST OF FIGURES \ TABLES

INTRODUCTION:

This activity is to gain hands-on processor design experience by reviewing RTL Verilog design code for the initial version of the single-cycle MIPS processor discussed in class. Apart from that, we learn the basic technique for functionally verifying a processor.

MY GROUP:

Name: Student_Team 6
Members: Tirumala Saiteja Goruganthu (016707210), Harish Marepalli (016707314)

GIVEN TASK:

The task is to carefully study the source code you obtained from the archive, then based on the source, draw the block diagrams of the following , for the initial design of the MIPS processor:

a) Datapath with microarchitecture details (not including memories).
b) Control unit with microarchitecture details.
c) Instruction memory and data memory.
d) Processor core (show interconnection between the datapath and its control unit, datapath internal microarchitecture is not required).
e) Complete processor (show interconnections between the processor core and its memories).

Another task is to use the resources included in the archive, build the DUT (the initial design of the single-cycle MIPS) and its test bench, then functionally verifying the MIPS processor based on the given sample program.

STEPS TAKEN TO COMPLETE THE TASK:

a. Spent time to understand the question carefully.
b. Noted down all the high level as well as low level datapath, controlpath, instruction memory, data memory, processor core, complete processor signals using the Verilog code given in the archive.
c. Drawn the datapath block diagram with the above notes and using Visio to build the building blocks.
d. Similarly drawn the controlpath, instruction memory, data memory, processor core, complete processor block diagrams with the above notes and using Visio to build the building blocks.
e. After completing the task1, started creating a new project in Vivado Xilinx tool.
f. Imported the given sources as well as the simulation testbench from the archive into the project.
g. Altered the *memfile.dat* location by putting the file in current working directory. This file is important since it has all the instruction machine codes that are needed for execution.
h. After the project setup is done, clicked on "*Run Simulation*" button to execute the test bench.
i. Finally, output waveforms are simulated and observed in the Vivado tool.

DISCUSSION SECTION:

1) Datapath Explanation:

From the Verilog file "*datapath.v*", we can observe that six leaf-level modules are used to instantiate the whole datapath (albeit some modules may be used multiple times). In this section, we will try to visually (in the form of tables) show the input-output structure of each leaf-level module and then we traverse our way back up to the datapath module.

a. The D-Register module named "*dreg*" contains three inputs and one output. The main logic here is that for every positive clock edge and positive reset edge we will start the execution. If the reset (*rst*) signal is asserted, then assign logic '*0*' to the output (*q*), else assign the input (*d*) to the output (*q*). The following table (*Table5.1*) will illustrate the Signal-Description relationship.

Signal Name	**Description (Input / Output)**
clk	Clock signal (Input)
rst	Reset Signal (Input)
d[31:0]	Input Signal (Input)
q[31:0]	Output Signal (Output)

Table5.1 – Signal-Description table for the D-Register

b. The 32-bit adder module named "*adder*" contains two inputs and one output. The main logic here is written in behavioral way where just a single instruction (*y* = *a*+*b*) is enough to realize a 32-bit adder. The following table (*Table5.2*) will illustrate the Signal-Description relationship.

Signal Name	**Description (Input / Output)**
a[31:0]	Operand 1 (Input)
b[31:0]	Operand 2 (Input)
y[31:0]	Result (Output)

Table5.2 – Signal-Description table for the 32-bit adder

c. The 2x1 multiplexer module named "*mux2*" contains two inputs, one output and one selection line to select the required input line. The main logic here is written in behavioral way using the ternary operator (*y* = *sel ? b : a*) is enough to realize a 2x1 multiplexer. The following table (*Table5.3*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[7:0]	Operand at 0 port (Input)
b[7:0]	Operand at 1 port (Input)
sel	Selection line (Input)
y[7:0]	Output Signal (Output)

Table5.3 – Signal-Description table for the 2x1 multiplexer

d. The sign extension module named "*signext*" contains one input and one output. The main logic here is to get the MSB of the input, duplicate this bit 16 times and append the new 16-bit string to the already existing 16-bit input (*a*) to form a 32-bit number as output (*y*). The following table (*Table5.4*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[15:0]	16-bit Input (Input)
y[31:0]	32-bit Output (Output)

Table5.4 – Signal-Description table for the Sign Extension Unit

e. The Register file module named "*regfile*" contains seven inputs and three outputs. Each register in this register file can hold 32-bit data. We realize this using a two-dimensional array. Initialize all the register values to zero and initialize "*sp*" value to 32-bit "b'*100*" value. The main logic here is that for every positive edge of the clock, if write enable signal (*we*) is enabled then write the data (*wd*) into the provided write address (*wa*). After that, output all three read-data ports (*rd1*, *rd2*, *rd3*) by assigning them with the content at addresses (*ra1*, *ra2*, *ra3*). The following table (*Table5.5*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
clk	Clock signal (Input)
we	Write Enable (Input)
ra1[4:0]	Read Address 1 (Input)
ra2[4:0]	Read Address 2 (Input)
ra3[4:0]	Read Address 3 (Input)
wa[4:0]	Write Address (Input)
wd[31:0]	Write Data (Input)
rd1[31:0]	Read Data 1 (Input)
rd2[31:0]	Read Data 2 (Input)
rd3[31:0]	Read Data 3 (Input)

Table5.5 – Signal-Description table for the Register File

f. The Arithmetic and Logical Unit (ALU) module named "*alu*" has three inputs and two outputs. The main logic here is that depending on the "*op*" value, the operation is determined. So, we use switch case to realize the ALU. Apart from that, a "*zero*" flag is set as output to the ALU which will be de-asserted only if the ALU result is logic low. The following table (*Table5.6*) will illustrate the Signal-Description relationship.

Signal Name	**Description (Input / Output)**
a[31:0]	Operand 1 (Input)
b[31:0]	Operand 2 (Input)
op[2:0]	Operation Select (Input)
zero	Zero Flag (Output)
y[31:0]	Result (Output)

Table5.6 – Signal-Description table for the ALU

The following Truth-Table (*Table5.7*) observed from the given Verilog code is as follows:

op value	**Corresponding Operation**
000	AND
001	OR
010	ADD
110	SUB
111	SLT

Table5.7 – Truth-Table observed in alu.v file

g. Finally, the whole datapath is realized in a module named "*datapath.v*." We use one or more of the leaf-level to achieve the required datapath. The following table (*Table5.8*) will illustrate the Signal-Description relationship.

The D-Register (*pc_reg*) is used to store the program counter value with every positive edge of the clock cycle.

The adder (*pc_plus_4*) is used to add number 4 to the current program counter to point to the next instruction.

The adder (*pc_plus_br*) is used to add pc_plus_4 and the branch address (calculated by multiplying the signImm value with 4).

The multiplexer (*pc_src_mux*) is used to select either next instruction program counter or the new branch target address.

The multiplexer (*pc_jmp_mux*) is used to select either next instruction program counter or the new jump target address.

The multiplexer (*rf_wa_mux*) is used to select either the R_d destination register or R_t destination register depending on the type of instruction.

The Register File (*rf*) is used to as a 32 32-bit register container used for processor operations.

The Sign Extension Unit (*se*) is used to extend the 16-bit input using the MSB and replicating the MSB 16 more times before appending to the original 16-bit input to form a 32-bit output.

The multiplexer (*alu_pb_mux*) is used to select either the data from the register file or the sign extension unit depending on the type of instruction.

The Alu (*alu*) is used to perform operations such as add, sub, and, or and slt where these operations are controlled by the op[2:0] bits.

The multiplexer (*rf_wd_mux*) is used to select either the data from the memory or the alu output depending on the type of instruction.

Signal Name	**Description (Input / Output)**
clk	Clock Signal (Input)
rst	Reset (Input)
branch	Branch Control (Input)
jump	Jump Control (Input)
reg_dst	Destination Register Select (Input)
we_reg	Write Enable for Register File (Input)
alu_src	ALU Source Steering (Input)
dm2reg	Data Memory to RF (Input)
alu_ctrl[2:0]	ALU Control (Input)
ra3[4:0]	Read Address Port 3 (Input)
instr[31:0]	32-bit Instruction (Input)
rd_dm[31:0]	Read data from Data memory (Input)
pc_current[31:0]	Current Program Counter (Output)
alu_out[31:0]	ALU Output (Output)
wd_dm[31:0]	Write data to Data memory (Output)
rd3[31:0]	Read Data Port 3 (Output)

Table5.8 – Signal-Description table for the Datapath

Below Figure (*Figure 5.9*) shows the Datapath Block Diagram for the Single-cycle MIPS CPU.

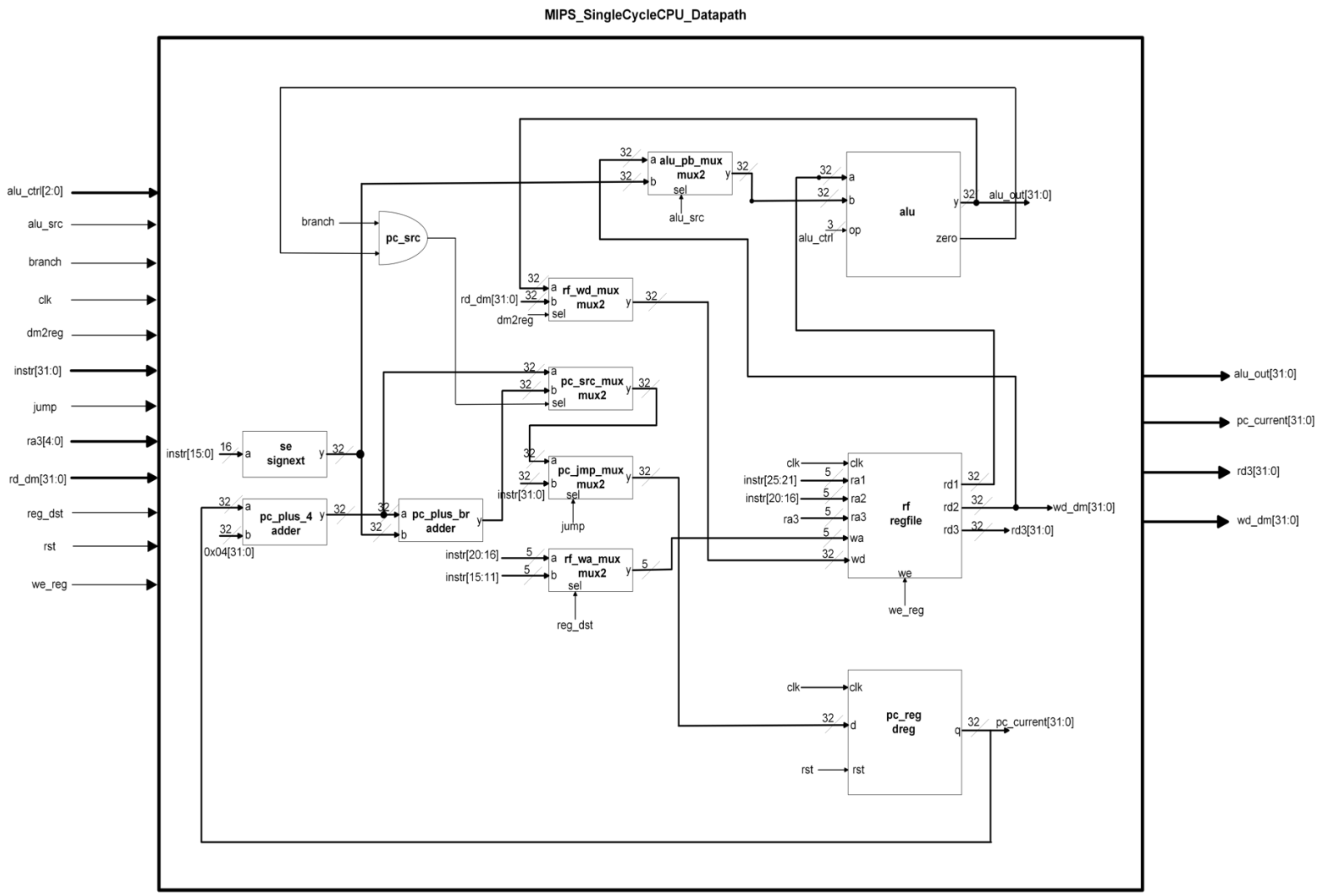


Figure5.9 – Datapath of the Single-Cycle MIPS Processor

2) Control Path Explanation:

From the Verilog file "*controlunit.v*", it is observed that the whole control unit is divided into two categories.

a. A main decoder to generate all the control signals such as mux selects, branch and jump using the opcode part of the instruction machine code and it also outputs a 2-bit *alu_op* signal which controls the second *auxilary* or ALU decoder. Depending upon the opcode bits, the corresponding operation is selected. The following table (*Table 5.10*) will illustrate the Signal-Description relationship.

Signal Name	**Description (Input / Output)**
opcode[5:0]	Opcode from the Machine Code (Input)
branch	Branch Control (Output)
jump	Jump Control (Output)
reg_dst	Destination Register Select (Output)
we_reg	Write Enable Register (Output)
alu_src	ALU Source Steering (Output)
we_dm	Write Enable to Data Memory (Output)
dm2reg	Data Memory to RF (Output)
alu_op[1:0]	2-bit ALU Result (Output)

Table5.10 – Signal-Description table for the Main Decoder

b. An Auxiliary or ALU decoder is another module which takes two inputs and sends out one output. Depending on the *funct* field and the *alu_op* signal, the corresponding operation is realized. If *alu_op* is either 00 or 01, then the operation is add or subtract, else, the *funct* field is used to realize a particular operation. The following table (*Table 5.11*) will illustrate the Signal-Description relationship.

Signal Name	**Description (Input / Output)**
alu_op[1:0]	ALU Input from Main Decoder
funct[5:0]	Function bits from the Machine Code
alu_ctrl[2:0]	ALU Control bits to ALU Module

Table5.11 – Signal-Description table for the Auxilary Decoder

c. Finally, the whole control unit is designed using the above two modules. The following figure (*Figure 5.12*) show the control unit block diagram for Single-cycle MIPS CPU.

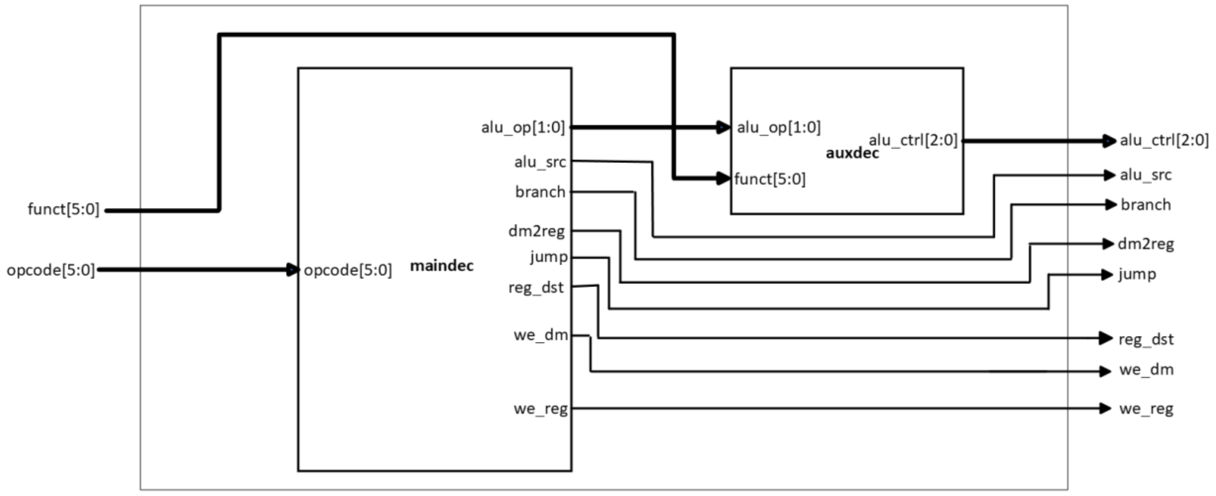


Figure5.12 – Control Unit of the Single-cycle MIPS Processor

3) Memory Explanation:

By observing the Verilog files related to memory unit, we have two memory partitions. They are instruction memory and data memory.

a. The instruction memory module named "*imem.v*" consists of one input for 5-bit address and one output for 32-bit data. The instruction memory is realized using a two-dimensional array with 64 rows and 32 columns. A separate "*memfile.dat*" is given in the assignment archive which consists of series of machine codes. When this module is instantiated, the machine codes from this file will be read into the above two-dimensional array. The following figure (*Figure5.13*) shows the block diagram of the instruction memory.

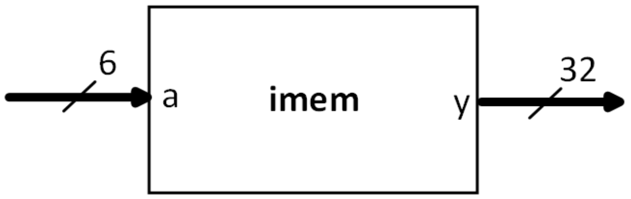


Figure5.13 – Block Diagram of the Instruction Memory

b. The data memory module named "*dmem.v*" consists of four inputs and one output. The data memory is realized using a two-dimensional array with 64 rows and 32 columns. When this module is instantiated, all values in the array are initialized to "*FFFFFFFF*." Now at every positive edge of the clock (*clk*), if the write enable (*we*) signal is asserted, then the data will be written into the array at the given address (*a*). For retrieval purposes, the array will be retrieved with address (*a*) and the corresponding data will be fetched and sent as output (*q*). The following figure (*Figure5.14*) shows the block diagram of the data memory.

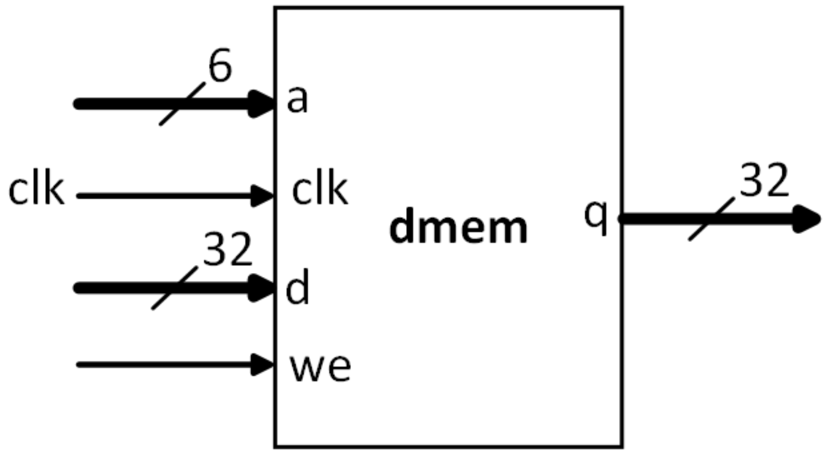


Figure5.14 – Block Diagram of the Data Memory

4) Processor Core Explanation:

By observing the Verilog files for the top-level module for datapath and controlpath, we see that there is another module named "*mips.v*" which instantiates both data and control paths by giving the required signals. It has five inputs and five outputs. The following figure (*Figure5.15*) shows the block diagram of the Processor Core.

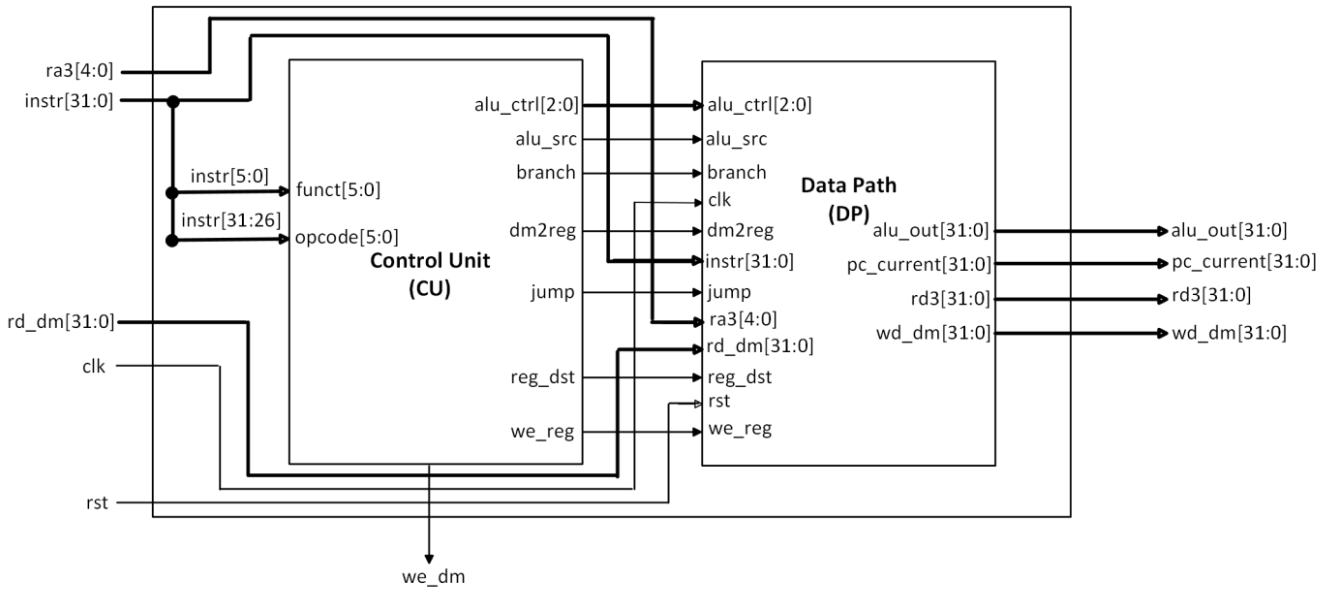


Figure5.15 – Block Diagram of the Processor Core

5) Complete Processor Explanation:

The previous Verilog file (*mips.v*) instantiates only datapath and controlpath modules, but the file (*mips_top.v*) instantiates the mips.v module, instruction memory and data memory. The following figure (*Figure5.16*) shows the block diagram of the Complete Processor.

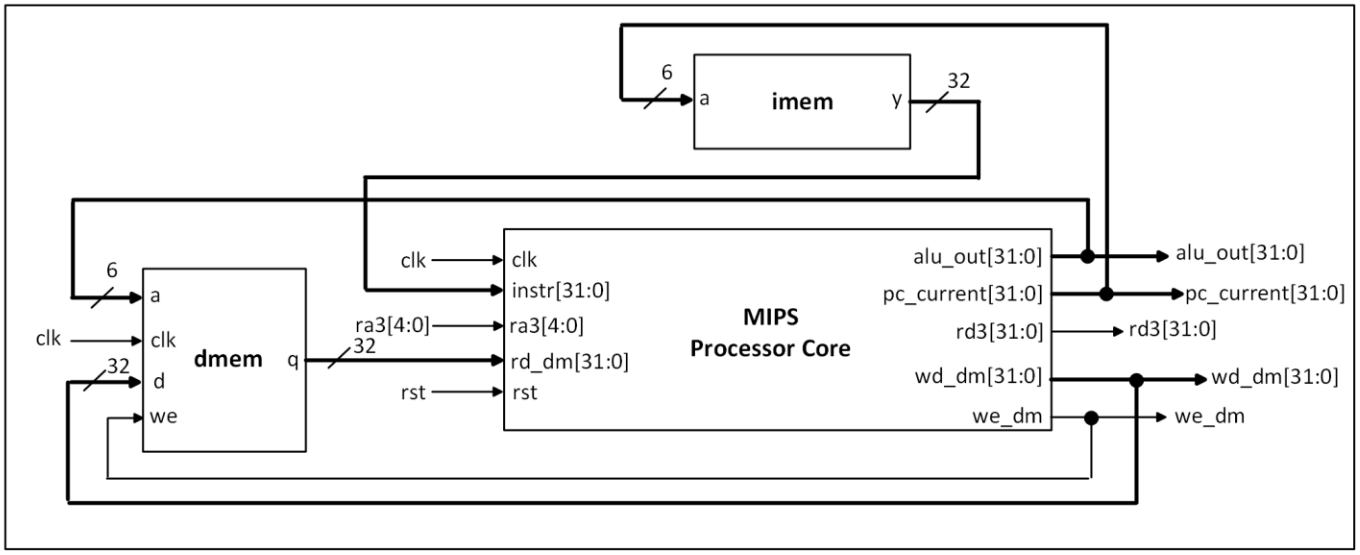


Figure5.16 – Block Diagram of the Complete Processor

6) Simulating in Vivado:

A test-bench file has already been given in the assignment archive which has been imported into the simulation sources of the Vivado Xilinx tool. After that, when the "*Run Simulation*" button is clicked, the waveforms will appear in the tool. The following figures (*Figure5.17* and *Figure5.18*) constitute the waveforms.

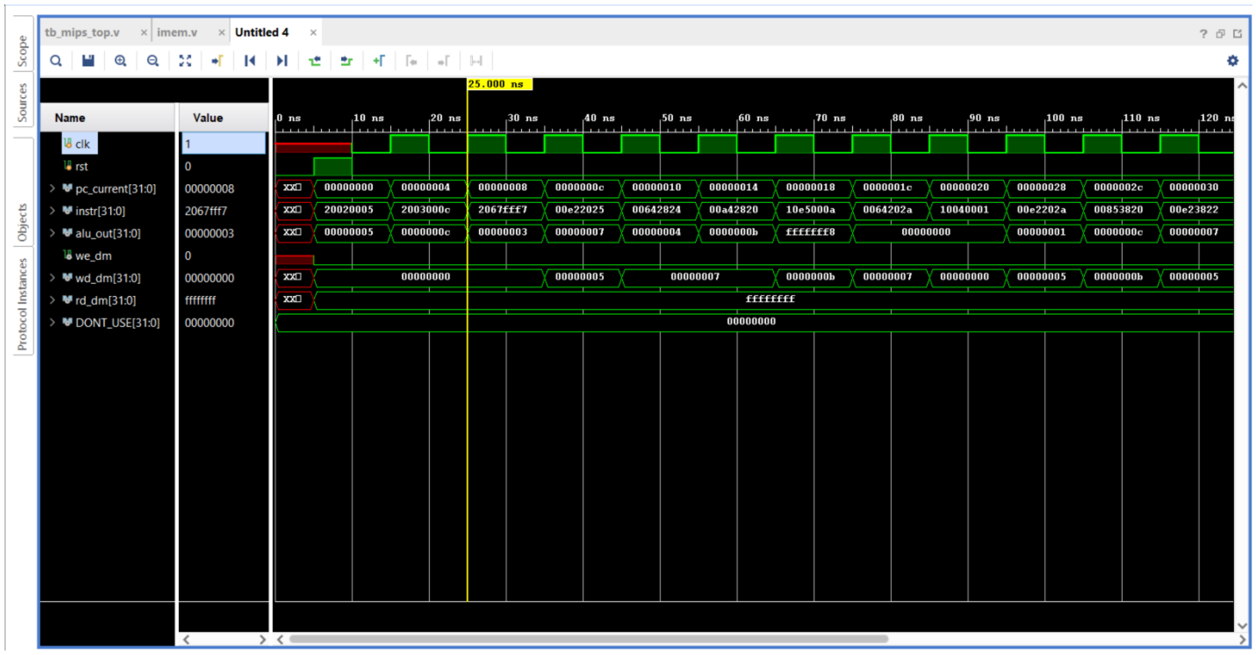

tb_mips_top.v
imem.v
Untitled 4
Scope
Sources
Objects
Protocol Instances
25.000 ns
Name
Value
clk 1
rst 0
pc_current[31:0] 00000008
instr[31:0] 2067fff7
alu_out[31:0] 00000003
we_dm 0
wd_dm[31:0] 00000000
rd_dm[31:0] ffffffff
DONT_USE[31:0] 00000000
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000028 0000002c 00000030
20020005 2003000c 2067fff7 00e22025 00642824 00a42820 10e5000a 0064202a 10040001 00e2202a 00853820 00e23822
00000005 0000000c 00000003 00000007 00000004 0000000b fffffff8 00000000 00000001 0000000c 00000007
00000000 00000005 00000007 0000000b 00000007 00000000 00000005 0000000b 00000005
ffffffff
00000000

Figure5.17 – Waveform after verifying the Single-cycle MIPS Processor Functionality.

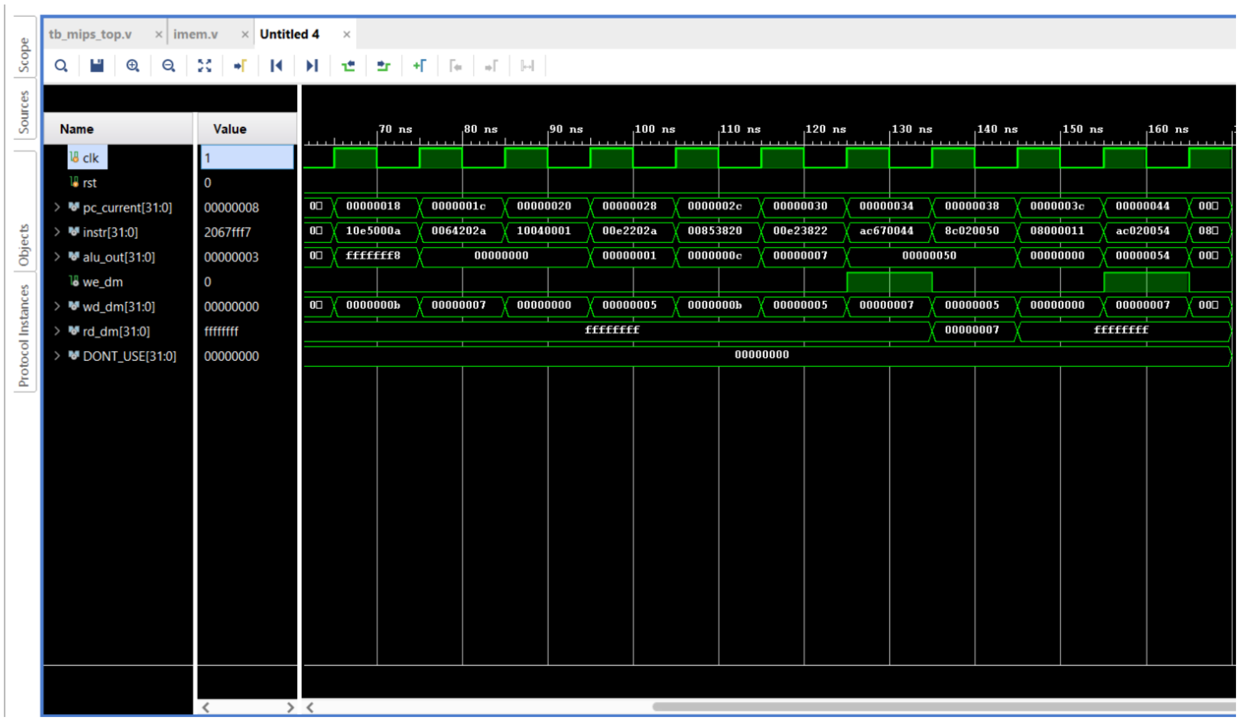

tb_mips_top.v
imem.v
Untitled 4
Scope
Sources
Objects
Protocol Instances
Name
Value
clk 1
rst 0
pc_current[31:0] 00000008
instr[31:0] 2067fff7
alu_out[31:0] 00000003
we_dm 0
wd_dm[31:0] 00000000
rd_dm[31:0] ffffffff
DONT_USE[31:0] 00000000
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
130 ns
140 ns
150 ns
160 ns
00000018 0000001c 00000020 00000028 0000002c 00000030 00000034 00000038 0000003c 00000044
10e5000a 0064202a 10040001 00e2202a 00853820 00e23822 ac670044 8c020050 08000011 ac020054
fffffff8 00000000 00000001 0000000c 00000007 00000050 00000000 00000054
0000000b 00000007 00000000 00000005 0000000b 00000005 00000007 00000005 00000000 00000007
ffffffff 00000007 ffffffff
00000000

Figure5.18 – Waveform after verifying the Single-cycle MIPS Processor Functionality.

COLLABORATION SECTION:

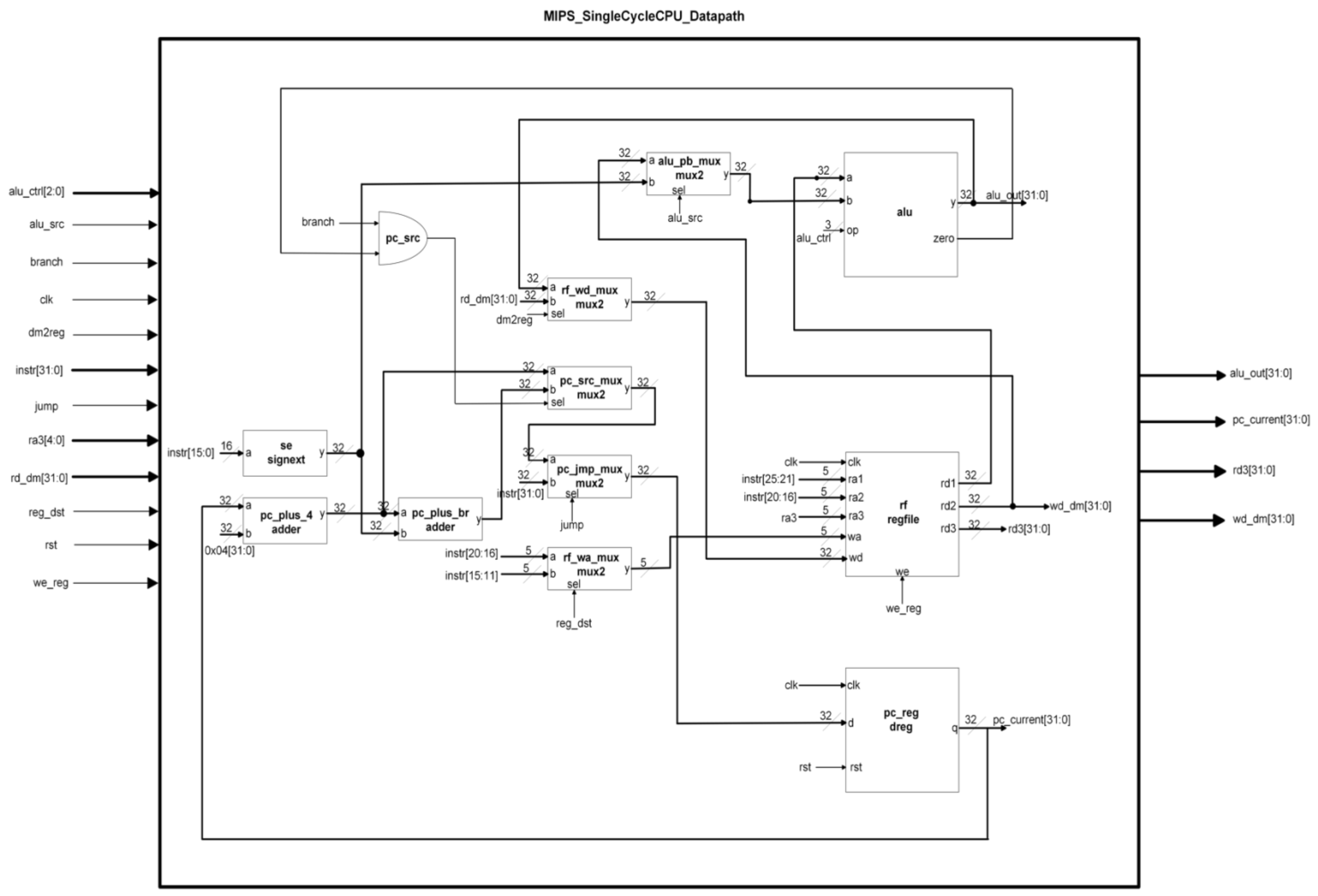
1. Worked together on the datapath and controlpath block diagram designs using Microsoft Visio tool.
2. By collaborating with each other, imported the given source files into Vivado and observed the output waveforms after the simulation.

CONCLUSION:

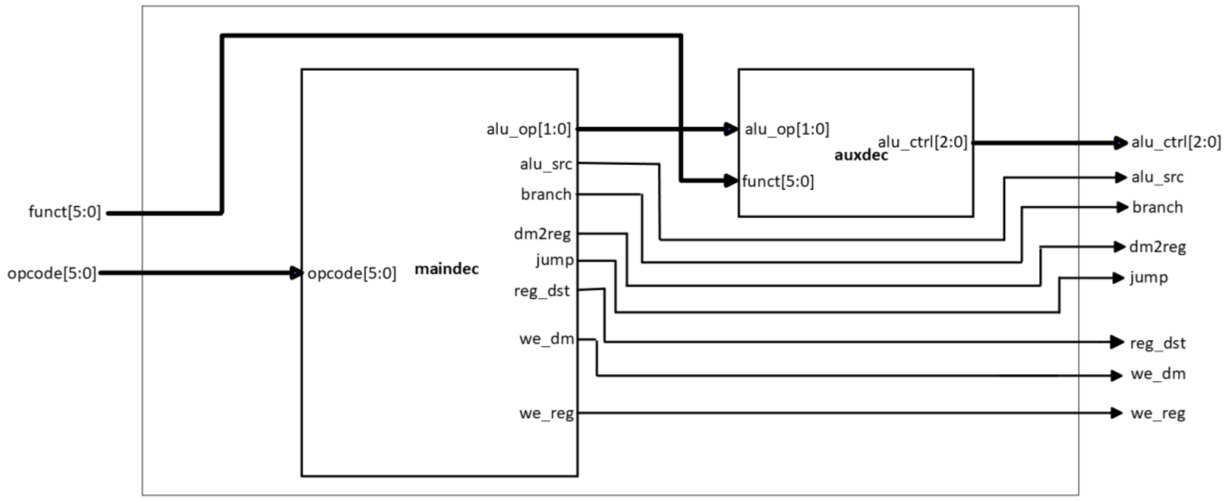
In conclusion, we end the report by understanding the given RTL Verilog code for the initial version of the single-cycle MIPS processor. We also were able to grasp the logic that was written in the Verilog files and along with that we were able to draw block diagrams based on the code. Apart from that, we learnt the basic techniques in functionally verifying a processor.

APPENDIX

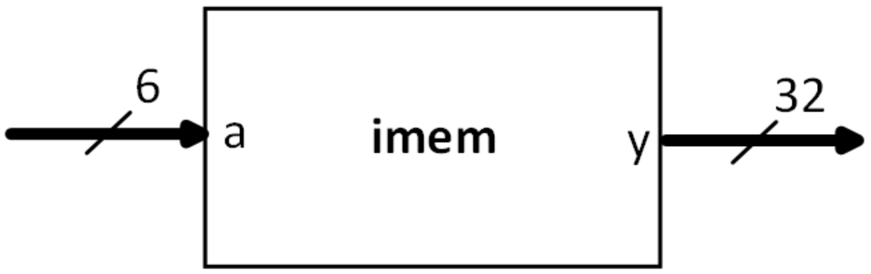
1) Block Diagram of the Single-Cycle MIPS Datapath



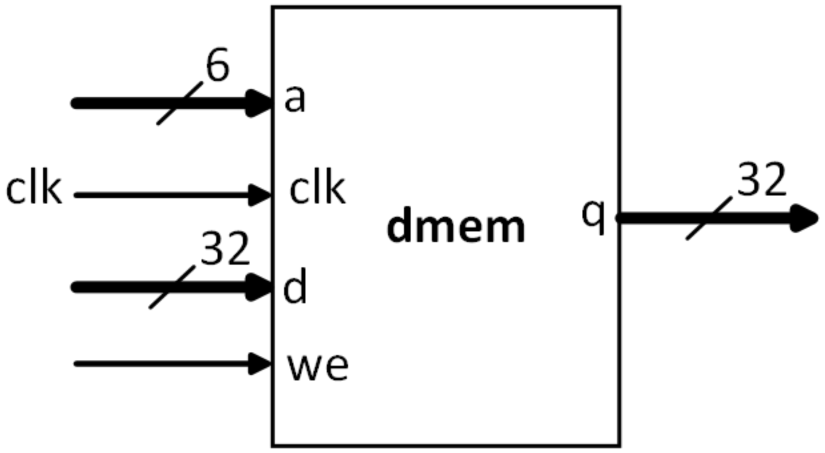
2) Block Diagram of the Single-Cycle MIPS Controlpath



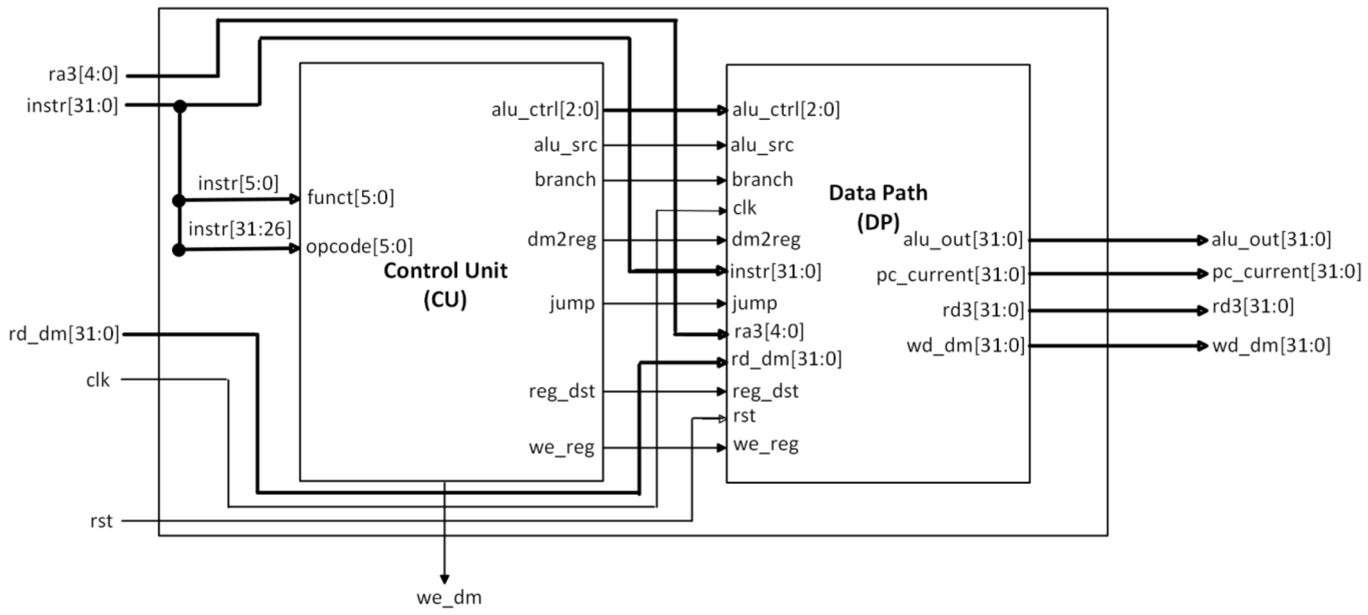
3) Block Diagram of the Instruction Memory



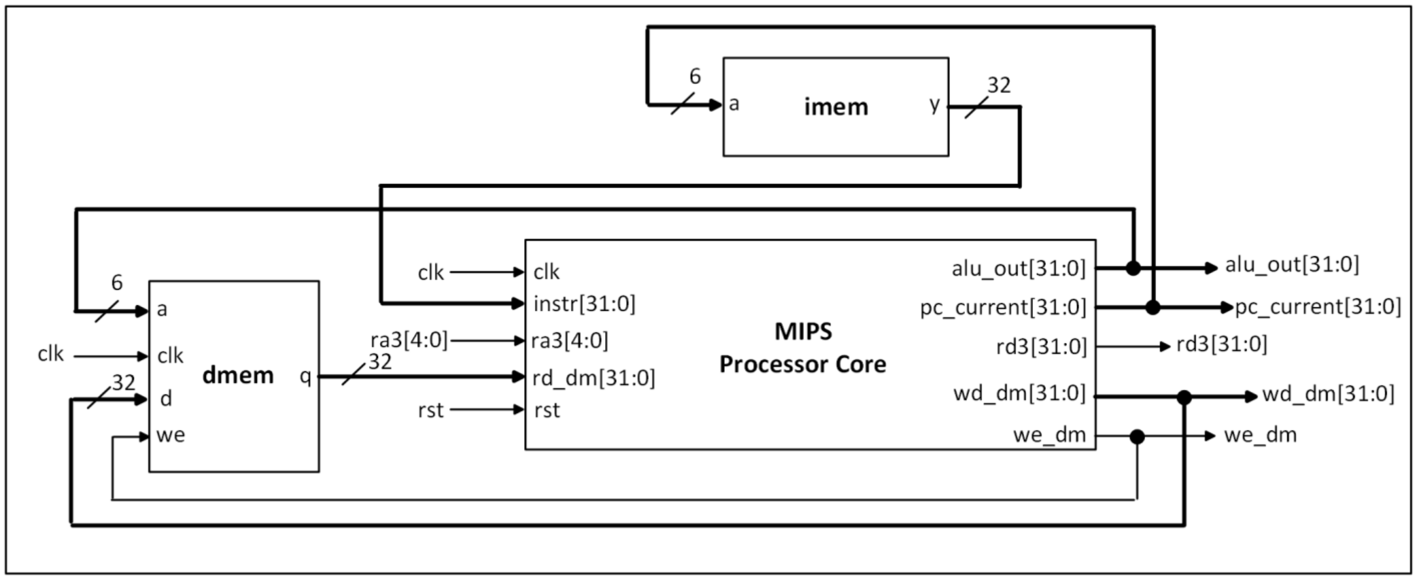
4) Block Diagram of the Data Memory



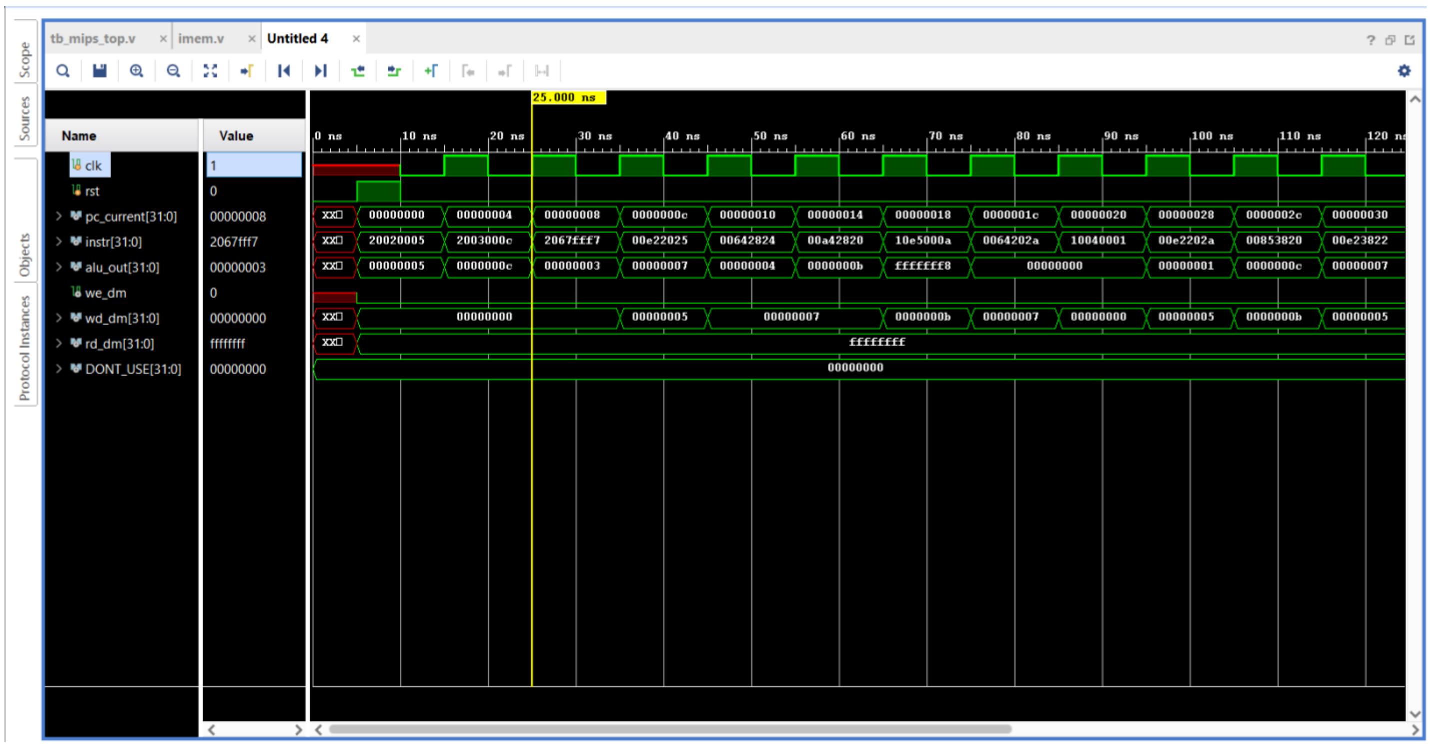
5) Block Diagram of the Processor-Core



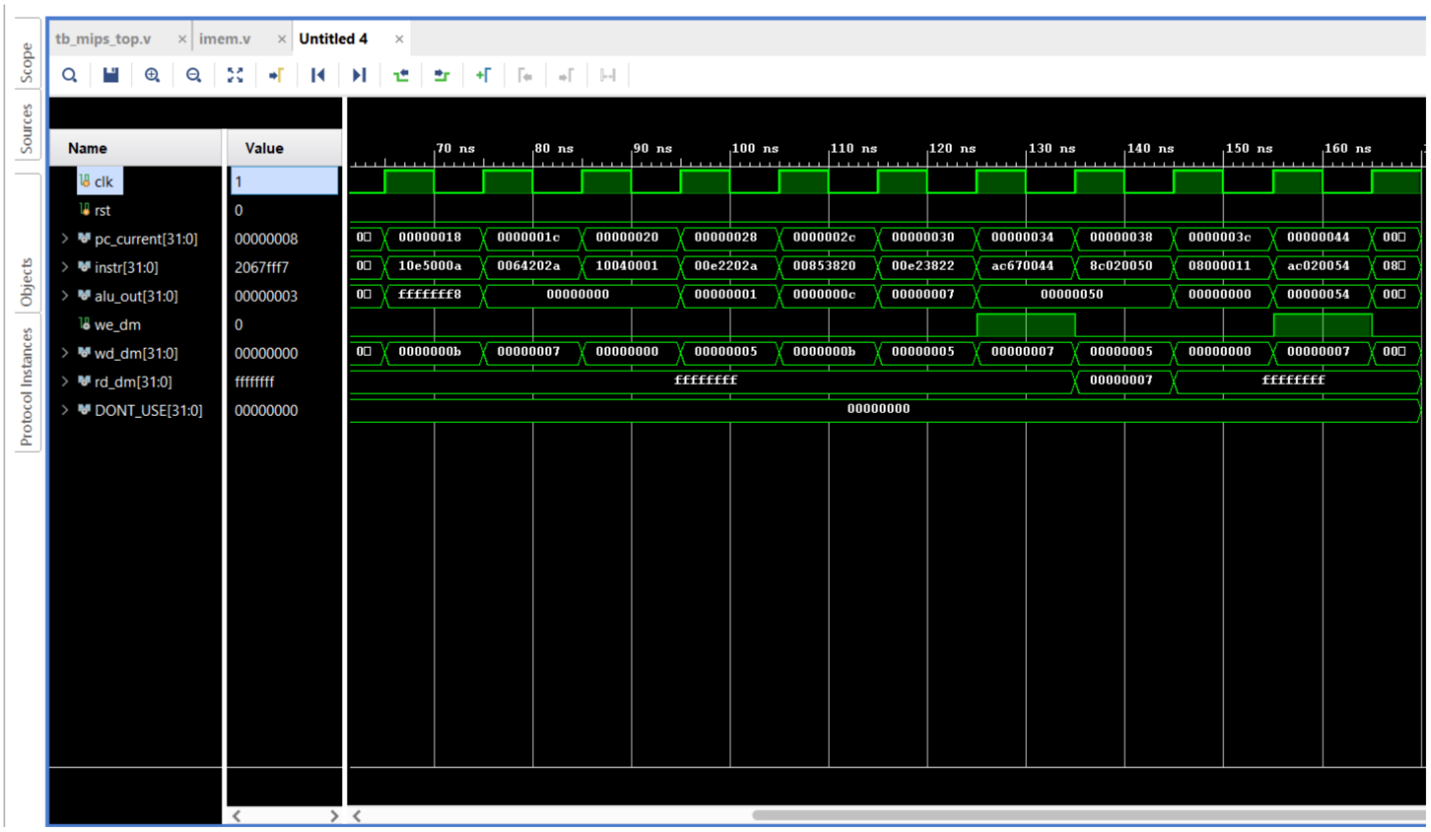
6) Block Diagram of the Complete Processor



7) Output Waveforms After the Simulation



tb_mips_top.v
imem.v
Untitled 4
Scope
Sources
Objects
Protocol Instances
25.000 ns
Name
Value
clk 1
rst 0
pc_current[31:0] 00000008
instr[31:0] 2067fff7
alu_out[31:0] 00000003
we_dm 0
wd_dm[31:0] 00000000
rd_dm[31:0] ffffffff
DONT_USE[31:0] 00000000
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000028 0000002c 00000030
20020005 2003000c 2067fff7 00e22025 00642824 00a42820 10e5000a 0064202a 10040001 00e2202a 00853820 00e23822
00000005 0000000c 00000003 00000007 00000004 0000000b fffffff8 00000000 00000001 0000000c 00000007
00000000 00000005 00000007 0000000b 00000007 00000000 00000005 0000000b 00000005
ffffffff
00000000



tb_mips_top.v
imem.v
Untitled 4
Scope
Sources
Objects
Protocol Instances
Name
Value
clk 1
rst 0
pc_current[31:0] 00000008
instr[31:0] 2067fff7
alu_out[31:0] 00000003
we_dm 0
wd_dm[31:0] 00000000
rd_dm[31:0] ffffffff
DONT_USE[31:0] 00000000
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
130 ns
140 ns
150 ns
160 ns
00000018 0000001c 00000020 00000028 0000002c 00000030 00000034 00000038 0000003c 00000044
10e5000a 0064202a 10040001 00e2202a 00853820 00e23822 ac670044 8c020050 08000011 ac020054
fffffff8 00000000 00000001 0000000c 00000007 00000050 00000000 00000054
0000000b 00000007 00000000 00000005 0000000b 00000005 00000007 00000005 00000000 00000007
ffffffff 00000007 ffffffff
00000000