NYU-6463 Processor Final Project Report Advanced Computer Hardware Design(ECE-GY 6463)

Instructors:
Prof. Ramesh Karri
Prof. Michail Maniatakos

By:-
Dhananjai Sharma(ds6365)
Puneet Rehal(pr1663)
Team:R2D2

Table of Contents

List of Figures

1. Block Diagram:

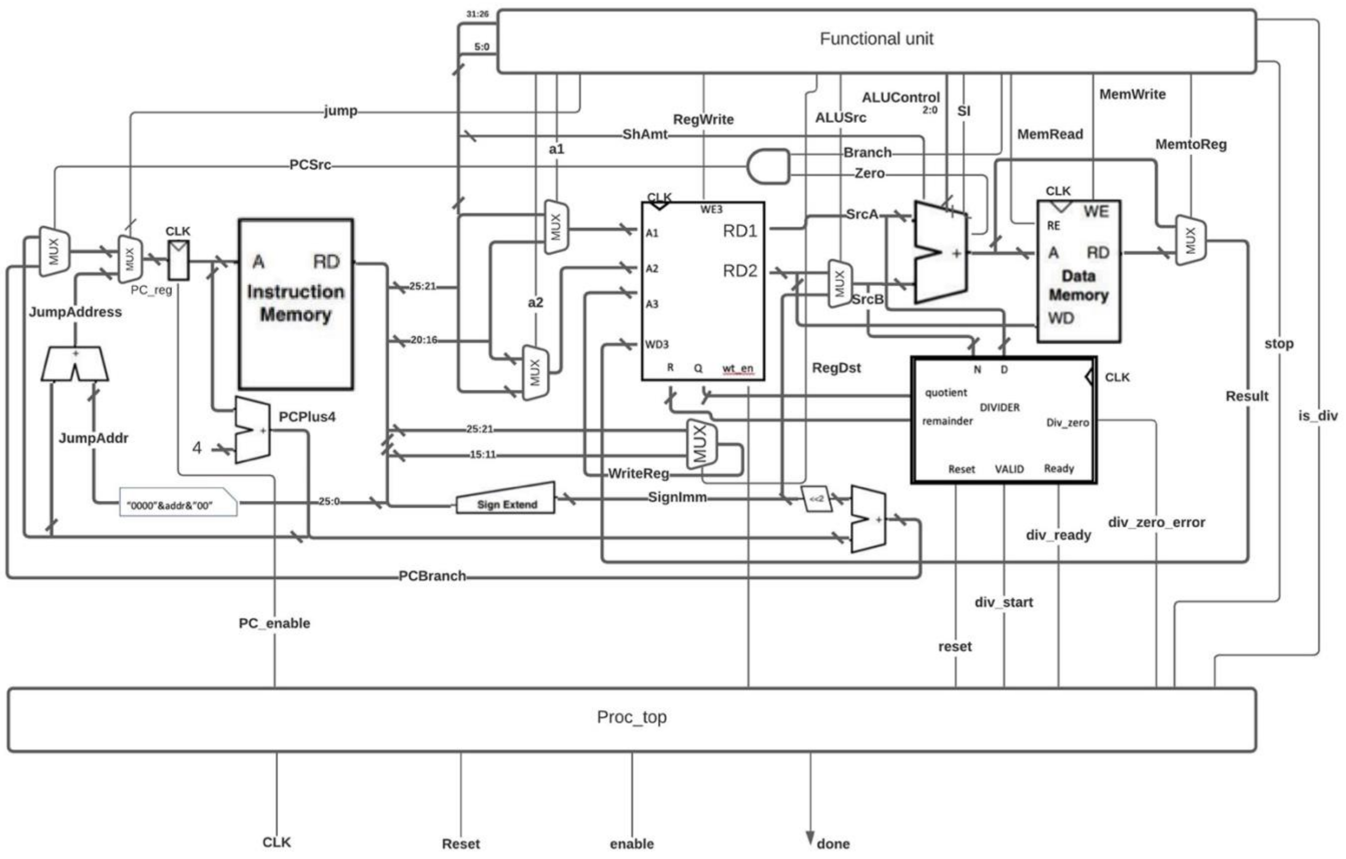
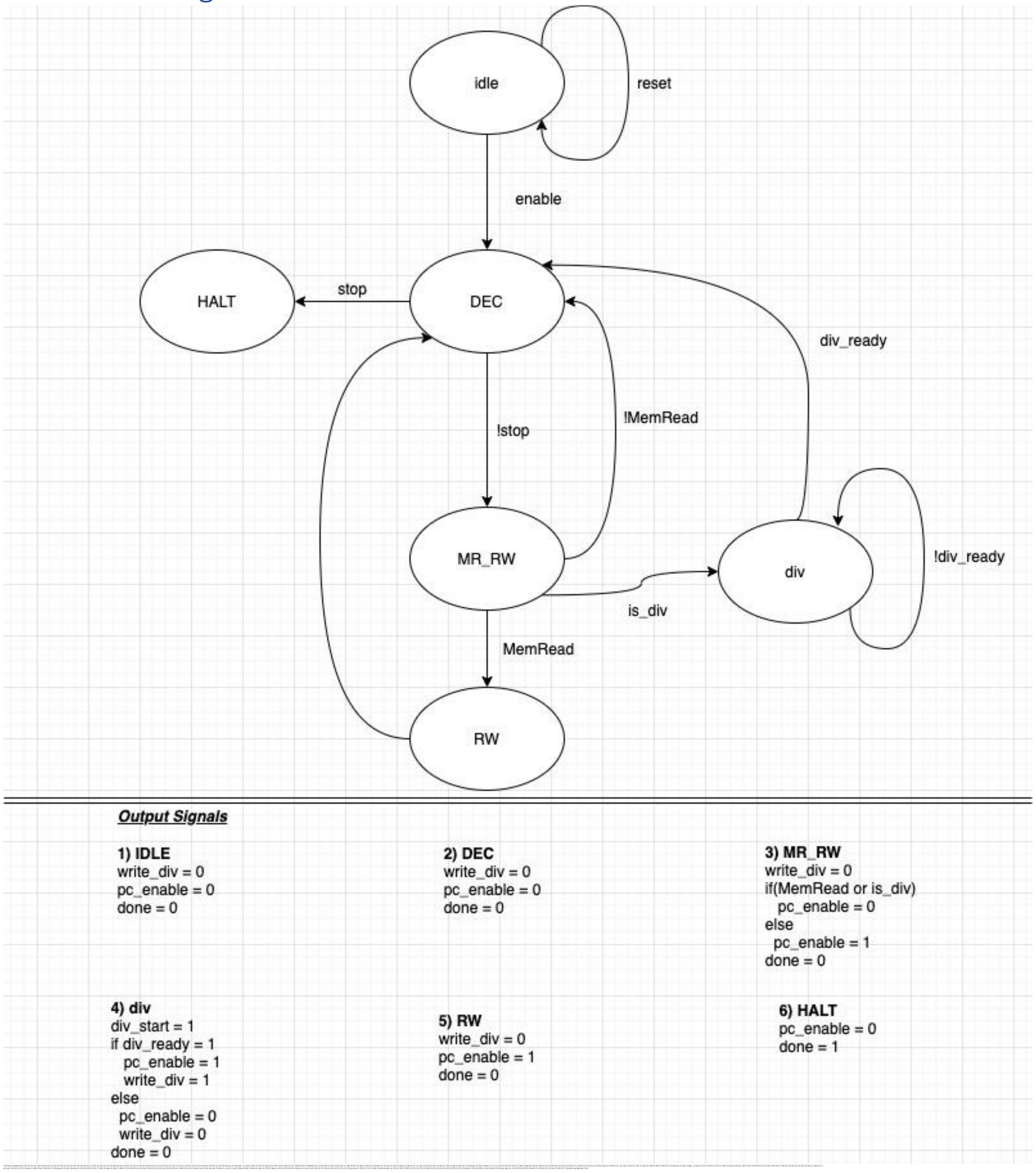


Figure 1: Modified diagram of NYU-6463 Processor

2. FSM Diagram:



Output Signals

1) IDLE
write_div = 0
pc_enable = 0
done = 0

2) DEC
write_div = 0
pc_enable = 0
done = 0

3) MR_RW
write_div = 0
if(MemRead or is_div)
  pc_enable = 0
else
  pc_enable = 1
done = 0

4) div
div_start = 1
if div_ready = 1
  pc_enable = 1
  write_div = 1
else
  pc_enable = 0
  write_div = 0
done = 0

5) RW
write_div = 0
pc_enable = 1
done = 0

6) HALT
pc_enable = 0
done = 1

Figure 2: FSM for NYU-6463 Processor implemented in proc_top

3. Design Hierarchy:

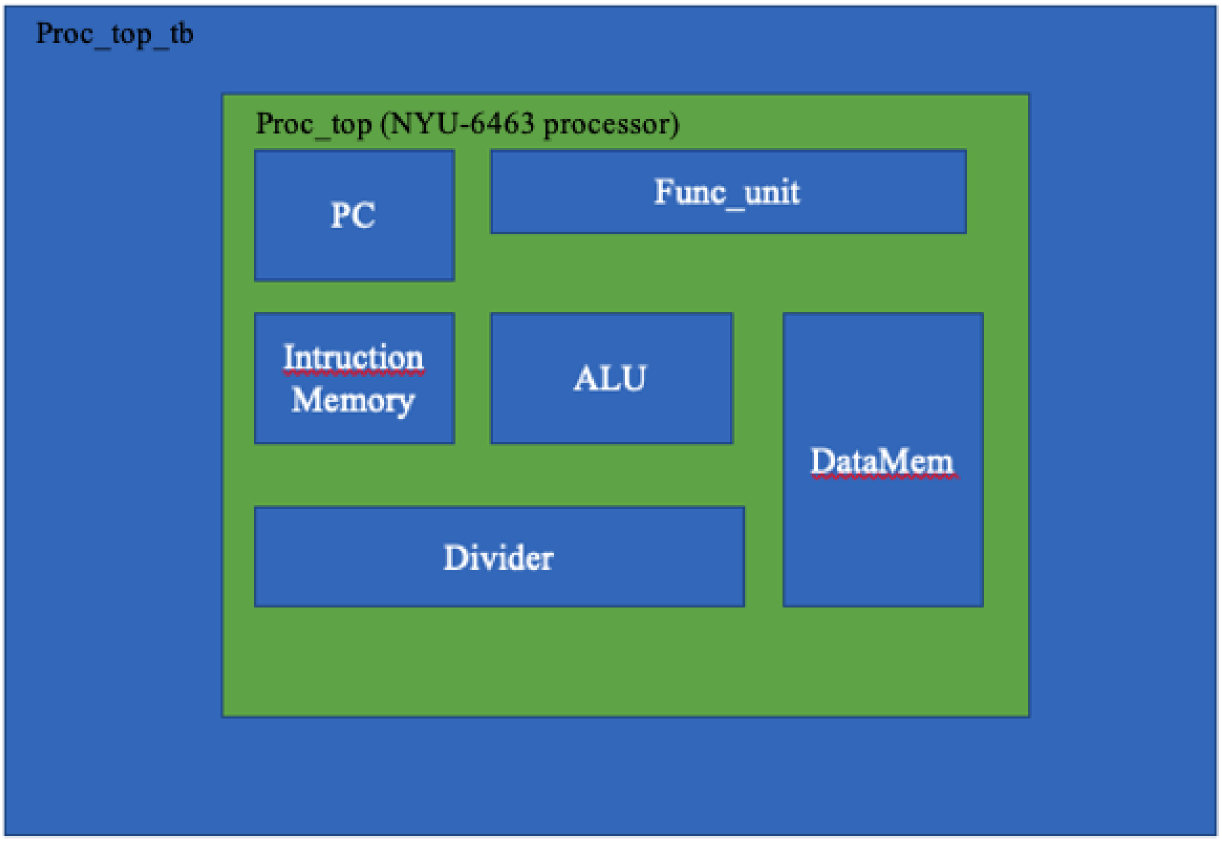


Figure 3:Organization of the major design modules

4. Types of Instructions

4.1 R-Type Instruction (Explanation & Simulation):

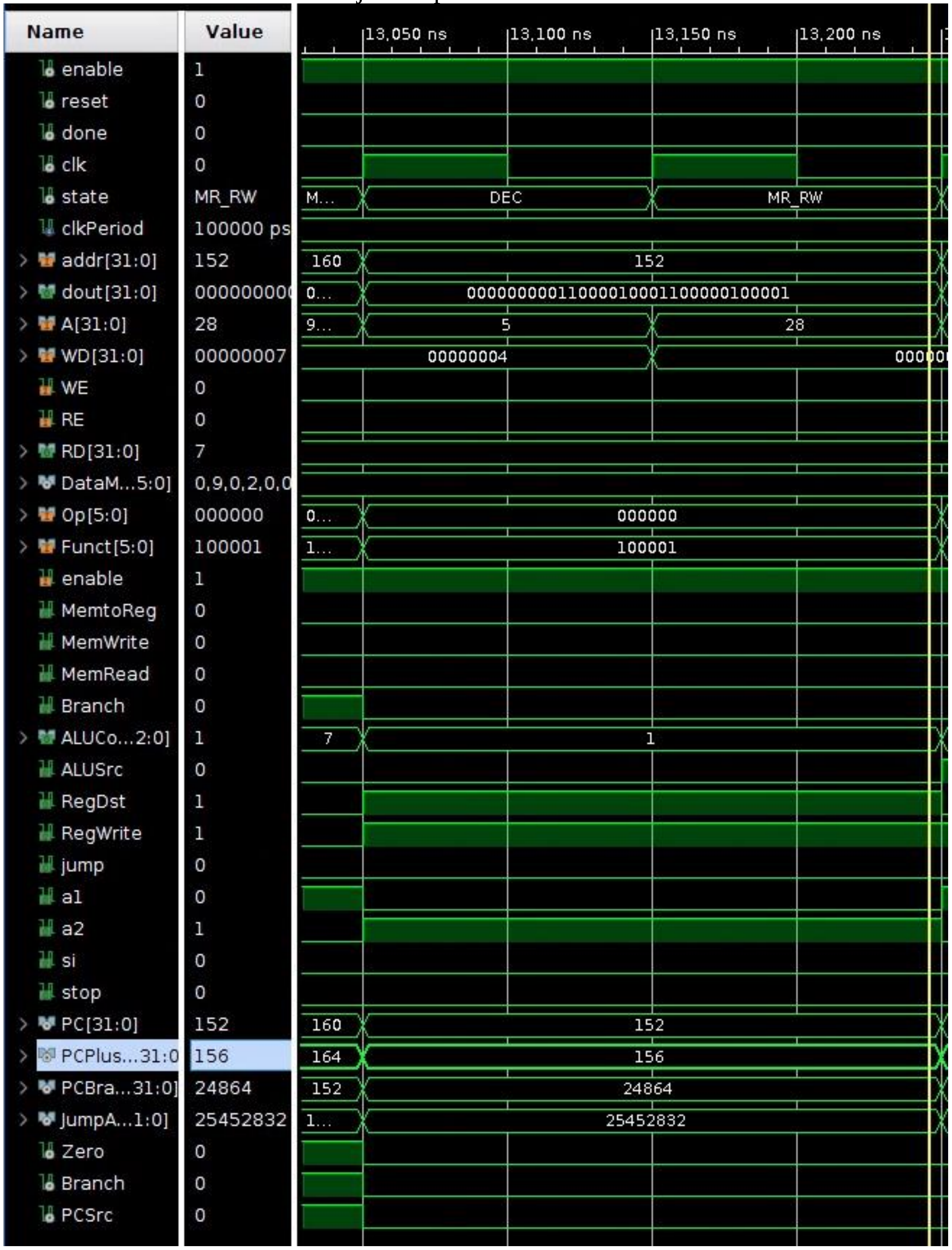
The R-Type instructions are executed in 2 cycles(though they can be executed in 1 cycle if the register file is operated at rising and falling edge, even though such a design was found to be functional in simulation but is non-synthesizable).
In cycle 1 , following operations are executed :-

1) Reading the instruction memory (assumed to be available instantly not clock triggered), the 32bit instruction is available on the output bus (RD) of. Instruction memory.
2) **The Functional Unit(FU)** decodes the instruction and issues control signals based on the 6 bit Opcode(31 downto 26) and 6 bit functional code(5 downto 0).
   The control signals issued by functional unit are as follow:
   a) RegWrite(high):Enabling the register file to be written
   b) MemWrite(low) : Disabling memory access
   c) MemRead(low) : Disabling memory access
   d) SI(high for special instructions else low)
   e) ALU Control

Sr. no.	SI	ALUControl	Operation
1	0	010	AND
2	0	011	OR
3	0	100	NOR
4	0	000	ADD
5	0	001	ADDU
6	1	010	DIVU
7	1	000	XRLR
8	1	001	RRXR
9	1	101	LRAD
10	1	110	SBRR

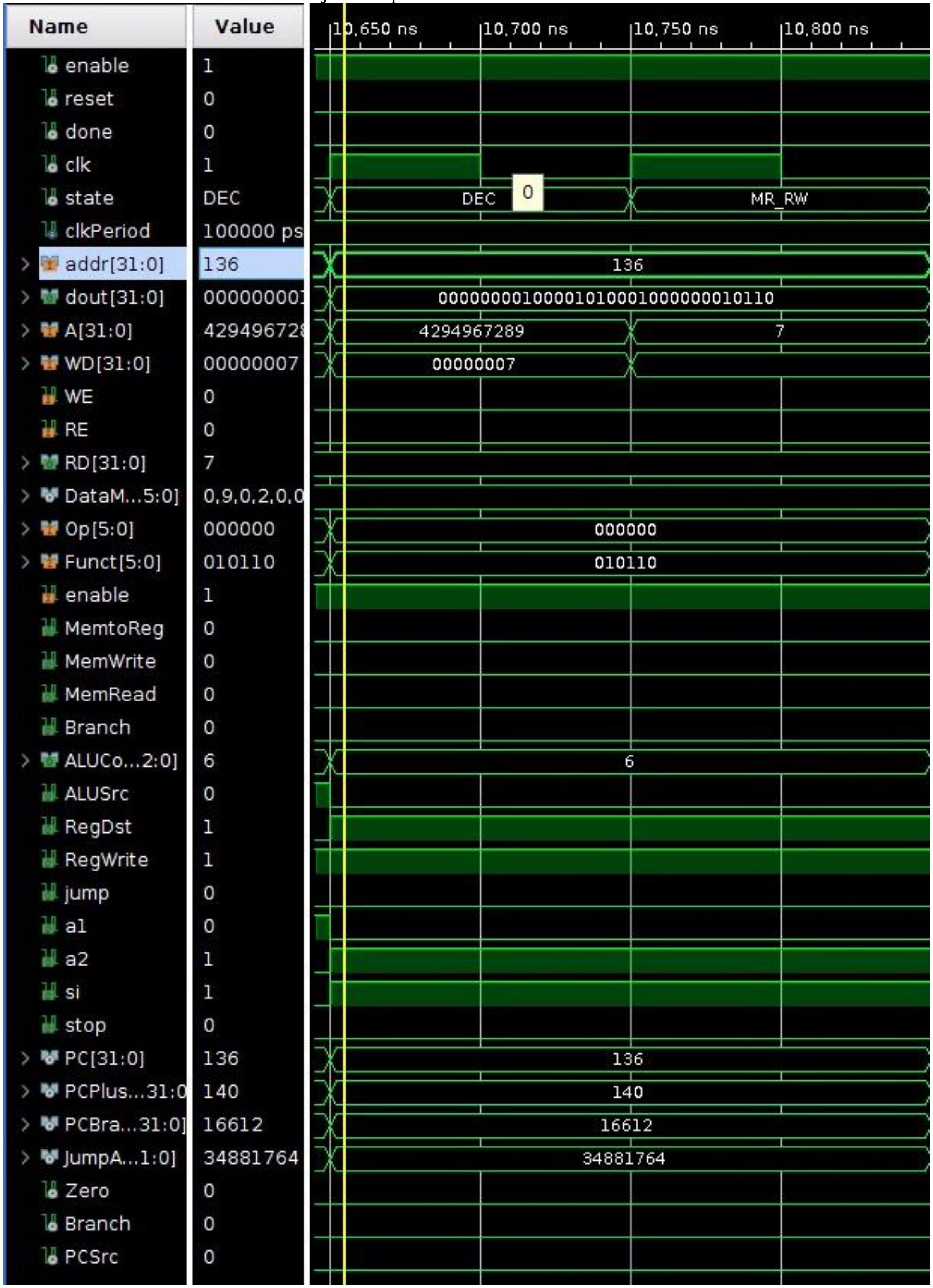
   f) Branch(LOW)
   g) MemtoReg(low)
   h) a1(low): selection bit to mux for register address A1 to the Register file (Instruction[25 downto 21]).
   i) a2(high) : selection bit to mux for register address A2 to the Register file (Instruction[20 downto 16]).
   j) RegDst: selection bit to select the address to A3 port of register file to select the address of the register to written.
3) The ALU calculates the desired result and the same is available at the **Result bus** at the end of clock 1.

In clock cycle 2, the register with address A3 is written.
Note- **The clock cycles** for the execution of the **unsigned division(DIVU)** is operand dependent as the division unit is separate from the ALU and the result of ALU operation is written to Register 30 and Register R31 directly after the Division unit asserts div_ready signal. Since, the division unit carries out the division using the Euclidean algorithm, **the number of clock cycles is equal to (3*Quotient+2) cycles.**



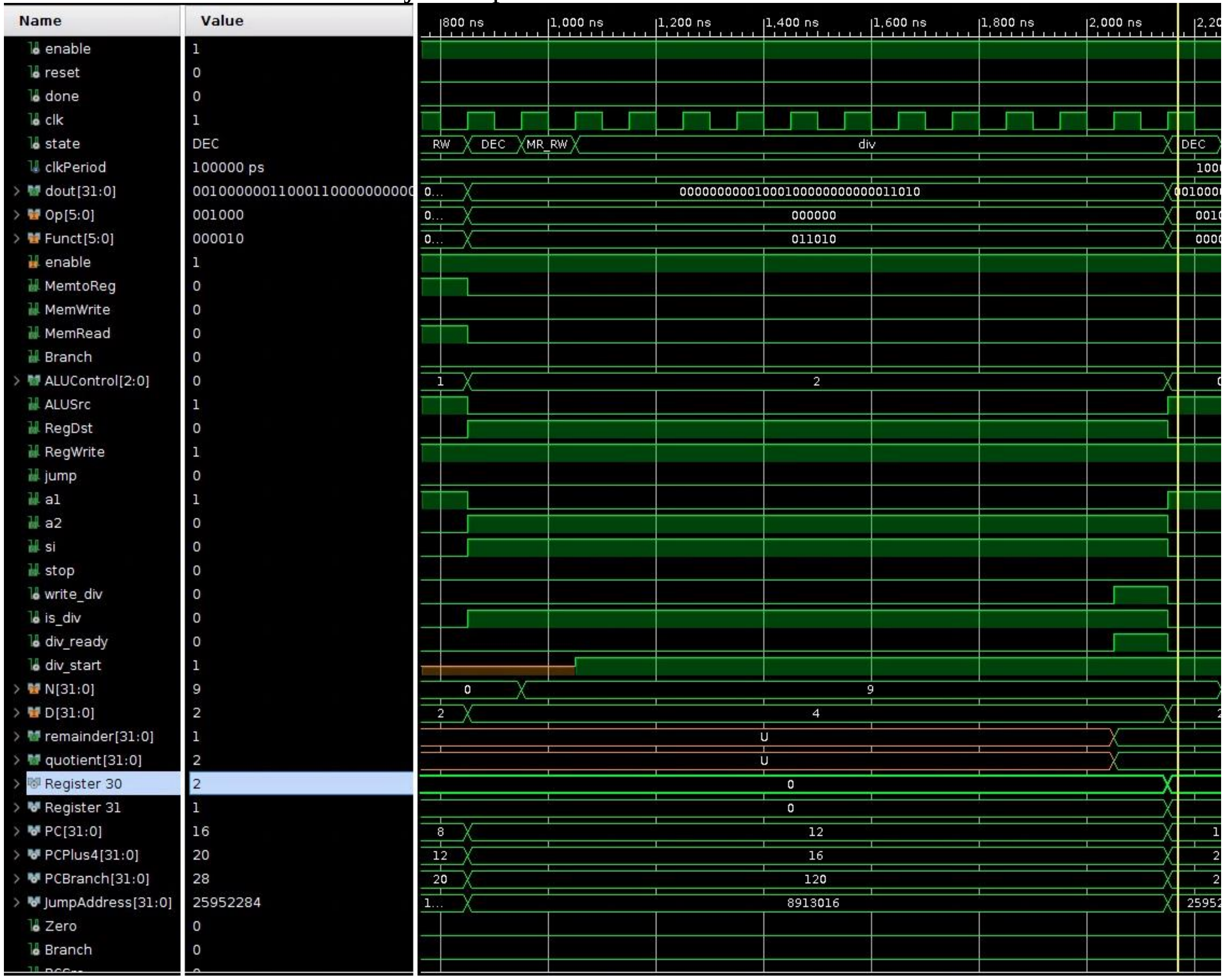
Name	Value
enable	1
reset	0
done	0
clk	0
state	MR_RW
clkPeriod	100000 ps
addr[31:0]	152
dout[31:0]	00000000
A[31:0]	28
WD[31:0]	00000007
WE	0
RE	0
RD[31:0]	7
DataM...5:0]	0,9,0,2,0,0
Op[5:0]	000000
Funct[5:0]	100001
enable	1
MemtoReg	0
MemWrite	0
MemRead	0
Branch	0
ALUCo...2:0]	1
ALUSrc	0
RegDst	1
RegWrite	1
jump	0
a1	0
a2	1
si	0
stop	0
PC[31:0]	152
PCPlus...31:0	156
PCBra...31:0]	24864
JumpA...1:0]	25452832
Zero	0
Branch	0
PCSrc	0

Figure 4: Execution of R-Type instruction (ADDU)



Name	Value
enable	1
reset	0
done	0
clk	1
state	DEC
clkPeriod	100000 ps
addr[31:0]	136
dout[31:0]	00000000
A[31:0]	42949672
WD[31:0]	00000007
WE	0
RE	0
RD[31:0]	7
DataM...5:0]	0,9,0,2,0,0
Op[5:0]	000000
Funct[5:0]	010110
enable	1
MemtoReg	0
MemWrite	0
MemRead	0
Branch	0
ALUCo...2:0]	6
ALUSrc	0
RegDst	1
RegWrite	1
jump	0
a1	0
a2	1
si	1
stop	0
PC[31:0]	136
PCPlus...31:0	140
PCBra...31:0]	16612
JumpA...1:0]	34881764
Zero	0
Branch	0
PCSrc	0

Figure 5: Execution of R-Type Special Instruction SUBRR



Name	Value
enable	1
reset	0
done	0
clk	1
state	DEC
clkPeriod	100000 ps
dout[31:0]	00100000011000110000000000
Op[5:0]	001000
Funct[5:0]	000010
enable	1
MemtoReg	0
MemWrite	0
MemRead	0
Branch	0
ALUControl[2:0]	0
ALUSrc	1
RegDst	0
RegWrite	1
jump	0
a1	1
a2	0
si	0
stop	0
write_div	0
is_div	0
div_ready	0
div_start	1
N[31:0]	9
D[31:0]	2
remainder[31:0]	1
quotient[31:0]	2
Register 30	2
Register 31	1
PC[31:0]	16
PCPlus4[31:0]	20
PCBranch[31:0]	28
JumpAddress[31:0]	25952284
Zero	0
Branch	0

Figure 6: Execution of R-Type special instruction :DIVU

4.2 I-Type Instruction (Explanation & Simulation):

The I-Type instruction takes 2 cycles(except Load instruction which takes 3 cycles) in the synthesizable version of the processor NYU-6463(triggering on only rising edge of clock) and 1 clock(except LOAD instruction which takes 2 cycles) on a design which is triggered on both falling and rising edge.

The operations carried by the end of 1st clock cycle.

1) Reading the instruction (assumed to be available instantly not clock triggered), the 32bit instruction is available on the output bus (RD) of Instruction memory.
2) **The Functional Unit(FU)** decodes the instruction and issues control signals based on the 6 bit Opcode(31 downto 26) and 6 bit functional code(5 downto 0).

   The control signals issued by functional unit are as follow:

   a) RegWrite: This signal is set high for I-Type instructions except for STORE instruction as the register file is not written in the STORE operation.
   b) MemWrite(low) : Is set low except in the case of STORE operation as the memory is written for the STORE operation.
   c) MemRead(low) : Is pulled low for I-Type instruction except in the case of Load instruction as the memory read operation is performed in order to read the value to be loaded in the target register address.
   d) SI: Is set low as all the special instructions are R-Type instructions
   e) ALU Control

Sr. no.	SI	ALUControl	Operation
1	0	010	ANDI
2	0	011	ORI
3	0	001	LW
4	0	001	SW
5	0	101	BLT
6	0	110	BEQ
7	0	111	BNE
8	0	000	ADDI

f) Branch :For opcodes 0x09,0x0A and 0x0B the functional unit sets the Branch signal otherwise the branch signal is set low.

g) MemtoReg: If the decoded instruction is Load then this control signal is set high otherwise low (for selecting the output of memory in case of STORE instruction otherwise result of ALU is forwarded to the ALU bus)

h) a1(high): selection bit to mux for register address A1 to the Register file (Instruction[20 downto 16]).

i) a2(low) : selection bit to mux for register address A2 to the Register file (Instruction[25 downto 21]).

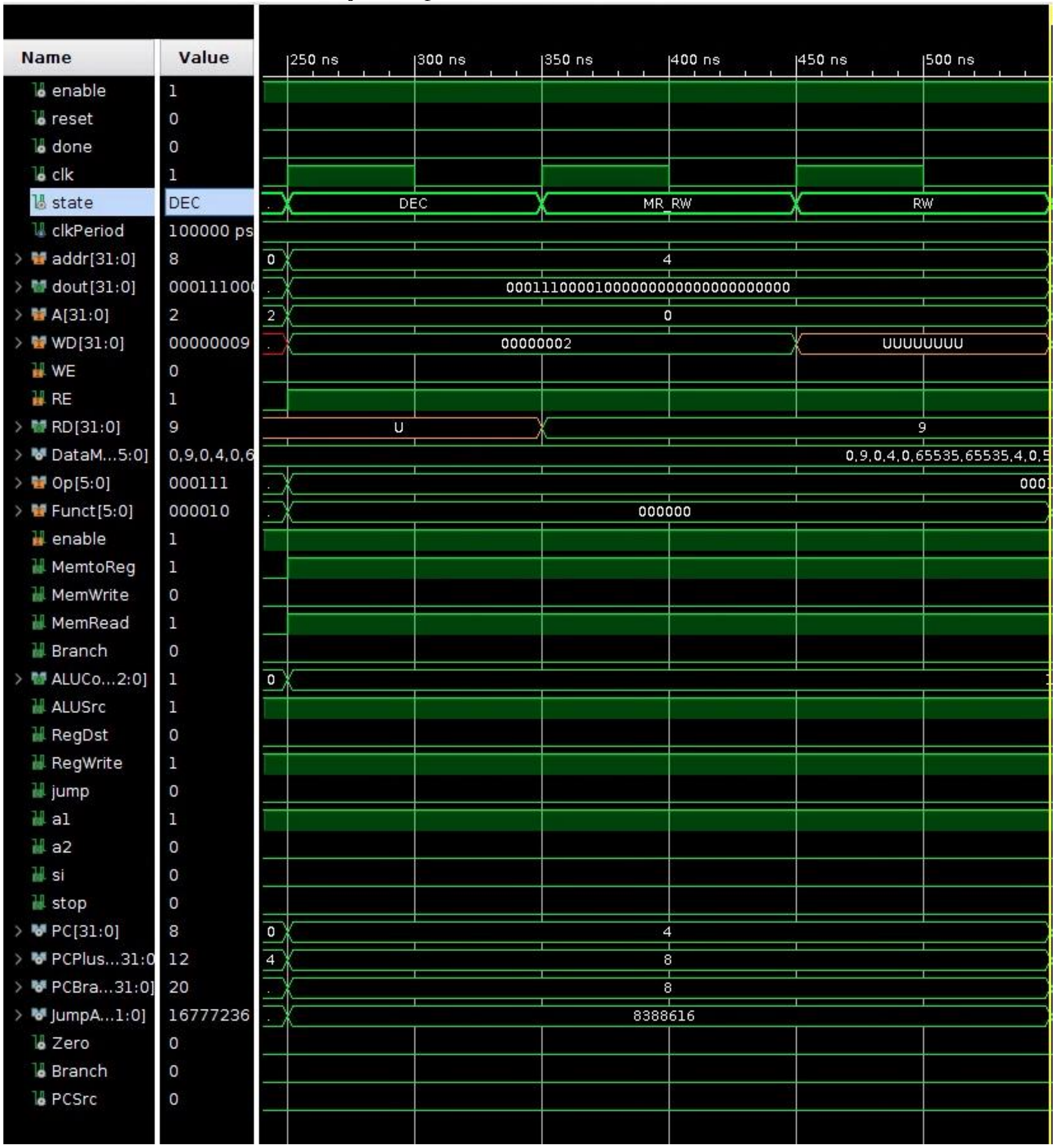
3) RegDst: selection bit to select the address to A3 port of register file to select the address which is extracted from Instruction[25 downto 21].

4) The Brach address is available at **PCBranch bus** (Sign Extended(Instruction(15 downto 0)&"00"+PC+4)

5) ALU operation is takes place(**operands are the value of register Rs and sign extended immediate**) where it outputs the computational result on the ALUResult bus and Zero which is high if the condition of branch is satisfied else it is pulled to low.

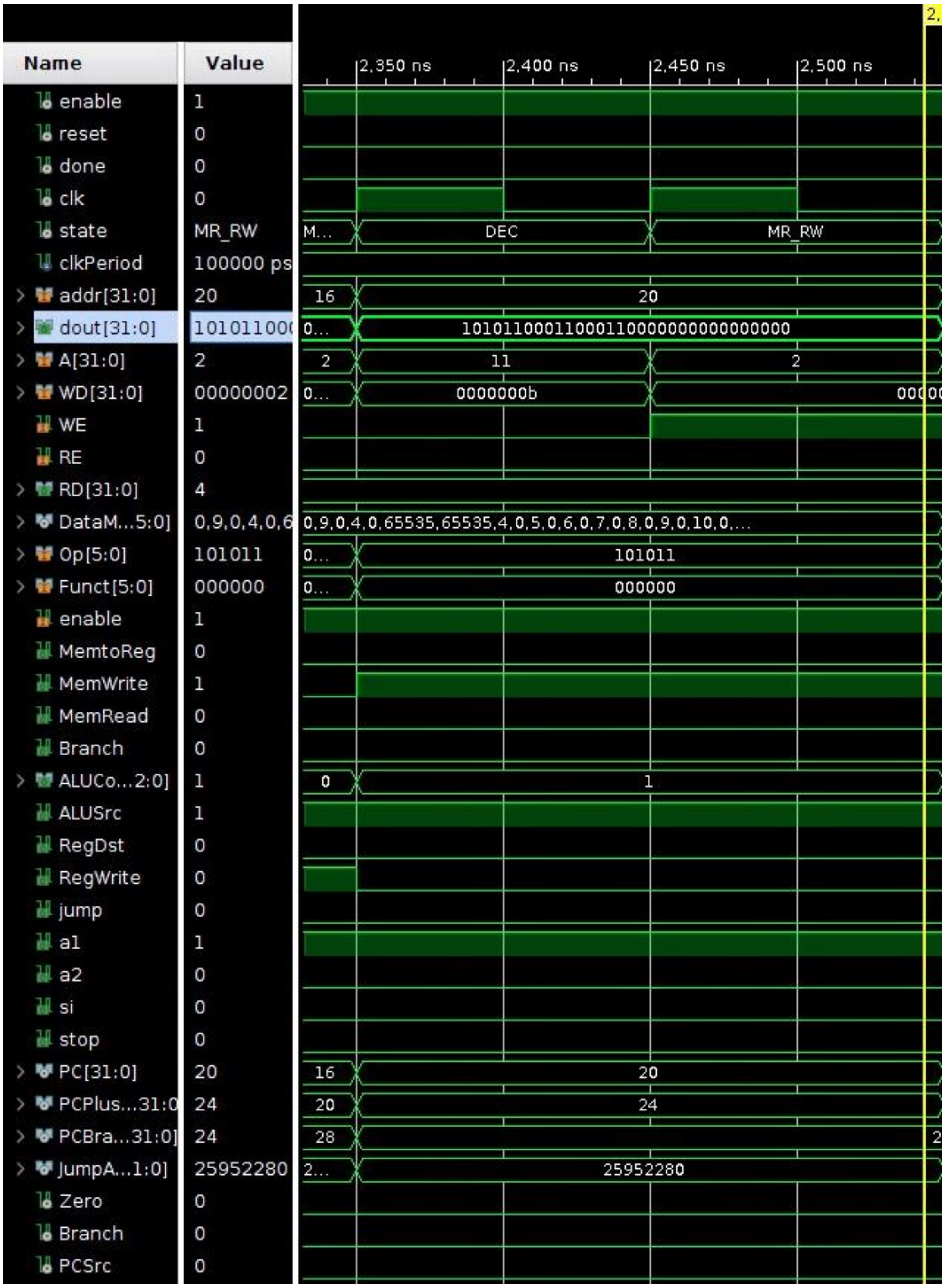
In cycle 2, if the decoded instruction is Load or Store, memory access is performed.
In case of Load instruction the data memory is read from the address available at the o/p of ALU and if the decoded instruction is LOAD the memory is written with the value read from the register file(available at the ALU o/p).
For other I-Type instructions, available ALU result(calculated **on sign extended immediate and register Rs**) on the **Result bus** is written in the register memory at the register Rt.
In cycle 3, if the decoded instruction is Load the data read from the Data memory is written into the register file at register Rt

If **PCSrc** is high(Branch and Zero are high) the calculated **branch address** on the **PCBranch** bus is loaded into the PC register for the next decode state.



Name	Value
enable	1
reset	0
done	0
clk	1
state	DEC
clkPeriod	100000 ps
addr[31:0]	8
dout[31:0]	000111000
A[31:0]	2
WD[31:0]	00000009
WE	0
RE	1
RD[31:0]	9
DataM...5:0]	0,9,0,4,0,6
Op[5:0]	000111
Funct[5:0]	000010
enable	1
MemtoReg	1
MemWrite	0
MemRead	1
Branch	0
ALUCo...2:0]	1
ALUSrc	1
RegDst	0
RegWrite	1
jump	0
a1	1
a2	0
si	0
stop	0
PC[31:0]	8
PCPlus...31:0	12
PCBra...31:0]	20
JumpA...1:0]	16777236
Zero	0
Branch	0
PCSrc	0

Figure 7: Execution of I-Type instruction: Load



```
Name            Value
enable          1
reset           0
done            0
clk             0
state           MR_RW
clkPeriod       100000 ps
addr[31:0]      20
dout[31:0]      10101100011000110000000000000000
A[31:0]         2
WD[31:0]        00000002
WE              1
RE              0
RD[31:0]        4
DataM...5:0]    0,9,0,4,0,65535,65535,4,0,5,0,6,0,7,0,8,0,9,0,10,0,...
Op[5:0]         101011
Funct[5:0]      000000
enable          1
MemtoReg        0
MemWrite        1
MemRead         0
Branch          0
ALUCo...2:0]    1
ALUSrc          1
RegDst          0
RegWrite        0
jump            0
a1              1
a2              0
si              0
stop            0
PC[31:0]        20
PCPlus...31:0   24
PCBra...31:0]   24
JumpA...1:0]    25952280
Zero            0
Branch          0
PCSrc           0
```

Figure 8:Execution of I-Type instruction: Store

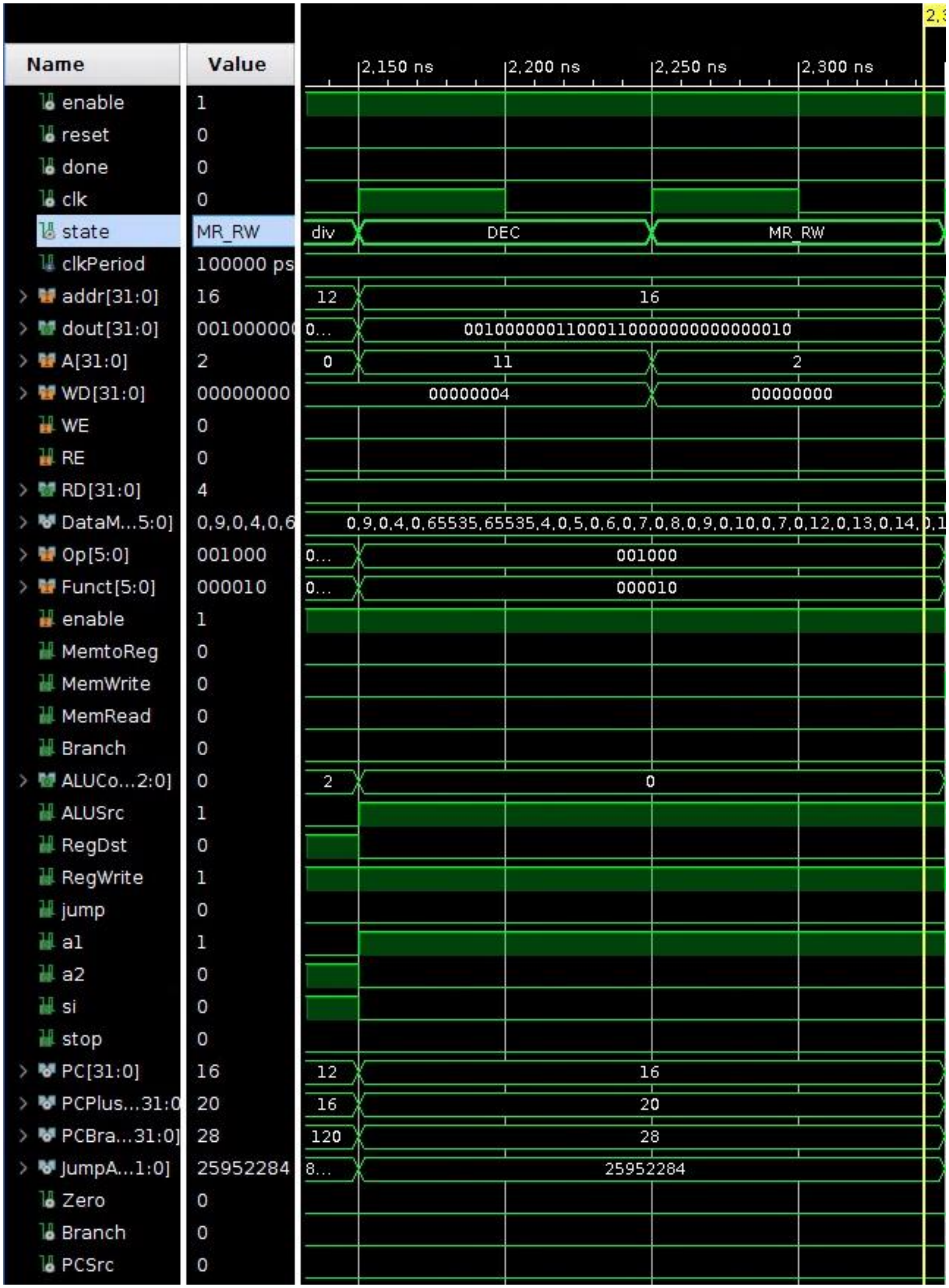
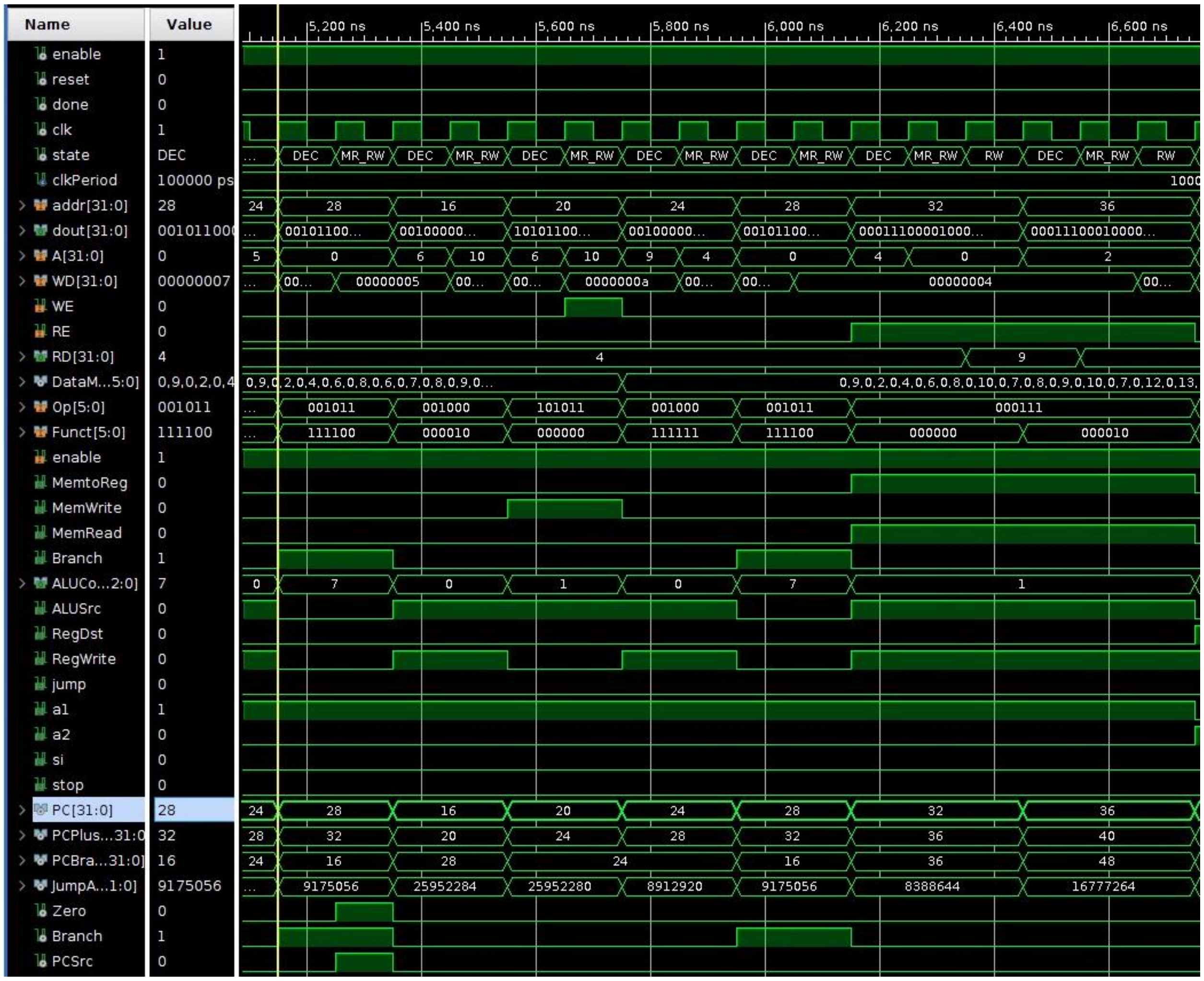


Figure 9: Execution of I-Type instruction: ADDI



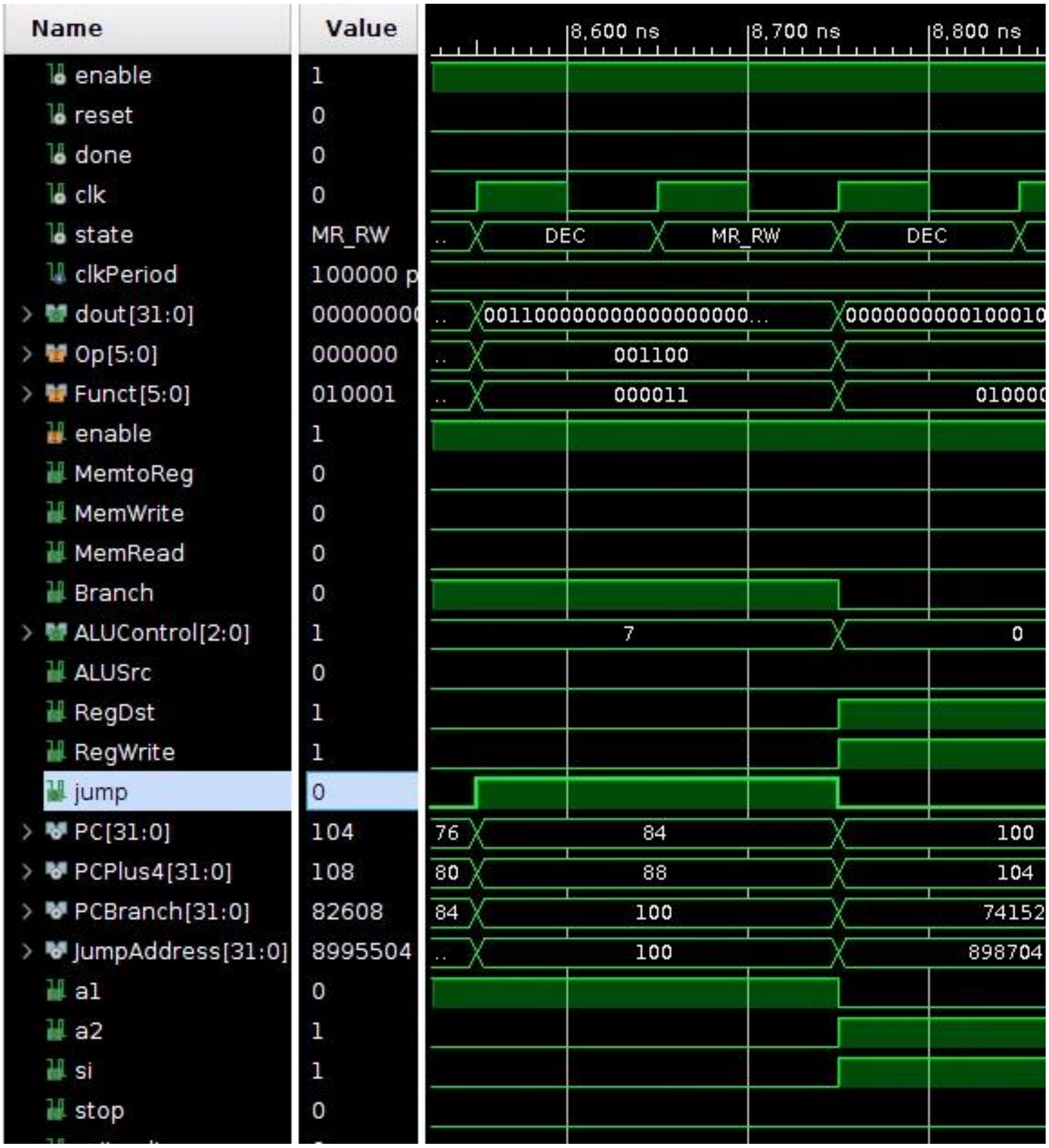
Name	Value
enable	1
reset	0
done	0
clk	1
state	DEC
clkPeriod	100000 ps
addr[31:0]	28
dout[31:0]	00101100(
A[31:0]	0
WD[31:0]	00000007
WE	0
RE	0
RD[31:0]	4
DataM...5:0]	0,9,0,2,0,4
Op[5:0]	001011
Funct[5:0]	111100
enable	1
MemtoReg	0
MemWrite	0
MemRead	0
Branch	1
ALUCo...2:0]	7
ALUSrc	0
RegDst	0
RegWrite	0
jump	0
a1	1
a2	0
si	0
stop	0
PC[31:0]	28
PCPlus...31:0	32
PCBra...31:0]	16
JumpA...1:0]	9175056
Zero	0
Branch	1
PCSrc	0

Figure 10: Execution of I-Type instruction: BNE

4.3 J-Type Instruction (Explanation & Simulation):

The J-Type instruction is executed in 1-clock cycle,

1) Reading the instruction (assumed to be available instantly not clock triggered), the 32bit instruction is available on the output bus (RD) of Instruction memory.
2) **The Functional Unit(FU)** decodes the instruction and issues control signals based on the 6 bit Opcode(31 downto 26) and 6 bit functional code(5 downto 0).
   The control signals issued by functional unit are as follow:
   a) To avoid any unwarranted memory and register file access the following signals are pulled low
      - RegWrite
      - MemWrite
      - MemRead
   b) jump signal is set high to select the source of PC input.
3) The Jump address is calculated by jump adder by adding the operands “0000”&Instruction[25 downto 0]&”00” and PCPlus4 and is available at the PC register i/p.
4) The calculated Jump address is loaded into the PC register



Name	Value
enable	1
reset	0
done	0
clk	0
state	MR_RW
clkPeriod	100000 p
dout[31:0]	0000000
Op[5:0]	000000
Funct[5:0]	010001
enable	1
MemtoReg	0
MemWrite	0
MemRead	0
Branch	0
ALUControl[2:0]	1
ALUSrc	0
RegDst	1
RegWrite	1
jump	0
PC[31:0]	104
PCPlus4[31:0]	108
PCBranch[31:0]	82608
JumpAddress[31:0]	8995504
a1	0
a2	1
si	1
stop	0

Figure 11:Simulation of a jump instruction (JMP #3)

5. Assembly Programs

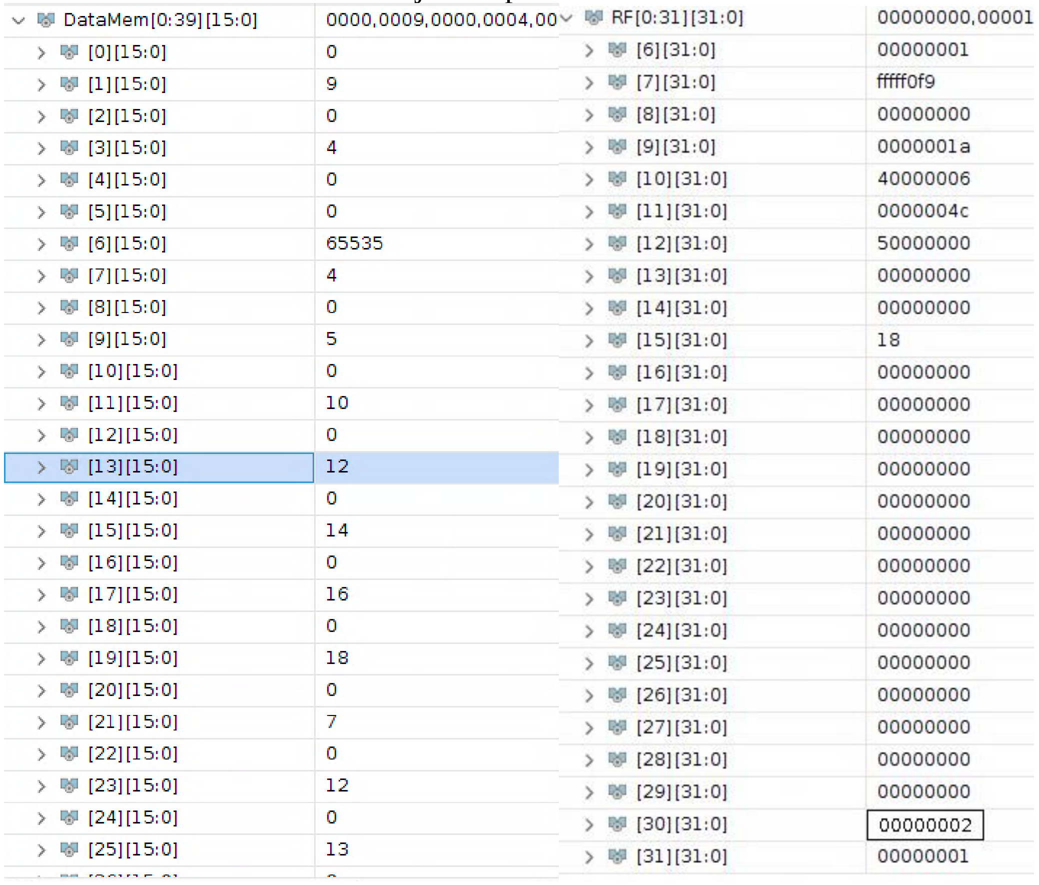
5.1 Simple Assembly Program (Simulation):

```
 "00100001","11100000","00000000","00001000",--#0 R[15] = R0 + 8
"00011100","00100000","00000000","00000000",--#4 R[1] = M[R[0]+0]
"00011100","01000000","00000000","00000010",--#8 R[2] = M[R[0]+2]
"00000000","00100010","00000000","00011010",--#12 division R1/R2
"00100001","11101111","00000000","00000010",--#16 R15 =R15+2
"10101101","11101111","00000000","00000000", --#20 LW R15,R0,#0 (M[R15+0]<=R15)
"00100000","00100001","11111111","11111111",--#24 R1 =R1-1
"00101100","00100010","11111111", "11111100",--#28 BNE R1,R2,#-4
```

Mnemonic representation of the simple program

```
#0 ADDI R0,R15,#8
#4 LW R0,R1,#0
#8 LW R0,R2,#0
#12 DIVU R1,R2
#16 ADDI R15,R15,#2
#20 LW R0 ,R15,R15
#24 ADDI R1,R1,#-1
#28 BNE R1,R2,#-4
```

In the program the registers R1 and R2 are initialized to 9 and 4 using the load instruction from memory locations #0 and #2.(Dout of memory is combination of two locations as each memory location is 16bits long). R15 is initialized to 8 using ADDI instruction.
Firstly, the value in R1(9) is divided by the value in R2(4) by the divider unit and the results are stored in R31 and R32 which are hardwired to the output of the divider and the said registers are loaded when the **write_div** signal is provided by the proc_top.
The result of division is reflected in the screenshot of the register file at **Figure -xxx.**
Then the value of R15 is incremented by 2 and is stored at location M[R15] at each iteration, now the value of R1(9) is decremented by 1 and then the decremented value of R1 is compared to static value of R2(4) in the next instruction is the result of comparison is false, the PC pushed back 3 instructions.[PC+4+(-4*4)].
Therefore after the 5 iterations, the PC moves forward as the value of R1 and R2 becomes equal and the memory locations 10,12,14,16 and 18 are written with values 10,12,14,16 and 18.



DataMem[0:39][15:0]	0000,0009,0000,0004,00
[0][15:0]	0
[1][15:0]	9
[2][15:0]	0
[3][15:0]	4
[4][15:0]	0
[5][15:0]	0
[6][15:0]	65535
[7][15:0]	4
[8][15:0]	0
[9][15:0]	5
[10][15:0]	0
[11][15:0]	10
[12][15:0]	0
[13][15:0]	12
[14][15:0]	0
[15][15:0]	14
[16][15:0]	0
[17][15:0]	16
[18][15:0]	0
[19][15:0]	18
[20][15:0]	0
[21][15:0]	7
[22][15:0]	0
[23][15:0]	12
[24][15:0]	0
[25][15:0]	13

RF[0:31][31:0]	00000000,00001
[6][31:0]	00000001
[7][31:0]	fffff0f9
[8][31:0]	00000000
[9][31:0]	0000001a
[10][31:0]	40000006
[11][31:0]	0000004c
[12][31:0]	50000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	18
[16][31:0]	00000000
[17][31:0]	00000000
[18][31:0]	00000000
[19][31:0]	00000000
[20][31:0]	00000000
[21][31:0]	00000000
[22][31:0]	00000000
[23][31:0]	00000000
[24][31:0]	00000000
[25][31:0]	00000000
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000
[30][31:0]	00000002
[31][31:0]	00000001

Figure 12: The value of Register File(R31 and R30 with the result of division operation M[0]/M[2]) and the DATA memory with the modified locations 11-15

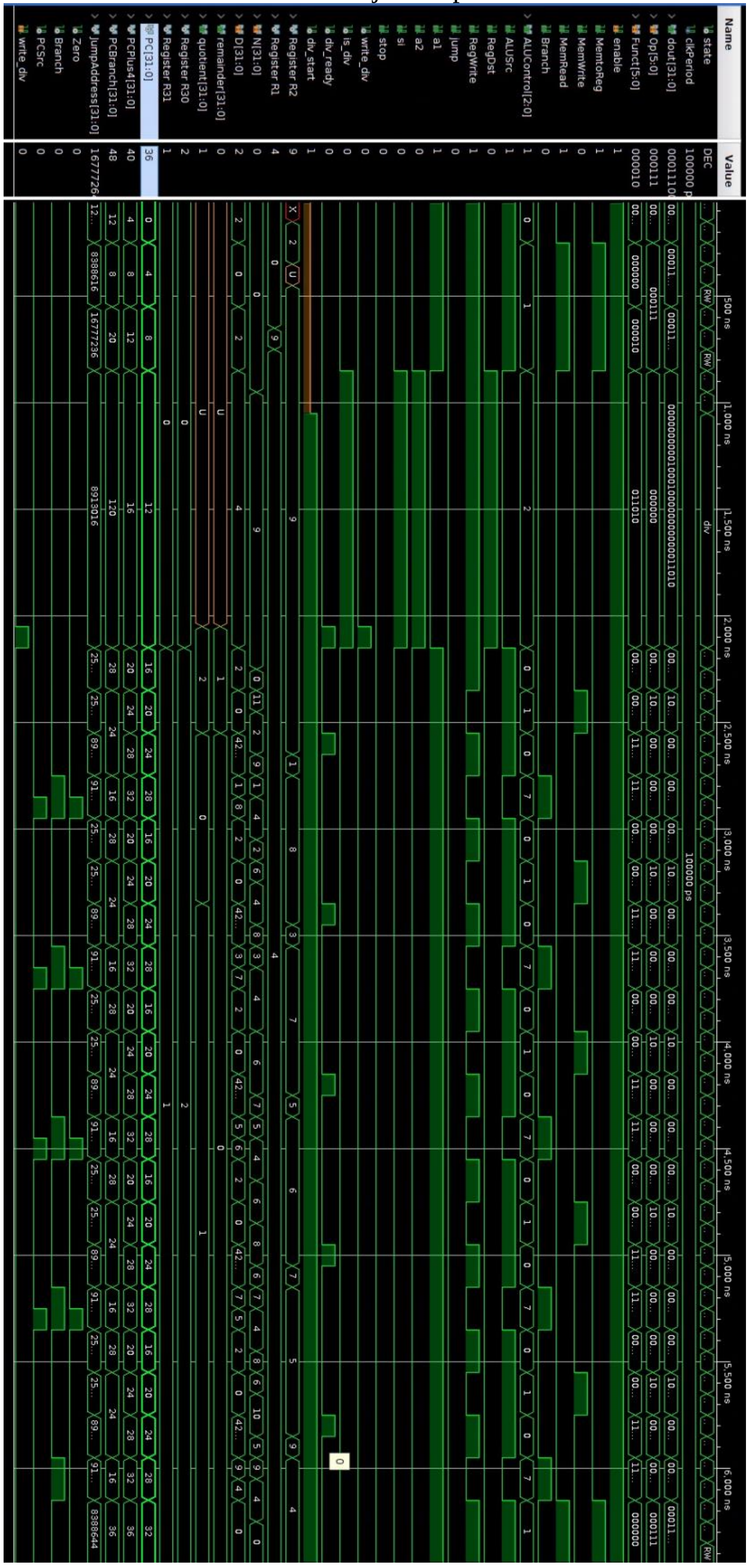


Figure 13: Simulation of a simple program

5.2 Complex Assembly Program (Simulation):

Program to calculate the factorial of a number stored at M[20] and put the result in R1.

```
"00100000","11000000","00000000","00000001",
"00100000","10100000","00000000","00000000",
"00011100","10000000","00000000","00010100",
"00100000","00100100","00000000","00000000",
"00100000","01100000","00000000","00000000",
"00000000","10000101","00010000","00010110",
"00100000","10100101","00000000","00000001",
"00101100","01000110","00000000","00000001",
"11111111","11111111", "11111111","11111111"
"00000000","01100001","00011000","00100001",
"00100000","01000010","11111111","11111111",
"00101100","01000110","11111111","11111101",
"00100000","00100011","00000000","00000000",
"00101000","00100001","11111111","11110110",
```

Setup

```
#116 ADDI R6,R0,#1          //  R6 = 1
#120 ADDI R5,R0,#0          // R5 = 0
#124 LW R4,R0,#20           //R4 =M[20]
#128 ADDI R1,R4,#0          //R1 = R4

#132 ADDI R3,R0,#0.         //R3 = 0
#136 SBRR R4,R5,R2,         //Subtract with right rotate(0 rotations) R2 = R4-R5
#140 ADDI R5,R5,#1.         //R5 =R5+1
#144 BNE R2,R6,#1
#148 HALT

---Function  to calculate the product of two numbers!!
#152 ADDU R3,R1,R3          //R3 = R1+R3
#156 ADDI R2,R2,#-1         //R2 = R2-1;
#160 BNE R2,R6,#-3.
#164 ADDI R1,R3,#0.         //R1 = R3
#168 BEQ R1,R1,#-10.        //ALWAYS BRANCH TO #132!
```

In the setup phase, R6,R5,R4 and R1 are initialized to 1,0, M[20] and M[20] in the setup phase. In the 5th instruction, R3 is initialized to zero each time the control comes there. R2 and R5 acts as a counters which start from R2 = M[20] and R5 = 0 respectively and have their values decremented and incremented by 1,respectively in each iteration. This part is executed until R2 counts down to 1.
In the function call, #152-#156, R1 is added R2-1 times and then the product is put into R1 and the BEQ instruction transfers the control to line 132 where the same process is repeated again until value of R2 is reduced to 1.

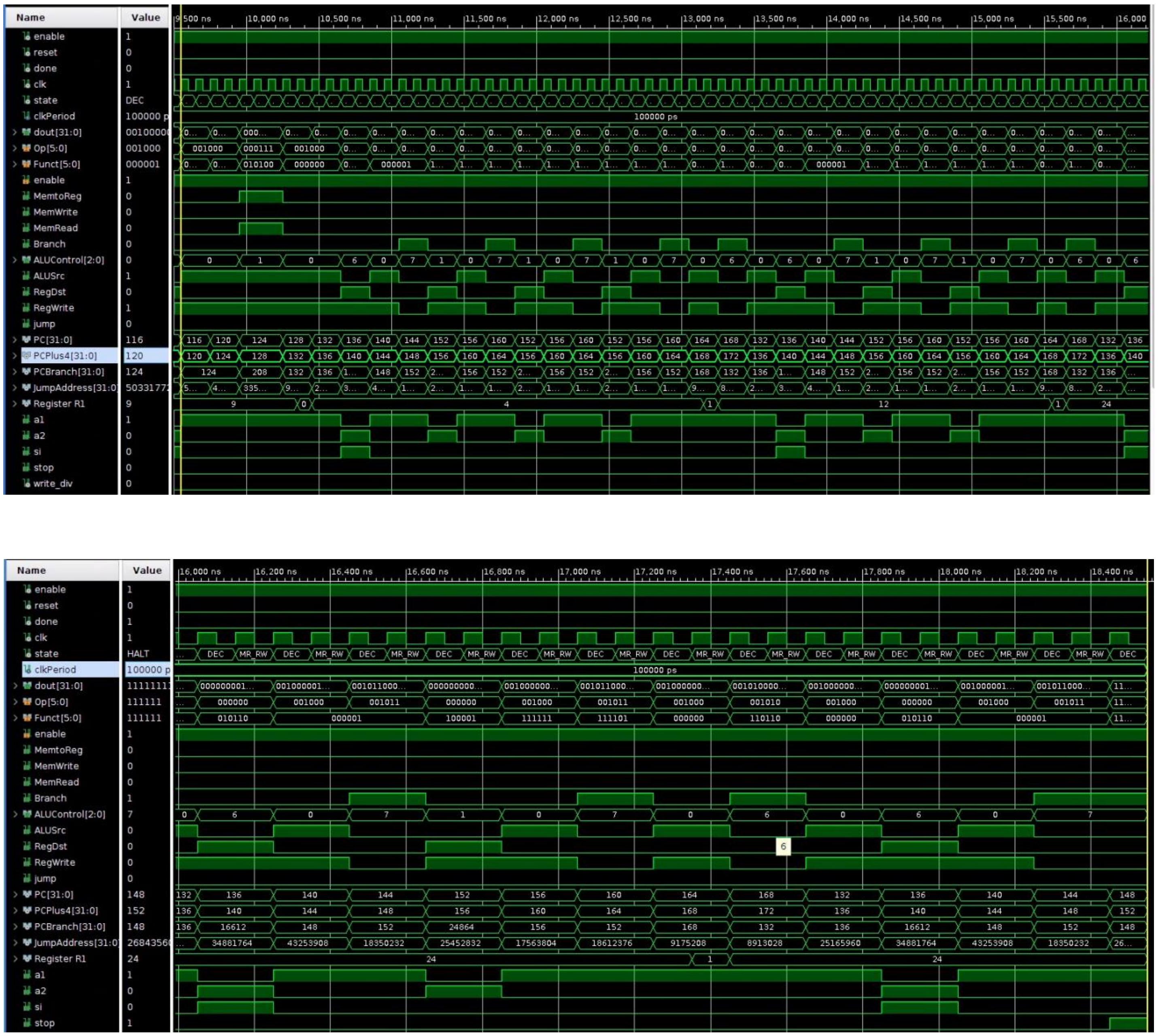
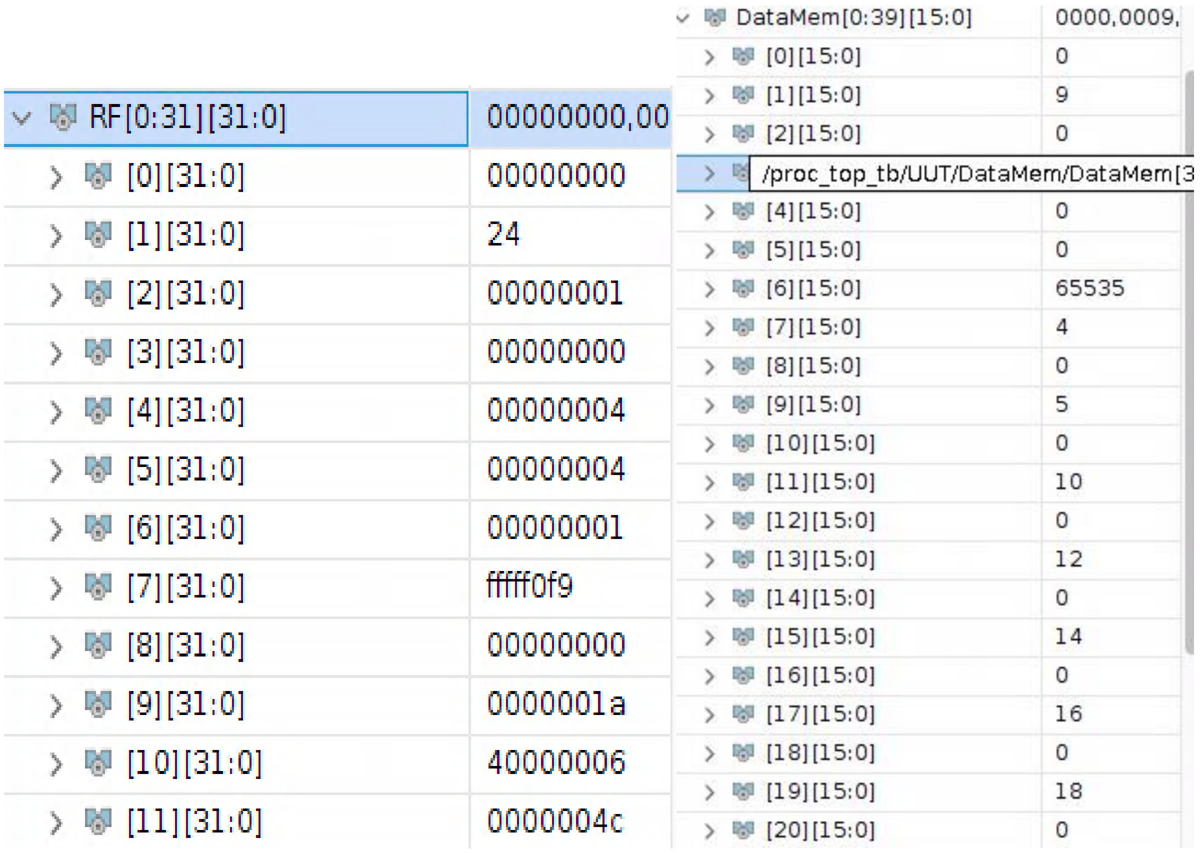


Figure 14:Waveform for execution of factorial program on NYU-6463



RF[0:31][31:0]	00000000,00
[0][31:0]	00000000
[1][31:0]	24
[2][31:0]	00000001
[3][31:0]	00000000
[4][31:0]	00000004
[5][31:0]	00000004
[6][31:0]	00000001
[7][31:0]	fffff0f9
[8][31:0]	00000000
[9][31:0]	0000001a
[10][31:0]	40000006
[11][31:0]	0000004c

DataMem[0:39][15:0]	0000,0009,
[0][15:0]	0
[1][15:0]	9
[2][15:0]	0
/proc_top_tb/UUT/DataMem/DataMem[3	
[4][15:0]	0
[5][15:0]	0
[6][15:0]	65535
[7][15:0]	4
[8][15:0]	0
[9][15:0]	5
[10][15:0]	0
[11][15:0]	10
[12][15:0]	0
[13][15:0]	12
[14][15:0]	0
[15][15:0]	14
[16][15:0]	0
[17][15:0]	16
[18][15:0]	0
[19][15:0]	18
[20][15:0]	0

Figure 15:Data memory(R[20] = 4) and Register file(R[1] = 24 the result of factorial program) after execution

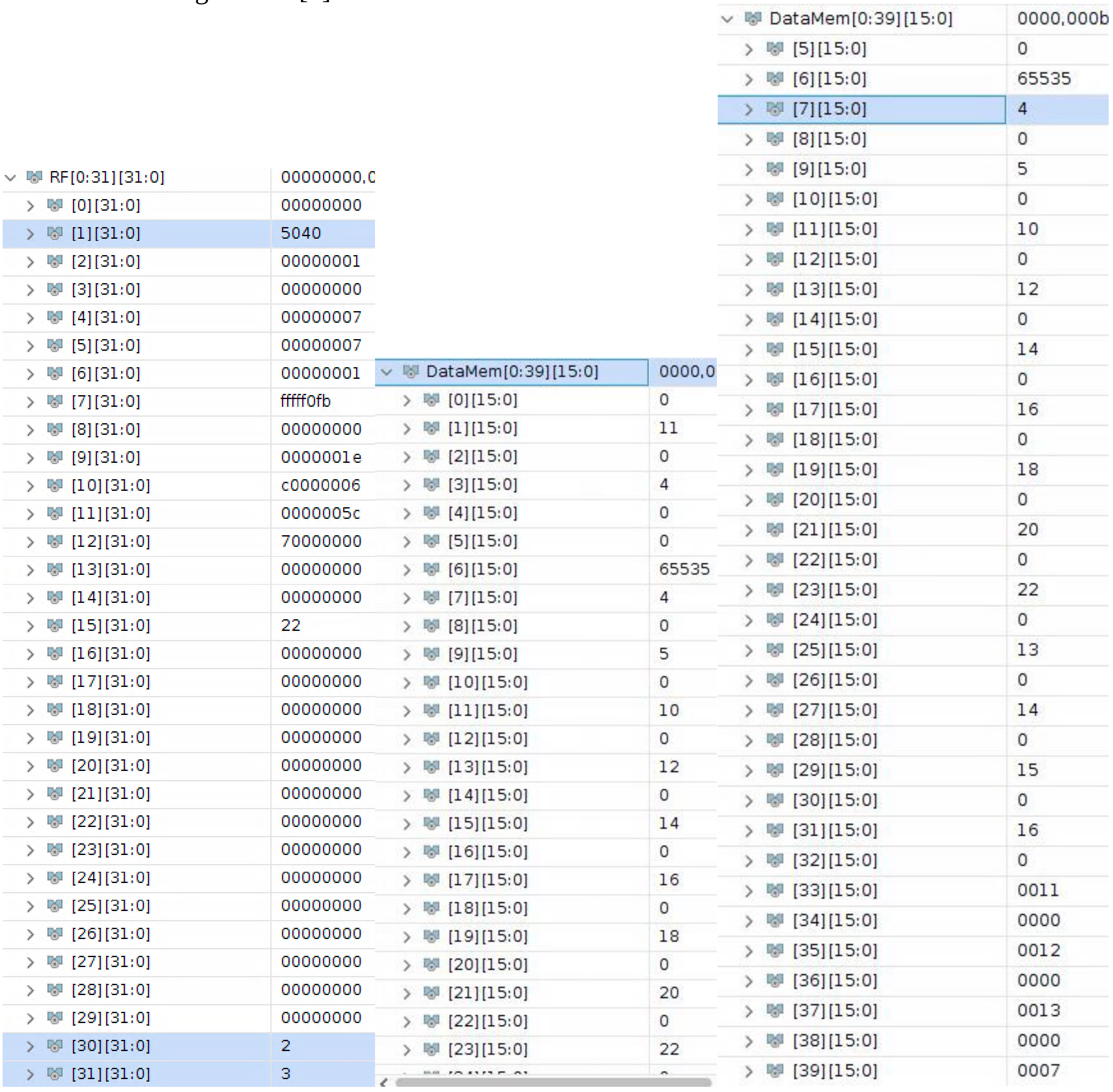
5.3 Test Cases

Test Case -I(Fig 11 and Fig 13)

M[0] and M[2] are picked up for division and execution of Simple program and M[20] is picked up for factorial program
Test case:
M[0] = 9
M[2] = 4
M[20] = 4(Picked up for factorial program)
Result:-
Simple program R30 = 2,R31 = 3 and M[10] to M[22]
Factorial program R[1] = 24

Test Case -II(Fig 14)

M[0] and M[2] are picked up for division and execution of Simple program
Test case:
M[0] = 11
M[2] = 4
M[39] = 7(Picked up for factorial program)
Result:-
Simple program R30 = 2,R31 = 3 and M[10] to M[22]
Factorial Program = R[1] = 5040



RF[0:31][31:0]	00000000,0
[0][31:0]	00000000
[1][31:0]	5040
[2][31:0]	00000001
[3][31:0]	00000000
[4][31:0]	00000007
[5][31:0]	00000007
[6][31:0]	00000001
[7][31:0]	fffff0fb
[8][31:0]	00000000
[9][31:0]	0000001e
[10][31:0]	c0000006
[11][31:0]	0000005c
[12][31:0]	70000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	22
[16][31:0]	00000000
[17][31:0]	00000000
[18][31:0]	00000000
[19][31:0]	00000000
[20][31:0]	00000000
[21][31:0]	00000000
[22][31:0]	00000000
[23][31:0]	00000000
[24][31:0]	00000000
[25][31:0]	00000000
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000
[30][31:0]	2
[31][31:0]	3

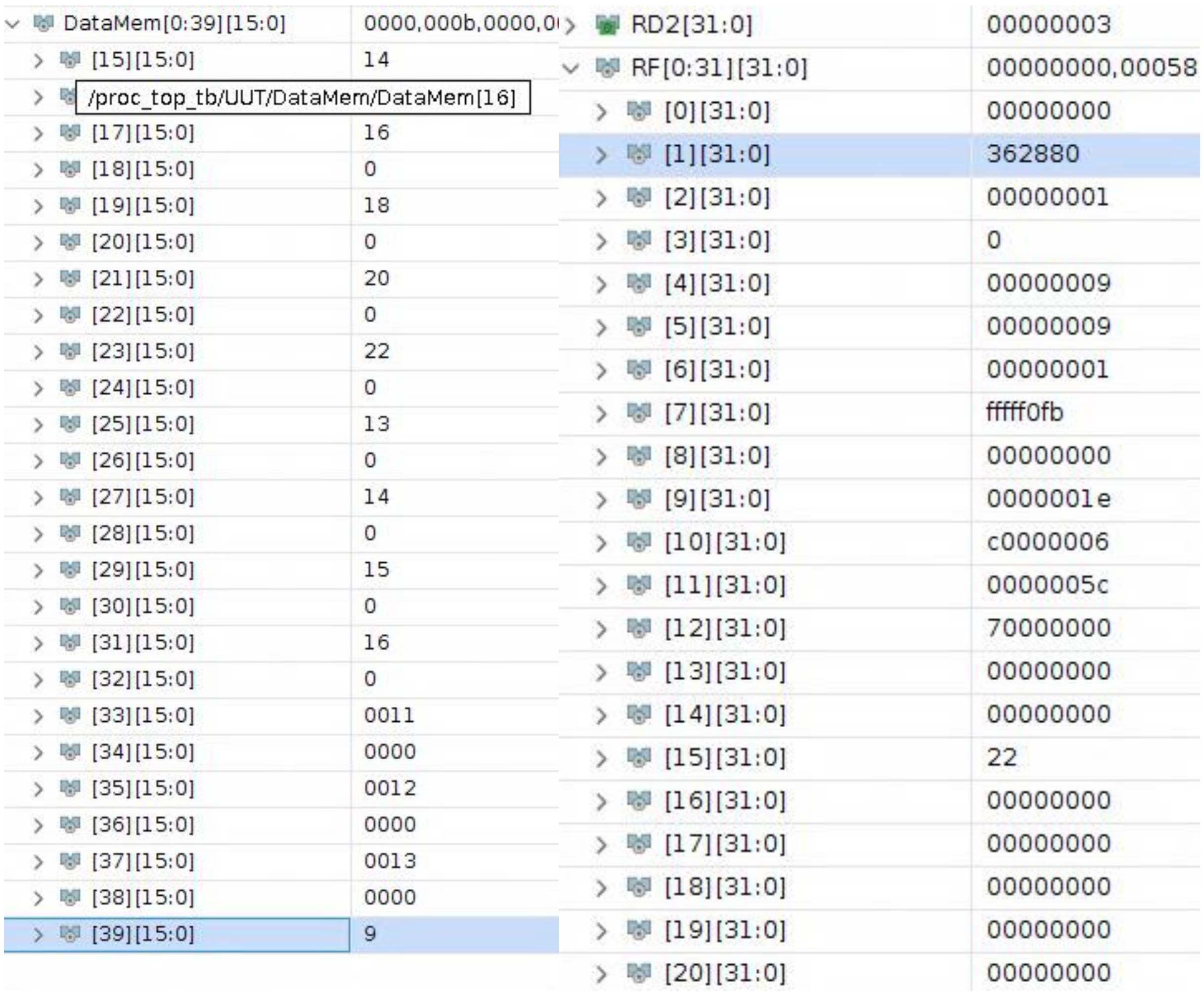
DataMem[0:39][15:0]	0000,0
[0][15:0]	0
[1][15:0]	11
[2][15:0]	0
[3][15:0]	4
[4][15:0]	0
[5][15:0]	0
[6][15:0]	65535
[7][15:0]	4
[8][15:0]	0
[9][15:0]	5
[10][15:0]	0
[11][15:0]	10
[12][15:0]	0
[13][15:0]	12
[14][15:0]	0
[15][15:0]	14
[16][15:0]	0
[17][15:0]	16
[18][15:0]	0
[19][15:0]	18
[20][15:0]	0
[21][15:0]	20
[22][15:0]	0
[23][15:0]	22

DataMem[0:39][15:0]	0000,000b
[5][15:0]	0
[6][15:0]	65535
[7][15:0]	4
[8][15:0]	0
[9][15:0]	5
[10][15:0]	0
[11][15:0]	10
[12][15:0]	0
[13][15:0]	12
[14][15:0]	0
[15][15:0]	14
[16][15:0]	0
[17][15:0]	16
[18][15:0]	0
[19][15:0]	18
[20][15:0]	0
[21][15:0]	20
[22][15:0]	0
[23][15:0]	22
[24][15:0]	0
[25][15:0]	13
[26][15:0]	0
[27][15:0]	14
[28][15:0]	0
[29][15:0]	15
[30][15:0]	0
[31][15:0]	16
[32][15:0]	0
[33][15:0]	0011
[34][15:0]	0000
[35][15:0]	0012
[36][15:0]	0000
[37][15:0]	0013
[38][15:0]	0000
[39][15:0]	0007

Figure 16: The state of Memory and the register file for test case II

Test Case -III (Fig 15 and Fig 16)

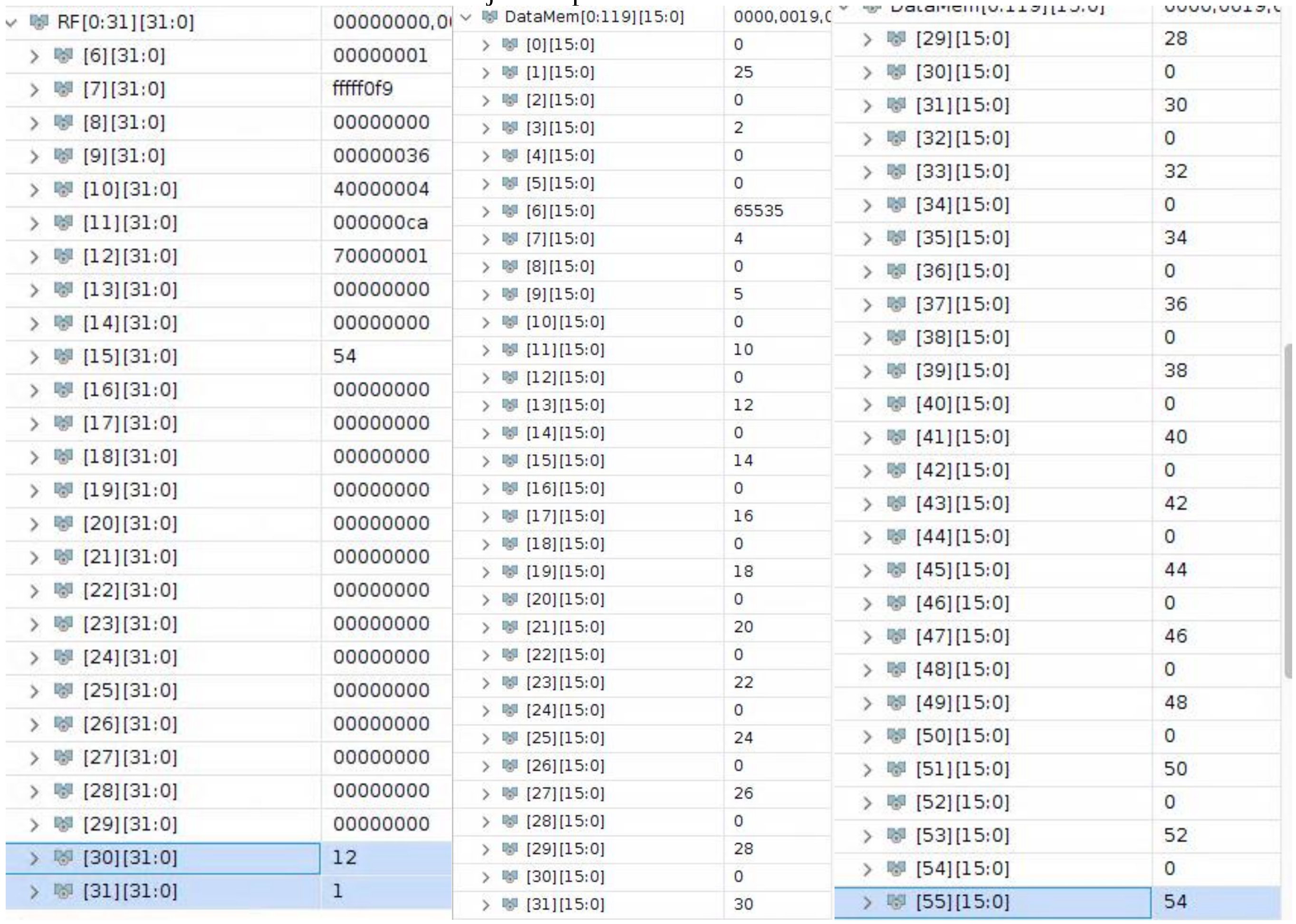
M[0] and M[2] are picked up for division and execution of Simple program
Test case:
M[0] = 25
M[2] = 2
M[39] = 9(Picked up for factorial program)
Result:-
Simple program R30 = 25,R31 = 1 and M[10] to M[22]
Factorial Program = R[1] = 3,62,880



DataMem[0:39][15:0]	0000,000b,0000,0
[15][15:0]	14
/proc_top_tb/UUT/DataMem/DataMem[16]	
[17][15:0]	16
[18][15:0]	0
[19][15:0]	18
[20][15:0]	0
[21][15:0]	20
[22][15:0]	0
[23][15:0]	22
[24][15:0]	0
[25][15:0]	13
[26][15:0]	0
[27][15:0]	14
[28][15:0]	0
[29][15:0]	15
[30][15:0]	0
[31][15:0]	16
[32][15:0]	0
[33][15:0]	0011
[34][15:0]	0000
[35][15:0]	0012
[36][15:0]	0000
[37][15:0]	0013
[38][15:0]	0000
[39][15:0]	9

RD2[31:0]	00000003
RF[0:31][31:0]	00000000,00058
[0][31:0]	00000000
[1][31:0]	362880
[2][31:0]	00000001
[3][31:0]	0
[4][31:0]	00000009
[5][31:0]	00000009
[6][31:0]	00000001
[7][31:0]	fffff0fb
[8][31:0]	00000000
[9][31:0]	0000001e
[10][31:0]	c0000006
[11][31:0]	0000005c
[12][31:0]	70000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	22
[16][31:0]	00000000
[17][31:0]	00000000
[18][31:0]	00000000
[19][31:0]	00000000
[20][31:0]	00000000

Figure 17:State of data memory and RF after execution of factorial program Test Case III!



RF[0:31][31:0]	00000000,0
[6][31:0]	00000001
[7][31:0]	fffff0f9
[8][31:0]	00000000
[9][31:0]	00000036
[10][31:0]	40000004
[11][31:0]	000000ca
[12][31:0]	70000001
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	54
[16][31:0]	00000000
[17][31:0]	00000000
[18][31:0]	00000000
[19][31:0]	00000000
[20][31:0]	00000000
[21][31:0]	00000000
[22][31:0]	00000000
[23][31:0]	00000000
[24][31:0]	00000000
[25][31:0]	00000000
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000
[30][31:0]	12
[31][31:0]	1

DataMem[0:119][15:0]	0000,0019,0
[0][15:0]	0
[1][15:0]	25
[2][15:0]	0
[3][15:0]	2
[4][15:0]	0
[5][15:0]	0
[6][15:0]	65535
[7][15:0]	4
[8][15:0]	0
[9][15:0]	5
[10][15:0]	0
[11][15:0]	10
[12][15:0]	0
[13][15:0]	12
[14][15:0]	0
[15][15:0]	14
[16][15:0]	0
[17][15:0]	16
[18][15:0]	0
[19][15:0]	18
[20][15:0]	0
[21][15:0]	20
[22][15:0]	0
[23][15:0]	22
[24][15:0]	0
[25][15:0]	24
[26][15:0]	0
[27][15:0]	26
[28][15:0]	0
[29][15:0]	28
[30][15:0]	0
[31][15:0]	30

DataMem[0:119][15:0]	0000,0019,0
[29][15:0]	28
[30][15:0]	0
[31][15:0]	30
[32][15:0]	0
[33][15:0]	32
[34][15:0]	0
[35][15:0]	34
[36][15:0]	0
[37][15:0]	36
[38][15:0]	0
[39][15:0]	38
[40][15:0]	0
[41][15:0]	40
[42][15:0]	0
[43][15:0]	42
[44][15:0]	0
[45][15:0]	44
[46][15:0]	0
[47][15:0]	46
[48][15:0]	0
[49][15:0]	48
[50][15:0]	0
[51][15:0]	50
[52][15:0]	0
[53][15:0]	52
[54][15:0]	0
[55][15:0]	54

Figure 18:The state of data memory and RF after execution of Simple program(Test case III)

6. Implementation of memory

The NYU-6463 has 3 memory elements i.e. **Instruction Memory, Register file and Data Memory.**

6.1 Instruction memory

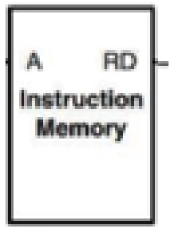
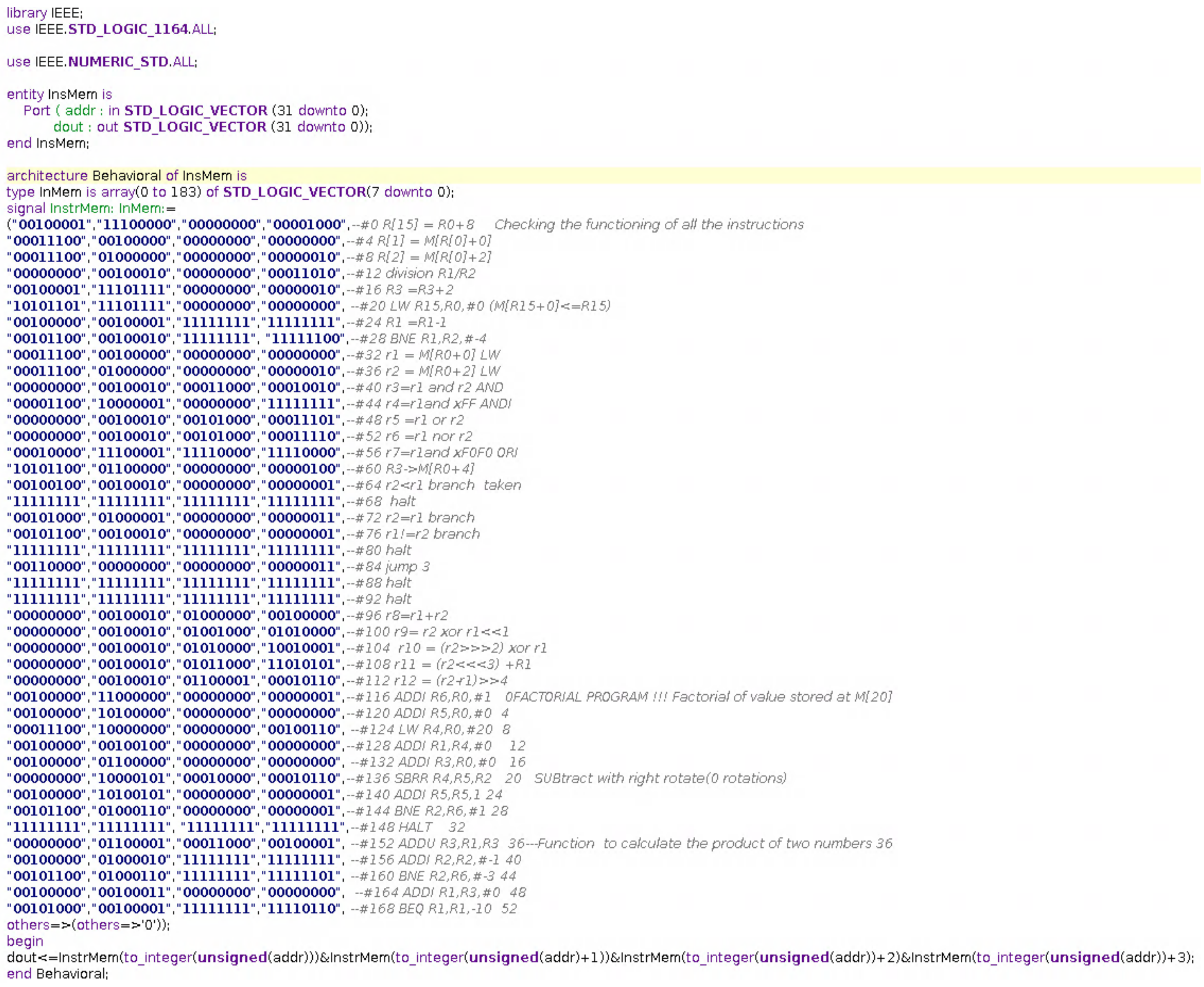


Figure 19: Block diagram of instruction memory

Interfaces:-
1)A – 32 bit address(i/p)
2)RD – 32-bit Instruction (o/p)



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity InsMem is
   Port ( addr : in STD_LOGIC_VECTOR (31 downto 0);
          dout : out STD_LOGIC_VECTOR (31 downto 0));
end InsMem;

architecture Behavioral of InsMem is
type InMem is array(0 to 183) of STD_LOGIC_VECTOR(7 downto 0);
signal InstrMem: InMem:=
("00100001","11100000","00000000","00001000",--#0 R[15] = R0+8     Checking the functioning of all the instructions
"00011100","00100000","00000000","00000000",--#4 R[1] = M[R[0]+0]
"00011100","01000000","00000000","00000010",--#8 R[2] = M[R[0]+2]
"00000000","00100010","00000000","00011010",--#12 division R1/R2
"00100001","11101111","00000000","00000010",--#16 R3 =R3+2
"10101101","11101111","00000000","00000000", --#20 LW R15,R0,#0 (M[R15+0]<=R15)
"00100000","00100001","11111111","11111111",--#24 R1 =R1-1
"00101100","00100010","11111111", "11111100",--#28 BNE R1,R2,#-4
"00011100","00100000","00000000","00000000",--#32 r1 = M[R0+0] LW
"00011100","01000000","00000000","00000010",--#36 r2 = M[R0+2] LW
"00000000","00100010","00011000","00010010",--#40 r3=r1 and r2 AND
"00001100","10000001","00000000","11111111",--#44 r4=r1and xFF ANDI
"00000000","00100010","00101000","00011101",--#48 r5 =r1 or r2
"00000000","00100010","00101000","00011110",--#52 r6 =r1 nor r2
"00010000","11100001","11110000","11110000",--#56 r7=r1and xF0F0 ORI
"10101100","01100000","00000000","00000100",--#60 R3->M[R0+4]
"00100100","00100010","00000000","00000001",--#64 r2<r1 branch  taken
"11111111","11111111","11111111","11111111",--#68  halt
"00101000","01000001","00000000","00000011",--#72 r2=r1 branch
"00101100","00100010","00000000","00000001",--#76 r1!=r2 branch
"11111111","11111111","11111111","11111111",--#80 halt
"00110000","00000000","00000000","00000011",--#84 jump 3
"11111111","11111111","11111111","11111111",--#88 halt
"11111111","11111111","11111111","11111111",--#92 halt
"00000000","00100010","01000000","00100000",--#96 r8=r1+r2
"00000000","00100010","01001000","01010000",--#100 r9= r2 xor r1<<1
"00000000","00100010","01010000","10010001",--#104  r10 = (r2>>>2) xor r1
"00000000","00100010","01011000","11010101",--#108 r11 = (r2<<<3) +R1
"00000000","00100010","01100001","00010110",--#112 r12 = (r2+r1)>>4
"00100000","11000000","00000000","00000001",--#116 ADDI R6,R0,#1   0FACTORIAL PROGRAM !!! Factorial of value stored at M[20]
"00100000","10100000","00000000","00000000",--#120 ADDI R5,R0,#0  4
"00011100","10000000","00000000","00100110", --#124 LW R4,R0,#20  8
"00100000","00100100","00000000","00000000",--#128 ADDI R1,R4,#0    12
"00100000","01100000","00000000","00000000", --#132 ADDI R3,R0,#0   16
"00000000","10000101","00010000","00010110",--#136 SBRR R4,R5,R2   20   SUBtract with right rotate(0 rotations)
"00100000","10100101","00000000","00000001",--#140 ADDI R5,R5,1 24
"00101100","01000110","00000000","00000001",--#144 BNE R2,R6,#1 28
"11111111","11111111", "11111111","11111111",--#148 HALT    32
"00000000","01100001","00011000","00100001", --#152 ADDU R3,R1,R3  36---Function  to calculate the product of two numbers 36
"00100000","01000010","11111111","11111111", --#156 ADDI R2,R2,#-1 40
"00101100","01000110","11111111","11111101", --#160 BNE R2,R6,#-3 44
"00100000","00100011","00000000","00000000",  --#164 ADDI R1,R3,#0  48
"00101000","00100001","11111111","11110110", --#168 BEQ R1,R1,-10  52
others=>(others=>'0'));
begin
dout<=InstrMem(to_integer(unsigned(addr)))&InstrMem(to_integer(unsigned(addr)+1))&InstrMem(to_integer(unsigned(addr))+2)&InstrMem(to_integer(unsigned(addr))+3);
end Behavioral;
```

Figure 20:Implementation of Instruction memory

The instruction memory has a data width of 8bits, therefore, for every address 4 locations are read since the instruction width of 32 bits.
For a given address **A** the locations A,A+1,A+2,A+3 are read and the 32-bit instruction is put on the output line.

6.2 Register File

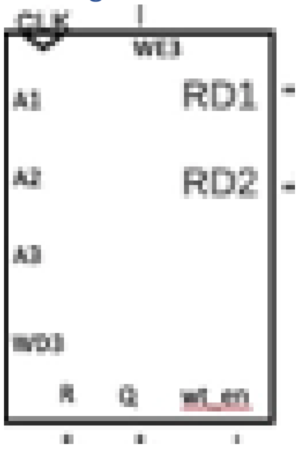
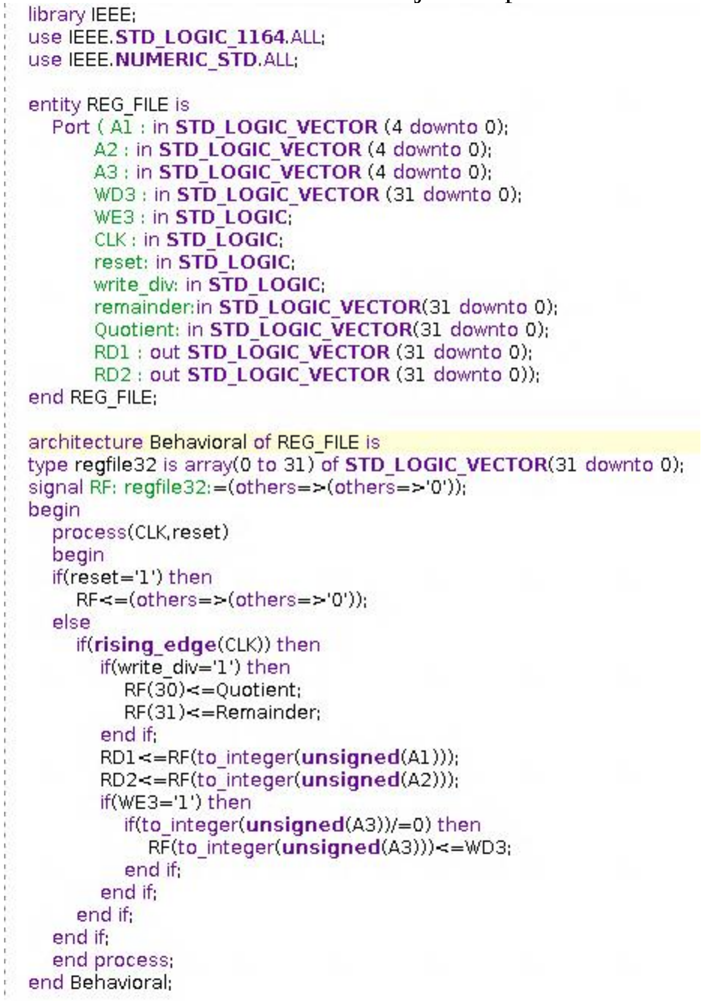


Figure 21:Block diagram of register file



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity REG_FILE is
   Port ( A1 : in STD_LOGIC_VECTOR (4 downto 0);
          A2 : in STD_LOGIC_VECTOR (4 downto 0);
          A3 : in STD_LOGIC_VECTOR (4 downto 0);
          WD3 : in STD_LOGIC_VECTOR (31 downto 0);
          WE3 : in STD_LOGIC;
          CLK : in STD_LOGIC;
          reset: in STD_LOGIC;
          write_div: in STD_LOGIC;
          remainder:in STD_LOGIC_VECTOR(31 downto 0);
          Quotient: in STD_LOGIC_VECTOR(31 downto 0);
          RD1 : out STD_LOGIC_VECTOR (31 downto 0);
          RD2 : out STD_LOGIC_VECTOR (31 downto 0));
end REG_FILE;

architecture Behavioral of REG_FILE is
type regfile32 is array(0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
signal RF: regfile32:=(others=>(others=>'0'));
begin
   process(CLK,reset)
   begin
   if(reset='1') then
      RF<=(others=>(others=>'0'));
   else
      if(rising_edge(CLK)) then
         if(write_div='1') then
            RF(30)<=Quotient;
            RF(31)<=Remainder;
         end if;
         RD1<=RF(to_integer(unsigned(A1)));
         RD2<=RF(to_integer(unsigned(A2)));
         if(WE3='1') then
            if(to_integer(unsigned(A3))/=0) then
               RF(to_integer(unsigned(A3)))<=WD3;
            end if;
         end if;
      end if;
   end if;
   end process;
end Behavioral;
```

Figure 22: Implementation of Register file in VHDL

Interfaces:-

a) A1:- 5-bit address field for reading the register Rs
b) A2:- 5-bit address field for reading the register Rt
c) A3:- 5-bit address field for supplying the address of the register to be written
d) WD3:- The 32-bit data to be written at address A3
e) R:- 32 bit remainder from the divider module to be written on R31
f) Q:- 32 bit quotient from the divider module to be written on R30
g) wt_en:- (1-bit) The value of R and Q are written on R31 and R30 when this signal is high
h) WE3:- (1-bit)The register at address A3 is written when this signal is high
i) RD1:- The 32-bit output read from the address A1
j) RD2:- The 32-bit output read from the address A2
k) CLK:- The clock signal

The width of each register is 32 bits and since there are 32 registers the width of the address field is 5 bits.
Whenever there is address present on A1 and A2 the data is read from the respective registers and put on RD1 and RD2 respectively

Similarly , whenever WE3 signal is high , the 32-bit data is written onto the register whose 5-bit address is presented on A3.

R and Q are directly wired to R31 and R30 directly, when the wt_en signal is high the data present on the R and Q line is written on to the two registers on subsequent rising edge.

All the operations are performed at rising edge of the clock.

6.3 Data Memory

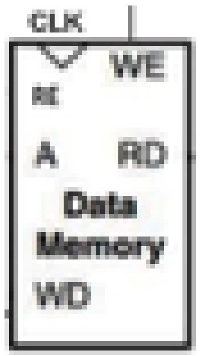
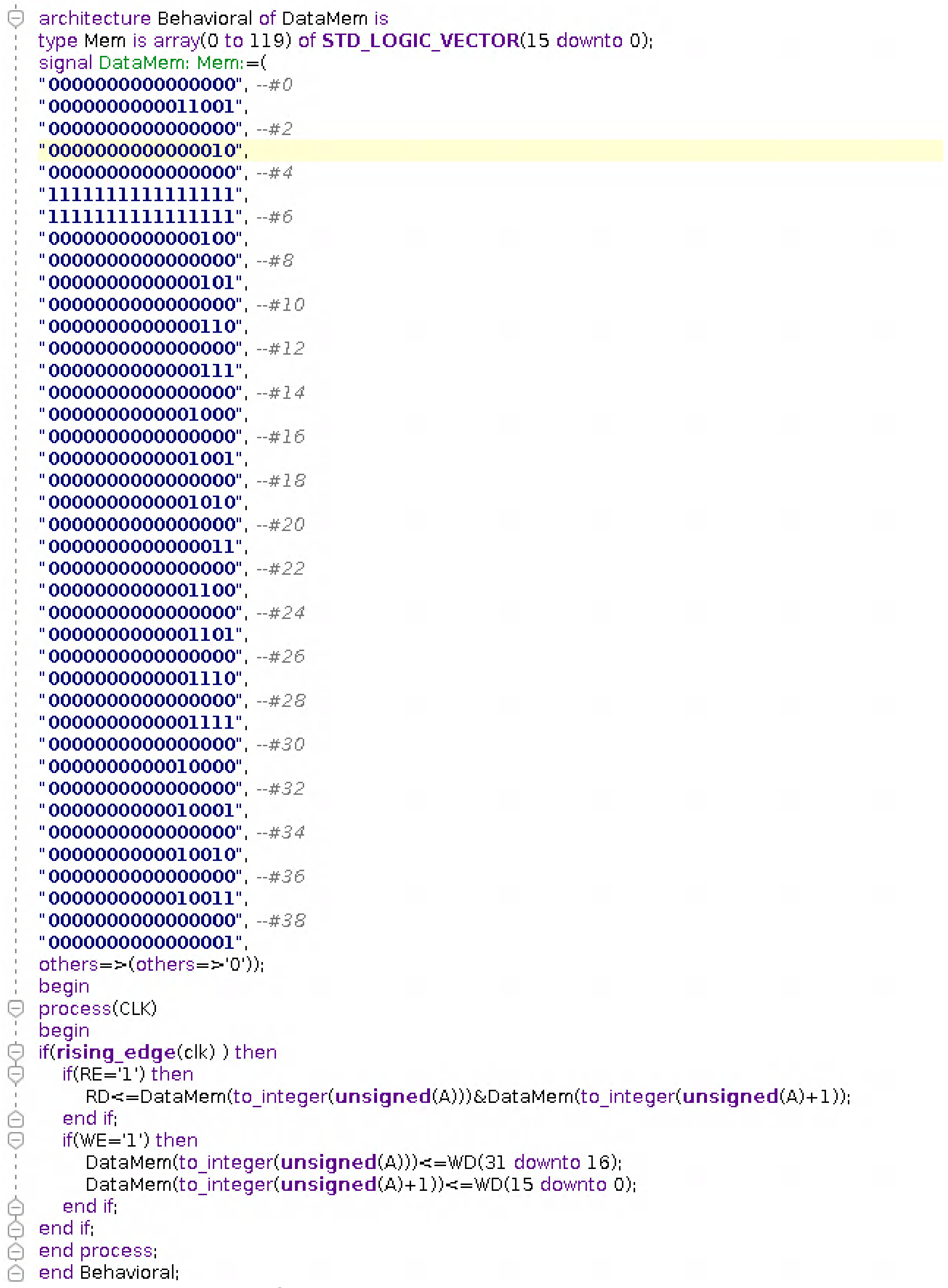


Figure 23: Block diagram of data memory

Interfaces:-

a) A:- 32 bit data address
b) WD:- 32-bit data to be written at address A
c) RD:- 32 bit data output read from the address A
d) WE:- The memory location A is written with WD when this signal is high

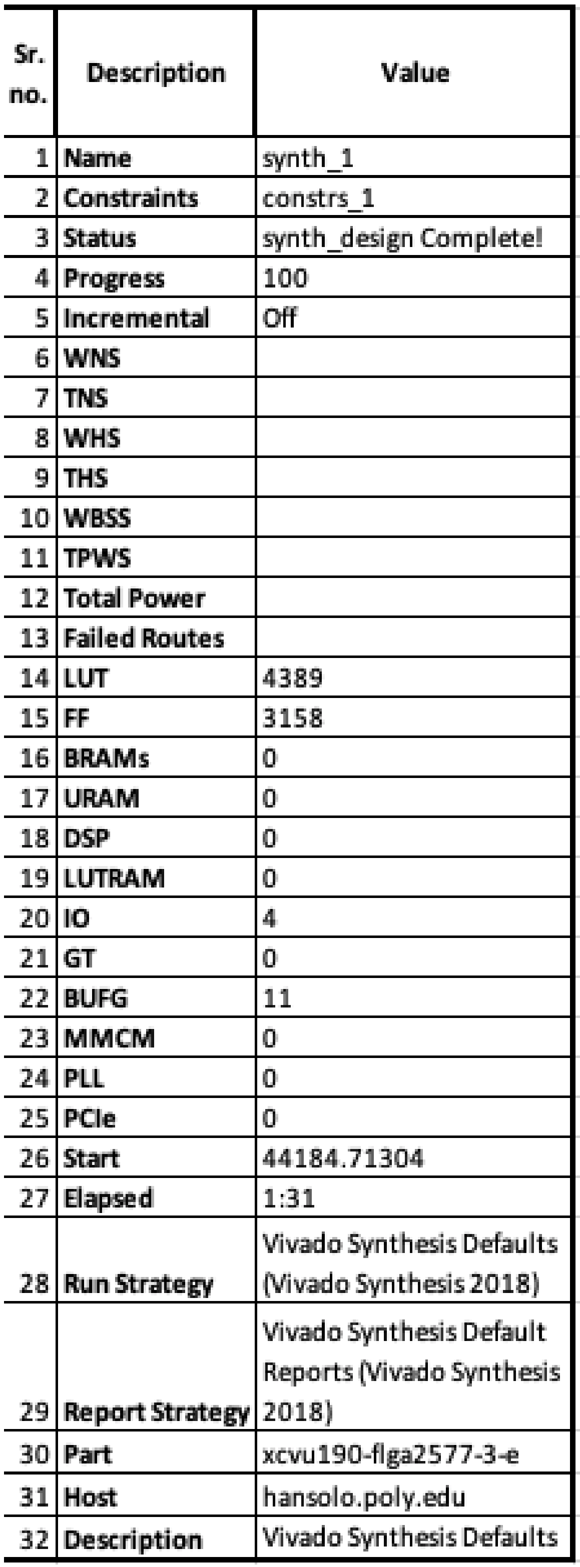


```vhdl
architecture Behavioral of DataMem is
type Mem is array(0 to 119) of STD_LOGIC_VECTOR(15 downto 0);
signal DataMem: Mem:=(
"0000000000000000", --#0
"0000000000011001",
"0000000000000000", --#2
"0000000000000010",
"0000000000000000", --#4
"1111111111111111",
"1111111111111111", --#6
"0000000000000100",
"0000000000000000", --#8
"0000000000000101",
"0000000000000000", --#10
"0000000000000110",
"0000000000000000", --#12
"0000000000000111",
"0000000000000000", --#14
"0000000000001000",
"0000000000000000", --#16
"0000000000001001",
"0000000000000000", --#18
"0000000000001010",
"0000000000000000", --#20
"0000000000000011",
"0000000000000000", --#22
"0000000000001100",
"0000000000000000", --#24
"0000000000001101",
"0000000000000000", --#26
"0000000000001110",
"0000000000000000", --#28
"0000000000001111",
"0000000000000000", --#30
"0000000000010000",
"0000000000000000", --#32
"0000000000010001",
"0000000000000000", --#34
"0000000000010010",
"0000000000000000", --#36
"0000000000010011",
"0000000000000000", --#38
"0000000000000001",
others=>(others=>'0'));
begin
process(CLK)
begin
if(rising_edge(clk) ) then
    if(RE='1') then
        RD<=DataMem(to_integer(unsigned(A)))&DataMem(to_integer(unsigned(A)+1));
    end if;
    if(WE='1') then
        DataMem(to_integer(unsigned(A)))<=WD(31 downto 16);
        DataMem(to_integer(unsigned(A)+1))<=WD(15 downto 0);
    end if;
end if;
end process;
end Behavioral;
```

Figure 24: Implementation of Data Memory in VHDL

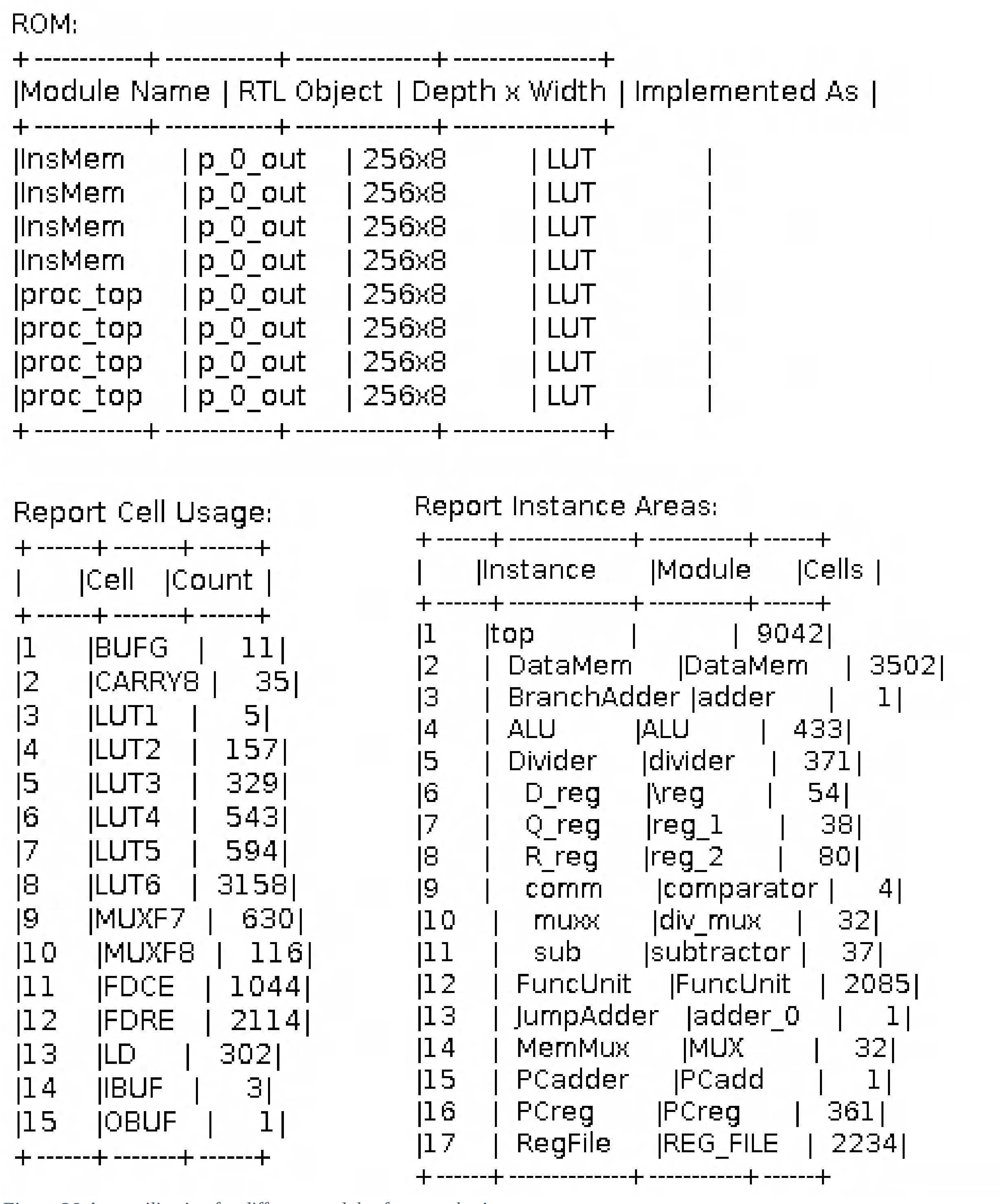
The instruction memory is 16 bit wide , when an address(5-bit) is presented at A , two data fields are read simultaneously and are concatenated and presented as a 32-bit data at the subsequent rising edge of the clock at presented at RD data line.
Similarly, when data is to be written at address A, the 32-bit data present at WD is split into 16-bit data fields and written at A and A+1 at the subsequent rising edge of the clock.

7. Area Utilization after design synthesis



Sr. no.	Description	Value
1	Name	synth_1
2	Constraints	constrs_1
3	Status	synth_design Complete!
4	Progress	100
5	Incremental	Off
6	WNS	
7	TNS	
8	WHS	
9	THS	
10	WBSS	
11	TPWS	
12	Total Power	
13	Failed Routes	
14	LUT	4389
15	FF	3158
16	BRAMs	0
17	URAM	0
18	DSP	0
19	LUTRAM	0
20	IO	4
21	GT	0
22	BUFG	11
23	MMCM	0
24	PLL	0
25	PCIe	0
26	Start	44184.71304
27	Elapsed	1:31
28	Run Strategy	Vivado Synthesis Defaults (Vivado Synthesis 2018)
29	Report Strategy	Vivado Synthesis Default Reports (Vivado Synthesis 2018)
30	Part	xcvu190-flga2577-3-e
31	Host	hansolo.poly.edu
32	Description	Vivado Synthesis Defaults

Figure 25:Table of resource utilization after synthesis



```
ROM:
+------------+------------+---------------+----------------+
|Module Name | RTL Object | Depth x Width | Implemented As |
+------------+------------+---------------+----------------+
|InsMem      | p_0_out    | 256x8         | LUT            |
|InsMem      | p_0_out    | 256x8         | LUT            |
|InsMem      | p_0_out    | 256x8         | LUT            |
|InsMem      | p_0_out    | 256x8         | LUT            |
|proc_top    | p_0_out    | 256x8         | LUT            |
|proc_top    | p_0_out    | 256x8         | LUT            |
|proc_top    | p_0_out    | 256x8         | LUT            |
|proc_top    | p_0_out    | 256x8         | LUT            |
+------------+------------+---------------+----------------+
```

```
Report Cell Usage:
+------+-------+------+
|      |Cell   |Count |
+------+-------+------+
|1     |BUFG   |    11|
|2     |CARRY8 |    35|
|3     |LUT1   |     5|
|4     |LUT2   |   157|
|5     |LUT3   |   329|
|6     |LUT4   |   543|
|7     |LUT5   |   594|
|8     |LUT6   |  3158|
|9     |MUXF7  |   630|
|10    |MUXF8  |   116|
|11    |FDCE   |  1044|
|12    |FDRE   |  2114|
|13    |LD     |   302|
|14    |IBUF   |     3|
|15    |OBUF   |     1|
+------+-------+------+
```

```
Report Instance Areas:
+------+-------------+-----------+------+
|      |Instance     |Module     |Cells |
+------+-------------+-----------+------+
|1     |top          |           |  9042|
|2     |  DataMem    |DataMem    |  3502|
|3     |  BranchAdder |adder     |     1|
|4     |  ALU        |ALU        |   433|
|5     |  Divider    |divider    |   371|
|6     |    D_reg    |\reg       |    54|
|7     |    Q_reg    |reg_1      |    38|
|8     |    R_reg    |reg_2      |    80|
|9     |    comm     |comparator |     4|
|10    |    muxx     |div_mux    |    32|
|11    |    sub      |subtractor |    37|
|12    |  FuncUnit   |FuncUnit   |  2085|
|13    |  JumpAdder  |adder_0    |     1|
|14    |  MemMux     |MUX        |    32|
|15    |  PCadder    |PCadd      |     1|
|16    |  PCreg      |PCreg      |   361|
|17    |  RegFile    |REG_FILE   |  2234|
+------+-------------+-----------+------+
```

Figure 26:Area utilization for different modules from synthesis report

The maximum utilization of area is by the following modules:-
1) Data Memory
2) FuncUnit
3) RegFile

As the size of the synthesized data memory is the maximum as the size of initialized memory is comparatively large (120x16 bits) i.e. 60 data memory locations.
The functional unit has a large amount of logic implemented into it for decoding the instruction and issuing the control signals therefore the utilization of resources is high for this module.

8. Timing

Cycle utilization of different type of instructions:-

1) **R-Type instructions:- 2 cycles**

   Exception:-

   a) **DIVU Instruction:-**

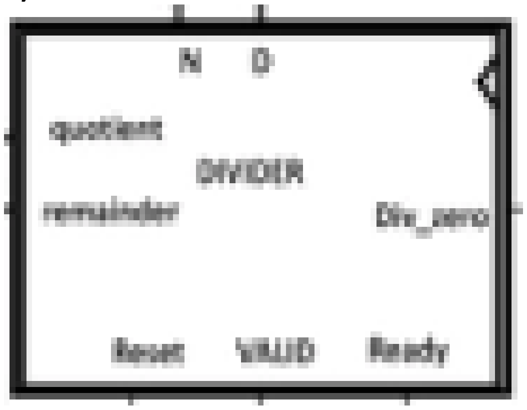


Figure 27:The block diagram of a divider module

As instructed the implemented divider is the one based on the instructions giver for **MIDTERM** exams. The same files used for midterm exams were included in the current project.
The divider is implemented using the **Euclidean algorithm**, therefore the number of cycles required to finish the algorithm depends upon the value of N and D.
The number of clock cycles required to complete the operation = **[3*(Quotient)+2] cycles**

2) **I-Type Instruction :- 2 cycles**

   Exception:-

   a) **Load Instruction**

      The load instruction takes 3 cycles to execute because it has three memory access which takes place at 3 consecutive rising clock edges.

3) **J-Type Instruction:- 2 cycles**

   The J-Type instruction takes place 2 cycles , in the first cycle the jump address is calculated and in the second cycles the PC is loaded with the jump address.

Thus, it can be seen the critical instruction which dominates the timing of the execution of the assembly level program is LOAD instruction and DIVU.

9. Observations and Scope for further improvements

1) The memory element like the data memory was implemented using LUTs instead of RAMs because it was sparse as indicated in the synthesize report.

2) In the simulation, the system can be made to trigger reading and writing memory at both the rising and the falling edge. Thus all the instructions can be made to execute all the instructions in 1 cycle except LOAD (2-cycles) and DIVU which will still be dependent upon the operands.
3) Such a design is not synthesizable thus the system was made to trigger just on rising edges.
4) In future, the design can be implemented with intermediatory buffers to implement a pipeline thus increasing the throughput of the processor.
5) Data memory and Register file can be triggered at different clock edges eg. Data memory being triggered at positive clock edge and data memory access triggered at falling edge which would help in executing the instructions in the same cycle.
6) Using a different algorithm for division module would help in reducing the critical path such that the cycles required for the execution of the operation doesn't depend on the input eg. Long division method.

10. Video Upload link

a) The required video explanation for the processor has been uploaded at link