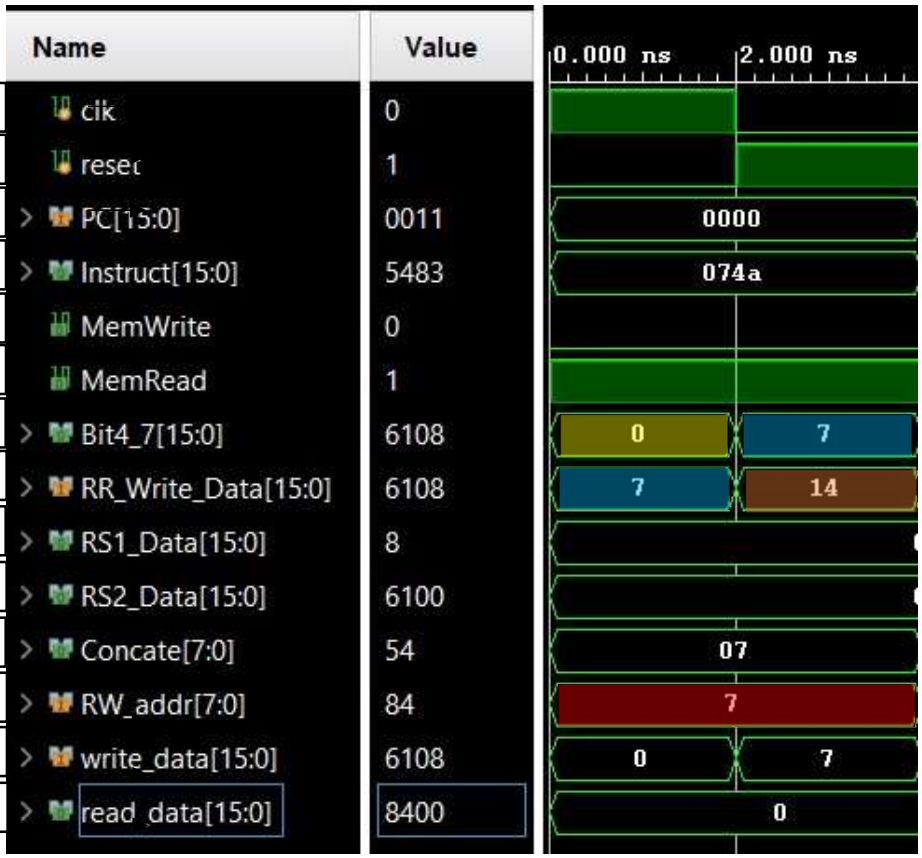
Karl Wegler

CPU Design: Part 5

Simulation Result Analysis

Clock Input
Reset Input
Current PC Address (1 < Instruction #)
Current Instruction Value in Hex
MemWrite Flag
MemRead Flag
Signed 16-Bit Data Value Currently in RR
Signed 16-Bit Data Being Written To RR
Signed 16-Bit Data Value Currently in RS1
Signed 16-Bit Data Value Currently in RS2
Concatenation of Instruction Bits [15:8]
RW_Address Used By DataMem for LW/SW
Signed 16-Bit Data Value Written to Mem.
Signed 16-Bit Data Value Read from Mem.



These are the values at the end of the program, disregard them in these images.

Current Signed 16-Bit Value Located in RR

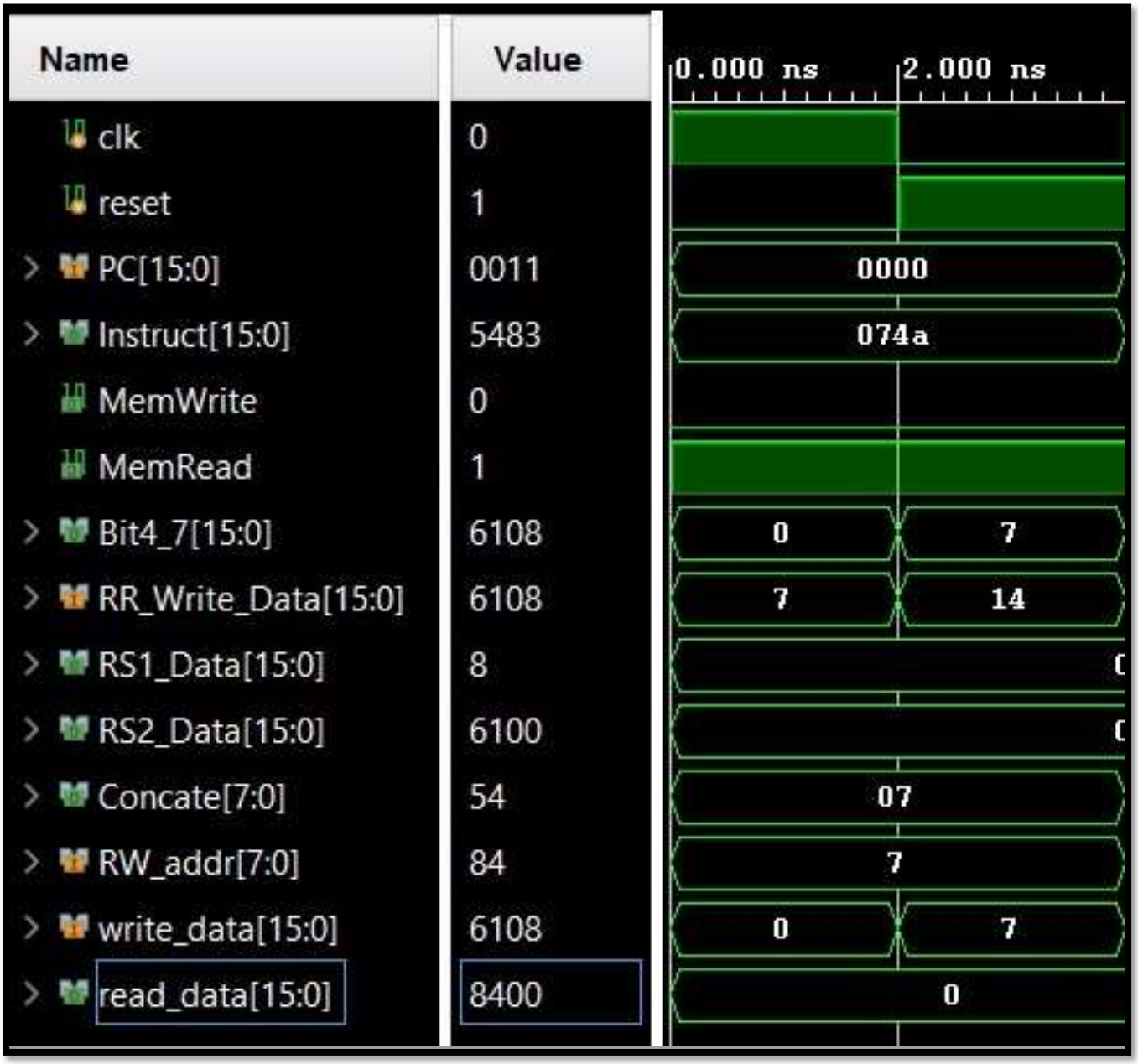
New Signed 16-Bit Value Located in RR

New Signed 16-Bit Value, multiplied by 2. However, this will not be assigned since it only is assigned on negedge of clock.

Same as Concatenate Value, but in Decimal Form

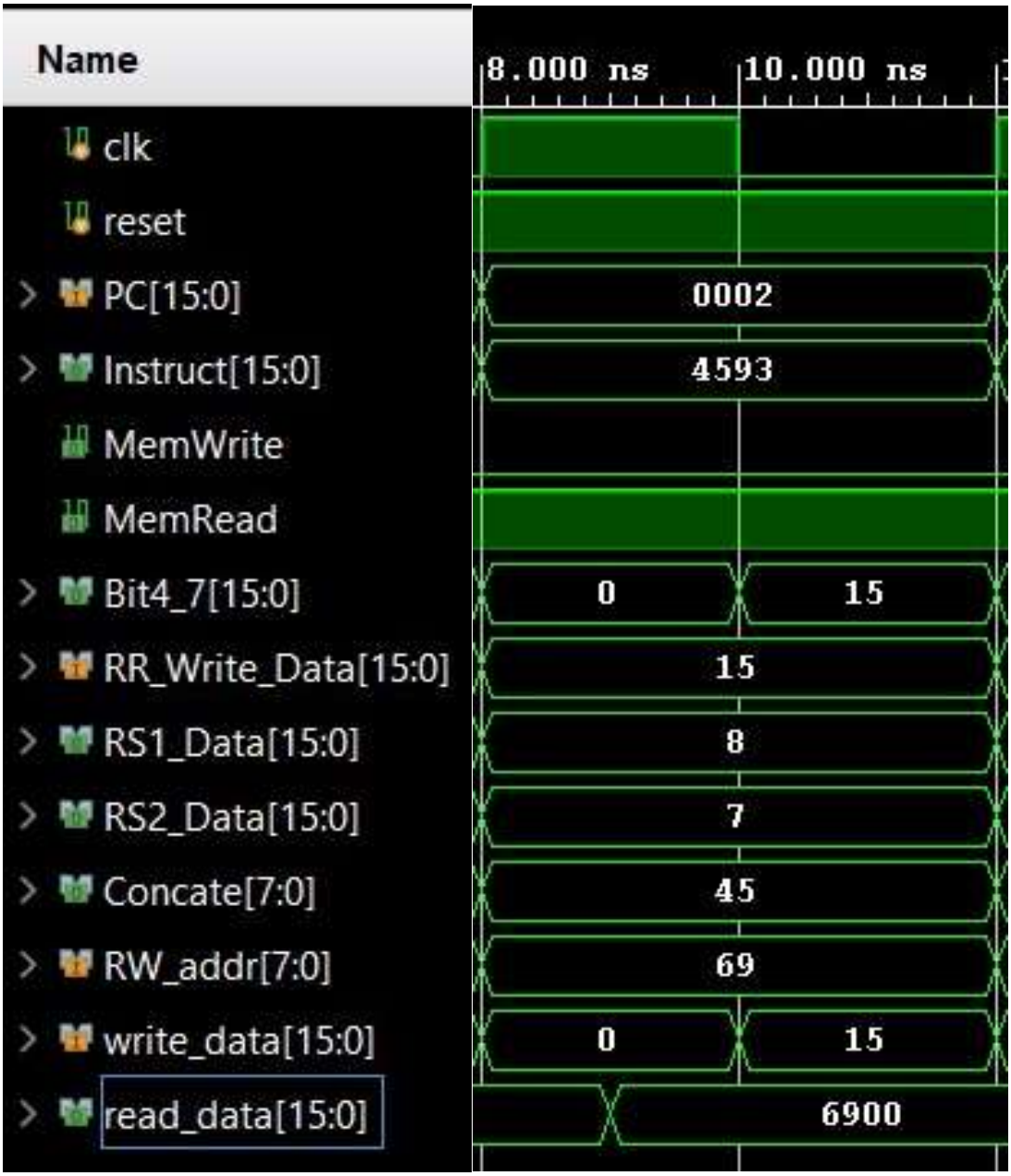
These are the values I have chosen to present in my Final Simulation Results. The code provided later in this document can be used to view every other wire utilized.

‘Add Immediate’ Instruction:



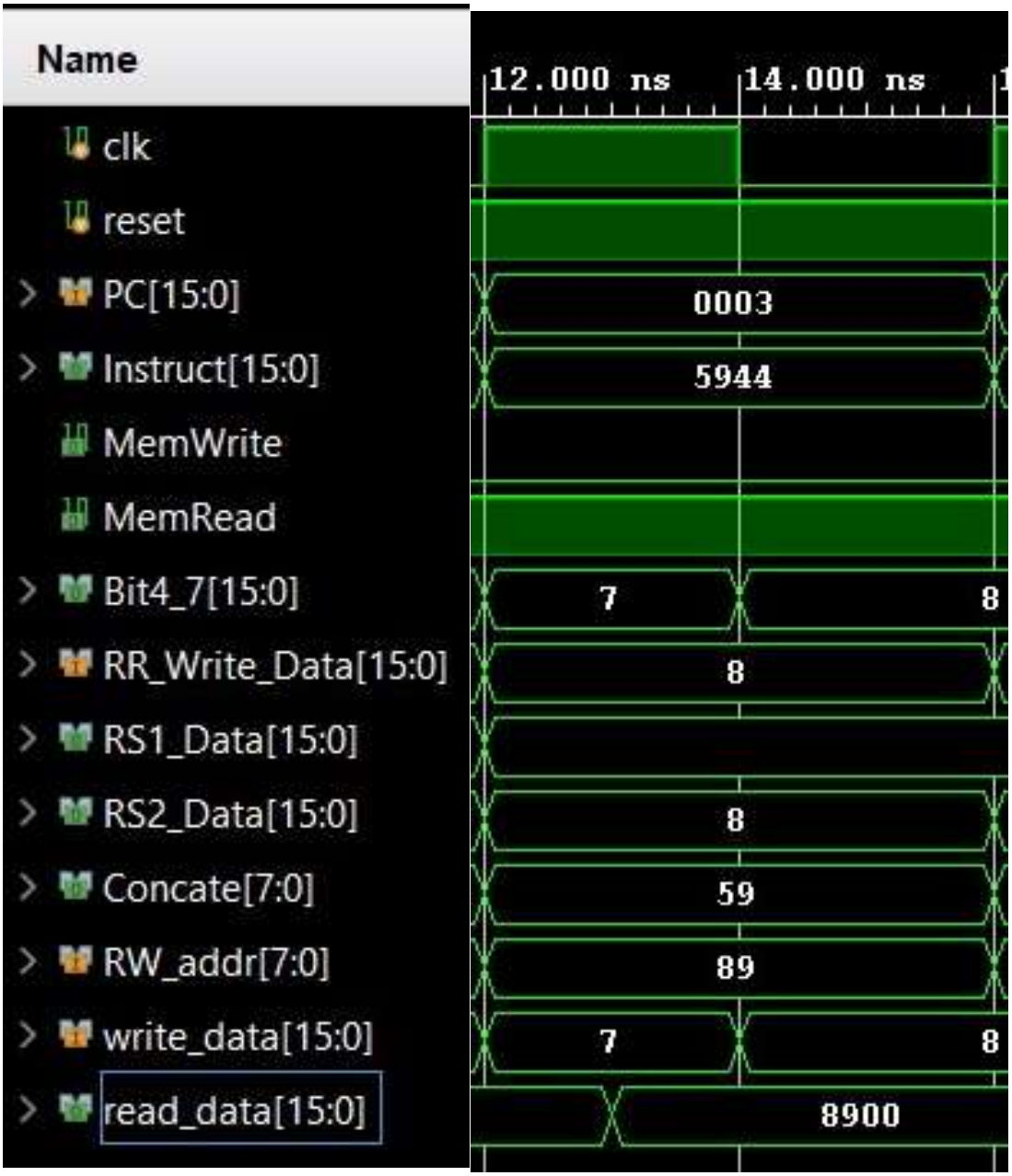
During the first 4ns of the above simulation, it is shown that an input instruction of ‘074A’ is received. The ‘A’ represents my opcode for an ADDI instruction. The ‘4’ is the RR (Return Register) where the immediate value will be stored after being added to the current ‘Bit4_7’ value ‘0000’. The immediate value in this case is ‘07’ or ‘7’ and was found through concatenation. This value is added to the current data value and writes a new ‘Bit4_7’ value of ‘0007’. The same process is implemented in Instruction 2 between 4-8ns.

‘ADD’ Instruction:



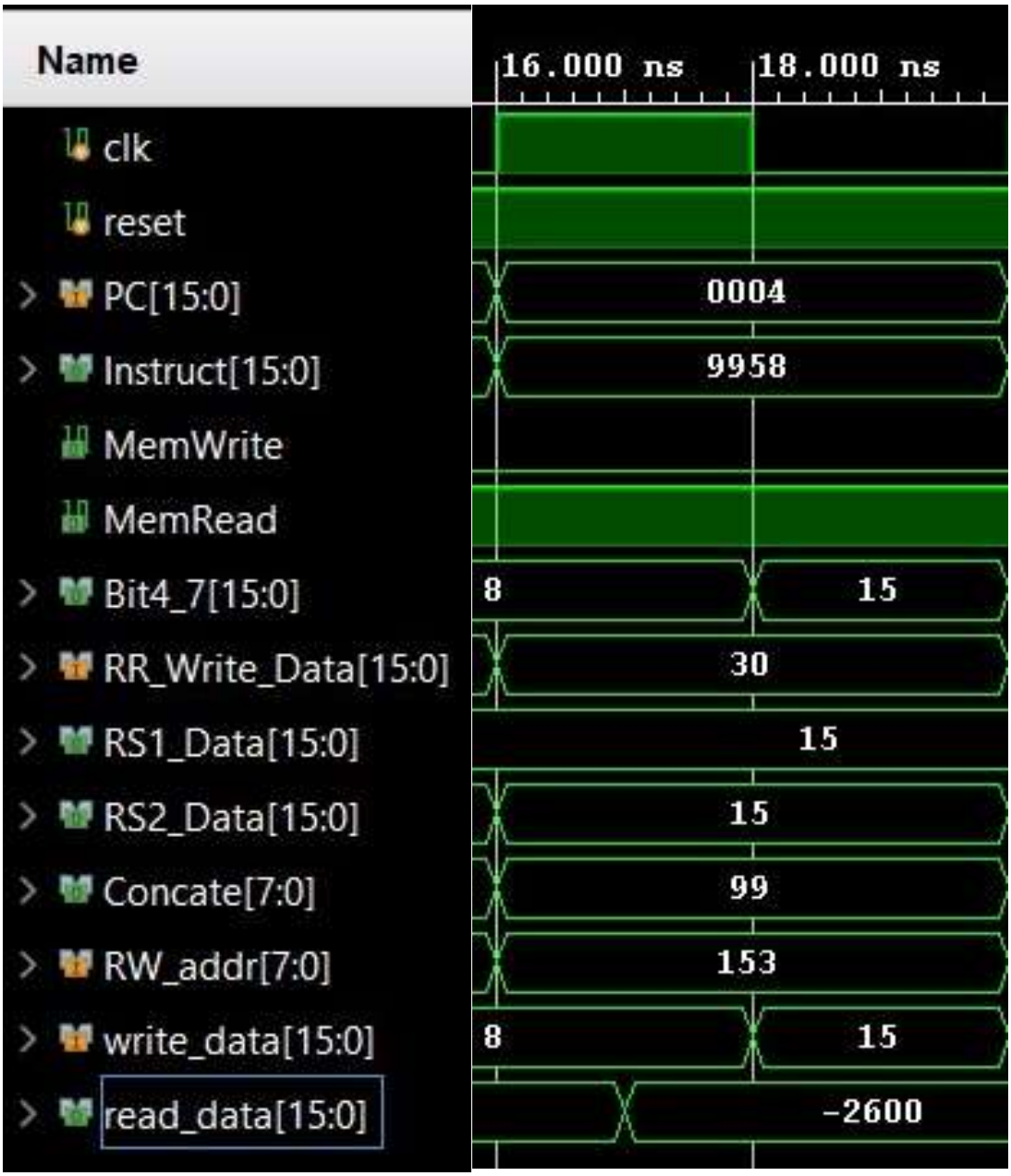
During the next 4ns of the simulation an ADD instruction is received. This is evident by the ‘Instruct’ of ‘4593’ where ‘3’ is the opcode representative of an ADD instruction. The ‘RR’ register in this instance is Register 9, and the values stored in RS1 and RS2 are added together via the ALU function. The ‘RS1_Data’ of Register 5 (RS1) is ‘0008’, and the ‘RS2_Data’ of Register 4 (RS2) is ‘0007’. This should give an ‘ALU_Out’ of ‘15’, or ‘000F’. This value is then written to Register 9 as can be seen by the negedge write on the ‘Bit4_7’ line.

‘AND’ Instruction:



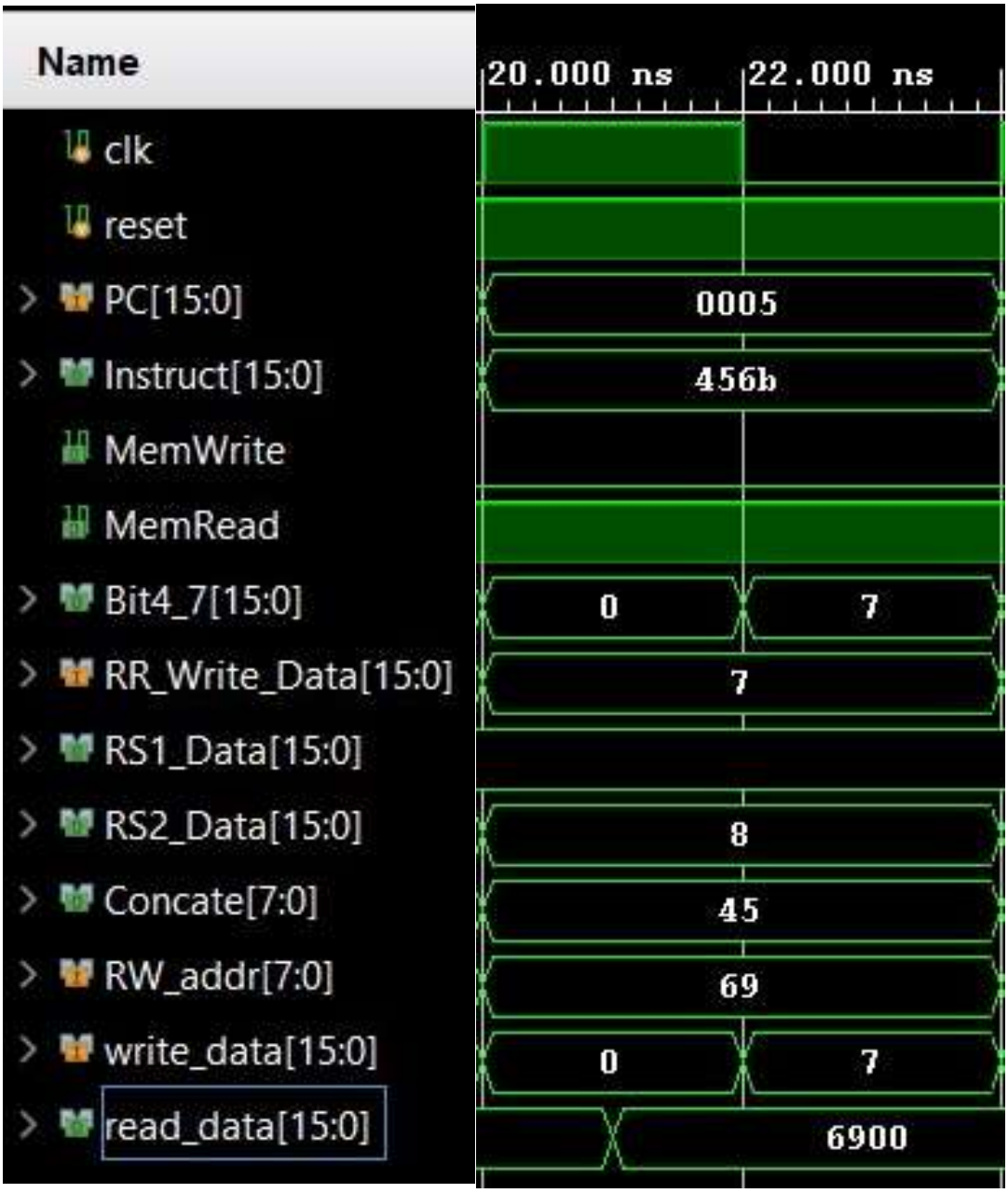
During the next 4ns of the simulation an AND instruciton is received. This is evident by the ‘Instruct’ of ‘5944’ where ‘4’ is the opcode representative of an AND instruciton. The ‘RR’ register in this instance is Register 4, and the values stored in RS1 and RS2 are compared using an AND function in the ALU. The ‘RS1_Data’ of Register 9 (RS1) is ‘000F’, and the ‘RS2_Data’ of Register 5 (RS2) is ‘0008’. This should give an ‘ALU_Out’ of ‘8’, or ‘0008’. This value is then written to Register 4 as can be seen by the negedge write on the ‘Bit4_7’ line.

‘MOV’ Instruction:



During the next 4ns of the simulation a MOV instruciton is received. This is evident by the ‘Instruct’ of ‘9958’ where ‘8’ is the opcode representative of a MOV instruciton. The ‘RR’ register in this instance is Register 5. The way a MOV instruction works is by ‘copying’ the data found in the RS1 and RS2 addresses. In this case, the ‘RS1_Data’ value ‘000F’ is copied from Register 9 (RS1) into the RR (Register 5). The ‘RS2_Data’ value is then copied into RS1. However, in this case RS1 and RS2 are the same so the value in RS1 does not change.

‘SUB’ Instruction:

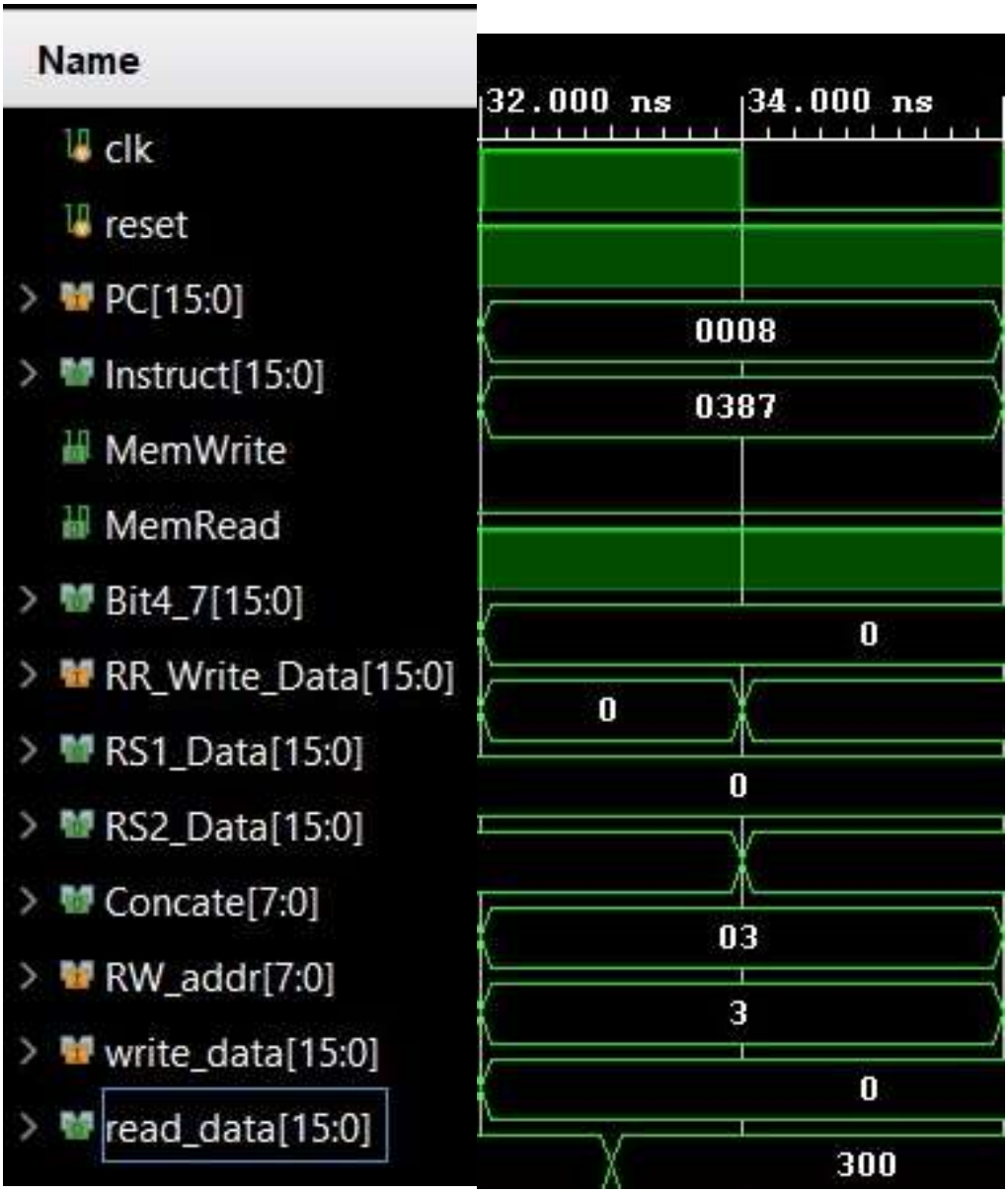


During the next 4ns of the simulation a SUB instruciton is received. This is evident by the ‘Instruct’ of ‘456B’ where ‘B’ is the opcode representative of a SUB instruciton. The ‘RR’ register in this instance is Register 6, and the values stored in RS1 and RS2 are subtracted from eachother via the ALU function. The ‘RS1_Data’ of Register 5 (RS1) is ‘000F’, and the ‘RS2_Data’ of Register 4 (RS2) is ‘0008’. This should give an ‘ALU_Out’ of ‘7’, or ‘0007’. This value is then written to Register 6 as can be seen by the negedge write on the ‘Bit4_7’ line.

‘OR’ Instruction:

During the next 4ns of the simulation an OR instruciton is received. This is evident by the ‘Instruct’ of ‘697C’ where ‘C’ is the opcode representative of a OR instruciton. The ‘RR’ register in this instance is Register 7, and the values stored in RS1 and RS2 are compared using a OR function in the ALU. The ‘RS1_Data’ of Register 9 (RS1) is ‘000F’, and the ‘RS2_Data’ of Register 6 (RS2) is ‘0007’. This should give an ‘ALU_Out’ of ‘15’, or ‘000F’. This value is then written to Register 7 as can be seen by the negedge write on the ‘Bit4_7’ line.

‘BM’ Instruction:



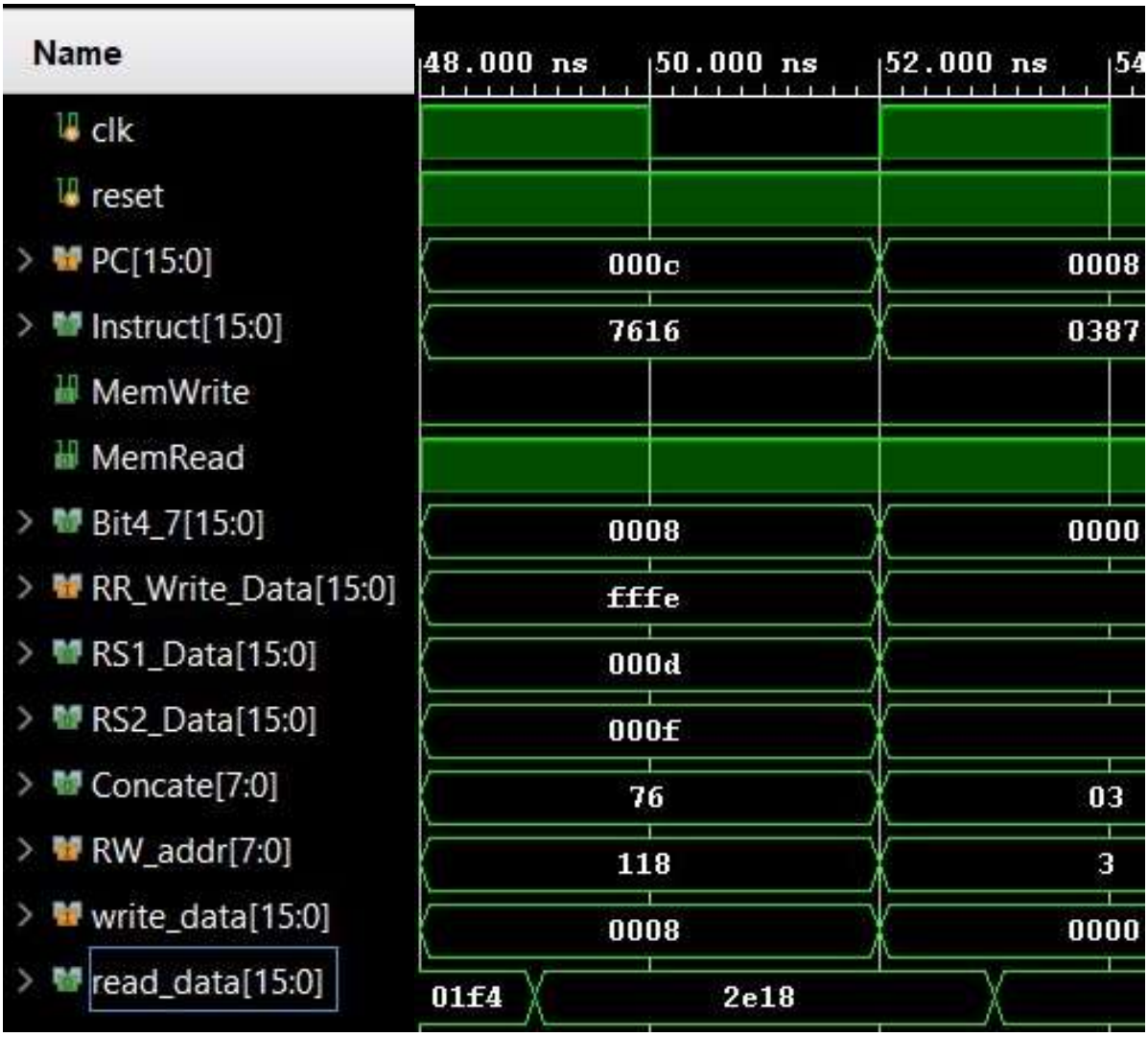
Now jumping to the 32ns mark where a BM instruction is received. This is evident by the ‘Instruct’ of ‘0387’ where ‘7’ is the opcode representative of a BM instruciton. The ‘RR’ register in this instance is actually **NOT** Register 8. In a BM instruction, only the 2 most LSB of the 3rd Byte are utilized to denote the RR; which in this case is Register 0. In this case, the binary 0000001110 00, is recied (excluding the opcode). The binary value on the left is equal to ‘14’ or ‘0E’ and is the value which gets stored within Register 0. This can be seen in ‘Reg0_3_Data’ where ‘000E’ is stored as the *Bookmark* address in Register 0. This operation is performed again at the 90ns mark for Register 1 holding another location.

‘LTE’ Instruction: (Branch Instructions)

At the 40ns mark (Instruction 11), a Less Than or Equal (LTE) Instruciton is received as denoted by ‘Instruct’ value ‘460D’ where ‘D’ is the opcode representative of LTE. In this case, the 3rd Byte, Register ‘0’ is used since it holds the memory location we want to jump to should this comparison be false. In this instance, the ‘RS1_Data’ value ‘0008’ is compared against ‘RS2_Data’ value ‘0008’ to see if ‘RS1_Data’ is Less Than or Equal to it. In this case, the statement proves to be true and the PC iterates by one and continues to the next instruction.

A separate module/component is used to compare the ‘ALU_Out’ value against the expected output of the requested branch instruction. In this case, ‘0000’ is ‘equal’ which proves to be TRUE.

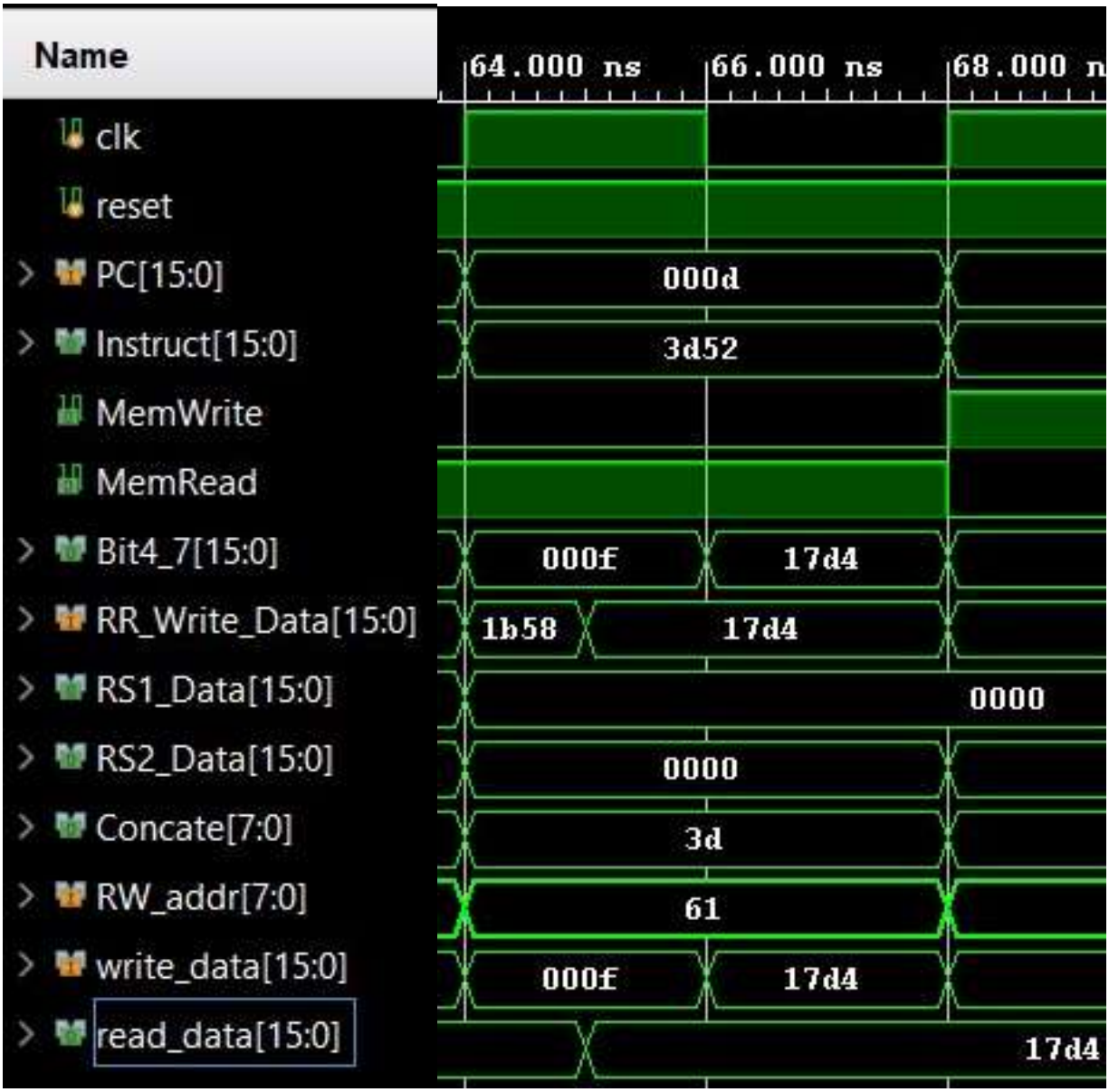
‘GT’ Instruction: (Branch Instructions)



At the 48ns mark (Instruction 13), a Greater Than (GT) Instruciton is received as denoted by ‘Instruct’ value ‘7616’ where ‘6’ is the opcode representative of GT. In this case, the 3rd Byte, Register ‘1’ is used since it holds the memory location we want to jump to should this comparison be false. In this instance, the ‘RS1_Data’ value ‘000D’ is compared against ‘RS2_Data’ value ‘000F’ to see if ‘RS1_Data’ is Greater Than it. In this case, the statement proves to be false and the PC jumps to the memory address stored in Register ‘1’ as denoted by the ‘Bit4_7’ value ‘0008’.

Due to the comparison being false, the PC jumps to 0008. This can be seen in the next ‘PC’ where the value should have been ‘000D’ had the comparison been true. Eventually when this check is performed again it will be true and pass through.

‘Load Word’ Instruction:



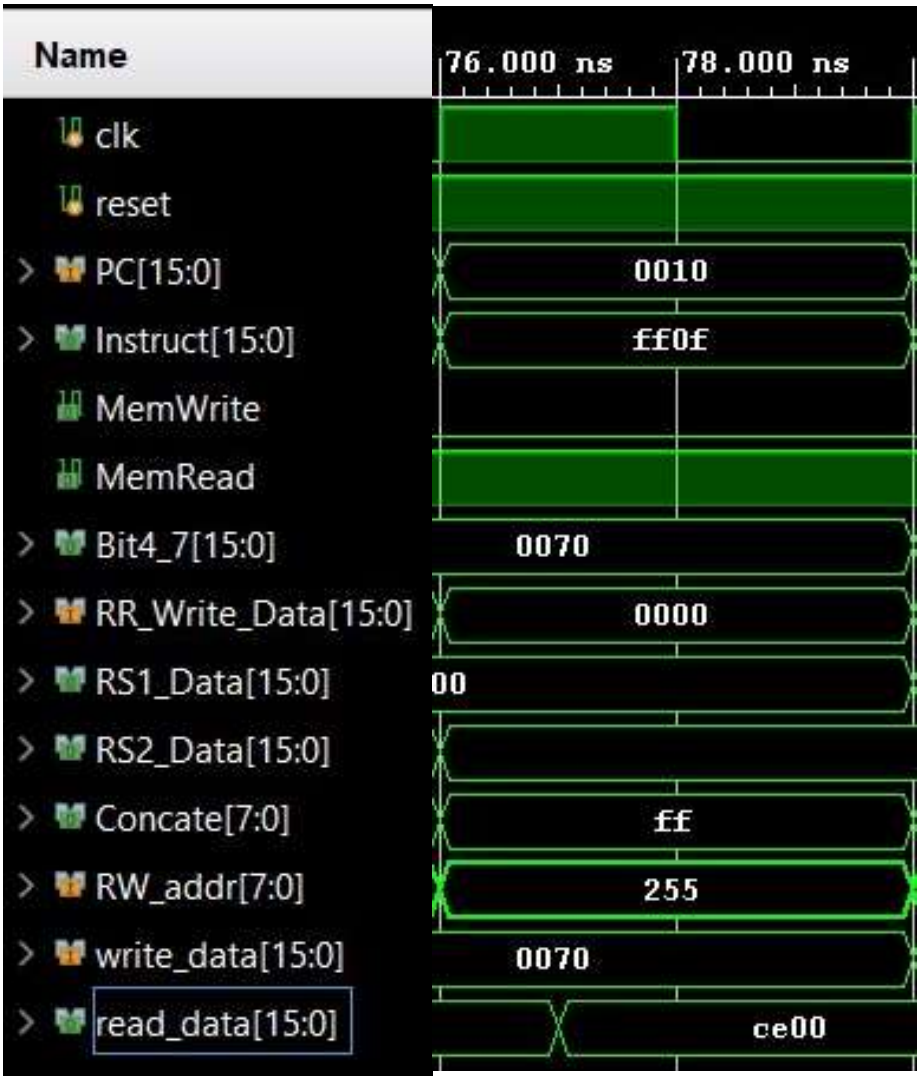
At the 64ns mark (Instruction 14), a Load Word (LW) Instruciton is received as denoted by ‘Instruct’ value ‘3D52’ where ‘2’ is the opcode representative of LW. The ‘LW’ instruction utilizes the ‘RW_addr’ value as the address in Data Memory to be read – “loaded” – into RR. In this case, the data value at Memory Address ‘61’ gets loaded into RR. This can be seen by the ‘read_data’ value changing to ‘17D4’ and the ‘Bit4_7’ value (RR) changing to match.

Note Memory Address [0 – 127] are equal to “Addr. * 100”. So the value found at Memory Address 61 is 6100, or 17D4. [128 - 255] = “(Addr. – 127) * (-100)”

‘Store Word’ Instruction:

At the 68ns mark (Instruction 15), a Store Word (SW) Instruciton is received as denoted by ‘Instruct’ value ‘0891’ where ‘1’ is the opcode representative of SW. The ‘SW’ instruction utilizes the ‘RW_addr’ value as the address in Data Memory to be written – “stored” – into. In this case, the data value within the RR gets loaded into Data Memory Address ‘8’. This can be seen by the ‘write_data’ value containing to ‘000F’ which is equal to the ‘Bit4_7’ value (RR).

‘EE’ Instruction: (Branch Instructions)



At the 76ns mark, an Equal (EE) Instruciton is received as denoted by ‘Instruct’ value ‘FF0F’ where ‘F’ is the opcode representative of EE. In this case, the 3rd Byte, Register ‘0’ is used since it holds the memory location we want to jump to should this comparison be false. In this instance, the ‘RS1_Data’ value ‘0000’ is compared against ‘RS2_Data’ value ‘0000’ to see if ‘RS1_Data’ is Equal to ‘RS2_Data’. In this case, the statement proves to be true and the PC continues on to the next instruction.

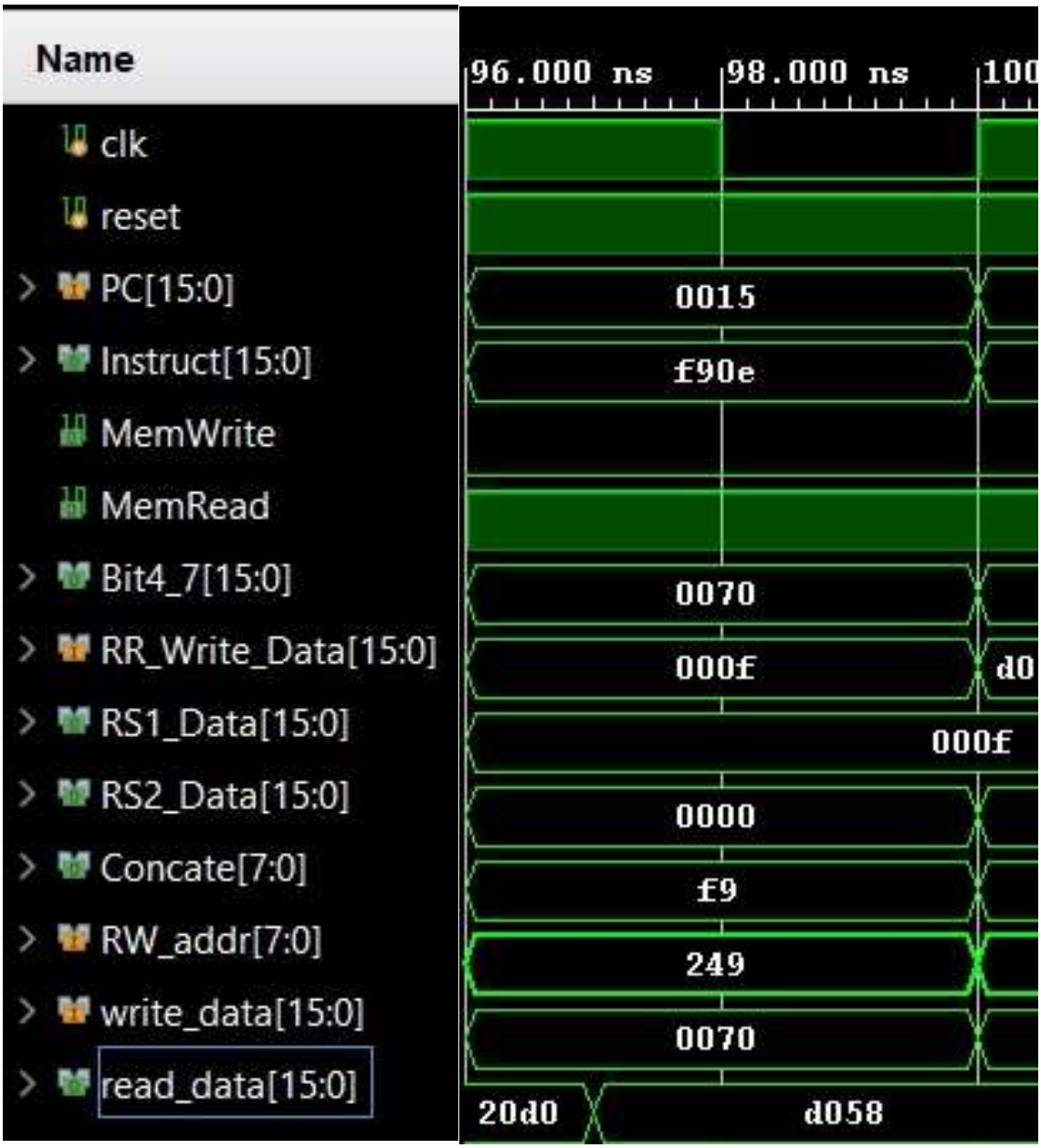
Doing an “EE” comparison with the same registers will of course prove true. Especially since the registers used in this case are R15, which is hard-wired to ground/zero.

‘NE’ Instruction: (Branch Instructions)

At the 88ns mark, a Not Equal (NE) Instruciton is received as denoted by ‘Instruct’ value ‘CF09’ where ‘9’ is the opcode representative of NE. In this case, the 3rd Byte, Register ‘0’ is used since it holds the memory location we want to jump to should this comparison be false. In this instance, the ‘RS1_Data’ value ‘0000’ is compared against ‘RS2_Data’ value ‘FF9C’ to see if ‘RS1_Data’ is Not Equal to ‘RS2_Data’. In this case, the statement proves to be true and the PC continues on to the next instruction.

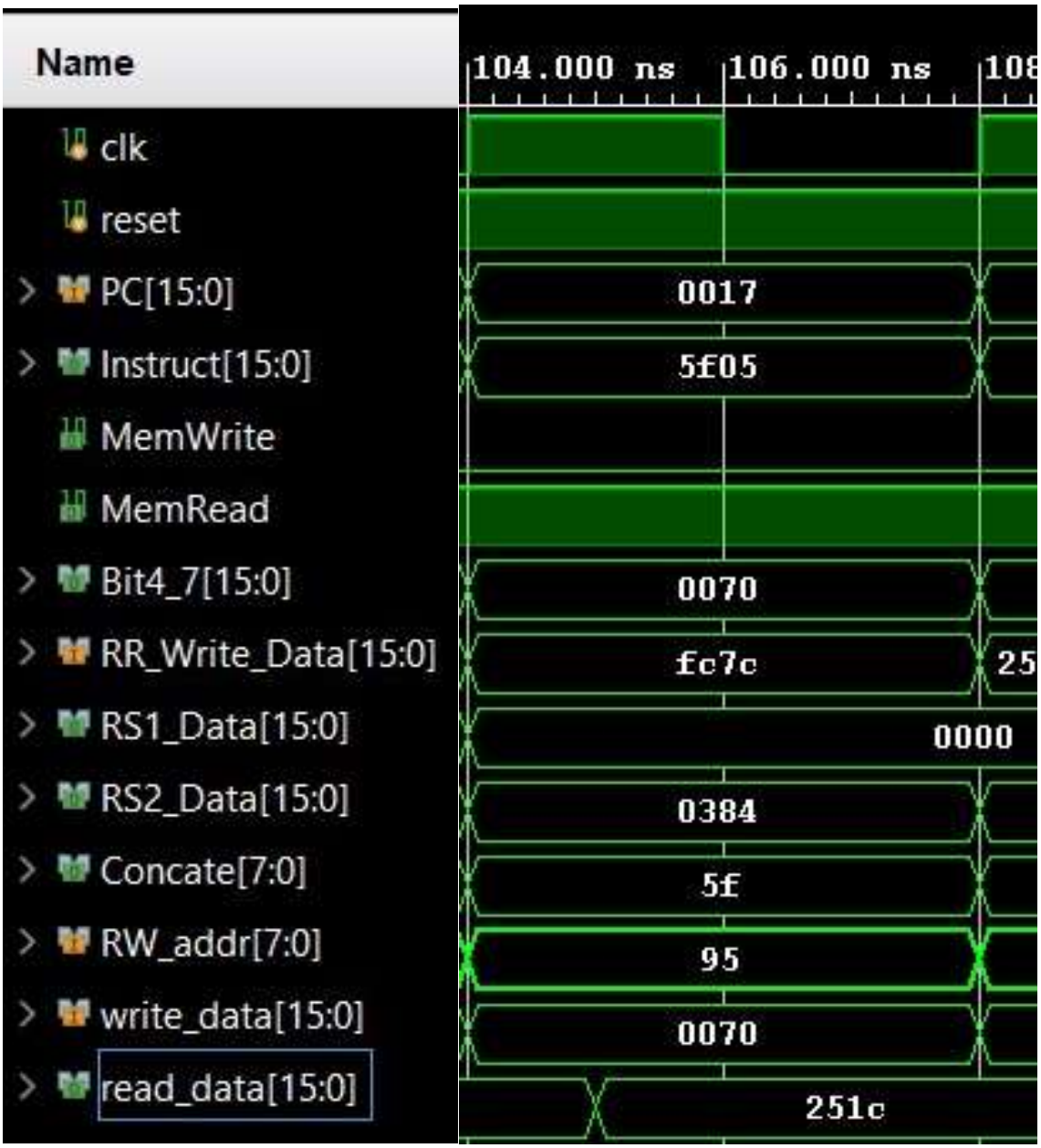
Doing an “NE” comparison with the same registers will of course prove false. This would be an easy way to force the PC to jump to the requested memory address (instruction).

‘GTE’ Instruction: (Branch Instructions)



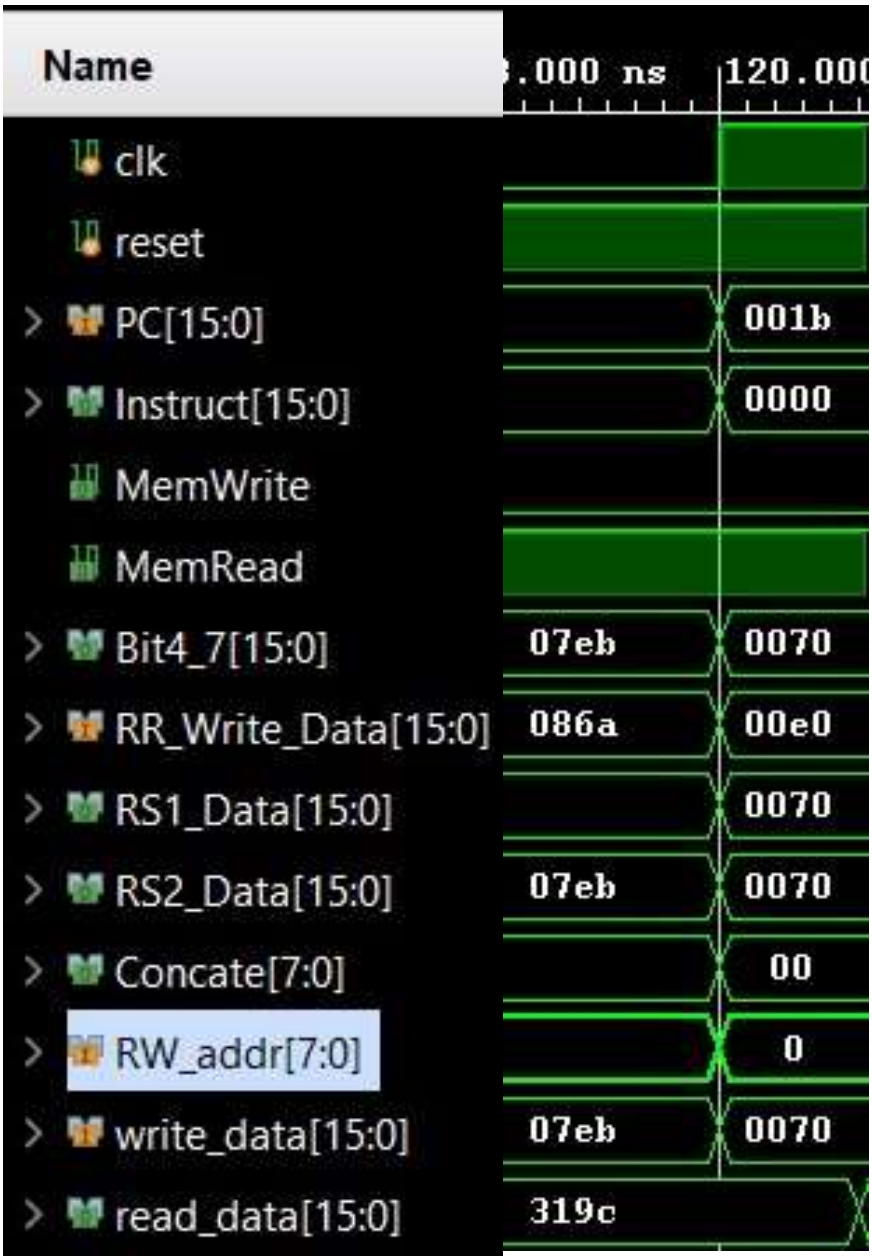
At the 96ns mark, a Greater Than or Equal (GTE) Instruciton is received as denoted by ‘Instruct’ value ‘F90E’ where ‘E’ is the opcode representative of GTE. In this case, the 3rd Byte, Register ‘0’ is used since it holds the memory location we want to jump to should this comparison be false. In this instance, the ‘RS1_Data’ value ‘000F’ is compared against ‘RS2_Data’ value ‘0000’ to see if ‘RS1_Data’ is Greater Than or Equal to ‘RS2_Data’. In this case, the statement proves to be true and the PC continues on to the next instruction.

‘LT’ Instruction: (Branch Instructions)



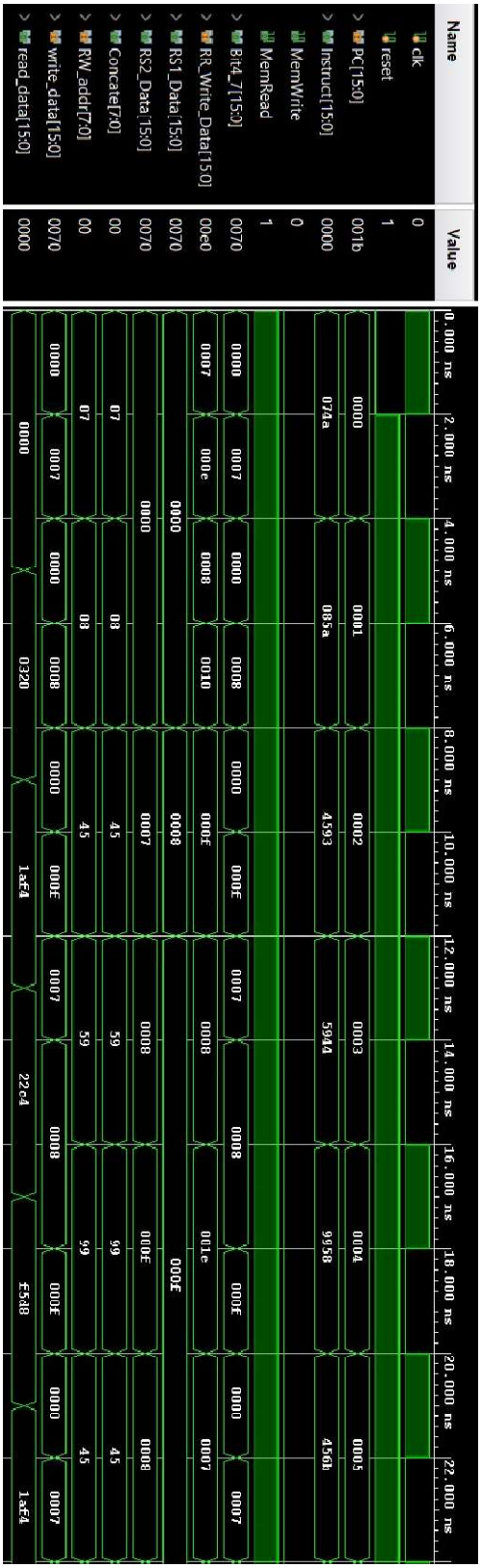
At the 104ns mark, a Less Than (LT) Instruciton is received as denoted by ‘Instruct’ value ‘5F05’ where ‘5’ is the opcode representative of LT. In this case, the 3rd Byte, Register ‘0’ is used since it holds the memory location we want to jump to should this comparison be false. In this instance, the ‘RS1_Data’ value ‘0000’ is compared against ‘RS2_Data’ value ‘0384’ to see if ‘RS1_Data’ is Less Than ‘RS2_Data’. In this case, the statement proves to be true and the PC continues on to the next instruction.

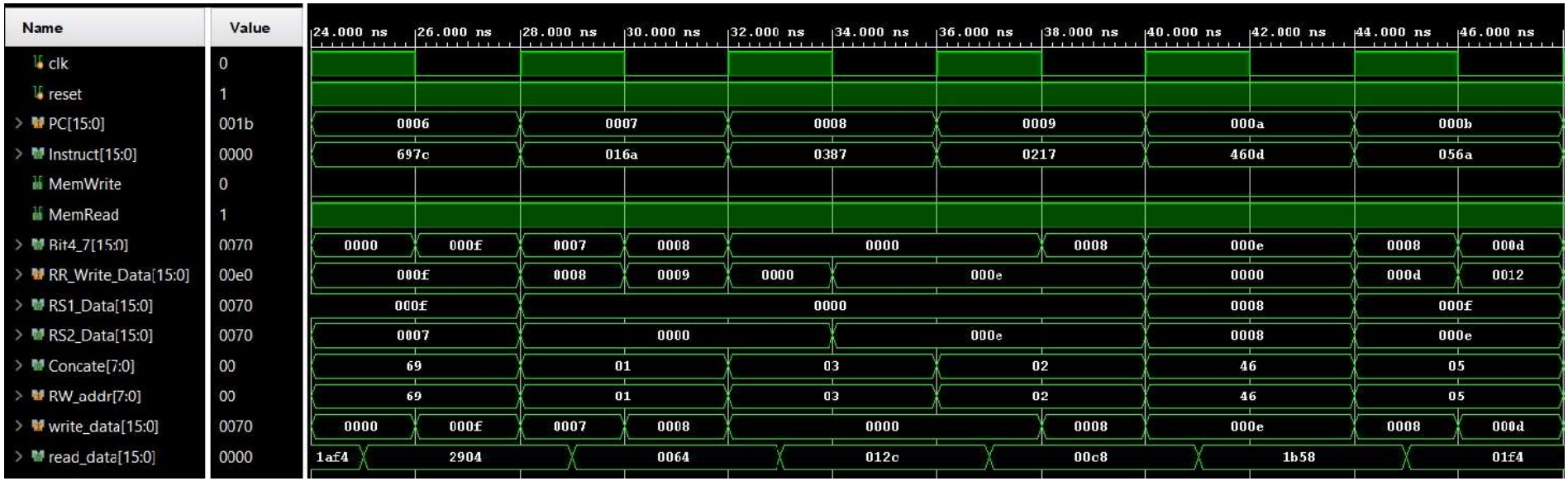
‘HALT’ Instruction:

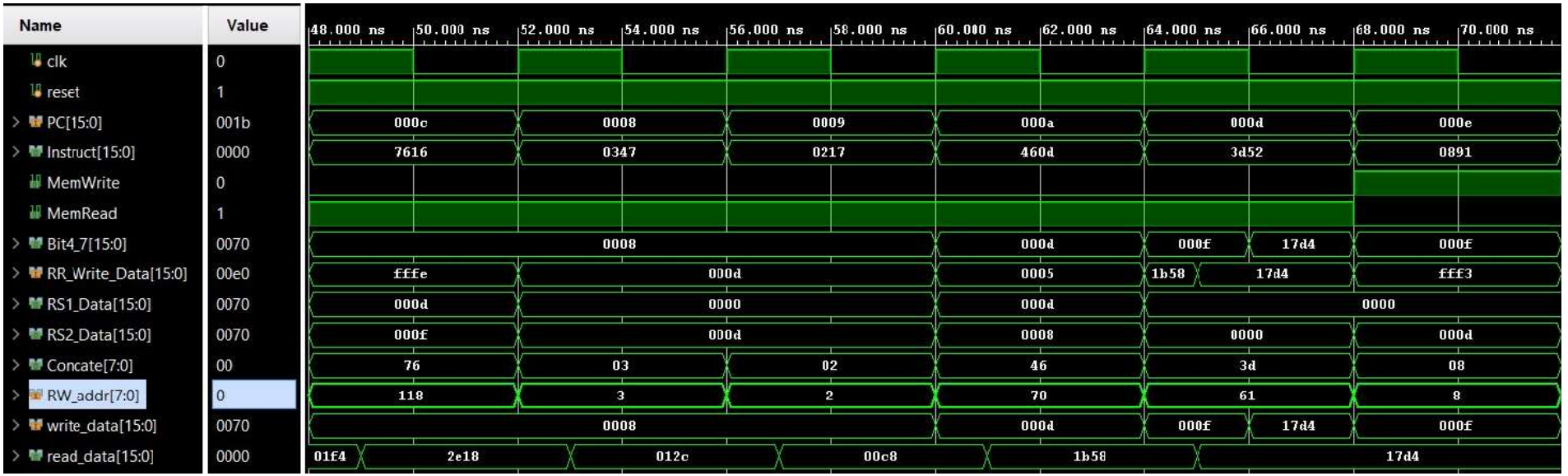


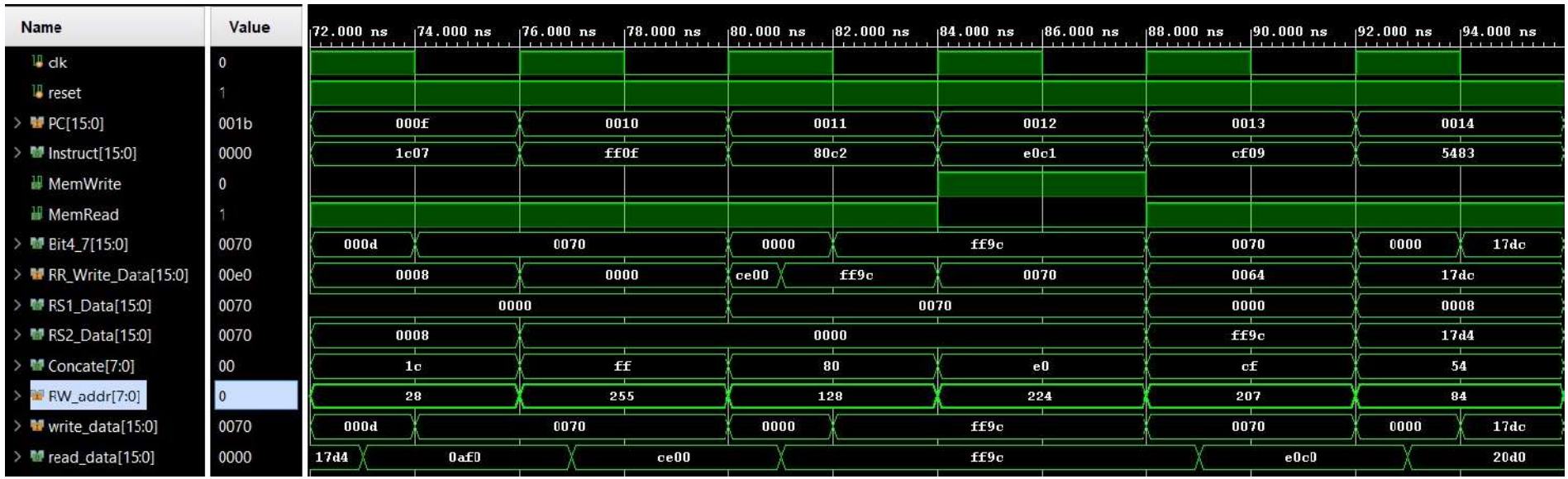
Eventually, at 120ns of the simulation a HALT instruciton is received. This is evident by the ‘Instruct’ value of ‘0000’ where ‘0’ is the opcode representative of a HALT instruciton. When a HALT instruction is received, the HALT flag is triggered which causes the program to stop iterating. When this happens, the program ‘stops’ until the active high ‘reset’ is dropped to 0. Upon the ‘reset’ being received, the program start over again from the beginning. When a HALT instruciton is received, all other bits besides the opcode are disregarded.

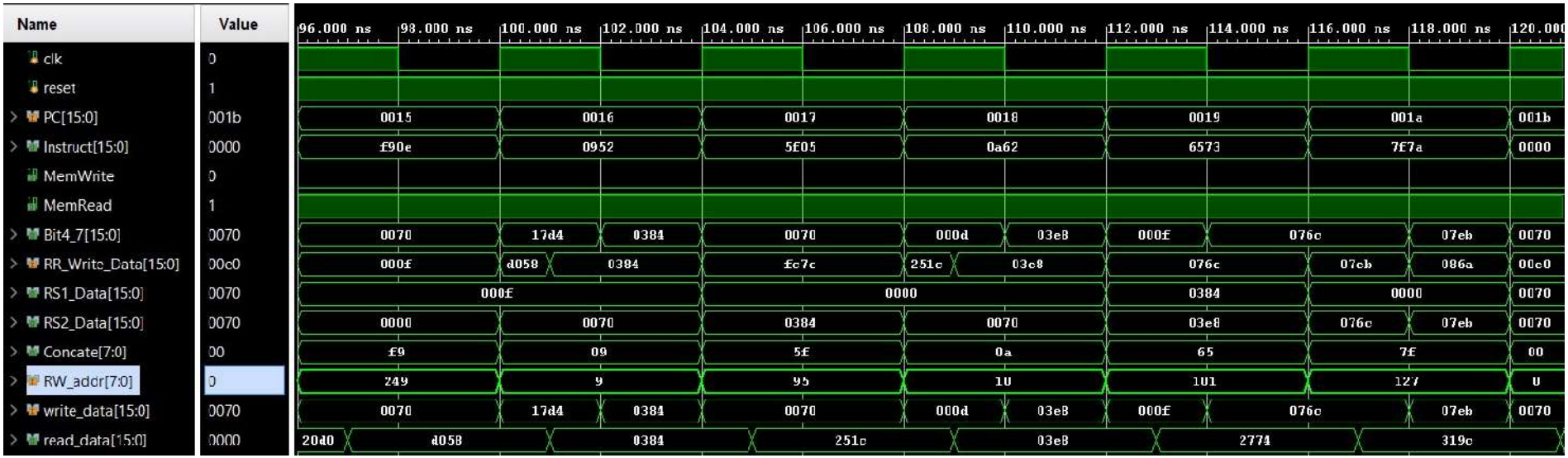
Full Simulation Results:




Name
Value
24.000 ns
26.000 ns
28.000 ns
30.000 ns
32.000 ns
34.000 ns
36.000 ns
38.000 ns
40.000 ns
42.000 ns
44.000 ns
46.000 ns
clk
0
reset
1
PC[15:0]
001b
0006
0007
0008
0009
000a
000b
Instruct[15:0]
0000
697c
016a
0387
0217
460d
056a
MemWrite
0
MemRead
1
Rit4_7[15:0]
0070
0000
000f
0007
0008
0000
0008
000e
0008
000d
RR_Write_Data[15:0]
00e0
000f
0008
0009
0000
000e
0000
000d
0012
RS1_Data[15:0]
0070
000f
0000
0008
000f
RS2_Data[15:0]
0070
0007
0000
000e
0008
000e
Concate[7:0]
00
69
01
03
02
46
05
RW_addr[7:0]
00
69
01
03
02
46
05
write_data[15:0]
0070
0000
000f
0007
0008
0000
0008
000e
0008
000d
read_data[15:0]
0000
1af4
2904
0064
012c
00c8
1b58
01f4


Name
Value
48.000 ns
50.000 ns
52.000 ns
54.000 ns
56.000 ns
58.000 ns
60.000 ns
62.000 ns
64.000 ns
66.000 ns
68.000 ns
70.000 ns
clk
0
reset
1
PC[15:0]
001b
000c
0008
0009
000a
000d
000e
Instruct[15:0]
0000
7616
0347
0217
460d
3d52
0891
MemWrite
0
MemRead
1
Bit4_7[15:0]
0070
0008
000d
000f
17d4
000f
RR_Write_Data[15:0]
00e0
fffe
000d
0005
1b58
17d4
fff3
RS1_Data[15:0]
0070
000d
0000
000d
0000
RS2_Data[15:0]
0070
000f
000d
0008
0000
000d
Concate[7:0]
00
76
03
02
46
3d
08
RW_addr[7:0]
0
118
3
2
70
61
8
write_data[15:0]
0070
0008
000d
000f
17d4
000f
read_data[15:0]
0000
01f4
2e18
012c
00c8
1b58
17d4


Name
Value
72.000 ns
74.000 ns
76.000 ns
78.000 ns
80.000 ns
82.000 ns
84.000 ns
86.000 ns
88.000 ns
90.000 ns
92.000 ns
94.000 ns
clk
0
reset
1
PC[15:0]
001b
000f
0010
0011
0012
0013
0014
Instruct[15:0]
0000
1c07
ff0f
80c2
e0c1
cf09
5483
MemWrite
0
MemRead
1
Bit4_7[15:0]
0070
000d
0070
0000
ff9c
0070
0000
17dc
RR_Write_Data[15:0]
00e0
0008
0000
ce00
ff9c
0070
0064
17dc
RS1_Data[15:0]
0070
0000
0070
0000
0008
RS2_Data[15:0]
0070
0008
0000
ff9c
17d4
Concate[7:0]
00
1c
ff
80
e0
cf
54
RW_addr[7:0]
0
28
255
128
224
207
84
write_data[15:0]
0070
000d
0070
0000
ff9c
0070
0000
17dc
read_data[15:0]
0000
17d4
0af0
ce00
ff9c
e0c0
20d0


Name
Value
clk
reset
PC[15:0]
Instruct[15:0]
MemWrite
MemRead
Bit4_7[15:0]
RR_Write_Data[15:0]
RS1_Data[15:0]
RS2_Data[15:0]
Concate[7:0]
RW_addr[7:0]
write_data[15:0]
read_data[15:0]
0
1
001b
0000
0
1
0070
00c0
0070
0070
00
0
0070
0000
96.000 ns
98.000 ns
100.000 ns
102.000 ns
104.000 ns
106.000 ns
108.000 ns
110.000 ns
112.000 ns
114.000 ns
116.000 ns
118.000 ns
120.000
0015
0016
0017
0018
0019
001a
001b
f90e
0952
5E05
0a62
6573
7E7a
0000
0070
17d4
0384
0070
000d
03e8
000f
076c
07eb
0070
000f
d058
0384
fc7c
251c
03e8
076c
07eb
086a
00c0
000f
0000
0384
0000
0070
0000
0070
0384
0070
03e8
076c
07eb
0070
f9
09
5f
0a
65
7f
00
249
9
95
10
101
127
0
0070
17d4
0384
0070
000d
03e8
000f
076c
07eb
0070
20d0
d058
0384
251c
03e8
2774
319c

I wrote a program which completed 16/16 of the instructions available within my CPU design.

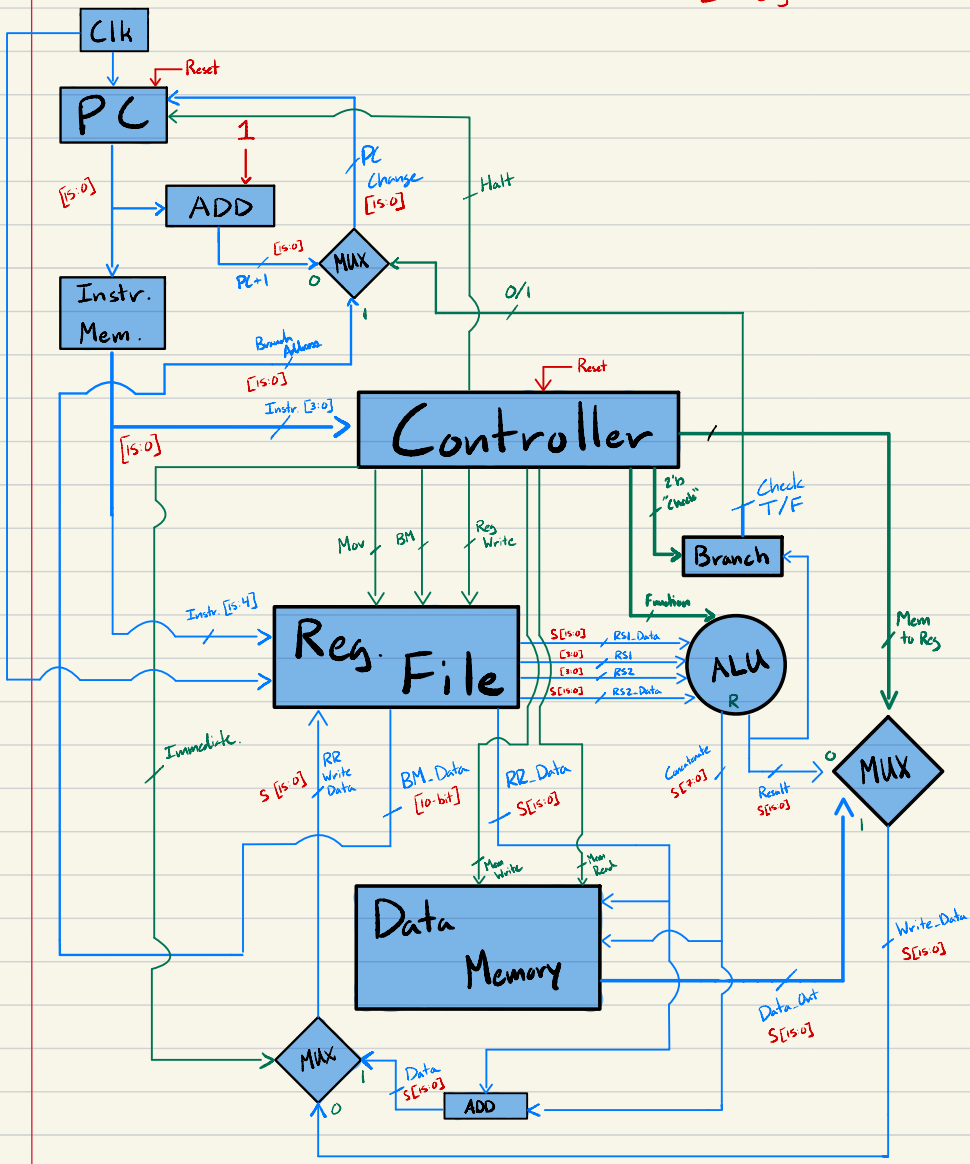
In the pages below, the code used for CPU Part 5 can be found, along with the Testbench code and the Instruction.mem file.

I have also attached the revised Datapath that was recreated during the coding process.

I really enjoyed working on this project and it makes me happy to see the final design being fully functional. I've created instructions unlike those found in RISC V and I'm happy to see them work as intended.

Final Datapath:

S = signed



OPCODE	Instr.	Immediate (1'b)	Mov (1'b)	BM (1'b)	Reg. Write (1'b)	Halt (1'b)	Mem Write (1'b)	Mem Read (1'b)	Branch check (3'b)	ALU Function (3'b)	Mem-Reg. (1'b)
0000	HALT	X	X	X	X	1	X	X	XXX	XXX	X
0001	STOR	0	0	0	0	0	1	0	000	100	0
0010	LOAD	0	0	0	1	0	0	1	000	100	1
0011	ADD	0	0	0	1	0	0	1	000	000	0
0100	AND	0	0	0	1	0	0	1	000	010	0
0101	LT	0	0	0	0	0	0	1	001	001	0
0110	GT	0	0	0	0	0	0	1	100	001	0
0111	BM	0	0	1	0	0	0	1	000	XXX	0
1000	MOV	0	1	0	0	0	0	1	000	XXX	0
1001	NE	0	0	0	0	0	0	1	101	001	0
1010	ADDI	1	0	0	1	0	0	1	000	100	0
1011	SUB	0	0	0	1	0	0	1	000	001	0
1100	OR	0	0	0	1	0	0	1	000	011	0
1101	LTE	0	0	0	0	0	0	1	011	001	0
1110	GTE	0	0	0	0	0	0	1	110	001	0
1111	EE	0	0	0	0	0	0	1	010	001	0

Test Program:

x4 = 7	ADDI x4, 7	00000111	0100 1010
x5 = 8	ADDI x5, 8	00001000	0101 1010
x9 = 15	ADD x9, x5, x4	0100 0101	1001 0011
x4 = 8	AND x4, x9, x5	0101 1001	0100 0100
x5 = 15	MOV x5, x9, x9	1001 1001	0101 1000
x6 = 7	SUB x6, x5, x4	0100 0101	0110 1011
x7 = 15	OR x7, x9, x6	0110 1001	0111 1100
x6 = 8	ADDI x6, 1	0000 0001	0110 1010
x0 = M(13)	BM x0, M(13)	0000001101	00 0111
x1 = M(8)	BM x1, M(8)	0000001000	01 0111
If false, jumps	LTE x0, x6, x4	0100 0110	0000 1101
x6 = 13	ADDI x6, 5	01000000	0111 1010
If false, jumps	GT x1, x6, x7	0111 0110	0001 0110
M(61) → x5	LW x5, M(61)	0011 1101	0101 0010
x9 → M(8)	SW x9, M(8)	0000 1000	1001 0001
x0 → M(28)	BM x0, M(28)	0001 1100	0000 0111
x15 = x15	EE x0, x15, x15	1111 1111	0000 1111
M(128) → x12	LW x12, M(128)	1000 0000	1100 0010
x12 → M(224)	SW x12, M(224)	1110 0000	1100 0001
x15 ≠ x12	NE x0, x15, x12	1100 1111	0000 1001
x3 = 6108	ADD x3, x4, x5	0101 0100	1000 0011
x9 ≥ x15	GTE x0, x9, x15	1111 1001	0000 1110
M(9) → x5	LW x5, M(9)	0000 1001	0101 0010
x15 < x5	LT x0, x15, x5	0101 1111	0000 0101
M(10) → x6	LW x6, M(10)	0000 1010	0110 0010
x7 = 1900	ADD x7, x5, x6	0110 0101	0111 0011
x7 = 2027	ADDI x7, 127	0111 1111	0111 1010
END	HALT	X X	X 0000

0000011101001010
0000100001011010
0100010110010011
0101100101000100
1001100101011000
0100010101101011
0110100101111100
0000000101101010
0000001101000111
0000001000010111
0100011000001101
0000010101101010
0111011000010110
0011110101010010
0000100010010001
0001110000000111
1111111100001111
1000000011000010
1110000011000001
1100111100001001
0101010010000011
1111100100001110
0000100101010010
0101111100000101
0000101001100010
0110010101110011
0111111101111010
0000000000000000

```
`timescale 1ns / 1ps

module Datapath(
    clk,
    reset

//    ,Next_PC,
//    Current_Instr,
//    MemWrite,
//    MemRead,
//    RR_Data,
//    Concatenate
//    ,DataMem
    );

    input clk;
    input reset;

//    input wire signed [15:0] DataMem;
//    output wire [15:0] Next_PC;
//    output wire [15:0] Current_Instr;
//    output wire MemWrite;
//    output wire MemRead;
//    output wire signed [15:0] RR_Data;
//    output wire signed [7:0] Concatenate;

    wire signed [15:0] DataMem;
//    wire [15:0] Next_PC;
//    wire [15:0] Current_Instr;
    wire MemWrite;
    wire MemRead;
    wire signed [15:0] RR_Data;
    wire signed [7:0] Concatenate;

    CPU_Core CPU(.clk(clk), .reset(reset), /*.Next_PC(Next_PC),
.Current_Instr(Current_Instr),*/ .MemWrite(MemWrite),
.MemRead(MemRead), .RR_Data(RR_Data), .Concatenate(Concatenate),
```

```
.DataMem(DataMem));

    Data_Mem Mem(.write_data(RR_Data), .RW_addr(Concatenate),
.memwrite(MemWrite), .memread(MemRead), .reset(reset), .clk(clk),
.read_data(DataMem));

endmodule
```

```
`timescale 1ns / 1ps

module Datapath_TB(

    );
    reg clk;
    reg reset;

//    wire [15:0] Next_PC;
//    wire [15:0] Current_Instr;
//    wire MemWrite;
//    wire MemRead;
//    wire signed [15:0] RR_Data;
//    wire signed [7:0] Concatenate;
//    wire signed [15:0] DataMem;

    Datapath DUT(
    .clk(clk),
    .reset(reset)
    //,.Next_PC(Next_PC), .Current_Instr(Current_Instr)
//Useful for identifying what the next PC address is and current
Instruction
//    , .MemWrite(MemWrite), .MemRead(MemRead)//,
.BranchCheck(BranchCheck), .ALU(ALU), .MemReg(MemReg)
//    , .RR_Data(RR_Data)
//    , .Concatenate(Concatenate)
//    ,.DataMem(DataMem)
    );

    always
        #2 clk = ~clk;

    initial
    begin
    clk = 1;
    reset = 0;
    #2 reset = 1;
```

```
    end
endmodule
```

```
`timescale 1ns / 1ps

module CPU_Core(
    clk,
    reset,
//    Halt,

    //Branch + PC_ADD + Instruct. Input/Output (For Testing)
//    Branch_Result
//    Branch_Address,   <----Only uncomment when testing issues
regarding PC

    //Useful for identifying what the next PC address is and
current Instruction
//    Next_PC,
//    Current_Instr,

    //Controller Input/Output (For Testing)
//    ,RegWrite,
//    Immediate,
//    Mov,
//    BM,
    MemWrite,
    MemRead,
//    BranchCheck,
//    ALU,
//    MemReg,

    //RegFile Input/Output (For Testing)
//    ,RR_Write_Data,
//    RS1_Data,
//    RS2_Data,
//    RS1,
//    RS2,
//    Reg0_3_Data,
    RR_Data,
```

```
    //ALU Input/Output (For Testing)
//    ,ALU_Out,
    Concatenate

    //ALU-Mem Mux Input/Output (For Testing)
    ,DataMem
//    Data

    //Immediate Add Input/Output (For Testing)
//    ,Result

    //Immediate Mux Input/Output (For Testing)
//    ,WriteBack    <--- Only uncomment when testing RR Write
issues or IM Mux
    );
    input clk;
    input reset;
//    input Halt;

    //Branch + PC_ADD + Instruct. Input/Output (For Testing)
//    input wire Branch_Result;
//    input wire [15:0] Branch_Address; <----Only uncomment when
testing issues regarding PC

    //Useful for identifying what the next PC address is and
current Instruction
//    output [15:0] Next_PC;
//    output [15:0] Current_Instr;

    //Controller Input/Output (For Testing)
//    output RegWrite;
//    output Immediate;
//    output Mov;
//    output BM;
    output MemWrite;
    output MemRead;
//    output [2:0] BranchCheck;
```

```
//    output [2:0] ALU;
//    output MemReg;

    //RegFile Input/Output (For Testing)
//    input wire signed [15:0] RR_Write_Data;
//    output signed [15:0] RS1_Data;
//    output signed [15:0] RS2_Data;
//    output [3:0] RS1;
//    output [3:0] RS2;
//    output [15:0] Reg0_3_Data;
    output signed [15:0] RR_Data;

    //ALU Input/Output (For Testing)
//    output signed [15:0] ALU_Out;
    output signed [7:0] Concatenate;

    //ALU-Mem Mux Input/Output (For Testing)
    input signed [15:0] DataMem;
//    output signed [15:0] Data;

    //Immediate Add Input/Output (For Testing)
//    output signed [15:0] Result;

    //Immediate Mux Input/Output (For Testing)
//    output signed [15:0] WriteBack;     <--- Only uncomment when
testing RR Write issues or IM Mux
//    assign RR_Write_Data = WriteBack;   <--- Only uncomment when
testing RR Write issues or IM Mux

    //PC (ADD, Mux) + Instr. Mem Input/Output Wires
    wire [15:0] PC;
    wire [15:0] PC_Out;
    wire [15:0] IM_Out;
    wire [15:0] P_Add_Out;
    wire Halt;
    wire Branch_Result;
```

```
    //Controller Input/Output Wires
    wire RegWrite;
    wire Immediate;
    wire Mov;
    wire BM;
//    wire MemWrite;
//    wire MemRead;
    wire [2:0] BranchCheck;
    wire [2:0] ALU;
    wire MemReg;

    //Reg File Input/Output Wires
    wire signed [15:0] RR_Write_Data;
    wire signed [15:0] RS1_Data;
    wire signed [15:0] RS2_Data;
    wire [3:0] RS1;
    wire [3:0] RS2;
    wire[15:0] Reg0_3_Data;
//    wire signed [15:0] RR_Data;

    //ALU Input/Output Wires
    wire signed [15:0] ALU_Out;
//    wire signed [7:0] Concatenate;

    //ALU-Mem Mux Input/Output Wires
//    wire signed [15:0] DataMem;
    wire signed [15:0] Data;

    //Immediate Add Input/Output Wire
    wire signed [15:0] Result;

    //Useful for identifying what the next PC address is and
current Instruction
//    wire [15:0] Next_PC;
//    wire [15:0] Current_Instr;
//    assign Next_PC = PC;
//    assign Current_Instr = IM_Out;
```

```
    PC Call(.clk(clk), .reset(reset), .PC_In(PC), .Halt(Halt),
.PC_Out(PC_Out));
    Instruction_Mem Inst(.PC(PC_Out), .Instruct(IM_Out));
    PC_Add P_Add(.PC(PC_Out), .Result(P_Add_Out));

    //In order to test, the input 'Reg0_3_Data' must be replaced
with 'Branch_Address'
    PC_Mux P_Mux(.AND_Out(Branch_Result), .ADD(P_Add_Out),
.Branch(Reg0_3_Data), .PC_Change(PC));

    Controller Contr(.ControllerIn(IM_Out[3:0]), .reset(reset),
.RegWrite(RegWrite), .Immediate(Immediate), .Mov(Mov), .BM(BM),
.MemWrite(MemWrite), .MemRead(MemRead), .BranchCheck(BranchCheck),
.ALU(ALU), .MemReg(MemReg), .Halt(Halt));

    RegFile Reg(.clk(clk), .Instruct(IM_Out[15:4]), .mov(Mov),
.RegWrite(RegWrite), .BM(BM), .RR_Write_Data(RR_Write_Data),
.RS1_Data(RS1_Data), .RS2_Data(RS2_Data), .RS1(RS1), .RS2(RS2),
.Reg0_3(Reg0_3_Data), .Bit4_7(RR_Data));

    ALU Load(.RS1(RS1_Data), .RS1_Reg(RS1), .RS2(RS2_Data),
.RS2_Reg(RS2), .Funct(ALU), .Out(ALU_Out), .Concate(Concatenate));

    MemReg_Mux Mem_Mux(.MemReg(MemReg), .ALU(ALU_Out),
.DataMem(DataMem), .Data(Data));

    Immediate_Add IM_Add(.Data(Concatenate), .RegData(RR_Data),
.Result(Result));

    //In order to test, the output 'RR_Write_Data' must be replaced
with 'WriteBack'
    Immediate_Mux IM_Mux(.Immediate(Immediate), .Data(Data),
.ADD(Result), .RegData(RR_Write_Data));

    Branch_Check Check(.ALU_Out(ALU_Out), .Branch(BranchCheck),
.Result(Branch_Result));
```

```
    //Used for Testing DataMem before making Top-Top Level Design
    //Data_Mem Mem(.write_addr(RR_Data), .read_addr(Concatenate),
.RR_write_data(Concatenate), .memwrite_read(MemWrite_MemRead),
.reset(reset), .clk(clk), .read_data(DataMem));

endmodule
```

```
`timescale 1ns / 1ps

module Data_Mem (
    input wire signed [15:0] write_data,
    input wire [7:0] RW_addr,
    input wire memwrite,
    input wire memread,
    input wire reset,
    input wire clk,                    // All synchronous elements,
including memories, should have a clock signal
    output reg signed [15:0] read_data
    );

    reg [15:0] MEMORY[0:255];  // 256 words of 16-bit memory

    integer i;

    initial begin
      read_data <= 0;
      for (i = 0; i < 256; i = i + 1) begin
        if (i <= 127) MEMORY[i] = i*100;
        else MEMORY[i] = (i-127)*(-100);
      end
    end

    // Using @(addr) will lead to unexpected behavior as memories
are synchronous elements like registers
    always @ (posedge clk) begin
      #1
      if (reset == 1'b0) begin
        read_data <= 0; end
      else if (memread == 1'b0) begin
        if (memwrite == 1'b1) MEMORY[RW_addr] <= write_data; end
      else begin read_data <= MEMORY[RW_addr];end
      end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/22/2023 08:53:10 PM
// Design Name:
// Module Name: ALU
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module ALU(
    input [15:0] RS1,
    input signed [3:0] RS1_Reg,
    input [15:0] RS2,
    input signed [3:0] RS2_Reg,
    input [2:0] Funct,
    output reg signed [15:0] Out,
    output reg signed [7:0] Concate
    );

    always @ *
        begin
        Out = (Funct == 3'b000) ? (RS1 + RS2) :
```

```
            (Funct == 3'b001) ? (RS1 - RS2) :
            (Funct == 3'b010) ? (RS1 & RS2) :
            (Funct == 3'b011) ? (RS1 | RS2) :
            (RS1 - RS2);
        Concate = {RS2_Reg , RS1_Reg};
        end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/22/2023 10:43:50 PM
// Design Name:
// Module Name: Branch_Check
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Branch_Check(
    input signed [15:0] ALU_Out,
    input [2:0] Branch,
    output reg Result
    );

    always @ (ALU_Out or Branch)
        begin
            case(Branch)
                3'b000: Result = 0;
                3'b001: begin
                    if (ALU_Out < 0) Result = 0;
                    else Result = 1; end
```

```
                3'b010: begin
                    if (ALU_Out == 0) Result = 0;
                    else Result = 1; end
                3'b011: begin
                    if (ALU_Out <= 0) Result = 0;
                    else Result = 1; end
                3'b100: begin
                    if (ALU_Out > 0) Result = 0;
                    else Result = 1; end
                3'b101: begin
                    if (ALU_Out != 0) Result = 0;
                    else Result = 1; end
                3'b110: begin
                    if (ALU_Out >= 0) Result = 0;
                    else Result = 1; end
                3'b111: Result = 0;
            endcase
        end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/20/2023 09:16:40 AM
// Design Name:
// Module Name: Controller
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Controller(
    input [3:0] ControllerIn,
    input reset,
    output reg RegWrite,
    output reg Immediate,
    output reg Mov,
    output reg BM,
    output reg MemWrite,
    output reg MemRead,
    output reg [2:0] BranchCheck,
    output reg [2:0] ALU,
    output reg MemReg,
    output reg Halt
```

```
);

always @ (negedge reset)
    Halt = 0;

always @ (ControllerIn)
    begin
        case(ControllerIn)
            4'b0000:
                begin
                    Immediate <= 1'b0;
                    Mov <= 1'b0;
                    BM <= 1'b0;
                    RegWrite <= 1'b0;
                    MemWrite <= 1'b0;
                    MemRead <= 1'b1;
                    BranchCheck <= 3'b000;
                    ALU <= 3'b000;
                    MemReg <= 1'b0;
                    Halt <= 1;
                end
            4'b0001:
                begin
                    Immediate <= 1'b0;
                    Mov <= 1'b0;
                    BM <= 1'b0;
                    RegWrite <= 1'b0;
                    MemWrite <= 1'b1;
                    MemRead <= 1'b0;
                    BranchCheck <= 3'b000;
                    ALU <= 3'b100;
                    MemReg <= 1'b0;
                    Halt <= 0;
                end
            4'b0010:
                begin
                    Immediate <= 1'b0;
```

```
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b100;
            MemReg <= 1'b1;
            Halt <= 0;
        end
    4'b0011:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b000;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b0100:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b010;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b0101:
```

```
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b0;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b001;
            ALU <= 3'b001;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b0110:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b0;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b100;
            ALU <= 3'b001;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b0111:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b1;
            RegWrite <= 1'b0;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b000;
            MemReg <= 1'b0;
            Halt <= 0;
```

```
        end
    4'b1000:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b1;
            BM <= 1'b0;
            RegWrite <= 1'b0;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b000;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b1001:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b0;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b101;
            ALU <= 3'b001;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b1010:
        begin
            Immediate <= 1'b1;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b100;
```

```
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b1011:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b001;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b1100:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b1;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
            BranchCheck <= 3'b000;
            ALU <= 3'b011;
            MemReg <= 1'b0;
            Halt <= 0;
        end
    4'b1101:
        begin
            Immediate <= 1'b0;
            Mov <= 1'b0;
            BM <= 1'b0;
            RegWrite <= 1'b0;
            MemRead <= 1'b1;
            MemWrite <= 1'b0;
```

```
                    BranchCheck <= 3'b011;
                    ALU <= 3'b001;
                    MemReg <= 1'b0;
                    Halt <= 0;
                end
            4'b1110:
                begin
                    Immediate <= 1'b0;
                    Mov <= 1'b0;
                    BM <= 1'b0;
                    RegWrite <= 1'b0;
                    MemRead <= 1'b1;
                    MemWrite <= 1'b0;
                    BranchCheck <= 3'b110;
                    ALU <= 3'b001;
                    MemReg <= 1'b0;
                    Halt <= 0;
                end
            4'b1111:
                begin
                    Immediate <= 1'b0;
                    Mov <= 1'b0;
                    BM <= 1'b0;
                    RegWrite <= 1'b0;
                    MemRead <= 1'b1;
                    MemWrite <= 1'b0;
                    BranchCheck <= 3'b010;
                    ALU <= 3'b001;
                    MemReg <= 1'b0;
                    Halt <= 0;
                end
        endcase
    end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/22/2023 11:27:36 PM
// Design Name:
// Module Name: Immediate_Add
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Immediate_Add(
    input signed [7:0] Data,
    input signed [15:0] RegData,
    output reg signed [15:0] Result
    );

    always @ (Data or RegData)
        Result = RegData + Data;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/22/2023 08:41:11 PM
// Design Name:
// Module Name: ImmediateMux
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Immediate_Mux(
    input Immediate,
    input signed [15:0] Data,
    input signed [15:0] ADD,
    output reg signed [15:0] RegData
    );

    //always @ (Immediate or Data or ADD)
    always @ *
        RegData = (Immediate == 1'b0) ? Data : ADD;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/26/2023 01:26:53 PM
// Design Name:
// Module Name: Instruction_Mem
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Instruction_Mem(
    input [15:0] PC,
    output reg [15:0] Instruct
    );

    reg [15:0] ROM [0:1024];

    initial $readmemb ("instr.mem", ROM, 0, 1024);

    always @ (PC)
        Instruct = ROM[{PC}];
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/22/2023 08:34:28 PM
// Design Name:
// Module Name: MemReg_Mux
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module MemReg_Mux(
    input MemReg,
    input signed [15:0] ALU,
    input signed [15:0] DataMem,
    output reg signed [15:0] Data
    );

    always @ (ALU or DataMem or MemReg)
    begin
        Data = (MemReg == 1'b0) ? ALU : DataMem;
    end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/25/2023 11:03:14 AM
// Design Name:
// Module Name: PC
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module PC(
    input wire clk,
    input wire reset,
    input wire [15:0] PC_In,
    input wire Halt,
    output reg [15:0] PC_Out
    );

    always @ (posedge clk or negedge reset) begin
        if (reset == 0) PC_Out = 16'b0000000000000000;
        else if (!Halt) PC_Out = PC_In;
    end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/24/2023 01:26:22 PM
// Design Name:
// Module Name: PC_Add
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module PC_Add(
    input wire [15:0] PC,
    output reg [15:0] Result
    );

    always @ (PC)
        Result = PC + 16'b0000000000000001;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/22/2023 08:18:17 PM
// Design Name:
// Module Name: PC_Mux
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module PC_Mux(
    input wire AND_Out,
    input wire [15:0] ADD,
    input wire [15:0] Branch,
    output reg [15:0] PC_Change
    );

    always @ (AND_Out or ADD or Branch)
        PC_Change = (AND_Out == 1'b0) ? ADD : Branch;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/26/2023 05:13:25 PM
// Design Name:
// Module Name: RegFile
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module RegFile(
    input clk,
    input [11:0] Instruct,
    input mov,
    input RegWrite,
    input BM,
    input signed [15:0] RR_Write_Data,
    output reg signed [15:0] RS1_Data,
    output reg signed [15:0] RS2_Data,
    output reg [3:0] RS1,
    output reg [3:0] RS2,
    output reg [15:0] Reg0_3,
    output reg signed [15:0] Bit4_7
```

```
    );

    reg [15:0] Register[0:15];
    reg [1:0] BMR;
    reg [3:0] RR;

    initial begin
    for (integer i = 0; i < 15; i=i+1) begin
            Register[i] = 0;
            BMR = 0;
            RR = 0;
            Register[15] = 0;
        end
    end

    always @ *
         begin

         BMR = Instruct [1:0];
         RR = Instruct[3:0];
         RS1 = Instruct[7:4];
         RS2 = Instruct[11:8];

         RS1_Data = Register[RS1]; //Data of RS1 Register
         RS2_Data = Register[RS2]; //Data of RS2 Register

         Reg0_3 = Register[BMR]; //Data of 2 LSBs of RR Register
for Branch Address

         Bit4_7 = Register[RR];  //Data of RR register for
Immediate Add

         Register[15] = 0;   //Constantly Assigns Register 15 to 0
so it can't be overwritten on accident
     end

     always @ (negedge clk) begin
```

```
        if (RegWrite == 1)
            Register[RR] = RR_Write_Data; //Data to be written back
into RR address
        if (mov == 1) begin
            Register[RR] = Register[RS1]; //Assigns RS1 Register
Data to Register RR
            Register[RS1] = Register[RS2]; //Assigns RS2 Register
Data to Register RS1
        end
        if (BM == 1) begin
            Register[BMR] = Instruct[11:2]; //Assigns bits 11-2 of
instruction to Register [BMR = RR]
        end
     end
endmodule
```