Berkay İPEK

EE446 -2304814

Laboratory Work 3 - Single Cycle Processor Design

Contents

1.2.1 Datapath Design (40% Credits) 3

1. (30% Credits) For the instructions in the CPU that you are going to design, list all the steps that are needed for the execution of each instruction. While adding each instruction show all of the changes in the connections of the data path. 3

2. (5% Credits) State the control signal inputs of your overall design. Draw a black box diagram of your architecture by indicating the inputs and outputs. 9

3. (5% Credits) Implement your datapath design in Schematic Editor of Quartus. 10

1.2.2 Controller Design (60% Credits) 11

1. (5% Credits) Draw the controller unit as a black box diagram and indicate its inputs and outputs 11

2. (30% Credits) While adding each instruction show all of the changes in the control signals. 12

3. (5% Credits) Give the truth table for the main controller of the single-cycle processor. 13

4. (20% Credits) Implement your controller in Verilog HDL. 14

1.2.3 Usage of Parameters (Bonus Credits) 16

Why is it important to use parameterized design? 16

Is it plausible to use parameters for the data-width in this laboratory? 16

What are the possible complications? 16

Would the design work with an arbitrary data-width? 16

1.2.1 Datapath Design (40% Credits)

1. (30% Credits) For the instructions in the CPU that you are going to design, list all the steps that are needed for the execution of each instruction. While adding each instruction show all of the changes in the connections of the data path.

Addition, Subtraction, AND, OR operations does not need ALUsrcB, MemToReg or RegSrc Multiplexers. Also, A2 connection is not there.

LSL and LSR operation need Extend module. Therefore, I added ALUsrcB MUX. Also, I modified the ALU so that it can logically shift operations.

STORE operation needs RegSrc MUX. On the other hand, LOAD operation needs MemToReg MUX.

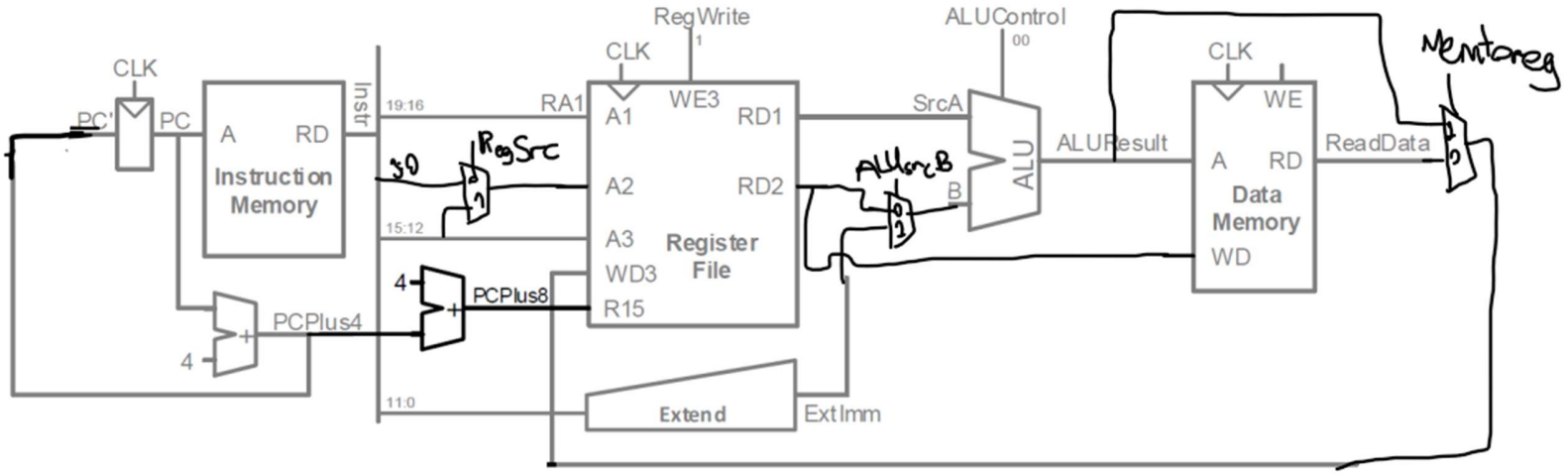
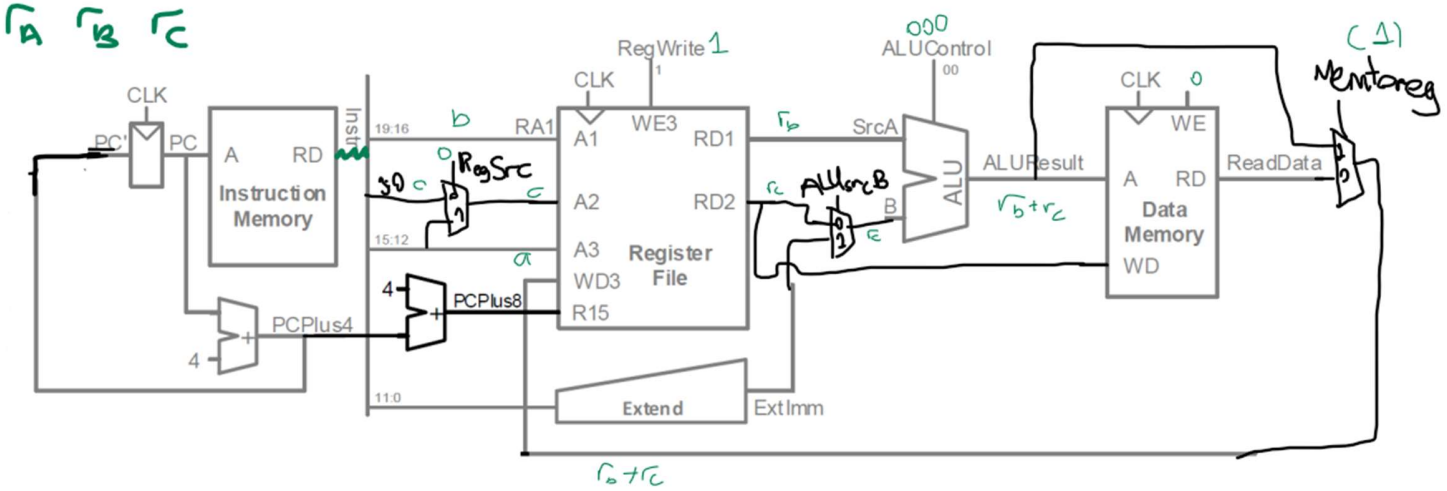


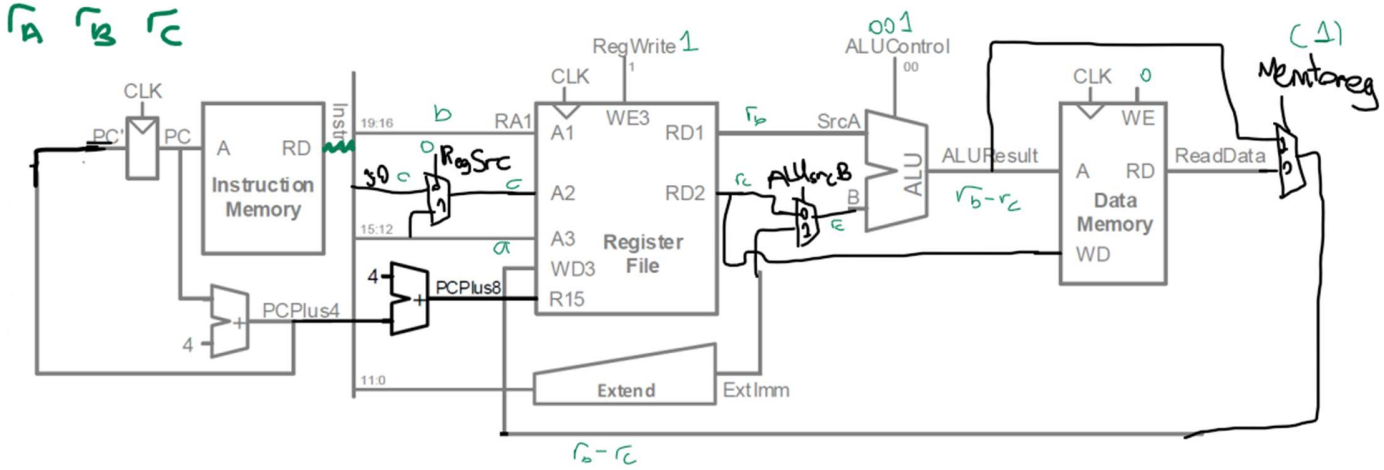
Figure 1: Data path Design

For each operation:

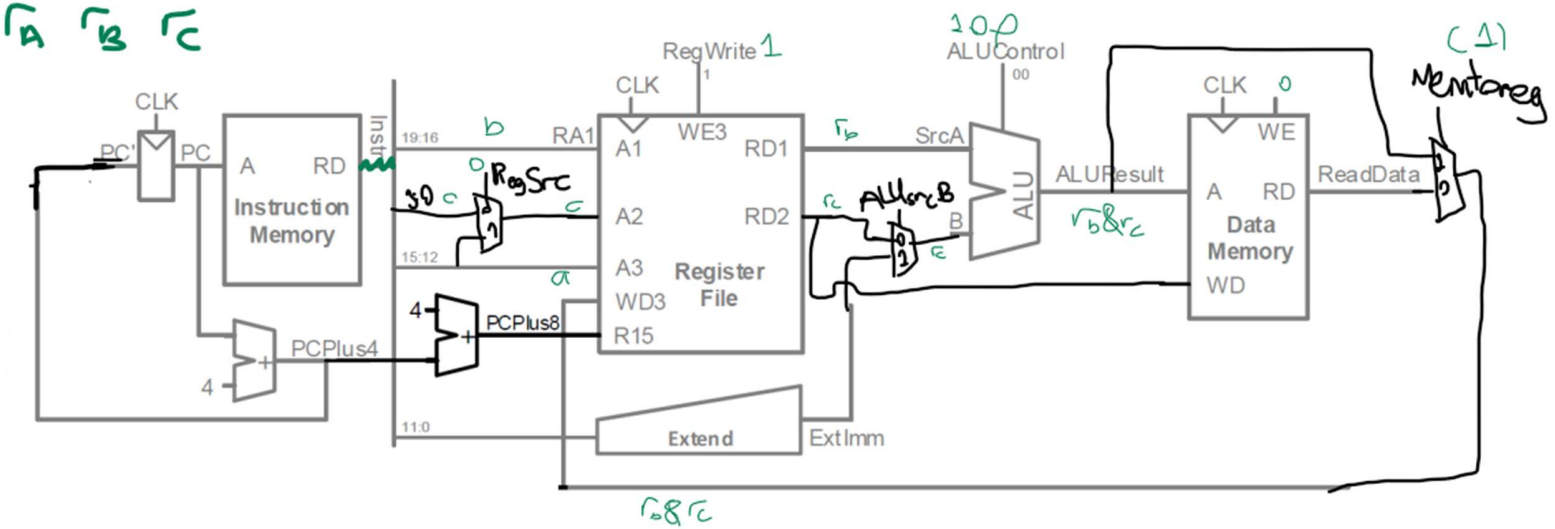
- Addition



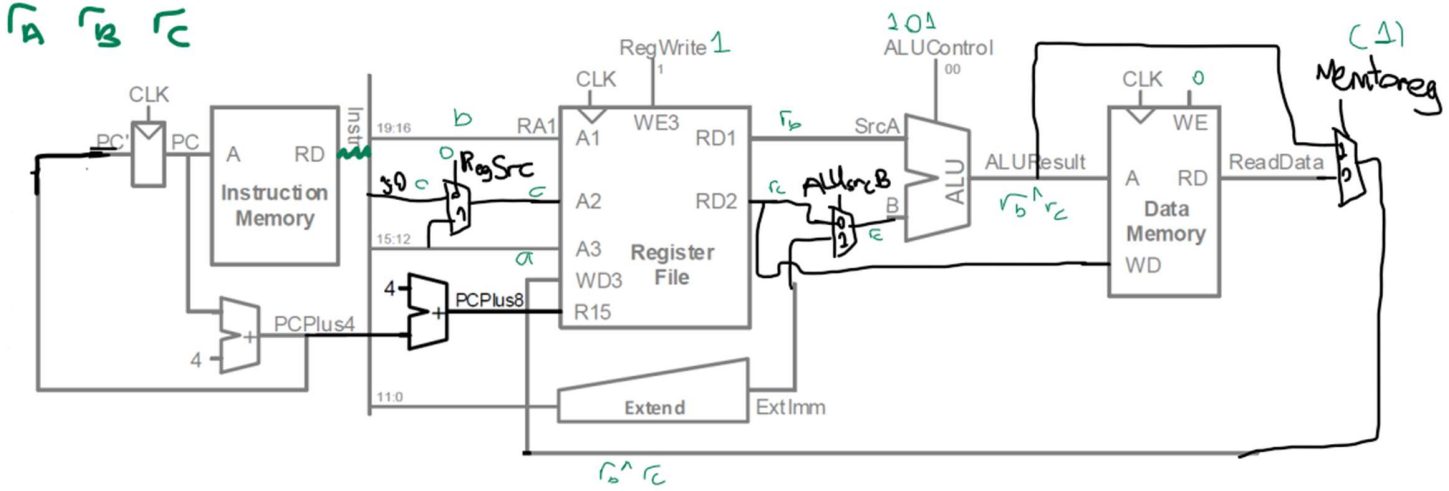
- Subtraction



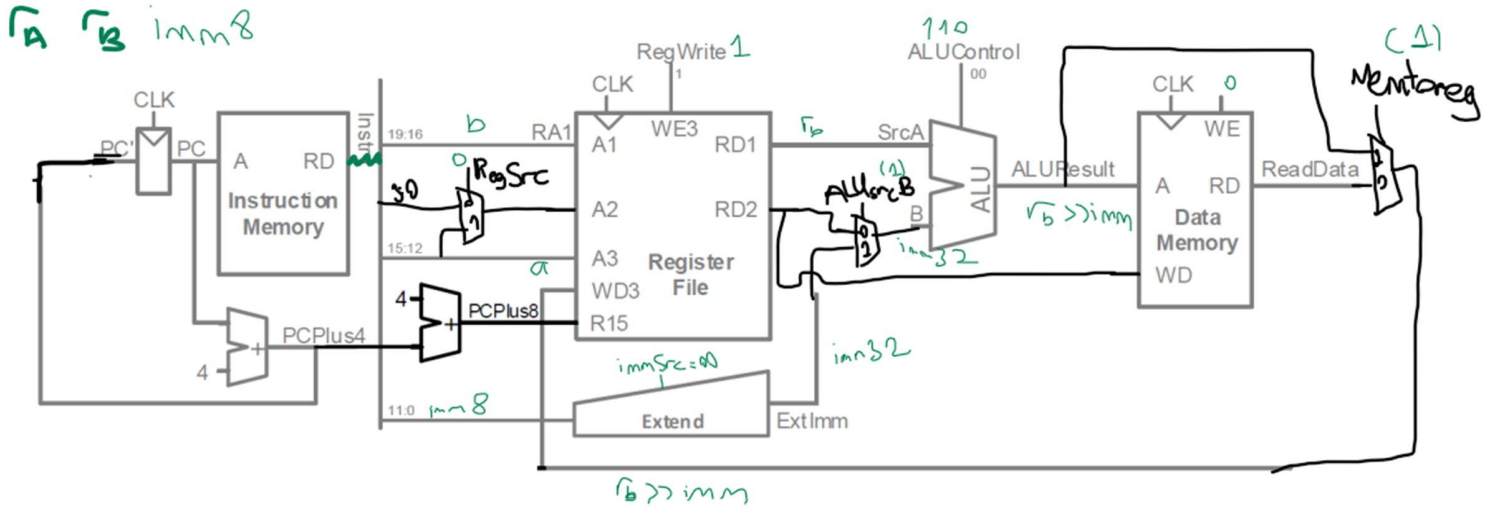
- And


RegWrite
1
ALUControl
00
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
RA1
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
RegSrc
30
15:12
SrcA
ALUSrcB
B
ALU
ALUResult
CLK
WE
A
RD
Data Memory
WD
ReadData
MemtoReg
4
PCPlus4
PCPlus8
11:0
Extend
ExtImm

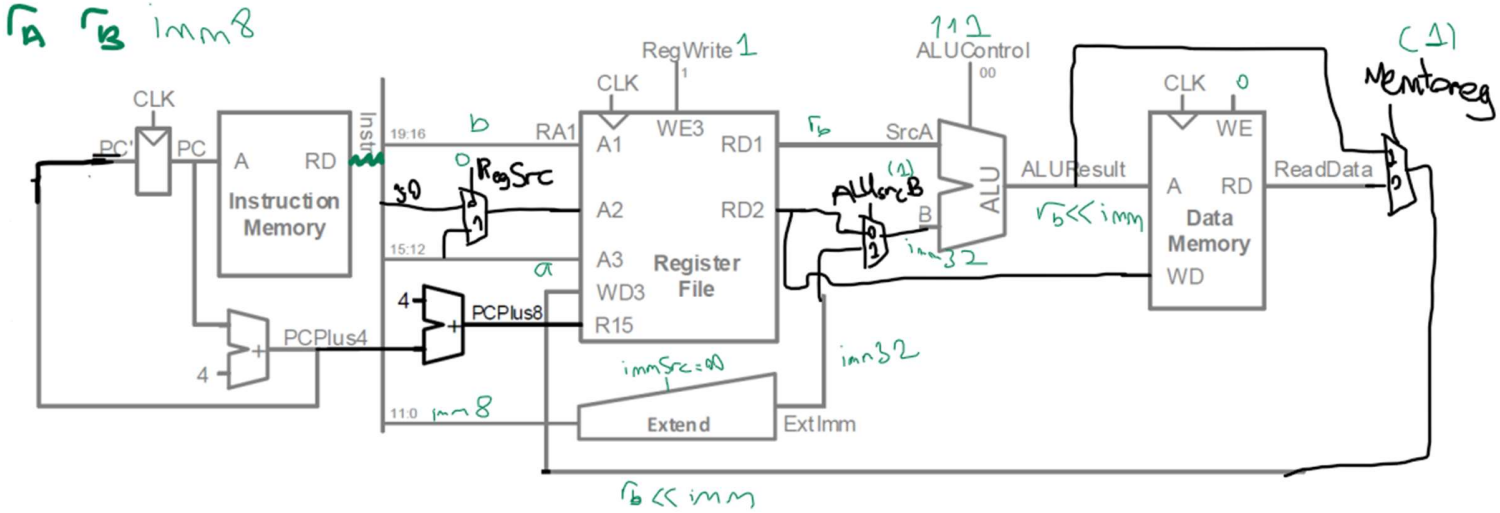
- Orr


RegWrite
1
ALUControl
00
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
RA1
A1
WE3
RD1
A2
RD2
A3
WD3
R15
Register File
RegSrc
30
15:12
SrcA
ALUSrcB
B
ALU
ALUResult
CLK
WE
A
RD
Data Memory
WD
ReadData
MemtoReg
4
PCPlus4
PCPlus8
11:0
Extend
ExtImm

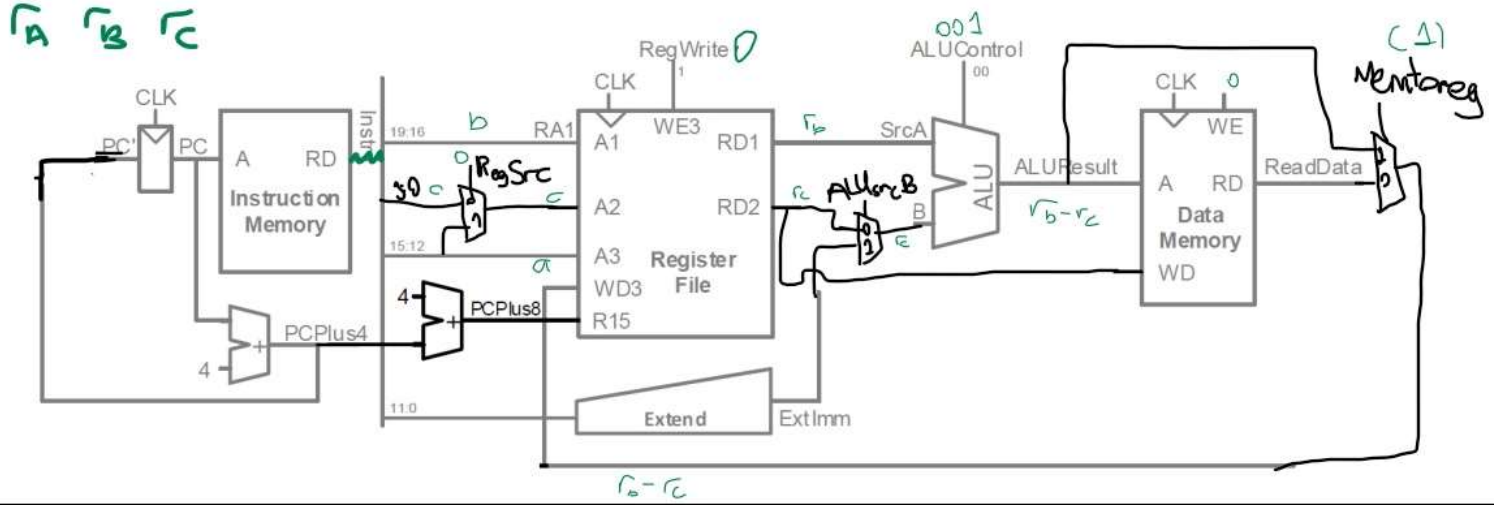
- LSR


CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
imm8
RegSrc
RegWrite
CLK
A1
A2
A3
WD3
R15
RA1
WE3
RD1
RD2
Register File
PCPlus4
PCPlus8
4
Extend
ImmSrc
ExtImm
SrcA
ALUControl
ALUSrcB
ALU
ALUResult
WE
ReadData
Data Memory
WD
MemtoReg
imm32
Rb >> imm

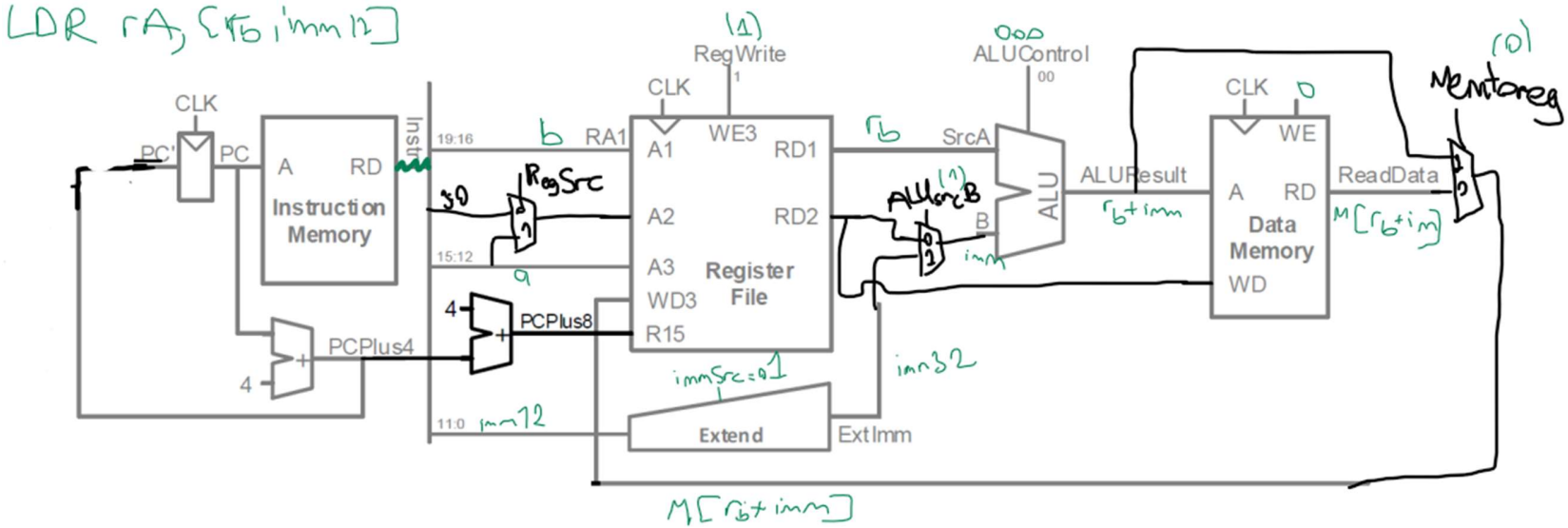
- LSL


CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
imm8
RegSrc
RegWrite
CLK
A1
A2
A3
WD3
R15
RA1
WE3
RD1
RD2
Register File
PCPlus4
PCPlus8
4
Extend
ImmSrc
ExtImm
SrcA
ALUControl
ALUSrcB
ALU
ALUResult
WE
ReadData
Data Memory
WD
MemtoReg
imm32
Rb << imm

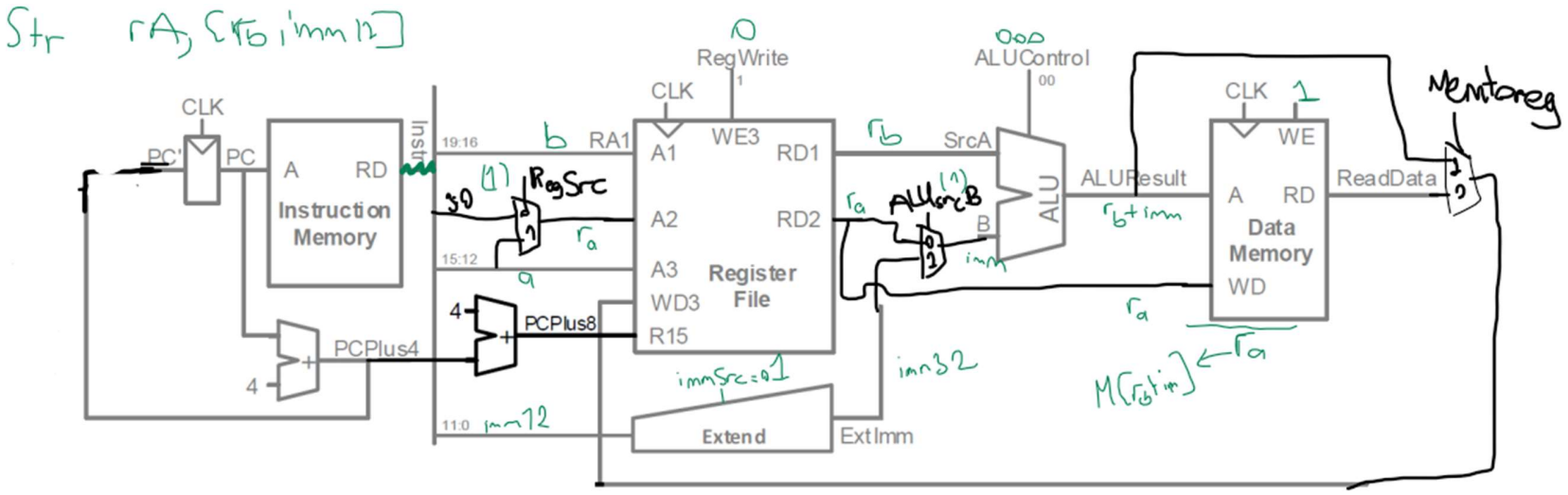
- CMP


CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
RA1
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
RegWrite
RegSrc
4
PCPlus4
PCPlus8
Extend
ExtImm
SrcA
ALUControl
ALU
ALUResult
B
WE
Data Memory
WD
ReadData
Memtoreg

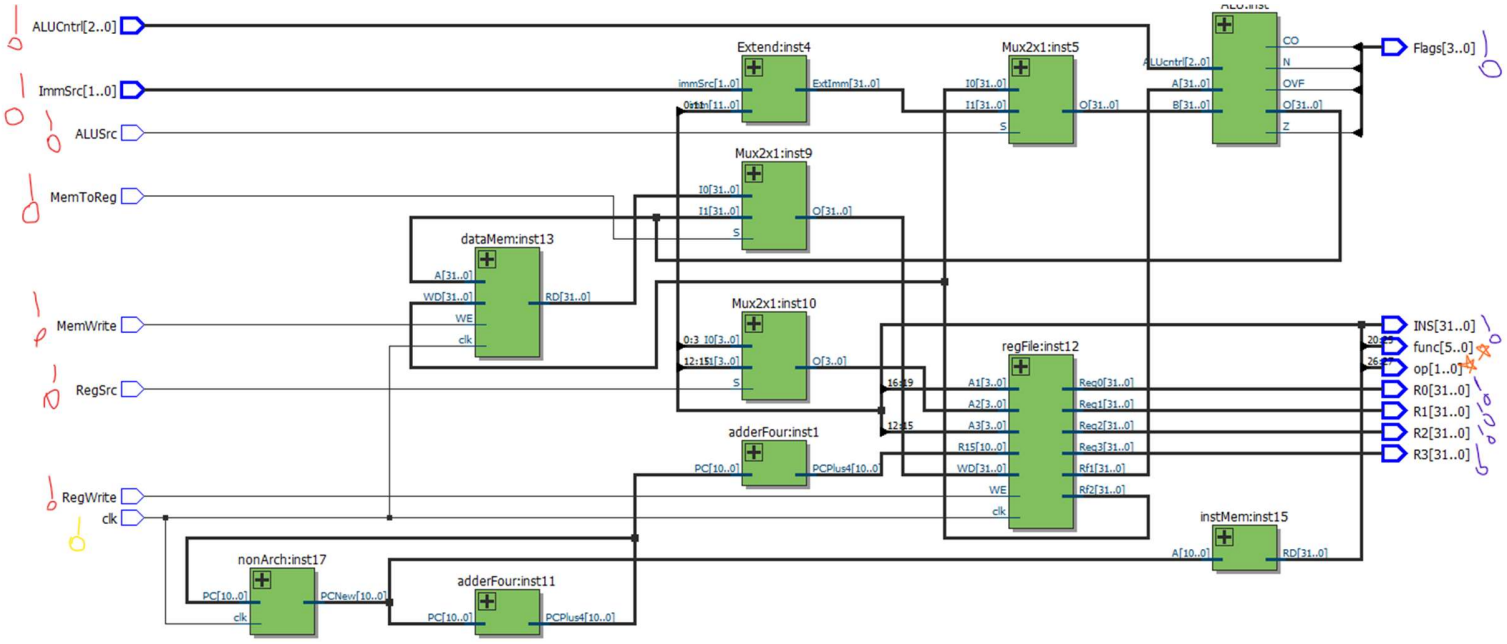
- LDR


LDR rA, [rB, imm12]
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15:12
11:0
RA1
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
RegWrite
RegSrc
4
PCPlus4
PCPlus8
Extend
ExtImm
SrcA
ALUControl
ALU
ALUResult
B
WE
Data Memory
WD
ReadData
Memtoreg
M[rb+imm]

- STR

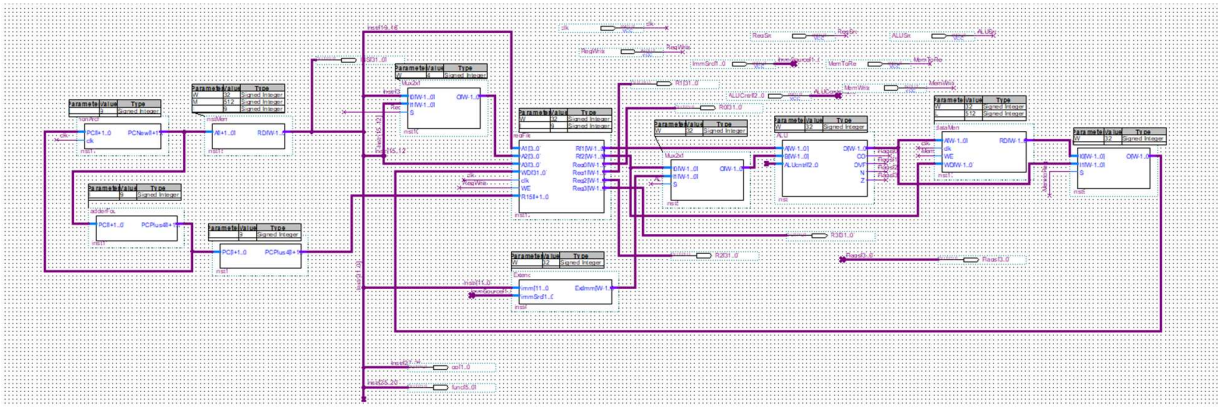

Str rA, [rb, imm12]
RegWrite
1
ALUControl
00
CLK
CLK
CLK
WE
PC'
PC
A
RD
Instruction Memory
Instr
19:16
RA1
A1
WE3
RD1
SrcA
ALUResult
ReadData
RegSrc
30
15:12
A2
RD2
ALUSrcB
B
ALU
A
RD
Data Memory
WD
A3
Register File
WD3
R15
4
PCPlus8
PCPlus4
4
11:0
imm12
immSrc=01
imm32
Extend
ExtImm
MemtoReg
M[rb+imm] ← ra

2. (5% Credits) State the control signal inputs of your overall design. Draw a black box diagram of your architecture by indicating the inputs and outputs.



Sign	Meaning
Red Exclusion Mark	Input Coming from Controller
Yellow Exclusion Mark	Input Coming from Outside
Blue Exclusion Mark	Output for Demonstration
Orange Star	Output for Controller

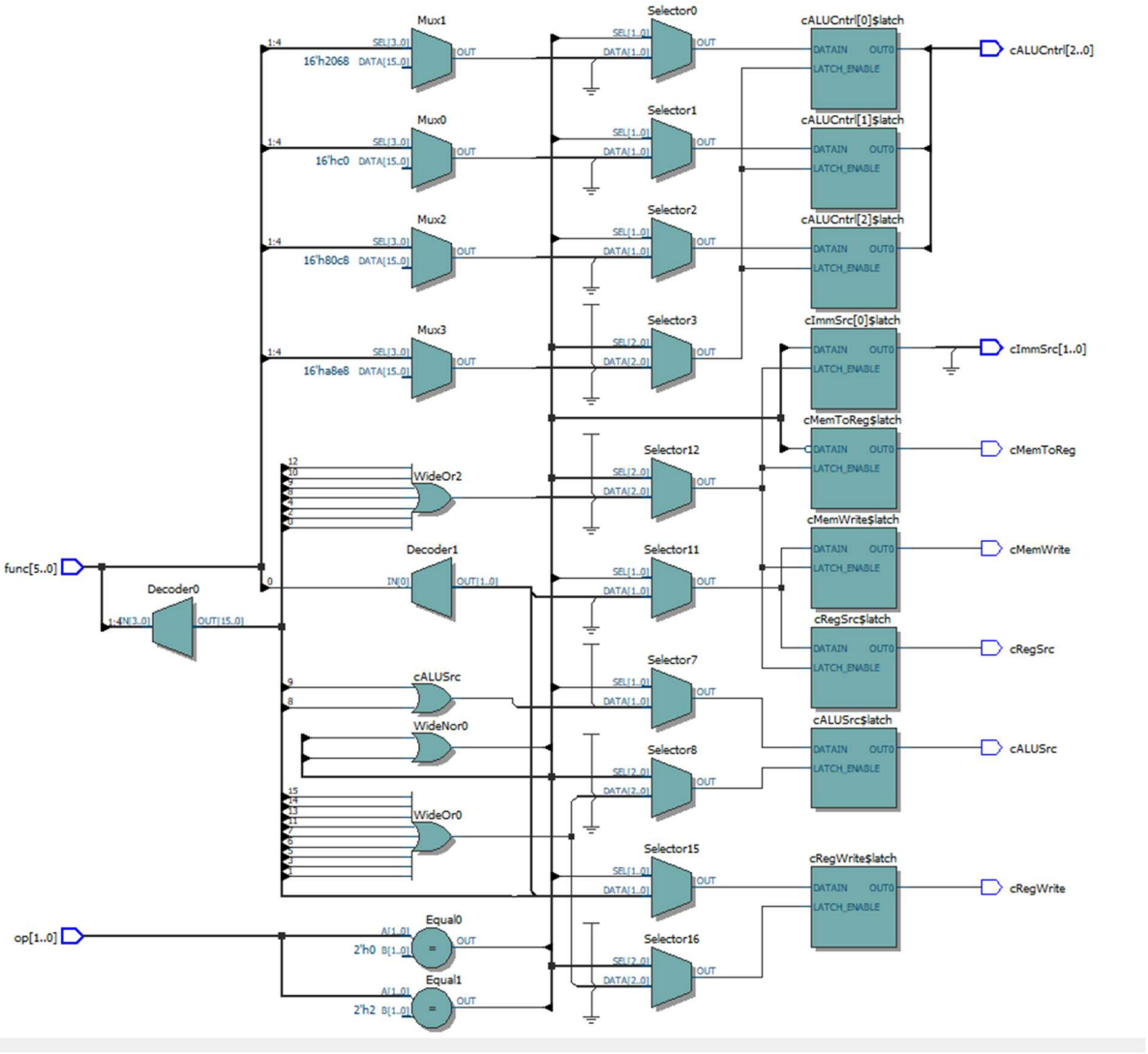
3. (5% Credits) Implement your datapath design in Schematic Editor of Quartus.



See “dp.bdf” file in the code for detailed version.

1.2.2 Controller Design (60% Credits)

1. (5% Credits) Draw the controller unit as a black box diagram and indicate its inputs and outputs



Left ones are coming from Data Path. On the other hand, for the left ones are going for data path.

2. (30% Credits) While adding each instruction show all of the changes in the control signals.

- Addition: RegWrite = 1; MemWrite = 0; ALUCntrl = 000.
- Subtraction: RegWrite = 1; MemWrite = 0; ALUCntrl = 001.
- AND: RegWrite = 1; MemWrite = 0; ALUCntrl = 100.
- ORR: RegWrite = 1; MemWrite = 0; ALUCntrl = 101.
- CMP: RegWrite = 0; MemWrite = 0; ALUCntrl = 001.
- LSL: RegWrite = 1; MemWrite = 0; ALUSrcB=1; immSrc=00; ALUCntrl = 111.
    - ALUSrcB is added. It was all zero for ADD, SUB, AND, ORR, CMP.
    - Extend module is added. immSrc is 'X' (Don't care) for previous commands.
- LSR: RegWrite = 1; MemWrite = 0; ALUSrcB=1; immSrc=00; ALUCntrl = 110.
- LDR: RegWrite = 1; MemWrite = 0; ALUSrcB=1; immSrc=01; MemToReg=0; ALUCntrl = 000.
    - MemToReg is added. MemToReg = 1 for all previous commands expect CMP. For CMP, it is 'X' (Don't Care)
- STR: RegSrc=1; RegWrite = 0; MemWrite = 1; ALUSrcB=1; immSrc=01; MemToReg=X; ALUCntrl = 000.
    - regSrc is added. RegSrc = 0 for ADD, SUB, AND, ORR, CMP. For others, it is 'X' (Don't care)

3. (5% Credits) Give the truth table for the main controller of the single-cycle processor.

	RegSrc	RegWrite	MemWrite	ALUSrcB	immSrc	MemToReg	ALUCntrl
ADD	0	1	0	0	XX	1	000
SUB	0	1	0	0	XX	1	001
AND	0	1	0	0	XX	1	100
ORR	0	1	0	0	XX	1	101
CMP	0	0	0	0	XX	X	001
LSL	X	1	0	1	00	1	111
LSR	X	1	0	1	00	1	110
LDR	X	1	0	1	01	0	000
STR	1	0	1	1	01	X	000

4. (20% Credits) Implement your controller in Verilog HDL.

```
module controller(
   input [1:0] op,
   input [5:0] func,
   output reg cRegWrite,
   output reg cRegSrc,
   output reg [1:0] cImmSrc,
   output reg cALUSrc,
   output reg cMemToReg,
   output reg cMemWrite,
   output reg [2:0] cALUCntrl);

always @(*)
begin
   case(op)
         2'b00://Data Processing ADD SUB AND ORR LSL LSR CMP
            case(func[4:1])//According to cmd, we will determine
                  4'b0100://Add
                     begin
                        cRegWrite=1'b1;
                        cRegSrc=1'b0;
                        cImmSrc=2'b00;//Don't care
                        cALUSrc=1'b0;
                        cMemToReg=1'b1;
                        cMemWrite=1'b0;
                        cALUCntrl=3'b000;
                     end
                  4'b0010://Sub
                     begin
                        cRegWrite=1'b1;
                        cRegSrc=1'b0;
                        cImmSrc=2'b00;//Don't care
                        cALUSrc=1'b0;
                        cMemToReg=1'b1;
                        cMemWrite=1'b0;
                        cALUCntrl=3'b001;
                     end
                  4'b0000://And
                     begin
                        cRegWrite=1'b1;
                        cRegSrc=1'b0;
                        cImmSrc=2'b00;//Don't care
                        cALUSrc=1'b0;
                        cMemToReg=1'b1;
                        cMemWrite=1'b0;
                        cALUCntrl=3'b100;
                     end
                  4'b1100://ORR
                     begin
                        cRegWrite=1'b1;
                        cRegSrc=1'b0;
                        cImmSrc=2'b00;//Don't care
                        cALUSrc=1'b0;
                        cMemToReg=1'b1;
                        cMemWrite=1'b0;
                        cALUCntrl=3'b101;
                     end
                  4'b1010://Cmp
                     begin
                        cRegWrite=1'b0;
                        cRegSrc=1'b0;
                        cImmSrc=2'b00;//Don't care
                        cALUSrc=1'b0;
                        cMemToReg=1'b1;//Don't care
                        cMemWrite=1'b0;
                        cALUCntrl=3'b001;
                     end
```

```
                    4'b1000://Logical Shift Right
                       begin
                          cRegWrite=1'b1;
                          cRegSrc=1'b0;//Don't care
                          cImmSrc=2'b00;
                          cALUSrc=1'b1;
                          cMemToReg=1'b1;
                          cMemWrite=1'b0;
                          cALUCntrl=3'b110;//Modified ALU
                       end
                    4'b1001://Logical Shift Left
                       begin
                          cRegWrite=1'b1;
                          cRegSrc=1'b0;//Don't care
                          cImmSrc=2'b00;
                          cALUSrc=1'b1;
                          cMemToReg=1'b1;
                          cMemWrite=1'b0;
                          cALUCntrl=3'b111;//Modified ALU
                       end
                endcase
            2'b01://Memory Operation STR or LDR
               case(func[0])//According to L bit , we will determine whether it is STR or Loa
                    1'b1://Load
                       begin
                          cRegWrite=1'b1;
                          cRegSrc=1'b0;//Don't Care
                          cImmSrc=2'b01;
                          cALUSrc=1'b1;
                          cMemToReg=1'b0;
                          cMemWrite=1'b0;
                          cALUCntrl=3'b000;//Modified ALU
                       end
                    1'b0://Store
                       begin
                          cRegWrite=1'b0;
                          cRegSrc=1'b1;
                          cImmSrc=2'b01;
                          cALUSrc=1'b1;
                          cMemToReg=1'b0;//Don't care
                          cMemWrite=1'b1;
                          cALUCntrl=3'b000;//Modified ALU
                       end
               endcase
         endcase
   end
   endmodule
```

See “controller.v” in the zip extended file.

1.2.3 Usage of Parameters (Bonus Credits)

Why is it important to use parameterized design?

According to lots of software engineer manifestos (Agile methods), it is noted that when there should be small change in the code, this must be done in a quick way. Therefore, if we think there should be change in a bit-size, it is better to have parameterized design. It helps us ***to change code without putting any efforts*****.**

Is it plausible to use parameters for the data-width in this laboratory?

Actually, at the beginning of the lab, I though the answer was yes. However, as I designed controller part, I realized that there is a signal coming from instruction set to controller, and these signals are determining control signal outputs. While determining it, we are looking at specific bits of instruction set. If we are going to change data width, this specific bit locations will be also changed. Therefore, it does not make sense when we parameterized data-width. We are building the whole controller design (and some part of data path, i.e after the fetching) according to input data-width coming from instruction memory. To sum up, when we want to change data-width of instruction set, we should be able to change lots of things in the design, not only this parameter. Therefore, it is ***not*** plausible to use parameters for the data-width in this laboratory.

What are the possible complications?

As I mentioned in the previous question, design does not only depend on data-width size. Changing data-width means changing instructions in binary form. Therefore, it will cause hazards.

Would the design work with an arbitrary data-width?

If it is changed to higher number of data-width, it can be worked. However, if it is changed to smaller size, it will cause some instruction set problems. For example, normally instruction[27...26] corresponds to OP code. If we change data-width to 25, it won't be working since there is no bit located at 27 or 26.

2 Experimental Work

Instruction memory is filled with as follows. (see InstMem.v)

```
11    initial begin
12       //First 16 memory location is fulled with insturctions
13       // For DP;
14       // cond(4) op(2'b00) I(1) cmd(4) s(1) rn(4) rd(4) smth5 sh 0 Rm(4)
15       Rd[0] <= 32'b0000 01 0 0000 1 0000 0001 000000000000;//LDR R1, [R0,#0]
16       //R1 <= 0
17       Rd[1] <= 32'b0000 01 0 0000 1 0000 0010 000000000100;//LDR R2, [R0,#4]
18       //R2 <= 8
19       Rd[2] <= 32'b0000 01 0 0000 1 0000 0011 000000001000;//LDR R3, [R0,#8]
20       //R3 <= 9
21       Rd[3] <= 32'b0000 01 0 0000 1 0000 0000 000000000000;//LDR R0, [R0,#0]
22       //R0 <= 0
23       Rd[4] <= 32'b0000 00 0 0100 0 0001 0000 000000000010;//ADD R0,R1,R2
24       //R0 <= 8
25       Rd[5] <= 32'b0000 00 0 0010 0 0010 0000 000000000000;//SUB R0, R2, R0
26       //R0 <= 0
27       Rd[6] <= 32'b0000 00 0 0010 0 0000 0000 000000000010;//SUB R0, R0, R2
28       //R0 <= -8
29       Rd[7] <= 32'b0000 00 0 0100 0 0000 0000 000000000011;//ADD R0, R0, R3
30       //R0 <= 1
31       Rd[8] <= 32'b0000 00 0 0000 0 0000 0000 000000000000;//AND R0, R0, R1
32       //R0<= 0
33       Rd[9] <= 32'b0000 00 0 1100 0 0000 0000 000000000011;//ORR R0, R0, R3
34       //R0 <= 9
35       Rd[10] <= 32'b0000 00 0 1000 0 0000 0000 000000000000;//LSR R0,#1
36       //R0 <= 4
37       Rd[11] <= 32'b0000 00 0 1001 0 0000 0000 000000000010;//LSL R0,#2
38       //R0 <= 16
39       Rd[12] <= 32'b0000 01 0 0000 0 0000 0011 000000010000;//STR R3,[R0,#16] (At the
    location 4)
40       //MEM[R0+16] <= 9
41       Rd[13] <= 32'b0000 01 0 0000 1 0000 0000 000000010000;//LDR R0,[R0,#16] (At the
    location 4)
42       //R0 <= 9
43       Rd[14] <= 32'b0000 00 0 1010 0 0010 0000 000000000000;//CMP R0,R2,R0 (set flags R2-R0,
    8-9= -1)
44       Rd[15] <= 32'b0000 00 0 1010 0 0010 0000 000000000000;//CMP R0,R0,R2 (set flags R2-R0,
    8-9= -1)
```

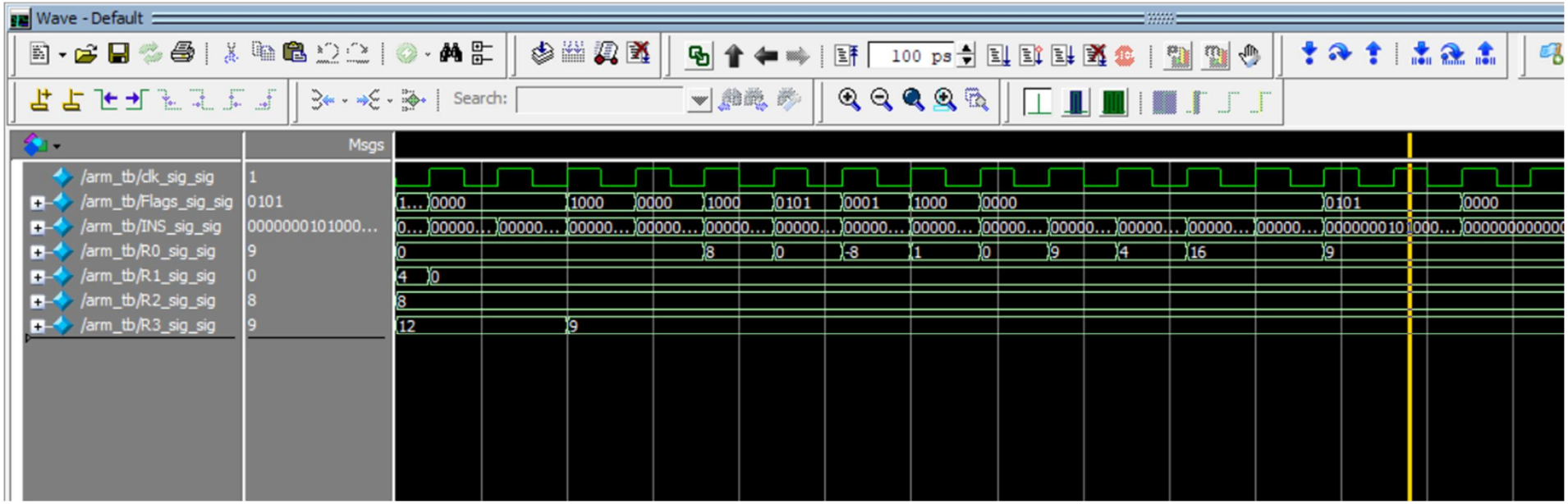
Initially, Rx has a value of (4*x). For example, R0 = 0*4 =0 ; R1 = 1*4=4. (see regFile.v)

On the other hand, for the memory, it is given as follows: (see dataMem.v)

```
initial begin
   mem[0] <= 32'b0000_00_0_0000_0_0000_0000_000000000000;//0 in decimal
   mem[1] <= 32'b0000_00_0_0000_0_0000_0000_000000001000;//8 in decimal
   mem[2] <= 32'b0000_00_0_0000_0_0000_0000_000000001001;//9 in decimal
   mem[3] <= 32'b0000_00_0_0000_0_0000_0000_000000001010;//10 in decimal
   mem[4] <= 32'b0000_00_0_0000_0_0000_0000_000000010100;//20 in decimal
   mem[5] <= 32'b0000_00_0_0000_0_0000_0000_000000000101;// 5 in decimal
   mem[6] <= 32'b0000_00_0_0000_0_0000_0000_000000000110;// 6 in decimal etc.
   mem[7] <= 32'b0000_00_0_0000_0_0000_0000_000000000111;
   mem[8] <= 32'b0000_00_0_0000_0_0000_0000_000000001000;
   mem[9] <= 32'b0000_00_0_0000_0_0000_0000_000000001001;
   mem[10] <= 32'b0000_00_0_0000_0_0000_0000_000000001010;
   mem[11] <= 32'b0000_00_0_0000_0_0000_0000_000000001011;
   mem[12] <= 32'b0000_00_0_0000_0_0000_0000_000000001100;
   mem[13] <= 32'b0000_00_0_0000_0_0000_0000_000000001101;
   mem[14] <= 32'b0000_00_0_0000_0_0000_0000_000000001110;
   mem[15] <= 32'b0000_00_0_0000_0_0000_0000_000000001111;
```

The results are perfect as I expected.

(Flags are the order of ZNOC)



For another example (that I will use in the lab), I used same instructions and different output. For this time I checked R0[3..0] and Flags[3..0].
And results are like this: