Laboratory Work 3 Report

Single Cycle Processor Design

Mustafa BIYIK

2231454

The report consists of three main parts.

The first part is answers of the preliminary work part as hard-copy.

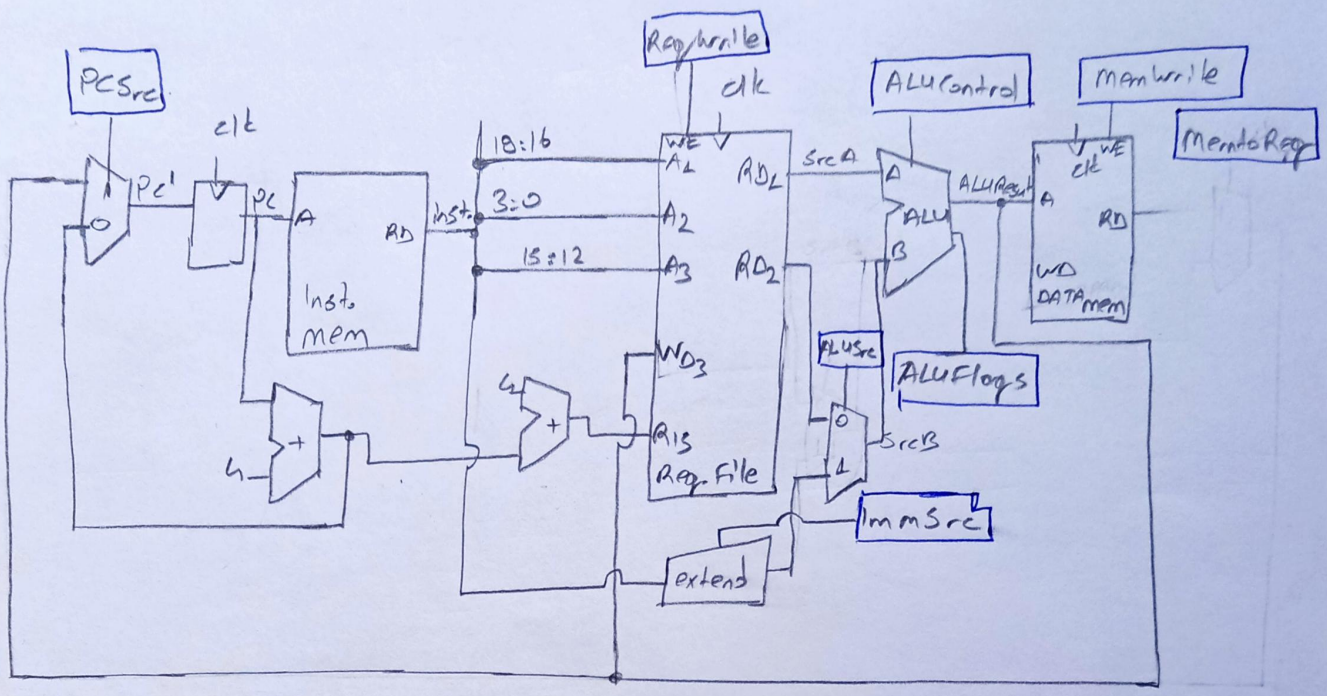
The second part is the simulation result of the experimental part. Most of the simulation results are obtained from the vwf. Also, I have written the testbench, and I have simulated some instruction.

The last part consists of controller_singleCycle, datapath_singleCycle, combination of the controller & the datapath and testbench code for the modelsim.

1. Preliminary Work Part(Hard-copy)

DATA PATH DESIGN

① Datapath for the ADD-SUB-ORR-AND-cmp



- PC is connected to the address input of the instruction memory
- RD is the instruction memory reads out
- Instr(19:16) is 4 bit source register containing the base address. Rn
- The register file reads the 32 bits register value [Rn] onto RD1
- Instr(3:0) is 4 bit source register containing the base address.
- Instr(15:12) is the destination register. Rd

If the RegWrite=1 then the value at WD_3 is written the register which is specified Inst(15:12). at the end of the clock cycle.

- After register file, the values in the registers are processed with the help of the ALU.
- If $Inst_{25}$ = 1 (I bit) then immediate. Therefore, ALUSrc control signal is equal to 1. (imm8) is used as SrcB.

According to the cmd($Instr_{2:0}$) value, (ADD, SUB, ORR, AND, CMP) instructions are specified.

cmd	Inst.	ALUCONTROL	NoWrite	S=1 $Flag W_{1:0}$	S=0 $Flag W_{1:0}$
0100	ADD	000	0	11	00
0010	SUB	001	0	11	00
0000	AND	010	0	10	00
1100	ORR	011	0	10	00
1010	CMP	001	1	11	

If the cmd($Instr_{2:0}$) is equal to 0100, then ADD operation occurs. ALUResult = SrcA + SrcB. MemToReg is selected as 0 then result of ALU directly connected to $WD3$ of register file. At the end of one clock cycle RD is loaded with ALU result. RegWrite = 1 during the operation.
ImmSrc value depends on the Immediate or register In Immediate case, ImmSrc control signal is 00.

If the cmd($Instr_{2:0}$) is equal to 0010 then SUB operation occurs, ALUResult = SrcA - SrcB. ALUResult is stored at the RD at the end of the clock cycle.

If cmd($Instr_{2:0}$) is equal to 0000 then AND operation occurs. ALUResult = SrcA AND SrcB. ALU result is stored at the RD at the end of the clock cycle.

If cmd($Instr_{2:0}$) is equal to 1100 then ORR operation occurs ALUResult = SrcA (ORR) SrcB. ALUResult is stored at the RD at the end of clock cycle.

(A) If cmd (Instr 24:22) is equal to 1010, cmp occurs. Different from the SUB, the SrcA-SrcB value is not written on the Rd register. flags are updated according to the result.

(B) Nowrite signal is equal to 1 during the cmp instruction In ADD, SUB, ORR, AND instruction Nowrite=0. The calculated value is written onto Rd register.

(C) All the 4 flags are updated after ADD, SUB, cmp.

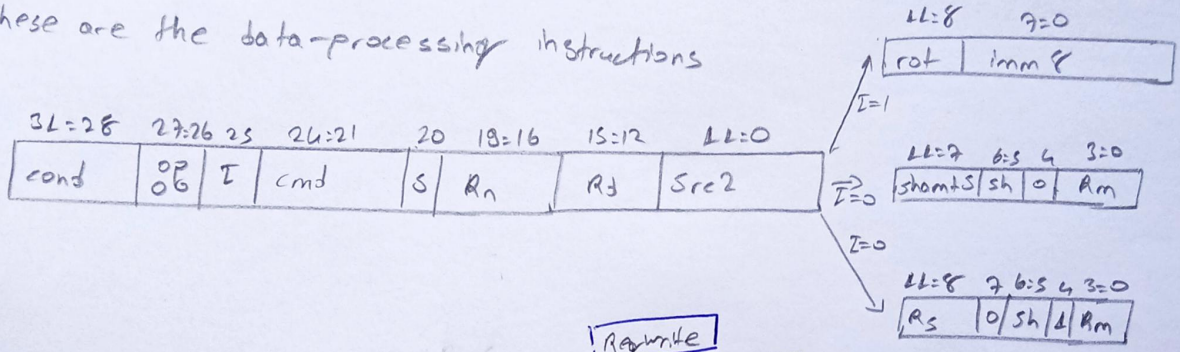
(D) Half of the flags are udated after the ORR, AND.

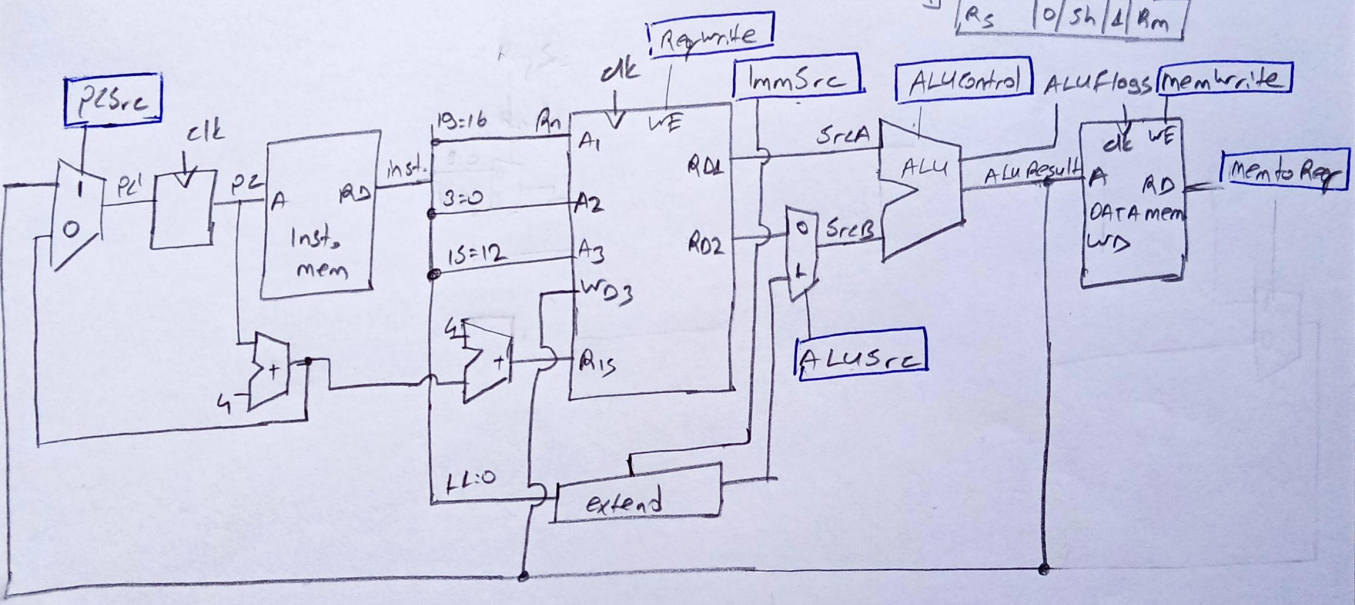
DATAPATH DESIGN

① list all the steps that are needed for the execution of each instruction

⇒ ADD - SUB - AND - ORR - CMP - LSR - LSL

These are the data-processing instructions





The datapath for the ADD - SUB - AND - ORR - CMP - LSR - LSL operations is the same.

By changing control signals, instructions can be changed according

LSR-LSL operations

- PC is connected to address instruction of the memory.
- LSR logical shift right immediate `lsr rA,rB,imm`
- LSL logical shift left immediate `lsl rA,rB,imm`
- $instr_{(19:16)}$ is connected to A_2
- $instr_{(15:12)}$ is connected to A_3
- $instr_{(11:0)}$ is connected to extend if $(instr_4 = 0)$, then our instructions will be valid. Rn value is shifted according to the value specified in (shamt5) part of the instruction. ALUSrc = 1.

ALUControl = 100 for logical shift right

ALUControl = 101 for logical shift left

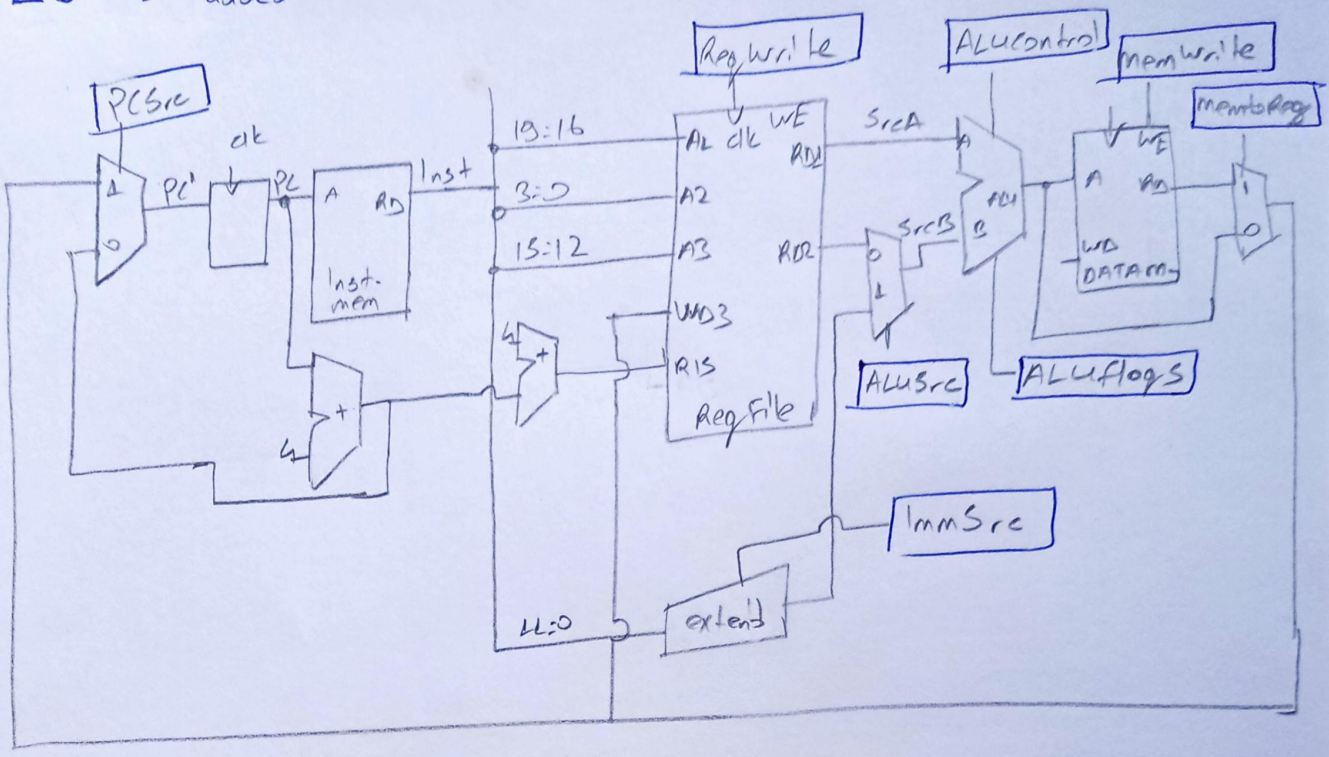
- ALUResult = SrcA << SrcB for lsl & ALUResult = SrcA >> SrcB for lsr
- MemWrite = 0, (no memory write) and the value at the WD_3 is written Rd register. To understand lsl or lsr operations

cmd = 1101 for shift & $instr_{6:5}$ = 00 for shift left & I = 0

$instr_{6:5}$ = 01 for shift right & I = 0

At the end of the cycle `lsl Rd, Rn, shamt5` Rn << shamt5 is written on Rd.

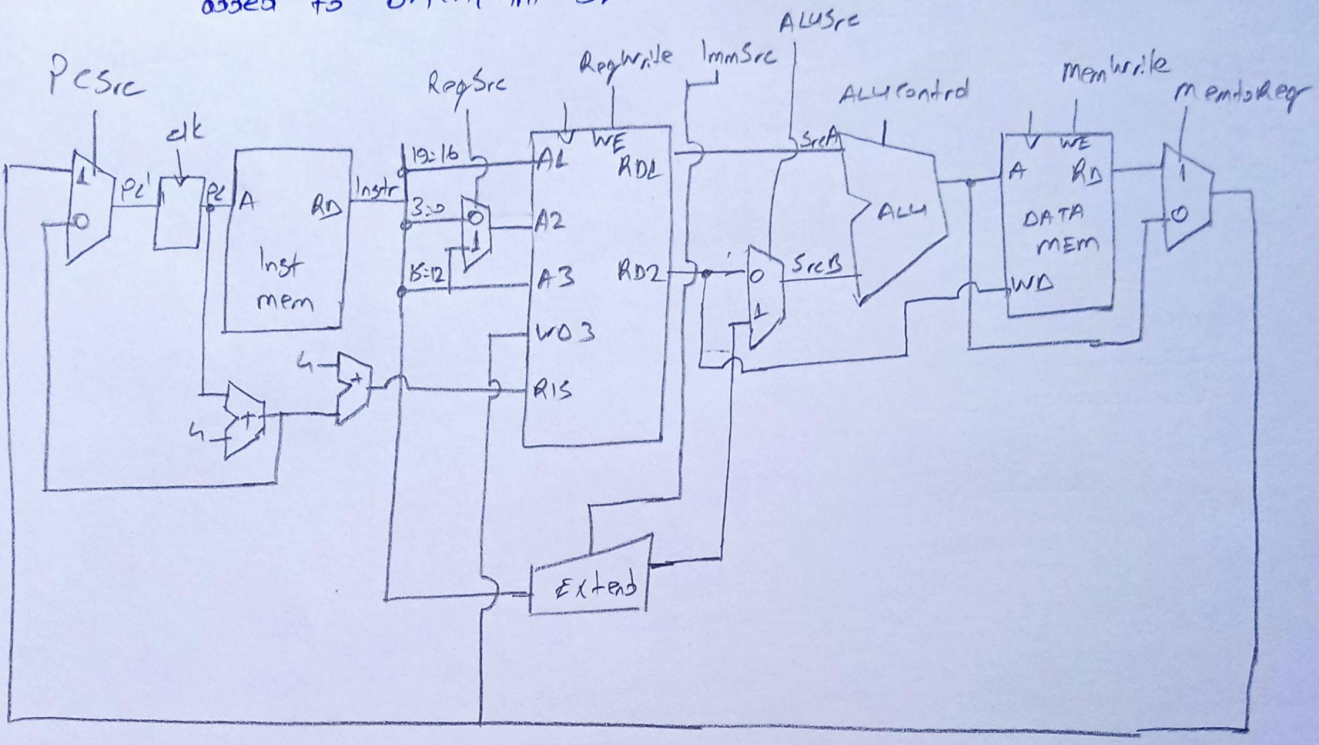
LDR is added to DATAPATH DESIGN



LDR Rd, [Rn, imm12]

- Instruction fetched from the instruction memory
- Source operands are read from the register file
- if the $\overline{I}$ bit of the instruction equal 0, that means immediate Imm12 is extended with zeros to be 32 bit number
- memory adress is computed [Rn] + ExtImm ALUSrc = 1 when immediate ImmSrc = 01, ALUcontrol = 000 (addition)
- data from the memory is read DMI Ext Imm + [Rn]} MemWrite = 0
- Data is written to register file MemtoReg = 1. RegWrite = 1

STR is added to DATAPATH DESIGN



STR Rd, [Rn, imm12] when $\bar{I}=0$

STR Rd, [Rn, Rm] when $\bar{I}=1$

(1) Instruction fetched from the instruction memory

(2) Source operands are read from the register file RegSrc = 1

(3) RD2 keeps the Rd value

(4) Extend the imm12 to calculate address

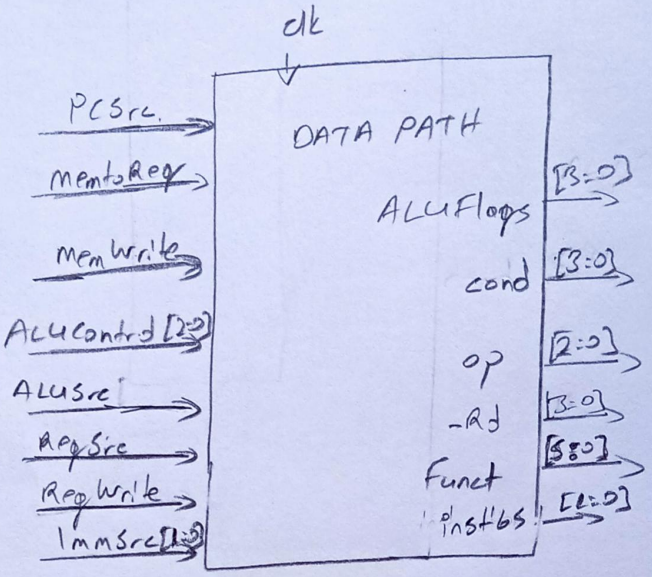
(5) memory address is calculated ImmSrc = 01, SrcB = 1, ALUcontrol = 000

(6) The [Rn] + Ext Imm value is of the DATA mem address pointers

(7) Data coming from RD2 is written to DATA MEMORY

(8) MemWrite = 1, MemToReg = x

(2) ALUFlags, $Instr_{31:28}$, $Instr_{27:26}$(op), $Instr_{25:20}$(funct), $Instr_{15:12}$(Rd) are inputs of the controller unit.

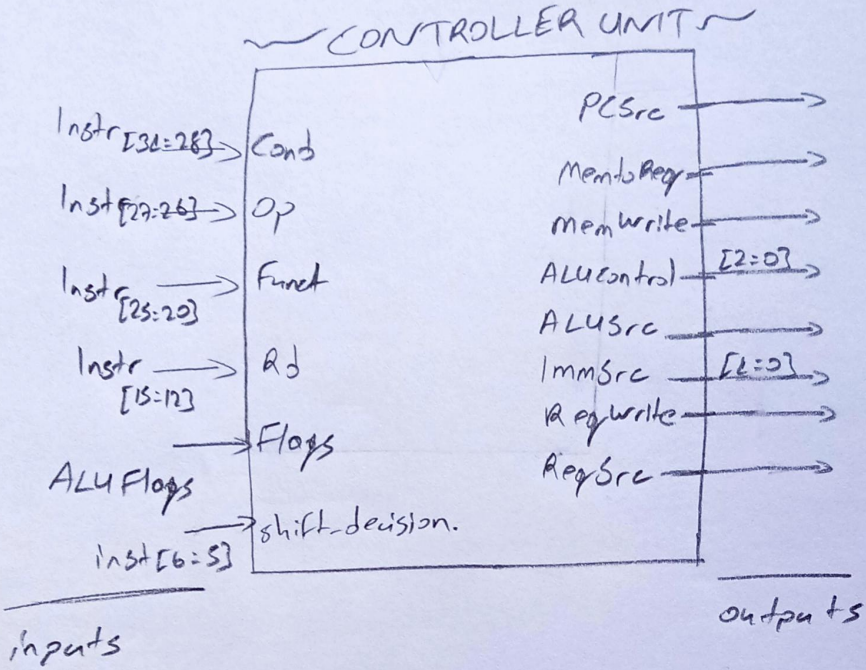


⊗ Instr6:5 is used to detect shift right or shift left operations

shamt5 specifies the shift amount.

2-Controller Design

(1) Controller Unit



(X) outputs of the Controller inputs of the DATAPATH

(X) Inputs of the Controller are the outputs of the DATAPATH.

(X) The controller unit consists of two submodule in the verilog code. One of them is Cond. Logic. Other one is Decoder. module.

(2) While adding each instructions sharing all of the changes in the control signal.

ADD:

- add rA, rB, rC
- PCSrc = 0 // no branch PC increases 4 by 4
- RegWrite = 1 // write occurs at the end of the clock cycle
- ImmSrc = 00 // data processing If I=1 (Instr25 = 1)
- ALUSrc = 1 // if I=1
- RegSrc = 0
- ALUSrc = 0 // if I=0
- MemtoReg = 0 (ALUResult directly goes to register file)
- ALUControl = 000 // addition occurs
- ALUResult is connected to WD3, therefore write operation can be occured
- MemWrite = 0

SUB

- sub rA, rB, rC
- PCSrc = 0 // no branch
- RegWrite = 1 // write to rA occurs at the end of clock cycle
- ImmSrc = 00 // data processing
- ALUSrc = 1 // if I=1 (immediate) immediate value is used while processing
- ALUSrc = 0 // if I=0 (register)
- ALUControl = 001 // subtraction occurs
- RegSrc = 0
- MemWrite = 0
- MemtoReg = 0
- ALUResult connected to WD3 for write operation

AND Logical AND

and RA, RB, RC => rA ← rB & rC

PCSrc = 0

RegWrite = 1

ImmSrc = 00

ALUSrc = 1 // if I = 1

ALUSrc = 0 // if I = 0

ALUcontrol = 010

RegSrc = 0

MemWrite = 0

MemtoReg = 0

ALU is connected to the WD for write operation

ORR Logical ORR

orr RA, RB, RC => RA ← RB | RC

PCSrc = 0

RegWrite = 1

ImmSrc = 00

ALUSrc = 1 // if I = 1

ALUSrc = 0 // if I = 0

ALUcontrol = 011

RegSrc = 0

MemWrite = 0

MemtoReg = 0

ALU is connected to the WD for write operation

LSR logical shift right immediate

lsr rA, rB, imm => rA ← rB >> imm

(1) different from others ImmSrc = 11 added

(2) PCSrc = 0

(3) RegWrite = 1

(4) Imm Src = 11 // for the shamts extend

(5) ALUSrc = 1

(6) ALUcontrol = 100

(7) regSrc = X (not important)

(8) memWrite = 0

(9) memtoReg = 0

(10) ALU is connected to WD of register file for write operation to RA register

LSL logical shift left immediate

lsl rA, rB, imm => rA ← rB << imm

(1) PCSrc = 0

(2) RegWrite = 1

(3) ImmSrc = 11 // for the shamts extend

(4) ALUSrc = 1

(5) ALUcontrol = 101 (shift left)

(6) regSrc = X

(7) memWrite = 0

(8) memtoreg = 0

(9) ALU result is written to RA register at the end of the clock cycle.

Cmp

PCSrc=0

RegWrite=0

ImmSrc=00

ALUSrc=1 // if I=1

ALUSrc=0 // if I=0

ALUcontrol=001 // subtraction

RegSrc=0

MemWrite=0

memtoReg=0

There is no register write

For comp there is internal control signal which is (No-write) when the Cmp is used, No-write make RegWrite=0

STR

- str RA,[RB,imm12]
- str RA,[RB,Rc]
- PCSrc=0, RegWrite=0
- ImmSrc=01 // memory
- ALUSrc=1 // if immediate
- ALUSrc=0 // if register
- ALUcontrol=000 (addition to compute address).
- RegSrc=1 (Rd is written at RA2
- MemWrite=1, memtoReg=X
- address provided to data memory then
- RA is written to the specified address
- MemWrite activated first time
- Also, RegSrc is added to the controller after STR instruction added

LDR

- ldr rA,[rB,imm12]
- ldr rA,[rB,rC]
- PCSrc=0, RegWrite=1
- RegSrc=0
- ImmSrc=01 // memory
- ALUSrc=1 // if immediate
- ALUSrc=0 // if register
- ALUcontrol=000 (addition to compute memory address
- MemWrite=0, memtoReg=1
- Readed value from the memory provided to WD of register file to write operations.

(3) There are 8 control signals & 9 instructions

Control signals / Instructions	RSrc	MemtoReg	MemWrite	ALUcontrol	ALUSrc		ImmSrc	RegWrite	RegSrc
					I=1	I=0			
ADD	0	0	0	000	1	0	00	1	0
SUB	0	0	0	001	1	0	00	1	0
AND	0	0	0	010	1	0	00	1	0
ORR	0	0	0	011	1	0	00	1	0
LSR	0	0	0	100	1		11	1	0
LSL	0	0	0	101	1		11	1	0
CMP	0	0	0	001	1	0	00	0	0
STR	0	X	1	000	1		01	0	1
LDR	0	1	0	000	1		01	1	0

(*) according to I(immediate specifier), we choose ExtImm or register value.

2. Simulation Results of the Experimental Work Part

In register file the registers are specified as output in the datapath to understand that the instructions are working or not.

```
    //output ports
    output          [31:0]    RD1,
    output          [31:0]    RD2,
    input           [31:0]    R15,
    //for the demostration purpose
    output          [31:0]    R1,
    output          [31:0]    R2
);
```

For the demostration purpose R1 and R2 is specified as output in the datapath.

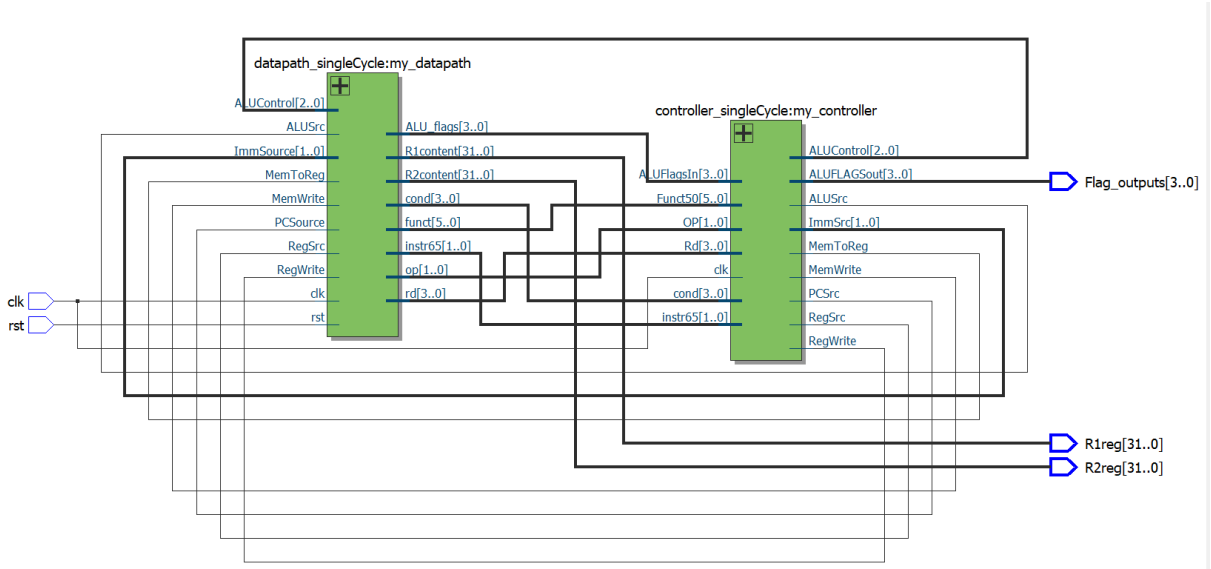


Figure 1:RTL Netlist view pf the datapath and controller design

Memory content in the text file

1st location 1

2nd location 2

3rd location 3

62nd location 62

Until 63rd location

64th location 1

65th location 2

66th location 3

67th location 4

Registers are all 0 initially

Memory content from 1 to 32

```
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000001000
00000000000000000000000000001001
00000000000000000000000000001010
00000000000000000000000000001011
00000000000000000000000000001100
00000000000000000000000000001101
00000000000000000000000000001110
00000000000000000000000000001111
00000000000000000000000000010000
00000000000000000000000000010001
00000000000000000000000000010010
00000000000000000000000000010011
00000000000000000000000000010100
00000000000000000000000000010101
00000000000000000000000000010110
00000000000000000000000000010111
00000000000000000000000000011000
00000000000000000000000000011001
00000000000000000000000000011010
00000000000000000000000000011011
00000000000000000000000000011100
00000000000000000000000000011101
00000000000000000000000000011110
00000000000000000000000000011111
00000000000000000000000000100000
```

```
          //always/op/I/cmd/S/rn/rd/src2
instr[1] =  32'b1110_0100_0001_0100_0001_0000_0010_0001;//ldr load r1
instr[2] =  32'b1110_0100_0001_0001_0010_0000_0000_0011;//ldr load r2
instr[3] =  32'b1110_0000_1000_0010_0001_0000_0000_1001; //add
instr[4] =  32'b1110_0000_0100_0001_0001_0000_0000_0010; //subtract
instr[5] =  32'b1110_0000_0000_0110_0010_0000_0000_0001; //and
instr[6] =  32'b1110_0001_1000_1000_0010_0000_0000_0001; //orr
instr[7] =  32'b1110_0100_0001_1001_0001_0000_0010_1111;//ldr load
instr[8] =  32'b1110_0001_1010_0001_0010_0001_1000_0000; //lsl
instr[9] =  32'b1110_0001_1010_0001_0010_0001_1010_0000; //lsr
instr[10] = 32'b1110_0001_0100_0000_0011_0000_0000_1010; //cmp
instr[11] = 32'b1110_0100_0000_1011_0001_0000_0000_0001; //str
instr[12] = 32'b1110_0100_0001_1010_0010_0000_0000_0001; //ldr
instr[13] = 32'b1110_0100_0000_1011_0001_0000_0000_1010; //str
instr[14] = 32'b1110_0100_0001_1010_0010_0000_0000_1010; //ldr
instr[15] = 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

1st inst is load to ldr r1,[r4,#33] //load r1 with [r4]+33 location content which means r1 loaded with 33

2nd inst is load to ldr r2, [r1, #3] //load r1 with [r1]+3 location content which means r2 loaded with 36

3rd add r1, r2, r7 ; r1=r2+r7

4th sub r1, r1, r2 ; r1=r1-r2

5th and r2,r6,r1 ; r2=r2 and r1 r6 is initially 0

6th orr r2,r8,r1 ; r8 is initially 0 orring gives the r1

7th ldr r1,[r9,#47]

8th lsl r2,r1,#3 ; meaning of that multiply with 8

9th lsr r2,r1,#3 ;meaning of that divide by 8

10th cmp r3,r10 ;r3-r10 then set the flags

11th str r1,[r11,#0] ;store r1 to the 0th memory location

12th ldr r2,[r10,#0] ;load r2 with the 0 th memory location to check str is working or not

13th str r1,[r11,#10] ;store r1 to the 10th memory location

14th ldr r2,[r10,#10] ;load r2 with the 10 th memory location to check str is working or not

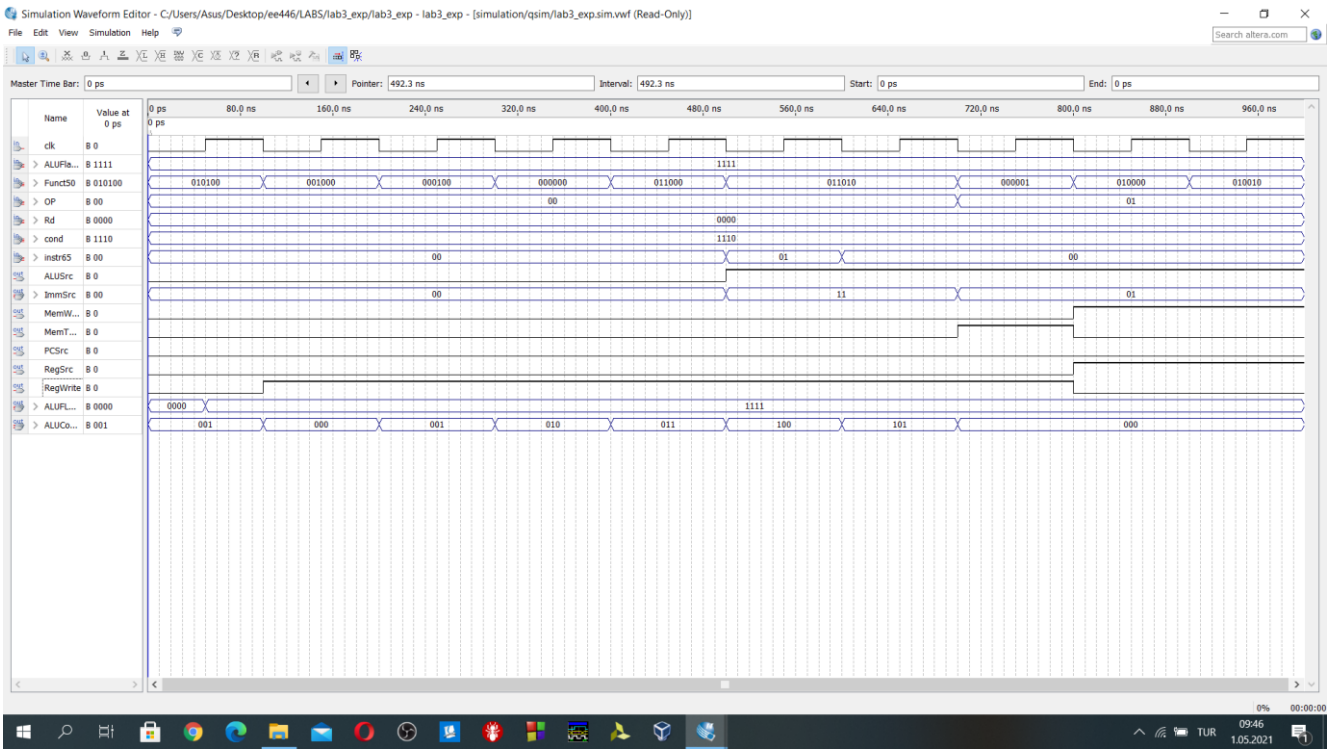


Figure 2:Controller waveform simulation

In controller design input control signals are as the following:

ALUflags[3:0], instr[31:28](cond), instr[27:26](OP), instr[25:20](Funct50), instr[15:12](Rd), instr[6:5](instr65),

Output of the controller and comments on the results:

//instr[25]=I

//instr[20]=S

ALUSrc: ALUSrc is equal to 0 during the data processing instructions when the instr[25] is equal to 0.(no immediate) . ALUSrc is equal to 1 during shifting and memory instructions.

ImmSrc: Specifies the extend proportion

ImmSrc=00 during add, sub, orr, and, cmp instructions

ImmSrc=11 during lsl,lsr instructions

ImmSrc=01 during str, ldr instructions

MemWrite: It is equal to 1 only str instruction. Otherwise, it equals to the 0.

MemToReg:It equals to 1 only memory read operation which is ldr. Otherwise, it equals to the 0.

PCSrc: It equals always 0 because PC increases 4 by 4. There is no other PC source in design.

RegSrc: It equals to 1 only str instructions. Otherwise 0.

ALUflags: They are updated in the first cycle because cmp is executed. Afterwards, there is no update the flags because instr[20]=S is equals to 0. That means no update for the flags.

ALUControl: It specifies the ALU operation.

In Figure 2,

1. 100 ns cmp→ ALUControl=001//subtraction
2. 100 ns add→ ALUControl=000//addition
3. 100 ns sub→ ALUControl=001// subtraction
4. 100 ns and→ ALUControl=010 //and
5. 100 ns orr→ ALUControl=011 //orr
6. 100 ns lsr→ ALUControl=100 //shift right
7. 100 ns lsl→ ALUControl=101 //shift left
8. 100 ns ldr→ ALUControl=000 //load operation
9. 100 ns str→ ALUControl=000 //store operation

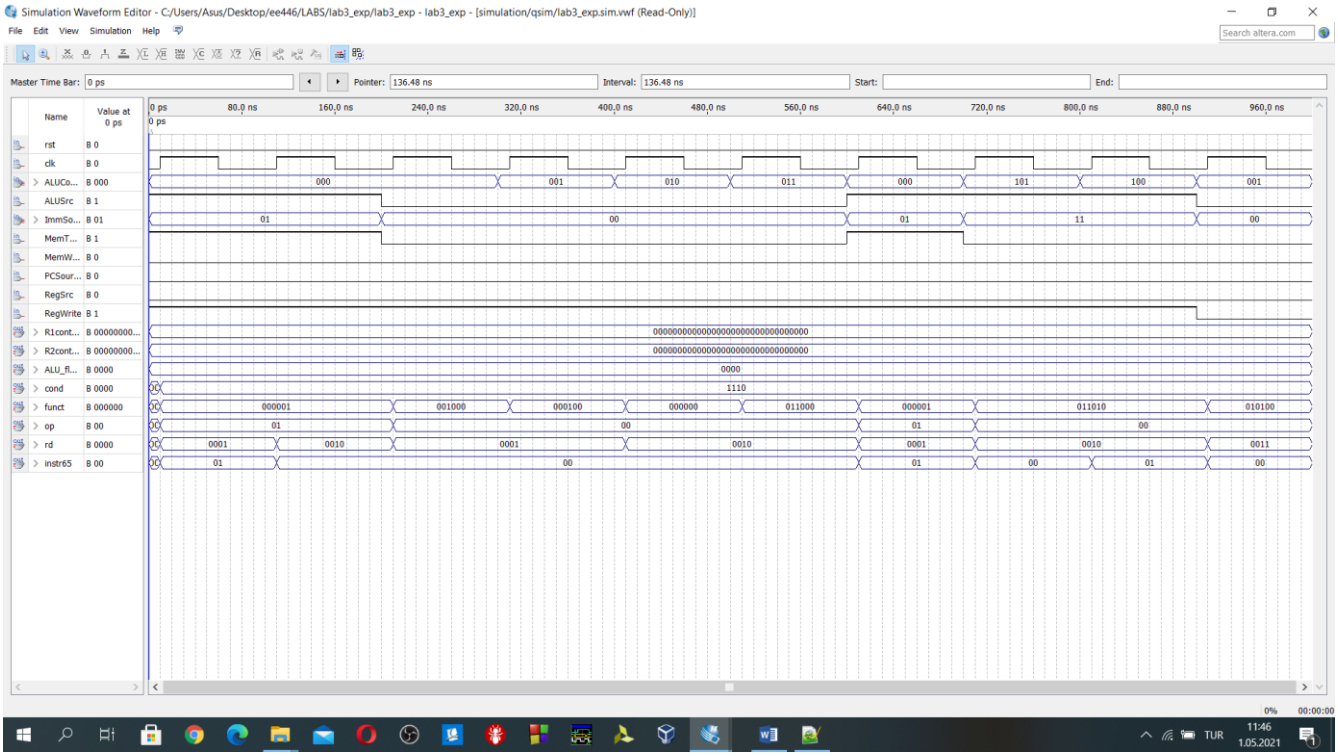


Figure 3:Memory content is 0

Then memory is loaded to load registers with ldr instruction.

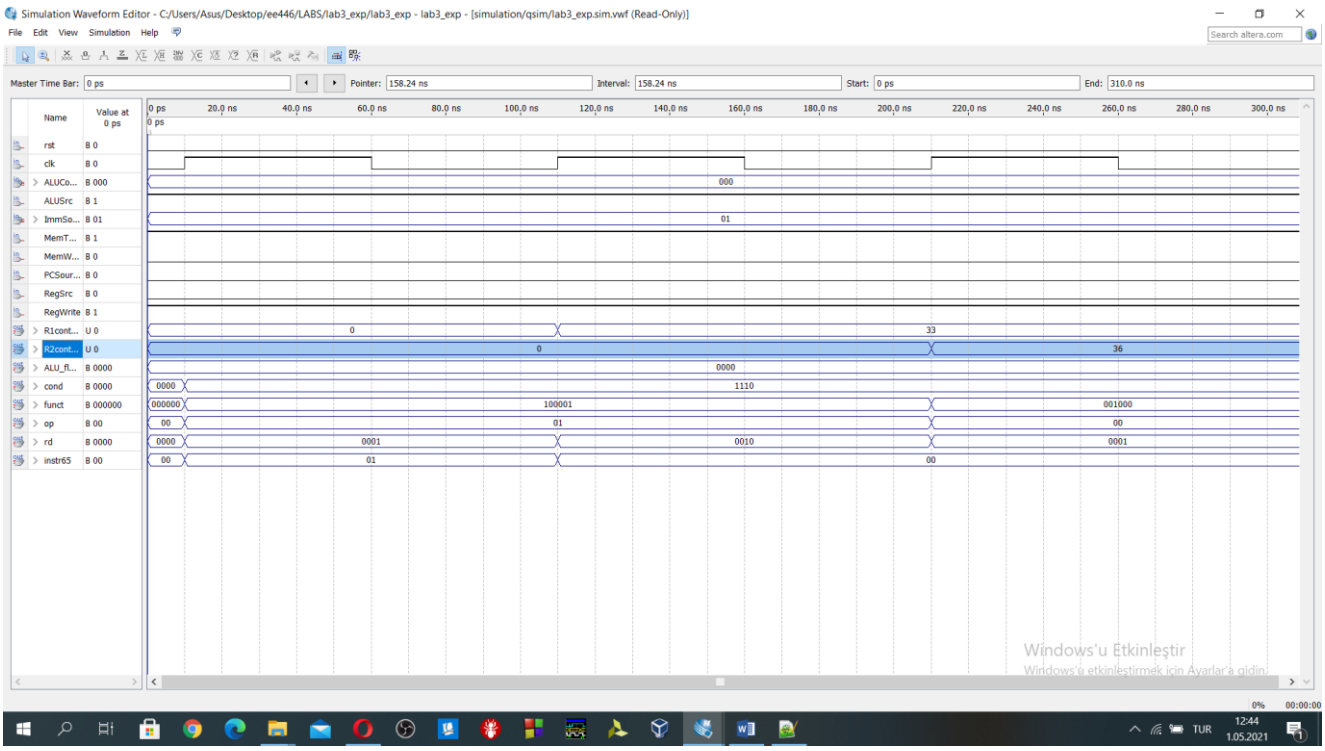


Figure 4: Loading R1 and R2 simulation result(LDR instruction)

In Figure 4 R1 is loaded with 33, and R2 is loaded with 36 according to the specified in

ldr r1, [r4, #33] //r4 content is initially 0

ldr r2, [r1, #3] // r2 is loaded with the memory location [r1] +3. memory location #36 content is 36. Therefore, r2 is loaded with 36 at at the end of the second clock cycle.

PCSource=0

Rst=0

MemToReg=1

MemWrite=0

RegWrite=1

ALUSrc=1

RegSrc=0

ALUControl=3’b000

ImmSource=2’b01

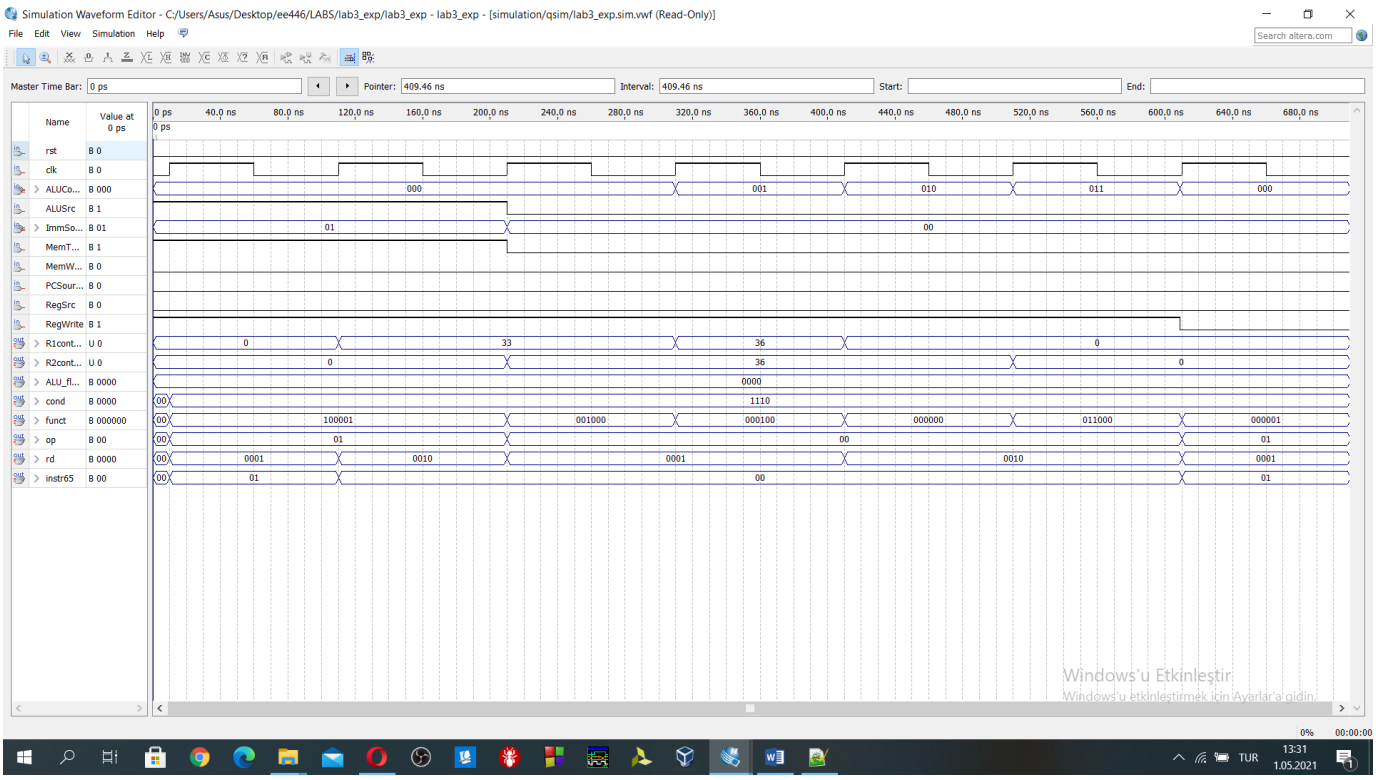


Figure 5:add-sub-and-orr instructions simulation

In figure 5 following instructions are executed.

3rd inst. add r1, r2, r7; r1=r2+r7 (210ns-310ns)

R2 content is 36 and r7 content is 0. After adding these registers' values, r1 is updated in the next clock cycle r1=0+36. After 310 ns, also r1's value will be 36.

4th inst. sub r1, r1, r2; r1=r1-r2 (310ns-410ns)

Beginning of the sub instructions r1 and r2 values are 36.

After r1-r2=0, r1 is updated as 0. After update, r1 value can be seen at the time interval 410ns-510ns.

5th inst. and r2, r6, r1; r2=r2 and r1 r6 is initially 0 (410ns-510ns)

R1 value is 0. R6 value also 0 because no value is loaded to r6. After the instruction r2 is loaded with 0. Because 36&0 gives 0 as a result

6th inst. orr r2, r8, r1; r8 is initially 0 orring gives the r1 (510ns-610ns)

R1 value is 0. Also r8 value is 0. Then orr operations give 0.

R2 is loaded with 0. That means r2 value stays constant in the next clock cycle as can be seen in Figure 5 (610ns-710ns)

Note: register values are updated after the instruction is executed because we need posedge of the clk to register load.

According to the specified controller input in the figure 5 instructions are executed.

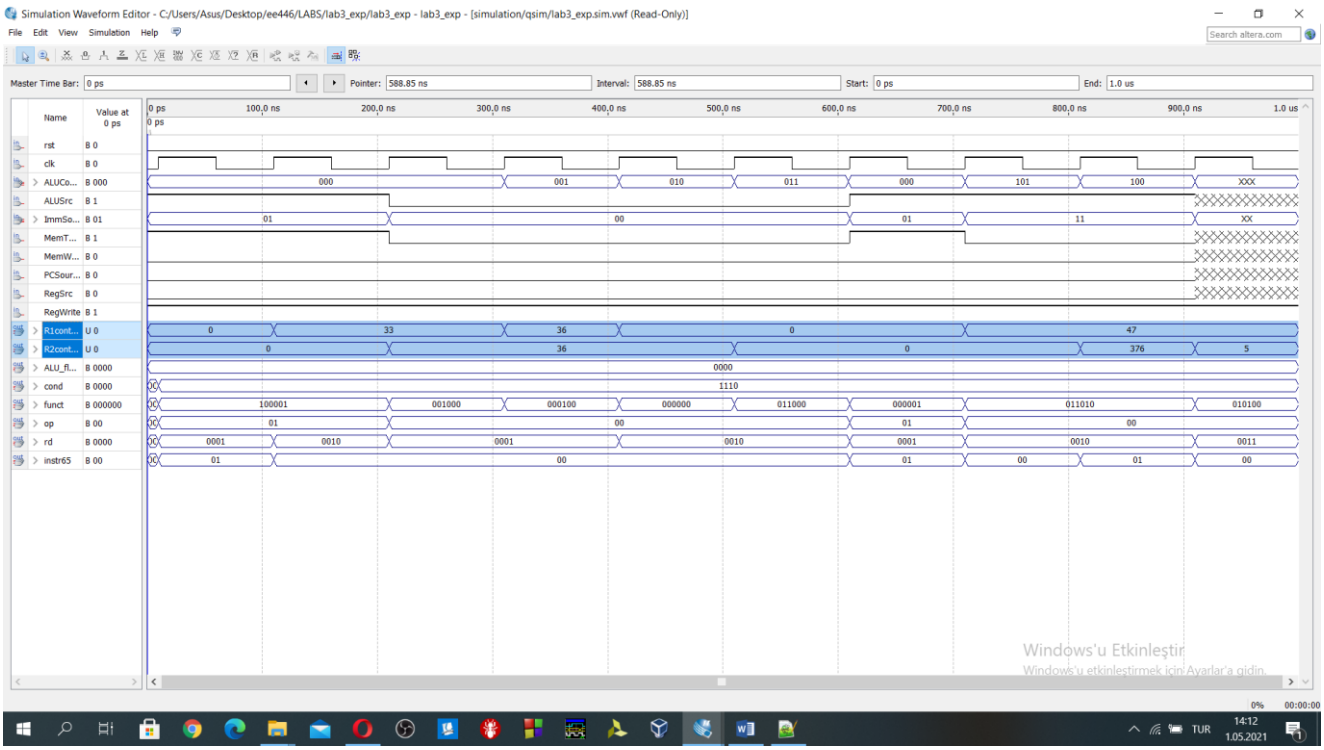


Figure 6:ldr-lsl-lsr instruction execution

7th inst. ldr r1,[r9,#47] (610ns-710ns)

Load from the memory occurs. R1 is updated at the end of the 710ns as 47

8th inst. lsl r2,r1,#3 ; meaning of that multiply with 8 (710ns-810ns)

R1 value is shifted to the left by 3. ImmSrc value is 11. ALUControl=101, ALUSrc=1, RegWrite=1.

The updated result can be seen in the figure 6 at the end of the 810ns.

Let's check 47*8=376 that is correct.

9th inst. lsr r2, r1, #3; meaning of that divide by 8 (810ns-910ns)

R1 value is shifted to the right by 3. ImmSrc value is 11. ALUControl=100, ALUSrc=1, RegWrite=1.

The updated result can be seen in the figure 6 at the end of the 910ns.

Let's check 47/8=5.9 that is correct.

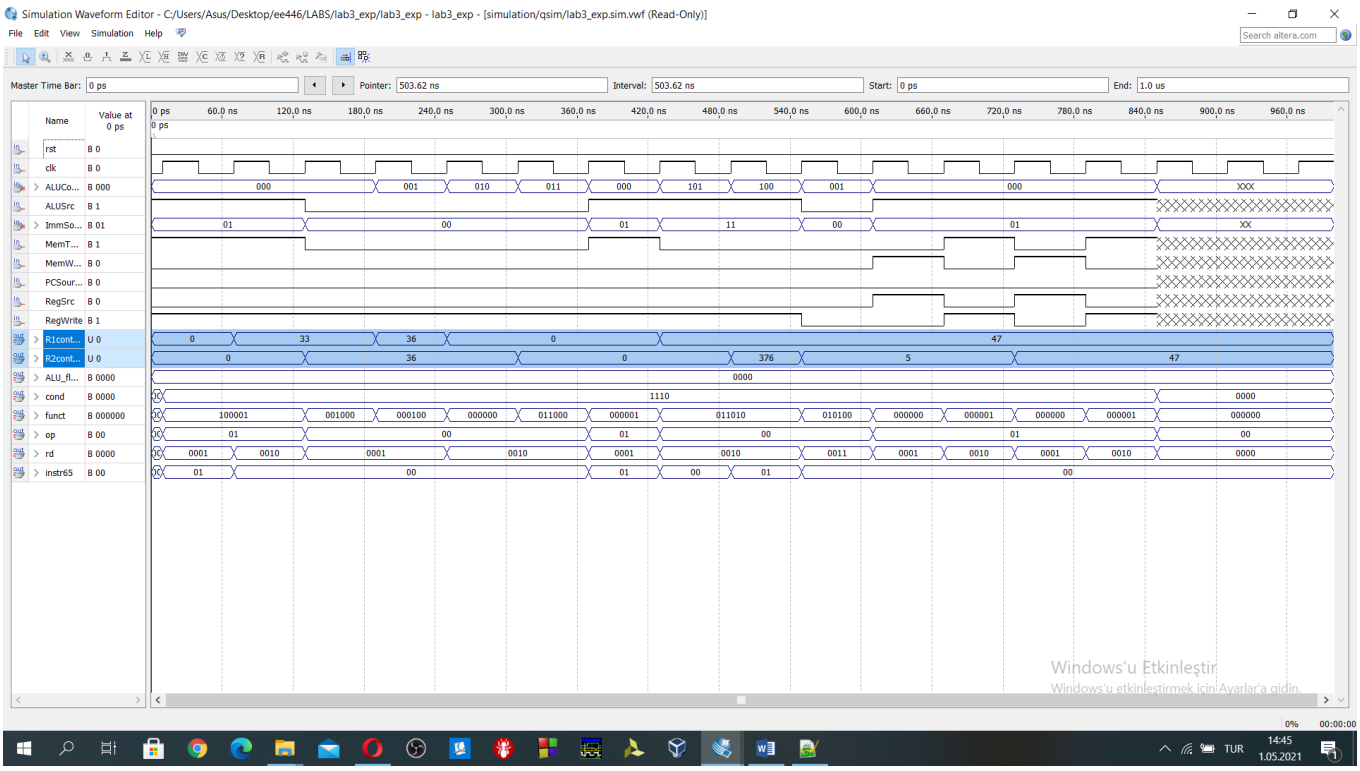


Figure 7:str-ldr-cmp instructions simulation

10th inst. cmp r3,r10 ;r3-r10 (550 ns – 610 ns)

During the cmp instruction r1 and r2 content does not change. Also, the result does not updated because RegWrite is equal to 0.

11th inst. str r1, [r11, #0]; store r1 to the 0th memory location (610 ns – 670 ns)

R1 value stored to the location #0 of the memory at the end of the 670 ns. There is no change on the value of the r1 and r2. However, in the memory the 0th location contains the stored value which is 47 as in the Figure 7.

12th inst. ldr r2, [r10, #0]; load r2 with the 0 th memory (670 ns – 730 ns)

R2 value loaded with the value at the location #0 of the memory at the end of the 730 ns. Now R2 value is also 47 as in the Figure 7.

13th inst. str r1, [r11, #10]; store r1 to the 10th memory location (730 ns – 790 ns)

R1 value stored to the location #10 of the memory at the end of the730 ns. There is no change on the value of the r1 and r2. However, in the memory the 10th location contains the stored value which is 47 as in the Figure 7.

14th inst. ldr r2, [r10, #10]; load r2 with the 10 th memory location (790 ns – 850 ns)

R2 value loaded with the value at the location #10 of the memory at the end of the 850 ns. Now R2 value is also 47 as in the Figure 7. It does not change.

```
module lab3_exp(
input clk,
input rst,
output wire [3:0] Flag_outputs,
output wire [31:0] R1reg,
output wire [31:0] R2reg
);
 wire   w_PCSource;
 wire   w_MemToReg;
 wire   w_MemWrite;
 wire   w_RegWrite;
 wire   w_ALUSrc;
 wire   w_RegSrc;
 wire   [2:0] w_ALUControl;
 wire   [1:0] w_ImmSource;
 wire   [3:0] w_ALU_flags;
 wire   [3:0] w_cond;
 wire   [5:0] w_funct;
 wire   [1:0] w_instr65;
 wire   [1:0] w_op;
 wire   [3:0] w_rd;

controller_singleCycle my_controller(
    .OP(w_op),
      .Funct50(w_funct),
        .cond(w_cond),
        .ALUFlagsIn(w_ALU_flags),
        .Rd(w_rd),
        .instr65(w_instr65),
        .clk(clk),
      .ImmSrc(w_ImmSource),
      .PCSrc(w_PCSource),
        .RegWrite(w_RegWrite),
        .MemWrite(w_MemWrite),
        .MemToReg(w_MemToReg),
        .ALUSrc(w_ALUSrc),
        .RegSrc(w_RegSrc),
        .ALUControl(w_ALUControl),
        .ALUFLAGSout(Flag_outputs)
    );

datapath_singleCycle my_datapath(
    .PCSource(w_PCSource),
    .rst(rst),
    .clk(clk),
    .MemToReg(w_MemToReg),
    .MemWrite(w_MemWrite),
    .RegWrite(w_RegWrite),
    .ALUSrc(w_ALUSrc),
    .RegSrc(w_RegSrc),
    .ALUControl(w_ALUControl),
    .ImmSource(w_ImmSource),
    .ALU_flags(w_ALU_flags),
    .cond(w_cond),
    .funct(w_funct),
    .instr65(w_instr65),
    .op(w_op),
    .rd(w_rd),
    .R1content(R1reg),
    .R2content(R2reg)
);
endmodule

//////////////**DATAPATH**/////////////////
```

```
//main code
//data memory at 178 line
//ext imm 218
//inst mem 255
//register file 296
module datapath_singleCycle(
    PCSource,
    rst,
    clk,
    MemToReg,
    MemWrite,
    RegWrite,
    ALUSrc,
    RegSrc,
    ALUControl,
    ImmSource,
    ALU_flags,
    cond,
    funct,
    instr65,
    op,
    rd,
    R1content,
    R2content
);


input wire  PCSource;
input wire  rst;
input wire  clk;
input wire  MemToReg;
input wire  MemWrite;
input wire  RegWrite;
input wire  ALUSrc;
input wire  RegSrc;
input wire  [2:0] ALUControl;
input wire  [1:0] ImmSource;
output wire [3:0] ALU_flags;
output wire [3:0] cond;
output wire [5:0] funct;
output wire [1:0] instr65;
output wire [1:0] op;
output wire [3:0] rd;
output  [31:0] R1content;
output  [31:0] R2content;

wire    [3:0] A2;
wire    [3:0] ALUflags;
wire    [31:0] I0;
wire    [31:0] Instruction;
wire    [31:0] out;
wire    [31:0] REGout;
wire    [31:0] SYNTHESIZED_WIRE_0;
wire    [31:0] SYNTHESIZED_WIRE_1;
wire    [31:0] SYNTHESIZED_WIRE_2;
wire    [31:0] SYNTHESIZED_WIRE_12;
wire    [31:0] SYNTHESIZED_WIRE_4;
wire    [31:0] SYNTHESIZED_WIRE_5;
wire    [31:0] SYNTHESIZED_WIRE_6;
wire    [31:0] SYNTHESIZED_WIRE_13;
wire    [31:0] SYNTHESIZED_WIRE_10;
wire    [31:0] SYNTHESIZED_WIRE_11;
```

```
simpleREG   b2v_inst(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_0),
    .REGout(REGout));
    defparam    b2v_inst.W = 32;


ALU b2v_inst10(
    .A(SYNTHESIZED_WIRE_1),
    .ALUcontrol(ALUControl),
    .B(SYNTHESIZED_WIRE_2),
    .Y(I0));
    defparam    b2v_inst10.W = 32;


adder32bits b2v_inst11(
    .A(SYNTHESIZED_WIRE_12),
    .B(SYNTHESIZED_WIRE_4),
    .Y(SYNTHESIZED_WIRE_6));


adder32bits b2v_inst12(
    .A(REGout),
    .B(SYNTHESIZED_WIRE_5),
    .Y(SYNTHESIZED_WIRE_12));


ConstantValueGenerator  b2v_inst13(
    .Data_on_Bus(SYNTHESIZED_WIRE_5));
    defparam    b2v_inst13.BUS_DATA = 3'b100;
    defparam    b2v_inst13.DATA_WIDTH = 32;


ConstantValueGenerator  b2v_inst14(
    .Data_on_Bus(SYNTHESIZED_WIRE_4));
    defparam    b2v_inst14.BUS_DATA = 3'b100;
    defparam    b2v_inst14.DATA_WIDTH = 32;


InstructionMemory   b2v_inst2(
    .CurrentPC(REGout),
    .Instruction(Instruction));


RegisterFile    b2v_inst3(
    .clk(clk),
    .rst(rst),
    .WE(RegWrite),
    .A1(Instruction[19:16]),
    .A2(A2),
    .A3(Instruction[15:12]),
    .R15(SYNTHESIZED_WIRE_6),
    .WD3(out),
    .RD1(SYNTHESIZED_WIRE_1),
    .RD2(SYNTHESIZED_WIRE_13),
    .R1(R1content),
    .R2(R2content)
    );


dataMemory  b2v_inst4(
    .clk(clk),
```

```
    .memWE(MemWrite),
    .memA(I0),
    .memWD(SYNTHESIZED_WIRE_13),
    .memRD(SYNTHESIZED_WIRE_11));


extImm  b2v_inst5(
    .ImmSrc(ImmSource),
    .Instr110(Instruction[11:0]),
    .extendedImm(SYNTHESIZED_WIRE_10));


muxWTwoToOne    b2v_inst6(
    .s0(PCSource),
    .I0(SYNTHESIZED_WIRE_12),
    .I1(out),
    .out(SYNTHESIZED_WIRE_0));
    defparam    b2v_inst6.W = 32;


muxWTwoToOne    b2v_inst7(
    .s0(RegSrc),
    .I0(Instruction[3:0]),
    .I1(Instruction[15:12]),
    .out(A2));
    defparam    b2v_inst7.W = 4;


muxWTwoToOne    b2v_inst8(
    .s0(ALUSrc),
    .I0(SYNTHESIZED_WIRE_13),
    .I1(SYNTHESIZED_WIRE_10),
    .out(SYNTHESIZED_WIRE_2));
    defparam    b2v_inst8.W = 32;


muxWTwoToOne    b2v_inst9(
    .s0(MemToReg),
    .I0(I0),
    .I1(SYNTHESIZED_WIRE_11),
    .out(out));
    defparam    b2v_inst9.W = 32;

assign  cond[3:0] = Instruction[31:28];
assign  funct[5:0] = Instruction[25:20];
assign  instr65[1:0] = Instruction[6:5];
assign  op[1:0] = Instruction[27:26];
assign  rd[3:0] = Instruction[15:12];

endmodule


//DATA MEMORY
 module dataMemory
 (      // input ports
      input                 clk,
      input     [31:0]      memA,//memory address according to the address write or read occur
      input     [31:0]      memWD,//memory write data, it specifies the memory data which can
be written
      input                 memWE,//memory write enable
      // output port
      output    [31:0]      memRD
 );

      reg        [31:0]          DATAmem  [255:0];
```

```
        //initialize the memory

        integer i;
      initial
          begin
           for(i=0;i<256;i=i+1)
                DATAmem[i] <= i;
          end
            /*
        initial begin
        $readmemh("memory.txt",DATAmem,0,255);
        end
                    */



      always @(posedge clk) begin
           if (memWE)
              begin
                DATAmem[memA] <= memWD;
              end
      end

      assign memRD = DATAmem[memA]; //it provides the data which is specified by the address

 endmodule

/*EXTIMM*/
module extImm
(
//input port
input  [1:0]    ImmSrc,
input  [11:0]   Instr110,
//output port
output reg [31:0]  extendedImm
);


always @(*)
begin
case(ImmSrc)
    2'b00://for data processing imm8
    begin
        extendedImm= {24'b0,Instr110[7:0]};
    end

    2'b01://for ldr and strong0
    begin
        extendedImm= {20'b0,Instr110[11:0]};
    end

    2'b11://for lsr and lsl
    begin
        extendedImm= {27'b0,Instr110[11:7]};
    end
    default
    begin
        extendedImm={24'b0,Instr110[7:0]};
    end
endcase
end

endmodule
```

```
/*instruction memory*/
module InstructionMemory
 (
     input          [31:0]     CurrentPC,
     output          [31:0]     Instruction
 );


       wire [31:0] counterInst;
     reg [31:0] instr[15:0];
       //if the PC 0 then implies first 32 bit instruction
       //if PC 4 then second instruction
       //PC 8 3 instruction
     //initialize the instructions
       initial
     begin              //always/op/I/cmd/S/rn/rd/src2
                 instr[1]    <=  32'b1110_0110_0001_0100_0001_0000_0010_0001;//ldr load r1
                 instr[2] <=  32'b1110_0110_0001_0001_0010_0000_0000_0011;//ldr load r2
             instr[3] <=  32'b1110_0000_1000_0010_0001_0000_0000_1001; //add
                  instr[4] <=  32'b1110_0000_0100_0001_0001_0000_0000_0010; //subtract
                  instr[5] <=  32'b1110_0000_0000_0110_0010_0000_0000_0001; //and
                  instr[6] <=  32'b1110_0001_1000_1000_0010_0000_0000_0001; //orr
                  instr[7] <=  32'b1110_0100_0001_1001_0001_0000_0010_1111;//ldr load
                  instr[8] <=  32'b1110_0001_1010_0001_0010_0001_1000_0000; //lsl
                  instr[9] <=  32'b1110_0001_1010_0001_0010_0001_1010_0000; //lsr
                  instr[10] <= 32'b1110_0001_0100_0000_0011_0000_0000_1010; //cmp
                  instr[11] <= 32'b1110_0100_0000_1011_0001_0000_0000_0001; //str
                  instr[12] <= 32'b1110_0100_0001_1010_0010_0000_0000_0001; //ldr
                  instr[13] <= 32'b1110_0100_0000_1011_0001_0000_0000_1010; //str
                  instr[14] <= 32'b1110_0100_0001_1010_0010_0000_0000_1010; //ldr
                  instr[15] <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;


       end

       assign counterInst={2'b00,CurrentPC[31:2]}; //it gives the instruction number because
PC increases 4 by 4
     assign Instruction = instr[counterInst];
 endmodule


 /*register File*/
 module RegisterFile
 (   //input ports
     input                   clk,
     input                   rst,
     input                   WE,  //write enable signal
     input          [3:0]     A1,
       input          [3:0]     A2,
       input          [3:0]     A3,
     input          [31:0]    WD3,
     //output ports
     output         [31:0]    RD1,
     output         [31:0]    RD2,
       input           [31:0]   R15,
       //for the demostration purpose
       output         [31:0]    R1,
       output         [31:0]    R2
 );


    reg  [31:0]   register_R [15:0]; //31 bits width and 16 bits length
    initial
    begin
```

```
                    register_R[0] = 32'b0;
                    register_R[1] = 32'b0;
                    register_R[2] = 32'b0;
                    register_R[3] = 32'b0;
                    register_R[4] = 32'b0;
                    register_R[5] = 32'b0;
                    register_R[6] = 32'b0;
                    register_R[7] = 32'b0;
                    register_R[8] = 32'b0;
                    register_R[9] = 32'b0;
                    register_R[10]= 32'b0;
                    register_R[11] = 32'b0;
                    register_R[12] = 32'b0;
                    register_R[13] = 32'b0;
                    register_R[14] = 32'b0;
                    register_R[15] = 32'b0;
    end

      always @ (posedge clk or posedge rst) begin

           if(rst) begin
               register_R[0] <= 32'b0;
                    register_R[1] <= 32'b0;
                    register_R[2] <= 32'b0;
                    register_R[3] <= 32'b0;
                    register_R[4] <= 32'b0;
                    register_R[5] <= 32'b0;
                    register_R[6] <= 32'b0;
                    register_R[7] <= 32'b0;
                    register_R[8] <= 32'b0;
                    register_R[9] <= 32'b0;
                    register_R[10] <= 32'b0;
                    register_R[11] <= 32'b0;
                    register_R[12] <= 32'b0;
                    register_R[13] <= 32'b0;
                    register_R[14] <= 32'b0;
                    register_R[15] <= 32'b0;
           end

           else
              begin
                if(WE) begin
                     register_R[A3] <= WD3;  //if WE is equal to 1 then corresponding register
is written
                end
           end

      end
      assign RD1 = register_R[A1];  //read data 1
      assign RD2 = register_R[A2];  //read data 2
        //to demostration
        assign R1  = register_R[1];
        assign R2  = register_R[2];

 endmodule




//it creates the constant value
module ConstantValueGenerator #(parameter DATA_WIDTH = 1,parameter BUS_DATA = 0) (
//port declerations
output wire [DATA_WIDTH-1:0] Data_on_Bus
```

```
);
//assign DATA_BUS to the bus
assign Data_on_Bus [DATA_WIDTH-1:0]=BUS_DATA;

endmodule

/*Arithmetic Logic Unit ALU*/
module ALU #(parameter W=32) (ALUcontrol,A,B,Y,N,Z,CO,OVF);
//negative and zero bits are affected by ALU op
//CO and OVF are affected by arithmetics
input [2:0] ALUcontrol;
input [W-1:0] A,B;
output reg [W-1:0] Y; //output
output reg CO,OVF,N,Z; //cpsr
wire [W-1:0] Bcomp=~B; //bitwise not
wire [W-1:0] Acomp=~A; //bitwise not
reg E;
always @(*)
begin
    case(ALUcontrol)
    3'b000: //add
    begin
        //update the overflow bit according to the signs
                {CO, Y} = A + B;
                    if (A[W-1] ~^ B[W-1]) //if the signs are the same
                        OVF = Y[W-1] ^ A[W-1];
                    else
                        OVF = 0;


    end
    3'b001: //subt a-b
    begin
//update the overflow bit according to the signs
                    {CO, Y} = A + Bcomp+1;
                    if ((A[W-1] ^ B[W-1]))
                        OVF = Y[W-1] ^ A[W-1];
                    else
                        OVF = 0;
    end
    3'b010:
    begin
            Y=A&B;   //and
            CO=0;
            OVF=0;

    end

        3'b011:
    begin

            Y=A|B;   //or
            CO=0;
            OVF=0;


    end

    3'b100:  //shift left lsr
    begin
           Y=A>>B;
    end

    3'b101:  //shift right lsl
    begin
```

```
            Y=A<<B;
    end

    3'b110:
    begin
            Y=A^B;   //exor
            CO=0;
            OVF=0;
    end


    3'b111:
    begin
            Y=A&Bcomp; //bit clear
            CO=0;
            OVF=0;
    end



    endcase
end


always @(*)
begin
        N = Y[W-1];
        Z = ~|Y;
end

endmodule




/*Simple register*/

module simpleREG #(parameter W=1) (DATA,clk,rst,REGout);
input rst,clk;
input [W-1:0] DATA;
output reg [W-1:0] REGout;
initial
begin
REGout=32'b0;
end

always @(posedge clk)
begin
    if(rst==1)
        begin
            REGout<=0;
        end
    else
        begin

            REGout<=DATA;
        end

end

endmodule

/*ADDER 32 BITS*/

module adder32bits(A,B,Y);
```

```
input [31:0] A;
input [31:0] B;
output [31:0] Y;
assign Y=A+B;

endmodule

/*
Implement a W-bit 2 to 1 and a W-bit 4 to 1 multiplexers, where W is a parameter specifying the
data width of the input.
*/
module muxWTwoToOne #(parameter W=1)(s0,I1,I0,out);

input [W-1:0] I1;
input [W-1:0] I0;
input s0;
output [W-1:0] out;

assign out=s0 ? I1 : I0;

endmodule


//////////****CONTROLLER****//////////
///1.option
//cond logic verilog code
 module controller_singleCycle_condLogic(
input [3:0] cond,
input [3:0] ALUFLa,
input [1:0] FlagW,
input  PCs,
input  RegW,
input  MemW,

output  PCSrc,
output  RegWrite,
output  MemWrite,
output  [3:0] ALUFLAGS,
input clk
);
//condex assumed as 1
reg [3:0] flagG;
reg condex;
always @(posedge clk)
begin
    if(condex==1)
        begin
         if(FlagW[1]==1)
            begin
                flagG[3:2]=ALUFLa[3:2];
            end

        else if(FlagW[0]==1)
            begin
                flagG[1:0]=ALUFLa[1:0];
            end
        else
         begin
               flagG=flagG;
         end
    end
        else
        begin
            flagG=flagG;
```

```
        end

end

assign ALUFLAGS=flagG;
always @(*)
begin
        case(cond)//nzcv aluflags[3:0]
        //z
        4'b0000: condex= ALUFLAGS[2];//equal
        4'b0001: condex= ~ALUFLAGS[2]; //not equal
        //c
        4'b0010: condex= ALUFLAGS[1]; //carry set
        4'b0011: condex= ~ALUFLAGS[1];
        //n
        4'b0100: condex= ALUFLAGS[3];
        4'b0101: condex= ~ALUFLAGS[3];
        //v
        4'b0110: condex= ALUFLAGS[0];
        4'b0111: condex= ~ALUFLAGS[0];

        4'b1000: condex= ~ALUFLAGS[2] & ALUFLAGS[1];
        4'b1001: condex= ~ALUFLAGS[1] | ALUFLAGS[2];

        4'b1010: condex= ~(ALUFLAGS[3] ^ ALUFLAGS[0]);
        4'b1011: condex= ALUFLAGS[3] ^ ALUFLAGS[0];

        4'b1100: condex= (~(ALUFLAGS[3] ^ ALUFLAGS[0])) & ~ALUFLAGS[2];
        4'b1101: condex= ((ALUFLAGS[3] ^ ALUFLAGS[0])) | ALUFLAGS[2];
        4'b1110: condex= 1 ;

        endcase


end
assign  PCSrc= condex & PCs;
assign RegWrite= condex & RegW;
assign MemWrite= condex & MemW;



endmodule


//decoder
module controller_singleCycle_decoder(

//input ports
input [1:0] op,
input [5:0] funct,
input [3:0] rd,
input [1:0] instr65,

//output ports
output reg [1:0] FlagW,
output reg PCs,
output reg RegW,
output reg MemW,
output reg [1:0] ImmSrc,
output reg MemToReg,
output reg ALUSrc,
output reg RegSrc,
output reg [2:0] ALUControl
);
```

```
always @(*)
begin
    case(op)
    2'b00://data processing
    begin //funct25=I
            //funct20=S
        PCs=0;
        MemToReg=0;
        MemW=0;
        RegSrc=0;

        if(funct[4:1]==4'b1010)//cmp
            begin
            RegW=0;
            end
        else //otherwise
            begin
            RegW=1;
            end

        if(funct[4:1]==4'b1101)//shift
            begin
            ImmSrc=2'b11;
            ALUSrc=1'b1;
            end
        else //otherwise
            begin
            ImmSrc=2'b00;
            ALUSrc=funct[5];
            end


        case(funct[4:1])
                4'b0100://add
                begin
                        ALUControl=3'b000;
                        //S control
                            if(funct[0])
                                begin
                                    FlagW=2'b11;
                                end
                            else
                                begin
                                    FlagW=2'b00;
                                end
                end

                4'b0010://sub
                begin
                        ALUControl=3'b001;
                        //S control
                            if(funct[0])
                                begin
                                    FlagW=2'b11;
                                end
                            else
                                begin
                                    FlagW=2'b00;
                                end
                end

                4'b0000://and
                begin
                        ALUControl=3'b010;
                        //S control
```

```
                        if(funct[0])
                            begin
                                FlagW=2'b10;
                            end
                        else
                            begin
                                FlagW=2'b00;
                            end
                end

                4'b1100://orr
                begin
                    ALUControl=3'b011;
                        //S control
                            if(funct[0])
                            begin
                                FlagW=2'b10;
                            end
                        else
                            begin
                                FlagW=2'b00;
                            end
                end

                4'b1010://cmp
                begin
                    ALUControl=3'b001;
                        //S control
                            if(funct[0])
                            begin
                                FlagW=2'b11;
                            end
                        else
                            begin
                                FlagW=2'b00;
                            end
                end

                4'b1101://shift
                begin
                FlagW=2'b00;
                    case(instr65)
                        2'b00://shift left and I=0
                        begin
                        ALUControl=3'b101;
                        end
                        2'b01://shift right and I=0
                        begin
                        ALUControl=3'b100;
                        end
                    endcase

                end


        endcase
    end

    2'b01://memory
    begin
    FlagW=2'b00;
    PCs=0;
    ALUControl=3'b000;
    ALUSrc=1;
    ImmSrc=op;
```

```
    RegW=funct[0];
    MemW=~funct[0];
    RegSrc=~funct[0];

        case(funct[0])
            1'b1://load
             begin
                MemToReg=1;
             end
            1'b0://store
             begin
             end
        endcase

    end
    endcase
end




endmodule

module controller_singleCycle
    (
    //input ports
    input [1:0] OP,
      input [5:0] Funct50,
        input [3:0] cond,
        input [3:0] ALUFlagsIn,
        input [3:0] Rd,
        input [1:0] instr65,
        input clk,
    //output ports
      output  [1:0] ImmSrc,
      output  PCSrc,
        output  RegWrite,
        output  MemWrite,
        output  MemToReg,
        output  ALUSrc,
        output  RegSrc,
        output  [2:0] ALUControl,
        output  [3:0] ALUFLAGSout
    );
    wire [1:0] w_FlagW;
    wire w_PCS;
    wire w_regW;
    wire w_memW;

//combine the controller submodules into one module
    controller_singleCycle_decoder
my_decoder(.op(OP),.funct(Funct50),.rd(Rd),.instr65(instr65),.FlagW(w_FlagW),.PCs(w_PCS),.RegW(
w_regW),.MemW(w_memW),.ImmSrc(ImmSrc),.MemToReg(MemToReg),.ALUSrc(ALUSrc),.RegSrc(RegSrc),.ALUC
ontrol(ALUControl));


    controller_singleCycle_condLogic
my_condlog(.cond(cond),.ALUFLa(ALUFlagsIn),.FlagW(w_FlagW),.PCs(w_PCS),.RegW(w_regW),.MemW(w_me
mW),.PCSrc(PCSrc),.RegWrite(RegWrite),.MemWrite(MemWrite),.ALUFLAGS(ALUFLAGSout),.clk(clk));


 endmodule


 /////////****CONTROLLER****/////////
```

```
///2.option
 module controller_singleCycle
    (
    //input ports
        input [1:0] OP,
        input [5:0] Funct50,
        input [3:0] cond,
        input [3:0] ALUFlagsIn,
        input [3:0] Rd,
        input [1:0] instr65,
        input clk,
    //output ports
        output  reg [1:0] ImmSrc,
        output  reg PCSrc,
        output  reg RegWrite,
        output  reg MemWrite,
        output  reg MemToReg,
        output  reg ALUSrc,
        output  reg RegSrc,
        output  reg [2:0] ALUControl,
        output  reg [3:0] ALUFLAGSout
    );
    reg [1:0] FlagW;
    always @(posedge clk)
begin
        if(FlagW==2'b10)
            begin
                ALUFLAGSout[3:2]=ALUFlagsIn[3:2];
            end

        else if(FlagW==2'b01)
            begin
                ALUFLAGSout[1:0]=ALUFlagsIn[1:0];
            end

        else if(FlagW==2'b11)
            begin
                ALUFLAGSout=ALUFlagsIn;
            end

end



    always @(*)
    begin
    case(OP)
    2'b00://data processing
    begin //funct25=I
            //funct20=S
        PCSrc=0;
        MemToReg=0;
        MemWrite=0;
        RegSrc=0;

        if(Funct50[4:1]==4'b1010)//cmp
            begin
            RegWrite=0;
            end
        else //otherwise
            begin
            RegWrite=1;
            end

        if(Funct50[4:1]==4'b1101)//shift
```

```
            begin
            ImmSrc=2'b11;
            ALUSrc=1'b1;
            end
        else //otherwise
            begin
            ImmSrc=2'b00;
            ALUSrc=Funct50[5];
            end


        case(Funct50[4:1])
                4'b0100://add
                begin
                        ALUControl=3'b000;
                        //S control
                            if(Funct50[0])
                                begin
                                    FlagW=2'b11;
                                end
                            else
                                begin
                                    FlagW=2'b00;
                                end
                end

                4'b0010://sub
                begin
                        ALUControl=3'b001;
                        //S control
                            if(Funct50[0])
                                begin
                                    FlagW=2'b11;
                                end
                            else
                                begin
                                    FlagW=2'b00;
                                end
                end

                4'b0000://and
                begin
                        ALUControl=3'b010;
                        //S control
                            if(Funct50[0])
                                begin
                                    FlagW=2'b10;
                                end
                            else
                                begin
                                    FlagW=2'b00;
                                end
                end

                4'b1100://orr
                begin
                    ALUControl=3'b011;
                        //S control
                            if(Funct50[0])
                                begin
                                    FlagW=2'b10;
                                end
                            else
                                begin
                                    FlagW=2'b00;
```

```
                                end
                end

                4'b1010://cmp
                begin
                    ALUControl=3'b001;
                    FlagW=2'b11;

                end

                4'b1101://shift
                begin
                FlagW=2'b00;
                    case(instr65)
                        2'b00://shift left and I=0
                        begin
                        ALUControl=3'b101;
                        end
                        2'b01://shift right and I=0
                        begin
                        ALUControl=3'b100;
                        end
                    endcase

                end


        endcase
    end

    2'b01://memory
    begin
    FlagW=2'b00;
    PCSrc=0;
    ALUControl=3'b000;
    ALUSrc=1;
    ImmSrc=OP;
    RegWrite=Funct50[0];
    MemWrite=~Funct50[0];
    RegSrc=~Funct50[0];

        case(Funct50[0])
            1'b1://load
             begin
                MemToReg=1;
             end
            1'b0://store
             begin
                MemToReg=0;
             end
        endcase

    end
    endcase
end



 endmodule
```