Laboratory Work 4 Report

ISA & Datapath Design for Multi-Cycle CPU

Mustafa BIYIK

2231454

The report consists of four main parts.

The first part is answers of the preliminary work part as hard-copy. Hardcopy contains ISA configurations and the design.

The second part is verification results of the changed modules in wvf platform of Quartus.

The third part is the simulation result of the experimental part. Testbench is applied on the Modelsim.

The last part contains the codes of the Datapath design and its testbench.

1. Preliminary Work Part(Hard-copy)

1.2 ISA Configuration

⊗ supports → general purpose computing → contain different instructions

sufficient variety of addressing modes (direct, indirect, immediate)

⊗ own mnemonics for the instructions and define the ordering of the operands

group the instructions that will use the same path for execution

(a) Data processing instructions

- => addition
- => addition indirect
- => subtraction
- => subtraction indirect
- => and
- => or
- => xor
- => clear

(b) Shift instructions

- => rotate left
- => rotate right
- => shift left
- => arithmetic shift right
- => logical shift right

(c) Branch instructions

- => Branch unconditional
- => Branch with link
- => Branch indirect
- => Branch if zero
- => Branch if not zero
- => Branch if carry set
- => Branch if carry clear

(d) memory instructions

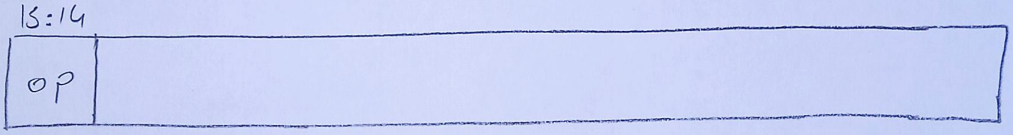
- => load to register from memory
- => load immediate to register
- => store from register to memory

Mnemonic	Name		Operation
ADD	Addition	add rA,rB,rC	rA ← rB + rC
ADDI	Addition indirect	add rA, rB, [#address]	rA ← rB + MEM[address]
SUB	Subtraction	sub rA,rB,rC	rA ← rB - rC
SUBI	subtraction indirect	sub rA,rB, [#address]	rA ← rB - MEM[address]
AND	logical and	and rA,rB,rC	rA ← rB & rC
ORR	logical or	orr rA,rB,rC	rA ← rB | rC
XOR	logical xor	xor rA,rB,rC	rA ← rB ^ rC
CLR	clear register	clr rA	rA ← 0
ROL	rotate left	rol rA	rA ← {rA[6:0], rA[7]}
ROR	rotate right	ror rA	rA ← {rA[0], rA[7:1]}
LSL	logical shift left	lsl rA	rA ← {rA[6:0], 0}
ASR	arith shift right	asr rA	rA ← {rA[7], rA[7:1]}
LSR	log. shift right	lsr rA	rA ← {0, rA[7:1]}
LDR	load register from mem.	ldr rA, [#address]	rA ← MEM[#address]
LDI	load imm. to reg.	ldi rA, #data	rA ← #data
STR	store from reg to mem.	str rA, [#address]	MEM[#address] ← rA
B	branch uncond.	add PC, (PC+8), imm8	PC ← (PC+8) + imm8
BL	branch with link	sub LR, (PC+8), 4 add PC, (PC+8), imm8	LR ← (PC+8) - 4; PC ← (PC+8) + imm8
BI	branch indirect	mov PC, [Rm]	PC ← [Rm]
BEQ	branch if zero	Z=1 then add PC, (PC+8), imm8	if Z=1, then PC ← PC+8+imm8
BNE	branch if not zero	Z=0 then add PC, (PC+8), imm8	if Z=0, then PC ← PC+8, +imm8
BC	branch if carry set	C=1 then add PC, (PC+8), imm8	if C=1 then, PC ← PC+8+imm8
BNC	branch if carry clear	C=0 then add PC, (PC+8), imm8	if C=0 then, PC ← PC+8 +imm8

* ordering of the representation important to understand more clearly

For example sub rA,rB,rC implies rA ← rB - rC } totally different operations
sub rA,rC,rB implies rA ← rC - rB } Therefore, operands order is important.

- register file consists of 8 registers. 1 LR, 7 general purpose registers
- PC register is separated from the register file
- no wired connection between IDM & ALP
- temporary registers can be used
- The length for the data/instruction memory is 16 bits however registers are 8 bits. Also, operations are conducted with 8 bits
- R0, R1, R2, R3, R4, R5 are multi-purpose registers
- R7 is link register, it is in the register file.
- R6 is PC, PC is seperated from the register file



op

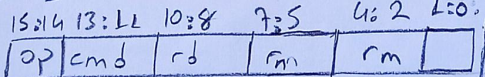
00 → DATA processing

01 → shift instructions

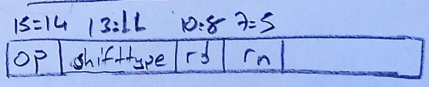
10 → memory instructions

11 → Branch instructions

DATA processing OP=00



shift inst. OP=01



cmd

000 → add rd, rn, rm

001 → addi rd, rn, MEM[inst 4:0]

010 → sub rd, rn, rm

011 → subi rd, rn, MEM[inst 4:0]

100 → and rd, rn, rm

101 → orr rd, rn, rm

110 → xor rd, rn, rm

111 → clr rd

shifttype "shift in & store at rd"

000 → rol rd, rn

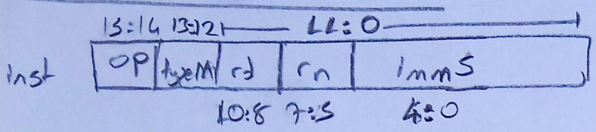
001 → ror rd, rn

010 → lsl rd, rn

011 → asr rd, rn

100 → lsr rd, rn

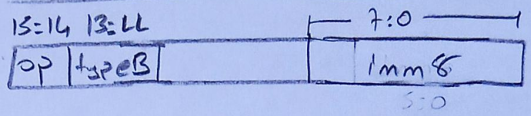
OP=10 memory instructions



typeM

- 00 → ldr rd, [rn, imm5]
- 01 → ldp rd, #inst7:0
- 10 → str rd, [rn, imm5]

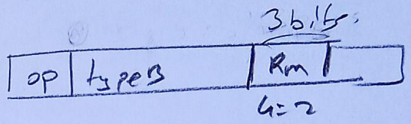
OP=11 Branch instructions



typeB

- 000 → B
- 001 → BL
- 010 → BI
- 011 → BEQ
- 100 → BNE
- 101 → BC
- 110 → BNC

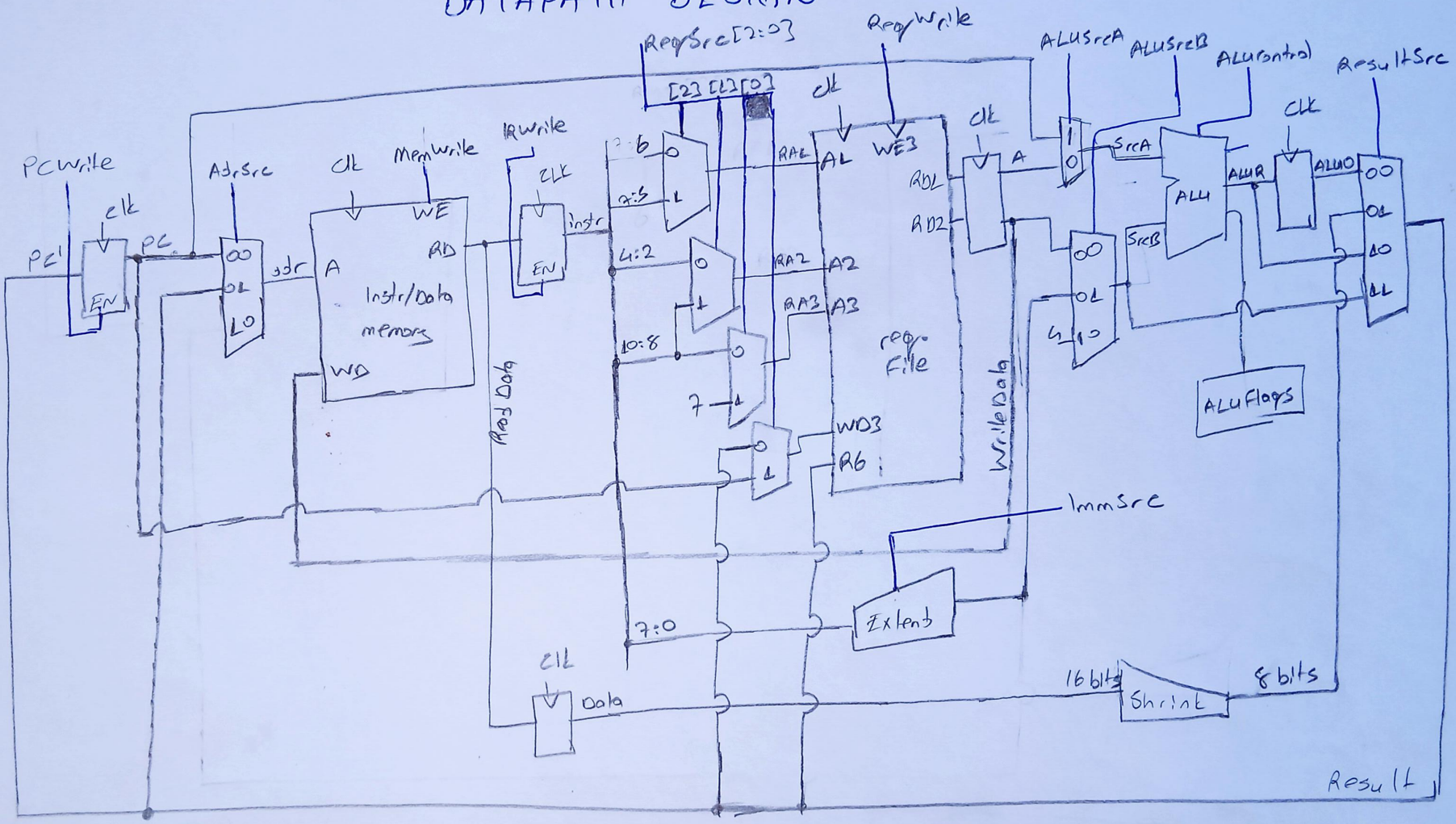
inst7:0 => imm8



- In indirect branch, the target address is specified indirectly either through memory or general purpose register
- Branch indirect takes "Rm" as its operand and causes a branch to address saved in Rm.
- In datapath ALU is updated, shift operations & data processing operations are conducted with the help of the ALU,

  ALU control control signal is provided from the control circuit.
- Just Data/Instruction memory word size is 16 bits. Other operation word sizes 8 bits.

DATAPATH DESIGN



1.2.7 Validation of Operation

(1) Explanation of each cycle & control signals

Instruction fetch (Cycle 1) (S0)

- Memory value pointed by the PC is loaded to Instruction register
- PC value is incremented by 4.

⇒ AdrSrc=00
⇒ PCWrite=1
⇒ ALUSrcA=1
⇒ ALUSrcB=10
⇒ ALUControl=4'b0000
⇒ ResultSrc=10
⇒ IRWrite=1

At the end of the clock cycle PC value incremented by 4.
M[PC] is in the instruction Regester end of the clock cycle

Decode (Cycle 2) (S1)

Rd, Rn, PC & are ready at Reg File. Inputs. Instruction in the instruction register is decoded.

- RegSrc 100
- PCWrite=0
- MemWrite=0
- IRWrite=0
- RegWrite=0
- ALUSrcA=1
- ALUSrcB=10
- ALUControl=00000(add)
- ResultSrc=10

- at the end of the clock cycle, values of the RD1, RD2 is loaded to the temporary registers.

- Data processing instructions => 4 cycles
  - Fetch, Decode, Execute, ALUWB
- Shift instructions => 4 cycles
  - Fetch, Decode, Execute, ALUWB
- Memory instructions: 3, 4 cycles or 5 cycles
  - LDI => 3 cycles: Fetch, Decode, ALUWBLDI
  - STR => 4 cycles: Fetch, Decode, MemAdr, MemWrite
  - LDR => 5 cycles: Fetch, Decode, MemAdr MemRead, MemWB
- Branch instructions
  - B, BL, BEQ, BNE, BC, BNC => 3 cycles
    Fetch, Decode, Branch
  - BLI => 3 cycles
    fetch, Decode, PCWrite
    - PCWrite: [Rm] value which is written in the register file written into PC.

Execute (Cycle 3 for Shift & Data) (S2)

- values of the temporary register is used for ALU operations

ALUSrcA=0
ALUSrcB=00
ALUop=1 (ALUControl from 0 to 11)
ResultSrc=00
RegWrite=0

the result is ready to register load at the end of the clock cycle.

ALUWB (cycle 4 for shifted Data) (S3)

- at the beginning of the clock cycle, in the entrance of the WD3, the result of the desired operation exists.

RegSrc[0]=0
RegWrite=1
ResultSrc=00

At the end of the clock cycle, data is written to specified register (Rd)

ALUWBLDI (cycle 3 for LDI instruction) (S4)

- ImmSrc=1, inst7:0 at the output of extend

The data value of the immediate bits directly loaded to Rd register.

ImmSrc=1
ALUSrcB=01
ResultSrc=11
RegSrc[0]=0
RegWrite=1

- load with immediate addressing is completed with that cycle. The data at the immediate part is loaded to the Rd register.

memAdr (cycle 3 for STR/LDR) (S5)

- the ImmS value in the instruction is added with Rn's value
- Then the added value is used to reach memory data

ImmSrc=0
ALUSrcB=01
ALUSrcA=0
ALUControl=0000

the memory addres calculated and loaded to temporary register
At the end of the clock cycle temporary register contains [Rn]+extend(imm)

MemWrite (Cycle 4 for STR) (56)

The stored value at the temporary register is selected with ResultSrc
Then value is stored to the memory.

ResultSrc=00
AdrSrc=01
MemWrite=1

(*) WD input comes from the temporary register

RegSrc in the decode stage must be 010 => For STR operation

At the end of the clock cycle Rd is loaded to MEM[Rn+imm5].

MemRead (Cycle 4 for LDR) (57)

with the calculated address in the (55) memory read occurs.

ResultSrc=00
AdrSrc=01
memWrite=0
the read data from the memory is loaded to DATA temporary register.

MemWB (Cycle 5 for LDR) (58)

the read value from the memory at the DATA temporary register beginning of the clock cycle
the data is loaded to Rd register at the end of the clock cycle with the help of the control signals.
Firstly 16 bits data is shrinked to the 8 bits data

ResultSrc=01
RegSrc[0]=0
RegWrite=1
} Rd is loaded with pointed value at the memory.

Branch (cycle 3 for B, BEQ, BNE, BC, BNC) (S9)

At the end of the clock cycle the calculated target address loaded to PC register

ALUSrcA=0

ALUSrcB=01

ALUcontrol=4'b0000 (add)

ResultSrc=10

Branch=1 then PCWrite=1

that means next cycle calculated branch target address is fetched

BL (cycle 3 for BL) (S10)

ALUSrcA=0

ALUSrcB=01

ALUcontrol=4'b0000 (add)

ResultSrc=10

RegWrite=1

Branch=1

Reg[0] = 1 while writing link register other procedures are the same as the "B".

BIPCwrite / BI (cycle 3 for BI) (SLL)

[Rm] value which is the value at the RD2's out temporary register is selected with ALUSrcB. Then, ResultSrc selects ALUSrcB content. Then, [Rm] value is written to PC with the help of the PCWrite.

ALUSrcB=00

ResultSrc=LL

PCWrite=1

(9) the end of the clock cycle [Rm] value is loaded to PC.

All in all;

(X) Branch instructions => 3 cycles

(X) memory instructions => 3, 4, 5 cycles (depends on the instruction)

(X) Shift & Data processing instructions => 4 cycles

2. Verification of the modules

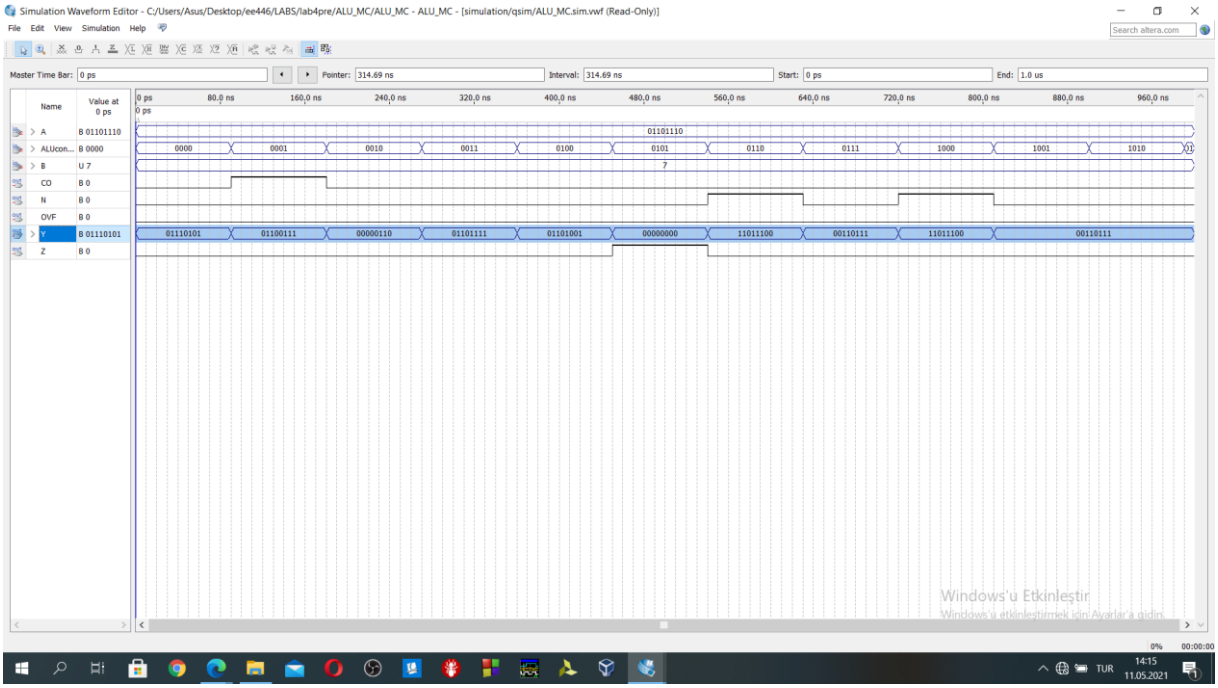


Figure 1: ALU simulation

In Figure 1, the simulation result of the ALU exists.

4'b0000: //add A (A)01101110+(B)00000111=01110101 matches with Y in the Figure 1(result)

4'b0001: //subt a-b (A)01101110-(B)00000111=01100111 matches with Y in the Figure 1(result)

4'b0010: //and (A)01101110&(B)00000111=00000110 matches with Y in the Figure 1(result)

4'b0011: //or (A)01101110|(B)00000111=01101111 matches with Y in the Figure 1(result)

4'b0100://xor (A)01101110+(B)00000111=01101001 matches with Y in the Figure 1(result)

4'b0101://clear clear then Y=0

4'b0110: //rol (A)01101110=11011100(Y) matches with Y in the Figure 1(result)

4'b0111://ror (A)01101110=00110111(Y) matches with Y in the Figure 1(result)

4'b1000: //shift left lsl (A)01101110=11011100(Y) matches with Y in the Figure 1(result)

4'b1001: //shift right lsr (A)01101110=00110111(Y) matches with Y in the Figure 1(result)

4'b1010 //arithmetic shift right (A)01101110=00110111 (Y) matches with Y in the Figure 1(result)

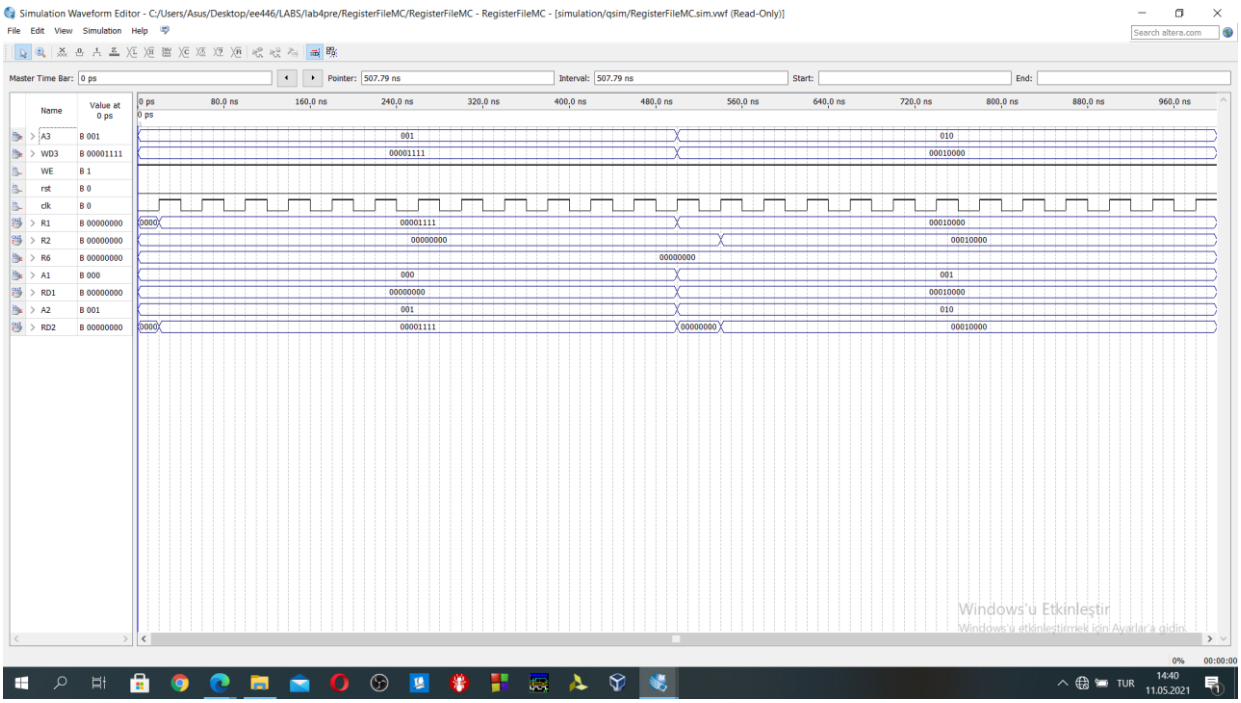


Figure 2:Register File Simulation

In Figure 2, register File for the Datapath design is designed. R1 and R2 are for the demonstration purposes. R6 is PC register. Firstly, R1(with A3) is loaded with 00001111 then the content of the R1 can be seen at the “R1” output. Also, when the A2=1, then RD2 value contains the current content of the R1 after the write operation of the R1. Then, both R1 and R2 is loaded with the 00010000 value, their value can be observed from the “R1” and “R2” outputs. They are also can be seen RD1 and RD2 when the A1 and A2 is 1 or 2.

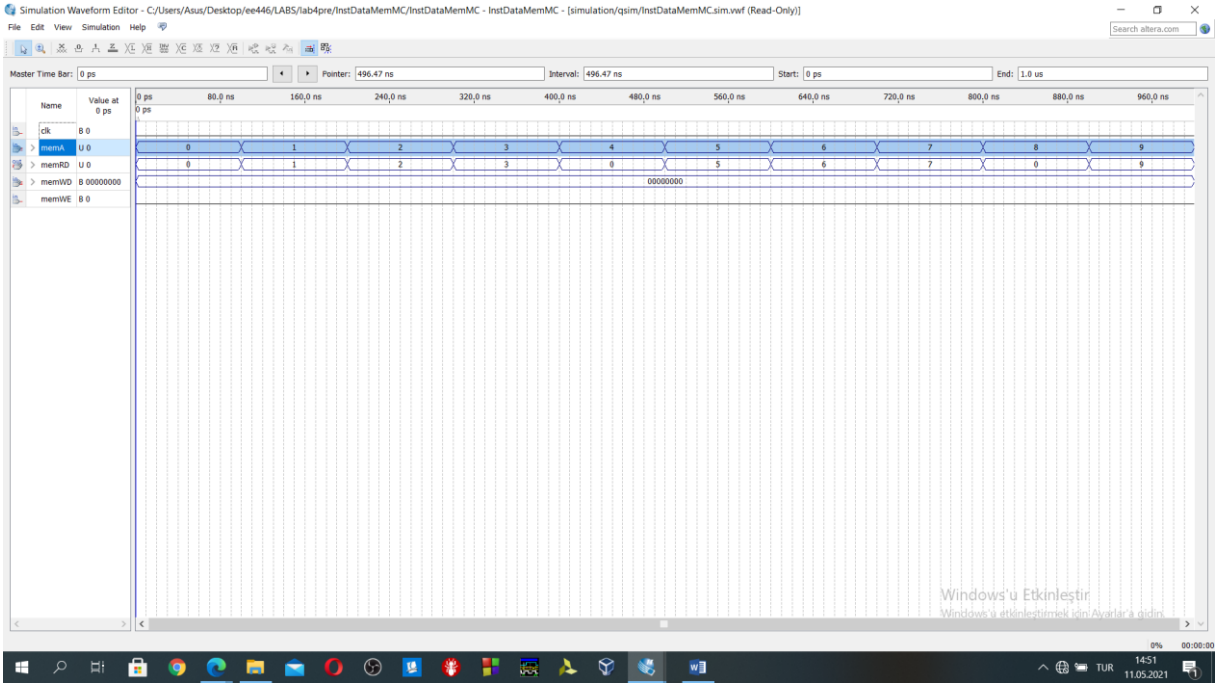


Figure 3:memory read simulation

In Figure 3, memA is memory address, memRD contains the memory read output which is pointed by memA. While initializing the memory, 0,4,8… contains the instructions. 1,2,3,5,6,7,9,10,11… contains

the data which is equal to the address no. Initially memory does not contain any instruction. Therefore, all of them are equal to 0.

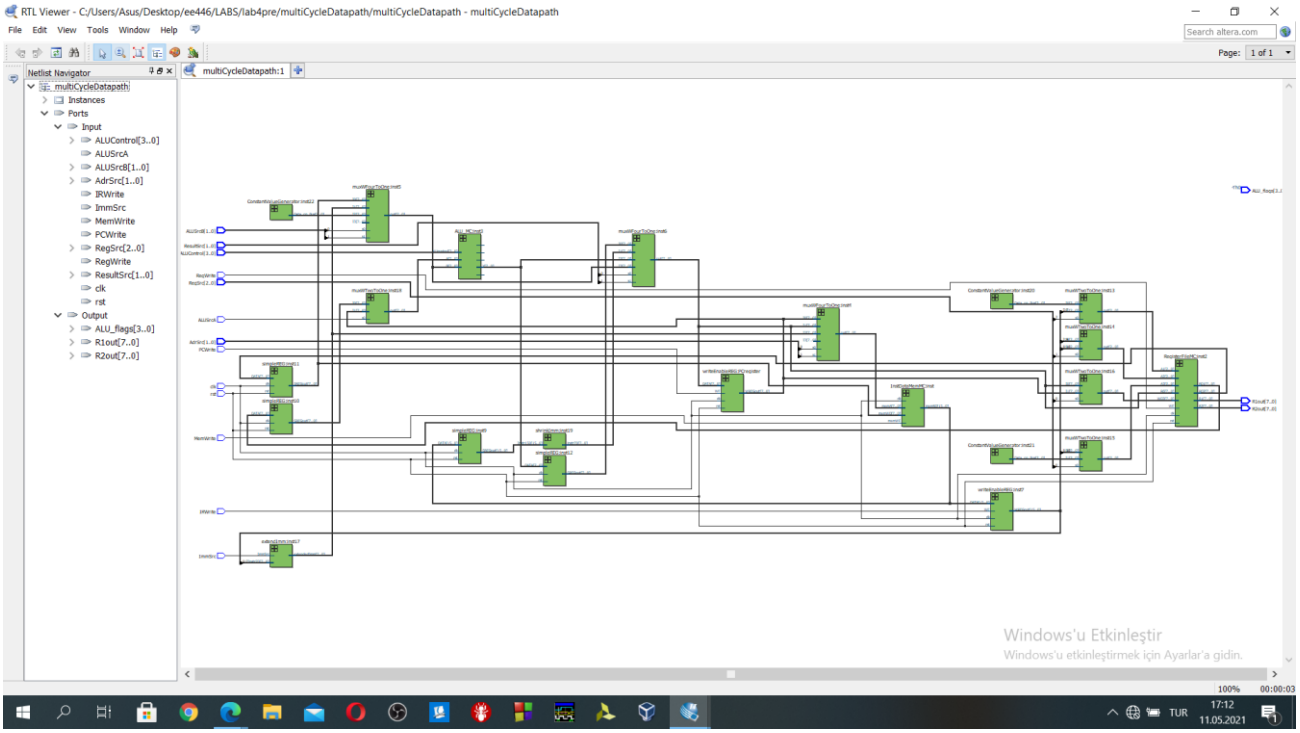


Figure 4:netlist view of the multiCycleDatapath

Figure 4 contains the netlist of the design.

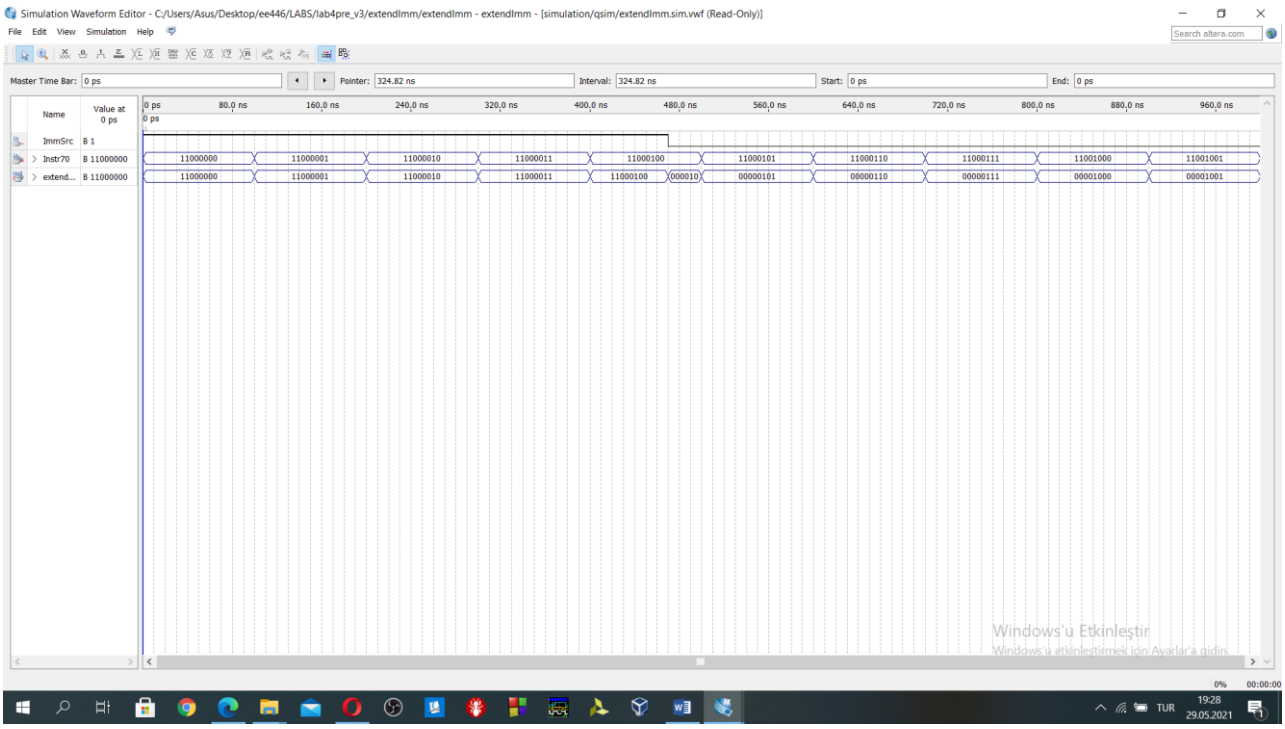


Figure 5: ExtendImm simulation result

When the ImmSrc=1, then input is directly served as output. When the ImmSrc=0, the last five bits of the input are extended as 8 bits by adding additional 0 to the first five bits of the input.

3. Simulation result of the Experimental work via Testbench

add-sub-and-orr-xor-clr-rol-ror-lsl-asr-lsr-ldr-ldi-str-b-bl-bi instructions in memory

```
                //instructionMemory initialization
                //we have 16 bits in the memORY it is used instruction
                //instructions
                /*
                    register file initial contents
                    register_R[3] = 8'b00001111; //15
                    register_R[4] = 8'b00011111; //31
                    register_R[5] = 8'b00111111; //63

                */
                   1 DATAmem[0]  =  16'b0000_0001_0111_0000;//add rd 1 rn 3 rm 4
then result is " 46" initially r3=15 r4=31 r5=63
                   2 DATAmem[4]  =  16'b0001_0010_1010_1100;//sub  r2=r5-r3 "48"
                   3 DATAmem[8]  =  16'b0010_0001_0111_0000;//and  r1=r4&r3 "15"
                   4 DATAmem[12] =  16'b0010_1010_0111_0100;//orr  r2=r3|r5  "63"
                   5 DATAmem[16] =  16'b0011_0001_0110_0000;//xor  r1=r3^r0  "15"
because r0 initially zero
                   6 DATAmem[20] =  16'b0011_1010_0000_0000;//clr r2 loaded with 0

                    //shift operations shift rn and store it in rd
                   7 DATAmem[24] =  16'b0100_0001_0110_0000;//rol r1=rol r3
                   8 DATAmem[28] =  16'b0100_1010_1000_0000;//ror r2= ror r4
                   9 DATAmem[32] =  16'b0101_0001_1010_0000;//lsl r1= r5*2 126
                   10 DATAmem[36] =  16'b0101_1010_1000_0000;//asr r2=asr r4
                   11 DATAmem[40] =  16'b0110_0001_1010_0000;//lsr r1= r5/2 31

                    //memory instructions rd=r2
                   12 DATAmem[44] =  16'b1000_0010_0110_0101; //ldr r2,[r3,5]
                   13 DATAmem[48] =  16'b1001_0010_1111_1111; //ldi r2 255
                   14 DATAmem[52] =16'b1010_0010_0110_0101; // str r2,[r3,5]
                    //rd=1
                   15 DATAmem[56] =16'b1000_0001_0110_0101; // ldr r1,[r3,5] check
stored content

                    //branch instructions
                   16 DATAmem[60] = 16'b1100_0000_0000_1000; //b pc+8+imm8(8)=76
                    //B TO #76 branch here "B 76"
                   17 DATAmem[76]=16'b1001_0010_0100_1100;// ldi r2 76(branch
check)
                   18 DATAmem[80]=16'b1100_1000_0001_0000; //bl branch with link
to the 104
                    //BL TO THE 104 "BL 104"
                   19 DATAmem[104] = 16'b1001_0010_0110_1000;//ldi r2 104 (bl
check)
                   20 DATAmem[108] = 16'b1100_0000_0101_1000; //b
pc+8+imm8(64)=204
                    //with branch at the 108 jump here
                   21 DATAmem[204] = 16'b1001_0010_1101_1000;//ldi r2 216(b check)
                    //verify branch indirect
                    //BI instruction
                   22 DATAmem[208]= 16'b1101_0000_0000_1000; //bi r2

                   23 DATAmem[216]= 16'b1001_0010_1111_1111;// ldi r2 104 (bi
check)jump to here after bi instruction
                //end of the instruction
```

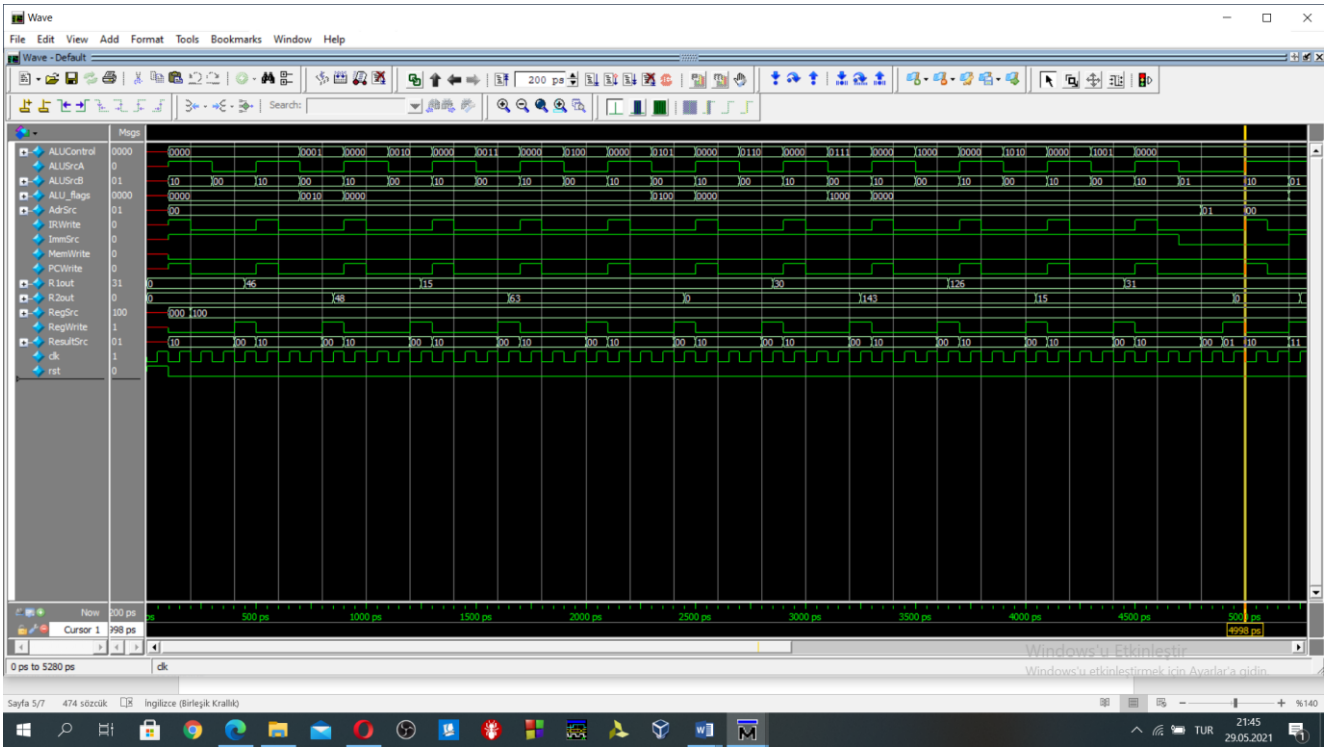


Figure 6: The simulation result between 0-5000ps

As in the datamem, the first executed instruction address is 0 because PC counter value is initially. It's value increases four by four as long as PC counter does not run into branch instruction.

R1out & R2out value are used as demonstration purposes. They provide r1 and r2 values in the register file.

1st instruction ADD(four cycles)(100ps-500ps) As in the Figure 6, R1out output value is updated as 46 after four clock cycle. r1=r3+r4 ➔ r1=15+31=46, calculated value and simulation value are matching.

2nd instruction SUB(four cycles)(500ps-900ps) As in the Figure 6, R2out output value is updated as 48 after four clock cycle. r2=r5-r3 ➔ r2=63-15=48, calculated value and simulation value are matching.

3rd instruction AND(four cycles)(900ps-1300ps) As in the Figure 6, R1out output value is updated as (15)1111 after four clock cycle. r1=r3&r4 ➔ r1=00001111(15)&00011111(31)=00001111(15), calculated value and simulation value are matching.

4th instruction ORR(four cycles)(1300ps-1700ps) As in the Figure 6, R2out output value is updated as (63)111111 after four clock cycle. r2=r3&r5 ➔ r2=00001111(15)&00111111(63)=00111111(63), calculated value and simulation value are matching.

5th instruction XOR(four cycles)(1700ps-2100ps) As in the Figure 6, R1out output value is updated as (15)1111 after four clock cycle. r1=r3^r0 ➔ r1=00001111(15)^00000000(0)=00001111(15), calculated value and simulation value are matching.

6th instruction CLR(four cycles)(2100ps-2500ps) As in the Figure 6, R2out output value is updated as 0 after four clock cycle. r2←0, ➔ r2=00000000(0) calculated value and simulation value are matching.

7th instruction ROL(four cycles)(2500ps-2900ps) As in the Figure 6, R1out output value is updated as (30)00011110 after four clock cycle. r1=r3(00001111) rotate left by 1 ➔ r1=00011110(30), calculated value and simulation value are matching.

8th instruction ROR(four cycles)(2900ps-3300ps) As in the Figure 6, R2out output value is updated as (143)10001111 after four clock cycle. r2=r3(00001111) rotate right by 1 ➔ r2=10001111(143), calculated value and simulation value are matching.

9th instruction LSL(four cycles)(3300ps-3700ps) As in the Figure 6, R1out output value is updated as (126)01111110 after four clock cycle. r1=r3(00001111) << 1 ➔ r1=01111110(126), calculated value and simulation value are matching.

10th instruction ASR(four cycles)(3700ps-4100ps) As in the Figure 6, R2out output value is updated as (15)00001111 after four clock cycle. r2=r4(00011111) arithmetic shift right by 1 ➔ r2=00001111(15), calculated value and simulation value are matching.

11th instruction LSR(four cycles)(4100ps-4500ps) As in the Figure 6, R1out output value is updated as (31)00011111 after four clock cycle. r1=r5(00111111) >> 1 ➔ r1=00011111(31), calculated value and simulation value are matching.

12th instruction LDR(five cycles)(4500ps-5000ps) As in the Figure 6, R2out output value is updated as 0 after five clock cycle. ldr r2,[r3,5] ➔ r2 is loaded with data at the memory address 20. The data is equal to 0 at pointed by the address. Calculated value and simulation value are matching. Why it is 0, because imm8 is 0000 0000.

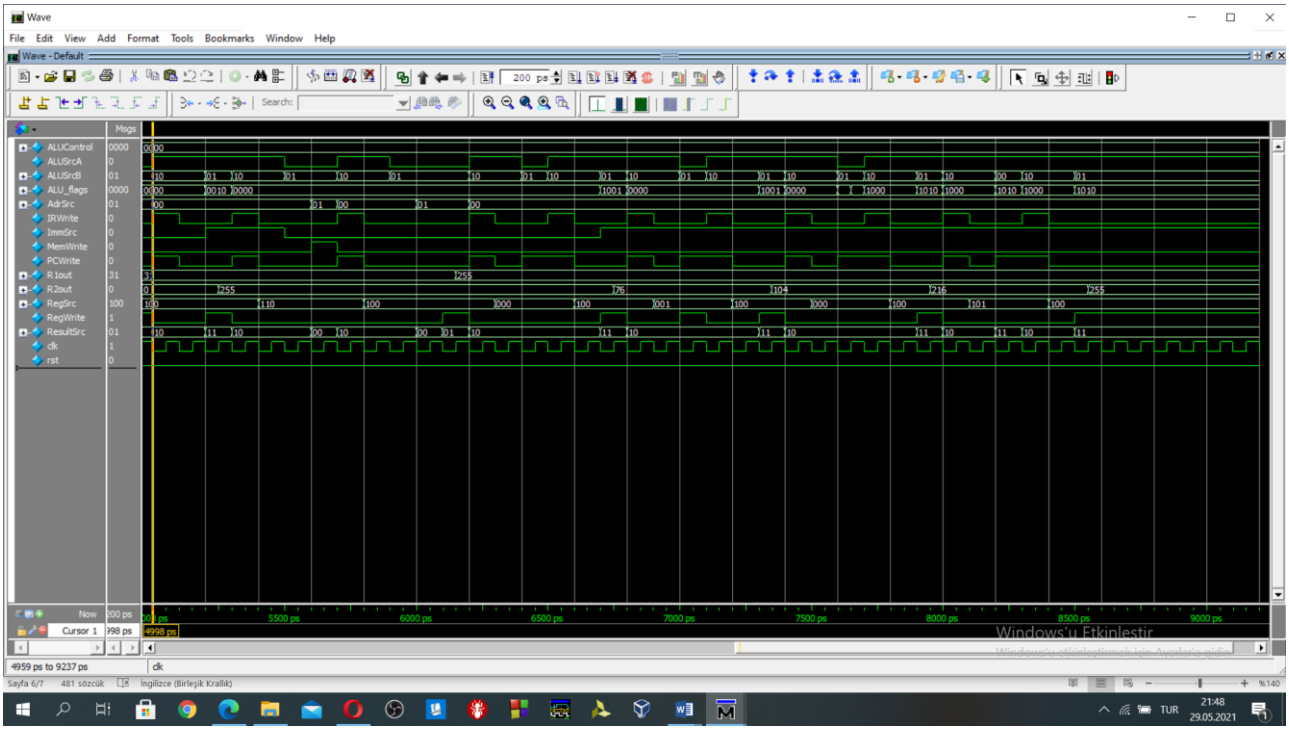


Figure 7: The simulation result between 5000ps-9000ps

13th instruction LDI(three cycles)(5000ps-5300ps) As in the Figure 7, R2out output value is updated as 255(1111 1111) after three clock cycle. imm8 value is moved to the r2 register. The moved value can be observed from Figure 7. Calculated value and simulation value are matching.

14th instruction STR(four cycles)(5300ps-5700ps) As in the Figure 7, str r2,[r3,5] ➔ r2 is stored to the "[r3]+5" address as 0000 0000 1111 1111.

//checking STR with LDR

15th instruction LDR (five cycles) (5700ps-6200ps) As in the Figure 7, R1out output value is updated as 255 after five clock cycle. ldr r1,[r3,5] ➔ r1 is loaded with data at the memory address 20. The data is equal to 255 at pointed by the address. Calculated value and simulation value are matching.

16th instruction B(branch)(three cycles)(6200ps-6500ps). When we look at the instruction memory, imm8 bits of the branch are 0000 1000(8), the next executed instruction must be PC+8+imm8. Current PC counter value is equal to 60. The next instruction must be at the address value 76.

Note: In my design, I do not multiply immediate with 4.

//check Branch working or not,

//imm8 value is equal to branched PC counter value. Then it's value is moved to the r2 as 76.

17th instruction LDI(three cycles)(6500ps-6800ps) As in the Figure 7, R2out output value is updated as 76(0100 1100) after three clock cycle. imm8 value is moved to the r2 register. The moved value can be observed from Figure 7. Calculated value and simulation value are matching.

18th instruction BL(branch with link)(three cycles)(6800ps-7100ps). When we look at the instruction memory, imm8 bits of the branch are 0001 0000(16), the next executed instruction must be PC+8+imm8. Current PC counter value is equal to 80. The next instruction must be at the address value 104.

19th instruction LDI(three cycles)(7100ps-7400ps) As in the Figure 7, R2out output value is updated as 104(0110 1000) after three clock cycle. imm8 value is moved to the r2 register. The moved value can be observed from Figure 7. Calculated value and simulation value are matching.

20th instruction B(branch)(three cycles)(7400ps-7700ps). When we look at the instruction memory, imm8 bits of the branch are 0101 1000(88), the next executed instruction must be PC+8+imm8. Current PC counter value is equal to 108. The next instruction must be at the address value 204.

21th instruction LDI(three cycles)(7700ps-8000ps) As in the Figure 7, R2out output value is updated as 216(1101 1000) after three clock cycle. imm8 value is moved to the r2 register. The moved value can be observed from Figure 7 as 216. Calculated value and simulation value are matching.

//R2=216, now I demonstrate branch indirect

22th instruction BI(branch indirect)(three cycles)(8000ps-8300ps). When we look at the instruction memory, rm=r2=216. The next executed instruction must be [r2] address. Current PC counter value is equal to 208. The next instruction must be at the address value 216 because r2's value is equal to 216.

23th instruction LDI(three cycles)(8300ps-8600ps) As in the Figure 7, R2out output value is updated as 255(1111 1111) after three clock cycle. imm8 value is moved to the r2 register. The moved value can be observed from Figure 7 as 255. Calculated value and simulation value are matching.

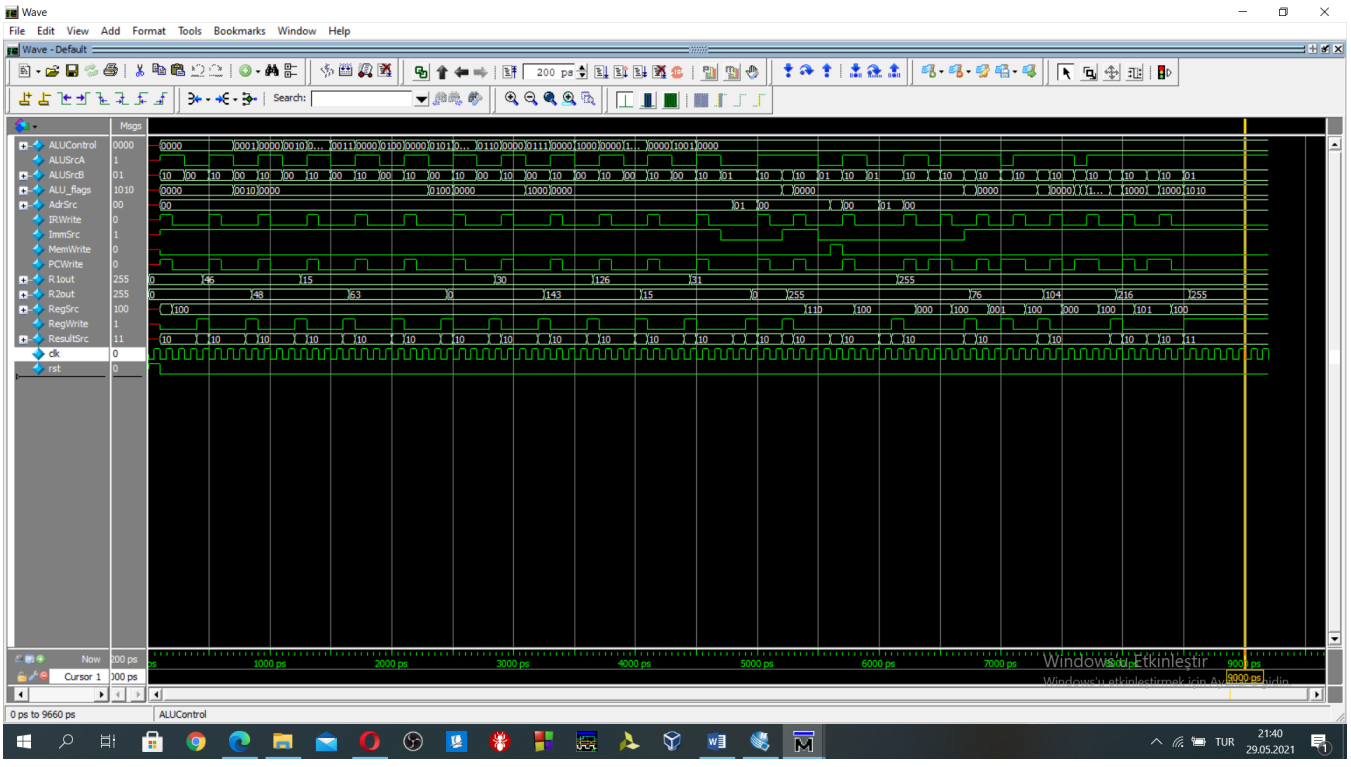


Figure 8:overall simulation result which includes memory, data-processing, shift and some branch (b, bl , bi) instructions

Additional demonstration parts for the BEQ,BNE,BNC,BC

```
//B TO #76 branch here "B 76"
                    DATAmem[76] =  16'b1001_0010_0100_1100;//ldi r2 76
                    DATAmem[80] =  16'b1100_1000_0001_0000; //bl branch with link
to the 104
                    //BL TO THE 104 "BL 104"
                    DATAmem[104] =  16'b1001_0010_0110_1000;//ldi r2 104
                    DATAmem[108] =  16'b0001_0001_1011_0100;//sub  r1=r5-r5 0
                    DATAmem[112] =  16'b1101_1000_0000_1000; //beq then branch 128
                    //equal then branch here "BEQ 128" 128
                    DATAmem[128] =  16'b1001_0010_1000_0000;//ldi r2 128
                    DATAmem[132] =  16'b0001_0001_1010_1100;//sub  r1=r5-r3 not
equal
                    DATAmem[136] =  16'b1110_0000_0000_1000; //bne then branch 152

                    //not equal then "BNE 152" to 152
                    DATAmem[152]    =  16'b1001_0010_1001_1000;//ldi r2 152
                    DATAmem[156]    =  16'b0001_0001_1010_1100;//sub  r1=r5-r3 not
equal
                    DATAmem[160]    =  16'b1111_0000_0000_1000; //bnc


                    //branch "BNC 176" 176 if the carry flag is 0
                    DATAmem[176]    =  16'b1001_0010_1011_0000;//ldi r2 176
                    DATAmem[180]    =  16'b0000_0001_0100_1000;//add rd 1 rn 2 rm
2 initially r2=176
                    DATAmem[184]    =  16'b1110_1000_0000_1000;//bc if carry flag
is set

                    //bc control "BC 200"
                    DATAmem[200]    =  16'b1001_0010_1100_1000;//ldi r2 200

                    DATAmem[204]    =  16'b1001_0010_1101_1000;//ldi r2 216
```

After the discussion in the ODTUclass, some part of the instruction memory excluded. I add also BNC, BC, BEQ, BNE. I have controlled just the flags in testbench. I follow the similar steps while verifying previous testbench. Firstly, I use branch instruction, then I have demonstrated with LDI instructions.

Instruction at 108 activates the 0 flag. The beq branch works. The PC jumps to the address 128 in this address r2 is loaded with 128 as seen in Figure 9. R2 value is equal to 128 at time 9500 ps. That means BEQ works perfectly.

Instruction at 132 does not activate 0 flag. BNE works. The PC jumps to the address 152 in this address r2 is loaded with 152 as seen in Figure 9. R2 value is equal to 152 at time 10500 ps. That means BNE works perfectly.

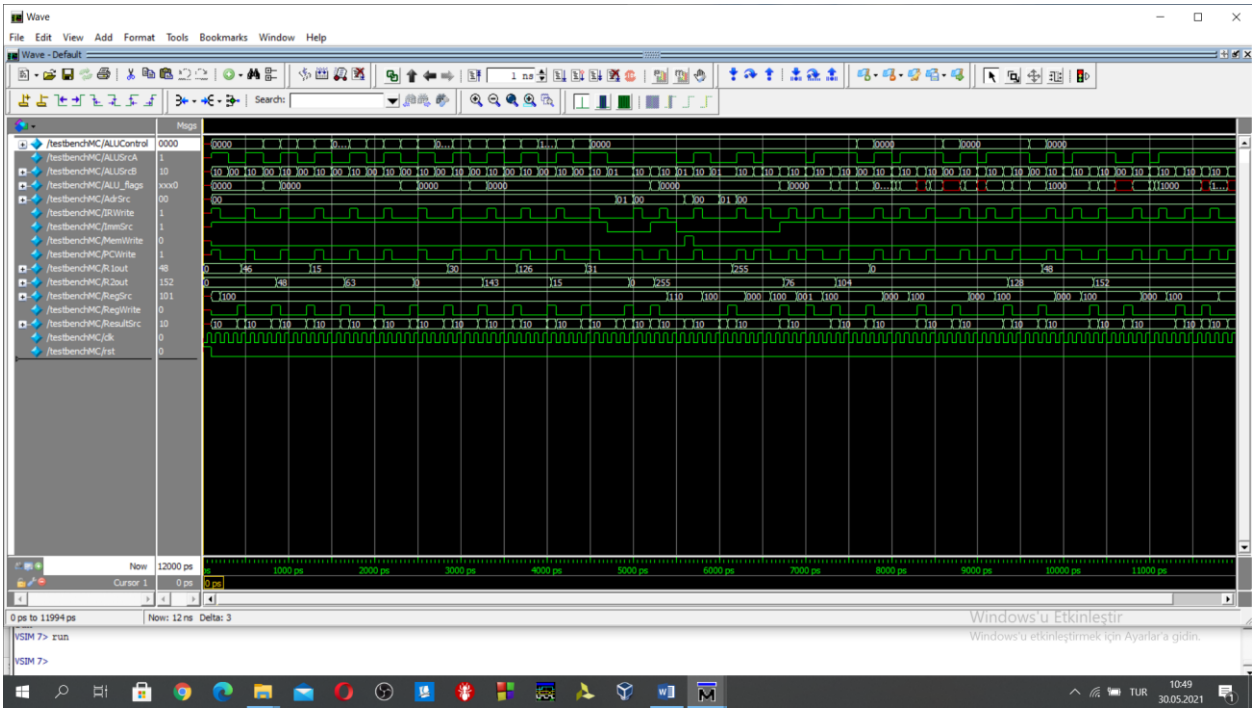


Figure 9: BC, BNC, BEQ, BNE instructions added to the simulation

4. Datapath design Code and Testbench code

```
///////////////multiCycleDatapath_code///////////////
module multiCycleDatapath_code(
    clk,
    rst,
    PCWrite,
    MemWrite,
    IRWrite,
    ImmSrc,
    RegWrite,
    ALUSrcA,
    AdrSrc,
    ALUControl,
    ALUSrcB,
    RegSrc,
    ResultSrc,
    ALU_flags,
    R1out,
    R2out
);

input wire  clk;
input wire  rst;
input wire  PCWrite;
input wire  MemWrite;
input wire  IRWrite;
input wire  ImmSrc;
input wire  RegWrite;
input wire  ALUSrcA;
input wire  [1:0] AdrSrc;
input wire  [3:0] ALUControl;
input wire  [1:0] ALUSrcB;
input wire  [2:0] RegSrc;
input wire  [1:0] ResultSrc;
output wire [3:0] ALU_flags;
output wire [7:0] R1out;
output wire [7:0] R2out;

wire    [15:0] WREGout;
wire    [3:0] X;
wire    [7:0] SYNTHESIZED_WIRE_0;
wire    [7:0] SYNTHESIZED_WIRE_31;
wire    [7:0] SYNTHESIZED_WIRE_2;
wire    [7:0] SYNTHESIZED_WIRE_3;
wire    [7:0] SYNTHESIZED_WIRE_32;
wire    [2:0] SYNTHESIZED_WIRE_5;
wire    [2:0] SYNTHESIZED_WIRE_6;
wire    [7:0] SYNTHESIZED_WIRE_33;
wire    [7:0] SYNTHESIZED_WIRE_34;
wire    [7:0] SYNTHESIZED_WIRE_9;
wire    [15:0] SYNTHESIZED_WIRE_11;
wire    [2:0] SYNTHESIZED_WIRE_12;
wire    [2:0] SYNTHESIZED_WIRE_13;
wire    [2:0] SYNTHESIZED_WIRE_14;
wire    [7:0] SYNTHESIZED_WIRE_16;
wire    [7:0] SYNTHESIZED_WIRE_17;
wire    [7:0] SYNTHESIZED_WIRE_35;
wire    [7:0] SYNTHESIZED_WIRE_22;
wire    [7:0] SYNTHESIZED_WIRE_23;
wire    [7:0] SYNTHESIZED_WIRE_24;
```

```
wire    [7:0] SYNTHESIZED_WIRE_25;
wire    [15:0] SYNTHESIZED_WIRE_36;




InstDataMemMC   b2v_inst(
    .clk(clk),
    .memWE(MemWrite),
    .memA(SYNTHESIZED_WIRE_0),
    .memWD(SYNTHESIZED_WIRE_31),
    .memRD(SYNTHESIZED_WIRE_36));


simpleREG   b2v_inst10(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_2),
    .SREGout(SYNTHESIZED_WIRE_9));
    defparam    b2v_inst10.W = 8;


simpleREG   b2v_inst11(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_3),
    .SREGout(SYNTHESIZED_WIRE_31));
    defparam    b2v_inst11.W = 8;


simpleREG   b2v_inst12(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_32),
    .SREGout(SYNTHESIZED_WIRE_24));
    defparam    b2v_inst12.W = 8;


muxWTwoToOne    b2v_inst13(
    .s0(RegSrc[2]),
    .I0(SYNTHESIZED_WIRE_5),
    .I1(WREGout[7:5]),
    .out(SYNTHESIZED_WIRE_12));
    defparam    b2v_inst13.W = 3;


muxWTwoToOne    b2v_inst14(
    .s0(RegSrc[1]),
    .I0(WREGout[4:2]),
    .I1(WREGout[10:8]),
    .out(SYNTHESIZED_WIRE_13));
    defparam    b2v_inst14.W = 3;


muxWTwoToOne    b2v_inst15(
    .s0(RegSrc[0]),
    .I0(WREGout[10:8]),
    .I1(SYNTHESIZED_WIRE_6),
```

```
    .out(SYNTHESIZED_WIRE_14));
    defparam    b2v_inst15.W = 3;


muxWTwoToOne    b2v_inst16(
    .s0(RegSrc[0]),
    .I0(SYNTHESIZED_WIRE_33),
    .I1(SYNTHESIZED_WIRE_34),
    .out(SYNTHESIZED_WIRE_16));
    defparam    b2v_inst16.W = 8;


extendImm   b2v_inst17(
    .ImmSrc(ImmSrc),
    .Instr70(WREGout[7:0]),
    .extendedImm(SYNTHESIZED_WIRE_22));


muxWTwoToOne    b2v_inst18(
    .s0(ALUSrcA),
    .I0(SYNTHESIZED_WIRE_9),
    .I1(SYNTHESIZED_WIRE_34),
    .out(SYNTHESIZED_WIRE_17));
    defparam    b2v_inst18.W = 8;


shrinkImm   b2v_inst19(
    .Instr150(SYNTHESIZED_WIRE_11),
    .instr70(SYNTHESIZED_WIRE_25));


RegisterFileMC  b2v_inst2(
    .clk(clk),
    .rst(rst),
    .WE(RegWrite),
    .A1(SYNTHESIZED_WIRE_12),
    .A2(SYNTHESIZED_WIRE_13),
    .A3(SYNTHESIZED_WIRE_14),
    .R6(SYNTHESIZED_WIRE_33),
    .WD3(SYNTHESIZED_WIRE_16),
    .R1(R1out),
    .R2(R2out),
    .RD1(SYNTHESIZED_WIRE_2),
    .RD2(SYNTHESIZED_WIRE_3));


ConstantValueGenerator  b2v_inst20(
    .Data_on_Bus(SYNTHESIZED_WIRE_5));
    defparam    b2v_inst20.BUS_DATA = 3'b110;
    defparam    b2v_inst20.DATA_WIDTH = 3;


ConstantValueGenerator  b2v_inst21(
    .Data_on_Bus(SYNTHESIZED_WIRE_6));
    defparam    b2v_inst21.BUS_DATA = 3'b111;
    defparam    b2v_inst21.DATA_WIDTH = 3;


ConstantValueGenerator  b2v_inst22(
```

```
    .Data_on_Bus(SYNTHESIZED_WIRE_23));
    defparam    b2v_inst22.BUS_DATA = 3'b100;
    defparam    b2v_inst22.DATA_WIDTH = 8;


ALU_MC  b2v_inst3(
    .A(SYNTHESIZED_WIRE_17),
    .ALUcontrol(ALUControl),
    .B(SYNTHESIZED_WIRE_35),
    .N(X[3]),
    .Z(X[2]),
    .CO(X[1]),
    .OVF(X[0]),
    .Y(SYNTHESIZED_WIRE_32));
    defparam    b2v_inst3.W = 8;


muxWFourToOne   b2v_inst4(
    .s0(AdrSrc[0]),
    .s1(AdrSrc[1]),
    .I0(SYNTHESIZED_WIRE_34),
    .I1(SYNTHESIZED_WIRE_33),


    .out(SYNTHESIZED_WIRE_0));
    defparam    b2v_inst4.W = 8;


muxWFourToOne   b2v_inst5(
    .s0(ALUSrcB[0]),
    .s1(ALUSrcB[1]),
    .I0(SYNTHESIZED_WIRE_31),
    .I1(SYNTHESIZED_WIRE_22),
    .I2(SYNTHESIZED_WIRE_23),

    .out(SYNTHESIZED_WIRE_35));
    defparam    b2v_inst5.W = 8;


muxWFourToOne   b2v_inst6(
    .s0(ResultSrc[0]),
    .s1(ResultSrc[1]),
    .I0(SYNTHESIZED_WIRE_24),
    .I1(SYNTHESIZED_WIRE_25),
    .I2(SYNTHESIZED_WIRE_32),
    .I3(SYNTHESIZED_WIRE_35),
    .out(SYNTHESIZED_WIRE_33));
    defparam    b2v_inst6.W = 8;


writeEnableREG  b2v_inst7(
    .clk(clk),
    .rst(rst),
    .WE(IRWrite),
    .DATA(SYNTHESIZED_WIRE_36),
    .WREGout(WREGout));
    defparam    b2v_inst7.W = 16;
```

```
simpleREG   b2v_inst9(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_36),
    .SREGout(SYNTHESIZED_WIRE_11));
    defparam    b2v_inst9.W = 16;


writeEnableREG4PC   b2v_PCregister(
    .clk(clk),
    .rst(rst),
    .WE(PCWrite),
    .DATA(SYNTHESIZED_WIRE_33),
    .WREGout(SYNTHESIZED_WIRE_34));

assign  ALU_flags = X;

endmodule
/*for the pc counter initialize with 0*/
module writeEnableREG4PC (DATA,clk,rst,WREGout,WE);
input rst,clk,WE;
input [7:0] DATA;
output reg [7:0] WREGout=8'b00000000;


always @(posedge clk)//due to sync reset issue
begin
    if(rst==1)
        begin
            WREGout<=0;
        end

    if(WE==1)
        begin
            WREGout<=DATA;
        end


end

endmodule




module ALU_MC #(parameter W=8) (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 [3:0] 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)
    4'b0000: //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
    4'b0001: //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
    4'b0010:
    begin
            Y=A&B;    //and
            CO=0;
            OVF=0;

    end

        4'b0011:
    begin

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


    end

        4'b0100:
    begin

            Y=A^B;    //xor
            CO=0;
            OVF=0;


    end

    4'b0101:
    begin

            Y=0;    //clear
            CO=0;
            OVF=0;


    end
```

```
    4'b0110:
    begin

            Y={A[6:0],A[7]};   //rol
            CO=0;
            OVF=0;


    end

        4'b0111:
    begin

            Y={A[0],A[7:1]};   //ror
            CO=0;
            OVF=0;


    end


    4'b1000:  //shift left lsl
    begin
           Y=A<<1;
            CO=0;
            OVF=0;
    end

    4'b1001:  //shift right lsr
    begin
            Y=A>>1;
            CO=0;
            OVF=0;
    end


    4'b1010:
    begin
            Y={A[7],A[7:1]};   //arithmetic shift right
            CO=0;
            OVF=0;
    end


    endcase
end


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

endmodule

module extendImm
(
//data,shift no needfor the extendedImm
```

```
//memory inst imm5 for the ldr and str
//memory inst imm8 for the immediate
//branch no need to extend
//input port
input     ImmSrc,
input     [7:0] Instr70,
//output port
output    [7:0]  extendedImm
);
//ImmSrc 1 no change
//ImmSrc 0 imm5
assign extendedImm=ImmSrc ? Instr70 :{3'b0,Instr70[4:0]};


endmodule

//DATA MEMORY

 module InstDataMemMC
 (      // input ports
      input                clk,
      input     [7:0]      memA,//memory address according to the address write or
read occur
      input     [7:0]      memWD,//memory write data, it specifies the memory data
which can be written
      input                memWE,//memory write enable
      // output port
      output    [15:0]      memRD
 );


      reg        [15:0]          DATAmem  [255:0];

        //also maximum PC value is 252
        //however PC values are 0-4-8...252
        //memWD values are extended to 16 bits
        //initialize the memory

        integer i;
      initial
          begin
                //instructionMemory initialization
                //we have 16 bits in the memORY it is used instruction
                //instructions
                /*
                    register file initial contents
                    register_R[3] = 8'b00001111; //15
                    register_R[4] = 8'b00011111; //31
                    register_R[5] = 8'b00111111; //63

                */

                    DATAmem[0]  =  16'b0000_0001_0111_0000;//add rd 1 rn 3 rm 4
then result is " 46" initially r3=15 r4=31 r5=63
                    DATAmem[4]  =  16'b0001_0010_1010_1100;//sub  r2=r5-r3 "48"
                    DATAmem[8]  =  16'b0010_0001_0111_0000;//and  r1=r4&r3 "15"
                    DATAmem[12] =  16'b0010_1010_0111_0100;//orr  r2=r3|r5  "63"
                    DATAmem[16] =  16'b0011_0001_0110_0000;//xor  r1=r3^r0  "15"
because r0 initially zero
```

```
                    DATAmem[20] =  16'b0011_1010_0000_0000;//clr r2 loaded with 0

                    //shift operations shift rn and store it in rd
                    DATAmem[24] =  16'b0100_0001_0110_0000;//rol r1=rol r3
                    DATAmem[28] =  16'b0100_1010_1000_0000;//ror r2= ror r4
                    DATAmem[32] =  16'b0101_0001_1010_0000;//lsl r1= r5*2 126
                    DATAmem[36] =  16'b0101_1010_1000_0000;//asr r2=asr r4
                    DATAmem[40] =  16'b0110_0001_1010_0000;//lsr r1= r5/2 31

                    //memory instructions rd=r2
                    DATAmem[44] =  16'b1000_0010_0110_0101; //ldr r2,[r3,5]
                    DATAmem[48] =  16'b1001_0010_1111_1111;//ldi r2 255
                    DATAmem[52] =  16'b1010_0010_0110_0101; // str r2,[r3,5]
                    //rd=1
                    DATAmem[56] =  16'b1000_0001_0110_0101; // ldr r1,[r3,5]
33+5=38

                    //branch instructions
                    DATAmem[60] =  16'b1100_0000_0000_1000; //b pc+8+imm8(8)=76

                    //B TO #76 branch here "B 76"
                    DATAmem[76] =  16'b1001_0010_0100_1100;//ldi r2 76
                    DATAmem[80] =  16'b1100_1000_0001_0000; //bl branch with link
to the 104

                    //BL TO THE 104 "BL 104"
                    DATAmem[104]    =  16'b1001_0010_0110_1000;//ldi r2 104
                    DATAmem[108]    =  16'b1100_0000_0101_1000; //b
pc+8+imm8(64)=204

                    //with branch at the 108 jump here
                    DATAmem[204]    =  16'b1001_0010_1101_1000;//ldi r2 216

                    //verfy branch indirect
                    //bi r2
                    DATAmem[208]    =  16'b1101_0000_0000_1000; //bi r2

                    DATAmem[216]    =  16'b1001_0010_1111_1111;//jump to here
after bi instruction
                //end of the instruction

                    DATAmem[220]    =  16'b0000_0000_0000_0000;
                    DATAmem[224]    =  16'b0000_0000_0000_0000;
                    DATAmem[228]    =  16'b0000_0000_0000_0000;
                    DATAmem[232]    =  16'b0000_0000_0000_0000;
                    DATAmem[236]    =  16'b0000_0000_0000_0000;
                    DATAmem[240]    =  16'b0000_0000_0000_0000;
                    DATAmem[244]    =  16'b0000_0000_0000_0000;
                    DATAmem[248]    =  16'b0000_0000_0000_0000;
                    DATAmem[252]    =  16'b0000_0000_0000_0000;



//data memory initialization
//data initialization
//instructions at the 0 4 8 12 ... 252 therefore others can be assigned randomly
           for(i=0;i<256;i=i+1)
              begin
                    if(i%3'd4!=0)
```

```
                        begin
                            DATAmem[i] = i;
                        end
                end

            end
              /*
            initial begin
            $readmemh("memory.txt",DATAmem,0,255);
            end
                        */


            //memory write operation
          always @(posedge clk) begin
               if (memWE)
                  begin
                   DATAmem[memA] = {8'b0,memWD};//extended value is stored the memory
                  end
                  else
                  begin
                  //nothing
                  end
          end

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

 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 muxWFourToOne #(parameter W=1)(s0,s1,I0,I1,I2,I3,out);

input s0;
input s1;
input [W-1:0] I0;
input [W-1:0] I1;
input [W-1:0] I2;
input [W-1:0] I3;

output reg [W-1:0] out;
wire [1:0] sel ={s1,s0};



always @(s0 or s1 or I0 or I1 or I2 or I3)
    begin
         case (sel)
         2'b00 : out = I0;
         2'b01 : out = I1;
         2'b10 : out = I2;
         2'b11 : out = I3;
         endcase
```

```
    end

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



module RegisterFileMC
 (   //input ports
      input                    clk,
      input                    rst,
      input                    WE,  //write enable signal
      input          [2:0]     A1 ,
        input          [2:0]     A2,
        input          [2:0]     A3,
      input          [7:0]    WD3,
      //output ports
      output         [7:0]    RD1,
      output         [7:0]    RD2,

        //demostration purposes
        output         [7:0]    R1,
      output         [7:0]    R2,

        input           [7:0]      R6
 );


        reg                [7:0]    register_R [7:0]; //8 bits width and 8 bits
length

      always @ (posedge clk or posedge rst) begin

          if(rst) begin
                 //general purpose registers
              register_R[0] = 8'b0;
                   register_R[1] = 8'b0;
                   register_R[2] = 8'b0;
                   register_R[3] = 8'b00001111; //15
                   register_R[4] = 8'b00011111; //31
                   register_R[5] = 8'b00111111; //63
```

```
                    //link register
                    register_R[7] = 8'b0;

                    //pc represent the r6

           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 = (A1==3'b110) ? R6:register_R[A1];  //read data 1
      assign RD2 = register_R[A2];  //read data 2
        assign R1 = register_R[1];
      assign R2 = register_R[2];
 endmodule

 module shrinkImm
(

input    [15:0] Instr150,
//output port
output    [7:0]  instr70
);

assign instr70=Instr150[7:0];


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

module simpleREG #(parameter W=1) (DATA,clk,rst,SREGout);
input rst,clk;
input [W-1:0] DATA;
output reg [W-1:0] SREGout;


always @(posedge clk)
begin
    if(rst==1)
        begin
            SREGout<=0;
```

```
        end
    else
        begin

            SREGout<=DATA;
        end

end

endmodule


module writeEnableREG #(parameter W=1) (DATA,clk,rst,WREGout,WE);
input rst,clk,WE;
input [W-1:0] DATA;
output reg [W-1:0] WREGout;


always @(posedge clk)//due to sync reset issue
begin
    if(rst==1)
        begin
            WREGout<=0;
        end
    else
        begin

            if(WE==1)
                begin
                    WREGout<=DATA;
                end

        end

end

endmodule
```

```
//////////////Textbench Multi Cycle Code///////////////
/*

input wire  clk;
input wire  rst;
input wire  PCWrite;
input wire  MemWrite;
input wire  IRWrite;
input wire  ImmSrc;
input wire  RegWrite;
input wire  ALUSrcA;
input wire  [1:0] AdrSrc;
input wire  [3:0] ALUControl;
input wire  [1:0] ALUSrcB;
input wire  [2:0] RegSrc;
input wire  [1:0] ResultSrc;

*/
module testbenchMC();
//inputs are reg
//outputs are wire
//assume that bus width is 3 to test the result
//inputs
reg clk;
reg rst;
reg PCWrite;
reg MemWrite;
reg IRWrite;
reg ImmSrc;
reg RegWrite;
reg ALUSrcA;
reg [1:0] AdrSrc;
reg   [3:0] ALUControl;
reg [1:0] ALUSrcB;
reg [2:0] RegSrc;
reg [1:0] ResultSrc;


//outputs
 wire   [3:0] ALU_flags;
 wire   [7:0] R1out;
 wire   [7:0] R2out;




// instantiate device under test
multiCycleDatapath_code DUT(
    clk,
    rst,
    PCWrite,
    MemWrite,
    IRWrite,
    ImmSrc,
    RegWrite,
    ALUSrcA,
    AdrSrc,
    ALUControl,
```

```
    ALUSrcB,
    RegSrc,
    ResultSrc,
    ALU_flags,
    R1out,
    R2out
);


initial
begin
rst=1;
#100;
rst=0;
PCWrite=0;
MemWrite=0;
IRWrite=0;
ImmSrc=1;
RegWrite=0;
ALUSrcA=1;
AdrSrc=2'b11;
ALUControl=4'b0000;
ALUSrcB=2'b11;
RegSrc=3'b000;
ResultSrc=2'b00;
end

// generate clock
always // no sensitivity list, so it always executes
    begin
    clk = 0; #50; clk = 1; #50;
    end

//change the input signals according to the instructions
initial // no sensitivity list, so it always executes
    begin
//ADD instruction
//fetch
#100;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;
```

```
//execute
#100;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//SUB instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0001;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//AND instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
```

```
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0010;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ORR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
```

```
#100;
ALUSrcA=0;
ALUControl=4'b0011;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//XOR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0100;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//CLR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
```

```
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0101;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ROL instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
```

```
ALUSrcA=0;
ALUControl=4'b0110;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ROR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0111;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//LSL instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
```

```
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b1000;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ASR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
```

```
ALUControl=4'b1010;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//LSR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b1001;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;
//shift and data processing operations are completed


//LDR instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
```

```
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//memADR Cycle 3
#100;
ImmSrc=0;
ALUSrcB=2'b01;
ALUSrcA=0;
ALUControl=4'b0000;

//memREAD Cycle 4
#100;
ResultSrc=2'b00;
AdrSrc=2'b01;
MemWrite=0;

//memWriteBack Cycle 5
#100;
ResultSrc=2'b01;
RegWrite=1;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;
```

```
//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;


//STR instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b110;
ResultSrc=2'b10;

//memADR Cycle 3
#100;
ImmSrc=0;
ALUSrcB=2'b01;
ALUSrcA=0;
ALUControl=4'b0000;

//memWrite
#100;
ResultSrc=2'b00;
AdrSrc=2'b01;
MemWrite=1;


//LDR instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;
```

```
//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//memADR Cycle 3
#100;
ImmSrc=0;
ALUSrcB=2'b01;
ALUSrcA=0;
ALUControl=4'b0000;

//memREAD Cycle 4
#100;
ResultSrc=2'b00;
AdrSrc=2'b01;
MemWrite=0;

//memWriteBack Cycle 5
#100;
ResultSrc=2'b01;
RegWrite=1;


//LDR and STR instructions are done
//after that branch instructions


//B instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b000;
```

```
ResultSrc=2'b10;


//branch cycle 3
#100;
PCWrite=1;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b01;
ResultSrc=2'b10;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;



//BL instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
```

```
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b001;
ResultSrc=2'b10;

//branch cycle 3
#100;
PCWrite=1;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b01;
ResultSrc=2'b10;
RegWrite=1;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;


//B instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
```

```
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b000;
ResultSrc=2'b10;


//branch cycle 3
#100;
PCWrite=1;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b01;
ResultSrc=2'b10;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;
```

```
//BI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b101;
ResultSrc=2'b10;

//bi pc write
#100;
ALUSrcB=2'b00;
ResultSrc=2'b11;
PCWrite=1;



//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;
```

```
//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;

end

endmodule
```