Laboratory Work 5 Report

Control Unit 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: From Mnemonics to Machine Code.

The second part contains the verification results of Multi-Cycle CPU controller design and lab4 comparison.

The third part consists of the three subroutines which is specified in “Validation of Operation via Microprogrammed Control” part of the preliminary work, their machine codes and the simulation results on the modelsim.

The last part contains Datapath design code, Controller Design code, unified code of Controller code and Datapath Code. Also, last part contains the controller design testbench and the unified design testbench.

1. Preliminary Work Part(Hard-copy)

Laboratory Works Preliminary work

1.2 ISA Configuration

We have a field for each distinct criterion in the instruction.

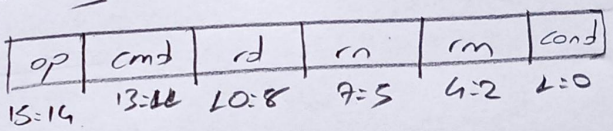
Binary codes are assigned to the each instruction in the entire ISA.

Instructions are 16 bits. According to the datapath, conditional information is just needed for conditional branch instructions.

(1) Let's have a look at the overwiev

there four types of instructions. These are Data-processing, shift, branch and memory instructions

Data processing



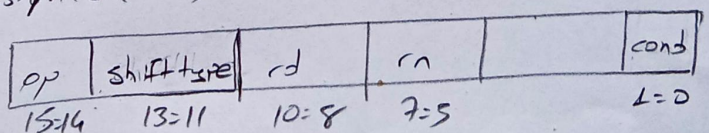
op=00

cmd
000 → add rd,rn,rm
001 → addi
010 → sub rd,rn,rm
011 → subi

cmd
100 → and rd,rn,rm
101 → orr rd,rn,rm
110 → xor rd,rn,rm
111 → clr rd

cond
00 → Equivalent (EQ)
01 → not equivalent (NE)
10 → lower unsigned (LO)
11 → higher or same unsigned (HS)

shift instructions



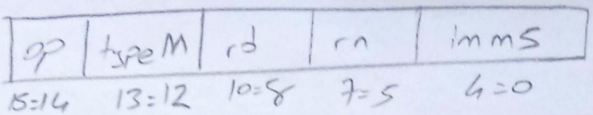
shift rn & store at rd

shift type (shift rn & store at rd)
000 → rol rd,rn
001 → ror rd,rn
010 → lsl rd,rn
011 → asr rd,rn
100 → lsr rd,rn

cond
00 → EQ
01 → NE
10 → LO
11 → HS

op=01

Memory inst.



OP = 10

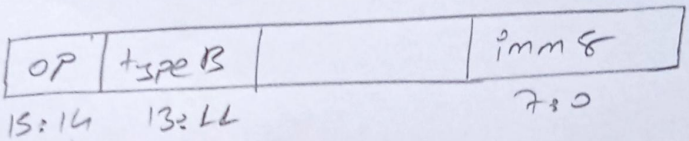
typeM

00 → ldr rd, [rn, imm5]

01 → ldi rd, #inst 7:0

10 → str rd, [rn, imm5]

Branch inst.



OP = 11

typeB

000 → B

001 → BL

011 → BEQ

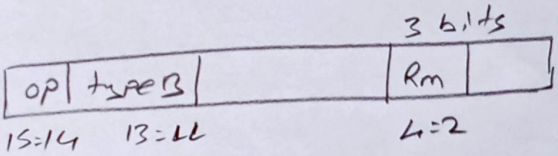
typeB

100 → BNE

101 → BC

110 → BNC

OP = 11 typeB = 010 =>



=> BI

(*) indirect branch, the target address is specified indirectly either through memory or general purpose registers

BI takes Rm as its operand and causes a branch to address saved in Rm.

(12) ISA Configuration

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

(2) I squeezed all the needed conditional information related to the instruction in a minimum number of bits.

Therefore, in my design instructions are 16 bits.

If I used 32 bits instructions, memory utilization would increase. To eliminate unnecessary memory utilization, I have used 16 bits instructions. However, control unit complexity increases. The trade-off between memory utilization and component consumption can be evaluated according to the system requirements. I try to decrease memory utilization. When I have used 32 bits instructions, I would obtain a simpler control unit with reduced number of functional components

(1.2.1) Multi-Cycle Controller Unit Design

Fetch => 1 clock cycle

Decode => 1 clock cycle

Execute => 1, 2, 3 clock cycle it depends on the instruction type.

FSM is going to be designed.

(A)
(B) END ISA => 1111_1111_1111_1111 => that forces the execution to be halted

RUN / RESET are necessary signals

RUN: incite the execution of the code from current instruction.

RESET: terminate the operation, sets the PC to very first slot in memory.

(1) Control signal for the fetch, decode, execute cycles

Fetch - PHASE (S0)

AdrSrc = 00

PCWrite = 1

ALUSrcA = 1

ALUSrcB = 10

ALUControl = 4'b0000

ResultSrc = 10

IRWrite = 1

RegWrite = 0

Decode - PHASE (S1)

RegSrc (specified according to OP)

PCWrite = 0

MemWrite = 0

IRWrite = 0

RegWrite = 0

ALUSrcA = 1

ALUSrcB = 2'b10

ALUControl = 0000 (add)

ResultSrc = 2'b10

EXECUTE - PHASE

DATA PROCESSING & SHIFT DATA

(S2) Execute_ALU

ALUSrcA = 0

ALUSrcB = 00

ALUControl (from 0 to 11)

RegSrc = 3'b100

ResultSrc = 2'b10

RegWrite = 0

(S3) ALU_WB

ResultSrc = 2'b00

RegWrite = 1

MEMORY INSTRUCTIONS

(S4) ALU_WB_LDI (cycle 3 LDI)

ImmSrc = 1;

ALUSrcB = 2'b01;

ResultSrc = 2'b11;

RegWrite = 1;

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

ImmSrc = 0

ALUSrcB = 2'b01

ALUSrcA = 0;

ALUControl = 4'b0000

(S6) memWRITE (cycle 4 for STR)

ResultSrc = 2'b00

AdrSrc = 2'b01

MemWrite = 1

(S7) MemRead (cycle 4 for LDR)

ResultSrc = 2'b00

AdrSrc = 2'b01

MemWrite = 0

(S8) mem_WB (cycle 5 for LDR)

ResultSrc = 2'b01

RegWrite = 1

BRANCH INSTRUCTIONS

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

PCWrite=(according to condition)

ALUSrcA=0

ALUControl=4'b0000

ALUSrcB=2'b01

ResultSrc=2'b10

(S10) BL_EX (cycle 3 for BL inst.)

PCWrite=1

ALUSrcA=0

ALUControl=4'b0000

ALUSrcB=2'b01

ResultSrc=2'b10

RegWrite=1

(S11) BI_PCWrite (cycle 3 for BI)

ALUSrcB=2'b00

ResultSrc=2'b11

PCWrite=1

(X) Fetch is completed in 1 clock cycle.

(X) Decode is completed in 1 clock cycle.

(X) According to the instruction execution clock cycle number changes from 1 to 3.

2. Multi-Cycle CPU controller design Modelsim Simulation Results

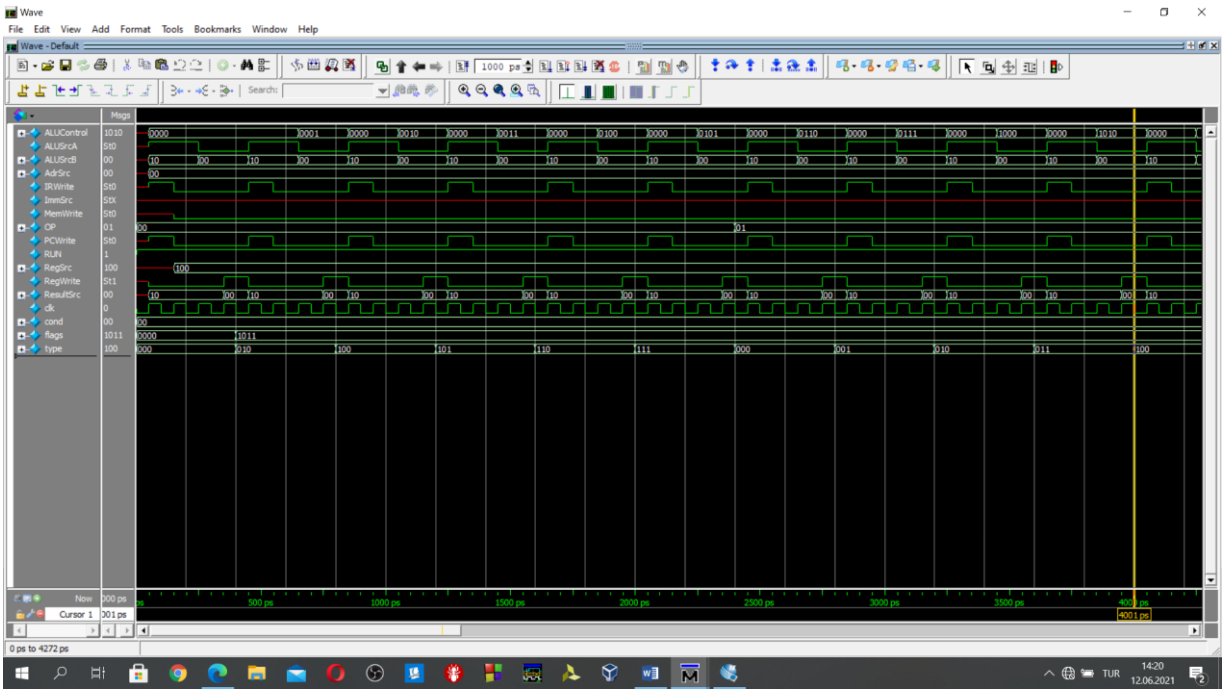


Figure 1: Multicycle controller unit simulation via modelsim 0-4000ns

```
//add: control signals are provided from the control unit in Figure 1 in 0-400ps
OP=2'b00; //op 00 means data processing
type=3'b000; //type împlies the add instruction in this case
#400;
//after the assigning the OP and type wait until the instruction ends.
//the clk period is 100 ns 400 ns implies 4 clock cycle,
//for the addition operaiton 4 clk cycles are needed. Following, by providing
//instruction OP and type to control unit, cpu can get proper control signals from
//the controller unit.

//sub:control signals are provided from the control unit in Figure 1 in 400-800 ps
flags=4'b1011;//flag is updated
OP=2'b00;
type=3'b010;
#400;

//and:control signals are provided from the control unit in Figure 1 in 800-1200ps
OP=2'b00;
type=3'b100;
#400;

//orr:control signals are provided from control unit in Figure 1 in 1200-1600ps
OP=2'b00;
type=3'b101;
#400;

//xor: control signals are provided from control unit in Figure 1 in 1600-2000ps
OP=2'b00;
type=3'b110;
#400;
```

```
//clr: control signals are provided from control unit in Figure 1 in 2000-2400ps
OP=2'b00;
type=3'b111;
#400;

//rol: control signals are provided from control unit in Figure 1 in 2400-2800ps
OP=2'b01; //op 01 implies shift operations
type=3'b000; //type implies shifht type rol-ror
#400;

//ror:control signals are provided from control unit in Figure 1 in 2800-3200ps
OP=2'b01;
type=3'b001;
#400;

//lsl:control signals are provided from control unit in Figure 1 in 3200-3600ps
OP=2'b01;
type=3'b010;
#400;

//asr:control signals are provided from control unit in Figure 1 in 3600-4000ps
OP=2'b01;
type=3'b011;
#400;
```

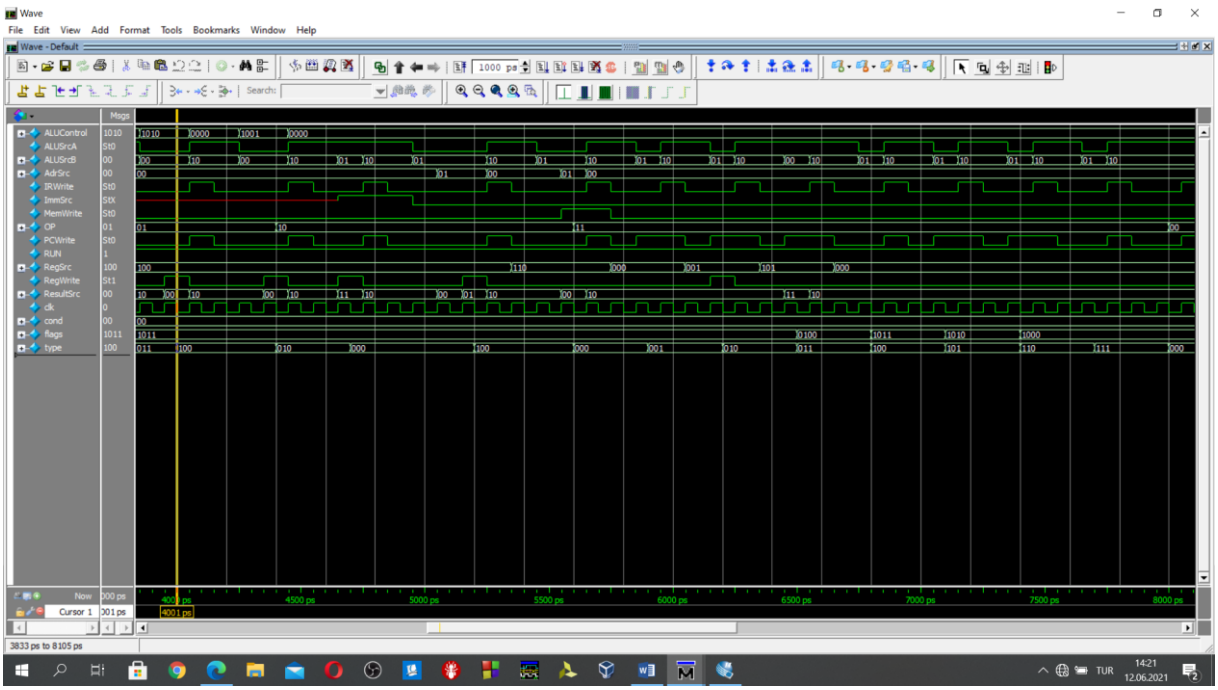


Figure 2: Multicycle controller unit simulation via modelsim 4000ns-8000ns

```
//lsr:control signals are provided from control unit in Figure 2 in 4000-4400ps
OP=2'b01;
type=3'b100;
#400;

//ldi:control signals are provided from control unit in Figure 2 in 4400-4700ps
OP=2'b10; //10 implies the memory instructions
type[2:1]=2'b01;// points the instruction type in mem. instructions
#300; // different from prev. İnst. 3 clocks(300ns) are enough for the execution
```

```
//ldr:control signals are provided from control unit in Figure 2 in 4700-5200ps
OP=2'b10;
type[2:1]=2'b00;
#500; // different from prev. İnst. 5 clocks(500ns) are enough for the execution

//str:control signals are provided from control unit in Figure 2 in 5200-5600ps
OP=2'b10;
type[2:1]=2'b10;
#400;

//branch uncond:control signals are provided from control unit in Figure 2 in
5600-5900ps
OP=2'b11;//OP 2’b11 implies the branch instructions
type=3'b000;//type shows the type of the branch
#300;

//branch with link:bl //control signals are provided from control unit in Figure 2
in 5900-6200ps
OP=2'b11;
type=3'b001;
#300;

//branch indirect: :control signals are provided from control unit in Figure 2 in
6200-6500ps
OP=2'b11;
type=3'b010;
#300;

//branch equivalent beq:control signals are provided from control unit in Figure 2
in 6500-6800ps
//nzcv
flags=4'b0100; //z=1 means equal case
OP=2'b11;
type=3'b011;
#300;

//branch not equivalent(bne):control signals are provided from control unit in
Figure 2 in 6800-7100ps
//nzcv
flags=4'b1011; //z=0 means not equal case
OP=2'b11;
type=3'b100;
#300;

//branch if carry set(bc):control signals are provided from control unit in Figure
2 in 7100-7400ps
flags=4'b1010; //c=1 means carry set case
OP=2'b11;
type=3'b101;
#300;

//branch if the carry not set(bnc):control signals are provided from control unit
in Figure 2 in 7400-7700ps
flags=4'b1000; //c=0 means not cary set case
OP=2'b11;
type=3'b110;
#300;
```

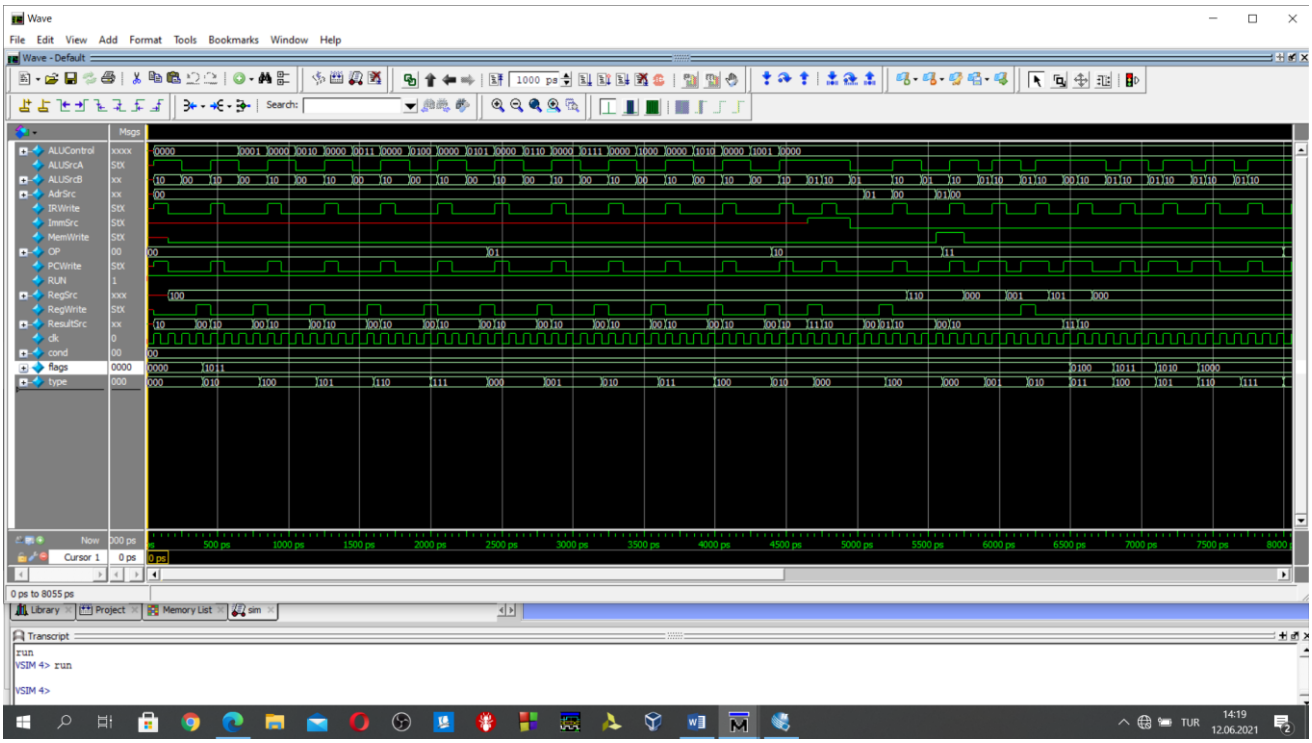


Figure 3:Multicycle controller unit simulation via modelsim 0-8000ns

Figure 3 shows the overall verification process of the multicycle controller design. There are unique control signals for each instruction according to the OP bits, type bits and flag bits. Also, these values are compatible with the control signals which is provided in lab4.

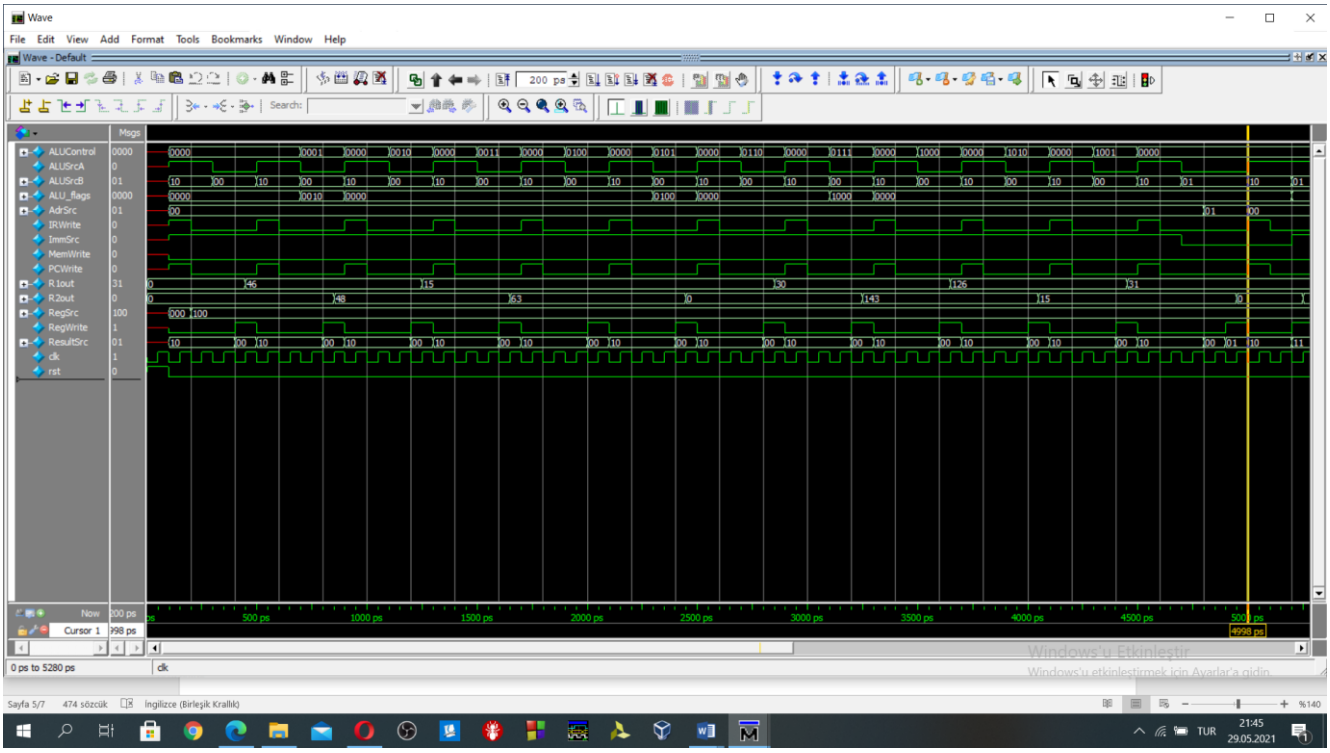


Figure 4:Multicycle datapath simulation result from lab4

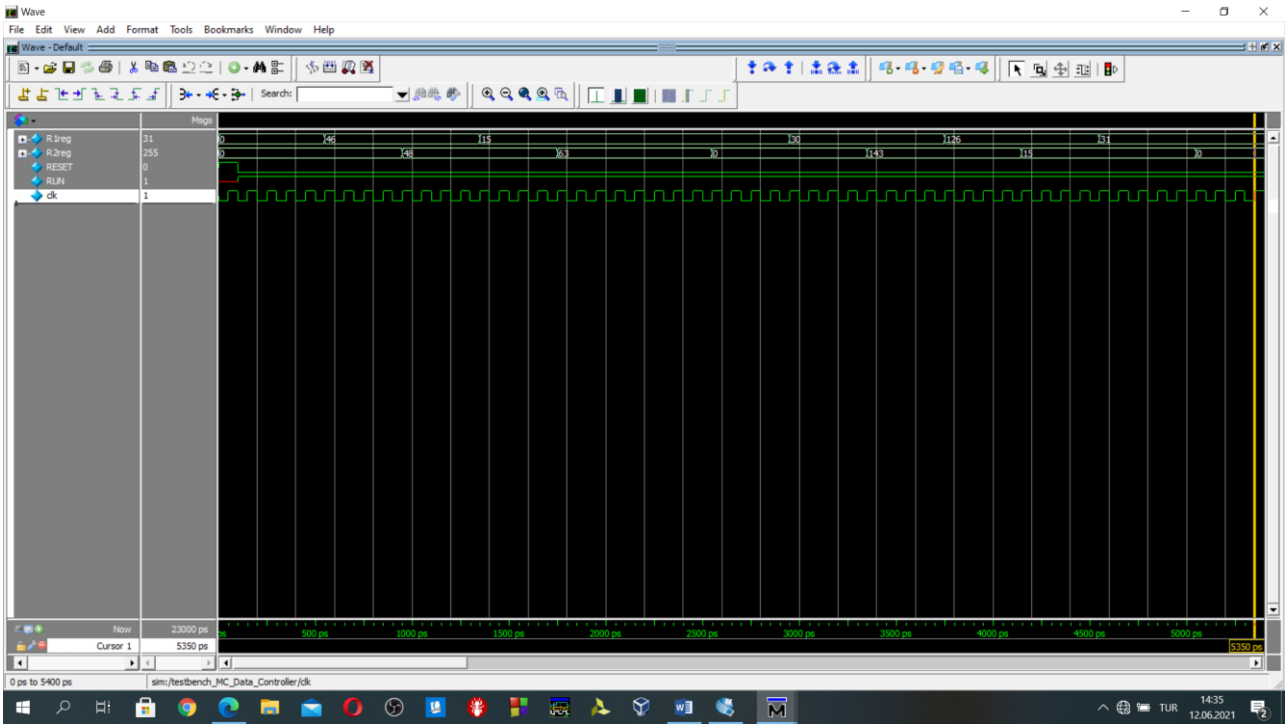


Figure 5:multicycle simulation which includes datapath and controller units

Figure 4 is adopted from lab4 preliminary work. Control signals are provided externally, there is no control unit for the simulation in Figure 4. On the other side, control signals are provided to datapath from the controller unit in Figure 5. In Figure 5, the unified version of datapath and controller design is simulated. The controller design is working because R1reg and R2reg values in both Figure 5 and Figure 4 are matching.

3. Validation of Operation via Microprogrammed Control & Modelsim Simulations

subroutines

```
//1. Write a subroutine that get an 8-bit number and computes it 2's complement.
/*/*//* 1. subroutine *//*/*/
//initially r0 is assigned 1111 1111
//r3 's content is 1
//then r1 contains the our number which is 8-bit number which will be computed it
2's complement
//lets say r1=00110011 we will compute its 2's complement

            xor r2,r1,r0  //take the complement of the number in r1
            add r2,r3,r2 //r2 contains the twos complement of the number which is
stored in r1
/*Machine Code of subroutine 1*/
//instructions for the subroutine 1
DATAmem[0]   =  16'b0011_0010_0010_0000;//xor  r2=r1^r0  d=2, n=1, m=0
DATAmem[4]   =  16'b0000_0010_0110_1000;//add rd 2 rn 3 rm 2
DATAmem[8]   =  16'b1111_1111_1111_1111; //end instruction
```

Simulation Result of the subroutine

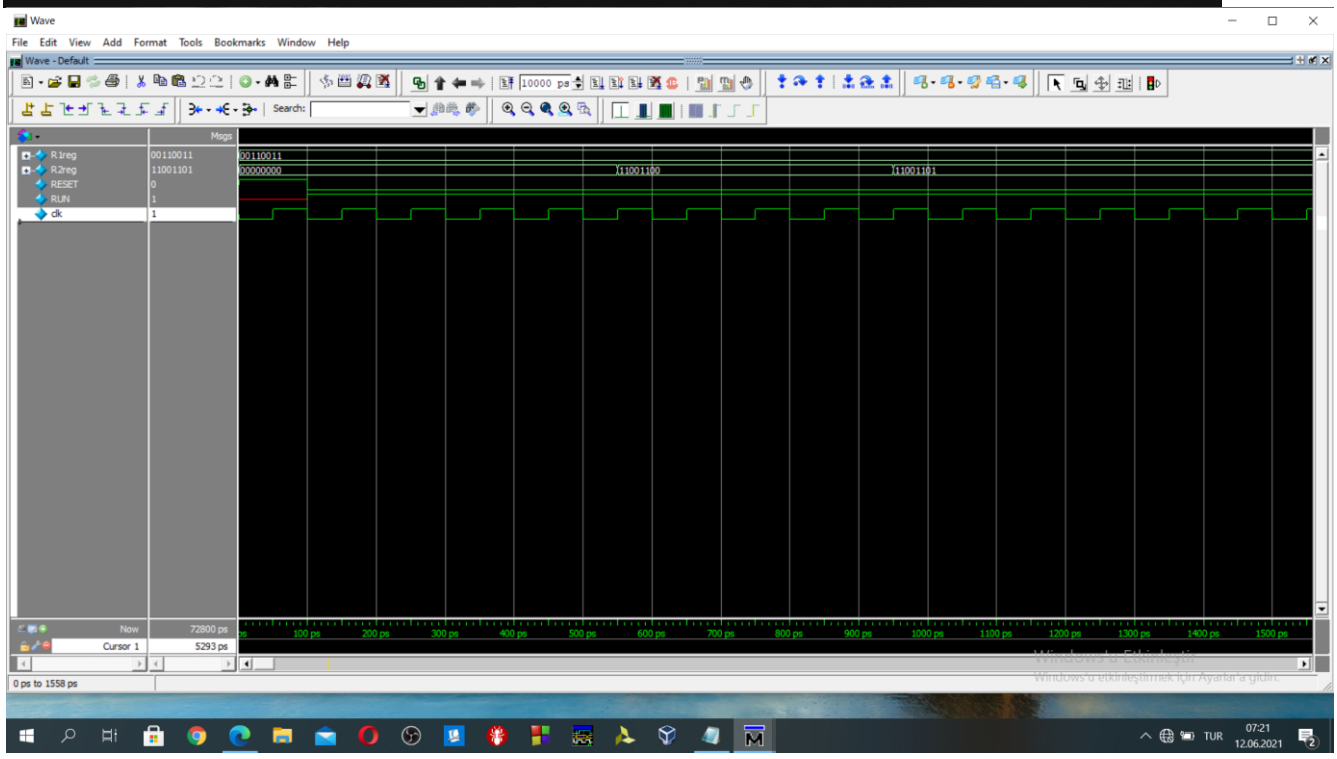


Figure 6:Modelsim Simulation of the subroutine that get an 8-bit number and computes it 2's complement(8'b00110011)

Initially 8 bits number is in R1(R1reg) as can be seen in Figure 6.

Then, R1 is exored with 8'b1111_1111. The result is assigned to R2 register as 11001100 as in Figure 6. Then, R2 is added with r3, which is 1. The 2's complement of the number can be seen in Figure 6 after 9500 ps in R2 register.

//2. Write a subroutine that computes the sum of an array of numbers and stores it in a memory location. The length of the array is constant and can be taken as 5.

```
/*/*//* 2. subroutine *//*/*/
//to compute summation
//initially r1's content is 0
//r3 keeps the array[0] memory location r3=112
//take number from memory address and add it with r1
//r4 is equal to 4 initially

            ldr r2,[r3] //ldr loads last 8 bits of addressed mem. Data to reg
            add r1,r1,r2 //add the stored number
             add r3,r3,r4 //increase the memory pointer location to reach next
array location
            ldr r2,[r3]
            add r1,r1,r2
             add r3,r3,r4
            ldr r2,[r3]
            add r1,r1,r2
             add r3,r3,r4
            ldr r2,[r3]
            add r1,r1,r2
            add r3,r3,r4
            ldr r2,[r3]
            add r1,r1,r2


/*Machine Code of subroutine 2*/
//instructions for the subroutine 2
DATAmem[0]  =  16'b1000_0010_0110_0000; //ldr r2,[r3]
DATAmem[4]  =  16'b0000_0001_0010_1000; //add rd 1 rn 1 rm 2  then result is " 3"
3=0+3
DATAmem[8]  =  16'b0000_0011_0111_0000; //add rd 3 rn 4 rm 3  then result r3+4
DATAmem[12] =  16'b1000_0010_0110_0100; //ldr r2,[r3]
DATAmem[16] =  16'b0000_0001_0010_1000; //add rd 1 rn 1 rm 2  then result is " 8"
8=5+3
DATAmem[20] =  16'b0000_0011_0111_0000; //add rd 3 rn 4 rm 3  then result r3+4
DATAmem[24] =  16'b1000_0010_0110_1000; //ldr r2,[r3]
DATAmem[28] =  16'b0000_0001_0010_1000; //add rd 1 rn 1 rm 2  then result is " 15"
15=7+8
DATAmem[32] =  16'b0000_0011_0111_0000; //add rd 3 rn 4 rm 3  then result r3+4
DATAmem[36] =  16'b1000_0010_0110_1100; //ldr r2,[r3]
DATAmem[40] =  16'b0000_0001_0010_1000; //add rd 1 rn 1 rm 2  then result is " 24"
24=9+15
DATAmem[44] =  16'b0000_0011_0111_0000; //add rd 3 rn 4 rm 3  then result r3+4
DATAmem[48] =  16'b1000_0010_0111_0000; //ldr r2,[r3]
DATAmem[52] =  16'b0000_0001_0010_1000; //add rd 1 rn 1 rm 2  then result is " 35"
35=24+11
DATAmem[56] =  16'b1111_1111_1111_1111;
//memory
DATAmem[112]    =  16'b0000_0000_0000_0011;//3
DATAmem[116]    =  16'b0000_0000_0000_0101;//5
DATAmem[120]    =  16'b0000_0000_0000_0111;//7
DATAmem[124]    =  16'b0000_0000_0000_1001;//9
DATAmem[128]    =  16'b0000_0000_0000_1011;//11
```

Simulation Result of the subroutine

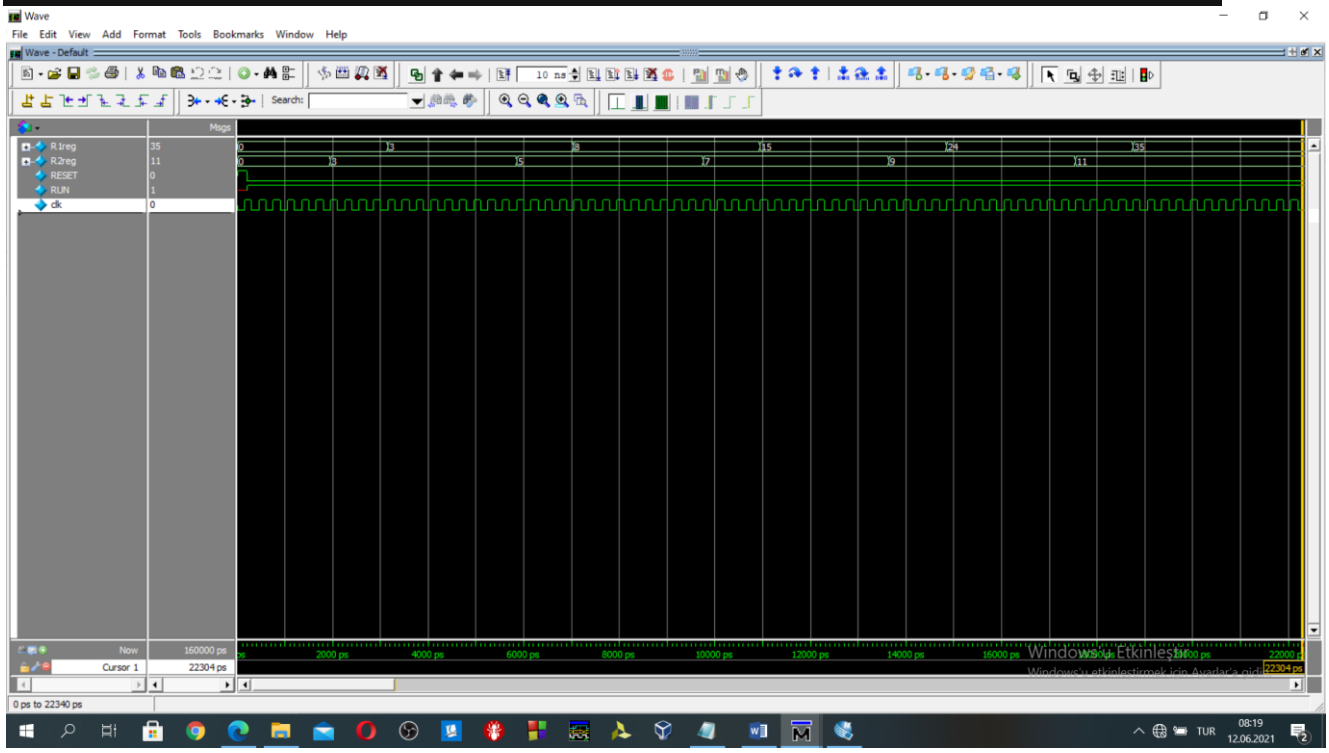


Figure 7: Modelsim Simulation of the subroutine that computes the sum of an array of numbers and stores it in a memory

R3=112 which points the first element of the array.

R4=4, it is added to R3 register to reach next element of the array

As can be seen in Figure 7, R2 register keeps the loaded value from the memory. This value is added to R1 register.

In Figure 7, R2 is loaded 3 then added to R1 this repeated 5 times.

```
//3. Write a subroutine that determines the evenness/oddity of a 8-bit number,
n7n6n5n4n3n2n1n0. If odd, the expected output is n4n3n2n10n7n6n5, otherwise, 0n4n3n2n1000.
/*/*//* 3. subroutine *//*/*/
//then r1 contains the our number which is 8-bit number which will be evaluated
//r0's content initialized as 0000 0001

            and r2,r1,r0 //if r2 is zero then even
            beq even_case
            //odd case manipulations
            // n7 n6 n5 n4 n3 n2 n1 n0-->n4 n3 n2 n1 0 n7 n6 n5
            // 0 n7 n6 n5 n4 n3 n2 n1 get rid of n0-->lsr

            lsr r1,r1//then rotate left
            rol r1,r1 // n7 n6 n5 n4 n3 n2 n1 0
            rol r1,r1 // n6 n5 n4 n3 n2 n1 0  n7
            rol r1,r1 // n5 n4 n3 n2 n1 0  n7 n6
            rol r1,r1 // n4 n3 n2 n1 0  n7 n6 n5
            b   jump
even_case
            //even case manipulations
            // n7 n6 n5 n4 n3 n2 n1 n0-->0 n4 n3 n2 n1 0 0 0
            lsr r1,r1 // 0 n7 n6 n5 n4 n3 n2 n1
            lsl r1,r1 // n7 n6 n5 n4 n3 n2 n1 0
            lsl r1,r1 // n6 n5 n4 n3 n2 n1 0  0
            lsl r1,r1 // n5 n4 n3 n2 n1 0  0  0
            lsl r1,r1 // n4 n3 n2 n1 0  0  0  0
            lsr r1,r1 // 0  n4 n3 n2 n1 0  0  0
jump
            END


/*Machine Code of subroutine 3*/
//instructions for the subroutine 3
            DATAmem[0]  =  16'b0010_0010_0010_0000;//and  r2=r1&r0
            DATAmem[4]  =  16'b1101_1000_0001_1100; //beq even_case then branch 40
            //beq will be added
            DATAmem[8]  =  16'b0110_0001_0010_0000;//lsr r1= lsr r1
            DATAmem[12] =  16'b0100_0001_0010_0000;//rol r1=rol r1
            DATAmem[16] =  16'b0100_0001_0010_0000;//rol r1=rol r1
            DATAmem[20] =  16'b0100_0001_0010_0000;//rol r1=rol r1
            DATAmem[24] =  16'b0100_0001_0010_0000;//rol r1=rol r1
            DATAmem[28] =  16'b1100_0000_0001_1100; //b jump pc+8+imm8(28)=64
//even_case
            DATAmem[40] =  16'b0110_0001_0010_0000;//lsr r1= lsr r1
            DATAmem[44] =  16'b0101_0001_0010_0000;//lsl r1= lsl r1
            DATAmem[48] =  16'b0101_0001_0010_0000;//lsl r1= lsl r1
            DATAmem[52] =  16'b0101_0001_0010_0000;//lsl r1= lsl r1
            DATAmem[56] =  16'b0101_0001_0010_0000;//lsl r1= lsl r1
            DATAmem[60] =  16'b0110_0001_0010_0000;//lsr r1= lsr r1
//jump
            DATAmem[64] =  16'b1111_1111_1111_1111;
```

Simulation Results of the subroutine

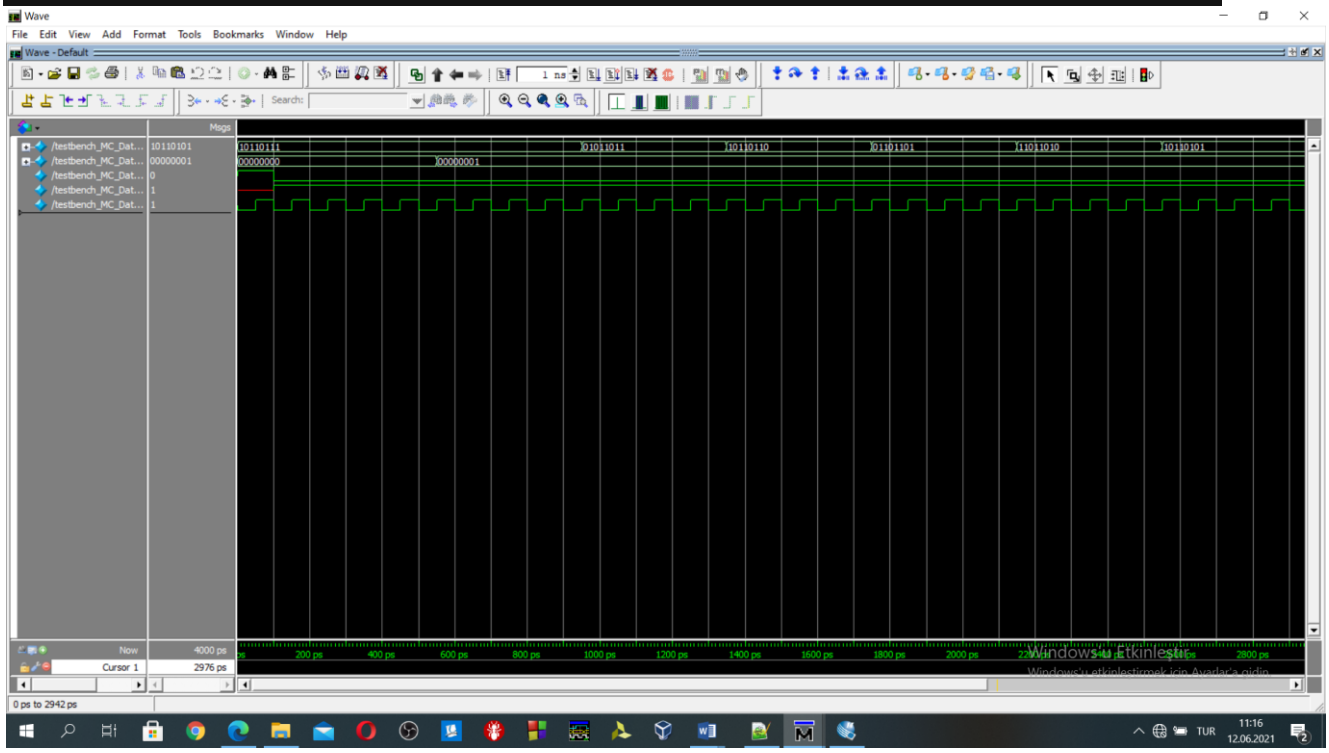


*Figure 8:modelsim simulation of the subroutine that determines the **oddity** of a 8-bit number(8'b10110111)*

After "and" instruction in memory location 0, by looking R2 register, we decide that the 8 bits number is odd or even. In Figure 8, R2's content is 1. PCwrite operation in beq branch operation to 40 does not occur. The PC continues with the PC+4. There is no jump until the unconditional branch at memory location 28. As can be seen in Figure 8 oddity control is working.

n7 n6 n5 n4 n3 n2 n1 n0-->n4 n3 n2 n1 0 n7 n6 n5

Test number is (8'b10110111) → (8'b10110101) matches ☺

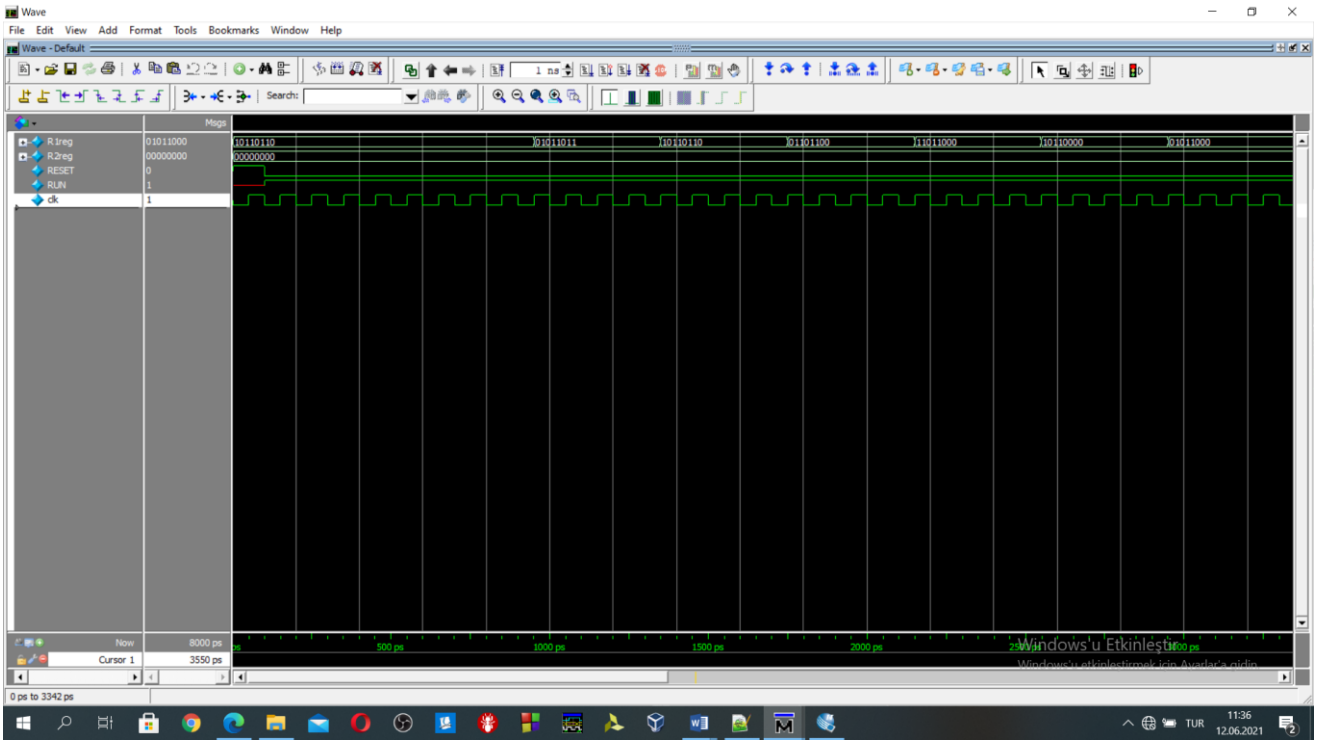


*Figure 9:modelsim simulation of the subroutine that determines the **evenness** of a 8-bit number(8'b10110110)*

After "and" instruction in memory location 0, by looking R2 register, we decide that the 8 bits number is odd or even. In Figure 8, R2's content is 0. PCwrite operation in beq branch operation to 40 occurs. As can be seen in Figure 8 **evenness** control is working.

```
n7 n6 n5 n4 n3 n2 n1 n0-->0 n4 n3 n2 n1 0 0 0
```

Test number is (8'b10110110) → (8'b01011000) matches ☺

```
    /// UNIFIED DATAPATH and CONTROLLER DESIGN CODE ///
module datapath_controller_MC(
input clk,
input RESET,
input RUN,
output wire [7:0] R1reg,
output wire [7:0] R2reg
);

wire [1:0]  w_cond;
wire [1:0]  w_OP;
wire [2:0]  w_type;
wire           w_PCWrite;
wire [1:0]  w_AdrSrc;
wire           w_MemWrite;
wire           w_IRWrite;
wire [2:0]  w_RegSrc;
wire           w_RegWrite;
wire           w_ImmSrc;
wire           w_ALUSrcA;
wire [1:0]  w_ALUSrcB;
wire [3:0]  w_ALUControl;
wire [1:0]  w_ResultSrc;
wire [3:0]  w_flags;
wire [2:0]  w_Rd;

MultiCycle_Controller my_MC_controller(
//inputs
.cond(w_cond),
.OP(w_OP),
.type(w_type),
.flags(w_flags),
.Rd(w_Rd),
.RUN(RUN),
.clk(clk),
.PCWrite(w_PCWrite),
.AdrSrc(w_AdrSrc),
.MemWrite(w_MemWrite),
.IRWrite(w_IRWrite),
.RegSrc(w_RegSrc),
.RegWrite(w_RegWrite),
.ImmSrc(w_ImmSrc),
.ALUSrcA(w_ALUSrcA),
.ALUSrcB(w_ALUSrcB),
.ALUControl(w_ALUControl),
.ResultSrc(w_ResultSrc)
);


 multi_cycle_datapath my_MC_datapath(
    .clk(clk),
    .rst(RESET),
    .PCWrite(w_PCWrite),
    .MemWrite(w_MemWrite),
    .IRWrite(w_IRWrite),
    .ImmSrc(w_ImmSrc),
    .RegWrite(w_RegWrite),
    .ALUSrcA(w_ALUSrcA),
    .AdrSrc(w_AdrSrc),
```

```
    .ALUControl(w_ALUControl),
    .ALUSrcB(w_ALUSrcB),
    .RegSrc(w_RegSrc),
    .ResultSrc(w_ResultSrc),
    .R1out(R1reg),
    .R2out(R2reg),
    .ALU_flags(w_flags),
    .cond(w_cond),
    .OP(w_OP),
    .type(w_type),
    .Rd(w_Rd)

);

endmodule


                        ///DATAPATH CODE////
module multi_cycle_datapath(
    clk,
    rst,
    //control signal inputs
    PCWrite,
    MemWrite,
    IRWrite,
    ImmSrc,
    RegWrite,
    ALUSrcA,
    AdrSrc,
    ALUControl,
    ALUSrcB,
    RegSrc,
    ResultSrc,

    //desmostration purpose registers
    R1out,
    R2out,

    //controller inputs which is output for the datapath
    ALU_flags,
    cond,
    OP,
    type,
    Rd

);


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 [7:0] R1out;
output wire [7:0] R2out;

output wire [1:0]  cond;
output wire [1:0]  OP;
output wire [2:0]  type;
output wire [3:0]  ALU_flags;
output wire [2:0]  Rd;



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));

    //assignment for the required controller inputs
assign  ALU_flags = X;
assign cond = WREGout[1:0];
assign OP = WREGout[15:14];
assign type = WREGout[13:11];
assign  Rd=WREGout[10:8];
```

```
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
```

//CONTROLLER DESIGN CODE//

```
module MultiCycle_Controller(
//inputs
input [1:0]  cond,
input [1:0]  OP,
input [2:0]  type,
input [3:0]  flags,
input [2:0]  Rd,
input        RUN,
input        clk,
//outputs
output reg       PCWrite,
output reg [1:0] AdrSrc,
output reg       MemWrite,
output reg       IRWrite,
output reg [2:0] RegSrc,
output reg       RegWrite,
output reg       ImmSrc,
output reg       ALUSrcA,
output reg [1:0] ALUSrcB,
output reg [3:0] ALUControl,
output reg [1:0] ResultSrc
);

reg [2:0] state_counter=3'b000;
reg [3:0] FLAG_REG;

always @(posedge clk)
            begin

                if(RUN==1)
                     begin
                        //fetch
```

```
                    if(state_counter==3'b000)
                        begin
                            PCWrite=1;
                            IRWrite=1;
                            ALUSrcA=1;
                            AdrSrc=2'b00;
                            RegWrite=0;
                            ALUControl=4'b0000;
                            ALUSrcB=2'b10;
                            ResultSrc=2'b10;
                            state_counter=state_counter+3'b001;
                        end

                    //decode
                    else if(state_counter==3'b001)
                        begin
                            PCWrite=0;
                            MemWrite=0;
                            IRWrite=0;
                            RegWrite=0;
                            ALUSrcA=1;
                            ALUControl=4'b0000;
                            ALUSrcB=2'b10;
                            //regSrc assignment
                            if(OP==2'b11) //branch
                                begin
                                    if(type==3'b000)//b
                                    begin
                                        RegSrc=3'b000;
                                    end

                                    else if(type==3'b001)//bl
                                    begin
                                        RegSrc=3'b001;
                                    end

                                    else if(type==3'b011)//beq
                                    begin
                                        RegSrc=3'b000;
                                    end

                                    else if(type==3'b100)//bne
                                    begin
                                        RegSrc=3'b000;
                                    end

                                    else if(type==3'b101)//bc
                                    begin
                                        RegSrc=3'b000;
                                    end
                                    else if(type==3'b110)//bnc
                                    begin
                                        RegSrc=3'b000;
                                    end
                                    else if(type==3'b010)//bi
                                    begin
                                        RegSrc=3'b101;
                                    end
                                end
```

```
                        else if(OP==2'b00) //data
                            begin
                                RegSrc=3'b100;
                            end
                        else if(OP==2'b01) //shift
                            begin
                                RegSrc=3'b100;
                            end

                        else if(OP==2'b10) //memory
                            begin
                            if(type[2:1]==2'b10)//str
                                begin
                                RegSrc=3'b110;
                                end
                            else // ldr/i
                                begin
                                RegSrc=3'b100;
                                end
                            end


                        state_counter=state_counter+3'b001;
                    end

                    //execution phase
                else if(state_counter==3'b010) //cycle 3
                    begin

                    if(OP==2'b00)//data
                        begin
                            ALUSrcA=0;

                            case(type)
                                    3'b000: ALUControl=4'b0000; //add
                                    3'b010: ALUControl=4'b0001; //sub
                                    3'b100: ALUControl=4'b0010; //and
                                    3'b101: ALUControl=4'b0011; //orr
                                    3'b110: ALUControl=4'b0100; //xor
                                    3'b111: ALUControl=4'b0101; //clr
                            endcase
                            ALUSrcB=2'b00;
                            RegSrc=3'b100;
                            ResultSrc=2'b10;
                            RegWrite=0;
                            FLAG_REG=flags; //update the flags for data
processing

                            state_counter=state_counter+3'b001;
                        end
                    else if(OP==2'b01)//shift
                        begin
                            ALUSrcA=0;

                            case(type)
                                    3'b000: ALUControl=4'b0110; //rol
                                    3'b001: ALUControl=4'b0111; //ror
                                    3'b010: ALUControl=4'b1000; //lsl
                                    3'b011: ALUControl=4'b1010; //asr
```

```
                                        3'b100: ALUControl=4'b1001; //lsr
                                    endcase
                                    ALUSrcB=2'b00;
                                    RegSrc=3'b100;
                                    ResultSrc=2'b10;
                                    RegWrite=0;
                                    FLAG_REG=flags;
                                    state_counter=state_counter+3'b001;
                                end
                            else if(OP==2'b10)//memory
                                begin

                                    case(type[2:1])
                                    2'b00://ldr cycle 3
                                        begin
                                            ImmSrc=0;
                                            ALUSrcB=2'b01;
                                            ALUSrcA=0;
                                            ALUControl=4'b0000;

                                            state_counter=state_counter+3'b001;
                                        end
                                    2'b01://ldi last cycle
                                        begin
                                            ImmSrc=1;
                                            ALUSrcB=2'b01;
                                            ResultSrc=2'b11;
                                            RegWrite=1;

                                            state_counter=3'b000;//go to next
cycle

                                        end
                                    2'b10://str cycle 3
                                        begin
                                            ImmSrc=0;
                                            ALUSrcB=2'b01;
                                            ALUSrcA=0;
                                            ALUControl=4'b0000;

                                            state_counter=state_counter+3'b001;
                                        end
                                    endcase
                                end

                            else if(OP==2'b11)//branch cycle 3 last cycle
                                    begin
                                        case(type)
                                        3'b000: //und. branch
                                        begin
                                            PCWrite=1;
                                            ALUSrcA=0;
                                            ALUControl=4'b0000;
                                            ALUSrcB=2'b01;
                                            ResultSrc=2'b10;
                                            state_counter=3'b000;
                                        end

                                            3'b001: // branch link
                                            begin
```

```
                                    PCWrite=1;
                                    ALUSrcA=0;
                                    ALUControl=4'b0000;
                                    ALUSrcB=2'b01;
                                    ResultSrc=2'b10;
                                    RegWrite=1;
                                    state_counter=3'b000;

                                end

                                3'b011: //beq
                                begin
                                //branch cycle 3 equal means Z=1 nzcv
                                //look at the FLAG_REG register value to
decide execue or not
                                if(FLAG_REG[2]==1)
                                begin
                                    PCWrite=1;
                                end
                                else
                                begin
                                PCWrite=0;
                                end

                                ALUSrcA=0;
                                ALUControl=4'b0000;
                                ALUSrcB=2'b01;
                                ResultSrc=2'b10;
                                state_counter=3'b000;

                                end

                                3'b100: //bne
                                //branch cycle 3 not equal means Z=0 nzcv
                                begin
                                if(FLAG_REG[2]==0)
                                begin
                                    PCWrite=1;
                                end
                                else
                                begin
                                PCWrite=0;
                                end

                                ALUSrcA=0;
                                ALUControl=4'b0000;
                                ALUSrcB=2'b01;
                                ResultSrc=2'b10;
                                state_counter=3'b000;

                                end

                                3'b101: //bc
                                begin
                                //branch cycle 3 carry set means c=1 nzcv
                                if(FLAG_REG[1]==1)
                                        begin
                                            PCWrite=1;
                                        end
```

```
                                        else
                                                begin
                                                    PCWrite=0;
                                                end

                                        ALUSrcA=0;
                                        ALUControl=4'b0000;
                                        ALUSrcB=2'b01;
                                        ResultSrc=2'b10;
                                        state_counter=3'b000;
                                        end

                                        3'b110: //bnc
                                        begin
                                        //branch cycle 3 not carry set means c=0
nzcv
                                        if(FLAG_REG[1]==0)
                                                begin
                                                    PCWrite=1;
                                                end
                                        else
                                                begin
                                                    PCWrite=0;
                                                end

                                            ALUSrcA=0;
                                            ALUControl=4'b0000;
                                            ALUSrcB=2'b01;
                                            ResultSrc=2'b10;
                                            state_counter=3'b000;

                                        end

                                        3'b010: //bi
                                        begin
                                            ALUSrcB=2'b00;
                                            ResultSrc=2'b11;
                                            PCWrite=1;
                                            state_counter=3'b000;
                                        end

                                        3'b111: //end
                                        begin
                                            //RUN=0; //end of the instructions
                                            state_counter=3'b000;
                                        end
                                        endcase
                                    end

                                end//cycle 3 ends

                                //execution phase
                                else if(state_counter==3'b011) //cycle 4
                                begin

                                    if(OP[1]==1'b0)
                                    begin
                                        ResultSrc=2'b00;
                                        RegWrite=1;
```

```
                                    state_counter=3'b000;//data and shift end
                                end
                                else if(OP==2'b10)
                                begin
                                    if(type[2:1]==2'b10)//str
                                            begin
                                            ResultSrc=2'b00;
                                            AdrSrc=2'b01;
                                            MemWrite=1;
                                            state_counter=3'b000; //end store
                                            end
                                    else if(type[2:1]==2'b00) //ldr
                                            begin
                                                ResultSrc=2'b00;
                                                AdrSrc=2'b01;
                                                MemWrite=0;

state_counter=state_counter+3'b001;//cycle 4 for the ldr
                                            end

                                end
                            end

                            else if(state_counter==3'b100) //cycle 5
                            begin
                            ResultSrc=2'b01;
                            RegWrite=1;
                            state_counter=3'b000; //end ldr
                            end



                    end//run


    end//always


endmodule
```

```
                ///TESTBENCH OF THE CONTROL UNIT///

module testbench_MC_Controller();
//inputs are reg
//outputs are wire
//assume that bus width is 3 to test the result
//inputs
reg [1:0]  cond;
reg [1:0]  OP;
reg [2:0]  type;
reg [3:0]  flags;
reg          RUN;
reg          clk;


//outputs
wire         PCWrite;
wire [1:0] AdrSrc;
wire         MemWrite;
wire         IRWrite;
wire [2:0] RegSrc;
wire         RegWrite;
wire         ImmSrc;
wire         ALUSrcA;
wire [1:0] ALUSrcB;
wire [3:0] ALUControl;
wire [1:0] ResultSrc;
// instantiate device under test
MultiCycle_Controller DUT(
//inputs
cond,
OP,
type,
flags,
RUN,
clk,

//outputs
PCWrite,
AdrSrc,
MemWrite,
IRWrite,
RegSrc,
RegWrite,
ImmSrc,
ALUSrcA,
ALUSrcB,
ALUControl,
ResultSrc
);



initial
begin

RUN=1;
//inputs
cond=2'b00;
```

```
OP=2'b00;
type=3'b000;
flags=4'b0000;

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
always // no sensitivity list, so it always executes
    begin
//fetch
OP=2'b00;
type=3'b000;

#400; //sub
flags=4'b1011;//////flag update
OP=2'b00;
type=3'b010;

#400; //and

OP=2'b00;
type=3'b100;

#400; //orr

OP=2'b00;
type=3'b101;

#400; //xor

OP=2'b00;
type=3'b110;

#400; //clr

OP=2'b00;
type=3'b111;

#400; //rol

OP=2'b01;
type=3'b000;

#400; //ror

OP=2'b01;
type=3'b001;

#400; //lsl

OP=2'b01;
type=3'b010;
```

```
#400; //asr

OP=2'b01;
type=3'b011;

#400; //lsr

OP=2'b01;
type=3'b100;

#400; //ldi

OP=2'b10;
type[2:1]=2'b01;

#300; //ldr

OP=2'b10;
type[2:1]=2'b00;

#500; //str

OP=2'b10;
type[2:1]=2'b10;

#400; //branch und.

OP=2'b11;
type=3'b000;

#300; //branch bl

OP=2'b11;
type=3'b001;

#300; //branch ind.

OP=2'b11;
type=3'b010;

#300; //branch eq

//nzcv
flags=4'b0100; //z=1 means equal case
OP=2'b11;
type=3'b011;

#300; //branch not eq

//nzcv
flags=4'b1011; //z=0 means not equal case
OP=2'b11;
type=3'b100;
#300;

//bc
flags=4'b1010; //c=1 means carry set case
OP=2'b11;
type=3'b101;
```

```
#300;

//bnc
flags=4'b1000; //c=0 means not cary set case
OP=2'b11;
type=3'b110;
#300;


//END inst
flags=4'b1000;
OP=2'b11;
type=3'b111;
#300;
end

endmodule
```

//THE TESTBENCH OF THE UNIFIED CODE//

```
module testbench_MC_Data_Controller();
//inputs are reg
//outputs are wire
//assume that bus width is 3 to test the result
//inputs
reg clk;
reg RESET;
reg RUN;
wire [7:0] R1reg;
wire [7:0] R2reg;
// instantiate device under test
datapath_controller_MC DUT(
clk,
RESET,
RUN,
R1reg,
R2reg
);

initial
begin
RESET=1;
#100;
RESET=0;
RUN=1;
end

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