2. Multi-Cycle CPU controller design Modelsim Simulation Results

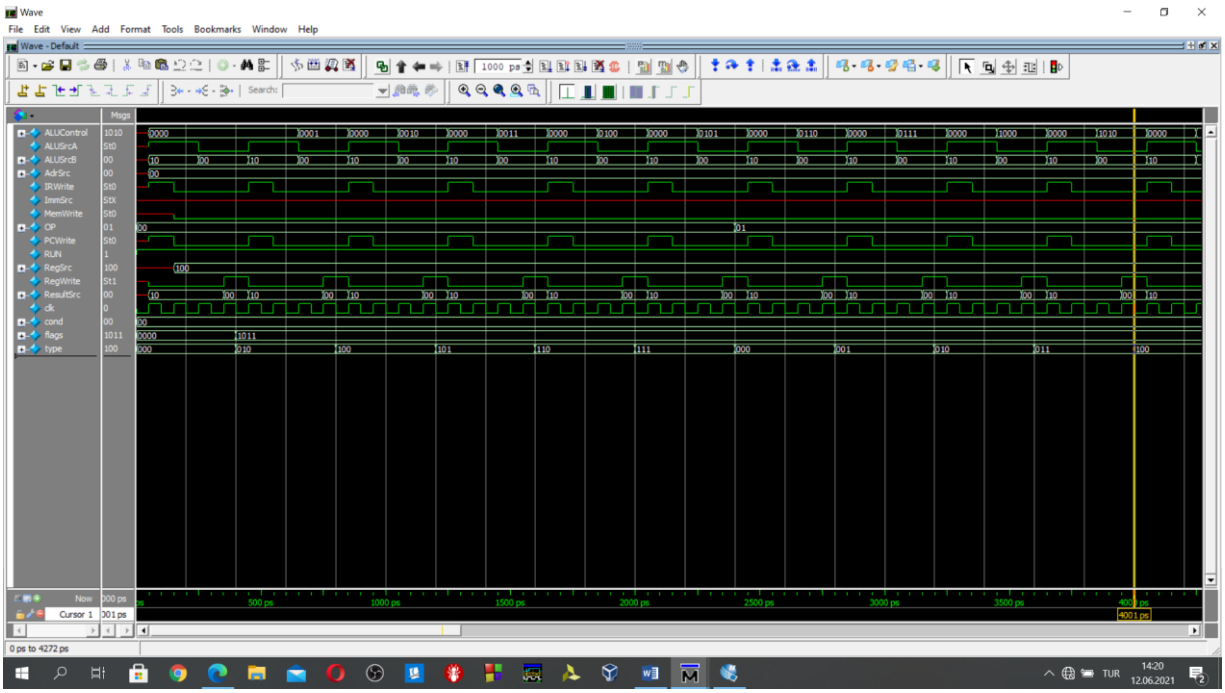

Figure 1: Multicycle controller unit simulation via modelsim 0-4000ns

```
//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;
```

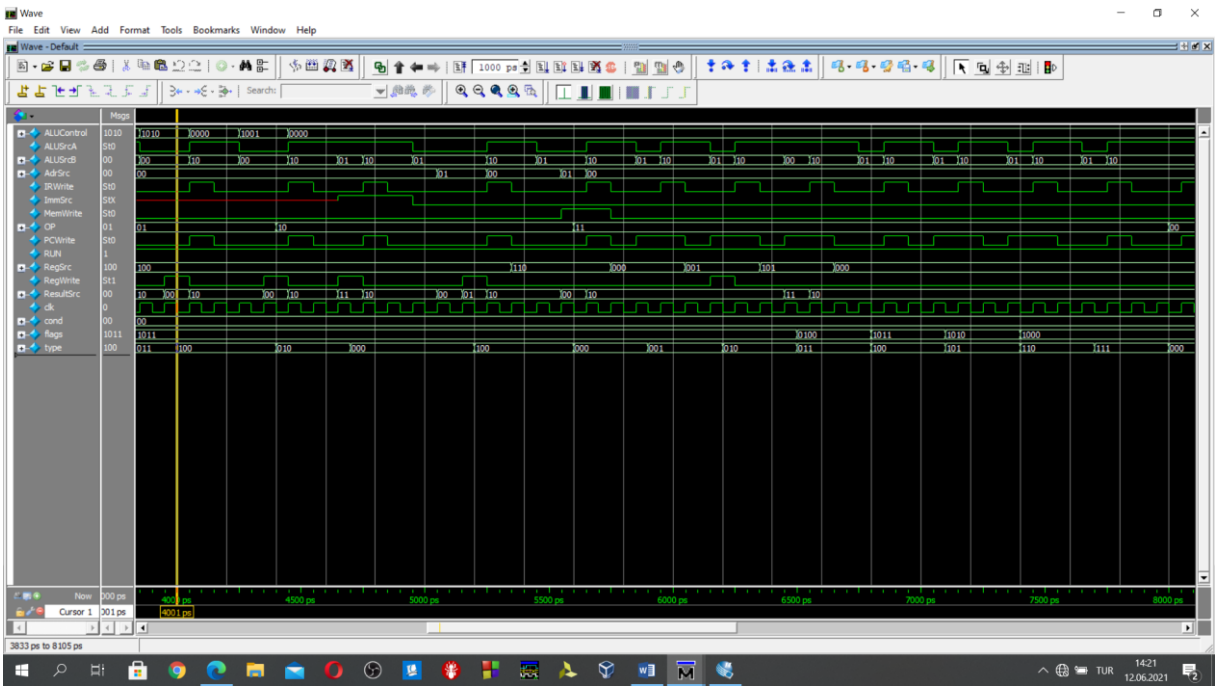

Figure 2: Multicycle controller unit simulation via modelsim 4000ns-8000ns

```
//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;
```

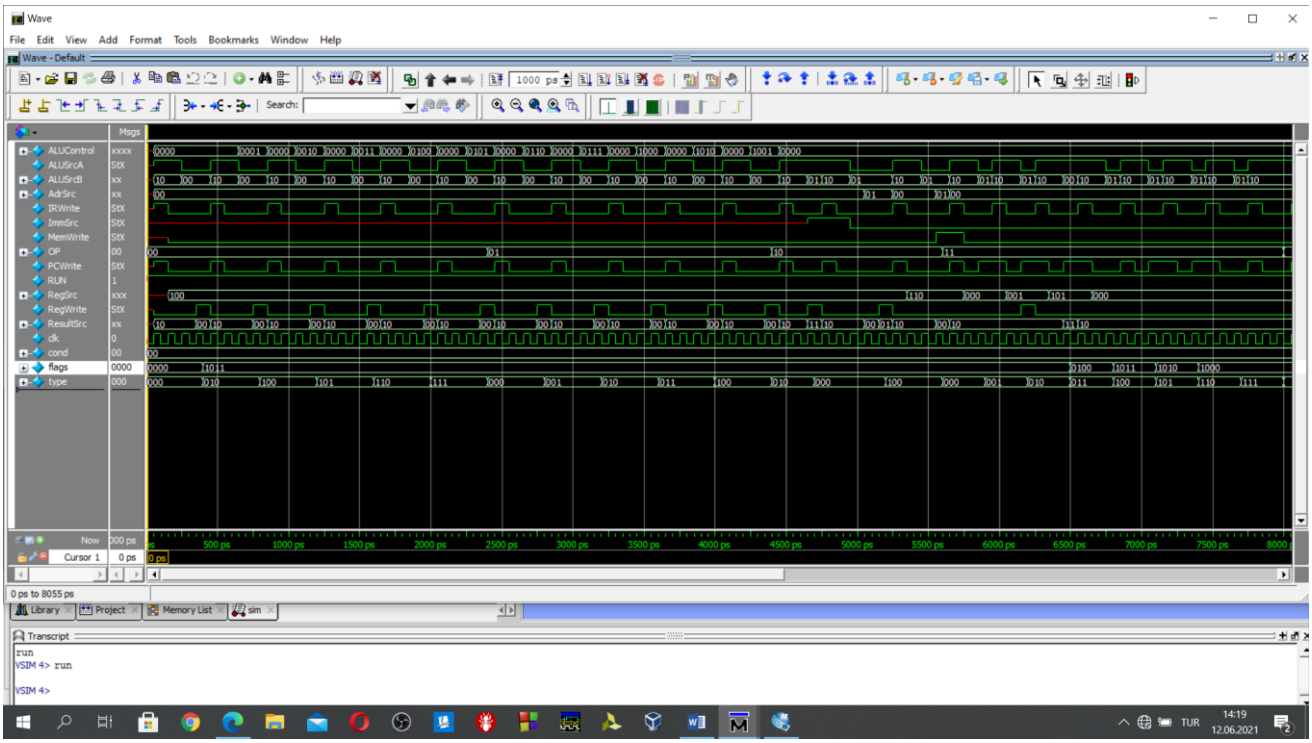

Figure 3:Multicycle controller unit simulation via modelsim 0-8000ns

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.

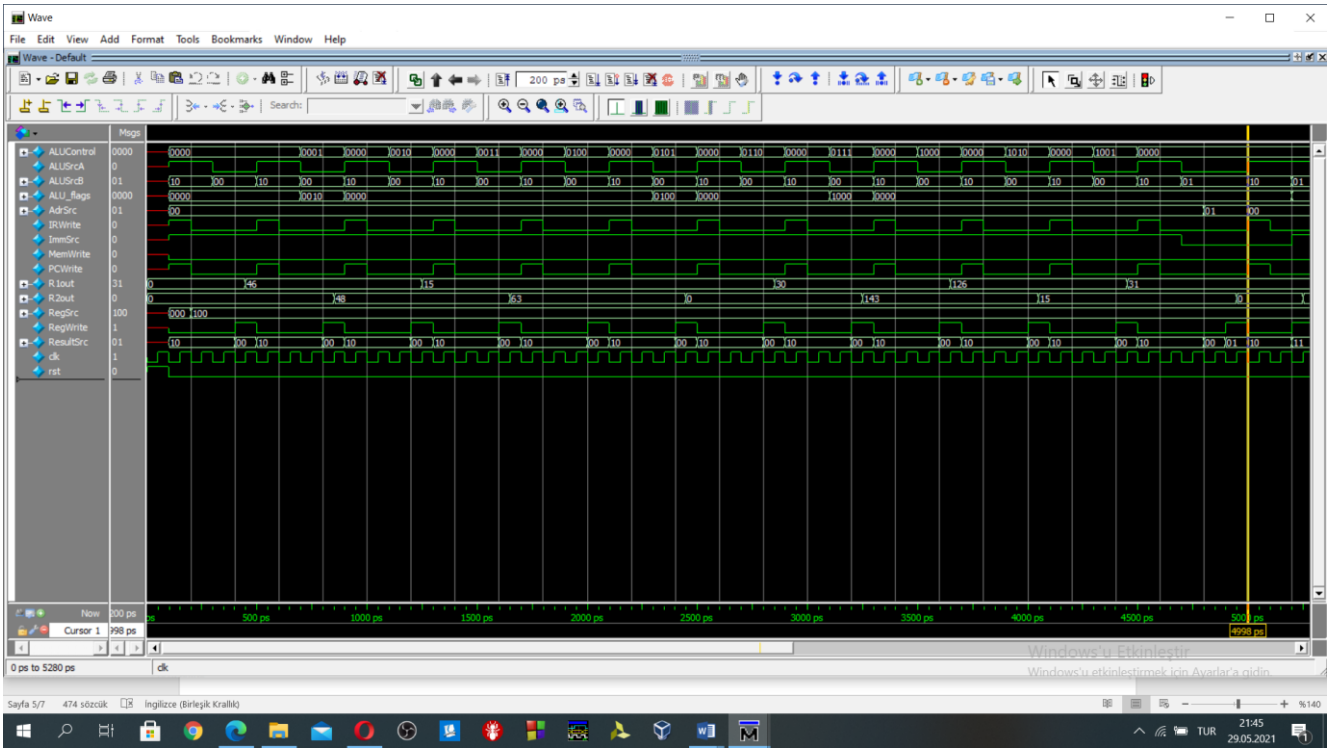

Figure 4:Multicycle datapath simulation result from lab4

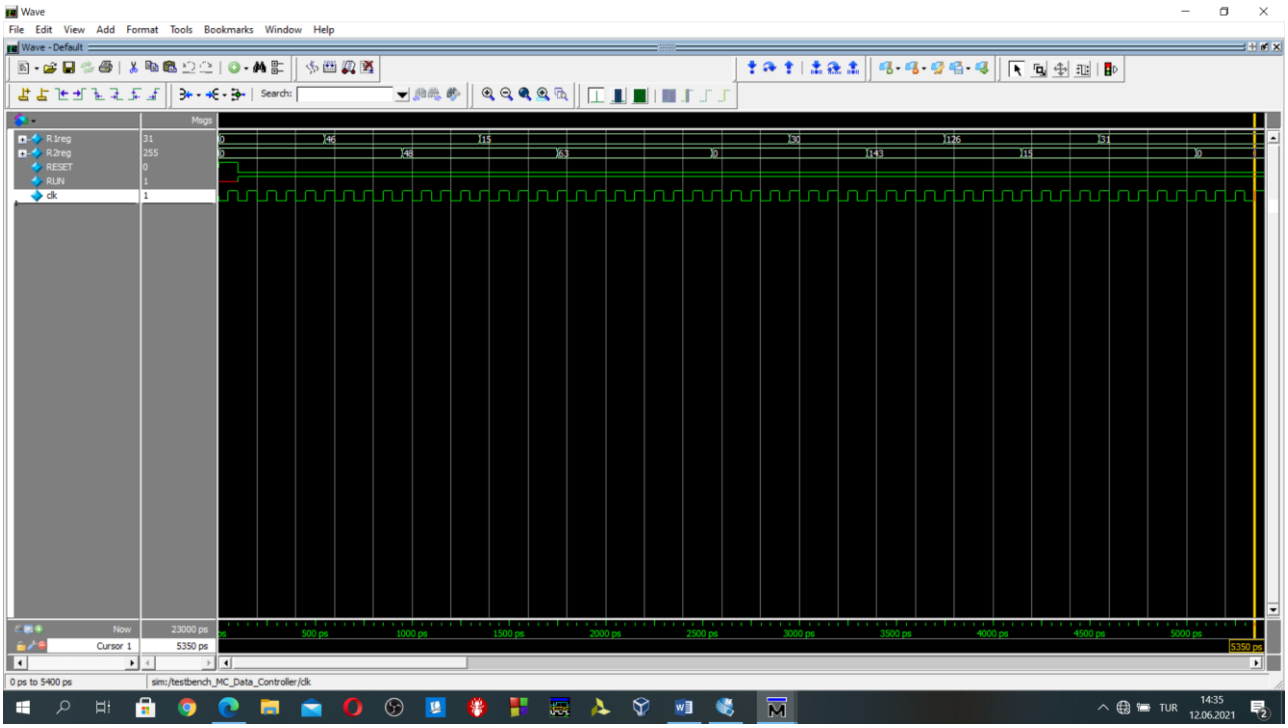

Figure 5:multicycle simulation which includes datapath and controller units

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

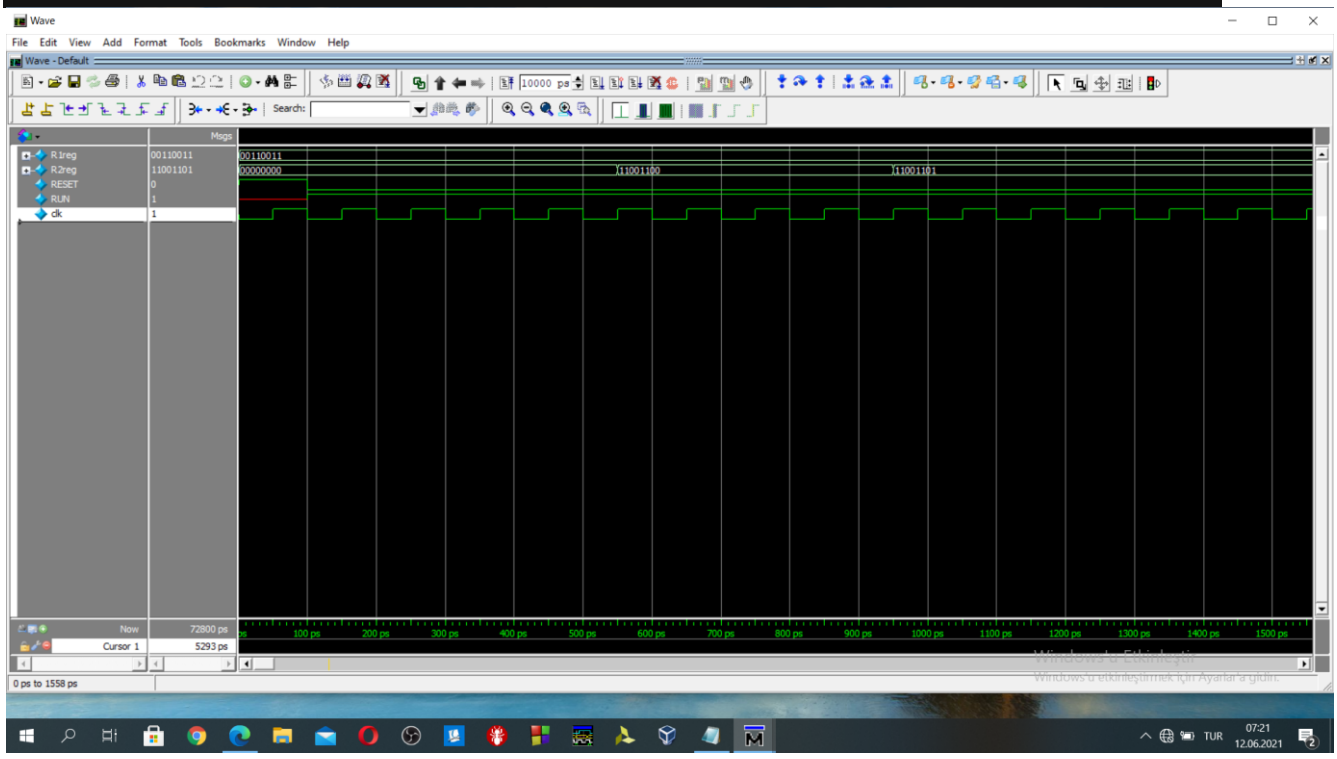

Figure 6:Modelsim Simulation of the subroutine that get an 8-bit number and computes it 2's complement(8'b00110011)

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

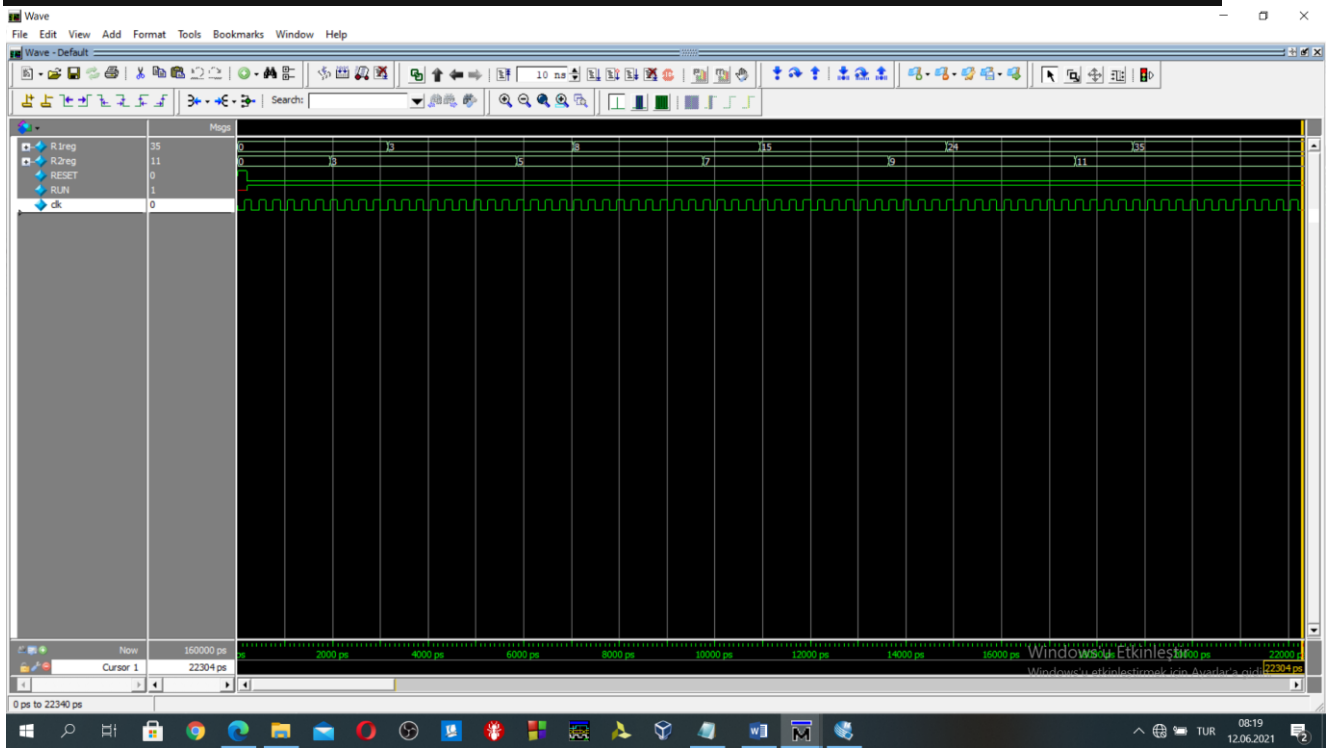

Figure 7: Modelsim Simulation of the subroutine that computes the sum of an array of numbers and stores it in a memory

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

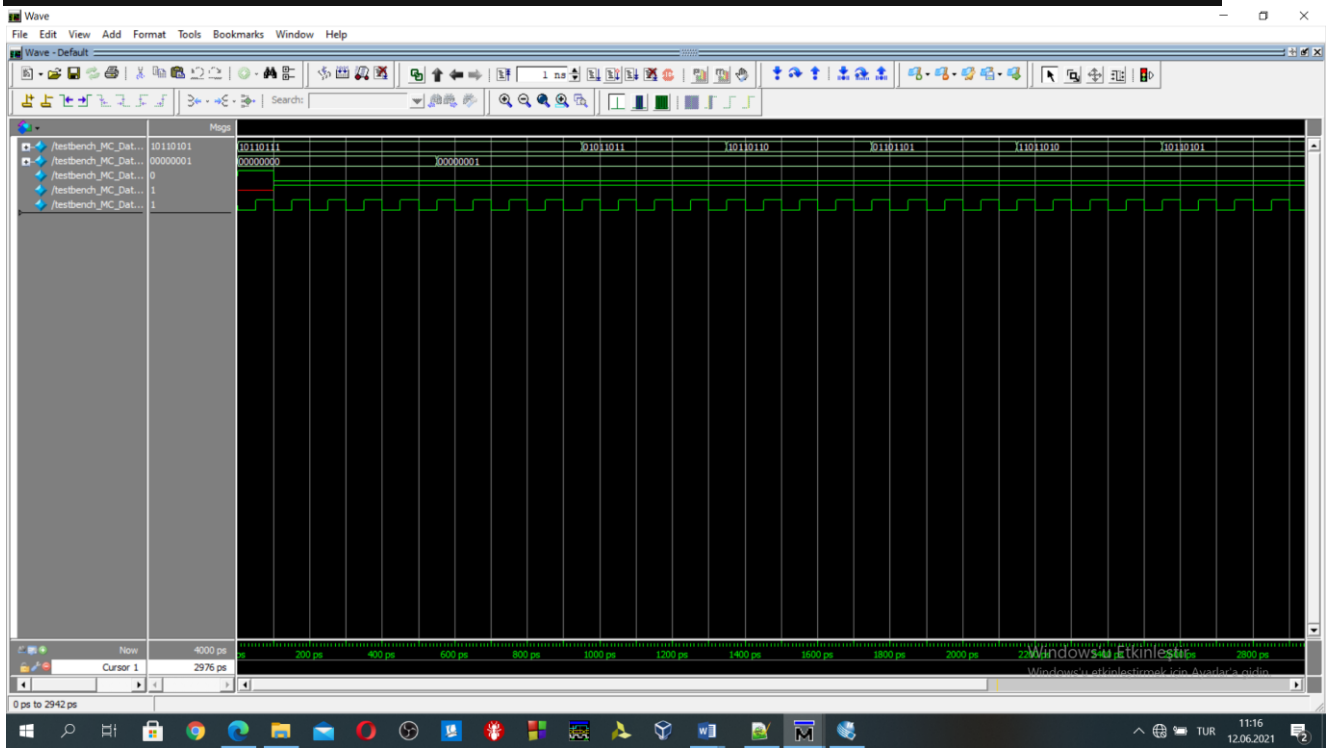

*Figure 8:modelsim simulation of the subroutine that determines the **oddity** of a 8-bit number(8'b10110111)*

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 ☺

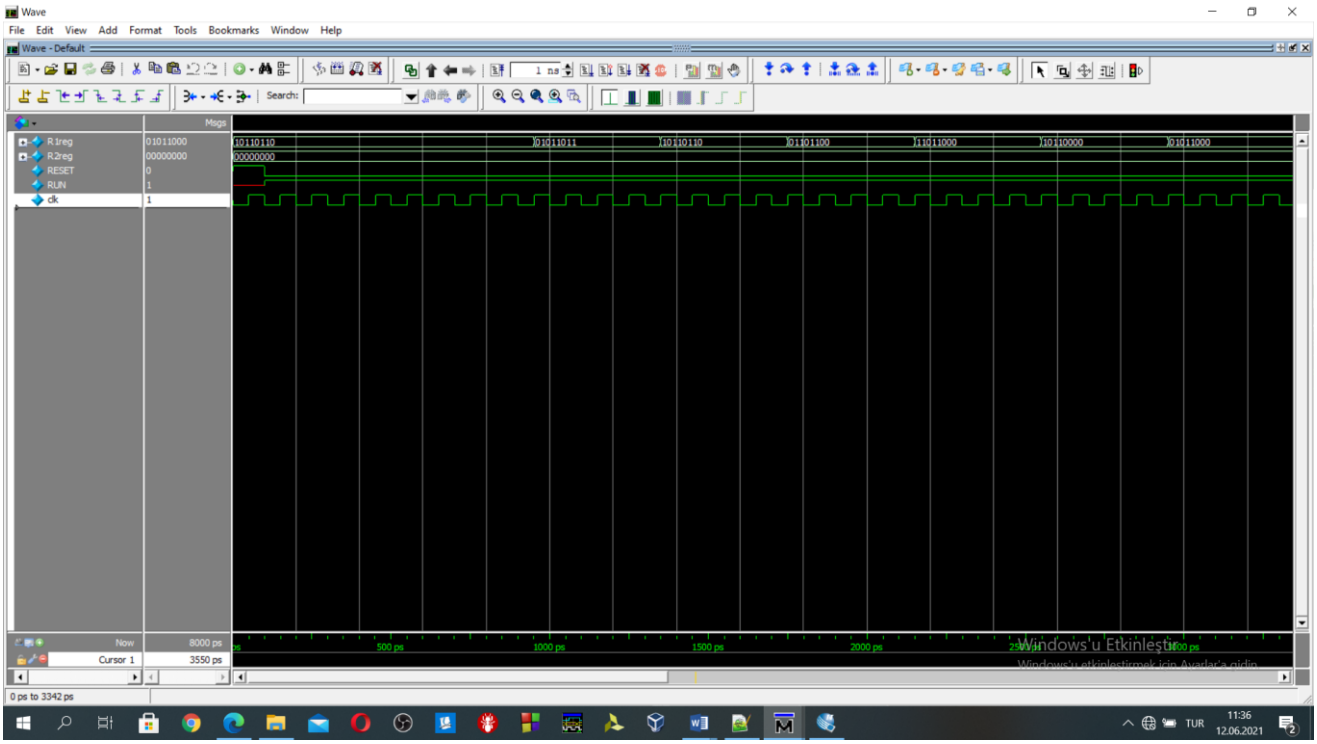

*Figure 9:modelsim simulation of the subroutine that determines the **evenness** of a 8-bit number(8'b10110110)*

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 ☺