CSE331 – Computer Organization
Assignment-4
Report

Date

07/01/2022

Author

Gökbey Gazi KESKİN

Table of Contents

Summary 3
Control Signals 4
Main Controller 4
a) 1-bit outputs 4
b) aluOp 4
ALU Controller 5
Testbenches 5
Registers 5
Instruction Memory 6
Data Memory 6
Processor 7
Arithmetic Operations 7
Logic Operations 7
Memory Operations 8
Branch Operations 9
.... 9

Summary

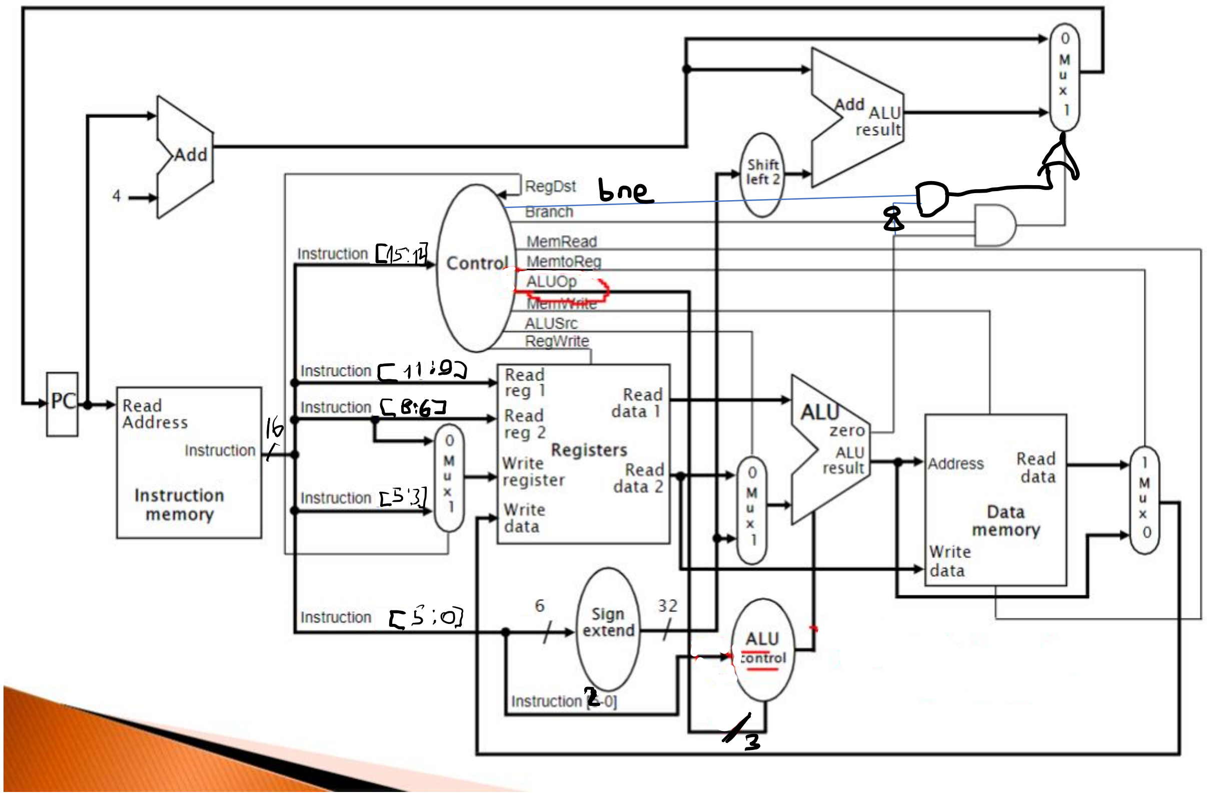


Figure 1

I followed this schematic while creating the processor. I added a bne signal for branch if not equal. I already had the ALU from last assignment, but I added zero output to it by using a comparator. New modules which I didn’t have from last assignment are:

-Instruction Memory

-Registers

-Data Memory

-Processor itself

So I only added testbenches of these 4 because other modules are tested on the previous assignment and project directory is already too crowded.

Everything works as expected. There is only one problem and I explained it on testbench of the processor.

Control Signals

Main Controller

a) 1-bit outputs

Opcode3	Opcode2	Opcode1	Opcode0	regDst	branch	memRead	memToReg	memWrite	aluSrc	regWrite
0	0	0	0	1	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	1	1
0	0	1	1	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1	1
0	1	0	1	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	0	0	1	1
1	0	0	0	0	0	1	1	0	1	1
1	0	0	1	0	0	0	0	1	1	0

regDst = opCode3’&opCode2’&opCode1’&opCode0’

branch= opCode3’&opCode2&(opCode1 xor opCode0)

memRead = opCode3&opCode2’&opCode1’&opCode0’

memToReg = memRead

memWrite= opCode3&opCode0

aluSrc = opCode3&opCode2’&opCode1’ |opCode3’(opCode2&opCode1’&opCode0’|opCode2’&opCode0 | opCode2’&opCode1 | opCode1&opCode0)

regWrite = (branch | memWrite)’

bne = opCode3’ & opCode2 & opCode1 & opCode0’

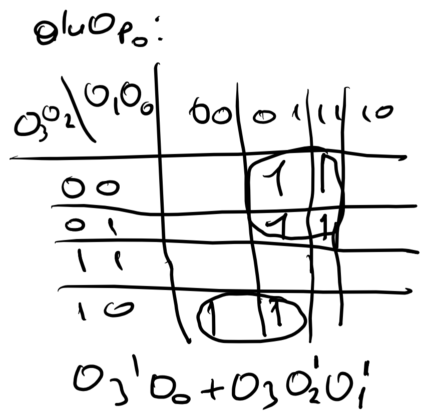
b) aluOp

Opcode3	Opcode2	Opcode1	Opcode0	aluOp2	aluOp1	aluOp0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	1	0	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	1	0	0	1

aluOp2 = opCode2

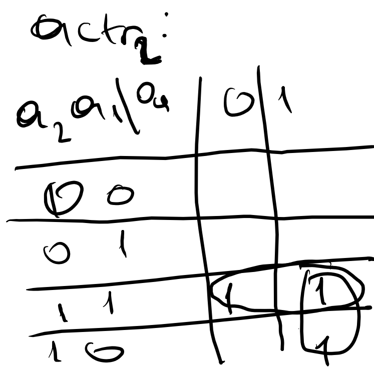
aluOp1 = opCode1

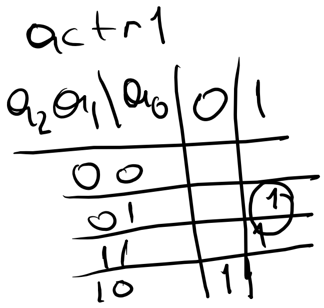
aluOp0 = opCode3’opCode0 + opCode3opCode2’opCode1’



ALU Controller

aluOp2	aluOp1	aluOp0	Func2	Func1	Func0	aluCtr2	aluCtr1	aluCtr0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0
0	0	0	0	1	0	1	0	1
0	0	0	0	1	1	1	0	0
0	0	0	1	0	0	0	1	1
0	0	0	1	0	1	0	1	0
0	0	1	X	X	X	0	0	0
0	1	0	X	X	X	0	0	1
0	1	1	X	X	X	0	1	0
1	0	0	X	X	X	0	1	1
1	0	1	X	X	X	1	0	1
1	1	0	X	X	X	1	0	1
1	1	1	X	X	X	1	1	0





isRtype = aluOp2’aluOp1’aluOp0’

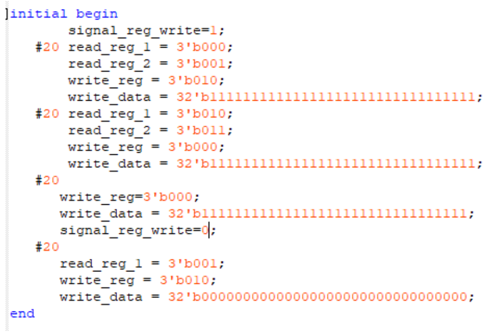
aluCtr2 = isRtype&func1 + aluOp2&(a1 + a0)

aluCtr1 = isRtype&func2 + aluOp2&aluOp1’&aluOp0’ + aluOp1aluOp0

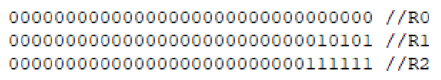
aluCtr0 = isRtype&func0’ + aluOp1&aluOp0’ + aluOp2aluOp1’

Testbenches

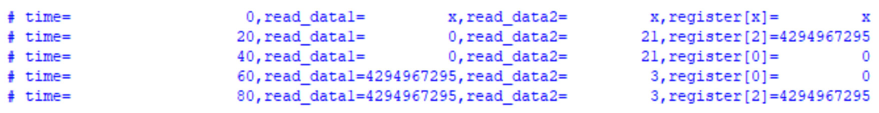
Registers



```
initial begin
        signal_reg_write=1;
    #20 read_reg_1 = 3'b000;
        read_reg_2 = 3'b001;
        write_reg = 3'b010;
        write_data = 32'b11111111111111111111111111111111;
    #20 read_reg_1 = 3'b010;
        read_reg_2 = 3'b011;
        write_reg = 3'b000;
        write_data = 32'b11111111111111111111111111111111;
    #20
       write_reg=3'b000;
       write_data = 32'b11111111111111111111111111111111;
       signal_reg_write=0;
    #20
       read_reg_1 = 3'b001;
       write_reg = 3'b010;
       write_data = 32'b00000000000000000000000000000000;
end
```



```
00000000000000000000000000000000 //R0
00000000000000000000000000010101 //R1
00000000000000000000000000111111 //R2
```



```
# time=                   0,read_data1=         x,read_data2=         x,register[x]=         x
# time=                  20,read_data1=         0,read_data2=        21,register[2]=4294967295
# time=                  40,read_data1=         0,read_data2=        21,register[0]=         0
# time=                  60,read_data1=4294967295,read_data2=         3,register[0]=         0
# time=                  80,read_data1=4294967295,read_data2=         3,register[2]=4294967295
```

At time 0, reg_write is set to 1

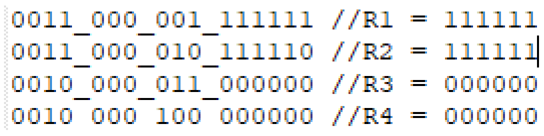
At time 20, registers are read correctly and 32 bit max-number is successfully writed to register[2] because reg_write is 1.

At time 40, registers are read correctly and 32 bit max-number couldn’t writed to register[0] because reg[0] can’t be changed.

At time 60, 32 bit max-number couldn’t writed to register[0] because reg[0] can’t be changed and reg_write is set to 0.

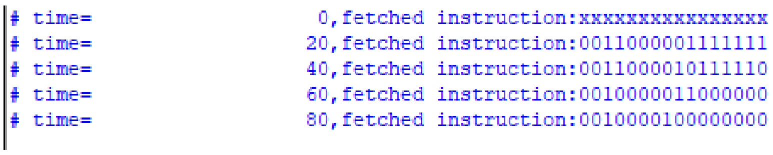
At time 80, 0 couldn’t writed to register[2] because reg_write is 0.

Instruction Memory



```
0011_000_001_111111 //R1 = 111111
0011_000_010_111110 //R2 = 111111
0010_000_011_000000 //R3 = 000000
0010_000_100_000000 //R4 = 000000
```

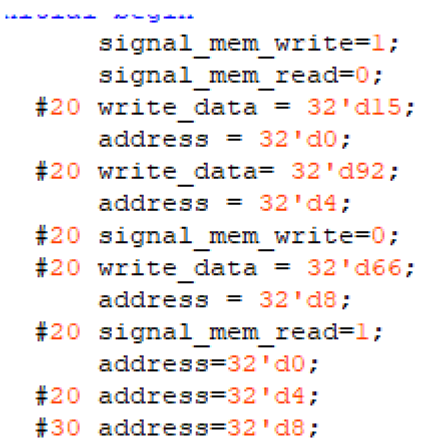
data.txt



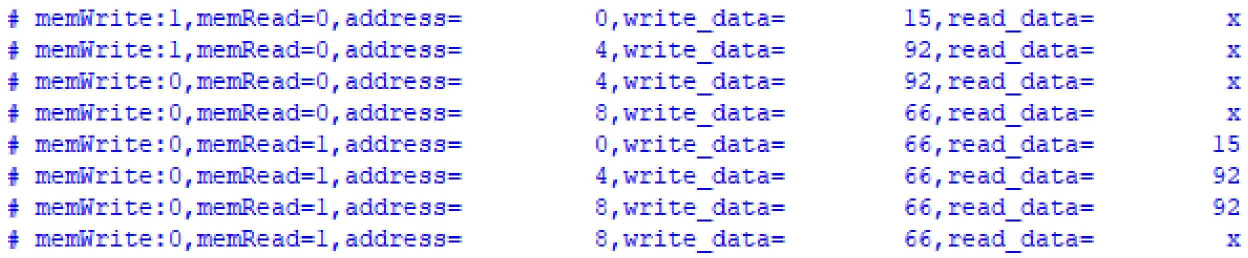
```
# time=                    0,fetched instruction:xxxxxxxxxxxxxxxx
# time=                   20,fetched instruction:0011000001111111
# time=                   40,fetched instruction:0011000010111110
# time=                   60,fetched instruction:0010000011000000
# time=                   80,fetched instruction:0010000100000000
```

testbench

Data Memory



```
      signal_mem_write=1;
      signal_mem_read=0;
  #20 write_data = 32'd15;
      address = 32'd0;
  #20 write_data= 32'd92;
      address = 32'd4;
  #20 signal_mem_write=0;
  #20 write_data = 32'd66;
      address = 32'd8;
  #20 signal_mem_read=1;
      address=32'd0;
  #20 address=32'd4;
  #30 address=32'd8;
```



```
# memWrite:1,memRead=0,address=          0,write_data=         15,read_data=          x
# memWrite:1,memRead=0,address=          4,write_data=         92,read_data=          x
# memWrite:0,memRead=0,address=          4,write_data=         92,read_data=          x
# memWrite:0,memRead=0,address=          8,write_data=         66,read_data=          x
# memWrite:0,memRead=1,address=          0,write_data=         66,read_data=         15
# memWrite:0,memRead=1,address=          4,write_data=         66,read_data=         92
# memWrite:0,memRead=1,address=          8,write_data=         66,read_data=         92
# memWrite:0,memRead=1,address=          8,write_data=         66,read_data=          x
```

-Step1: 15 is successfully written to mem[0] because memWrite is 1. No data read at this step because memRead = 0

-Step2: 92 is successfully written to mem[4] because memWrite is 1. No data read at this step because memRead = 0

-Step3: 66 is not written to mem[8] because memWrite is 0. No data read at this step because memRead = 0

-Step4: 15 is successfully readed from mem[0] because memRead is 1. No data is writed at this step because memWrite = 0

-Step5: 92 is successfully readed from mem[4] because memRead is 1. No data is writed at this step because memWrite = 0

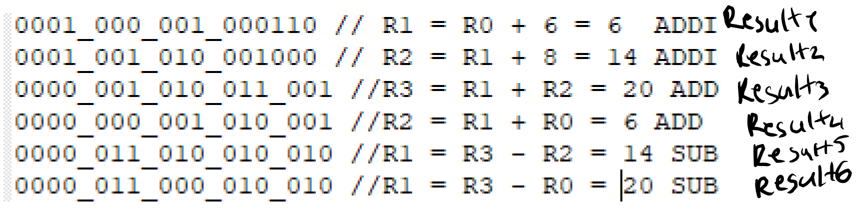
Last step: Although memRead is 1, nothing is readed from mem[8] because nothing is writed (check step 3).

Processor

Important Note: **For every program counter value, the last result is the correct one. Values before the correct result are dummy outputs. This is probably because correct value is determined at negedge and I couldn't find a correct prog_ctr and clk combination. So dummy means value is not calculated yet. I couldn't fix it without breaking the working structure, so I left it as it is.**

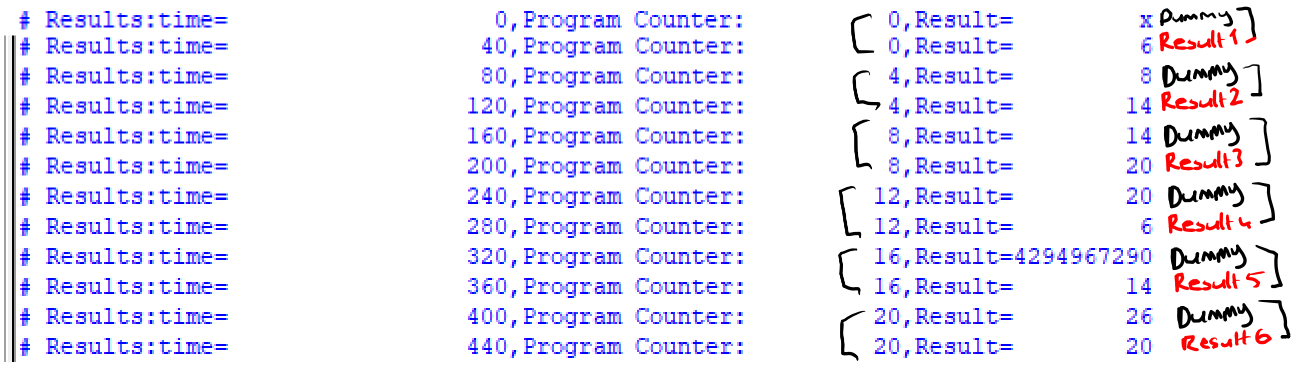
Arithmetic Operations

I printed the outputs as decimal values in this chapter for convenience.



```
0001_000_001_000110 // R1 = R0 + 6 = 6   ADDI Result1
0001_001_010_001000 // R2 = R1 + 8 = 14 ADDI Result2
0000_001_010_011_001 //R3 = R1 + R2 = 20 ADD Result3
0000_000_001_010_001 //R2 = R1 + R0 = 6 ADD  Result4
0000_011_010_010_010 //R1 = R3 - R2 = 14 SUB Result5
0000_011_000_010_010 //R1 = R3 - R0 = 20 SUB Result6
```

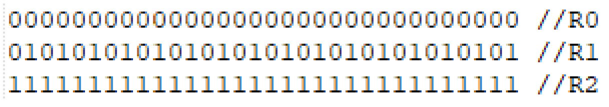
data.txt



```
# Results:time=          0,Program Counter:   0,Result=          x Dummy
# Results:time=         40,Program Counter:   0,Result=          6 Result1
# Results:time=         80,Program Counter:   4,Result=          8 Dummy
# Results:time=        120,Program Counter:   4,Result=         14 Result2
# Results:time=        160,Program Counter:   8,Result=         14 Dummy
# Results:time=        200,Program Counter:   8,Result=         20 Result3
# Results:time=        240,Program Counter:  12,Result=         20 Dummy
# Results:time=        280,Program Counter:  12,Result=          6 Result4
# Results:time=        320,Program Counter:  16,Result=4294967290 Dummy
# Results:time=        360,Program Counter:  16,Result=         14 Result5
# Results:time=        400,Program Counter:  20,Result=         26 Dummy
# Results:time=        440,Program Counter:  20,Result=         20 Result6
```

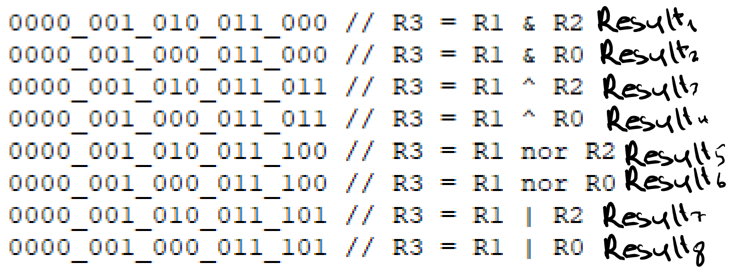
Logic Operations

R Type

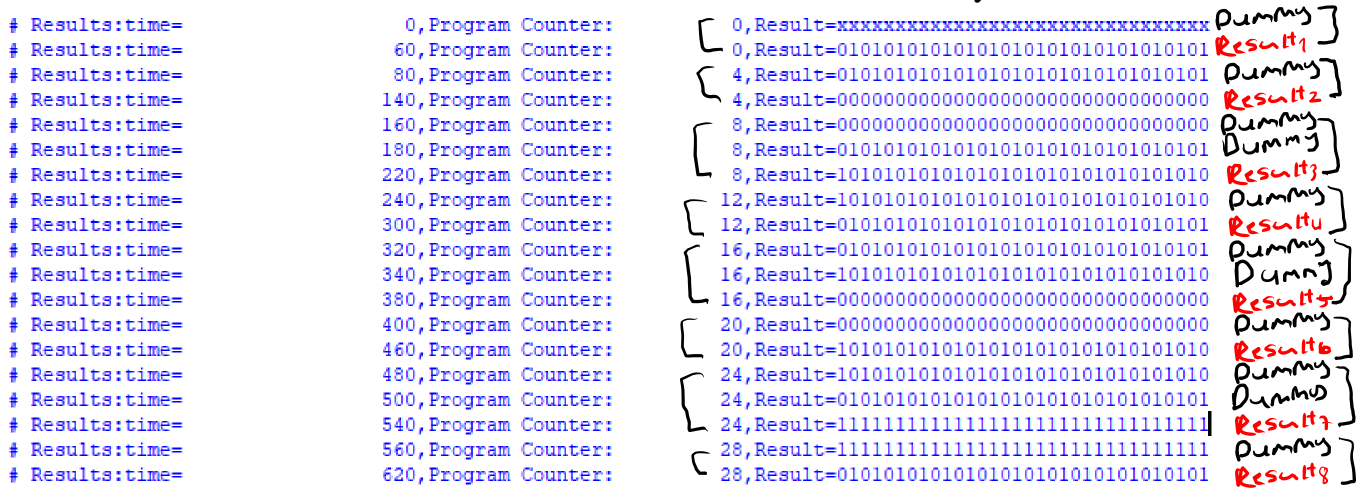


```
00000000000000000000000000000000 //R0
01010101010101010101010101010101 //R1
11111111111111111111111111111111 //R2
```

Initial Values of registers

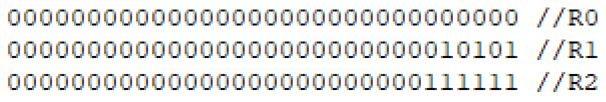


```
0000_001_010_011_000 // R3 = R1 & R2 Result1
0000_001_000_011_000 // R3 = R1 & R0 Result2
0000_001_010_011_011 // R3 = R1 ^ R2 Result3
0000_001_000_011_011 // R3 = R1 ^ R0 Result4
0000_001_010_011_100 // R3 = R1 nor R2 Result5
0000_001_000_011_100 // R3 = R1 nor R0 Result6
0000_001_010_011_101 // R3 = R1 | R2 Result7
0000_001_000_011_101 // R3 = R1 | R0 Result8
```



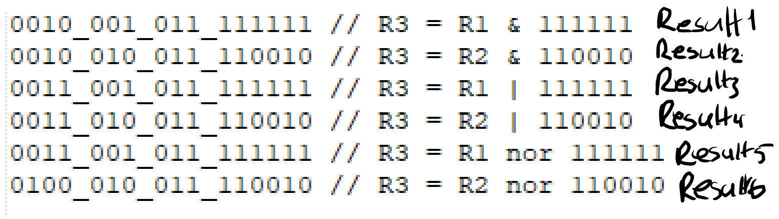
```
# Results:time=          0,Program Counter:   0,Result=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Dummy
# Results:time=         60,Program Counter:   0,Result=01010101010101010101010101010101 Result1
# Results:time=         80,Program Counter:   4,Result=01010101010101010101010101010101 Dummy
# Results:time=        140,Program Counter:   4,Result=00000000000000000000000000000000 Result2
# Results:time=        160,Program Counter:   8,Result=00000000000000000000000000000000 Dummy
# Results:time=        180,Program Counter:   8,Result=01010101010101010101010101010101 Dummy
# Results:time=        220,Program Counter:   8,Result=10101010101010101010101010101010 Result3
# Results:time=        240,Program Counter:  12,Result=10101010101010101010101010101010 Dummy
# Results:time=        300,Program Counter:  12,Result=01010101010101010101010101010101 Result4
# Results:time=        320,Program Counter:  16,Result=01010101010101010101010101010101 Dummy
# Results:time=        340,Program Counter:  16,Result=10101010101010101010101010101010 Dummy
# Results:time=        380,Program Counter:  16,Result=00000000000000000000000000000000 Result5
# Results:time=        400,Program Counter:  20,Result=00000000000000000000000000000000 Dummy
# Results:time=        460,Program Counter:  20,Result=10101010101010101010101010101010 Result6
# Results:time=        480,Program Counter:  24,Result=10101010101010101010101010101010 Dummy
# Results:time=        500,Program Counter:  24,Result=01010101010101010101010101010101 Dummy
# Results:time=        540,Program Counter:  24,Result=11111111111111111111111111111111 Result7
# Results:time=        560,Program Counter:  28,Result=11111111111111111111111111111111 Dummy
# Results:time=        620,Program Counter:  28,Result=01010101010101010101010101010101 Result8
```

I type

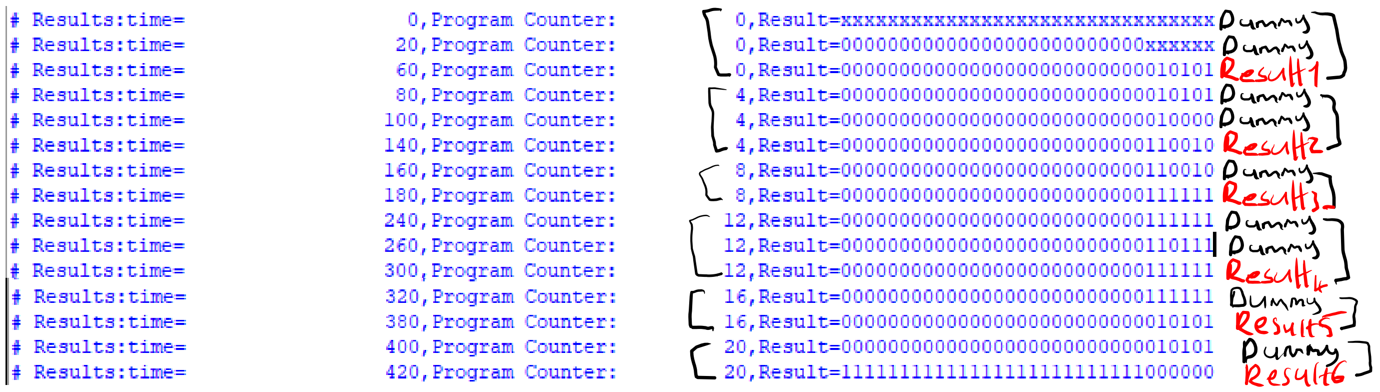


```
00000000000000000000000000000000 //R0
00000000000000000000000000010101 //R1
00000000000000000000000000111111 //R2
```

Initial Values of registers



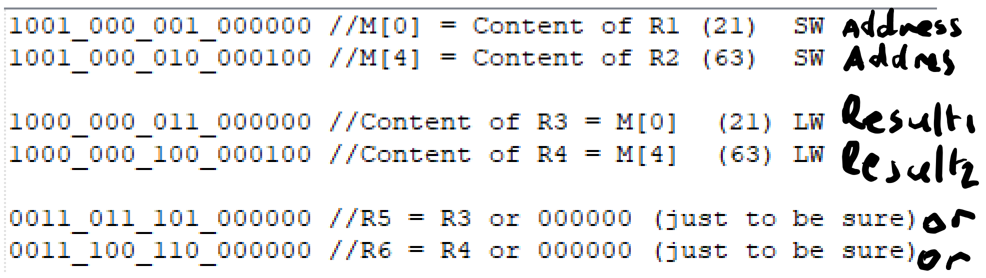
```
0010_001_011_111111 // R3 = R1 & 111111  Result1
0010_010_011_110010 // R3 = R2 & 110010  Result2
0011_001_011_111111 // R3 = R1 | 111111  Result3
0011_010_011_110010 // R3 = R2 | 110010  Result4
0011_001_011_111111 // R3 = R1 nor 111111 Result5
0100_010_011_110010 // R3 = R2 nor 110010 Result6
```



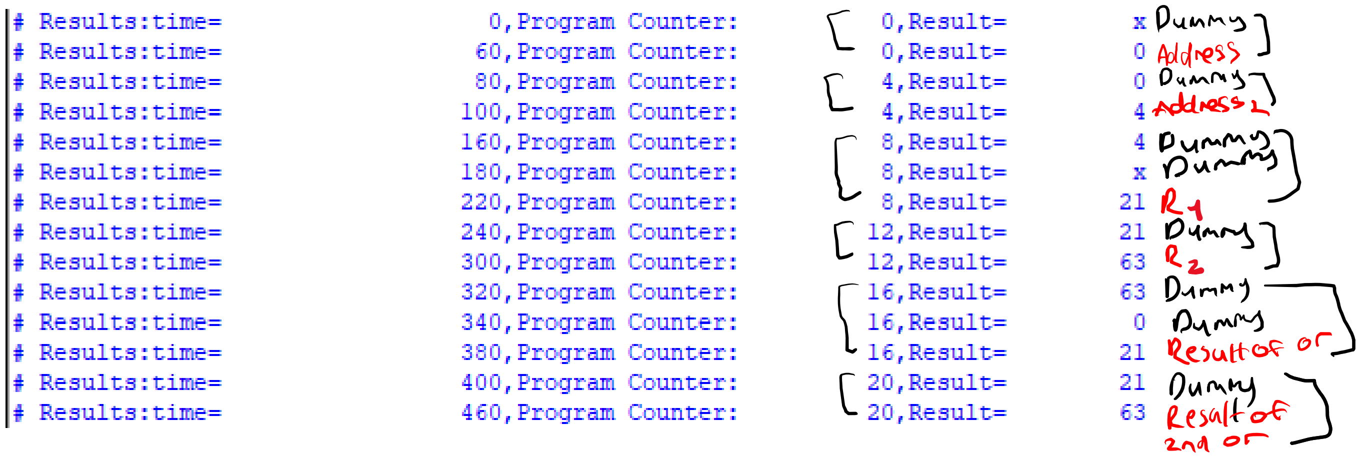
```
# Results:time=                 0,Program Counter:    0,Result=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Dummy
# Results:time=                20,Program Counter:    0,Result=00000000000000000000000000xxxxxx Dummy
# Results:time=                60,Program Counter:    0,Result=00000000000000000000000000010101 Result1
# Results:time=                80,Program Counter:    4,Result=00000000000000000000000000010101 Dummy
# Results:time=               100,Program Counter:    4,Result=00000000000000000000000000010000 Dummy
# Results:time=               140,Program Counter:    4,Result=00000000000000000000000000110010 Result2
# Results:time=               160,Program Counter:    8,Result=00000000000000000000000000110010 Dummy
# Results:time=               180,Program Counter:    8,Result=00000000000000000000000000111111 Result3
# Results:time=               240,Program Counter:   12,Result=00000000000000000000000000111111 Dummy
# Results:time=               260,Program Counter:   12,Result=00000000000000000000000000110111 Dummy
# Results:time=               300,Program Counter:   12,Result=00000000000000000000000000111111 Result4
# Results:time=               320,Program Counter:   16,Result=00000000000000000000000000111111 Dummy
# Results:time=               380,Program Counter:   16,Result=00000000000000000000000000010101 Result5
# Results:time=               400,Program Counter:   20,Result=00000000000000000000000000010101 Dummy
# Results:time=               420,Program Counter:   20,Result=11111111111111111111111111000000 Result6
```

Memory Operations

I printed the outputs as decimal values in this chapter for convenience.

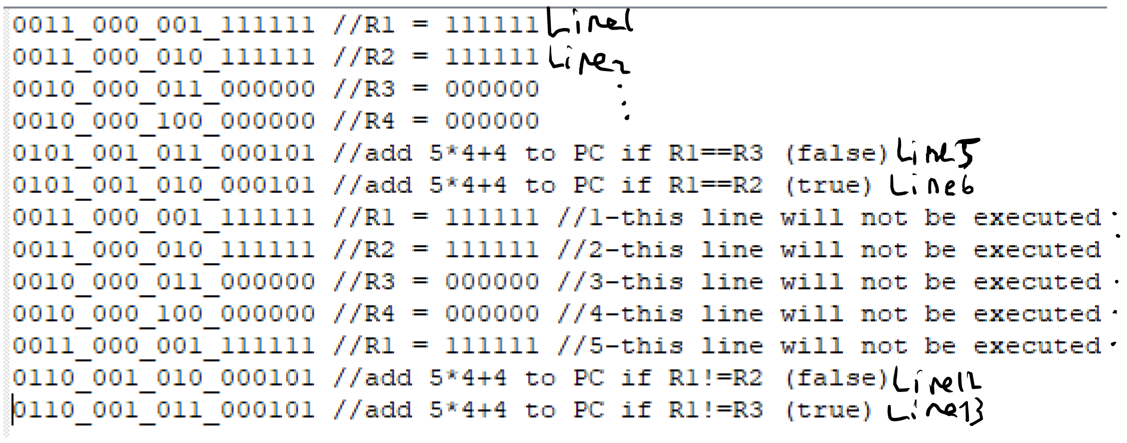


```
1001_000_001_000000 //M[0] = Content of R1 (21)  SW Address
1001_000_010_000100 //M[4] = Content of R2 (63)  SW Address

1000_000_011_000000 //Content of R3 = M[0]  (21) LW Result1
1000_000_100_000100 //Content of R4 = M[4]  (63) LW Result2

0011_011_101_000000 //R5 = R3 or 000000 (just to be sure) or
0011_100_110_000000 //R6 = R4 or 000000 (just to be sure) or
```

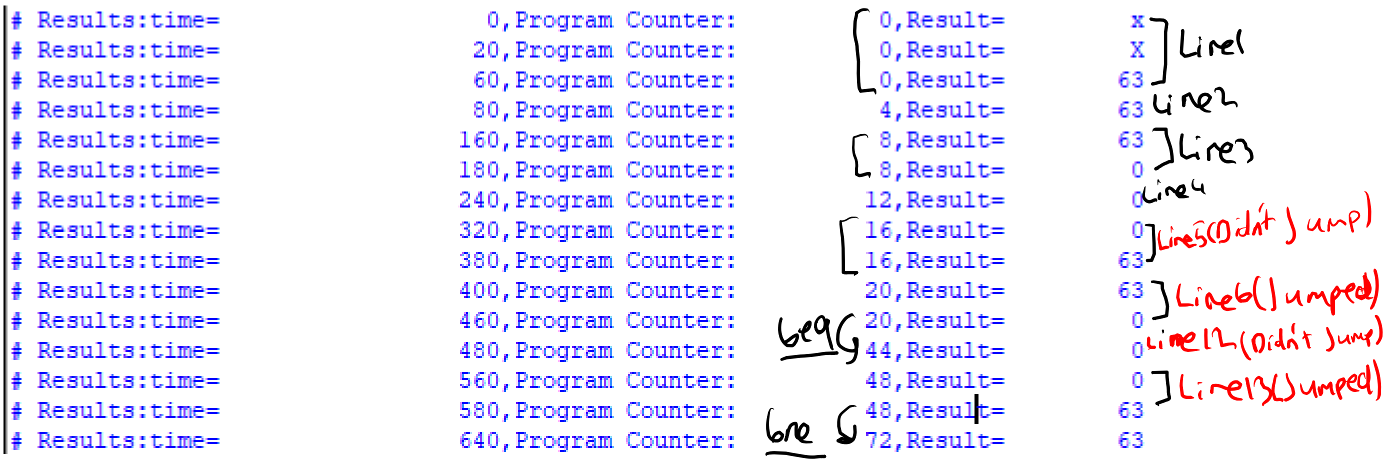


```
# Results:time=                 0,Program Counter:    0,Result=          x Dummy
# Results:time=                60,Program Counter:    0,Result=          0 Address
# Results:time=                80,Program Counter:    4,Result=          0 Dummy
# Results:time=               100,Program Counter:    4,Result=          4 Address2
# Results:time=               160,Program Counter:    8,Result=          4 Dummy
# Results:time=               180,Program Counter:    8,Result=          x Dummy
# Results:time=               220,Program Counter:    8,Result=         21 R1
# Results:time=               240,Program Counter:   12,Result=         21 Dummy
# Results:time=               300,Program Counter:   12,Result=         63 R2
# Results:time=               320,Program Counter:   16,Result=         63 Dummy
# Results:time=               340,Program Counter:   16,Result=          0 Dummy
# Results:time=               380,Program Counter:   16,Result=         21 Result of or
# Results:time=               400,Program Counter:   20,Result=         21 Dummy
# Results:time=               460,Program Counter:   20,Result=         63 Result of 2nd or
```

Branch Operations



```
0011_000_001_111111 //R1 = 111111 Line1
0011_000_010_111111 //R2 = 111111 Line2
0010_000_011_000000 //R3 = 000000   :
0010_000_100_000000 //R4 = 000000   :
0101_001_011_000101 //add 5*4+4 to PC if R1==R3 (false) Line5
0101_001_010_000101 //add 5*4+4 to PC if R1==R2 (true) Line6
0011_000_001_111111 //R1 = 111111 //1-this line will not be executed.
0011_000_010_111111 //R2 = 111111 //2-this line will not be executed.
0010_000_011_000000 //R3 = 000000 //3-this line will not be executed.
0010_000_100_000000 //R4 = 000000 //4-this line will not be executed.
0011_000_001_111111 //R1 = 111111 //5-this line will not be executed.
0110_001_010_000101 //add 5*4+4 to PC if R1!=R2 (false) Line12
0110_001_011_000101 //add 5*4+4 to PC if R1!=R3 (true) Line13
```



```
# Results:time=          0,Program Counter:       0,Result=        x ] Line1
# Results:time=         20,Program Counter:       0,Result=        X ]
# Results:time=         60,Program Counter:       0,Result=       63 ]
# Results:time=         80,Program Counter:       4,Result=       63 Line2
# Results:time=        160,Program Counter:       8,Result=       63 ] Line3
# Results:time=        180,Program Counter:       8,Result=        0 ]
# Results:time=        240,Program Counter:      12,Result=        0 Line4
# Results:time=        320,Program Counter:      16,Result=        0 ] Line5(Didn't Jump)
# Results:time=        380,Program Counter:      16,Result=       63 ]
# Results:time=        400,Program Counter:      20,Result=       63 ] Line6(Jumped)
# Results:time=        460,Program Counter:  beq  20,Result=        0 ]
# Results:time=        480,Program Counter:       44,Result=        0 Line12(Didn't Jump)
# Results:time=        560,Program Counter:       48,Result=        0 ] Line13(Jumped)
# Results:time=        580,Program Counter:       48,Result=       63 ]
# Results:time=        640,Program Counter:  bne  72,Result=       63
```