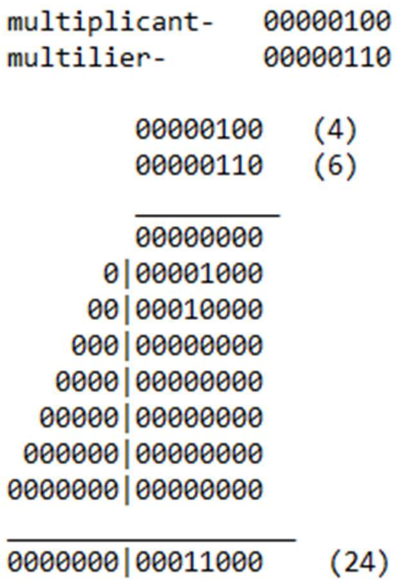
CO224 Lab05_part5
Group11

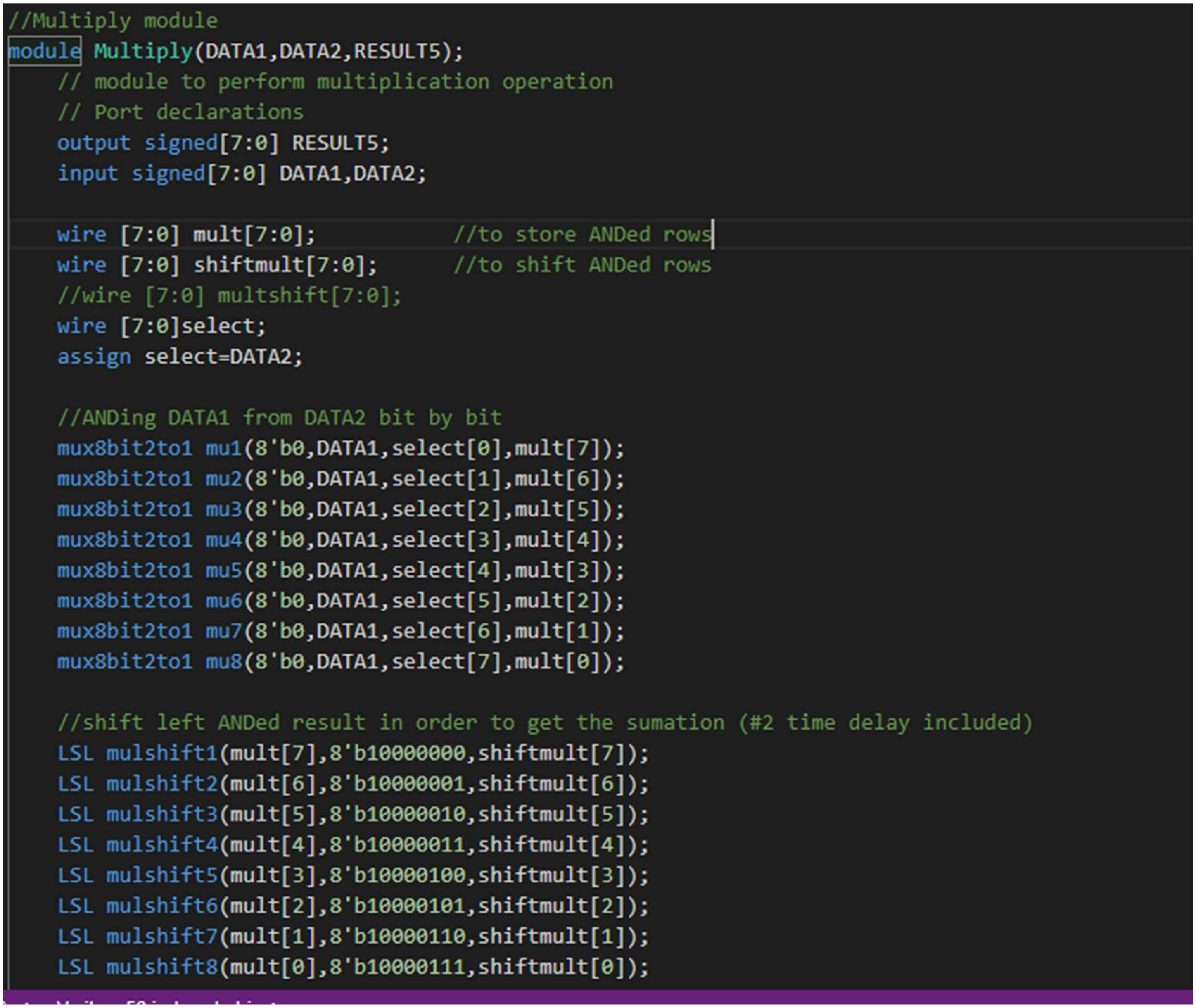
1) MULTIPLICATION

mult 4 1 2 (multiply value in register 1 by value in register 2, and place the result in register 4)



```
multiplicant-   00000100
multilier-      00000110

        00000100   (4)
        00000110   (6)
        _________
        00000000
       0|00001000
      00|00010000
     000|00000000
    0000|00000000
   00000|00000000
  000000|00000000
 0000000|00000000
 __________________
 0000000|00011000    (24)
```

Multiplicand is DATA1 and multiplier is DATA2. DATA1 is ANDed from bits in DATA2 respectively and got the sum. For ANDing it is easier to use a mux where mux output DATA1 as it is when DATA2[i] is equal to one and
Output zero when DATA[i] is equal to zero.

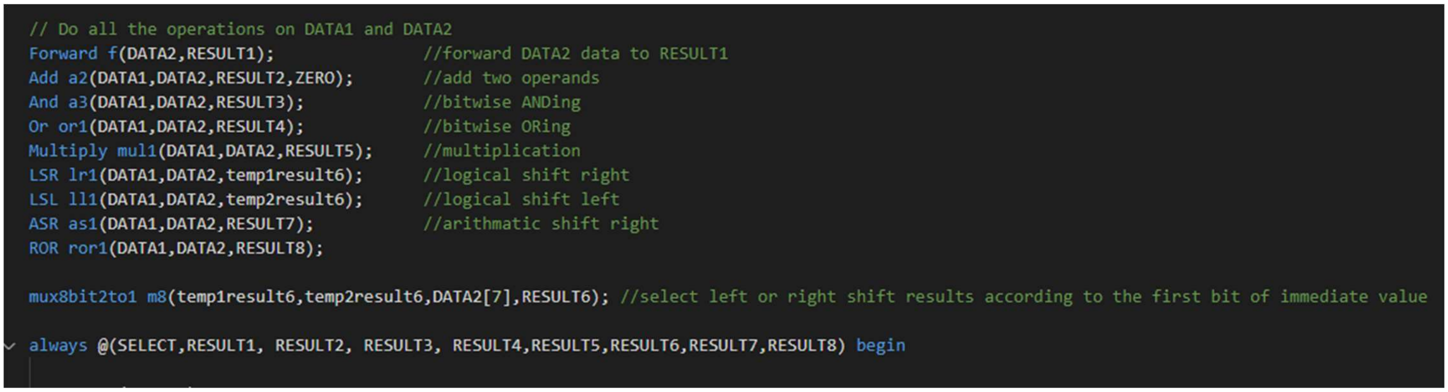


```
//Multiply module
module Multiply(DATA1,DATA2,RESULT5);
    // module to perform multiplication operation
    // Port declarations
    output signed[7:0] RESULT5;
    input signed[7:0] DATA1,DATA2;

    wire [7:0] mult[7:0];           //to store ANDed rows
    wire [7:0] shiftmult[7:0];      //to shift ANDed rows
    //wire [7:0] multshift[7:0];
    wire [7:0]select;
    assign select=DATA2;

    //ANDing DATA1 from DATA2 bit by bit
    mux8bit2to1 mu1(8'b0,DATA1,select[0],mult[7]);
    mux8bit2to1 mu2(8'b0,DATA1,select[1],mult[6]);
    mux8bit2to1 mu3(8'b0,DATA1,select[2],mult[5]);
    mux8bit2to1 mu4(8'b0,DATA1,select[3],mult[4]);
    mux8bit2to1 mu5(8'b0,DATA1,select[4],mult[3]);
    mux8bit2to1 mu6(8'b0,DATA1,select[5],mult[2]);
    mux8bit2to1 mu7(8'b0,DATA1,select[6],mult[1]);
    mux8bit2to1 mu8(8'b0,DATA1,select[7],mult[0]);

    //shift left ANDed result in order to get the sumation (#2 time delay included)
    LSL mulshift1(mult[7],8'b10000000,shiftmult[7]);
    LSL mulshift2(mult[6],8'b10000001,shiftmult[6]);
    LSL mulshift3(mult[5],8'b10000010,shiftmult[5]);
    LSL mulshift4(mult[4],8'b10000011,shiftmult[4]);
    LSL mulshift5(mult[3],8'b10000100,shiftmult[3]);
    LSL mulshift6(mult[2],8'b10000101,shiftmult[2]);
    LSL mulshift7(mult[1],8'b10000110,shiftmult[1]);
    LSL mulshift8(mult[0],8'b10000111,shiftmult[0]);
```

2)SHIFT LOGICALY LEFT

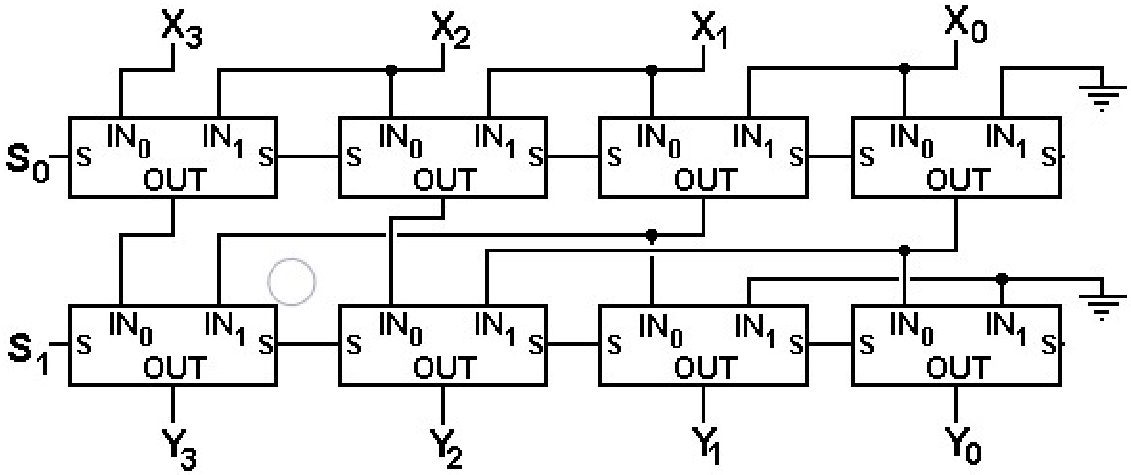
sll 4 1 0x02 (apply logical shift left 2 times on value in register 1, and place the result in register 4)

As we can only have 8 different ALUOPs we set first IMMEDIATE value to 1 for logically left shift. And take same ALUOP for logical shift left and logical shift right. In ALU module we set which operation to do according to the first bit of the immediate value.

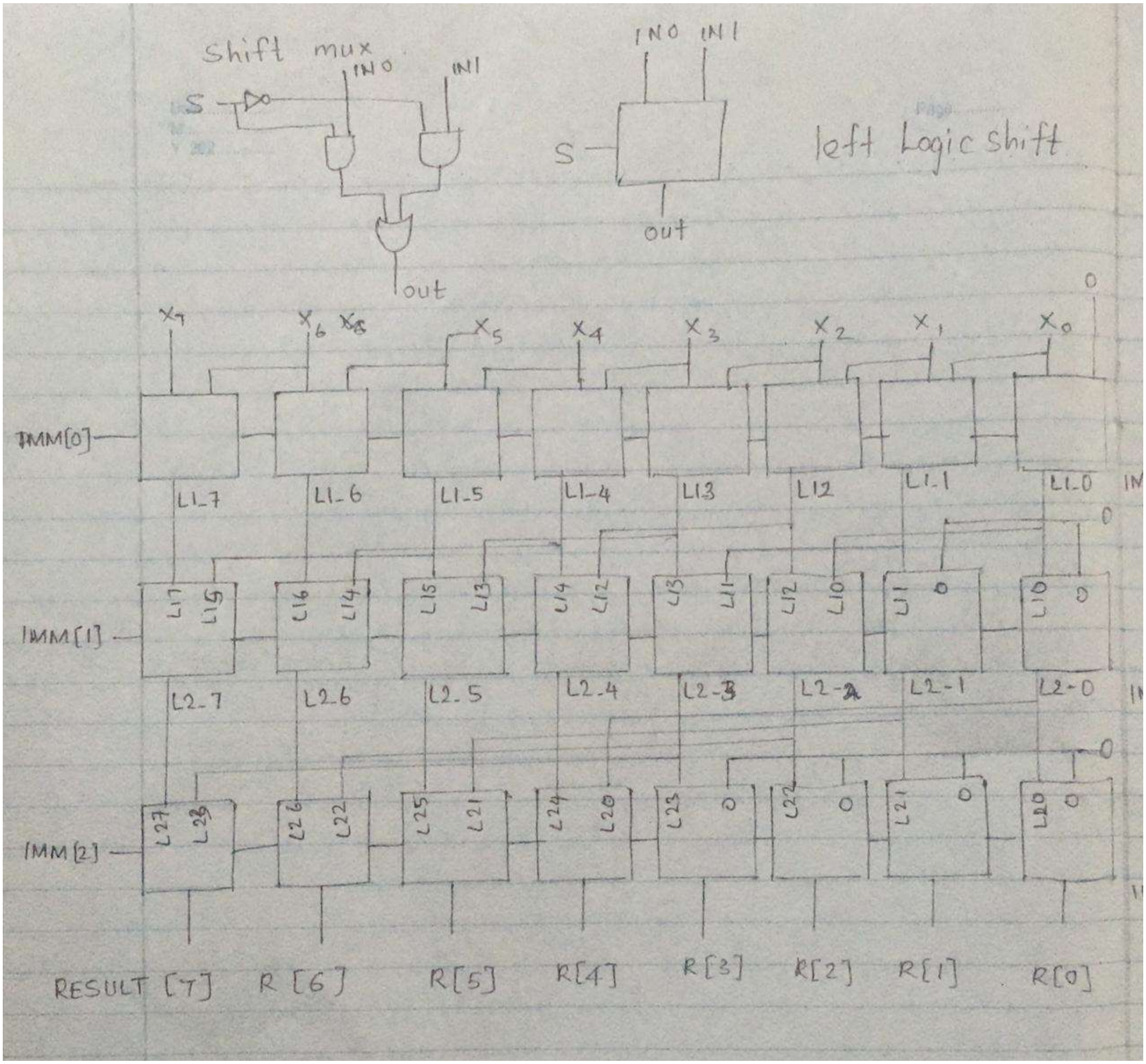


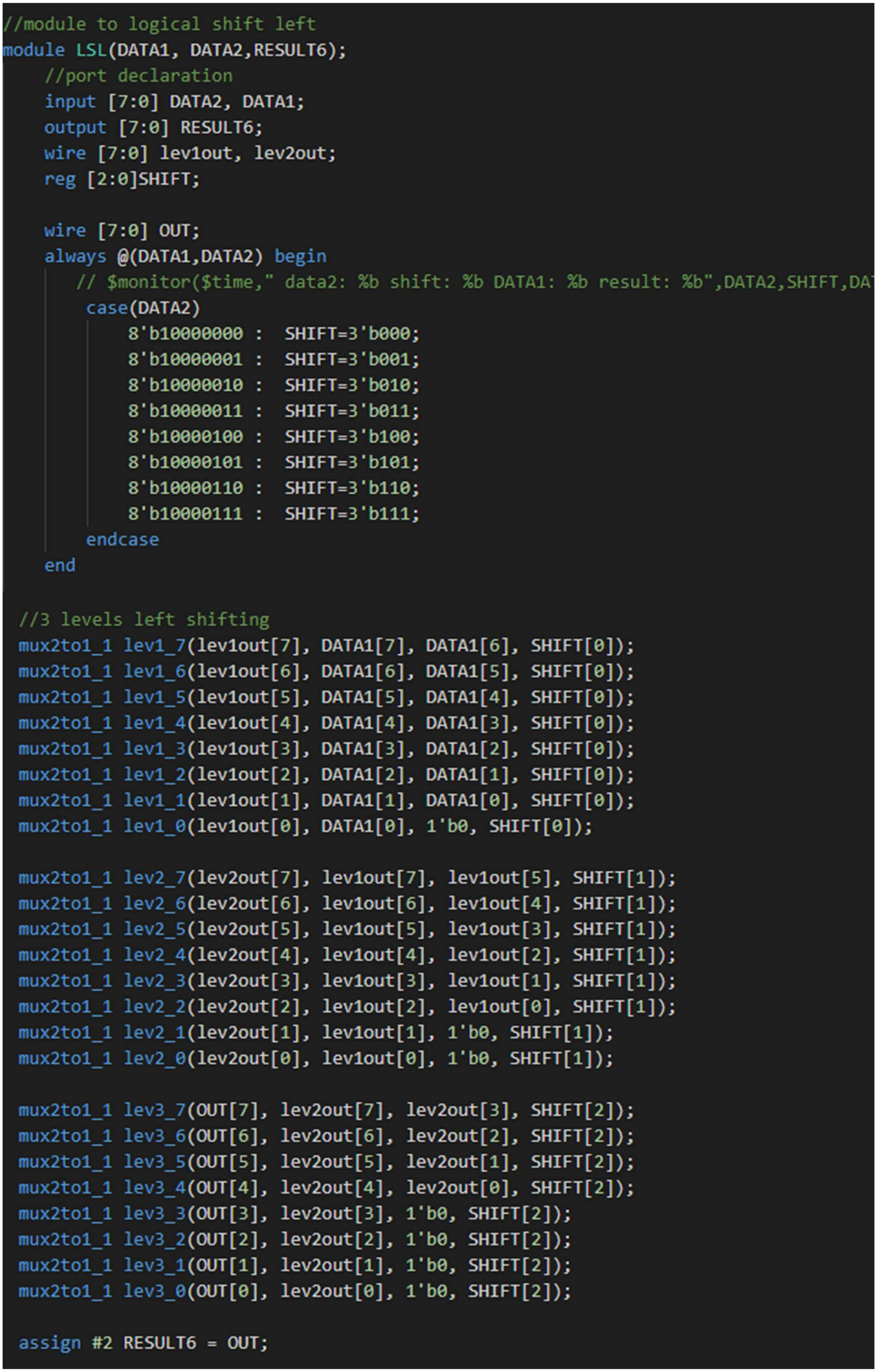
```
// Do all the operations on DATA1 and DATA2
Forward f(DATA2,RESULT1);                 //forward DATA2 data to RESULT1
Add a2(DATA1,DATA2,RESULT2,ZERO);         //add two operands
And a3(DATA1,DATA2,RESULT3);              //bitwise ANDing
Or or1(DATA1,DATA2,RESULT4);              //bitwise ORing
Multiply mul1(DATA1,DATA2,RESULT5);       //multiplication
LSR lr1(DATA1,DATA2,temp1result6);        //logical shift right
LSL ll1(DATA1,DATA2,temp2result6);        //logical shift left
ASR as1(DATA1,DATA2,RESULT7);             //arithmatic shift right
ROR ror1(DATA1,DATA2,RESULT8);

mux8bit2to1 m8(temp1result6,temp2result6,DATA2[7],RESULT6); //select left or right shift results according to the first bit of immediate value

always @(SELECT,RESULT1, RESULT2, RESULT3, RESULT4,RESULT5,RESULT6,RESULT7,RESULT8) begin
```

The following diagram shows the left shifter. This 2 levels module can shift up to   3 bits. We add another level such that we can shift up to 7 bits.



Signals		Action
$S_1 = 0$	$S_0 = 0$	No shift
$S_1 = 0$	$S_0 = 1$	Left shift one place
$S_1 = 1$	$S_0 = 0$	Left shift two places
$S_1 = 1$	$S_0 = 1$	Left shift three places


Shift mux
S
IN0
IN1
out
IN0 IN1
S
out
left Logic shift
0
X_7
X_6 ~~X_6~~
X_5
X_4
X_3
X_2
X_1
X_0
IMM[0]
L1_7
L1_6
L1_5
L1_4
L13
L12
L1.1
L1_0
0
L17 L15
L16 L14
L15 L13
L14 L12
L13 L11
L12 L10
L11 0
L10 0
IMM[1]
L2_7
L2.6
L2.5
L2.4
L2-3
L2-2
L2-1
L2-0
0
0
L27 L23
L26 L22
L25 L21
L24 L20
L23 0
L22 0
L21 0
L20 0
IMM[2]
RESULT [7]
R [6]
R[5]
R[4]
R[3]
R[2]
R[1]
R[0]



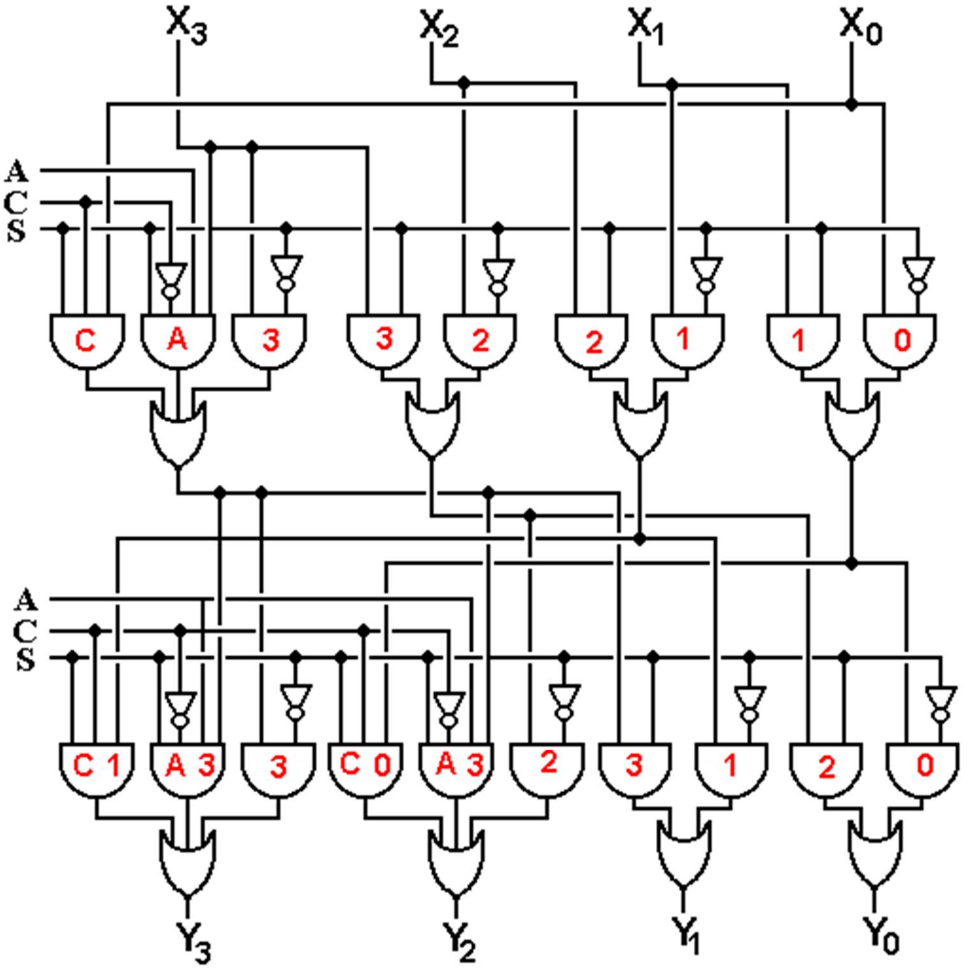
```
//module to logical shift left
module LSL(DATA1, DATA2,RESULT6);
    //port declaration
    input [7:0] DATA2, DATA1;
    output [7:0] RESULT6;
    wire [7:0] lev1out, lev2out;
    reg [2:0]SHIFT;

    wire [7:0] OUT;
    always @(DATA1,DATA2) begin
       // $monitor($time," data2: %b shift: %b DATA1: %b result: %b",DATA2,SHIFT,DA
        case(DATA2)
            8'b10000000 :  SHIFT=3'b000;
            8'b10000001 :  SHIFT=3'b001;
            8'b10000010 :  SHIFT=3'b010;
            8'b10000011 :  SHIFT=3'b011;
            8'b10000100 :  SHIFT=3'b100;
            8'b10000101 :  SHIFT=3'b101;
            8'b10000110 :  SHIFT=3'b110;
            8'b10000111 :  SHIFT=3'b111;
        endcase
    end

  //3 levels left shifting
  mux2to1_1 lev1_7(lev1out[7], DATA1[7], DATA1[6], SHIFT[0]);
  mux2to1_1 lev1_6(lev1out[6], DATA1[6], DATA1[5], SHIFT[0]);
  mux2to1_1 lev1_5(lev1out[5], DATA1[5], DATA1[4], SHIFT[0]);
  mux2to1_1 lev1_4(lev1out[4], DATA1[4], DATA1[3], SHIFT[0]);
  mux2to1_1 lev1_3(lev1out[3], DATA1[3], DATA1[2], SHIFT[0]);
  mux2to1_1 lev1_2(lev1out[2], DATA1[2], DATA1[1], SHIFT[0]);
  mux2to1_1 lev1_1(lev1out[1], DATA1[1], DATA1[0], SHIFT[0]);
  mux2to1_1 lev1_0(lev1out[0], DATA1[0], 1'b0, SHIFT[0]);

  mux2to1_1 lev2_7(lev2out[7], lev1out[7], lev1out[5], SHIFT[1]);
  mux2to1_1 lev2_6(lev2out[6], lev1out[6], lev1out[4], SHIFT[1]);
  mux2to1_1 lev2_5(lev2out[5], lev1out[5], lev1out[3], SHIFT[1]);
  mux2to1_1 lev2_4(lev2out[4], lev1out[4], lev1out[2], SHIFT[1]);
  mux2to1_1 lev2_3(lev2out[3], lev1out[3], lev1out[1], SHIFT[1]);
  mux2to1_1 lev2_2(lev2out[2], lev1out[2], lev1out[0], SHIFT[1]);
  mux2to1_1 lev2_1(lev2out[1], lev1out[1], 1'b0, SHIFT[1]);
  mux2to1_1 lev2_0(lev2out[0], lev1out[0], 1'b0, SHIFT[1]);

  mux2to1_1 lev3_7(OUT[7], lev2out[7], lev2out[3], SHIFT[2]);
  mux2to1_1 lev3_6(OUT[6], lev2out[6], lev2out[2], SHIFT[2]);
  mux2to1_1 lev3_5(OUT[5], lev2out[5], lev2out[1], SHIFT[2]);
  mux2to1_1 lev3_4(OUT[4], lev2out[4], lev2out[0], SHIFT[2]);
  mux2to1_1 lev3_3(OUT[3], lev2out[3], 1'b0, SHIFT[2]);
  mux2to1_1 lev3_2(OUT[2], lev2out[2], 1'b0, SHIFT[2]);
  mux2to1_1 lev3_1(OUT[1], lev2out[1], 1'b0, SHIFT[2]);
  mux2to1_1 lev3_0(OUT[0], lev2out[0], 1'b0, SHIFT[2]);

  assign #2 RESULT6 = OUT;
```

3) LOGICAL SHIFT RIGHT, ARITHMATIC SHIFT RIGHT, ROTATION

srl 4 1 0x02 (apply logical shift right 2 times on value in register 1, and place the result in register 4)

sra 4 1 0x02 (apply arithmetic shift right 2 times on value in register 1, and place the result in register 4)

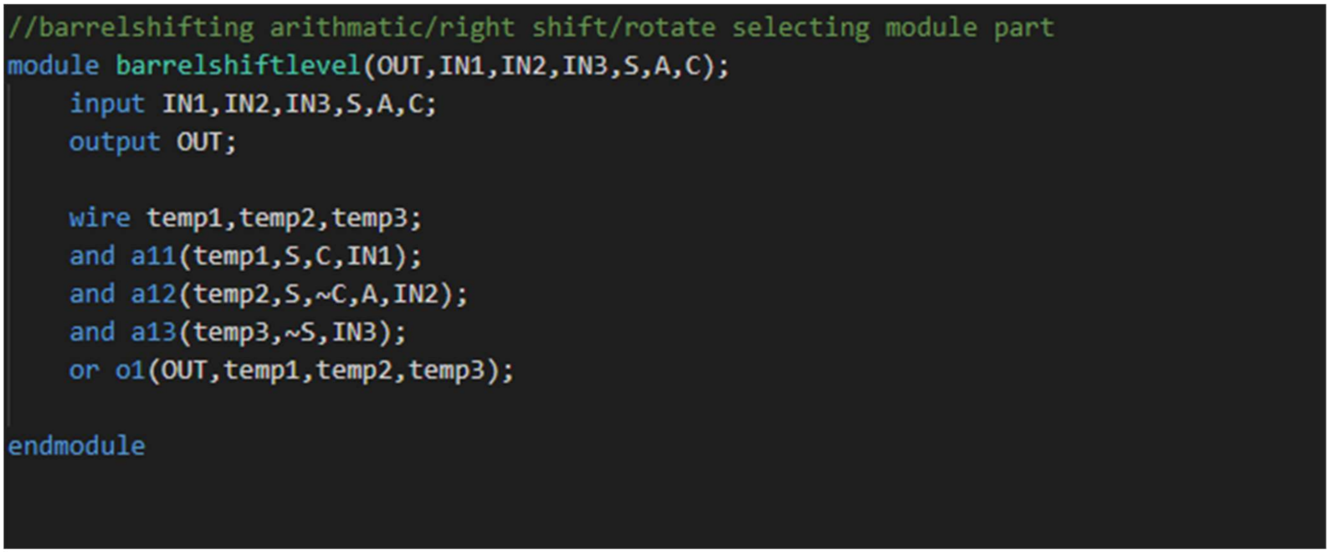
ror 4 1 0x02 (apply rotate right 2 times on value in register 1, and place the result in register 4)



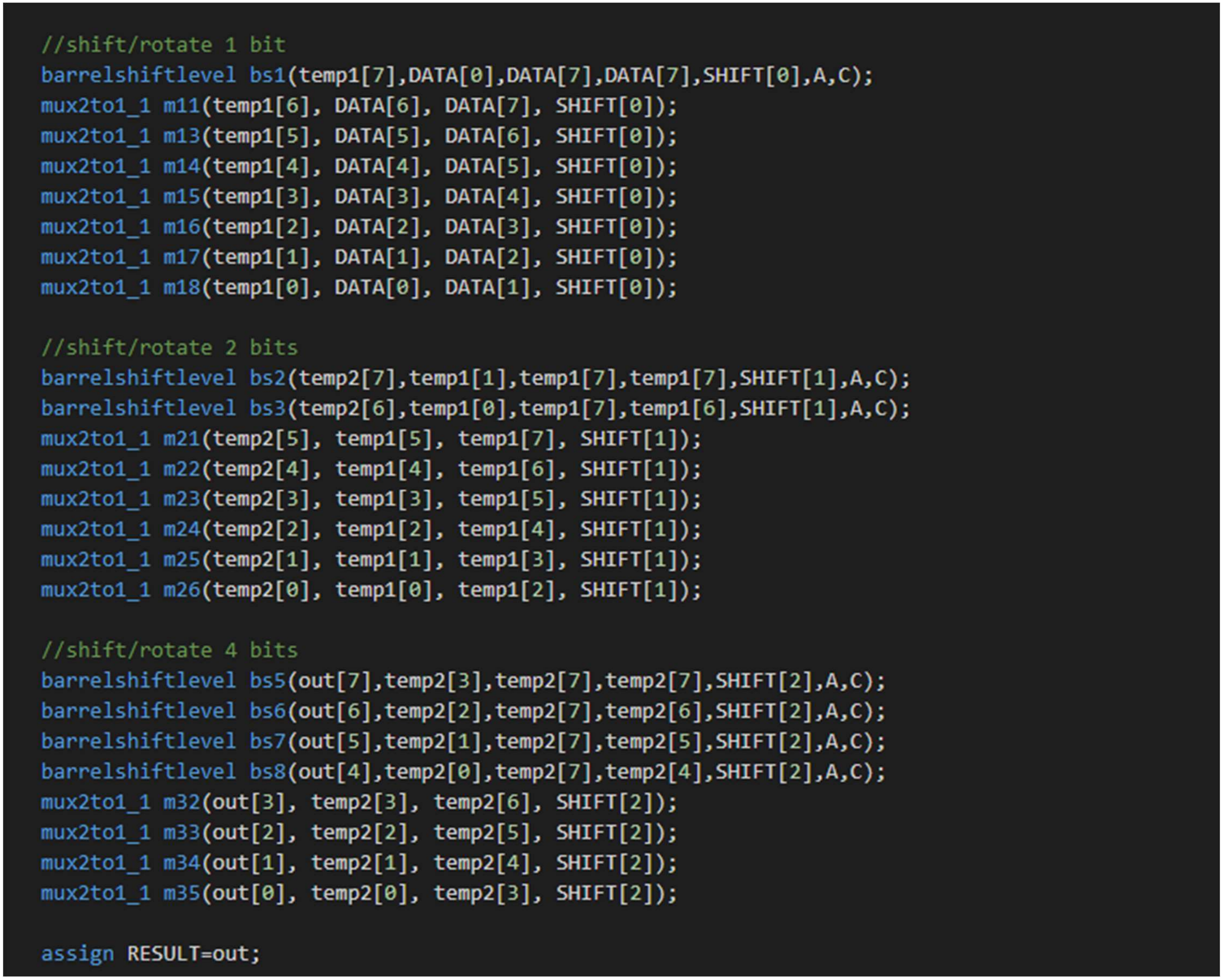
Barrell shift is capable of doing logically right shifting, arithmetically right shifting and rotation

S=0	no shifting
S=1	logical shift right
S=1 A=1	arithmetic shift right
S=1 C=1	rotation

Two levels module can shift up to 3 bits. And 3 levels module can shift up to 7bits. We implement a 3 level Barrell shifter using 2to1 muxes.



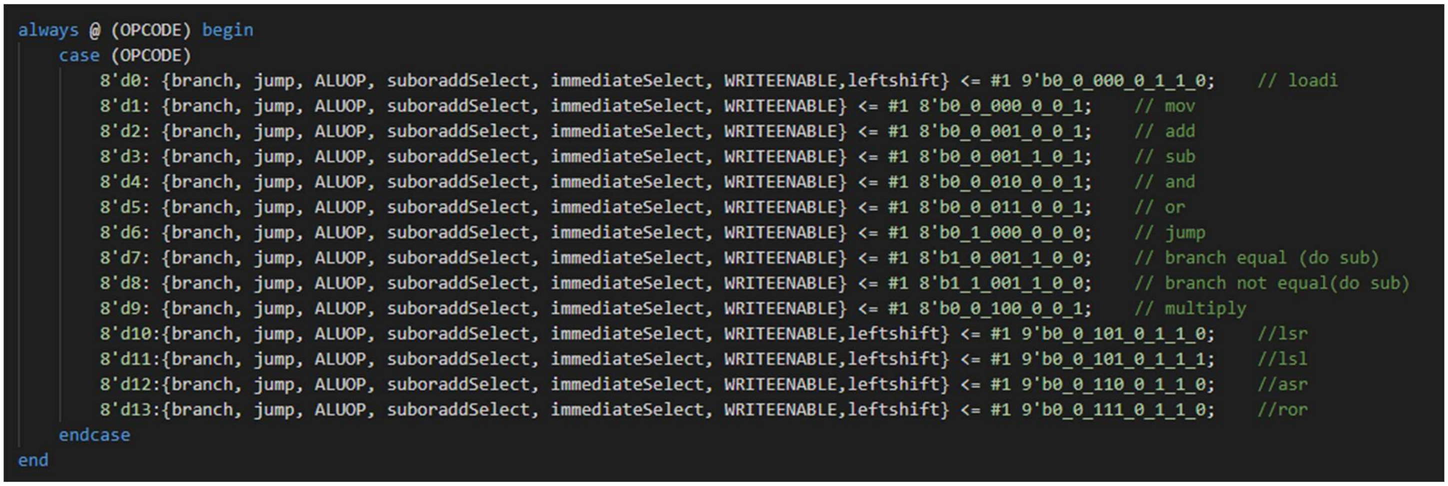
```
//barrelshifting arithmatic/right shift/rotate selecting module part
module barrelshiftlevel(OUT,IN1,IN2,IN3,S,A,C);
    input IN1,IN2,IN3,S,A,C;
    output OUT;

    wire temp1,temp2,temp3;
    and a11(temp1,S,C,IN1);
    and a12(temp2,S,~C,A,IN2);
    and a13(temp3,~S,IN3);
    or o1(OUT,temp1,temp2,temp3);

endmodule
```



```
   //shift/rotate 1 bit
   barrelshiftlevel bs1(temp1[7],DATA[0],DATA[7],DATA[7],SHIFT[0],A,C);
   mux2to1_1 m11(temp1[6], DATA[6], DATA[7], SHIFT[0]);
   mux2to1_1 m13(temp1[5], DATA[5], DATA[6], SHIFT[0]);
   mux2to1_1 m14(temp1[4], DATA[4], DATA[5], SHIFT[0]);
   mux2to1_1 m15(temp1[3], DATA[3], DATA[4], SHIFT[0]);
   mux2to1_1 m16(temp1[2], DATA[2], DATA[3], SHIFT[0]);
   mux2to1_1 m17(temp1[1], DATA[1], DATA[2], SHIFT[0]);
   mux2to1_1 m18(temp1[0], DATA[0], DATA[1], SHIFT[0]);

   //shift/rotate 2 bits
   barrelshiftlevel bs2(temp2[7],temp1[1],temp1[7],temp1[7],SHIFT[1],A,C);
   barrelshiftlevel bs3(temp2[6],temp1[0],temp1[7],temp1[6],SHIFT[1],A,C);
   mux2to1_1 m21(temp2[5], temp1[5], temp1[7], SHIFT[1]);
   mux2to1_1 m22(temp2[4], temp1[4], temp1[6], SHIFT[1]);
   mux2to1_1 m23(temp2[3], temp1[3], temp1[5], SHIFT[1]);
   mux2to1_1 m24(temp2[2], temp1[2], temp1[4], SHIFT[1]);
   mux2to1_1 m25(temp2[1], temp1[1], temp1[3], SHIFT[1]);
   mux2to1_1 m26(temp2[0], temp1[0], temp1[2], SHIFT[1]);

   //shift/rotate 4 bits
   barrelshiftlevel bs5(out[7],temp2[3],temp2[7],temp2[7],SHIFT[2],A,C);
   barrelshiftlevel bs6(out[6],temp2[2],temp2[7],temp2[6],SHIFT[2],A,C);
   barrelshiftlevel bs7(out[5],temp2[1],temp2[7],temp2[5],SHIFT[2],A,C);
   barrelshiftlevel bs8(out[4],temp2[0],temp2[7],temp2[4],SHIFT[2],A,C);
   mux2to1_1 m32(out[3], temp2[3], temp2[6], SHIFT[2]);
   mux2to1_1 m33(out[2], temp2[2], temp2[5], SHIFT[2]);
   mux2to1_1 m34(out[1], temp2[1], temp2[4], SHIFT[2]);
   mux2to1_1 m35(out[0], temp2[0], temp2[3], SHIFT[2]);

   assign RESULT=out;
```

4) BRANCH NOT EQUAL

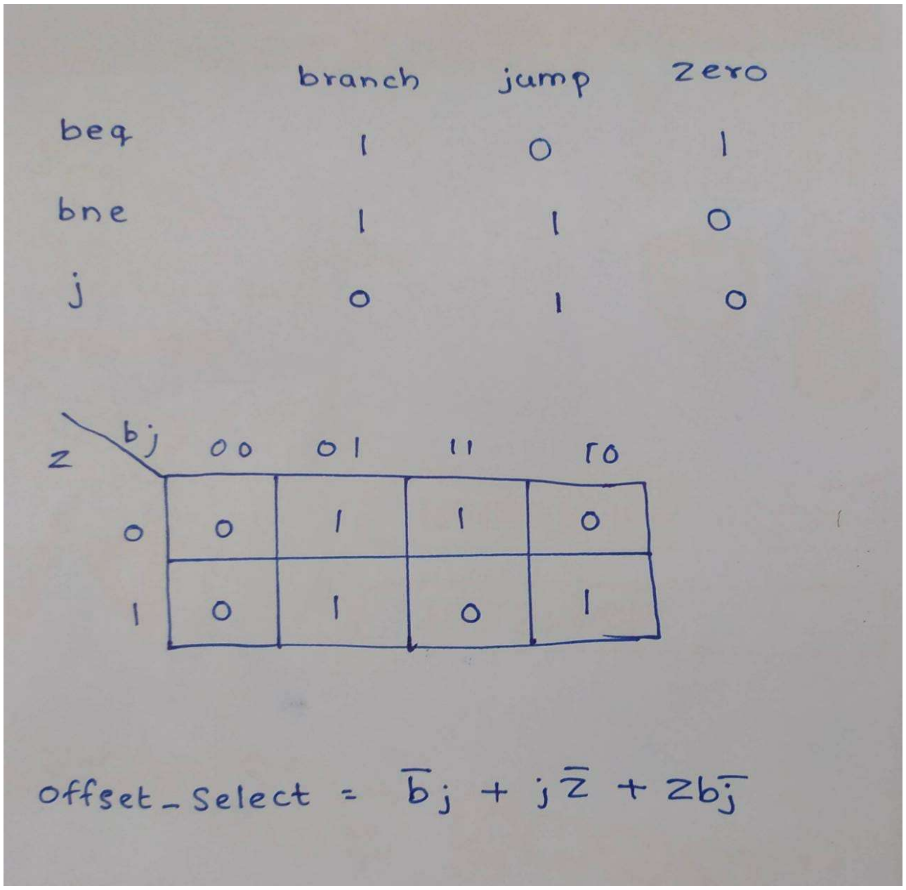
bne 0x02 1 2 (if values in registers 1 and 2 are not equal, branch 2 instructions forward)



```
always @ (OPCODE) begin
    case (OPCODE)
        8'd0: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE,leftshift} <= #1 9'b0_0_000_0_1_1_0;    // loadi
        8'd1: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_0_000_0_0_1;    // mov
        8'd2: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_0_001_0_0_1;    // add
        8'd3: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_0_001_1_0_1;    // sub
        8'd4: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_0_010_0_0_1;    // and
        8'd5: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_0_011_0_0_1;    // or
        8'd6: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_1_000_0_0_0;    // jump
        8'd7: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b1_0_001_1_0_0;    // branch equal (do sub)
        8'd8: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b1_1_001_1_0_0;    // branch not equal(do sub)
        8'd9: {branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE} <= #1 8'b0_0_100_0_0_1;    // multiply
        8'd10:{branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE,leftshift} <= #1 9'b0_0_101_0_1_1_0;    //lsr
        8'd11:{branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE,leftshift} <= #1 9'b0_0_101_0_1_1_1;    //lsl
        8'd12:{branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE,leftshift} <= #1 9'b0_0_110_0_1_1_0;    //asr
        8'd13:{branch, jump, ALUOP, suboraddSelect, immediateSelect, WRITEENABLE,leftshift} <= #1 9'b0_0_111_0_1_1_0;    //ror
    endcase
end
```

For JUMP instruction	j=1	branch=0
For BEQ instruction	j=0	branch=1
For BNE instruction	j=1	branch=1

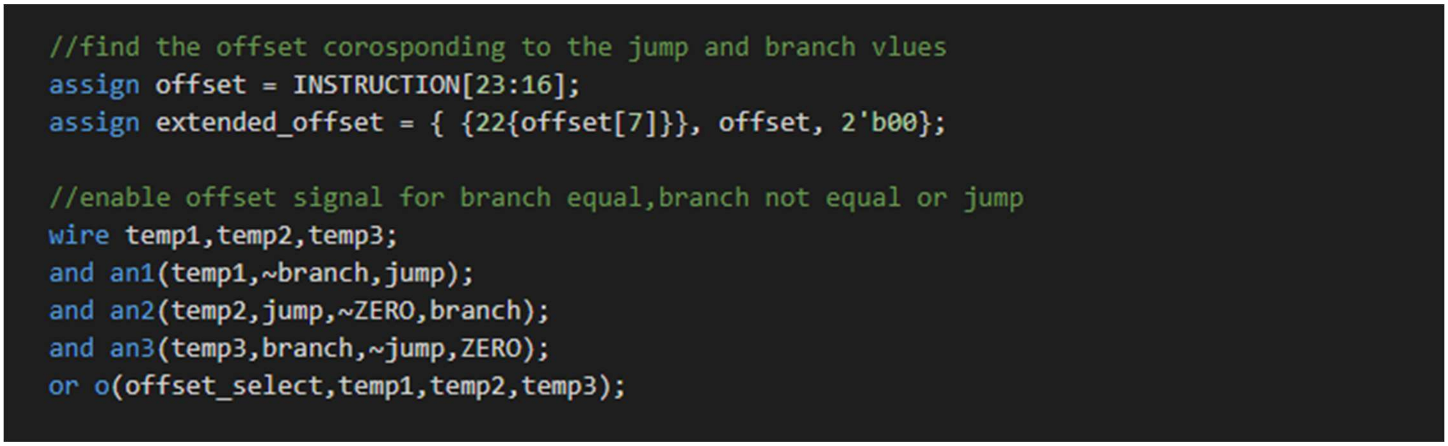
For the above three instructions we have to get the offset. We set offset_select to 1 when we have to take the offset.



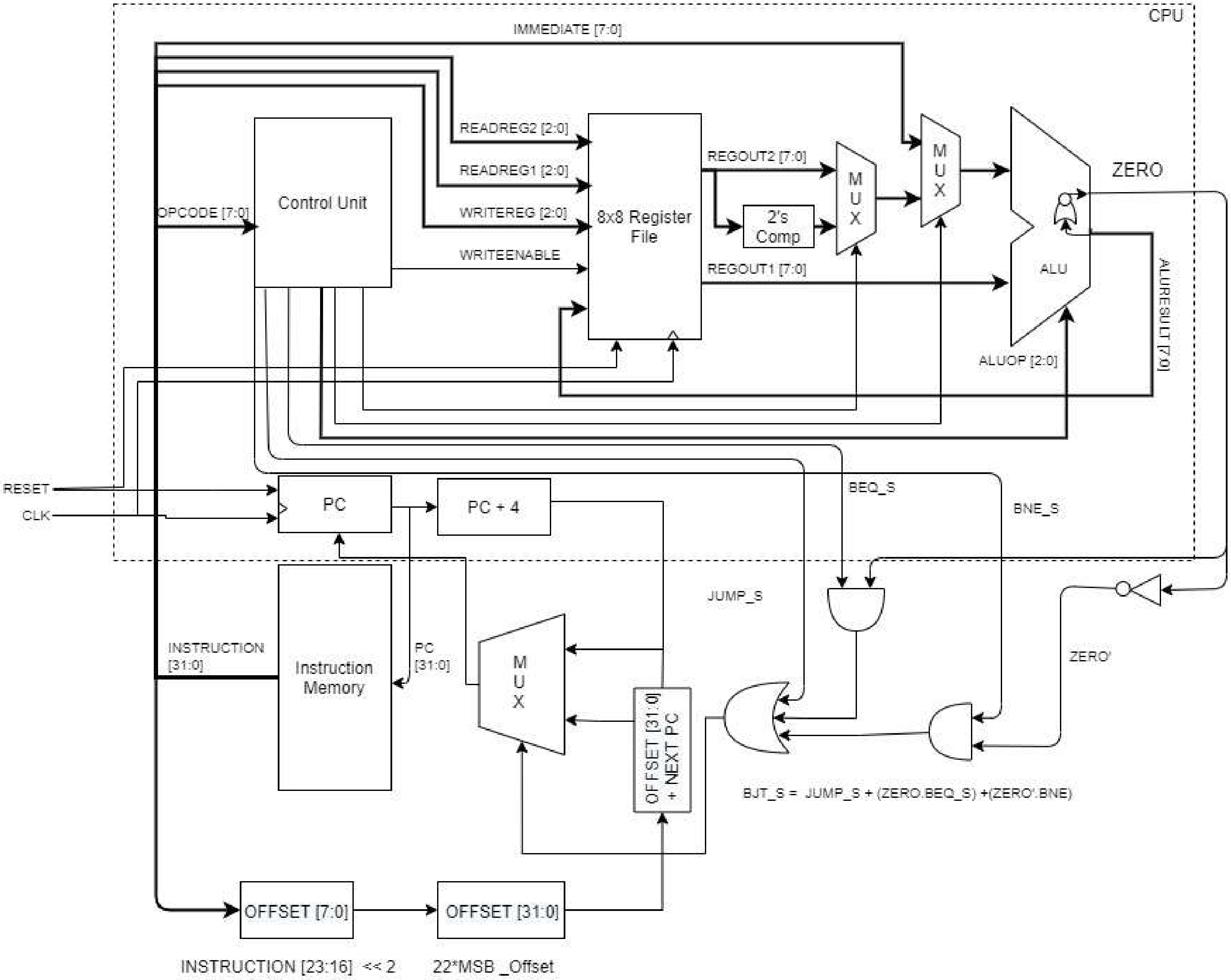
	branch	jump	zero
beq	1	0	1
bne	1	1	0
j	0	1	0

z \ bj	00	01	11	10
0	0	1	1	0
1	0	1	0	1

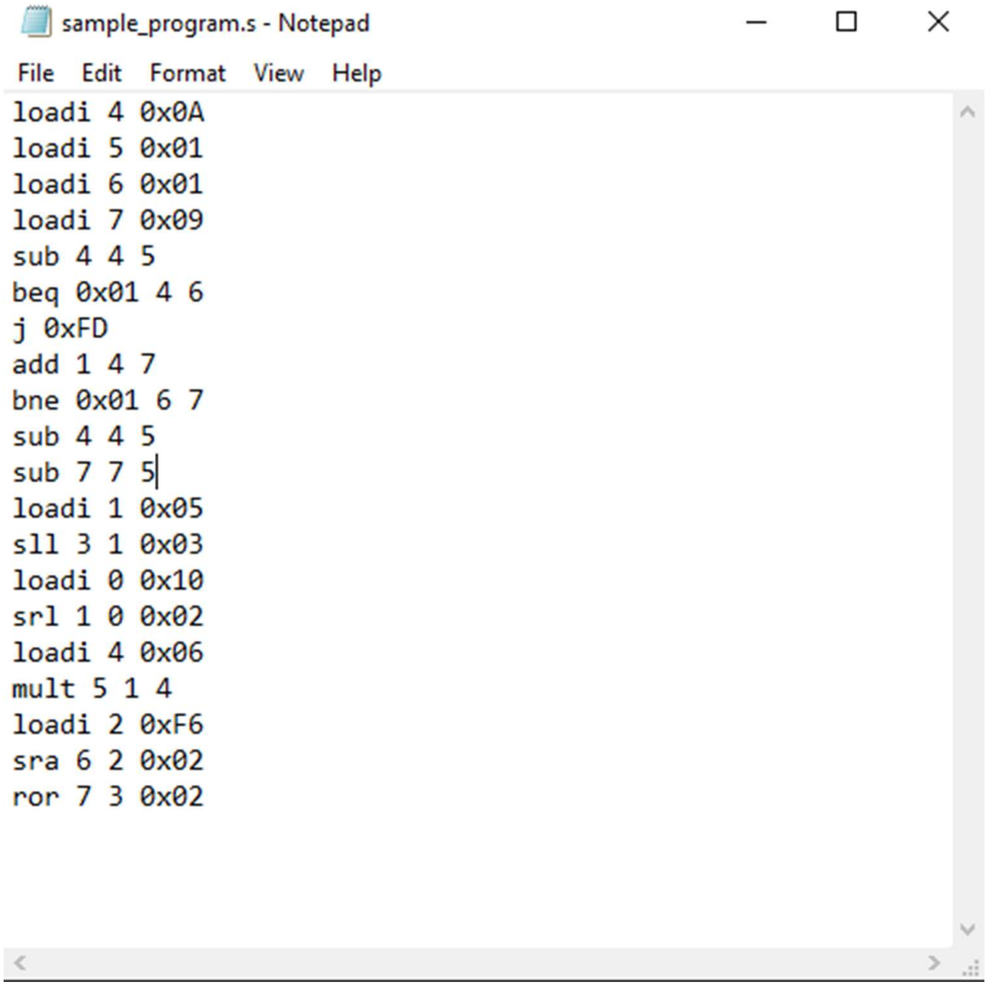
offset_Select = $\bar{b}j + j\bar{z} + zb\bar{j}$



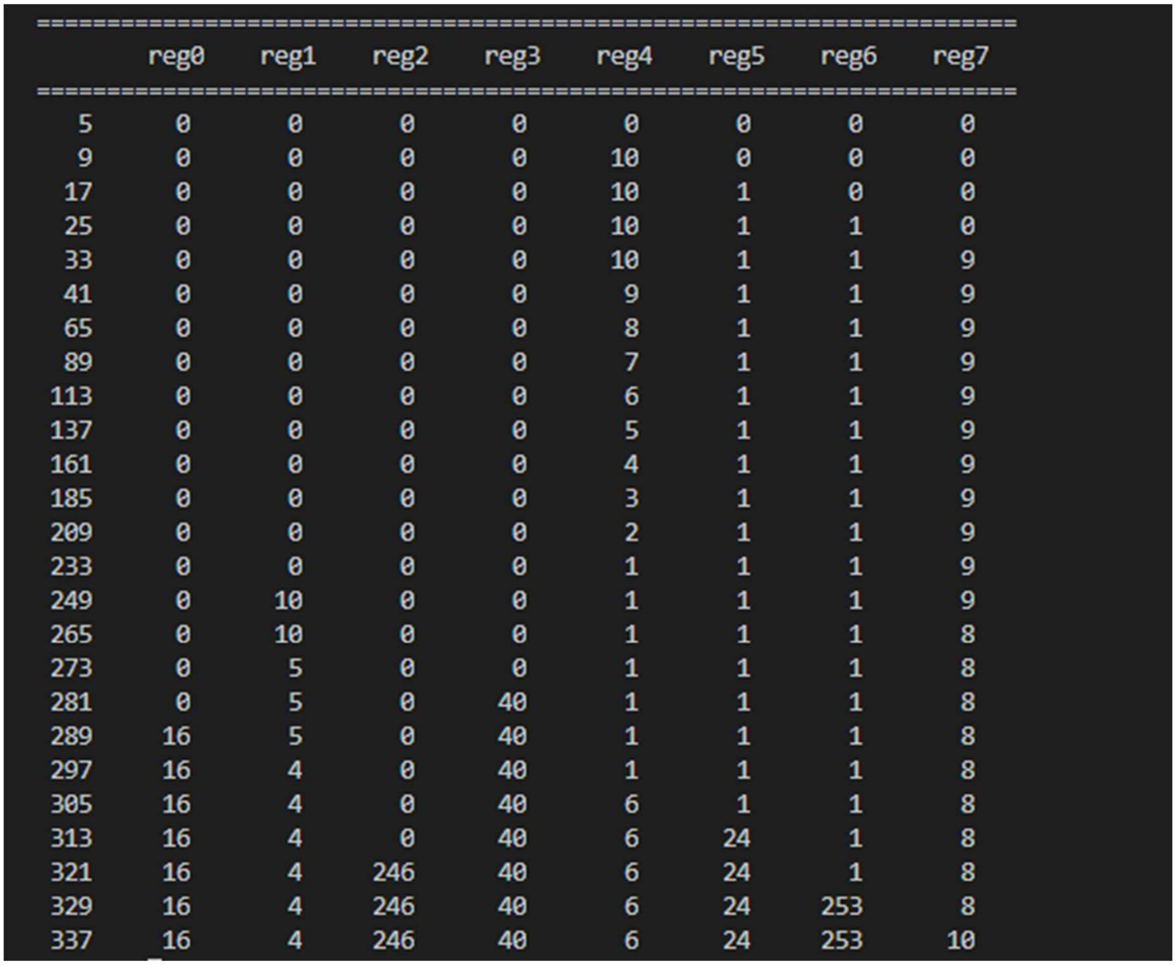
```
//find the offset corosponding to the jump and branch vlues
assign offset = INSTRUCTION[23:16];
assign extended_offset = { {22{offset[7]}}, offset, 2'b00};

//enable offset signal for branch equal,branch not equal or jump
wire temp1,temp2,temp3;
and an1(temp1,~branch,jump);
and an2(temp2,jump,~ZERO,branch);
and an3(temp3,branch,~jump,ZERO);
or o(offset_select,temp1,temp2,temp3);
```



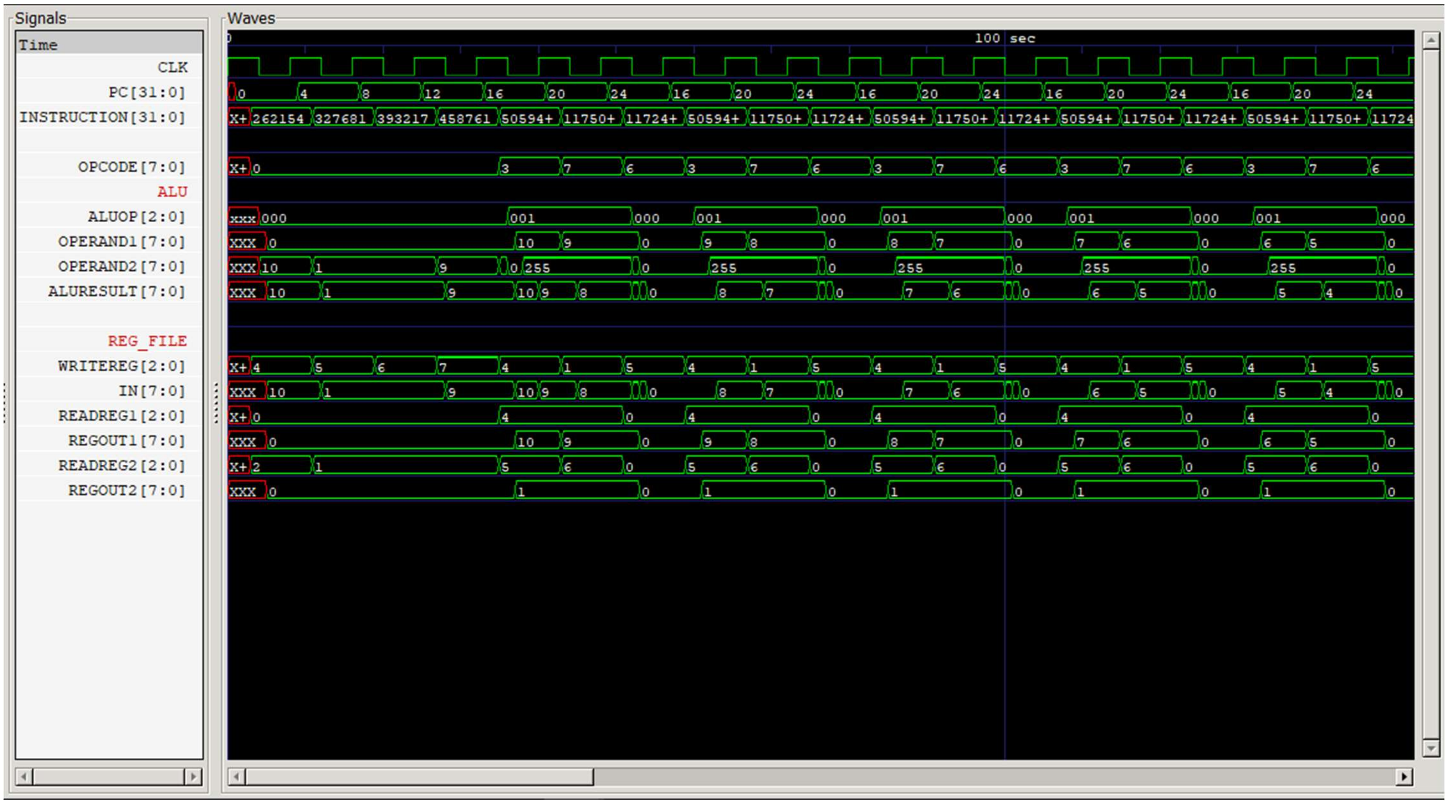
Sample testing:

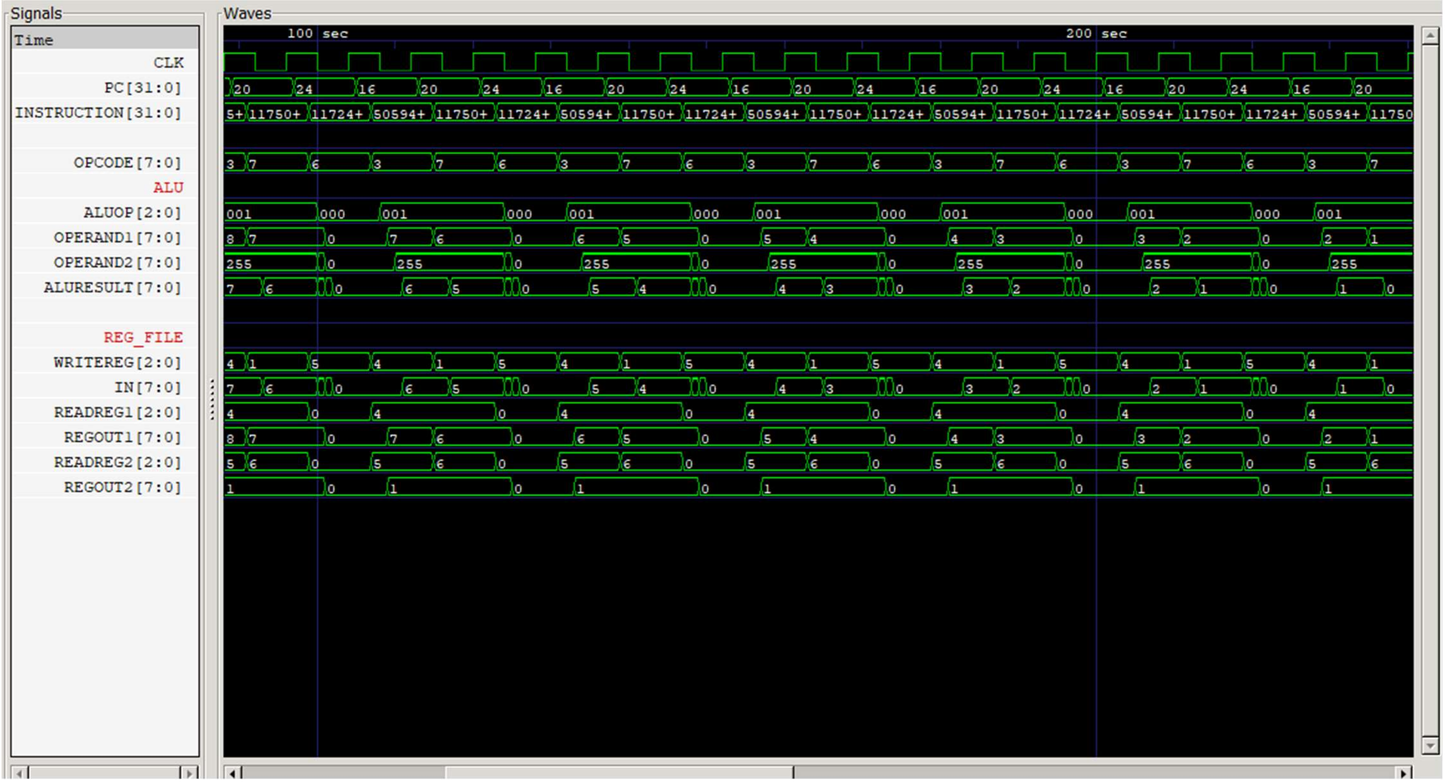


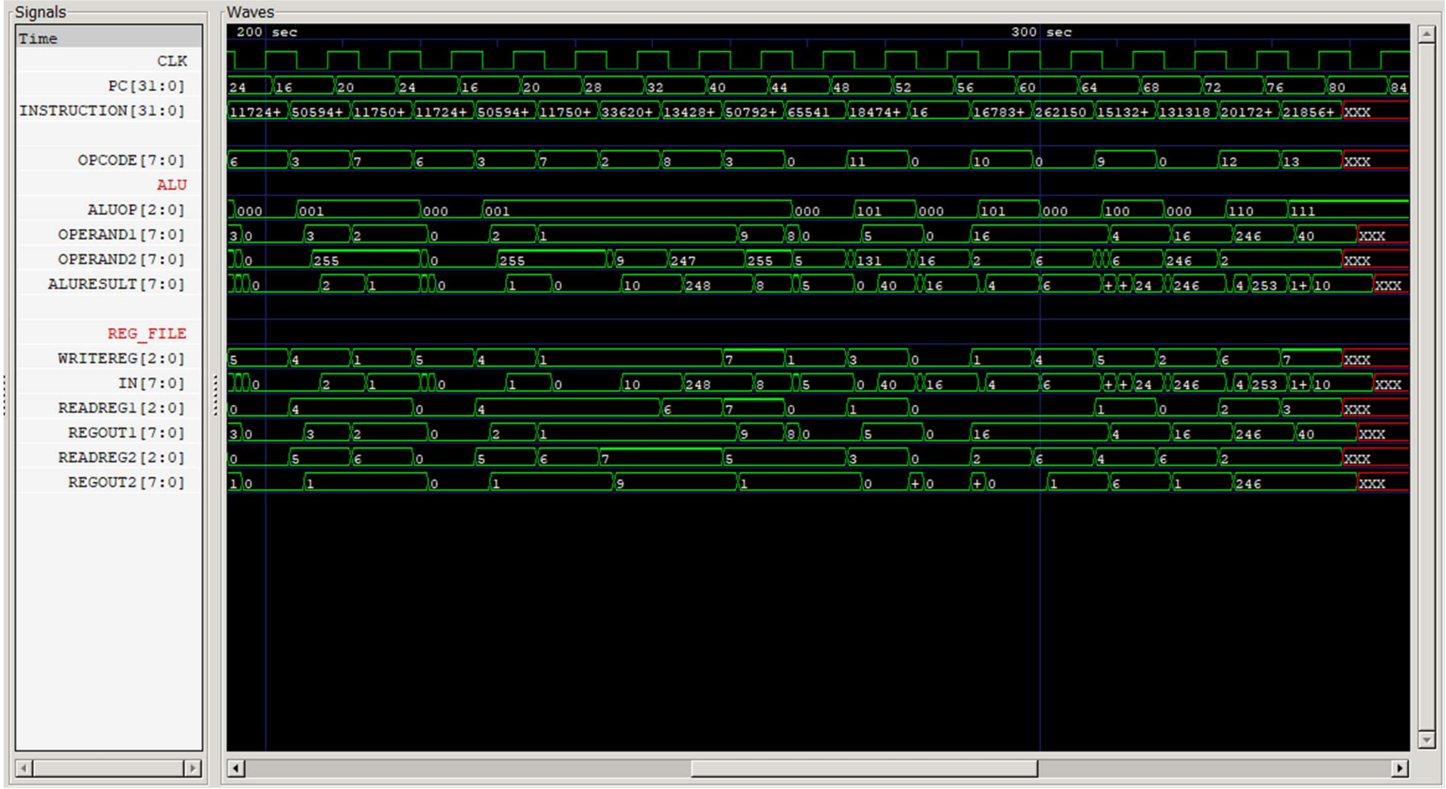
```
loadi 4 0x0A
loadi 5 0x01
loadi 6 0x01
loadi 7 0x09
sub 4 4 5
beq 0x01 4 6
j 0xFD
add 1 4 7
bne 0x01 6 7
sub 4 4 5
sub 7 7 5
loadi 1 0x05
sll 3 1 0x03
loadi 0 0x10
srl 1 0 0x02
loadi 4 0x06
mult 5 1 4
loadi 2 0xF6
sra 6 2 0x02
ror 7 3 0x02
```



```
 ======================================================================
          reg0     reg1     reg2     reg3     reg4     reg5     reg6     reg7
 ======================================================================
    5        0        0        0        0        0        0        0        0
    9        0        0        0        0       10        0        0        0
   17        0        0        0        0       10        1        0        0
   25        0        0        0        0       10        1        1        0
   33        0        0        0        0       10        1        1        9
   41        0        0        0        0        9        1        1        9
   65        0        0        0        0        8        1        1        9
   89        0        0        0        0        7        1        1        9
  113        0        0        0        0        6        1        1        9
  137        0        0        0        0        5        1        1        9
  161        0        0        0        0        4        1        1        9
  185        0        0        0        0        3        1        1        9
  209        0        0        0        0        2        1        1        9
  233        0        0        0        0        1        1        1        9
  249        0       10        0        0        1        1        1        9
  265        0       10        0        0        1        1        1        8
  273        0        5        0        0        1        1        1        8
  281        0        5        0       40        1        1        1        8
  289       16        5        0       40        1        1        1        8
  297       16        4        0       40        1        1        1        8
  305       16        4        0       40        6        1        1        8
  313       16        4        0       40        6       24        1        8
  321       16        4      246       40        6       24        1        8
  329       16        4      246       40        6       24      253        8
  337       16        4      246       40        6       24      253       10
```


Signals
Waves
Time
100 sec
CLK
PC[31:0]
0 4 8 12 16 20 24 16 20 24 16 20 24 16 20 24 16 20 24
INSTRUCTION[31:0]
X+ 262154 327681 393217 458761 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724
OPCODE[7:0]
X+ 0 3 7 6 3 7 6 3 7 6 3 7 6 3 7 6
ALU
ALUOP[2:0]
xxx 000 001 000 001 000 001 000 001 000 001 000
OPERAND1[7:0]
XXX 0 10 9 0 9 8 0 8 7 0 7 6 0 6 5 0
OPERAND2[7:0]
XXX 10 1 9 0 255 0 255 0 255 0 255 0 255 0
ALURESULT[7:0]
XXX 10 1 9 10 9 8 0 8 7 0 7 6 0 6 5 0 5 4 0
REG_FILE
WRITEREG[2:0]
X+ 4 5 6 7 4 1 5 4 1 5 4 1 5 4 1 5 4 1 5
IN[7:0]
XXX 10 1 9 10 9 8 0 8 7 0 7 6 0 6 5 0 5 4 0
READREG1[2:0]
X+ 0 4 0 4 0 4 0 4 0 4 0
REGOUT1[7:0]
XXX 0 10 9 0 9 8 0 8 7 0 7 6 0 6 5 0
READREG2[2:0]
X+ 2 1 5 6 0 5 6 0 5 6 0 5 6 0 5 6 0
REGOUT2[7:0]
XXX 0 1 0 1 0 1 0 1 0 1 0


Signals
Waves
Time
100 sec
200 sec
CLK
PC[31:0]
20 24 16 20 24 16 20 24 16 20 24 16 20 24 16 20 24 16 20
INSTRUCTION[31:0]
5+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 11724+ 50594+ 11750
OPCODE[7:0]
3 7 6 3 7 6 3 7 6 3 7 6 3 7 6 3 7 6 3 7
ALU
ALUOP[2:0]
001 000 001 000 001 000 001 000 001 000 001 000 001
OPERAND1[7:0]
8 7 0 7 6 0 6 5 0 5 4 0 4 3 0 3 2 0 2 1
OPERAND2[7:0]
255 0 255 0 255 0 255 0 255 0 255 0 255
ALURESULT[7:0]
7 6 0 6 5 0 5 4 0 4 3 0 3 2 0 2 1 0 1 0
REG_FILE
WRITEREG[2:0]
4 1 5 4 1 5 4 1 5 4 1 5 4 1 5 4 1 5 4 1
IN[7:0]
7 6 0 6 5 0 5 4 0 4 3 0 3 2 0 2 1 0 1 0
READREG1[2:0]
4 0 4 0 4 0 4 0 4 0 4 0 4
REGOUT1[7:0]
8 7 0 7 6 0 6 5 0 5 4 0 4 3 0 3 2 0 2 1
READREG2[2:0]
5 6 0 5 6 0 5 6 0 5 6 0 5 6 0 5 6 0 5 6
REGOUT2[7:0]
1 0 1 0 1 0 1 0 1 0 1 0 1


Signals
Waves
Time
200 sec
300 sec
CLK
PC[31:0]
24 16 20 24 16 20 28 32 40 44 48 52 56 60 64 68 72 76 80 84
INSTRUCTION[31:0]
11724+ 50594+ 11750+ 11724+ 50594+ 11750+ 33620+ 13428+ 50792+ 65541 18474+ 16 16783+ 262150 15132+ 131318 20172+ 21856+ XXX
OPCODE[7:0]
6 3 7 6 3 7 2 8 3 0 11 0 10 0 9 0 12 13 XXX
ALU
ALUOP[2:0]
000 001 000 001 000 101 000 101 000 100 000 110 111
OPERAND1[7:0]
3 0 3 2 0 2 1 9 8 0 5 0 16 4 16 246 40 XXX
OPERAND2[7:0]
0 255 0 255 9 247 255 5 131 16 2 6 6 246 2 XXX
ALURESULT[7:0]
0 2 1 0 1 0 10 248 8 5 0 40 16 4 6 + + 24 246 4 253 1+ 10 XXX
REG_FILE
WRITEREG[2:0]
5 4 1 5 4 1 7 1 3 0 1 4 5 2 6 7 XXX
IN[7:0]
0 2 1 0 1 0 10 248 8 5 0 40 16 4 6 + + 24 246 4 253 1+ 10 XXX
READREG1[2:0]
0 4 0 4 6 7 0 1 0 1 0 2 3 XXX
REGOUT1[7:0]
3 0 3 2 0 2 1 9 8 0 5 0 16 4 16 246 40 XXX
READREG2[2:0]
0 5 6 0 5 6 7 5 3 0 2 6 4 6 2 XXX
REGOUT2[7:0]
1 0 1 0 1 9 1 0 + 0 + 0 1 6 1 246 XXX