**Digital ASIC Design, ECE521, Lab 5**

Arijit Sengupta, 001441748

**Verilog Codes:**

adder.v alu.v systolic.v testbenchALU.v

```
1  `timescale 1ns / 1ps
2
3  module HalfAdder(sum, Cout, a, b);
4      input     a,b;
5      output    Cout, sum;
6      assign    sum  = a ^ b;      // Bitwise XOR            xor(sum, a, b);
7      assign    Cout = a & b;      // Bitwise AND            and(Cout, a, b);
8  endmodule
9
10 module FullAdder(sum, Cout, a, b, Cin);
11     input         a, b, Cin;
12     output        sum, Cout;
13     wire          w1, w2, w3;
14     HalfAdder    HA1 (w1, w2, a, b);
15     HalfAdder    HA2 (sum, w3, Cin, w1);
16     assign       Cout = w2 | w3;          // Bitwise OR
17 endmodule
18
```

adder.v

adder.v alu.v systolic.v testbenchALU.v

```
1  `timescale 1ns/10ps
2
3  module systolicH(SUMout, Cout, SUMin, a, b);
4      input a, b;
5      output SUMout, Cout, SUMin;
6      wire w1;
7      assign w1 = a & b;
8      HalfAdder  HA1(SUMout, Cout, SUMin, w1);
9  endmodule
10
11 module systolicF(SUMout, Cout, SUMin, a, b, Cin);
12     input a, b, Cin;
13     output SUMout, Cout, SUMin;
14     wire w1;
15     assign w1 = a & b;
16     FullAdder  FA1(SUMout, Cout, SUMin, w1, Cin);
17 endmodule
18
19 module systolicMult4x4(p, a, b);
20     input  [3:0] a, b;
21     output [7:0] p;
22     wire  [11:0] c;
23     wire  [11:0] s;
24     wire         a0b0, a1b0, a2b0, a3b0;
25
26     assign a0b0 = a[0] & b[0];
27     assign a1b0 = a[1] & b[0];
28     assign a2b0 = a[2] & b[0];
29     assign a3b0 = a[3] & b[0];
30     assign p[0] = a0b0;
31
```

```
32      systolicH S00(s[0], c[0], a1b0, a[0], b[1]);
33      systolicF S01(s[1], c[1], a2b0, a[1], b[1], c[0]);
34      systolicF S02(s[2], c[2], a3b0, a[2], b[1], c[1]);
35      systolicH S03(s[3], c[3], c[2], a[3], b[1]);
36      systolicH S04(s[4], c[4], s[1], a[0], b[2]);
37      systolicF S05(s[5], c[5], s[2], a[1], b[2], c[4]);
38      systolicF S06(s[6], c[6], s[3], a[2], b[2], c[5]);
39      systolicF S07(s[7], c[7], c[3], a[3], b[2], c[6]);
40      systolicH S08(s[8], c[8], s[5], a[0], b[3]);
41      systolicF S09(s[9], c[9], s[6], a[1], b[3], c[8]);
42      systolicF S10(s[10], c[10], s[7], a[2], b[3], c[9]);
43      systolicF S11(s[11], c[11], c[7], a[3], b[3], c[10]);
44
45      assign p[1] = s[0];
46      assign p[2] = s[4];
47      assign p[3] = s[8];
48      assign p[4] = s[9];
49      assign p[5] = s[10];
50      assign p[6] = s[11];
51      assign p[7] = c[11];
52  endmodule
53
54
```

systolic.v

adder.v | alu.v | systolic.v | testbenchALU.v

```
1   `timescale 1ns/10ps
2   `define  st0   0
3   `define  st1   1
4   `define  st2   2
5   `define  st3   3
6   module ALU (REGX, REGY, REGA, REGB, REGF, REGS);
7       input       [1:0] REGF;
8       input       [7:0] REGA, REGB;
9       output reg  [7:0] REGX, REGY;
10      output      [1:0] REGS;
11      reg         [3:0] x,y,s,t;
12      wire        [7:0] xt, xs, yt, ys;
13
14      systolicMult4x4 sysmul1(yt, y, t);
15      systolicMult4x4 sysmul2(xs, x, s);
16      systolicMult4x4 sysmul3(xt, x, t);
17      systolicMult4x4 sysmul4(ys, y, s);
18
19      always @*
20          case(REGF)
21              2'b00: begin //AND
22                      REGS = `st0;
23                      REGX = REGA & REGB;
24                      REGY = 0;
25                     end
26
27              2'b01: begin //ADD
28                      REGS = `st0;
29                      {REGY, REGX} = (REGA + REGB);
30                     end
31
```

```
            2'b10: begin  //MULTIPLY
                        x = REGA[7:4];
                        y = REGA[3:0];
                        s = REGB[7:4];
                        t = REGB[3:0];
                        REGS = `st0;
                        REGX = yt;
                        REGS = `st1;
                        REGY = xs;
                        REGS = `st2;
                        {REGY, REGX} = {REGY, REGX} + (xt<<4);
                        REGS = `st3;
                        {REGY, REGX} = {REGY, REGX} + (ys<<4);
                        REGS = `st0;
                    end

            2'b11: begin  //XOR
                     REGS = `st0;
                     REGX = REGA ^ REGB;
                     REGY = 0;
                    end
        endcase
    endmodule
```

alu.v

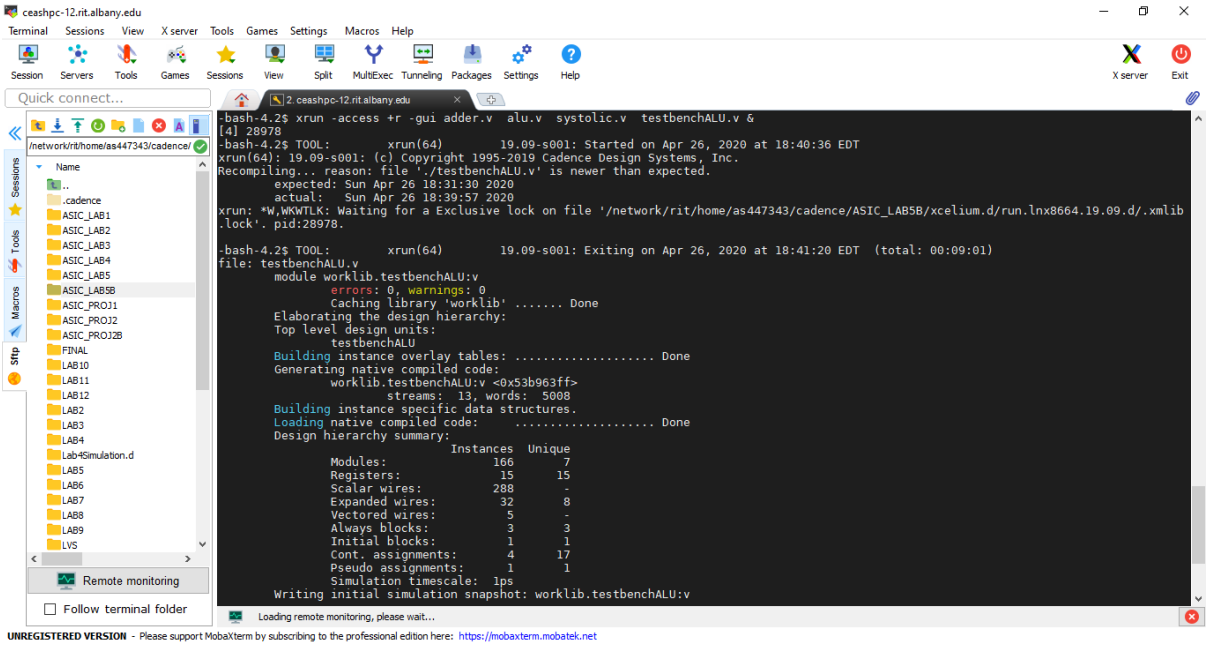
adder.v | alu.v | systolic.v | testbenchALU.v

```
`timescale 10ps/10ps

module testbenchALU();
    reg [15:0] prevcount, count;
    reg [7:0] inp0, inp1;
    wire [7:0] out0, out1;
    reg  [1:0] prevfunc, func;
    reg  [2:0] cyc;
    reg [19:0] calc;

    initial begin
        count <= 16'b1010_1100_1110_0001;
        func <= 1;
        prevfunc <= 0;
        prevcount <= count;
        calc <= 1;
    end

    always @(calc)
        begin
            if (prevfunc == 0) begin #20 func <=2; cyc <= 0; end
            else if (prevfunc == 1) #20 func <=3;
            else if (prevfunc == 2) begin
                if (cyc < 3)
                    #20 func <= 1;
                else begin
                    #20 func <= 2;
                    cyc <= cyc+1;
                end
            end
            else if (prevfunc == 3) #20 func <=0;
            calc <= calc + 1;
        end
```

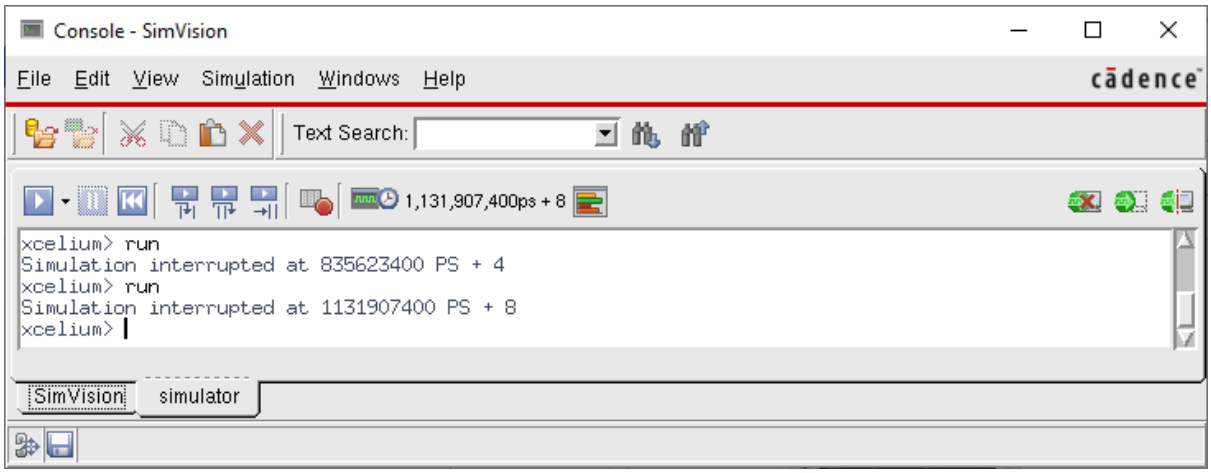
```
34
35      always @(func)
36          begin
37              count[14:0] <= prevcount[15:1];
38              count[15] <= prevcount[0] ^ prevcount[2] ^ prevcount[3] ^ prevcount[5] ^ prevcount[15];
39              prevfunc <= func;
40              prevcount <= count;
41              inp0 <= count[7:0];
42              inp1 <= count[15:8];
43          end
44
45      ALU testALU(.REGX(out0), .REGY(out1), .REGA(inp0), .REGB(inp1), .REGF(func));
46  endmodule
47
48
```

testbenchALU.v

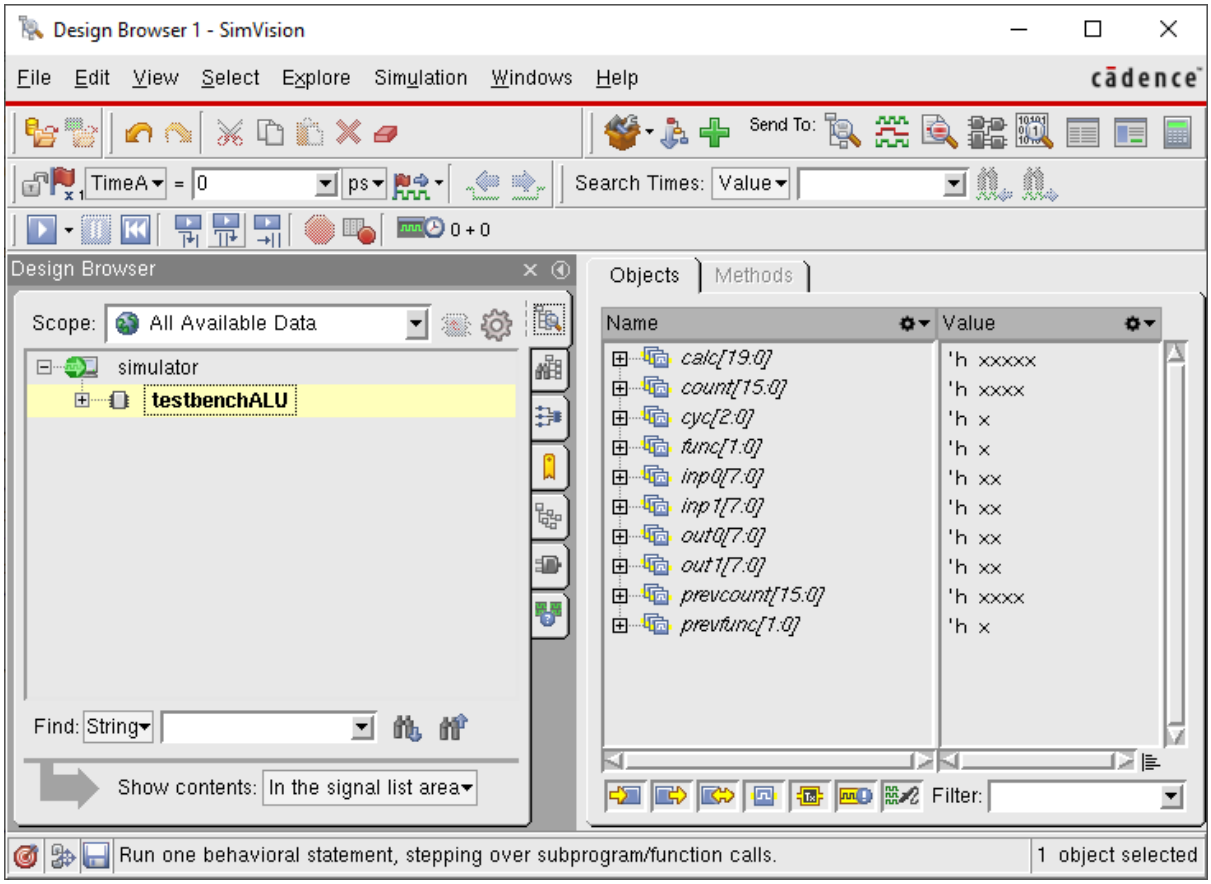
**Compilation and Results:**



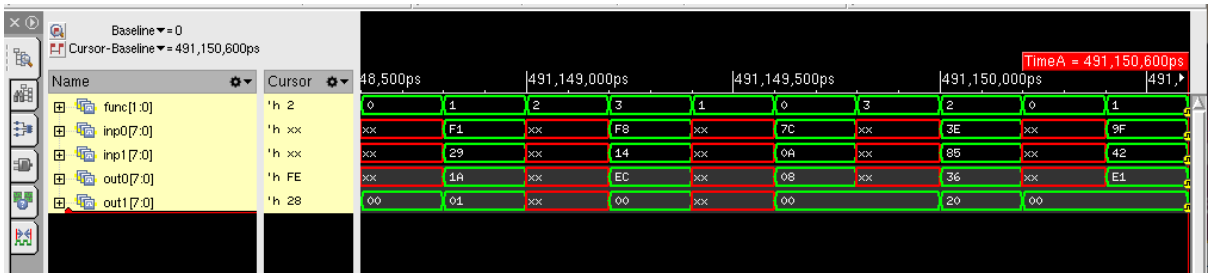

SSH terminal message




SimVision Console




SimVision Design Browser




SimVision Waveform

**Explanation and Manual Verification:**

I have used a relatively uniform high delay in between the calculator operations (200ps, as evident from #20 in testbenchALU file) which ensures there is no overlap of input or output bits. The downside for that though is it skips each operation every alternate cycle, due to the input being delayed (as seen from the XX in red signifying undefined/not available).

The func[1:0] variable represents the REGF which shows which is the operation that's being performed – 00 corresponds to AND, 01 corresponds to ADD, 10 corresponds to MULTIPLY and 11 corresponds to XOR.

Let's verify each function from the above waveform to see if they work correctly.

- 1-> ADD , F1-> A, 29-> B. Since the base is in hexadecimal, adding them results into F1 + 29 = **11A,** which is 01-> Y and 1A-> X. The answer matches, hence it is operating correctly.

- 3-> XOR, F8-> A, 14-> B. Now XOR gives 1 for different bits and 0 for same bits.
  F8 = 1111 1000
  14 = 0001 0100
  Res = 1110 1100 = **EC** in hex, which is EC-> X and 00-> Y. The answer matches, hence it is operating correctly.

- 0-> AND, 7C-> A, 0A-> B. Now AND gives 1 only when both bits are 1, rest is 0.
  7C = 0111 1100
  0A = 0000 1010
  Res = 0000 1000 = **08** in hex, which is 08-> X and 00-> Y. The answer matches, hence it is operating correctly.

- 2-> MUL, 3E-> A, 85-> B. Now 3E is 62 in decimal and 85 is 133 in decimal. 62*133 = 8246 in decimal, which comes to 2036 in hex that is 20 -> Y and 36 -> X. The answer matches, hence it is operating correctly.

Therefore, the calculator is working perfectly and all the operations are performed correctly.