Homework #4

Part A:

We have an immediate function on an 8-bit number, so lets choose a 9-bit encoding. For a push (which will require the immediate value) the MSB will be 1, otherwise we can look to our 2 LSB to determine the operation:

⇒ Push: 1 X X X X X X X X

opcode 1 — 8-bit immediate value

⇒ 0 0 0 0 0 0 X X

opcode 1 — filler — opcode 2 — where

00: add
01: subtract
10: multiply
11: halt

Part B:

Example A:

1 0 0 0 0 1 0 0 0	push 8
1 0 0 0 0 0 1 0 1	push 5
1 0 0 0 0 0 0 1 1	push 3
0 0 0 0 0 0 0 1 0	mult
0 0 0 0 0 0 0 0 1	sub
0 0 0 0 0 0 0 1 1	halt

Example B:

1 0 0 0 0 0 1 0 1	push 5
1 0 0 0 0 0 0 0 1	push 1
1 0 0 0 0 0 0 1 0	push 2
0 0 0 0 0 0 0 0 0	add
1 0 0 0 0 0 1 0 0	push 4
0 0 0 0 0 0 0 1 0	mult.
0 0 0 0 0 0 0 0 0	add
1 0 0 0 0 0 0 1 1	push 3
0 0 0 0 0 0 0 0 1	sub
0 0 0 0 0 0 0 1 1	halt

Part C:

I'll split my code into sections — the ALU based on

the opcode (which will push, as well as do the arithmatic operands), on to do the double dabble algorithm on the TOS, and another to take the output double dabble and display it.

Part D:

Testbench will just be a clock, since the ROM will be included in the main module.

See appendix for main code, testbench code, testbench simulation, ALM resources, and RTL circuit (Figures 1-9, pages 3-7)

Part E:

My final ALM usage was 83. Initially, my usage was ≈450, mainly due to the fact that I was using an algorithm to convert binary to BCD using division. I changed it to use the double dabble algorithm which saved a lot of ALMs! I also seperated my modules, which ultimately cleaned my code a ton and made it a lot easier to change small things like minimizing bit sizes - which I did in my double dabble after realizing the 100's can't go above 5 (only ever 1 or 0). I also used assign blocks in my ALU to check if my stack 9+8 or stack 9-8 was positive or negative instead of repeating the code. Overall, I couldn't get it to under 67 ALMs but made large improvement from the original 450+!

Assignment 4 Appendix

```
module SplitStack(
    input CLOCK_50,
    output wire [6:0] HEX0,
    output wire [6:0] HEX1,
    output wire [6:0] HEX2,
    output wire [6:6] HEX3,
    output wire [0:0] LEDR);

 wire signed [7:0] stack [9:0];
 wire [8:0] dabble;
 wire [4:0] pc;
 wire [8:0] rom;

 //ROMA   ROMA_inst (
 // .address ( pc ),
 // .clock ( ~CLOCK_50 ),
 // .q ( rom ));

ROMB  ROMB_inst (
    .address ( pc ),
    .clock ( ~CLOCK_50 ),
    .q ( rom )
    );

AbsDab A1(
    .a ( stack[9] ),
    .c ( dabble ),
    .d ( HEX3 ));

Display D2(
    .clk ( CLOCK_50 ),
    .a ( dabble ),
    .b ( HEX2 ),
    .c ( HEX1 ),
    .d ( HEX0 ));

ALU A2(
    .clk ( CLOCK_50 ),
    .a ( rom ),
    .b9 ( stack[9] ), .b8 ( stack[8] ), .b7 ( stack[7] ), .b6 ( stack[6] ), .b5 ( stack[5] ),
    .b4 ( stack[4] ), .b3 ( stack[3] ), .b2 ( stack[2] ), .b1 ( stack[1] ), .b0 ( stack[0] ),
    .c ( pc ),
    .d ( LEDR ));

endmodule
```

Figure 1. Main module code containing instantiations of all the project modules.

```
module AbsDab(
    input clk,
    input signed [7:0] a,
    output reg [8:0] c,
    output wire d);

 integer i;
 assign d = ~a[7];
 reg [7:0] b;

 always @ (negedge clk)
    begin
    if (a < 0)
        b = -a;
    else
        b = a;

    c = 0;
    for (i = 0; i <= 7; i = i + 1)
        begin
        c = {c[7:0], b[7 - i]};
        if (i < 7 && c[3:0] > 4)
            c[3:0] = c[3:0] + 3;
        else if (i < 7 && c[7:4] > 4)
            c[7:4] = c[7:4] + 3;
        end
    end
endmodule
```

Figure 2. Module that takes the absolute value of the top of stack and performs the double dabble algorithm on the absolute value, for export to the display module.

```
module Display(
    input clk,
    input [8:0] a,
    output reg [6:0] b,
    output reg [6:0] c,
    output reg [6:0] d);

 always @ (posedge clk)
    begin
    case(a[8])
    0: b = 7'b1000000;
    1: b = 7'b1111001;
    default: b = 7'b1000000;
    endcase

    case (a[7:4])
    0: c = 7'b1000000;
    1: c = 7'b1111001;
    2: c = 7'b0100100;
    3: c = 7'b0110000;
    4: c = 7'b0011001;
    5: c = 7'b0010010;
    6: c = 7'b0000010;
    7: c = 7'b1111000;
    8: c = 7'b0000000;
    9: c = 7'b0011000;
    endcase

    case (a[3:0])
    0: d = 7'b1000000;
    1: d = 7'b1111001;
    2: d = 7'b0100100;
    3: d = 7'b0110000;
    4: d = 7'b0011001;
    5: d = 7'b0010010;
    6: d = 7'b0000010;
    7: d = 7'b1111000;
    8: d = 7'b0000000;
    9: d = 7'b0011000;
    endcase
    end
endmodule
```

Figure 3. Display module code that takes in the double dabble number computed in the previous module, and displays the BCD on the three number hex display at each negedge of the clock, just after the ALU does computations.

```
module ALU(
    input clk,
    input [8:0] a,
    output reg signed [7:0] b9,
    output reg signed [7:0] b8,
    output reg signed [7:0] b7,
    output reg signed [7:0] b6,
    output reg signed [7:0] b5,
    output reg signed [7:0] b4,
    output reg signed [7:0] b3,
    output reg signed [7:0] b2,
    output reg signed [7:0] b1,
    output reg signed [7:0] b0,
    output reg [4:0] c,
    output reg [0:0] d);

 integer i;
 assign x = b9 + b8;
 assign y = b9 - b8;

 initial
    begin
    i = 0;
    d = 0;
    c = 0;
    end

 always @ (posedge clk)
    begin
    c <= c + 1;
    if (a[8])
       begin
       b9 <= a[7:0];
       b8 <= b9;
       b7 <= b8;
       b6 <= b7;
       b5 <= b6;
       b4 <= b5;
       b3 <= b4;
       b2 <= b3;
       b1 <= b2;
       b0 <= b1;
       end
```

```
43          end
44      else
45          begin
46          case (a[1:0])
47          0: begin
48              if ((b9 > 0 && b8 > 0 && x < 0) || (b9 < 0 && b8 < 0 && x > 0))
49                  d <= 1;
50              else
51                  b9 <= b9 + b8;
52              end
53          1: begin
54              if ((b9 > 0 && b8 < 0 && y < 0) || (b9 < 0 && b8 > 0 && y > 0))
55                  d <= 1;
56              else
57                  b9 <= b9 - b8;
58              end
59          2: begin
60              b9 <= b9 * b8;
61              end
62          3: begin c <= c - 1; end
63          endcase
64          b8 <= b7;
65          b7 <= b6;
66          b6 <= b5;
67          b5 <= b4;
68          b4 <= b3;
69          b3 <= b2;
70          b2 <= b1;
71          b1 <= b0;
72          b0 <= 0;
73          end
74      end
75
76  endmodule
```

Figure 4. Main ALU code, that takes in the instruction from the rom, updates the pc for the next instruction, and has operations to push a number to the top of stack while shifting the rest down, and the compute addition subtraction (with overflow detection) and multiplication while moving the rest of the stuck up. Halt will ensure the pc remains the same to complete the computations.

```
1   `timescale 10ns/1ns
2
3   module SplitStack_tb();
4
5       reg clk;
6
7       SplitStack SS1(clk);
8
9       initial
10          begin
11              #0
12              clk = 0;
13              $monitor("$d $b $b $b $b $b", $realtime, HEX0, HEX1, HEX2, HEX3, LEDR);
14
15              #15
16              $stop;
17          end
18
19      always
20          begin
21          #1 clk = ~clk;
22          end
23
24  endmodule
```

Figure 5. Testbench code, to which registers for the hex displays and led were added after the taking of this photo. Full code for the testbench can be found in the archived project from quartus.

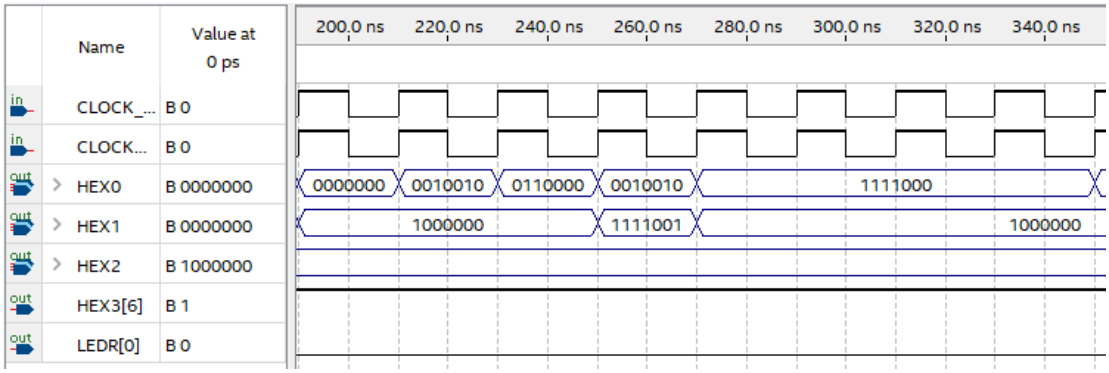


Figure 6. Simulation for ROMA, using waveform simulation because modelsim could not retrieve the altsyncram library. Shows proper function, outputting a value of 7 on the hex displays as expected through hand calculations and as seen on the FPGA board.

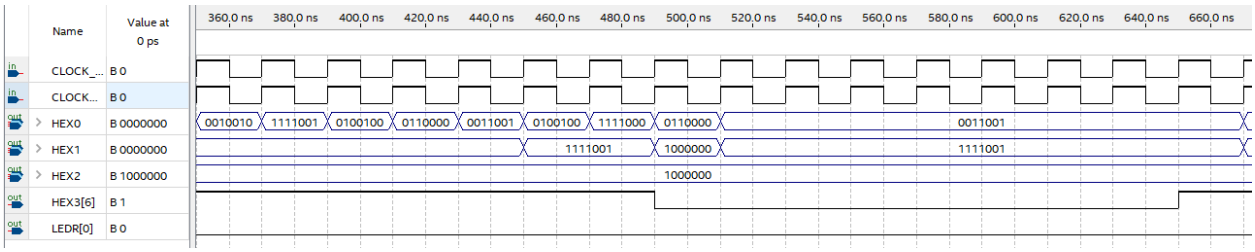
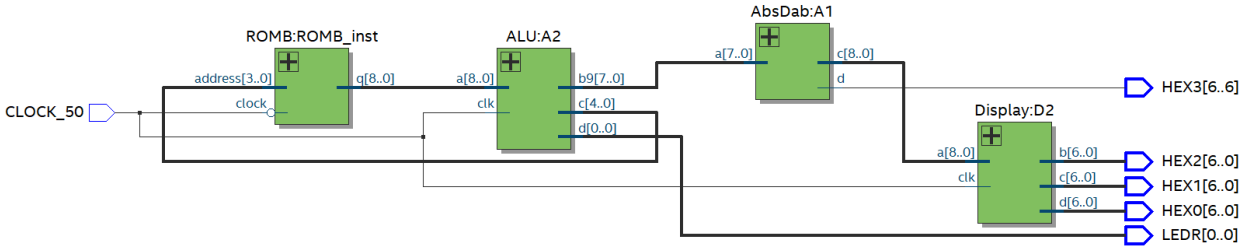
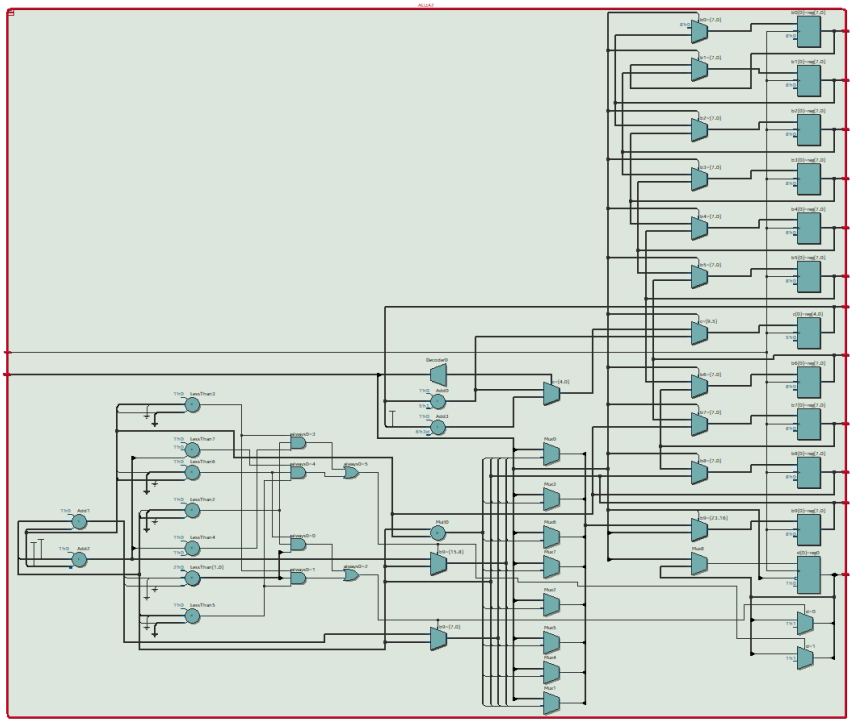


Figure 7. Simulation for ROM B (example B) showing proper functioning again, outputting the expected value of -14. Again, simulation was done in the waveform editor as the ROM could not be synthesized outside of the ALU and Main Module Code using the altsyncram library on the laboratory computers.

Figure 8. Logic Utilization of the main module, showing a usage of 83 ALMs. Optimization procedures can be found in the written report if documentation need to be seen.





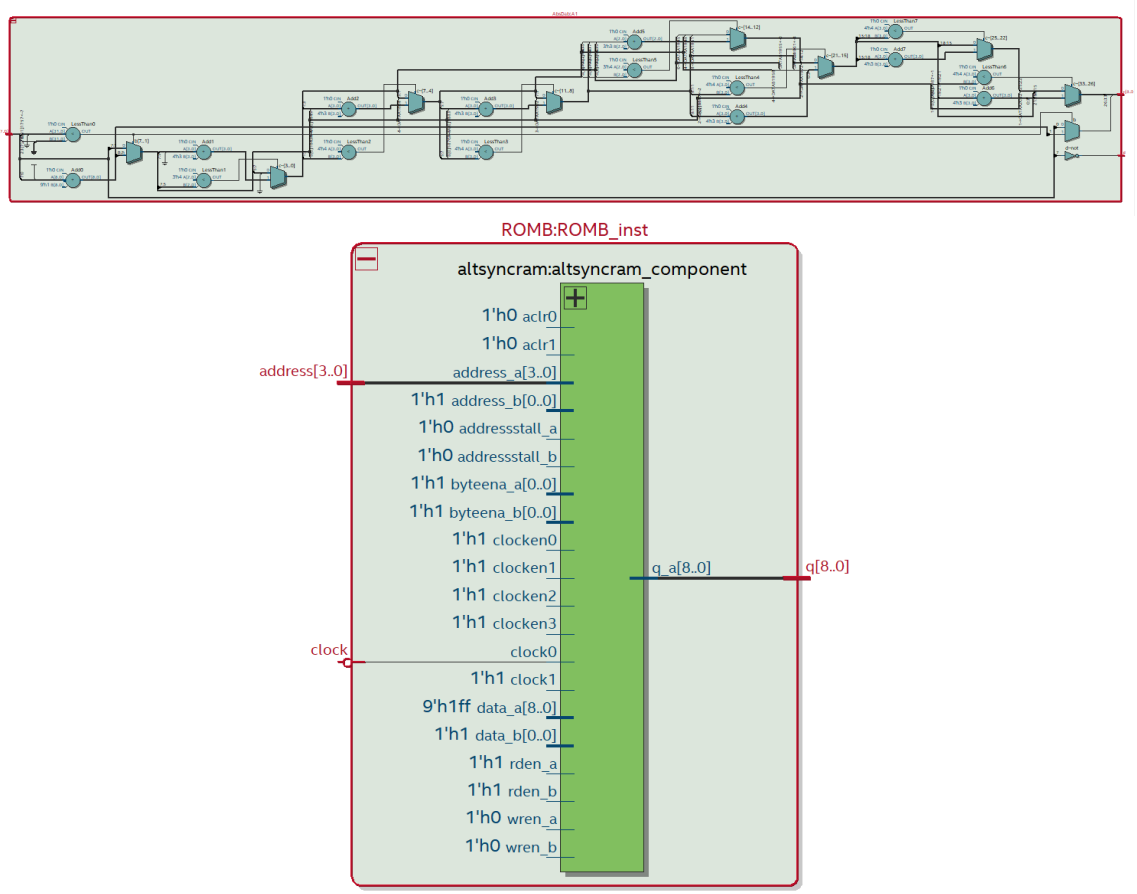


Figure 9. RTL View of the module, including the instantiation of ROMB which is very similar to ROMA. A more detailed view can be found in the main Quartus project.