Vincent Han

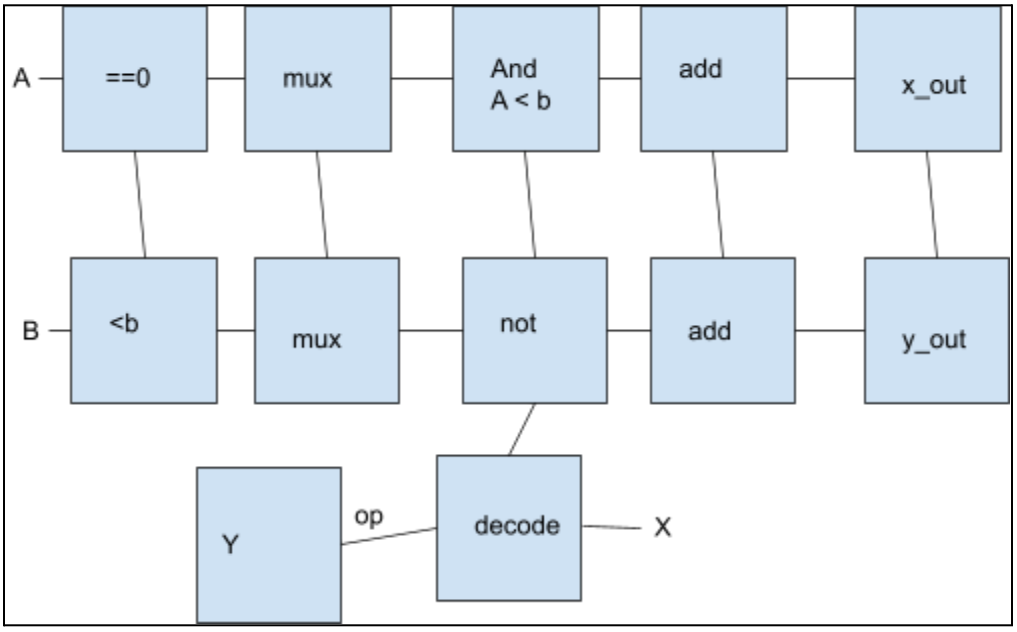
Midterm 2

Problem 1:

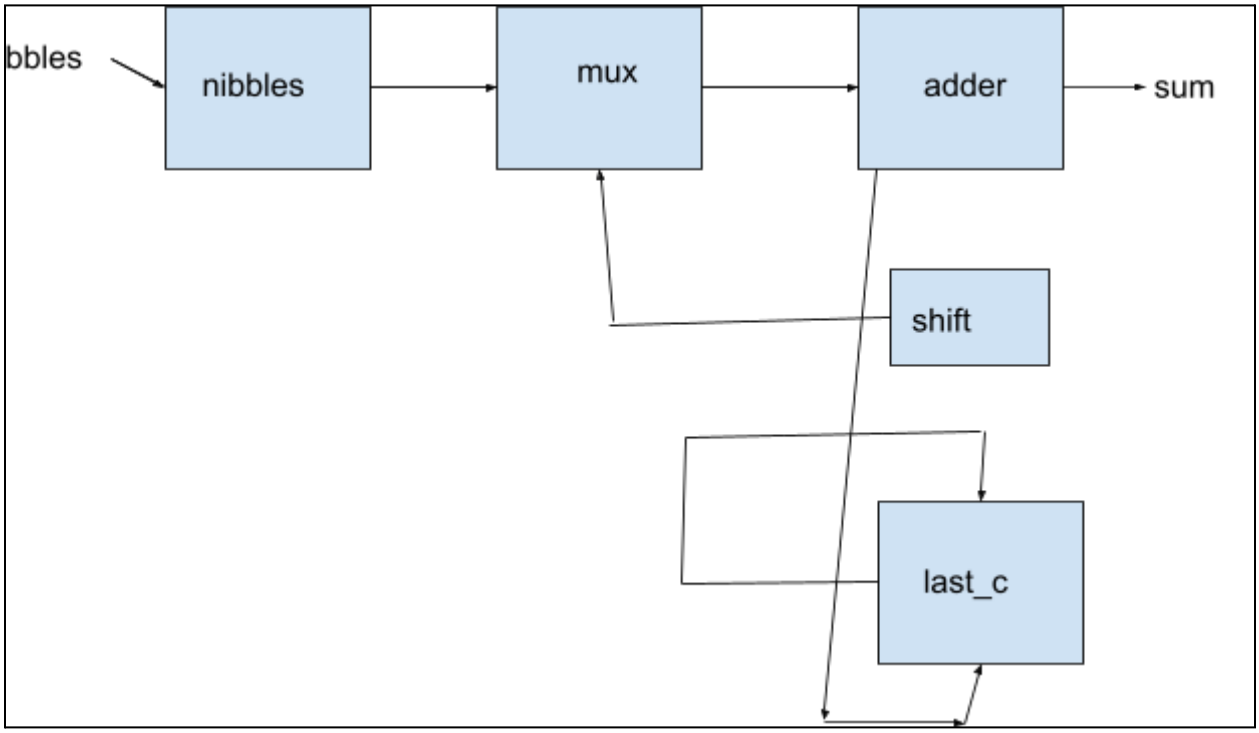
```
module expl_beh(x, y, a, b, c);
  input a, b, c;
  output x, y;

  wire na, nb, nc;
  not #(1) n1(na, a);
  not #(1) n2(nb, b);
  not #(1) n3(nc, c);

  wire t3, t5, t6;
  and #(1) a1(t3, na, b, c);
  and #(1) a2(t5, a, nb, c);
  and #(1) a3(t6, a, b, nc);

  reg x_out, y_out;
  always @(t3, t6)
    x_out = t3 | t6;
  always @(a, t5)
    y_out = a | #(3) t5;

  assign x = x_out;
  assign y = y_out;
endmodule
```

Problem 2:

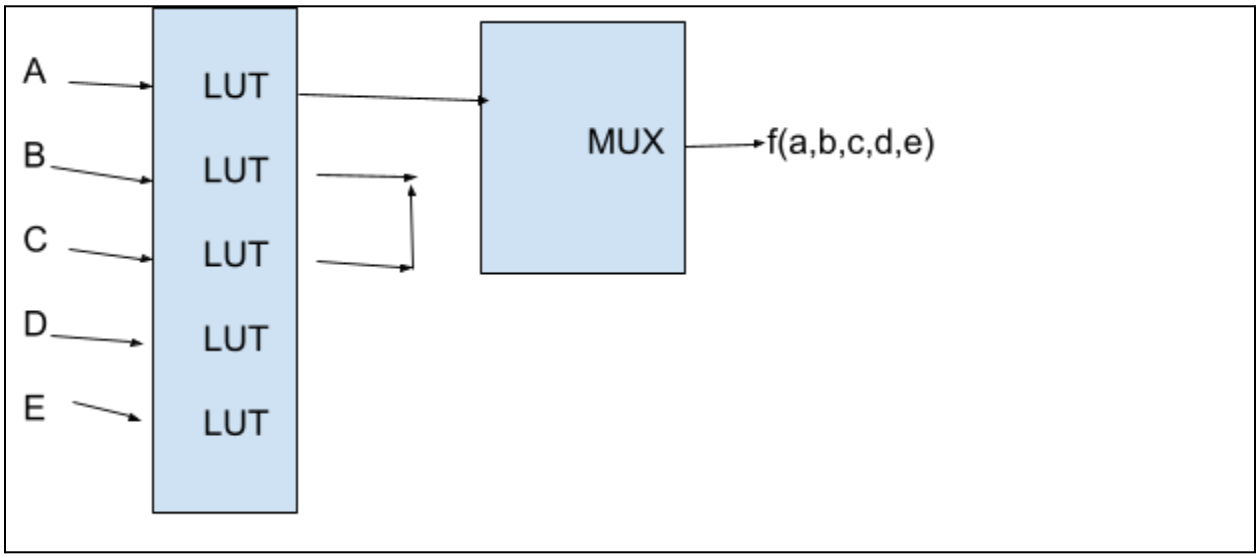
Code A:



Code B:



Problem 3: Show how to implement with 2 LUT4s and a 2-1 MUX.



```
1. LUT1: This LUT takes inputs b, c, d, and e. It is programmed to implement the
function f1(b, c, d, e) based on the truth table of f(a, b, c, d, e) where a = 0
(complemented value). The output of LUT1 represents f1(b, c, d, e).

2. LUT2: This LUT also takes inputs b, c, d, and e. It is programmed to implement
the function f2(b, c, d, e) based on the truth table of f(a, b, c, d, e) where a
= 1. The output of LUT2 represents f2(b, c, d, e).

3. MUX: The 2-1 MUX takes the outputs of LUT1 and LUT2 as its inputs. The
selection input of the MUX is a. When a = 0, the output of LUT1 (f1(b, c, d, e))
is selected as the output of the MUX. When a = 1, the output of LUT2 (f2(b, c, d,
e)) is selected as the output of the MUX.

LUT1 (f1(b, c, d, e)):
Inputs: b, c, d, e
Output: f1(b, c, d, e)

LUT2 (f2(b, c, d, e)):
Inputs: b, c, d, e
Output: f2(b, c, d, e)

MUX:
Inputs: f1(b, c, d, e), f2(b, c, d, e), a
Output: f(a, b, c, d, e)
```

Problem 4:

```
(a) Initial Contents:
Product Register (12 bits): 00000000 101010 (Initial multiplier "10 1001" loaded
in the least significant 6 bits)
Multiplicand Register (6 bits): 000010 (Multiplicand "10 1010")
```

```
(b) Arithmetic and Shift Operations for each iteration:

1. Initial Step:
   - Product Register: 00000000 101010 (Multiplier)
   - Multiplicand Register: 000010 (Multiplicand)

2. Iteration 1:
   - Right Shift (Shift multiplier one position to the right)
   - Product Register: 00000001 010100 (After Shift)
   - Add (Since the last bit of the multiplier is 1, add the multiplicand to the
   product register)
   - Product Register: 00001011 101100 (After Add)
```

```
3. Iteration 2:
   - Right Shift
   - Product Register: 00000101 110110 (After Shift)
   - Add
   - Product Register: 00010111 011000 (After Add)

4. Iteration 3:
   - Right Shift
   - Product Register: 00001011 101100 (After Shift)
   - Add
   - Product Register: 00110011 110000 (After Add)

5. Iteration 4:
   - Right Shift
   - Product Register: 00011001 111000 (After Shift)
   - No Add (Since the last bit of the multiplier is 0, no addition is performed)

6. Iteration 5:
   - Right Shift
   - Product Register: 00001100 111100 (After Shift)
   - No Add

7. Iteration 6:
   - Right Shift
   - Product Register: 00000110 011110 (After Shift)
   - Add
   - Product Register: 00111001 011100 (After Add)
```

```
(c) Contents of the Product Register after the first iteration:
Product Register: 00001011 101100 (After the first iteration)
```

Problem 5:

```
module multiplier(
    input wire signed [7:0] M,  // Multiplicand
    input wire signed [7:0] Q,  // Multiplier
    output reg signed [7:0] A,  // Accumulator
    input wire [3:0] n,         // Counter
    output reg done            // Done
);

    localparam [2:0]
        S_START = 3'b000,
        S_INIT = 3'b001,
        S_CHECK_Q = 3'b010,
        S_SUBTRACT = 3'b011,
        S_ADD = 3'b100,
        S_SHIFT = 3'b101,
        S_CHECK_N = 3'b110,
        S_STOP = 3'b111;

    reg [2:0] state, next_state;
    always @(posedge clk) begin
        if (reset)
            state <= S_START;
        else
            state <= next_state;
    end

    // Output register to hold the result (Qres)
    reg signed [7:0] Qres;

    input wire clk, reset;
```

```
    always @* begin
        case (state)
            S_START: next_state = S_INIT;
            S_INIT: next_state = S_CHECK_Q;
            S_CHECK_Q: next_state = (Q[1:0] == 2'b10) ? S_SUBTRACT : S_ADD;
            S_SUBTRACT: next_state = S_SHIFT;
            S_ADD: next_state = S_SHIFT;
            S_SHIFT: next_state = S_CHECK_N;
            S_CHECK_N: next_state = (n == 4'b0000) ? S_STOP : S_CHECK_Q;
            S_STOP: next_state = S_STOP;
            default: next_state = S_START;
        endcase
    end
```

```
    always @(posedge clk) begin
        case (state)
            S_START: begin
                A <= 0;
                Qres <= 0;
                done <= 0;
            end
            S_INIT: begin
                A <= 0;
                Qres <= 0;
                done <= 0;
            end
            S_CHECK_Q: begin

            end
            S_SUBTRACT: begin
                A <= A - M;
            end
            S_ADD: begin
                A <= A + M;
            end
            S_SHIFT: begin
                {Q, Qres, A} <= {Q[0], A[7:1], Q[7]};
                n <= n - 1;
            end
            S_CHECK_N: begin

            end
            S_STOP: begin
                done <= 1;
            end
            default: begin

            end
        endcase
    end
```