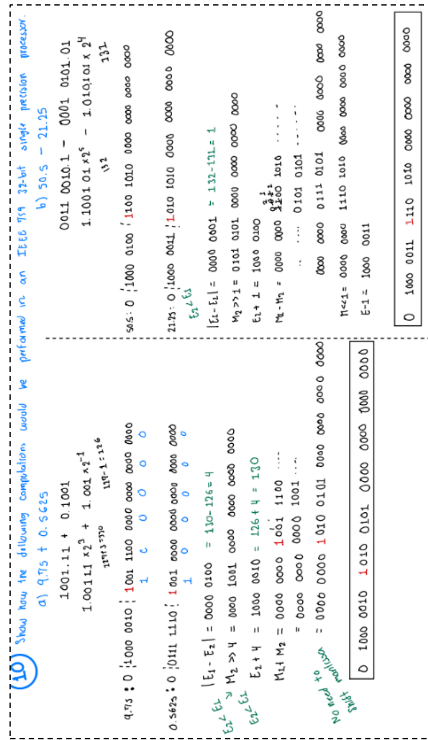
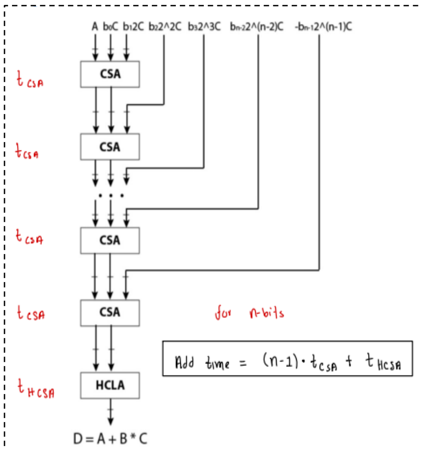
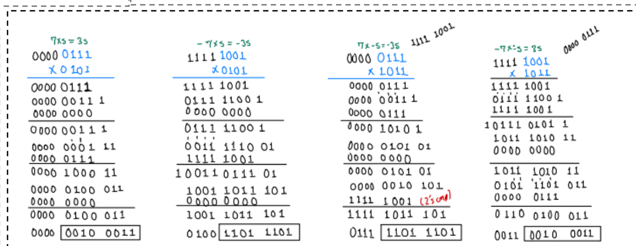
Combinational: No clock needed to operate logic. Output is computer based entirely on current inputs. Multiplication operations can be performed in a parallel manner: fast but not precise.
Pros: Fast and efficient
Cons: Scalability is an issue.
Sequential: Clock dependent. Output is computed based entirely on previous and current inputs. Memory units are introduced to keep history of previous inputs. Multiplication operations are computed in a sequential manner; slow but precise.
Pros: Tackle Scalability
Cons: Slower and more complex
Data path width Number of bito processed simultaneously in multiplication operation. Increasing the width would allow for more bits to be processed -simultaneously.
Pros: Faster computation of data
Cons: Higher Complexity
Recoding Used in high-speed Multipliers to reduce the number of partial products that must be computed and added to produce the product of 2 numbers.
Pros: Reduces number of addition operations required; faster.
Cons: Introduces higher complexity to the overall process.
Pipelining Multiplication is broken down into Sequential stages, allowing for overlapping of such. Each stage performs an operation of the multiplication, and the result is passed onto the next step without waiting for previous operations to be done.

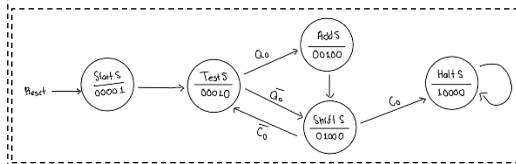
Make an 8x8 Multiplier: Control path no change. Modify data path to accommodate for 8-bit inputs; register Q & M. Also, accommodate Reg A to 9 bits and the counter to 3 bits.
Make it faster: Recoding could be used. Booth's encoding could be implemented; this would decrease the number of partial products to be added.
Allow it to handle signed numbers: Extend Reg M and A to 8 & 9 bits. Check if M and Q are neg by checking MSB. If M is neg, extended signs bits are set, upper half; else they are not set; do Multiplication. If Q is neg, on last add operation of process add the 2's comp of M to reg A; else regular M us added. If both M and Q are neg, both operations are performed.

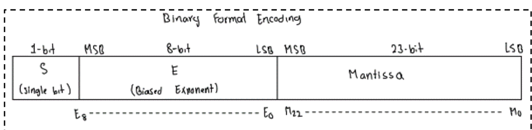
Floating-Point Range
Neg #s: $-2(2-2^{-23}) * 2^{128}$ to 2^{-127}
Pos #s: 2^{-127} to $(2-2^{-23}) * 2^{128}$
Normalization: this entails formatting the mantissa with the implicit 1 to the left of the decimal point.
Exponent Overflow/Underflow: the exponent can be represented in only 8-bits, and thus there is a limited range of numbers that can be represented. Going over max val = overflow. Going below min val = underflow.
Significand overflow/underflow: 23-bits used to represent mantissa. Limited range. If number magnitude exceeds max val = overflow. If number magnitude is too close to zero = underflow.
+ infinity = 0 | 1111 1111 | 000 0000 0000 0000 0000 0000
- infinity = 1 | 1111 1111 | 000 0000 0000 0000 0000 0000







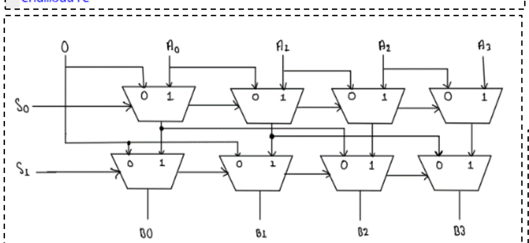




i. 1.6328125×2^{-20}	=	0	0110 1011	101 0001 0000 0000 0000 0000
ii. 1.6328125×2^{20}	=	0	1001 0011	101 0001 0000 0000 0000 0000
iii. $-1.6328125 \times 2^{-20}$	=	1	0110 1011	101 0001 0000 0000 0000 0000
iv. -1.6328125×2^{20}	=	1	1001 0011	101 0001 0000 0000 0000 0000

Binary Array Mult vs Wallace Mult
Binary Array Mult: algo based on hand-written multiplication process, where each bit of multiplicand is multiplied w each bit of multiplier, and the partial products are then added to compute result.
Wallace Mult; uses Wallace tree reduction to group partial products into smaller groups and perform parallel reduction to minimize number of addition operations.

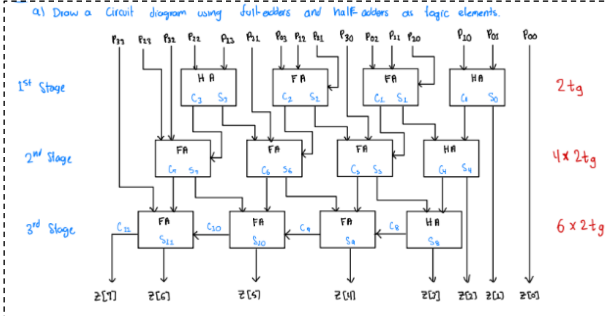
```
module LeftShift
(
    input [3:0] in,
    input [1:0] ctl,
    output logic [3:0] out
);
    logic [3:0] x;

    // 2-bit shift
    mux2x1 m1 (.in0(in[3]), .in1(in[1]), .sel(ctl[1]), .out(x[3]));
    mux2x1 m2 (.in0(in[2]), .in1(in[0]), .sel(ctl[1]), .out(x[2]));
    mux2x1 m3 (.in0(in[1]), .in1(1'b0), .sel(ctl[1]), .out(x[1]));
    mux2x1 m4 (.in0(in[0]), .in1(1'b0), .sel(ctl[1]), .out(x[0]));

    // 1 bit shift
    mux2x1 m5 (.in0(x[3]), .in1(x[2]), .sel(ctl[0]), .out(out[3]));
    mux2x1 m6 (.in0(x[2]), .in1(x[1]), .sel(ctl[0]), .out(out[2]));
    mux2x1 m7 (.in0(x[1]), .in1(x[0]), .sel(ctl[0]), .out(out[1]));
    mux2x1 m8 (.in0(x[0]), .in1(1'b0), .sel(ctl[0]), .out(out[0]));
endmodule
```



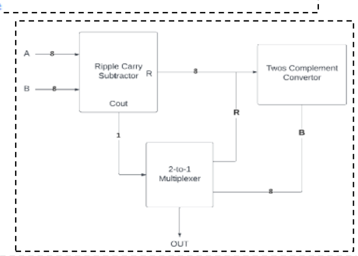
Shift Barrel can perform a shift in one clock cycle.
Advantages: Much faster than regular Shifter
Same time of execution regardless of # bits shifted.
Disadvantages: Much more complex to implement logic
Limited Shift Distance

```
module wallace4x4
(
    input [3:0] A,B,
    output logic [7:0] Z
);
    logic s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
    logic c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11;

    HA h1 (s0, c0, A[0]&B[1], A[1]&B[0]);
    FA f1 (s1, c1, A[0]&B[2], A[1]&B[1], A[2]&B[0]);
    FA f2 (s2, c1, A[0]&B[3], A[1]&B[2], A[2]&B[1]);
    FA h2 (s3, c3, A[2]&B[2], A[1]&B[3]);

    HA h3 (s4, c4, s1, c0);
    FA f3 (s5, c5, s2, A[3]&B[0], c1);
    FA f4 (s6, c6, s3, A[3]&B[1], c2);
    FA f5 (s7, c7, A[2]&B[3], A[3]&B[2], c3);

    HA h4 (s8, c8, s5, c4);
    FA f6 (s9, c9, s6, c5, c8);
    FA f7 (s10, c10, s7, c6, c9);
    FA f8 (s11, c11, A[3]&B[3], c7, c10);

    assign Z = {c11, s11, s10, s9, s8, s4, s0, A[0]&B[0]};
endmodule
```

a) Draw a Circuit diagram using full-adders and half adders as logic elements.



```
module ModeSubtractor #(parameter W = 8)
(
    input [W-1:0] A,
    input [W-1:0] B,
    input OpCode,
    output[W-1:0] R,
    output Cout
);
    wire [W-1:0] out;
    wire [W-1:0] outFinal;

RippleCarrySub #(.N(W)) subtractor
(
    .A(A),
    .B(B),
    .OpCode(OpCode),
    .R(out),
    .Cout(Cout)
);
twoSIGN #(.N(W)) twosComplement
(
    .A(out),
    .B(outFinal)
);
assign R = (Cout == 1'b1) ? outFinal : out;
endmodule
```

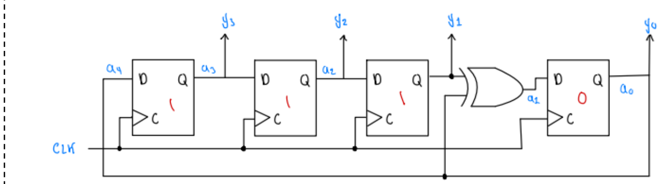


The purpose of the modular exponent subtractor is to find the absolute value of the difference of the two exponent values. This absolute is used in the processing of addition or subtraction of two floating point numbers.

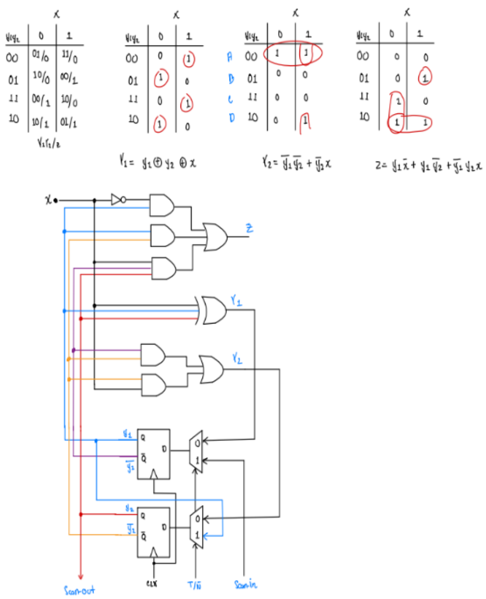
```
module FMS
(
   input X, CLK, ScanIn, T_N
   output logic Z, ScanOut
);
   logic Y1, y1;
   logic Y2, y2;

   assign Z = (y1 & ~X) | (y1 & ~y2) | (~y1 & y2 & X);
   assign Y1 = y1 ^ y2 ^ X;
   assign Y2 = (~y1 & ~y2) | (~y2 & X);
   assign ScanOut = y2;

   always_ff @(negedge CLK) begin
      if(TN == 1)
         Y1 <= ScanIn;
         Y2 <= y1;
      end
      else begin
         Y1 <= y1;
         Y2 <= y2;
   end
endmodule
```

Test Clock Cycles	Test Inputs		Test Outputs	
	Shift In New State y_1y_2	Apply Input x	Verify Output z	Shift Out Previous Next State Y_1Y_2
1-3	0 0	0	0	--
4-6	0 0	1	0	0 1
7-9	0 1	0	0	1 1
10-12	0 1	1	1	1 0
13-15	1 1	0	1	0 0
16-18	1 1	1	0	0 0
19-21	1 0	0	1	1 0
22-25	1 0	1	1	1 0
26-27	--	--	--	0 1

a) Draw the circuit diagram of Type 2 ALFSR described by polynomial $\boxed{x^4+x^3+1}$



Clocks	a_3	a_2	a_1	a_0
1	1	1	1	1
2	1	1	1	0
3	0	1	1	1
4	1	0	1	0
5	0	1	0	1
6	1	0	1	1
7	1	1	0	0
8	0	1	1	0
9	0	0	1	1
10	1	0	0	0
11	0	1	0	0
12	0	0	1	0
13	0	0	0	1
14	1	0	0	1
15	1	1	0	1
16	1	1	1	1




```
//Multiplier controller. Verilog behavioral model.
module MultControl (
   input Clock, Reset, Q0, C0,          //declare inputs
   output Start, Add, Shift, Halt       //declare outputs
);
   logic [4:0] state;                   //five states (one hot - one flip-flop per state)

   //one-hot state assignments for five states
   parameter StartS=5'b00001, TestS=5'b00010, AddS=5'b00100, ShiftS=5'b01000, HaltS=5'b10000;

   logic [1:0] Counter; //2-bit counter for # of algorithm iterations

   // State transitions on positive edge of Clock or Resets
   always_ff @(negedge Clock, negedge Reset)
      if (Reset==0) state <= StartS;          //enter StartS state on Reset
      else                                    //change state on Clock
         case (state)
            StartS: state <= TestS;           // StartS to TestS
            TestS: if (Q0) state <= AddS;     // TestS to AddS if Q0=1
                   else state <= ShiftS;      // TestS to ShiftS if Q0=0
            AddS: state <= ShiftS;            // AddS to ShiftS
            ShiftS: if (C0) state <= HaltS;   // ShiftS to HaltS if C0=1
                    else state <= TestS;      // ShiftS to TestS if C0=0
            HaltS: state <= HaltS;            // stay in HaltS
         endcase

   // Moore model - activate one output per state
   assign Start = state[0];    // Start=1 in state StartS, else 0
   assign Add = state[2];      // Add=1 in state AddS, else 0
   assign Shift = state[3];    // Shift=1 in state ShiftS, else 0
   assign Halt = state[4];     // Halt=1 in state HaltS, else 0
endmodule
```

2

a) 1011 1111

-65_{10}

1011 1111 0

10 01 11 11 11 11 11 10

$\boxed{1\ 1\ 0\ 0\ 0\ 0\ 0\ 1}$

b) 0101 0101

0101 0101 0

01 10 01 10 01 10 01 10

$\boxed{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

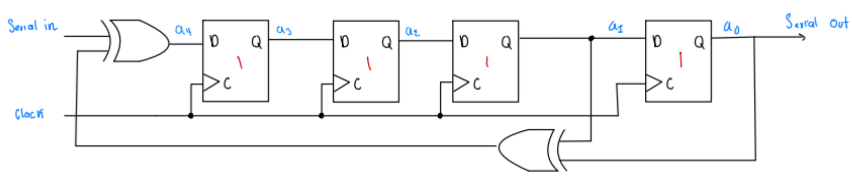
c) Compute 0000 0011 x 1011 1111 using Booth encoding for multiplier

3 x -65 = -195

$\boxed{1111\ 1111\ 0011\ 1101}$

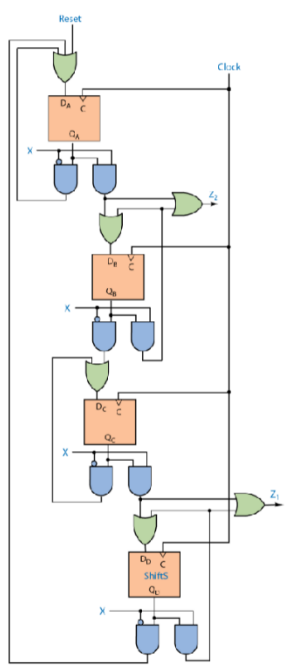
```
   0000 0011
x  1 100 ~~0001~~
------------
   0000 0011
   1111 1100
           1
------------
   1111 1101
   0000 000
   0000 00
   0000 0
   0000
   000
   11
   11
------------
1111 1111 0011 1101
```

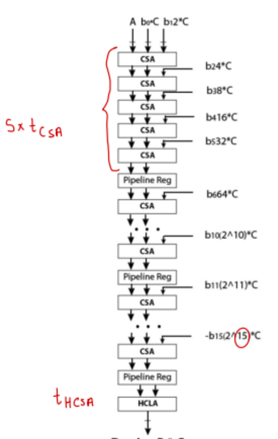
a) Draw the circuit diagram for a serial input signature register (SISR) based on the polynomial x^4+x+1



Serial in	a_3	a_2	a_1	a_0	
1	0	0	1	0	0
0	1	0	0	1	1
1	1	1	0	0	0
0	1	1	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	0	0
0	0	1	1	1	1
0	0	0	1	1	1
1	0	0	0	1	1
0	0	0	0	0	0

	Serial in	a_3	a_2	a_1	a_0	
	1	0	0	1	0	0
	0	1	0	0	1	1
	1	1	1	0	0	0
	0	1	1	1	0	0
	1	0	1	1	1	1
*	0	0	0	1	1	1
	0	0	0	0	1	1
	0	0	0	0	0	0
	0	0	0	0	0	1
	0	0	0	0	0	1
	1	1	0	0	0	1
	0	0	1	0	0	0





$\boxed{1 \text{ clock cycle}}$

?

Compile Time = $(5 \times t_{CLA}) \cdot 3 + t_{HCLA}$

= $\boxed{15\, t_{CSA} + t_{HCSA}}$

The pipeline registers would have to be 22, 27 and 32 bits wide respectively.

0-17
0-18
0-19
0-20
0-$\boxed{21}$
0-22
0-23
0-24
0-25
0-$\boxed{26}$
0-27
0-28
0-29
0-30
0-$\boxed{31}$