EECS 151/251A Homework 9

Instructor: Prof. John Wawrzynek, TAs: Christopher Yarp, Arya Reais-Parsi

Due Monday, May 6th, 2019

Problem 1: Multiplying Signed Numbers by Hand [8 pts]

Using the method shown in class, multiply *by hand* the following **signed** 5-bit numbers. Show your work.

(a) 12×5

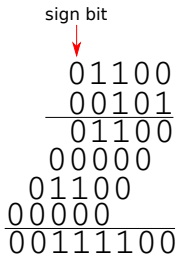
(b) 3×-12

(c) -15×-1

(d) -8×7

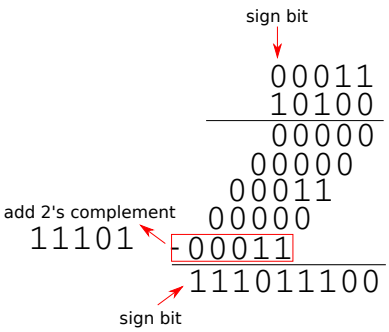
Solution:

1. $12_{10} = 01100_b$; $5_{10} = 00101_b$. $00111100_b = 60_{10}$.

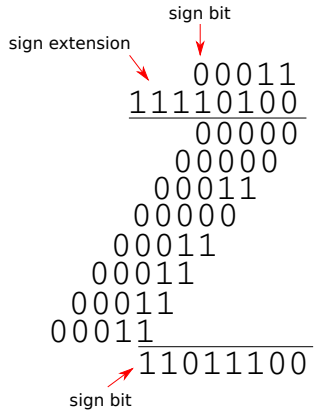


2. For the negative multiplicand, perform multiplication as normal but subtract the final partial product (since the sign bit has negative weight).

   $3_{10} = 00011_b$; $-12_{10} = 10100_b$. $11011100_b = -36_{10}$ (2's complement).

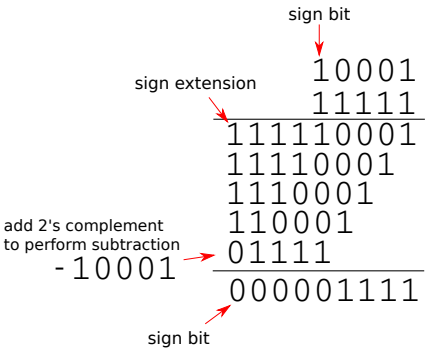


Another way to do this is to sign extend the multiplier directly, as shown below. The trick here is to extend it as far as is appropriate. Multiplying two 5-bit signed numbers will yield a 9-bit signed number. At bit 7 the pattern starts repeating, so higher order bits have been omitted (which is fine to do since our number fits into fewer bits).

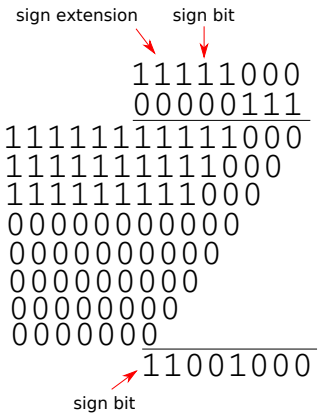


3. When the multiplicand is negative, we proceed as before, but we sign-extend the multiplicand in each partial product:

   $-15_{10} = 10001_b$; $-1_{10} = 11111_b$. $00001111_b = 15_{10}$.



4. $-8_{10} = 11000_b$; $7_{10} = 00111_b$. $11001000_b = -56_{10}$. Again the result has been truncated:



Problem 2: Analyzing Multiplier Structures [14 pts]

Consider multiplying 7×5 using the **unsigned** array multiplier presented in class.

(a) Assume that we use carry-propagate adders, and that the delay of each adder is τ_{FA}. How long does it take to compute the product? What are the intermediate signals in the array (label these signals in a diagram)? In what order are the product bits ready?

(b) Now, assume that we use a carry-save adder instead. How long would it take to compute the product? What are the intermediate signals in the array (label these signals in a diagram)? In what order are the product bits ready?

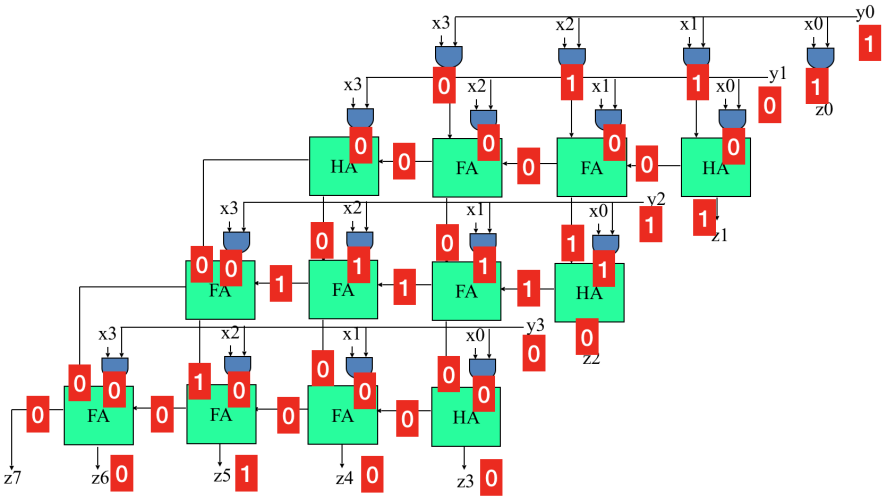
Next, consider multiplying a -7×-5 using a **signed** carry-propagate array multiplier.

(c) Assume that the multiplier does not use any of the Baugh-Wooley simplifications. How long would it take to compute the product? What are the intermediate signals in the array (label these signals in a diagram)? In what order are the product bits ready?

(d) Now, assume that the multiplier uses the Baugh-Wooley simplifications. How long would it take to compute the product? What are the intermediate signals in the array (label these signals in a diagram)? In what order are the product bits ready?

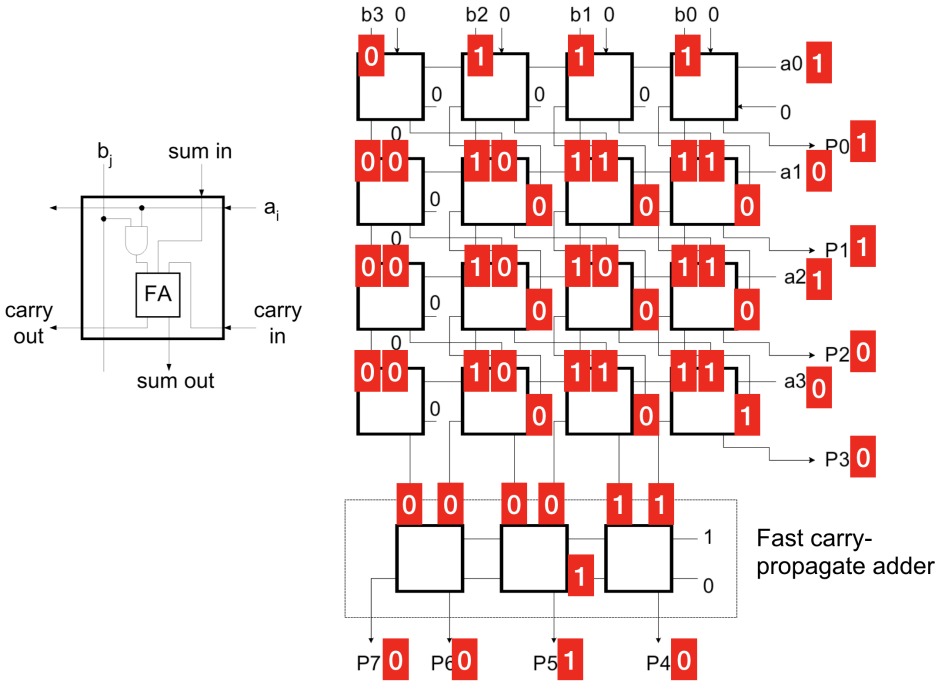
Solution:

1. There are 8 adders in the critical path. Since our analysis is for the sake of cursory comparison, we assume that the AND gate itself has no delay, half- and full-adders both have delay τ_{FA}, and that the first row of adders (which add 0) is omitted (it is not drawn). Then the approximate delay is $6\tau_{FA}$. Products bits are ready in the order from least to most significant bit: $z0, z1, z2, \cdots, z7$.



2. The delay is $6\tau_{FA}$, if we discount the first row (which adds 0 to the first partial product)

as in the figure we have for the CPA in lecture (above). Using the scheme drawn, the bits are ready in the same order as with the CPA above.



3. We have to include an additional adder over that shown in (1). The delay is now $9\tau_{FA}$. Bits are produced in the same order. The diagram of intermediate signals is omitted. A major disadvantage of this approach is the additional area of the adders needed for the sign-extended partial products.

4. Based on our extremely simplistic assumptions, the delay with the Baugh-Wooley simplifiations is still $9\tau_{FA}$. At this point, however, the assumptions break down. We have only added a *half*-adder, not as full-adder as in (3), so this is necessary less. We also saved turning one of our existing half-adders into a full-adder. But most importantly, we saved a large a mount of area by avoiding unnecessary adders.

Problem 3: Booth Recoding [12 pts]

Design a circuit to perform multiplication of two **unsigned** 4-bit numbers with Booth recoding.

Solution:

This solution describes one possible design and should serve as a guide.

The general structure of the circuit is to first set up the necessary constants, and to then accumulate partial products in accordance with Booth's algorithm until done. Note that not all register `enable` inputs are shown. The important ones are those needed to alternate between enabling the setup registers (A, $-A$, B) and the processing registers (X).

There are seven parts:

1. The 4-bit multiplicand input, A
2. Circuitry to generate the 2's complement of A, $-A$
3. The 4-bit multiplier input, B, with an extra bit in a shift-register configuration
4. Lgic to manage the setup and computation steps within the circuit
5. A bank of 5 full-adders to compute partial products on each non-setup cycle
6. Logic to control the input to the full adders according to Booth's algorithm, selecting between adding 0, A, $2A$, $-A$, $-2A$
7. The output register bank X, also configured as a shift register, to accumulate the partial products and store the final result

It is assumed that a "start" signal of some kind is given when the input values are provided. This signal is propagated through shift registers, a one-hot encoding of the state of the circuit. The 4-bit by 4-bit multiplication should be complete in 5 cycles: when `done` is true, the result is the value in $X_7, \cdots, X_0$.

At the end of every cycle "setup", the multiplier and the output product are shifted by 2. The lower three bits of the multiplier (remembering that one 0 is appended to the initial value) determine the action to be taken to accumulate the partial sum, per Booth's algorithm. Since there are five actions we need to generate three control signals, `add_0`, s_1 and s_2, as follows:

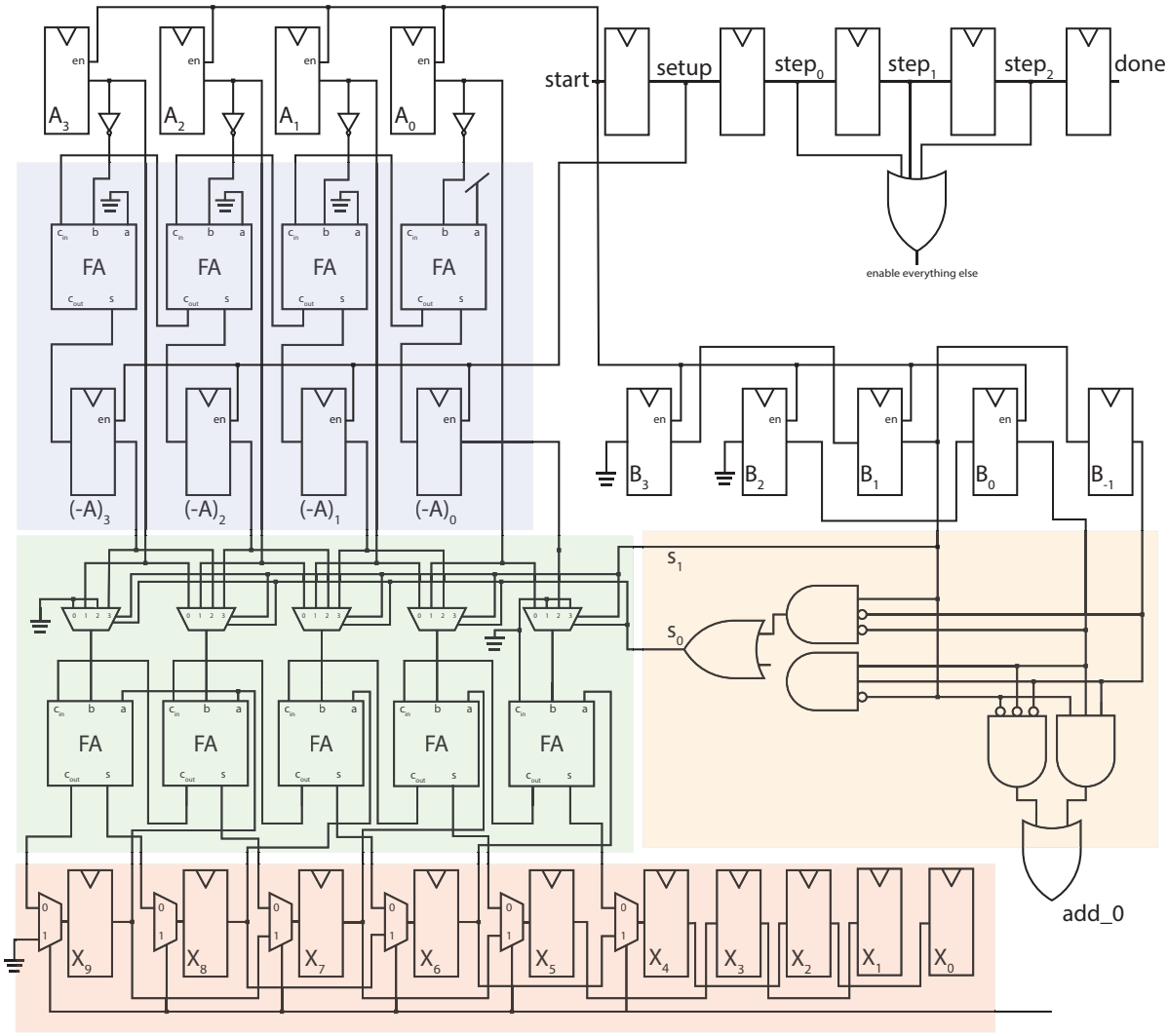
B_{k+1}	B_k	B_{k-1}	action	`add_0`	s_1	s_0
0	0	0	add 0	1	x	x
0	0	1	add A	0	0	0
0	1	0	add A	0	0	0
0	1	1	add 2×A	0	0	1
1	0	0	sub 2×A	0	1	1
1	0	1	sub A	0	1	0
1	1	0	sub A	0	1	0
1	1	1	add 0	1	x	x

(This table is reproduced from that in Lecture 21.)

We can determine the simplified expressions for the control signals by reading a SoP-form expression from this table or by creating a Karnaugh map. The control signals can be assigned arbitrarily, but as written they simplify reasonably well.

Note also that, in order to "add 0", we simply skip the full-adders entirely. This saves us having to use 3-to-1 multiplexors on their inputs.

The output shift register is 2 bits wider than the final product because it must handle all intermediary sums, which can be up to 6 bits wide (5 bits plus a carry). To save using a separate cycle to shift these values, the output of the partial sum computation is inserted into the shifted position at the end of the cycle.



Problem 4: Constant Multiplication [8 pts]

Derive the minimal canonic signed digit (CSD) multiplier for:

(a) $Y = 415 * X$

(b) $Y = 238 * X$

Show your conversion to CSD form and the resulting multiplier circuit using adders and subtractors. Assume that X is a 12-bit unsigned integer.

Solution:

Part a:

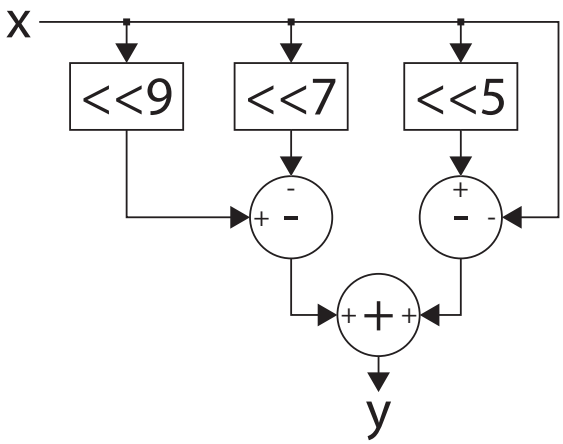
$415 =$	0110011111
Pass 1: $01\ldots10 \rightarrow \bar{1}0$	$10\bar{1}010000\bar{1}$
Pass 2: $01\bar{1}0 \rightarrow 0010, 0\bar{1}10 \rightarrow 00\bar{1}0$	$10\bar{1}010000\bar{1}$

Thus, the CSD form of 415 is $10\bar{1}010000\bar{1}$.

Checking the solution: $512 - 128 + 32 - 1 = 415$.

$415x = (512 - 128 + 32 - 1)x = 2^9x - 2^7x + 2^5x - 2^0x = (x << 9) - (x << 7) + (x << 5) - x$

Thus, the CSD implementation of the constant coefficient multiply is:



Part b:

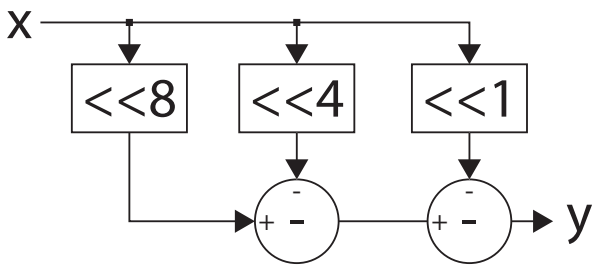
$238 =$	011101110
Pass 1: $01\ldots10 \rightarrow \bar{1}0$	$100\bar{1}100\bar{1}0$
Pass 2: $01\bar{1}0 \rightarrow 0010, 0\bar{1}10 \rightarrow 00\bar{1}0$	$1000\bar{1}00\bar{1}0$

Thus, the CSD form of 415 is $1000\bar{1}00\bar{1}0$.

Checking the solution: $256 - 16 - 2 = 238$.

$238x = (256 - 16 - 2)x = 2^8x - 2^4x - 2^1x = (x << 8) - (x << 4) - (x << 1)$

Thus, the CSD implementation of the constant coefficient multiply is:



Problem 5: *251A Only.* Constant Multiplication [5 pts]

The CSD representation multiplier circuits presented in lecture are exclusive to unsigned values for X and C. Explain how the circuits would need to change to accommodate signed two's-complement values for X and for C.

Solution:

Supporting signed X: Recall that 2's complement numbers can be added just like unsigned numbers. Shifts to the left also accomplish multiplications by powers of 2, just like with unsigned numbers:

$$2^n\alpha_n + -2^{n-1}\alpha_{n-1} + \ldots + 2^1\alpha_1 + 2^0\alpha_0) \cdot 2 = -2^{n+1}\alpha_n + -2^n\alpha_{n-1} + \ldots + 2^2\alpha_1 + 2^1\alpha_0 + 2^0 \cdot 0$$

Therefore, no modifications to the adder structure itself are required. However, it is important to sign extend the shifted values of x before adding.

Supporting signed C: We can exploit the fact that CSD allows digits to have positive, zero, or negative weight when deriving support negative coefficients.

Negative coefficients can be handled using the following procedure:

1. Determine if C is negative. If it is positive, proceed to derive the CSD representation as usual. If not, continue to step 2.
2. Get the absolute value of the negative number by taking its 2's complement (bitwise negate and add 1).
3. Use the procedure from lecture to attain the CSD representation of the absolute value.
4. Replace all 1's in the CSD representation with $\bar{1}$ and all $\bar{1}$'s with 1. This inverts the weights of the components that, when summed together add up to the absolute value of the original digit. Using the distributed property, this is equivalent to multiplying the CSD expression by -1. Since the absolute value is positive, this leads to a CSD representation of the negative number.

Note that, since only the non-zero weights were changed, the number of nonzero weights in the CSD representation remains unchanged between the positive and negative versions of the number. The only difference between them is that the nonzero weights switch sign.

An example of this process is shown below: CSD of -238:

- -238 is negative, compute CSD of absolute value (238)
- CSD of 238 = $1000\bar{1}00\bar{1} = 256 - 16 - 2$
- Swapping 1 for $\bar{1}$ and $\bar{1}$ for 1: $\bar{1}0001001 = -256 + 16 + 2 = -(256 - 16 - 2) = -238$

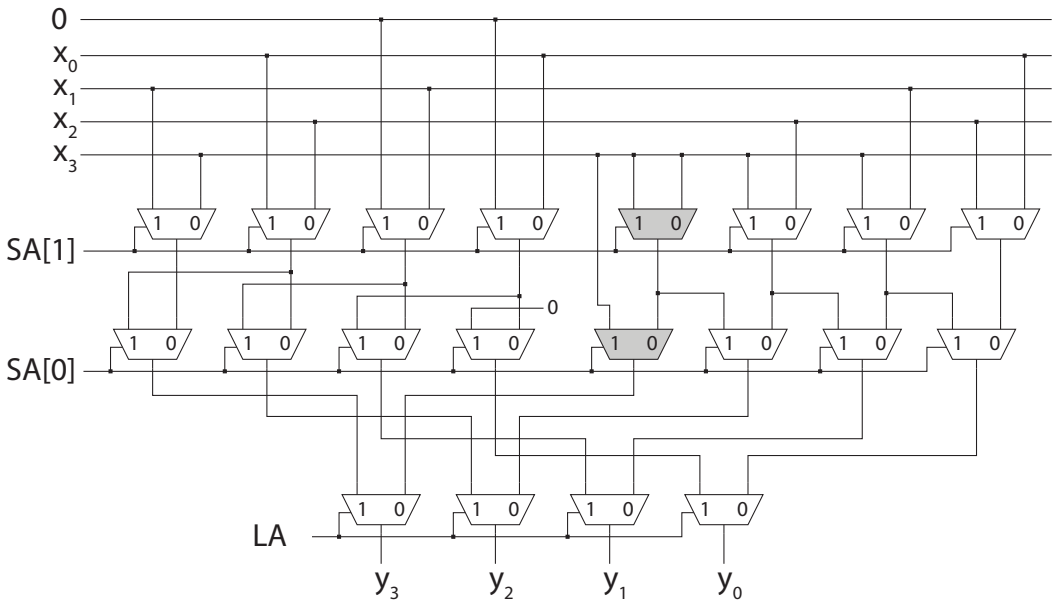
Problem 6: Shifter [8 pts]

Using 2-input multiplexers and simple logic gates, if you need them, draw a combinational circuit that is capable of performing both a left shift or a right arithmetic shift. The circuit takes a 4-bit data input value, X, a 2-bit control for shift amount, SA, and a control bit LR. If LR=1 the circuit shifts left, otherwise it shift right.

Solution:

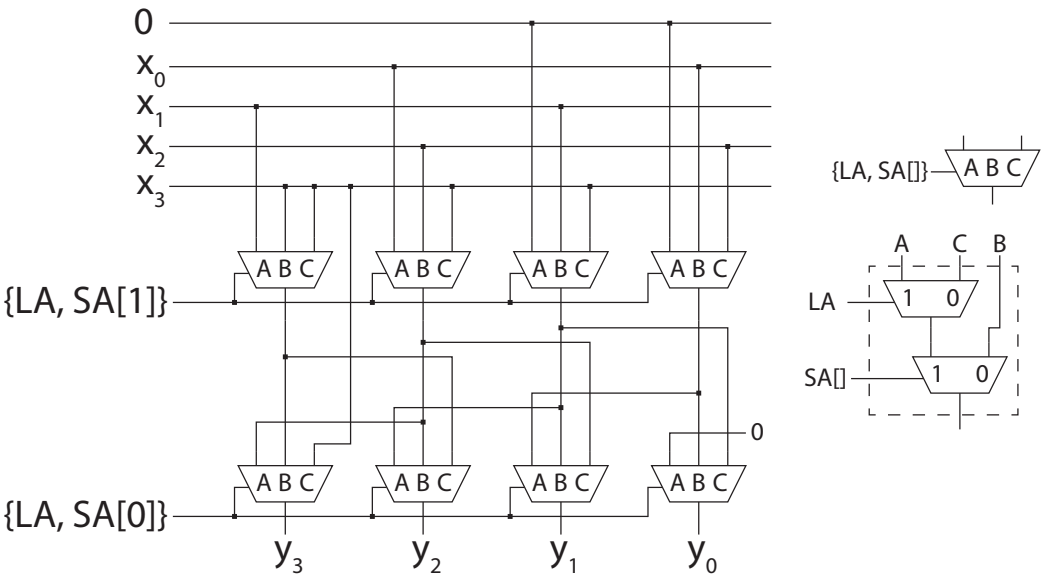
There are multiple possible solutions for implementing this shifter out of 2 MUXs and logic gates. Two possible solutions are presented here and are based on the Log Shifter design.

The first version, shown below, constructs a left shifter and a right shifter which are then selected between. The two gray multiplexers can be optimized out.



The next version uses 3 MUXs which are constructed out of 2 MUXs. The advantage of this

version is that fewer multiplexers are used. However, the depth is larger resulting in more delay.



Problem 7: FSM Design with Counters [15 pts]

Suppose you wish to design the controller for a packet processor in the interface to a wireless network. A message packet comes into your circuitry bit serially, from left to right as defined by the following packet format:

| preamble (128-bits) | header (32 bits) | body (1024 bit) |

The job of your controller is to split off the three fields of the incoming packets, feeding them each into external (to your controller) shift registers to hold each one. The three external shift registers each have a SE (shift enable) input. These shift enable inputs are named `PRE_se`, `HDR_se`, and `BDY_se`, for preamble, header, and body, respectively.

Your block has these inputs: `Din`, `DV` (data valid from the wireless interface, asserted on a per clock cycle basis), and `RST`.

After reset `DV` will be asserted on each cycle where `Din` holds a valid input. The bits of the packet are passed in serially starting with the preamble. Outputs from your block should be the three enable signals.

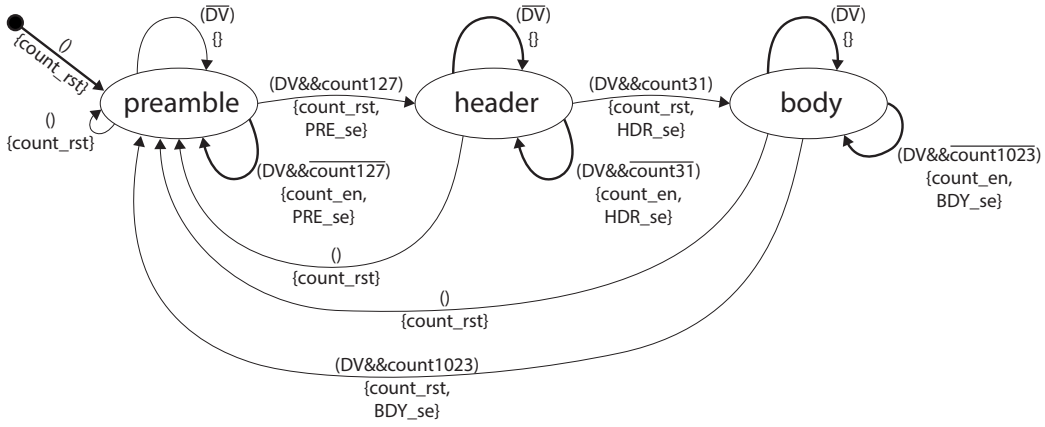
Design this controller using a FSM and counter(s). You don't need to show the detailed design of the counters, but show the details of the FSM circuitry. To keep the FSM implementation simple, you might want to consider using a one-hot encoded state machine.

Solution:

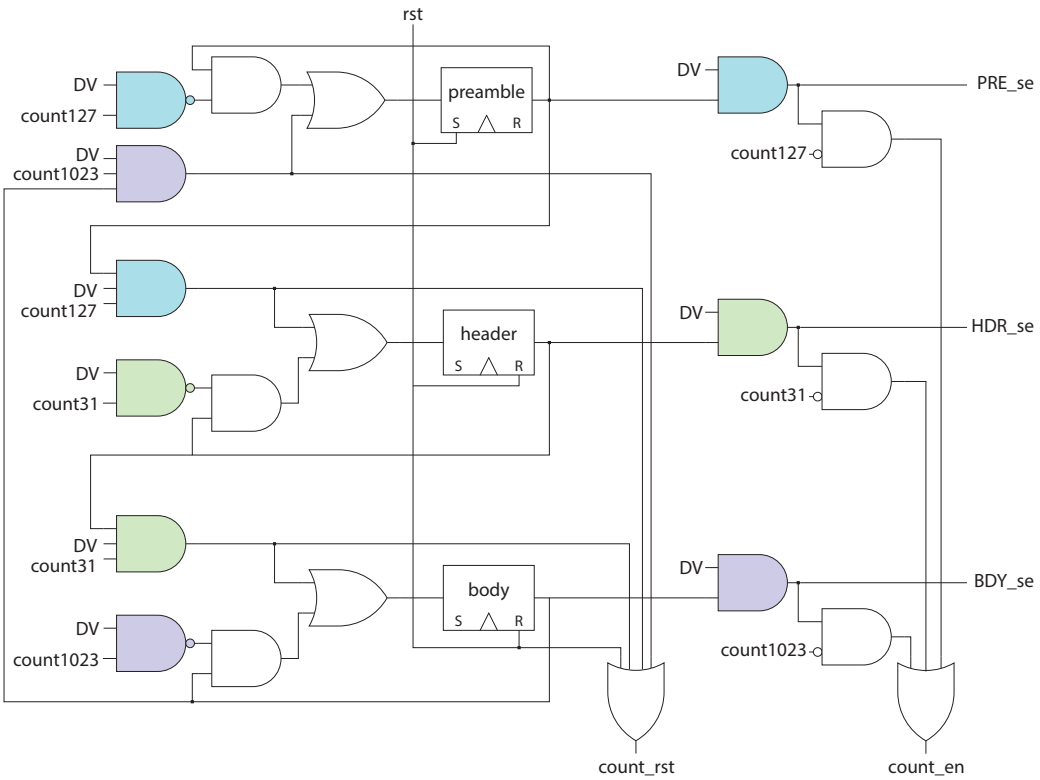
For this problem, we will assume all shift register inputs are wired to `Din`. We will also use a

single 10 bit counter that has outputs when the count reaches 31, 127, and 1023. Since these are all powers of 2 minus 1, this can be accomplished using a parallel prefix AND tree. These outputs of the counter will be referred to as `count31`, `count127`, and `count1023` respectively. This counter has an enable signal `count_en` that causes the counter to increment on the next positive edge of the clock. It also has a synchronous reset signal `count_rst` that resets the counter back to 0 on the next positive edge of the clock.

The State Transition Diagram of the FSM is shown below. All outputs are 0 except when listed. When listed, the outputs have a value of 1.



A one hot implementation of this FSM is shown below. Some of the gates can be merged together (gates shaded in the same color) but are shown separately for clarity.



Problem 8: Counter Design [15 pts]

(a) Write a Verilog generator that builds an AND-reduction parallel prefix tree for N inputs, where N is always a power of 2. Note that there are many possible structures for parallel prefix. However, for this problem, please implement the the "alternative parallel prefix circuit" shown in the lecture notes under synchronous counters, and also sometimes referred to as "Kogge-Stone" and shown in the lecture on fast adder circuits.

(b) Using that module write the a Verilog generator to define an N-bit binary counter, inputs are CE, CLK, and outputs are Q[N-1:0], and TC.

Here are some hints on writing the generator: The compile time function, $clog2 is used to take the log of a parameter. For instance `localparam levels = $clog2(N);` Also, the Verilog compile time operator "**", as in a**b, is equivalent to pow(a, b) in the C language.

Solution:

Parallel Prefix

```
module PrefixAnd #(parameter N=16) //Assuming N is a power of 2
  ( input [N-1:0] In,
   output [N-1:0] Out);
```

```
  localparam levels = $clog2(N);
  //$clog2 is function that was introduced in Verilog-2005 and may not be
  //supported by all EDA tools.
  //See https://www.xilinx.com/support/answers/44586.html for a workaround

  wire [N-1:0] array [levels:0]; //Need an extra entry to hold the

  assign array[0] = In;

  genvar i;
  genvar j;

  generate
    for(i = 1; i<=levels; i = i+1) begin:rows
      //Pass complete columns
      //Note: a**b is pow(a, b) in C
      for(j = 0; j<2**(i-1); j = j+1) begin:pass_cols
        assign array[i][j] = array[i-1][j];
      end

      //And remaining columns
      for(j = 2**(i-1); j<N; j = j+1) begin:calc_cols
        assign array[i][j] = array[i-1][j] & array[i-1][j-(2**(i-1))];
      end
    end
  endgenerate

  assign Out = array[levels];

endmodule


module Counter #(parameter N=16) //Assuming N is a power of 2
  ( input clk,
           input ce,
   output [N-1:0] Q,
   output tc);

  //See the "Synchronous Counters" slide of
  //http://inst.eecs.berkeley.edu/~eecs151/sp19/files/lec23-count-shift.pdf

  reg [N-1:0] count = 0;

  wire [N:0] ppInput;
  assign ppInput = {count, ce};

  wire [N:0] ppOutput;
```

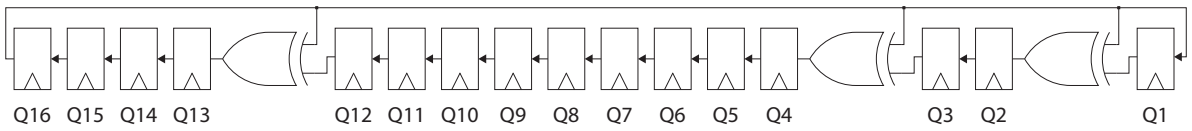
```
  PrefixAnd #(N+1) prefixAnd (.In(ppInput), .Out(ppOutput));

  wire [N-1:0] next_count = count ^ ppOutput[N-1:0];

  always @(posedge clk) begin
    count <= next_count;
  end

  assign Q = count;
  assign tc = ppOutput[N];

endmodule
```

Problem 9: LFSR [4 pts]

Draw the circuit for a 16-bit LFSR.

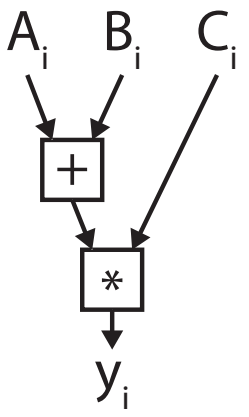
Solution:

To form a 16 bit LFSR, you need to use a primative polynomial of order 16. One such polynomial was given in lecture: $x^{16} + x^{12} + x^3 + x + 1$. This primitive polynomial was used to create the 16 bit LFSR shown below:



Problem 10: Modulo Scheduling [12 pts]

The following graph represents one iteration of a repeating computation: $y_i = (A_i + B_i) * C_i$. A_i, B_i, and C_i are stored in memory and must be read before they can be used. The result, y_i, must also be stored back into memory. The target hardware platforms for this algorithm include registers that are necessary for pipelining and scheduling. Mem read, write, *, and + each take 1 clock cycle to complete.



Using modulo scheduling, draw the characteristic section schedule with subscripts for the following hardware platforms:

(a) 2 read ports, 1 write port, 1 adder, and 1 multiplier

(b) 3 read ports, 1 write port, 1 adder, and 1 multiplier

(c) 2 read ports, 1 write port, 1 adder, and 1 pipelined multiplier (2 cycles per multiply)

(d) 3 read ports, 1 write port, 1 adder, and 1 pipelined multiplier (2 cycles per multiply)

Solution:

a: 2 read ports, 1 write port, 1 adder, and 1 multiplier

RD0 RD1	A_i B_i	C_i
WR0		$(A_{i-1} + B_{i-1})C_{i-1}$
$+$		$A_i + B_i$
$*$	$(A_{i-1} + B_{i-1})C_{i-1}$	

The characteristic section cannot be reduced further due to a lack of available read ports.

b: 3 read ports, 1 write port, 1 adder, and 1 multiplier

RD0 RD1 RD2	A_i B_i C_i
WR0	$(A_{i-3} + B_{i-3})C_{i-3}$
$+$	$A_{i-1} + B_{i-1}$
$*$	$(A_{i-2} + B_{i-2})C_{i-2}$

This is perfect pipelining!

c: 2 read ports, 1 write port, 1 adder, and 1 pipelined multiplier (2 cycles per multiply)

RD0 RD1	A_i B_i	C_i
WR0	$(A_{i-2} + B_{i-2})C_{i-2}$	
+		$A_i + B_i$
* pt 1 * pt 1	$(A_{i-1} + B_{i-1}) *_{pt1} C_{i-1}$	$(A_{i-1} + B_{i-1}) *_{pt2} C_{i-1}$

The characteristic section cannot be reduced further due to a lack of available read ports.

d: 3 read ports, 1 write port, 1 adder, and 1 pipelined multiplier (2 cycles per multiply)

RD0 RD1 RD2	A_i B_i C_i
WR0	$(A_{i-4} + B_{i-4})C_{i-4}$
+	$A_{i-1} + B_{i-1}$
* pt 1 * pt 2	$(A_{i-2} + B_{i-2}) *_{pt1} C_{i-2}$ $(A_{i-3} + B_{i-3}) *_{pt2} C_{i-3}$

This is perfect pipelining! Note that this is only possible because the multiplier is pipelined.