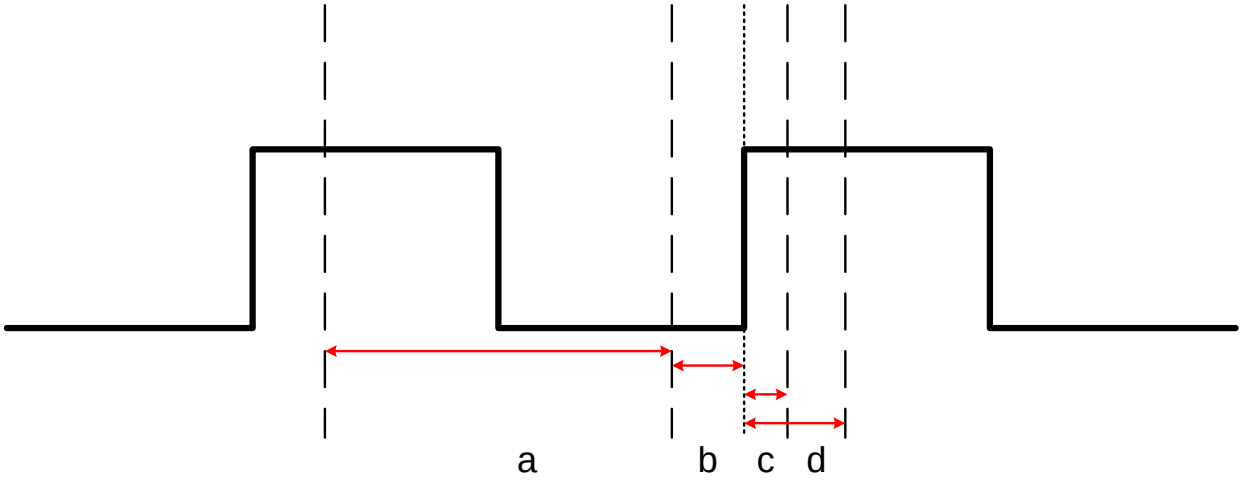
Registers

Timing

Consider the diagram shown below:



1. Given the above diagram, label:
    a.
    b.
    c.
    d.

Hint: the possible answers are **setup time**, **hold time, clock-to-Q time**, and **maximum logic delay**. If multiple time frames can be labeled with multiple terms, list them all for completeness.

Now let's see if you really understand the concepts!

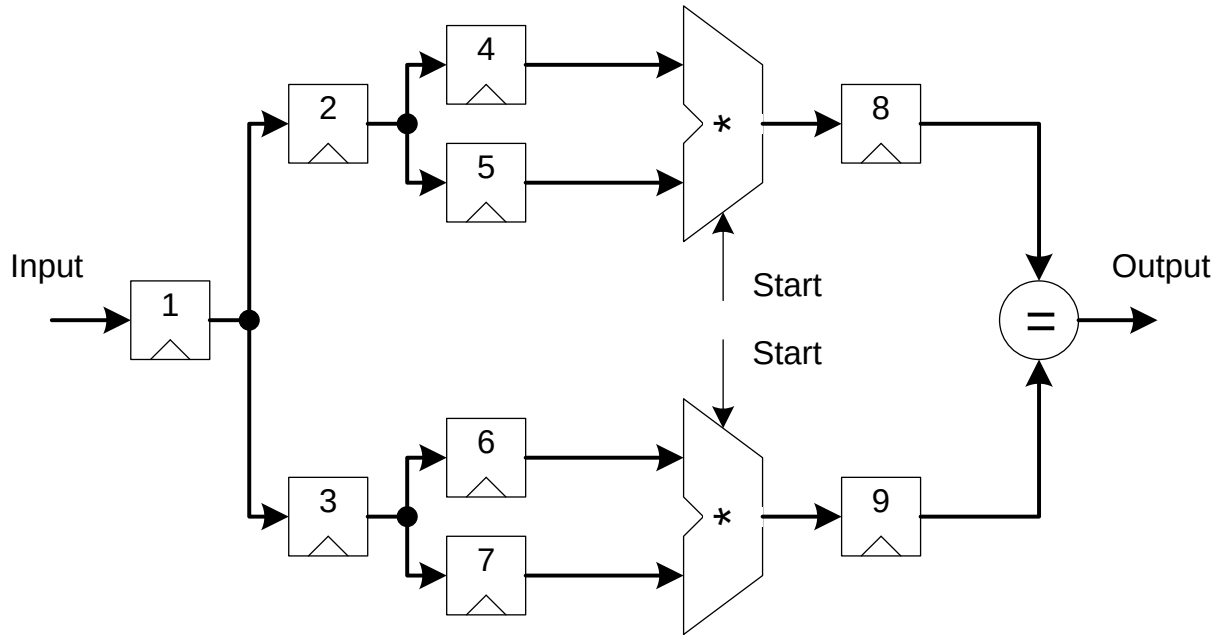
2. If I were to tell you that your register had the following characteristics:
    a. **Negative** setup time
    b. **Negative** hold time

    How would you describe what was going on?

    Note: Both of these notions do in fact exist; the Virtex-5 FPGAs that you have been using have negative setup times!

Design Optimizations

Consider the system below:



We want to use the above system to calculate A*B == A*C, given 3 values A, B, and C. **In truth, '*' isn't the ideal operator for this problem because we can reduce the entire computation down to B==C. For the sake of the problem, assume '*' performs some arbitrary operation such that the result of the equality check cannot possibly be determined by looking at just one operand going into each "multiplier" block.** You may make the following assumptions:

1. Each register can fit exactly one value at one time and that each has **Reset** and **Enable** signals.
2. The multiply ('*') blocks take 4 cycles (the values on registers 4, 5, 6, 7 must be steady while a multiply takes place) after "Start" is asserted. If "Start" is asserted starting right after the rising edge, the cycle it goes high in counts as cycle 1.
3. The delay through the equality ('=') block is 0.

Please answer the following questions:

1. Assuming time starts at a rising edge (count the first cycle as cycle 1), what is the fastest that the A*B == A*C can complete? (I.E. by what cycle #?). Assume that you can assert one of A, B, or C on the "Input" line at a given time.
   To help organize this problem, you may want to consider listing what value is held by "Input" over what time ranges and what the control signals (Reset, Enable) for each register are set to over different periods of time.

2. We want to optimize this system. What registers can we remove, while maintaining functionality?

Verilog

For a comprehensive list of topics that will be covered, relating to Verilog, see the Midterm information document on the website.

Verilog → RTL

Consider the following Verilog statement:

```
module complete_mystery(clk, we, a, di, d0);

       parameter     Width = 4; // MUST be a power of 2
       localparam    MemWidth = 16,
                     MemDepth = 64,
                     AWidth = `log2(MemDepth)+`log2(Width);

       input clk;
       input we;
       input [AWidth-1:0] a;
       input [MemWidth-1:0] di;
       output d0;
       wire [(Width*MemWidth)-1:0] memD0;
       genvar i;

       generate for (i = 0; i < Width; i = i + 1)
              begin:bit
mystery unit(.clk(  clk),
              .we(   we & (a[AWidth-1:`log2(MemDepth)] == i)),
              .a(    a[5:0]),
              .di(   di),
              .d0(   memD0[((i+1)*MemWidth)-1:i*MemWidth]));
              end
       endgenerate

       assign d0 = ~|memD0;

endmodule

// NOTES:
      The bugs with complete_mystery have been fixed, but the
circuit is still kind of odd function-wise.  The most
important thing to take away from this problem is how the
memories are being cascaded, and what is going on with the 'a'
line into each memory (hint: separate the 'a' line into two
parts: a part that indexes into a single memory (low bits) and
```

a part that chooses which memory we are indexing into (high bits)).

The `log2() macro can be found in Const.v, and does what you'd think it does. I said 'Width' had to be a power of 2 because depending on Const.v (and there are multiple versions out there… the non-power-of-2 case will either work or not work. Let's assume it doesn't for this problem.

```
module mystery (clk, we, a, di, d0);

	localparam MemWidth = 16, MemDepth = 64;

	input clk;
	input we;
	input [5:0] a;
	input [MemWidth-1:0] di;
	output [MemWidth-1:0] d0;
	reg [MemWidth-1:0] ram [MemDepth-1:0];
	reg [`log2(MemDepth)-1:0] read_a;

	always @(posedge clk) begin
		if (we) ram[a] <= di;
		read_a <= a;
	end

	assign d0 = ram[read_a];

endmodule
```

Draw the circuit that this Verilog generates. Make sure to include all relevant specifications to memories that you draw, including memory type, synchronous/asynchronous, reads/writes, etc.

Verilog Stumpers

1. always@(*) vs. always@(posedge …)

2. wires vs. regs

3. FSMs in Verilog

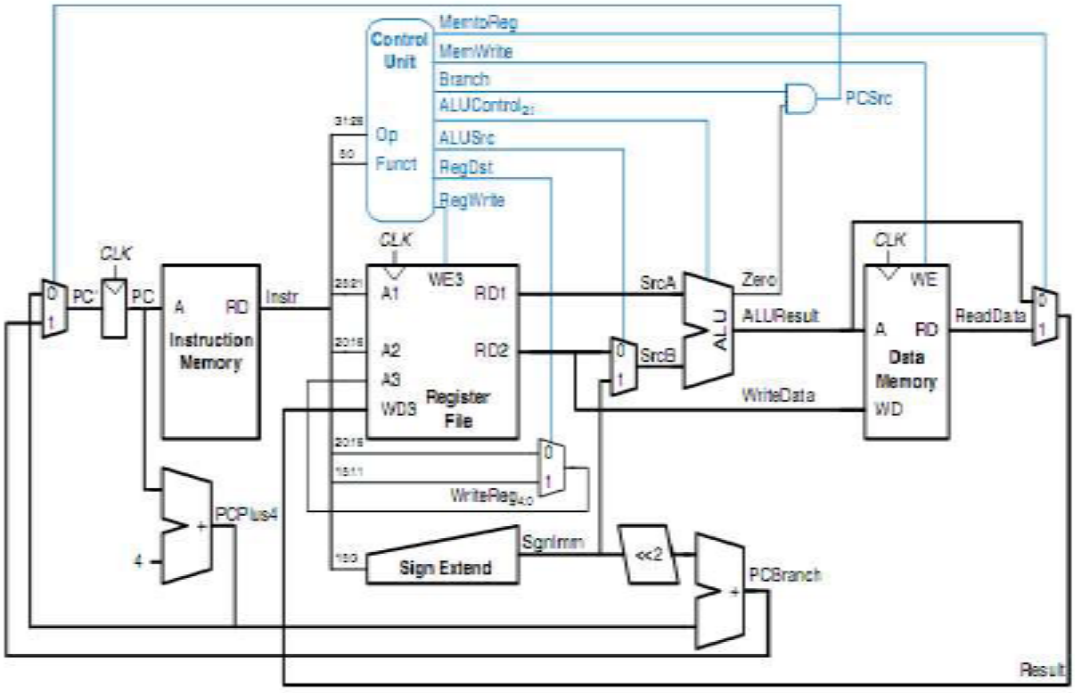
4. How can you generate latches?

5. parameters / generate

Processors

Adding to a Datapath

1. Consider the single-cycle datapath (which should look familiar) below:



Now consider the instruction: **XAdd r_d, r_s**

XAdd performs the following operation:

temp ← M[r_d] + r_s
r_s ← M[r_d]
M[r_d] ← temp

Add whatever elements/control signals are necessary to implement the XAdd instruction. **As an extra challenge: try your best to minimize the number of cycles that it takes to implement the complete XAdd instruction.**

2. What can we do to remove the load delay slot? In other words, we want to be able to execute this code without stalling the machine or stipulating a delay instruction:

```
lw $1, 0($sp)
add $2, $1, $1
```

You may add as much additional hardware as you need to complete this problem. Be sure, however, to add paths/control signals to whatever hardware you add so that your pipeline still works.