

Name _____

Formatted For Two-Sided Printing

Digital Design using HDLs
LSU EE 4755
Final Examination
Wednesday, 8 December 2021 7:30 CST

Problem 1 _____ (30 pts)

Problem 2 _____ (35 pts)

Problem 3 _____ (15 pts)

Problem 4 _____ (20 pts)

Alias _____

Exam Total _____ (100 pts)

Good Luck!

Problem 1: [30 pts] For the modules in this problem input `sample` holds a new value each cycle, and output `r_avg` holds the average of the last `n_samples` inputs. (Ignore the fact that the module needs but lacks a reset.)

(a) For the module below show the hardware that will be inferred when instantiated with default parameters. Be sure to optimize for the default value of `n_samples`.

```

module ravg2 #( int w = 8, n_samples = 4 )
  ( output logic [w-1:0] r_avg,
    input uwire [w-1:0] sample,  input uwire clk );

  logic [w-1:0] samples[n_samples];

  parameter int wm = $clog2( n_samples );
  parameter int ws = w + wm;
  logic [ws-1:0] tot;

  always_ff @( posedge clk ) begin

    samples[0] <= sample;

    for ( int i=1; i<n_samples; i++ ) samples[i] <= samples[i-1];

    tot <= tot - samples[n_samples-1] + samples[0];

  end

  always_comb r_avg = tot / n_samples;

endmodule

```

Show hardware for the module above using default parameter values.

(b) The module to the right is similar to `ravg3` except that it has three arithmetic unit instantiations: an adder, a subtractor, and a divide-by-constant unit. Modify `ravg3` so that it uses these modules. For full credit connect them so that the critical path passes through at most one module per cycle. In a correct solution `r_avg` will arrive at the output of `ravg3` later than it would in module `ravg2`.

- Modify `ravg3` so that it uses the three arithmetic units.
- For full credit, the critical path can go through at most one arithmetic unit per cycle.
- The connections to the arithmetic units can be changed (say from `aa1` to something else).

```

module ravg3 #( int w = 8, n_samples = 4 )
  ( output logic [w-1:0] r_avg,
    input uwire [w-1:0] sample,
    input uwire clk );

  logic [w-1:0] samples[n_samples];

  parameter int wm = $clog2( n_samples );
  parameter int ws = w + wm;
  logic [ws-1:0] tot;

  always_ff @( posedge clk ) begin

    samples[0] <= sample;

    for ( int i=1; i<n_samples; i++ ) samples[i] <= samples[i-1];

    tot <= tot - samples[n_samples-1] + samples[0]; // Modify or eliminate this line.

  end

  always_comb r_avg = tot / n_samples;

  uwire [ws-1:0] sum, diff;

  uwire [ws-1:0] aa1, aa2, da1;

  uwire [w-1:0] quot;

  uwire [w-1:0] sa1, sa2;

  our_adder #(ws,ws)      add1( sum,      aa1,      aa2 );

  our_sub #(ws,w)        sub2( diff,     sa1,     sa2 );

  our_div_by #(w,ws,n_samples) div3( quot,  da1 );

endmodule

```

Problem 2: [35 pts] Appearing below is a Verilog description of a lower-cost version of the bit_keeper module from Homework 4 and a diagram of the hardware.

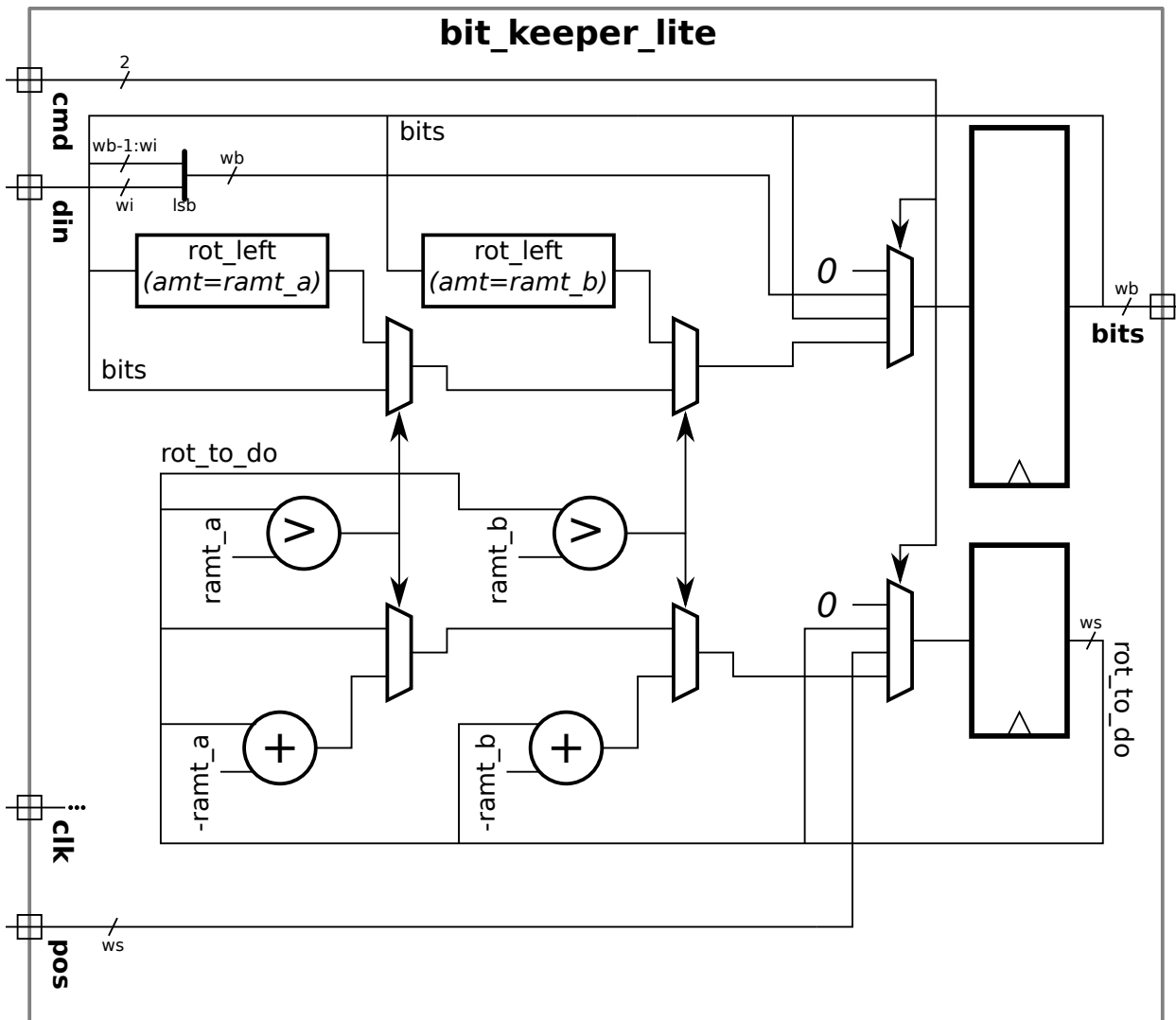
```
typedef enum { Cmd_Reset=0, Cmd_Write=1, Cmd_Rot_To=2, Cmd_Nop=3, Cmd_SIZE } Command;
module rot_left #( int w = 10, amt = 1 )
  ( output uwire [w-1:0] r, input uwire [w-1:0] a );
  assign r = { a[w-amt-1:0], a[w-1:w-amt] };
endmodule
module bit_keeper_lite #( int wb = 64, wi = 8, ws = $clog2(wb) )
  ( output logic [wb-1:0] bits, output uwire ready,
    input uwire [1:0] cmd, input uwire [wi-1:0] din,
    input uwire [ws-1:0] pos, input uwire clk );
  localparam int ramt_a = 1; // Specify Rotation Amounts
  localparam int ramt_b = 1 << ( ws >> 1 );
  uwire [wb-1:0] ra, rb;
  rot_left #(wb,ramt_a) r1l(ra,bits);
  rot_left #(wb,ramt_b) r18(rb,bits);
  logic [ws-1:0] rot_to_do; // Remaining amount of rotation to do.

  assign ready = rot_to_do == 0;
  always_ff @( posedge clk ) case ( cmd )
    Cmd_Reset: begin bits = 0; rot_to_do = 0; end
    Cmd_Rot_To: rot_to_do = pos; // Initialize rotation. Rotate during Nop.
    Cmd_Write: bits[wi-1:0] = din;
    Cmd_Nop: // Continue Executing a Cmd_Rot_To
      if ( rot_to_do >= ramt_b ) begin
        bits = rb; // Use output of larger rot module.
        rot_to_do -= ramt_b; // Decrement remaining rot amt.
      end else if ( rot_to_do >= ramt_a ) begin
        bits = ra; // Use output of smaller rot module.
        rot_to_do -= ramt_a; // Decrement remaining rot amt.
      end
  endcase
endmodule
```

(a) Find the cost and delay of the illustrated hardware using the simple model. Take into account the presence of constants. For the addition and comparison units assume a ripple implementation. Show any assumptions made. (See the next part before solving this one.)

Show cost in terms of w_b , w_i , and w_s .

Show delays on the diagram, and highlight the critical path. Delays should be in terms of w_b , w_i , and w_s .



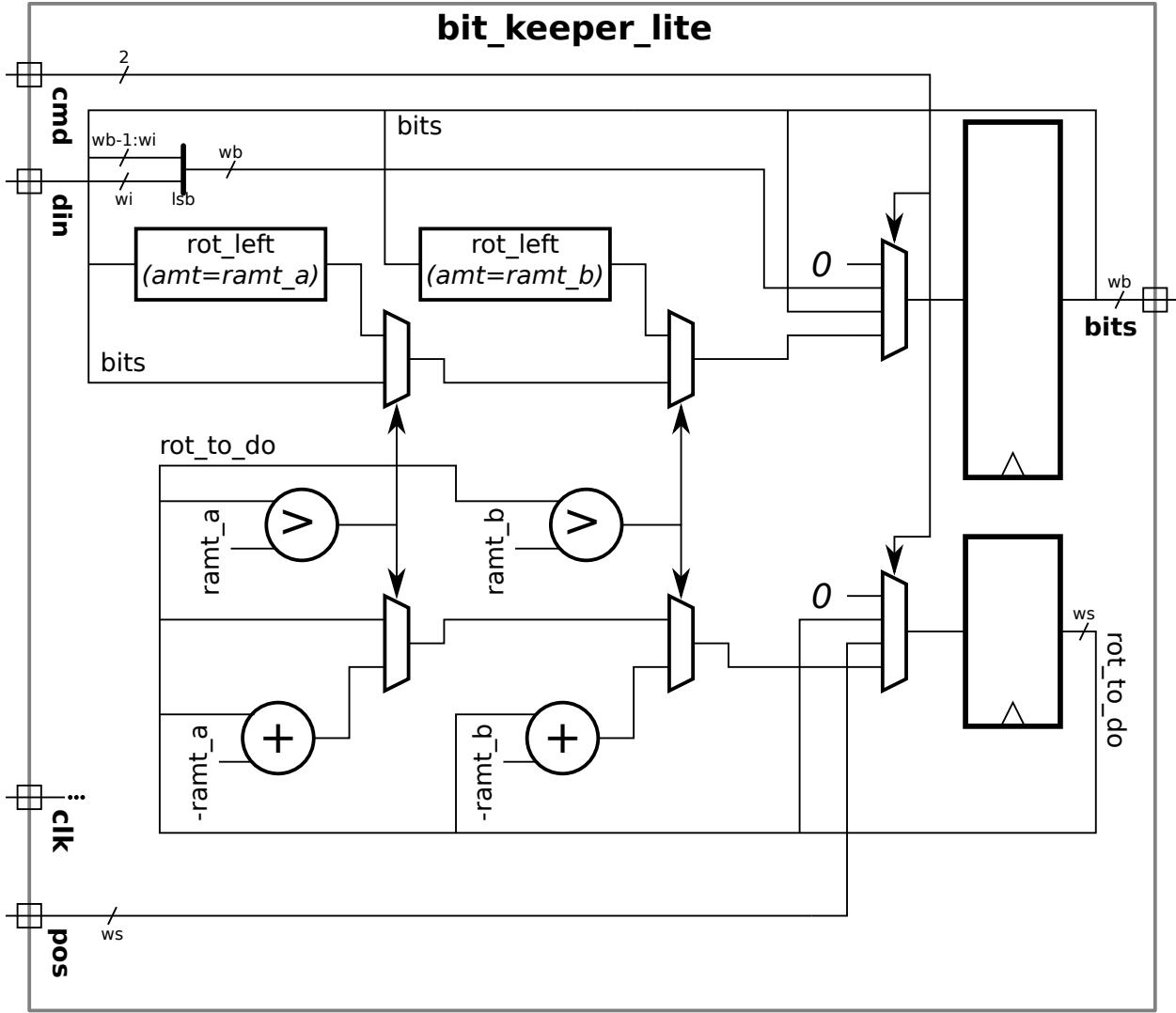
(b) In class we assume that a four-input mux is implemented using a reduction tree of 3 two-input muxen. For the illustrated hardware that would have a longer delay than is necessary. Modify the diagram on the right to show a better way of implementing the four-input multiplexors.

Replace four-input multiplexors with two-input muxen connected to reduce delay.

(c) Notice that care was taken to ensure that `ramt_b` is a power of 2. Explain how the fact that `ramt_b` is a power of two reduces the cost of the adder and comparison unit operating on `ramb_b`. Also explain how a power-of-2 `ramb_b` can reduce the cost of the other adder and comparison unit, if the synthesis program is clever enough. *Hint: Consider the binary representation of `rot_to_do`.*

Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_b` are lower cost because:

Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_a` (yes, a) are lower cost because:



(d) Appearing below is a version of `bit_keeper_lite` with four ready outputs, `r1`, `r2`, `r3`, and `r4`. On the diagram add hardware that will be synthesized for each.

```

module bit_keeper_lite #( int wb = 64, wi = 8, ws = $clog2(wb) )
  ( output logic [wb-1:0] bits,    output uwire r1, output logic r2, r3, r4,
    input uwire [1:0] cmd,        input uwire [wi-1:0] din,
    input uwire [ws-1:0] pos,     input uwire clk );

  localparam int ramt_a = 1;
  localparam int ramt_b = 1 << ( ws >> 1 );

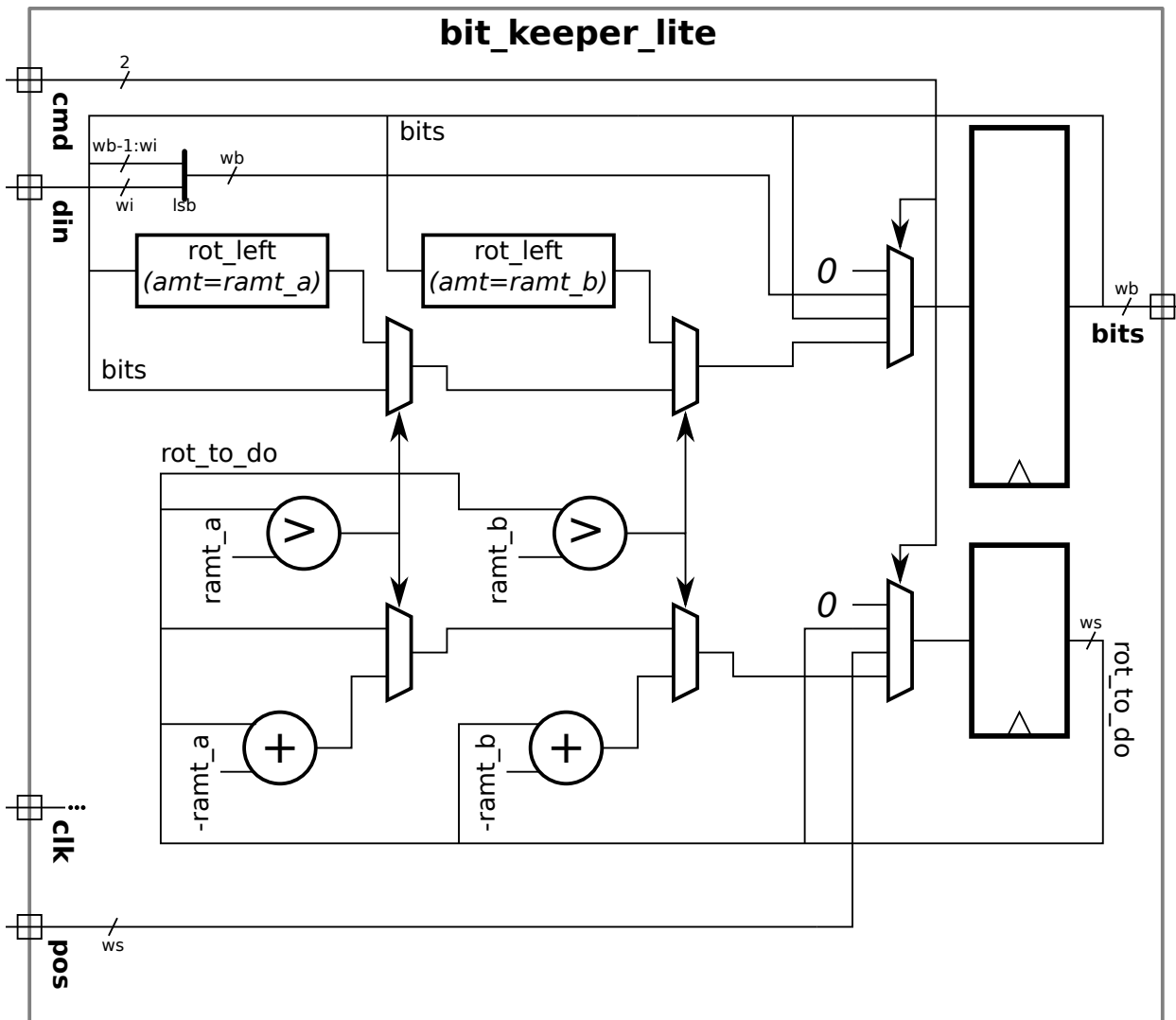
  uwire [wb-1:0] ra, rb;
  rot_left #(wb,ramt_a) r1l(ra,bits);
  rot_left #(wb,ramt_b) r18(rb,bits);

  logic [ws-1:0] rot_to_do;
  assign r1 = rot_to_do == 0;      // [ ] Show hardware for r1.

  always_ff @( _posedge clk ) begin
    r2 = rot_to_do == 0;          // [ ] Show hardware for r2.
    case ( cmd )
      Cmd_Reset: begin bits = 0; rot_to_do = 0; end
      Cmd_Rot_To: rot_to_do = pos;
      Cmd_Write: bits[wi-1:0] = din;
      Cmd_Nop: begin
        if ( rot_to_do >= ramt_b ) begin
          bits = rb;
          rot_to_do -= ramt_b;
        end else if ( rot_to_do >= ramt_a ) begin
          bits = ra;
          rot_to_do -= ramt_a;
        end
        r3 = rot_to_do == 0;      // [ ] Show hardware for r3.
      end
    endcase
    r4 = rot_to_do == 0;        // [ ] Show hardware for r4.
  end
endmodule

```

Show hardware that will be synthesized for `r1`, `r2`, `r3`, and `r4`.



Problem 3: [15 pts] Consider the modules below.

```
module ba
  ( output logic [15:0] next_x, next_y, x, y,
    input uwire [15:0] a, c, input uwire clk );

  always_ff @( posedge clk ) x = next_x;
  assign next_x = a;
  assign next_y = x + c;
  always_ff @( posedge clk ) y = next_y;

endmodule

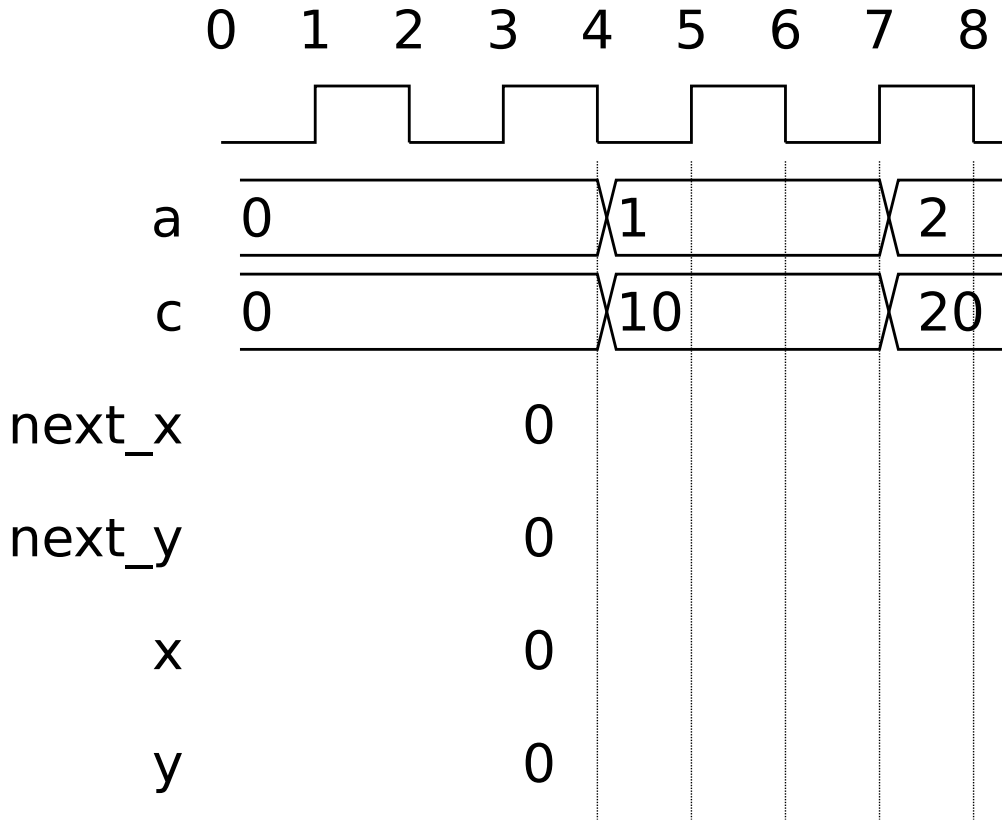
module test_ba;

  uwire [15:0] x, y, next_x, next_y;
  logic [15:0] a, c;
  logic clk;

  ba ba1( next_x, next_y, x, y, a, c, clk );

  initial begin
    // t = 0
    clk = 0;
    a = 0; c = 0;
    #1; // t = 1
    clk = 1;
    #1; // t = 2
    clk = 0;
    #1; // t = 3
    clk = 1;
    #1; // t = 4
    clk = 0; a = 1; c = 10;
    #1; // t = 5
    clk = 1;
    #1; // t = 6
    clk = 0;
    #1; // t = 7
    clk = 1; a = 2; c = 20;
    #1; // t = 8
    clk = 0;
  end

endmodule
```



(a) Complete the timing diagram so that it shows the values of `next_x`, `next_y`, `x`, and `y` that would be produced with the modules below.

- Complete timing diagram from $t = 4$ to $t = 6.9$. Note that there is a **negative** clock edge $t = 4$.

(b) At $t = 5$ we can be sure that `y=next_y` will execute before `next_y=x+c`. Explain how this ordering is assured by the rules for the event queue.

- Explain how event queue regions assure `y=next_y` executes before `next_y=x+c` at $t = 5$.

(c) The order of execution is ambiguous at $t = 7$ (but is okay before then).

- Would non-blocking assignments `x <= next_x` and `y <= next_y` remove the ambiguity? Explain.

Problem 4: [20 pts] Answer each question below.

(a) The foolish `sqrt` module below has several issues.

```
module sqrt #( int w = 16 )
  ( output logic [w-1:0] r, input uwire [w-1:0] a );

  always_comb begin

    r = 0;
    while ( r * r < a ) r++;

  end

endmodule
```

Explain why, due to the Verilog rules for bit widths, the expression `r * r < a` won't compute the intended result.

Why is the `sqrt` module likely not synthesizable?

What would be the problem with the hardware if it were synthesizable?

(b) Consider the two division modules below. In the first `a2` is a parameter, in the second it is a module port. Use the `div_demo` module for your answers to the questions below.

```
module our_div_by
  #( int wq = 5, wd = 10, logic [wd-1:0] a2 = 4 )
  ( output uwire [wq-1:0] quot, input uwire [wd-1:0] a1 );
  assign quot = a1/a2;
endmodule

module our_div
  #( int wq = 5, wd = 10 )
  ( output uwire [wq-1:0] quot, input uwire [wd-1:0] a1, a2 );
  // cadence inline
  assign quot = a1/a2;
endmodule

module div_demo
  #( int w = 21 )
  ( output uwire [w-1:0] d1, d2,
    input uwire [w-1:0] x1, x2, x3, x4 );

  localparam logic [w-1:0] y1 = 4755;
```

```
endmodule
```

- Show an instantiation of `our_div` for which `our_div_by` would not work.
- Explain how the use of the `cadence inline` pragma in `our_div` makes it possible to instantiate `our_div` in places that otherwise might need `our_div_by`.

(c) Answer the following questions about latency and throughput.

Define latency.

Define throughput.

Consider a sequential circuit (such as `mult_step` from Homework 6) and a pipelined version of the sequential circuit (such as `multi_step_pipe`). Assume that both have the same clock frequency.

Compared to the sequential version, does the pipelined version typically have
 lower latency , *the same latency* , or *higher latency* . Explain.

Compared to the sequential version, does the pipelined version typically have
 lower throughput , *the same throughput* , or *higher throughput* . Explain.

Compared to the sequential version, does the pipelined version typically have
 lower cost , *the same cost* , or *higher cost* . Explain.

