Name Solution _____ *Formatted For Two-Sided Printing*

Staple This Side

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)
Exam Total	_____	(100 pts)

Alias Good Luck JWST! _____

Good Luck!

Staple This Side

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
```

Solution on next page.

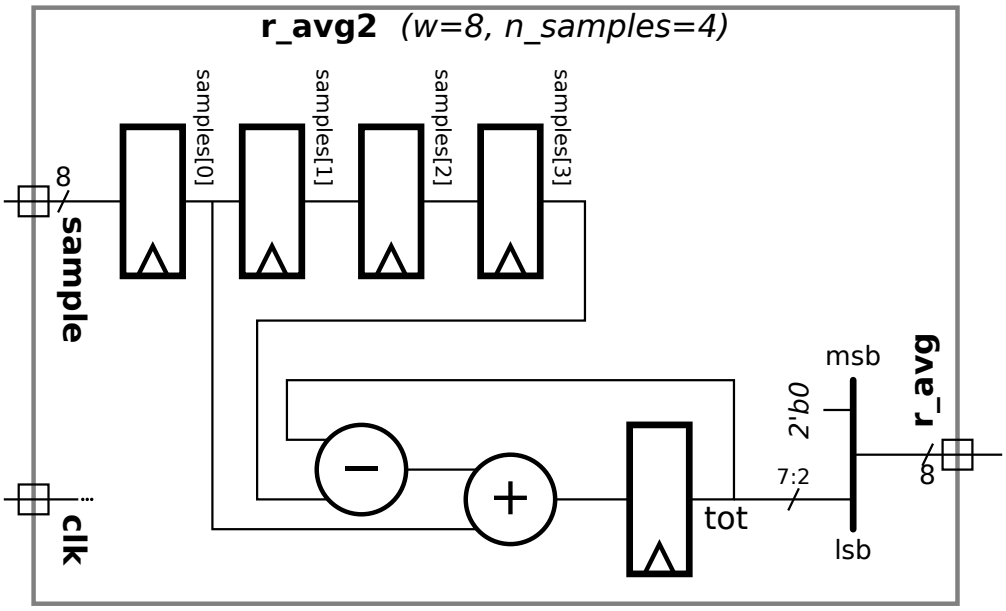
Staple This Side

☑ Show hardware for the module above using default parameter values.

☑ Optimize for these parameter values.

Solution shown below. Notice that because non-blocking assignments were used to assign `samples[i]`, the computation of `tot` uses the register outputs. In particular `samples[0]` is the register output, which is the value of `sample` from the previous cycle.

Because `n_samples=4` is a power of 2, the division, `tot/n_samples`, can be done by shifting right by two bits. Since the shift is constant just use bits `7:2` of `tot` and place two bits of zero in the MSB of the output.



(*b*) The module to the right is similar to `ravg2` 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).

☑ Do not add unnecessary cost or delay.

Solution appears below.

Please be sure to understand the following important points.

So that the critical path passes through at most one arithmetic module, the inputs to the arithmetic modules cannot connect to arithmetic module outputs. Instead, they connect to registers, such as `tot` and `samples[0]`.

So that the running sum is correct, the values of `samples[0]` and `samples[n_samples-1]` must be used in the same cycle. For that reason the subtractor is used to compute `samples[0] - samples[n_samples-1]`. It would not be correct to compute `diff = tot - samples[n_samples-1]` in one cycle and `tot = diff-samples[0]` in the next cycle because `samples[0]` is the wrong value.

Notice that `samples[0]` was directly connected to the subtractor input. That's more convenient than using an intermediate variable, say `sa1`.

Staple This Side

```
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;

   // SOLUTION - Declare a register to hold output of subtractor.
   logic [ws-1:0] pl_diff;

   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.

      // SOLUTION - Write output of subtractor and adder into registers.
      pl_diff <= diff;
      tot <= sum;

   end

   // always_comb r_avg = tot / n_samples;               // Modify or eliminate this line.

   // SOLUTION - Remove unneeded declarations. (aa1, etc.)
   uwire [ws-1:0] sum, diff;

   // SOLUTION - Use subtract to compute samples[0] - samples[n_samples-1]
   our_sub #(ws,w) sub2( diff,   samples[0],  samples[n_samples-1]   );

   // SOLUTION - Use adder to compute new value of tot.
   our_adder #(ws,ws) adder1( sum, tot, pl_diff );

   // SOLUTION - Use divider to compute r_avg.
   our_div_by #(w,ws,n_samples) div3( r_avg,     tot );

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_Rot_To=1, Cmd_Write=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) rl1(ra,bits);
   rot_left #(wb,ramt_b) rl8(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. ☑ Take into account constants.

The hardware consists of registers, multiplexors, adders, comparison units, and constant shifters.

Shifters: Since they shift by a constant amount the total shifter cost is zero.

Registers: The cost of a w-bit register is $7w\,\mathrm{u_c}$. There are two registers, `bits` and `rot_to_do`. There sizes are w_b and w_s, so their combined cost is $7(w_b + w_s)\,\mathrm{u_c}$.

Two-Input Multiplexors: The cost of a w-bit, 2-input mux is $3w\,\mathrm{u_c}$. In the illustrated hardware there are two w_b-bit 2-input muxen and two w_s-bit 2-input muxen. (None of their inputs are constant.) Their total cost is $[2 \times 3\,w_b + 2 \times 3\,w_s]\,\mathrm{u_c} = 6(w_b + w_s)\,\mathrm{u_c}$.

Four-Input Multiplexors: A w-bit four-input mux can be constructed from three 2-input muxen, and so its cost would is $3 \times 3\,w\,\mathrm{u_c} = 9w\,\mathrm{u_c}$. The cost of a w-bit, 2-input mux with a constant data input is $w\,\mathrm{u_c}$. Each of the four-input muxen has a

Staple This Side

constant data input, reducing the cost to $(2 \times 3 + 1)w\,\mathrm{u_c} = 7w\,\mathrm{u_c}$. The total cost of the two four-input muxen accounting for the constant input is $7(w_b + w_s)\,\mathrm{u_c}$.

Adders: An ordinary w-bit ripple adder costs $9w\,\mathrm{u_c}$. A w-bit ripple adder with one constant input costs $4w\,\mathrm{u_c}$. The two adders each have one constant input. Based on just that their costs are $4 \times 2w_s\,\mathrm{u_c}$. But the value of `ramt_b` is $2^{w_s/2}$, and so the $w_s/2$ least-significant bits of `ramt_b` are zero. That means the adder passes those low bits through unchanged, reducing the adder cost to just $w_s/2\,\mathrm{u_c}$. Looking at the `ramt_a` adder in isolation one would have to conclude that its cost is $4w_s\,\mathrm{u_c}$ with `ramt_a=1`. But the output of the adder is ignored if `rot_to_do>ramt_b` meaning that we can assume the input to the `ramt_a` adder is no greater than `ramt_b` and so we only need a $w_s/2$-bit adder. With both of those optimizations the total adder cost is $2 \times 4\frac{w_s}{2}\,\mathrm{u_c} = 4w_s\,\mathrm{u_c}$.

Comparison Unit: Recall that a ripple comparison unit is constructed from the carry logic of ripple subtractor. The cost of a w-bit comparison is $4w\,\mathrm{u_c}$. But one constant input reduces the cost to just $w\,\mathrm{u_c}$. With no further optimizations the cost of the two comparison units is $2w_s\,\mathrm{u_c}$.

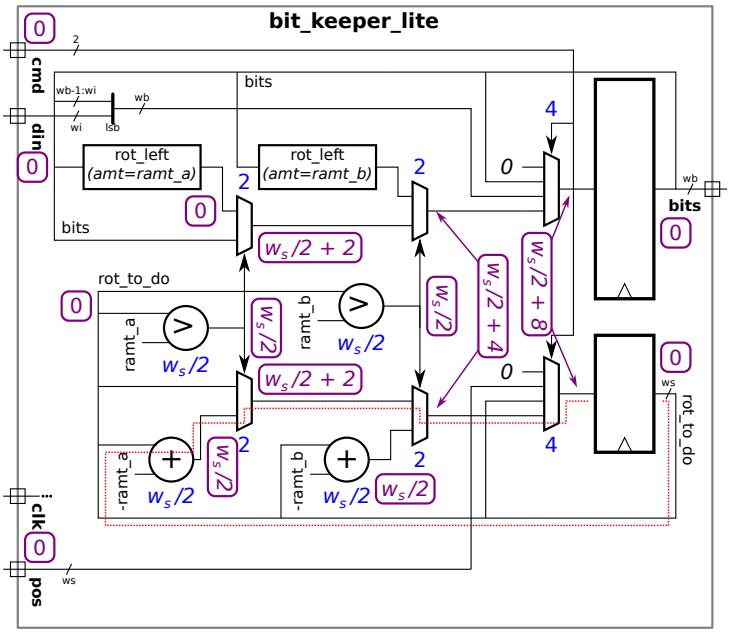
The `ramt_a` comparison is irrelevant if `rot_to_to` is greater than `ramt_b`, and so only $w_s/2$ bits need be examined. If the `ramb_b` comparison operation were $\geq$ then it could just examine $w_s/2$ bits. But since the operation is strictly greater than all bits must be considered. But using the output of the `ramt_a` comparison the `>ramt_b` comparison could be done by examining $w_s/2$ more bits. The total comparison cost is $2 \times \frac{w_s}{2}\,\mathrm{u_c} = w_s\,\mathrm{u_c}$.

☑ Show delays and arrival times on the diagram, and ☑ highlight the critical path. These should be in terms of w_b, w_i, and w_s.

The timings and critical path are shown on the diagram. Blue shows the delay through a component, such as 2 for two-input multiplexors. Circled times show the delay of the longest path starting at module inputs and register outputs. A critical path is shown as a red dotted line. Note that there are several critical paths in this circuit though only one is illustrated.

Multiplexor Delay: The delay of an ordinary two-input mux is $2\,\mathrm{u_t}$. If one input is constant the delay is $1\,\mathrm{u_t}$. The delay of an n-input mux is $\lceil \lg n \rceil 2\,\mathrm{u_t}$, which works out to $4\,\mathrm{u_t}$ for a four-input mux. The next sub-problem shows how that delay can effectively be reduced to $2\,\mathrm{u_t}$ on the critical path. The diagram to the right does not reflect that optimization.

Adder Delay: The delay of a w-bit ripple adder with a constant input is $w\,\mathrm{u_t}$. The timings in the diagram are based on $w_s/2$-bit adders.



Comparison Delay: The delay of a w-bit ripple comparison unit with a constant input is $w\,\mathrm{u_t}$. The timings in the diagram are based on $w_s/2$-bit comparison units.

(*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 result in a longer critical path 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 critical path.

Solution appears on the lower half of the next page. The four-input mux has been replaced by three two-input muxen, but not connected in a reduction tree. The benefit of this non-tree connection is that one of the inputs, the fourth as used here, has a delay of only $2\,\mathrm{u_t}$. That is the input that carries the critical path, and so the critical path delay is reduced by $2\,\mathrm{u_t}$.

(*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`.

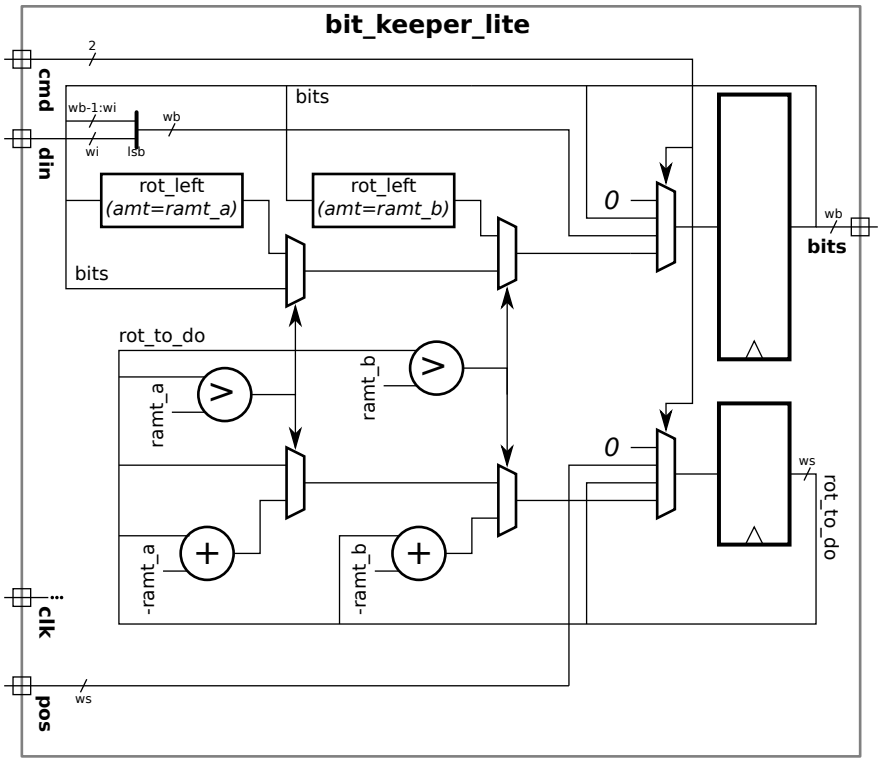
☑ Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_b` are lower cost because:

Because the lower $w_s/2$ bits of `ramt_b` are all zero. Because `ramt_b` is also a constant there is no need for an adder at all for the least significant $w_s/2$ bits.

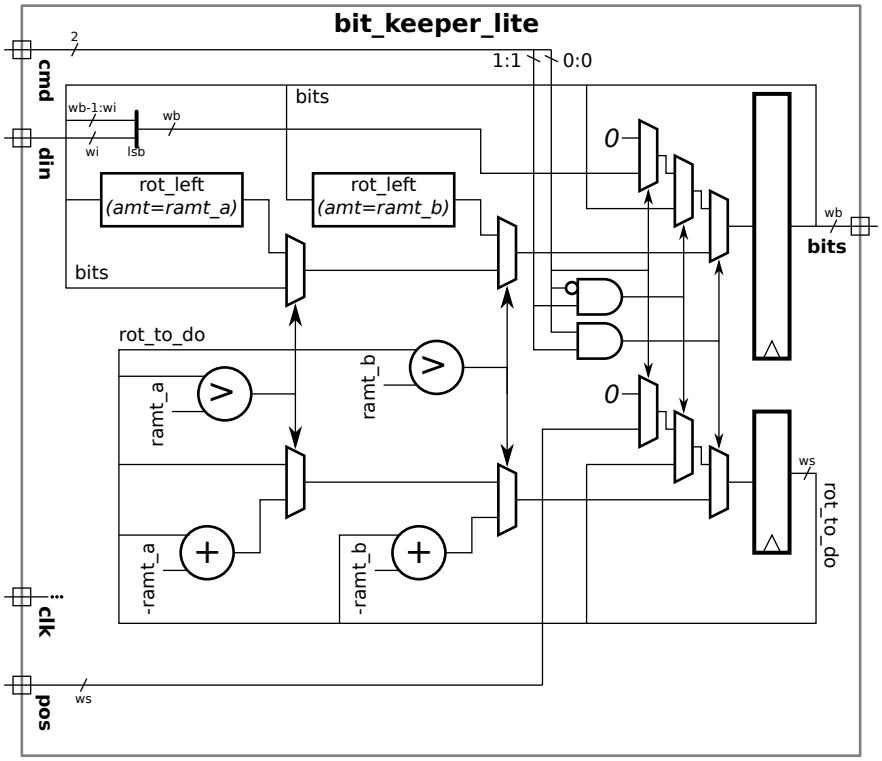
☑ Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_a` (yes, a) are lower cost because:

Because the output of the `ramt_a` adder is only used if `rot_to_do <= ramt_b`. Therefore there is no point in providing an adder that can handle more than $w_s/2$ bits. For the same reason the comparison unit need only consider the lower $w_s/2$ bits.

Staple This Side



Solution appears below.



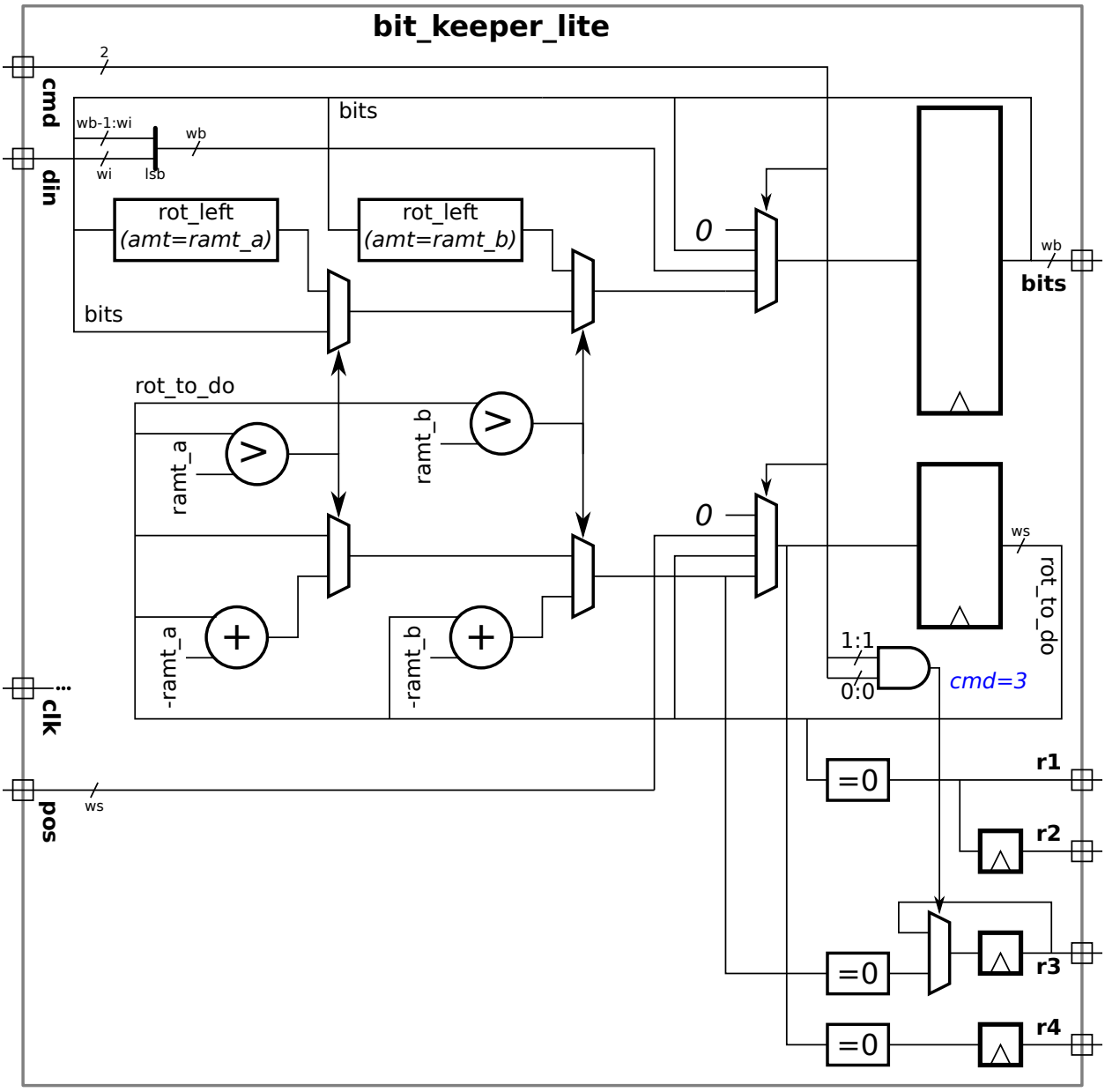
(*d*) Appearing below is a version of `bit_keeper_lite` with four ready outputs, `r1`, `r2`, `r3`, and `r4`. On the diagram add hardware that will be synthesized for each.

```
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) rl1(ra,bits);
   rot_left #(wb,ramt_b) rl8(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`.

Solution appears on the next page. Because they are assigned in an `always_ff`, the values of `r2`, `r3`, and `r4` visible outside the block come from registers. Pay close attention to where `rot_to_do` is assigned and where its value is referenced. For `r1` it is referenced outside of the `always_ff` block and so the value is from the register. The value of `rot_to_do` used for `r2` also comes from the register output because it had not been assigned yet in the block. For `r3` the value of `rot_to_do` assigned in the `cmd=Cmd_Nop` case is used. A mux keeps `r3` unchanged when `cmd` is not `Cmd_Nop`. (The value of enumeration constant `Cmd_Nop` is 3.) Finally, `r4` is assigned at the end of the block, so it uses the value of `rot_to_do` that will be written to the register.

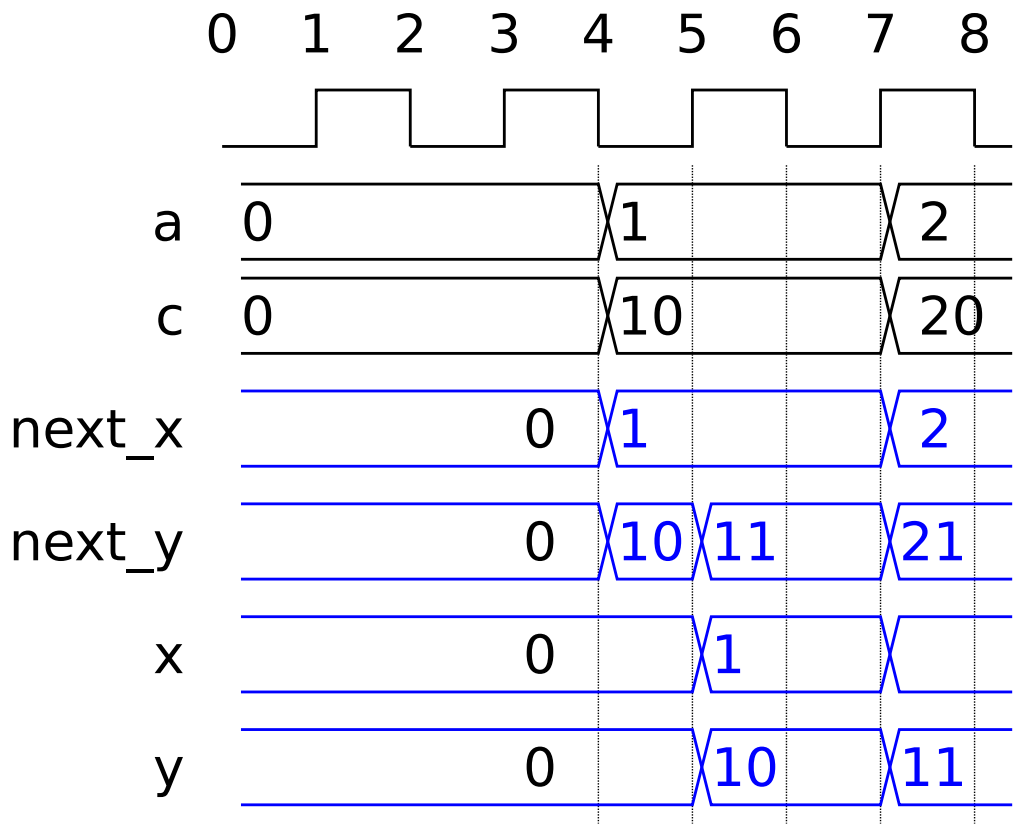
Staple This Side



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;  // Line t4
      #1; // t = 5
      clk = 1;
      #1; // t = 6
      clk = 0;
      #1; // t = 7
      clk = 1;  a <= 2;   c <= 20;  // Line t7
      #1; // t = 8
      clk = 0;
   end

endmodule
```


Staple This Side



(*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 above. *Note: In the original exam* `test\_ba` *did not use non-blocking assignments to* `a` *and* `c`*.*

☑ Complete timing diagram from $t = 4$ to $t = 8.1$. ☑ Note that there is a **negative** clock edge at $t = 4$.

Solution appears above.

(*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 $t = 5$.

At $t = 5$ `clk` changes from `0` to `1`, resulting in the two `always_ff` items being scheduled. The two will eventually reach the active region of the event queue, and one of them will be chosen first. Assume that the first `always_ff` is chosen first. The `next_y` assignment has `x` and `c` in its sensitivity list, and so it is only scheduled for execution when at least one of these changes. At $t = 5$ `x` changes, and that will result in the `next_y` assignment being placed in the inactive region of the event queue. The scheduler will continue to remove and execute events from the active region until the active region is empty. Therefore the second `always_ff` is guaranteed to execute before the `next_y` assignment.

(*c*) Notice that `a` and `c` are assigned using non-blocking assignments on Lines t4 and t7. Explain why the order of execution would be ambiguous at $t = 7$ if line t7 used blocking assignments: `a=1; c=10;`. *Note: This question was not in the original exam.*

☐ Describe ambiguity (more than one possible execution order) if blocking assignments were used.

☐ 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.

Because `r` and `a` are 16 bits the computation will be done to 16 bits of precision, and so due to overflow `r*r<a` can be false when it should be true.

☑ Why is the `sqrt` module likely not synthesizeable?

Because the maximum number of iterations of the `while` loop cannot be directly determined. The maximum number of iterations in fact will be about $2^{w/2}$, and it's not impossible that a synthesis program would figure that out. It's just not likely because this is not the typical loop that would be used to describe hardware.

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

The maximum number of iterations is $2^{w/2}$. For the default value that's $2^8 = 256$. There would need to be 256 multiply units, 256 comparison units, and 256 muxen. That's alot of hardware. And anyway there are much better ways of computing a square root.

Staple This Side

Staple This Side

(*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;

   // Could replace our_div with our_div_by because y1 is constant.
   our_div #(w,w) dwould_work(d1, x1, y1);

   // Could not replace our_div with our_div_by because
   // divisor (x2) not a constant.
   our_div #(w,w) dwould_not_work(d2, x1, x2);

endmodule
```

☑ Show an instantiation of `our_div` for which `our_div_by` could work.

☑ Show an instantiation of `our_div` for which `our_div_by` could not work.

Solution appears above. To use `our_div_by` the divisor needs to be a constant. That's the case in the first example, but not in the second example

☑ 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`.

It ensures that each instantiation of `our_div` will be optimized separately based on its arguments. Without the pragma the synthesis program might optimize `our_div` once, assuming two non-constant inputs, and then copy the optimized description to places where there are constant inputs.

Staple This Side

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

☑ Define latency.

Latency is the amount of time needed to compute a result from start to finish. What a result is depends on the context. The result might be computed combinationally, or sequentially over several cycles.

☑ Define throughput.

Throughput is the number of results computed per unit time. For example, if over 10 seconds 200 results are computed, the throughput is $200/10 = 20$ results per second.

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.

☑ Remembering that the clock frequencies are the same, compared to the sequential version, does the pipelined version typically have
◯ *lower latency*, ⊗ *the same latency*, or ⊗ *higher latency*. ☑ Explain.

It depends. In a reasonable design the latency of the sequential version will be equal to or possibly greater than the pipelined version. A sequential design can re-use hardware, and so if it prioritizes low cost it will use less hardware over a greater number of cycles resulting in a higher latency than a pipelined design.

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

By definition, a pipelined circuit computes a result each clock cycle, and so its throughput is high. A sequential circuit will require several cycles to compute something and so its throughput will be lower.

☑ **Ignoring the cost of registers**, compared to the sequential version, does the pipelined version typically have
◯ *lower cost*, ◯ *the same cost*, or ⊗ *higher cost*. ☑ Explain.

The sequential version re-uses units (such as arithmetic units) over multiple cycles. The pipelined version must have one unit for each operation, and so its cost will be higher.

Staple This Side

Staple This Side