LSU EE 4755 Homework 4 Solution Due: 11 November 2021

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2021/hw04.v.html`.

Problem 0: If necessary, follow the instructions at `https://www.ece.lsu.edu/koppel/v/proc.html` to set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Teamwork

Students can work on this assignment in teams. Each student should submit his or her own assignment but list team members. It is recommended that one team member be responsible for learning SimVision.

Every member of a team that has completed a project, must be capable of re-solving the problem. It is recommended that all team members re-solve the problem on their own for their own pedagogical benefit.

Problem 1: Module `bit_keeper` has a w_b-bit output `bits` ($_b$ is for width of buffer) and a 1-bit output `ready`. Think of output `bits` as a long bit vector (w_b bits long) that is edited using the module's inputs. Commands to edit `bits` are given using four-bit input `cmd` (command), w_i-bit input `din` (data in), and w_s-bit input `pos` (position). The module is to operate sequentially using input `clk`.

Complete `bit_keeper` as described below, and make sure that it is synthesizable. As always, code should be written clearly, and designs should not be costly or slow.

When completed `bit_keeper` should operate as follows. On a positive edge of `clk` action is taken based on the value of `cmd`. The possible values of `cmd` are: `Cmd_Reset`, `Cmd_None`, `Cmd_Write`, and `Cmd_Rot_To`. (These can be used as constants in your code. The constants are defined by `enum Command`.) Some commands will be complete in one cycle (the cycle in which the `cmd` is set up to the positive edge of `clk`). Other commands will take multiple cycles.

Be sure to understand the details of how multi-cycle commands execute. When a multi-cycle command starts the `ready` output must be set to zero and must be held at zero until the command completes. The command and its arguments will only be held at the inputs **for one cycle**, and so at the next positive clock edge they will be gone. The `cmd` input will be set to `Cmd_Nop`, and the `pos` and `din` inputs will be set to random values. This means that the inputs of multi-cycle commands that will be needed in subsequent cycles must be saved in registers.

The testbench can emit a trace of commands and their effects. This trace is used below to illustrate what the module is supposed to do. The trace is collected after the command completes. A trace entry starts with the word `Cycle`. The cycle number is shown, followed by command details, followed by the state of bits.

For `Cmd_Reset` output `bits` should be set to zero. Also, any internal registers should be set to zero. The command should complete at the positive edge. This should set `ready` to 1. In the trace below the reset command set bits back to zero. Notice that the command completes in one cycle (based on the cycle numbers).

```
Cycle 307 -- test 73: Cmd_Nop                    : bits = 01401f4
Cycle 308 -- test 74: Cmd_Reset                  : bits = 0000000
```

For `Cmd_Rot_To` the value in `bits` must be rotated so that the contents of `bits[0]` is moved to `bits[pos]`, `bits[1]` is moved to `bits[(pos+1)%wb]`, and so on. This is like a left shift of `pos` bits, except that the most significant `pos` bits of `bits` are rotated into the the `pos` least significant bits. In the trace below the rotate command rotates four bits (one hexadecimal digit). Notice that the most-significant digit on the first line is rotated to the least significant digit after the rotation command.

```
Cycle 301 -- test 71: Cmd_Nop                    : bits = 401401f
Cycle 306 -- test 72: Cmd_Rot_To pos 4           : bits = 01401f4
```

This rotation **must be performed** using two instances of module `rot_left`. One instance should rotate by 1, the other rotates by a larger value, call it r_b, of your choosing. Each clock cycle the value of `bits` is rotated using one of these, but never both in the same clock cycle. Use the r_b-bit rotate instance until the number of bit positions to shift is $\leq r_b$, then use the 1-bit rotate instance.

Command `Cmd_Write` has two forms based on the value of input `pos`. If `pos` is zero then the least significant w_b bits of `bits` should be written with `din`. This should complete at the positive edge. Otherwise, bits `pos` through `pos+wi-1` of `bits` should be written with `din`—but not directly. Instead, `bits` should be rotated so that bit `pos` is at the least-significant position, then the data should be written, then `bits` should be rotated back to its original position. Use only the two `rot_left` instances.

The trace below shows a write with `pos=0`:

```
Cycle 417 -- test 86: Cmd_Nop                    : bits = 0000240000
Cycle 418 -- test 87: Cmd_Write  pos 0, data 7   : bits = 0000240007
```

When `pos` is non-zero the writes take longer:

```
Cycle  96 -- test 20: Cmd_Nop                    : bits = 0a0000003c
Cycle 107 -- test 21: Cmd_Write  pos 27, data 4  : bits = 0a2000003c
```

No action is needed for command `Cmd_Nop`. In fact, this is the command that will be present while the external hardware, including the testbench, is waiting for other commands to complete.

The testbench will test `bit_keeper` at two sizes. At each size detailed information is given for the first few errors. That includes a trace of commands leading up to the error, followed by the erroneous command, and what the `bits` should have been. After each error the testbench sets its shadow value of `bits` to the erroneous output so that subsequent tests can pass. Here is in example of the output:

```
Cycle  22 -- test  0: Cmd_Rot_To pos 20          : bits = 0000000000
Cycle  54 -- test  1: Cmd_Rot_To pos 31          : bits = 0000000000
Cycle  55 -- test  2: Cmd_Nop                    : bits = 0000000000
Cycle  96 -- test  3: Cmd_Write  pos 37, data 2  : bits = 4000000000
Cycle  97 -- test  4: Cmd_Nop                    : bits = 4000000000
Cycle 103 -- test  5: Cmd_Rot_To pos 5           : bits = 0000000008
Cycle 104 -- test  6: Cmd_Write  pos 0, data 3   : bits = 0000000003
Error in test  7: Cmd_Write  pos 1, data 2   : 0000000c04 != 0000000005 (correct)
```

For multi-cycle commands the testbench will wait for `ready` to go to zero and then back to one. If that does not happen after a certain number of cycles the testbench will *timeout*, meaning that it will give up waiting and print a `CYCLE LIMIT EXCEEDED` message. If there is a timeout while a command is in progress (meaning that `ready` did go to zero, but did not return to one) the testbench will show a trace of recent history, followed by an indication of what it was waiting for:

```
Exit from clock loop at cycle 16000, limit 16000, ** CYCLE LIMIT EXCEEDED **
```

```
** Preceding Commands **
Cycle   7 -- test  0: Cmd_Rot_To pos 20         : bits = 0000000000
Cycle  14 -- test  1: Cmd_Rot_To pos 31         : bits = 0000000000
Cycle  15 -- test  2: Cmd_Nop                   : bits = 0000000000

** In-Progress Command **
test  3: Cmd_Write  pos 37, data 2
 -- Awaiting ready = 1.
```

If the testbench does not timeout then it will print a tally of the number of errors after testing each `bit_keeper` instance. Also, as a measure of quality, the testbench reports the average number of cycles to perform `Cmd_Rot_To` and `Cmd_Write` (with non-zero `pos`). For example,

```
Starting tests for (wb=40,wi=4)
Finished 200 tests for (wb=40,wi=4), 0 errors.
Avg cyc  Cmd_Rot_To 5.5 (67)  Cmd_Write 10.6 (35)

Starting tests for (wb=28,wi=8)
Finished 140 tests for (wb=28,wi=8), 0 errors.
Avg cyc  Cmd_Rot_To 4.2 (57)  Cmd_Write 8.2 (18)
```

The lines starting `Avg cyc` report timing. The number in parentheses is the number of times the command was issued. So for the first set of tests `Cmd_Rot_To` was tried 67 times, and the average number of cycles taken to complete it was 5.5.

A lower number for `Avg cyc` can indicate a good design, or that certain rules were not followed.

It is very important that debugging tools are used. Take advantage of the testbench messages to see what is going wrong. Run SimVision to get a detailed look at what your module is doing.

The solution has been copied to the homework directory, and an htmlized version has been posted at `https://www.ece.lsu.edu/koppel/v/2021/hw04-sol.v.html`. For the discussion below the solution is shown in pieces, shorn of most comments. Following that is the complete solution. The solution starts by specifying rotate amounts for the two rotation modules, followed by their instantiation.

```
localparam int rot_amt_a = 1;
localparam int rot_amt_b = 1 << ( ws >> 1 );

uwire [wb-1:0] ra, rb;
rot_left #(wb,rot_amt_a) rl1(ra,bits);
rot_left #(wb,rot_amt_b) rl8(rb,bits);
```

The rotate amount of the first module is set to 1, but a localparam is used for its value. To minimize the number of rotations the rotate amount for the second module, `rot_amt_b`, should be set to the square root of `wb`. To minimize delay it should be set to a power of 2. Here it is set to a power of 2 close to the square root of `wb`.

Rotations are to be done over several cycles. As stated in the problem commands are presented at the inputs for just for one cycle, and are then replaced with a `Cmd_Nop` until the `ready` returns to 1. To remember what needs to be done three registers will be used, `rot_to_do`, `rot_to_return`, and `wval`. Register `rot_to_do` is set to the number of bits of rotation that still need to be done. For `Cmd_Rot_To` it is initialized to `pos` and for `Cmd_Write` with `pos!=0` it is initialized to `wb - pos`. Register `rot_to_return` is set to the amount of rotation needed after the write is performed. Register `wval` is the value to write.

The `ready` output is set to 1 when both `rot_to_do` and `rot_to_return` are both zero.

```
logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.
logic [ws-1:0] rot_to_return;  // Amount of rotation needed after write.
```

```
logic [wi-1:0] wval;            // Value to write.
assign ready = rot_to_do == 0 && rot_to_return == 0;
```

The main `always_ff` has just a single `case` statement. `Cmd_Reset` is straightforward:

```
always_ff @( posedge clk ) begin
   case ( cmd )

     Cmd_Reset: begin
        bits = 0;
        rot_to_do = 0;
        rot_to_return = 0;
     end
```

For `Cmd_Rot_To` the rotate amount is saved in `rot_to_do`. The work of rotating is done when `cmd` is `Cmd_Nop`.

```
     Cmd_Rot_To: begin rot_to_do = pos; end
```

What `Cmd_Write` does depends on `pos`. If it's zero the write is done immediately. Otherwise `rot_to_do` is set to an amount that will bring bit `pos` to the least-significant position. Variable `rot_to_return` is set to the rotation to use after the write completes, one which moves the least-significant bit back to where it was. Also, the write value is saved.

```
     Cmd_Write:
       if ( pos == 0 ) begin
          bits[wi-1:0] = din;
       end else begin
          rot_to_do = wb - pos;
          wval = din;
          rot_to_return = pos;
       end
```

The work of rotating is done when `cmd` is set to `Cmd_Nop`. If `rot_to_do` is non-zero (which means $\geq$ `rot_amt_a`) then `bits` is set to the output of the appropriate rotation module and `rot_to_do` is decremented. Note that the rotation being performed can be for one of three purposes: a `Cmd_Rot_To`, the rotation before a write, or the rotation after a write.

```
     Cmd_Nop: begin
        if ( rot_to_do >= rot_amt_b ) begin
           bits = rb;                // Use output of larger rot module.
           rot_to_do -= rot_amt_b;   // Decrement remaining rot amt.
        end else if ( rot_to_do >= rot_amt_a ) begin
           bits = ra;                // Use output of smaller rot module.
           rot_to_do -= rot_amt_a;   // Decrement remaining rot amt.
        end
      // More Cmd_Nop code below
```

Next, `Cmd_Nop` needs to check whether a write needs to be done now. (A write needs to be done if `rot_to_return` is non-zero and it needs to be done now if also `rot_to_do` is zero.) If so, the write is performed and `rot_to_do` is set so that `bits` is rotated back to its original position.

```
        if ( rot_to_do == 0 && rot_to_return !=0 ) begin
           bits[wi-1:0] = wval;
           rot_to_do = rot_to_return;
           rot_to_return = 0;
```

```
        end
```

The entire solution with more comments appears below.

Grading Notes: In many solutions there were three separate pieces of code to perform rotate: one used for `Cmd_Rot_To`, one used before a write, and one used after a write. That code duplication makes it harder for humans to read, and could also lead to more costly and slower designs.

```
module bit_keeper
  #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,
     output uwire ready,
     input uwire [3:0] cmd,
     input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,
     input uwire clk );

   /// SOLUTION

   // Specify Rotation Amounts
   //
   localparam int rot_amt_a = 1;
   localparam int rot_amt_b = 1 << ( ws >> 1 );
   //
   // To minimize the number of rotations, rot_amt_b should be set to
   // the square root of wb. But, to minimize delay it should be set
   // to a power of 2. Here it is set to a power of 2 close to the
   // square root of wb.

   // Instantiate Rotation Modules
   //
   uwire [wb-1:0] ra, rb;
   rot_left #(wb,rot_amt_a) rl1(ra,bits);
   rot_left #(wb,rot_amt_b) rl8(rb,bits);

   logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.
   logic [ws-1:0] rot_to_return;  // Amount of rotation needed after write.
   logic [wi-1:0] wval;           // Value to write.

   // The module is ready if there is no remaining rotation to do.
   //
   assign ready = rot_to_do == 0 && rot_to_return == 0;
```

```
always_ff @( posedge clk ) begin

   case ( cmd )

     Cmd_Reset: begin
        //
        // Perform Reset

        bits = 0;
        rot_to_do = 0;
        rot_to_return = 0;
     end

     Cmd_Rot_To: begin
        //
        // Set Amount of Rotation
        //
        // The rotation will be performed in subsequent cycles.

        rot_to_do = pos;
     end

     Cmd_Write:

       if ( pos == 0 ) begin
          //
          // Perform Write Immediately

          bits[wi-1:0] = din;

       end else begin
          //
          // Perform Write Later

          // Set amount of rotation needed before the write, ..
          //
          rot_to_do = wb - pos;
          //
          // .. save the value that will be written, ..
          //
          wval = din;
          //
          // .. and save the amount of rotation needed after the write.
          //
          rot_to_return = pos;

       end
```

```
        Cmd_Nop: begin
           //
           // Continue Executing a Cmd_Rot_To or Cmd_Write.

           // If necessary, set bits to a rotated value.
           //
           if ( rot_to_do >= rot_amt_b ) begin
              //
              // Still need to rotate by at least rot_amt_b bits.

              bits = rb;                // Use output of larger rot module.
              rot_to_do -= rot_amt_b;   // Decrement remaining rot amt.

           end else if ( rot_to_do >= rot_amt_a ) begin
              //
              // Still need to rotate by at least rot_amt_a (1) bit.

              bits = ra;                // Use output of smaller rot module.
              rot_to_do -= rot_amt_a;   // Decrement remaining rot amt.
           end

           // Check whether a write is pending and can now be performed.
           //
           if ( rot_to_do == 0 && rot_to_return !=0 ) begin
              //
              // Write value, and set amount of rotation to return to
              // original positioning.

              bits[wi-1:0] = wval;
              rot_to_do = rot_to_return;
              rot_to_return = 0;
           end

        end

      endcase

   end

endmodule
```