Lecture 5: Projects 1 and 2

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Fibonacci Calculator: 3 States

```
module fibonacci_calculator (input logic clk, reset_n,
                             input logic [4:0] input_s,
                             input logic begin_fibo,
                            output logic [15:0] fibo_out,
                            output logic done);
  enum logic [1:0] {IDLE=2'b00, COMPUTE=2'b01, DONE=2'b10} state;

  logic  [4:0] count;
  logic [15:0] R0, R1;

  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n) begin
      state <= IDLE;
      done <= 0;
    end else
      case (state)
        IDLE:
          if (begin_fibo) begin
            count <= input_s;
            R0 <= 1;
            R1 <= 0;
            state <= COMPUTE;
          end
        COMPUTE:
          if (count > 1) begin
            count <= count - 1;
            R0 <= R0 + R1;
            R1 <= R0;
          end else begin
            state <= DONE;
            done <= 1;
            fibo_out <= R0;
          end
        DONE:
          state <= IDLE;
      endcase
  end
endmodule
```

in clocked always stmts, D-FFs keep track of previous value, so the missing "else" part will just keep "state" at IDLE.

Fibonacci Calculator: 2 States

```
module fibonacci_calculator (input logic clk, reset_n,
                             input logic [4:0] input_s,
                             input logic begin_fibo,
                            output logic [15:0] fibo_out,
                            output logic done);
    enum logic {IDLE=1'b0, COMPUTE=1'b1} state;

    logic  [4:0] count;
    logic [15:0] R0, R1;

    assign done = (count == 1);
    assign fibo_out = R0;

    always_ff @(posedge clk, negedge reset_n)
    begin
          if (!reset_n) begin
              state <= IDLE;
              count <= 0;
          end else case (state)
              IDLE:
                    if (begin_fibo) begin
                        count <= input_s;
                        R0 <= 1;
                        R1 <= 0;
                        state <= COMPUTE;
                    end
```

no FFs will be generated

```
              COMPUTE:
                  if (count > 1) begin
                    count <= count - 1;
                    R0 <= R0 + R1;
                    R1 <= R0;
                  end else
                    state <= IDLE;
          endcase
      end
  endmodule
```


Fibonacci Calculator: No State

```
module fibonacci_calculator (input logic clk, reset_n,
                             input logic [4:0] input_s,
                             input logic begin_fibo,
                            output logic [15:0] fibo_out,
                            output logic done);
    logic  [4:0] count;
    logic [15:0] R0, R1;
    assign done = (count == 1);
    assign fibo_out = R0;

    always_ff @(posedge clk, negedge reset_n) begin
          if (!reset_n)
              count <= 0;
          else begin
              if (begin_fibo) begin
                    count <= input_s;
                    R0 <= 1;
                    R1 <= 0;
              end else if (count > 1) begin
                    count <= count - 1;
                    R0 <= R0 + R1;
                    R1 <= R0;
              end
          end
    end
endmodule
```

no FFs will be generated

Fibonacci Calculator: Logic Only

```
module fibonacci_calculator (input logic clk, reset_n,
                             input logic [4:0] input_s,
                             input logic begin_fibo,
                            output logic [15:0] fibo_out,
                            output logic done);

    function logic [15:0] myfibo (input logic [4:0] n);
        logic [15:0] F[0:31]; // can also be written as F[31:0] or F[32]
        begin
            F[0] = 0;
            F[1] = 1;
            for (int i = 2; i <= 31; i++) begin
                F[i] = F[i-1] + F[i-2];
            end
            myfibo = F[n]; // this will be implemented as 32:1 MUX
        end
     endfunction

    assign done = 1;
    assign fibo_out = myfibo(input_s);
endmodule
```

This logic simplifies to a 32:1 MUX with 5-bit "n" as selector.

Blocking vs. Nonblocking Assignment

- For `always_ff` statements, use nonblocking assignments (`<=`)
  - All nonblocking assignments happen together at the end after next clock tick. e.g.,

    ```
    always_ff@(posedge clk) begin
        …
        q <= q + 1;
    end
    ```

    a flip-flop gets created for "q", RHS (`q + 1`) is current value of the "q" flip-flop, and LHS (`q <= …`) means the "q" flip-flop gets updated on next clock tick.

- For `always_comb` statements, use blocking assignment (`=`)
  - All blocking assignments happen in order of appearance. e.g.,

    ```
    always_comb begin
        t = a & b;
        f = t & c;
    end
    ```

    a wire gets generated for "t" as output of the AND-gate `a & b`.

- Using or mixing blocking assignment (`=`) in `always_ff` or nonblocking assignment (`<=`) in `always_comb` possible, but **<u>not recommended</u>**.

Using Functions

- You can use a function to "encapsulate" a large chunk of logic statements using blocking assignments (=).
- Then you can instantiate the function from within an `always_ff` statement using nonblocking assignments (<=).

```
function logic [159:0] add4(input logic [31:0] a, b, c, d);
    ...
endfunction

always_ff @(posedge clk, negedge reset_n) begin
  if (!reset_n) begin
    ...
  end else begin
    ...
    f <= add4(a, b, c, d);
    ...
  end
end
```


RTL Example Revisited

```
module rtl_example(input logic clk, reset_n,
                   input logic [7:0] a, b,
                   output logic [7:0] f, count);

  // default encoding:
  // s0=2'b00, s1=2'b01, s2=2'b10
  enum logic [1:0] {s0, s1, s2} state;
  logic [7:0] c, t;

  function logic [7:0] myinc(input logic [7:0] x);
    logic [7:0] sum;
    logic c; // this is an internal c
  begin
    c = 1;
    for (int i = 0; i < 8; i++) begin
        sum[i] = x[i] ^ c;
        c = c & x[i];
    end
    myinc = sum;
  end
  endfunction

  always_comb
  begin
    t = c << 1;
    f = t + 1;
  end

  always_ff@(posedge clk, negedge reset_n)
  if (!reset_n) begin
    count <= 0;
    d <= 0;
    state <= s0;
  end else begin
    case (state)
    s0: begin
          count <= 0;
          d <= 0;
          state <= s1;
        end
    s1: begin
          count <= count + 1;
          d <= a + b;
          state <= s2;
        end
    s2: begin
          count <= myinc(count);
          d <= a - b;
          state <= s0;
        end
    endcase
  end

endmodule
```

Project 2: Byte Rotation

- The goal of this project is to learn about the memory model that we will be using for our remaining projects.

Byte Rotation

- Suppose each memory word is 32-bits, comprising of 4 bytes.
- Write back to memory with each memory word left-rotated by one byte.
- Example (word shown in hexadecimal):

  M[0] = 32'h**01**234567
  M[1] = 32'h**02**468ace

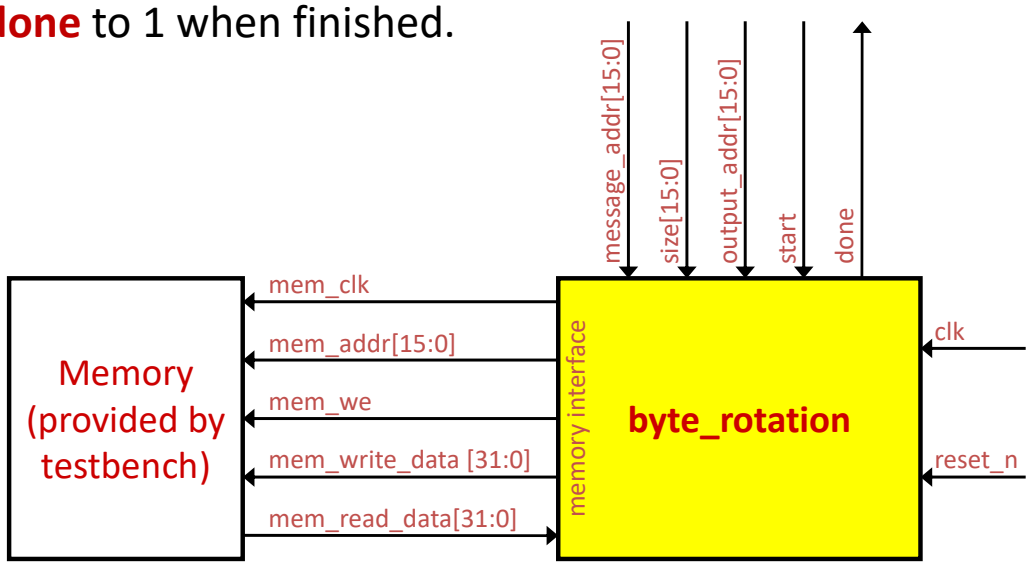
  Each "hexadecimal" digit is 4 bits. e.g., "67" means "0110_0111" (8 bits)

- Write back to memory with "most significant byte" left-rotated to the "least significant byte" position.

  M[100] = 32'h23456**701**
  M[111] = 32'h468ace**02**

Module Interface

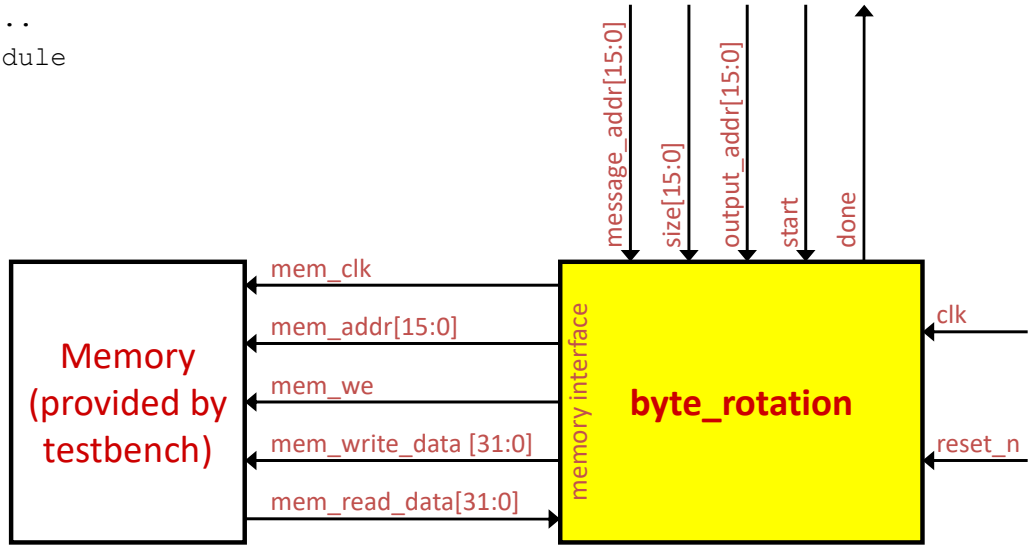
- Wait in idle state for **start**, read message starting at **message_addr**, left-rotate each word by one byte, and write output to memory starting at **output_addr**.
- The **message_addr** and **output_addr** are word addresses.
- The **size** is given in number of words.
- Set **done** to 1 when finished.



Module Interface

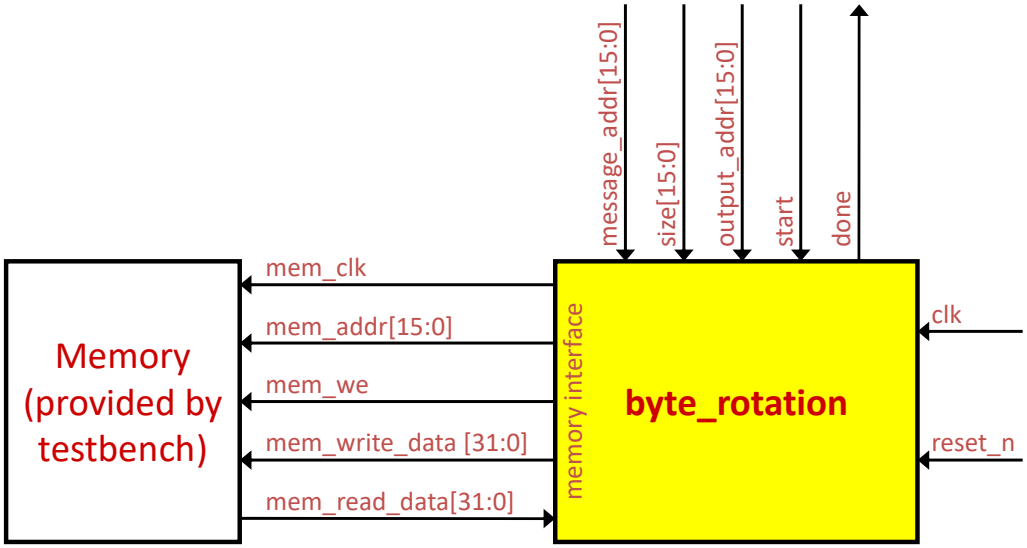
- Your assignment is to design the yellow box:

```
module byte_rotation(input logic clk, reset_n, start,
                     input logic [15:0] message_addr, size, output_addr,
                    output logic done, mem_clk, mem_we,
                    output logic [15:0] mem_addr,
                    output logic [31:0] mem_write_data,
                     input logic [31:0] mem_read_data);
    ...
endmodule
```



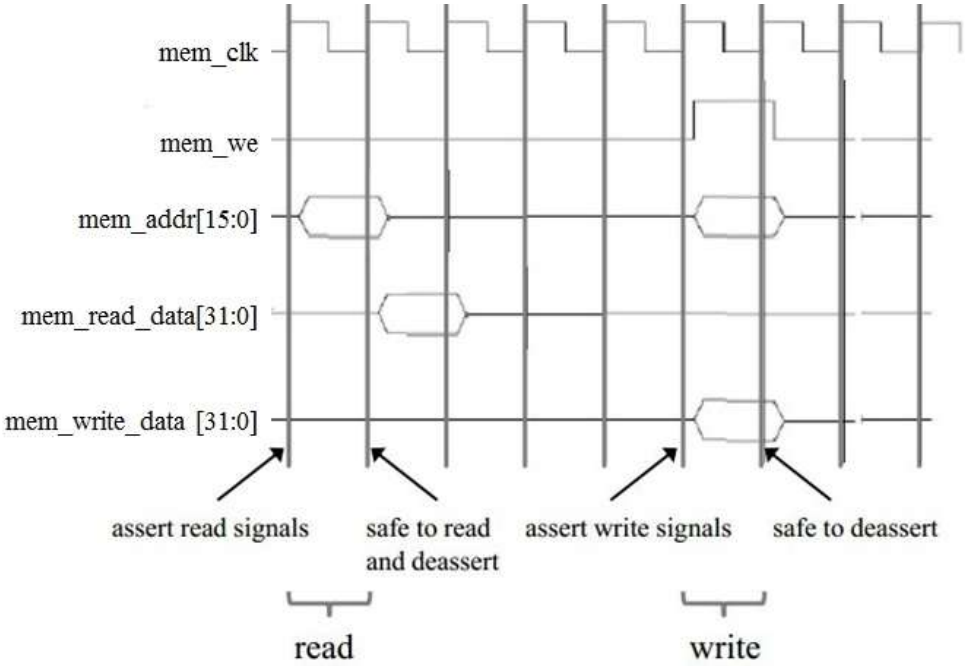
Memory Model

- To **read** from the memory:
  - Set **mem_addr** = 0x0000, **mem_we** = 0
  - At next clock cycle, read data from **mem_read_data**
- To **write** to the memory:
  - Set **mem_addr** = 0x0004, **mem_we** = 1, **mem_write_data** = data that you wish to write



Memory Model

- You can issue a new **read** or **write** command every cycle, **but** you have to wait for next cycle for data to be available on **mem_read_data** for a **read** command.
- **Be careful** that if you set **mem_addr** and **mem_we** inside **always_ff** block, compiler will produce flip-flops for them, which means external memory will not see the address and write-enable until another cycle later.



Be Careful

THIS IS INCORRECT

```
always_ff @(posedge clk, negedge reset_n) begin
   if (!reset_n) begin
      state <= S0;
   end else
      case (state)
        S0: begin
               mem_we <= 0;
               mem_addr <= 100;
               state <= S1;
        end
        S1: begin
               value <= mem_read_data; // data not yet available
               state <= S2;
        end
        ...
```


Be Careful

Have to wait an extra cycle

```
always_ff @(posedge clk, negedge reset_n) begin
   if (!reset_n) begin
      state <= S0;
   end else
      case (state)
        S0: begin
               mem_we <= 0;
               mem_addr <= 100;
               state <= S1;
        end
        S1: // memory only sees addr 100 in this cycle
               state <= S2;
        S2: begin
               value <= mem_read_data; // for addr 100
        ...
```

But we can pipeline the memory

```
case (state)
    S0: begin
            mem_we <= 0;
            mem_addr <= 100;
            state <= S1;
    end
    S1: begin
            mem_we <= 0;
            mem_addr <= 101;
            state <= S2;
    end
    S2: begin
            value <= mem_read_data; // for addr 100
            state <= S3;
    end
    S3: begin
            value <= mem_read_data; // for addr 101
            state <= S4;
    ...
```


Byte Rotation

- Use this function

```
function logic [31:0] byte_rotate(input logic [31:0] value);
    byte_rotate = {value[23:16], value[15:8], value[7:0], value[31:24]};
endfunction
```

- In your SystemVerilog code, you can do something like this:

```
mem_write_data <= byte_rotate(mem_read_data);
```

Project 2

- Use the provided “tb_byte_rotation.sv” testbench.
- Name your file “byte_rotation.sv”.

Your Design Should Produce This

- Testbench will internally generate a 16-word message:

This is shown in hexidecimal where “**01**” is a “byte”

These are not shown as ASCii characters

starting message	converted message
01234567	23456**01**
02468ace	468ace02
048d159c	8d159c04
091a2b38	1a2b3809
12345670	34567012
2468ace0	68ace024
48d159c0	d159c048
91a2b380	a2b38091
23456701	45670123
468ace02	8ace0246
8d159c04	159c048d
1a2b3809	2b38091a
34567012	56701234
68ace024	ace02468
d159c048	59c048d1
a2b38091	b38091a2