FPGA Systems Design and Practice (ET5009701)

Why do you need to take this course?

Date: September 13, 2019. **Instructor: M. B. Lin**

The essential elements of digital systems (VLSI and FPGA) design:

- Modularity
- Regularity
- Locality
- Reusability

For writing a **hardware description language** (HDL) (Verilog HDL, SystemVerilog, or VHDL) module to **describe** or **model** a hardware design, one should always keep in mind that:

- HDL is merely a *hardware description language* (HDL).
- HDL programming is not software programming.
- The hardware mind is very important. *When in Rome, do as the Romans do.*
- HDL is nothing but a tool that is used to describe a hardware design in text.
- HDL is merely an entry method to input a design into a computer-aided design (CAD) system. As a consequence, **you must already have a design and then write an HDL module to describe or model it.** (Please note that the word "**describe**" is used here.)

A case study:

To explore the importance of the above key points, in what follows, we use a practical example to illustrate how the hardware mind is vital in writing a HDL (Verilog HDL or VHDL) module. As we have mentioned, HDL is only a language for describing a hardware design so as to input the design to a CAD system for being implemented (realized) by a real-world device. As a consequence, *to write an HDL module, one should already have the hardware module in one's mind.* This is very important; without the hardware module in the mind, the resulting HDL module will be poor, inefficient in timing, hardware cost, and power consumption, even totally incorrect.

Problem statement:

Design an HDL module that will blink an LED at a specified frequency of 100 Hz, 50 Hz, 10 Hz, or 1 Hz. For each of the blink frequencies, the LED will be set to 50% duty cycle (i.e., it will be on half the time). The LED frequency is chosen via two switches, switch_1 and switch_2. In addition, an enable signal is used. When it is "1," the LED is turned on at the selected frequency; when it is "0," the LED is to turn off. The input frequency of the module is 25 kHz.

Original software-minded Verilog HDL module:

https://www.nandland.com/vhdl/tutorials/tutorial-your-first-vhdl-program-part1.html

Based on the above problem statement, we may draw a conceptual block diagram of the module to be designed. As illustrated in Figure 1, four concurrent counter processes may be needed. Here, the "concurrency" means that they are all running at the exact same time. Their job is to keep track of the number of clock pulses seen for each of the different frequencies. Even if the switches are not selecting that particular frequency, the counters are still running!

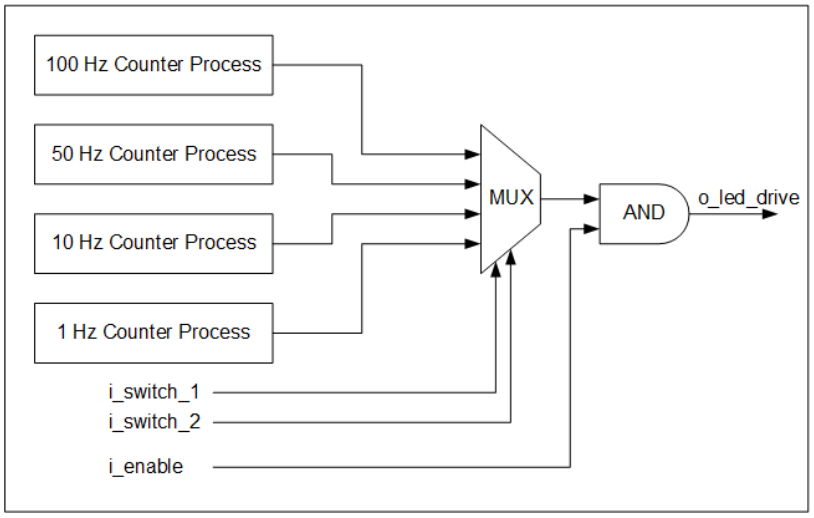

Figure 1: The (software-minded) block diagram of the running example.

The switches only serve to select which output of counters to use. This is realized by a 4-to-1 multiplexer. The resulting Verilog HDL module based-on software mind is shown below.

Without explicit reset operation.

The following Verilog HDL module is the first version, where variables are initialized with 0s. Although this is acceptable by most FPGA implementations and legal in Verilog HDL, it violates the principles of hardware circuits and cannot be accepted by the synthesis tools with cell-based libraries.

```
// an example of using the software programming style
// it consumes much more hardware then hard-mind style
module LED_blinker
      (input i_clock,i_enable, i_switch_1,
       input i_switch_2, reset_n,
       output o_led_drive);

// Constants (parameters) to create the frequencies needed:
// Input clock is 25 kHz, chosen arbitrarily.
// Formula is: (25 kHz / 100 Hz * 50% duty cycle)
// So for 100 Hz: 25,000 / 100 * 0.5 = 125
parameter c_CNT_100HZ = 125;
parameter c_CNT_50HZ  = 250;
parameter c_CNT_10HZ  = 1250;
parameter c_CNT_1HZ   = 12500;

// These signals will be the counters:
reg [31:0] r_CNT_100HZ = 0;
reg [31:0] r_CNT_50HZ  = 0;
reg [31:0] r_CNT_10HZ  = 0;
reg [31:0] r_CNT_1HZ   = 0;

// These signals will toggle at the frequencies needed:
```

```
reg         r_TOGGLE_100HZ = 1'b0;
reg         r_TOGGLE_50HZ  = 1'b0;
reg         r_TOGGLE_10HZ  = 1'b0;
reg         r_TOGGLE_1HZ   = 1'b0;

// One bit select
reg         r_LED_SELECT;

// All always blocks toggle a specific signal at a different frequency.
// They all run continuously even if the switches are
// not selecting their particular output.

always @ (posedge i_clock) begin
    if (r_CNT_100HZ == c_CNT_100HZ-1) begin
        r_TOGGLE_100HZ <= !r_TOGGLE_100HZ; // should use ~
        r_CNT_100HZ <= 0;
    end else
        r_CNT_100HZ <= r_CNT_100HZ + 1;
end

always @ (posedge i_clock) begin
    if (r_CNT_50HZ == c_CNT_50HZ-1) begin
        r_TOGGLE_50HZ <= !r_TOGGLE_50HZ; // should use ~
        r_CNT_50HZ <= 0;
    end else
        r_CNT_50HZ <= r_CNT_50HZ + 1;
end

always @ (posedge i_clock) begin
    if (r_CNT_10HZ == c_CNT_10HZ-1) begin
        r_TOGGLE_10HZ <= !r_TOGGLE_10HZ; // should use ~
        r_CNT_10HZ <= 0;
    end else
        r_CNT_10HZ <= r_CNT_10HZ + 1;
end

always @ (posedge i_clock) begin
    if (r_CNT_1HZ == c_CNT_1HZ-1) begin
        r_TOGGLE_1HZ <= !r_TOGGLE_1HZ;  // should use ~
        r_CNT_1HZ <= 0;
    end else
        r_CNT_1HZ <= r_CNT_1HZ + 1;
end

// Create a multiplexer based on switch inputs
always @(*) begin
    case ({i_switch_1, i_switch_2})         // concatenation Operator { }
```

```
        2'b11: r_LED_SELECT <= r_TOGGLE_1HZ; // should use = rather than <=
        2'b10: r_LED_SELECT <= r_TOGGLE_10HZ;
        2'b01: r_LED_SELECT <= r_TOGGLE_50HZ;
        2'b00: r_LED_SELECT <= r_TOGGLE_100HZ;
     endcase
end

assign o_led_drive = r_LED_SELECT & i_enable;
endmodule
```

The test bench used to verify the above module and the simulation results in both behavior and timing are as follows.

```
// a test bench for the register_ce_8bits module
`timescale 1ns / 1ps
module LED_blinker_tb;
// internal signals declarations
parameter clock_period = 40;
reg clock, enable;
reg switch_1, switch_2;
wire led_drive;
integer i;
// Unit Under Test port map
LED_blinker UUT (
    .i_clock(clock), .i_enable(enable),
    .i_switch_1(switch_1), .i_switch_2(switch_2),
    .o_led_drive(led_drive));
// generate a 25-kHz clock signal
initial clock <= 1'b0;
always begin
   #(clock_period/2) clock <= 1'b0;
   #(clock_period/2) clock <= 1'b1;
end
initial begin
   enable <=1'b0;
   {switch_1,switch_2} = 2'b00;
   repeat (5) @(negedge clock) enable <= 1'b0;
   for (i = 0; i < 50; i = i + 1) begin
       enable <=1'b1;
       {switch_1,switch_2} <= i[1:0];
       repeat (12000) @(negedge clock);
       enable <=1'b0;
       repeat (10) @(negedge clock);
   end
end
initial #6000000 $finish;
initial
   $monitor($realtime,"ns %h %h %h %h %h n",
            clock, enable, switch_1, switch_2, led_drive);
```

endmodule

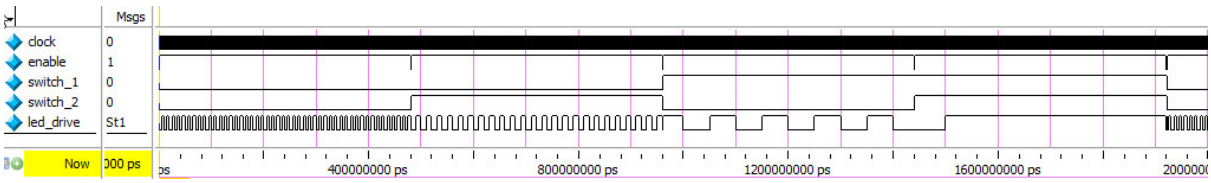

Figure 2: The behavioral simulation result of the LED_blinker module.

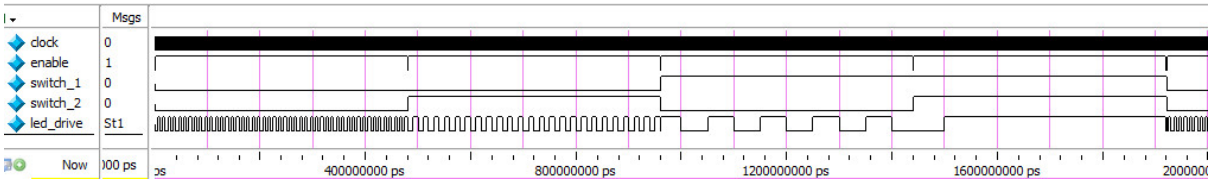

Figure 3: The timing simulation result of the LED_blinker module.

The warning messages regarded to the declarations of initial assignment are shown in Figure 4. The synthesis tool (Design Compiler) just ignores those declarations with initial assignment. From this, we may conclude that *to initialize a* **reg** *variable or vector, an explicit way must be employed.* This exactly matches the features of hardware circuits. Based on this, we also reemphasize that HDL is just a **hardware description language.** As a consequence, it is incorrect to say something like this: *we use Verilog HDL to design a logic circuit*, such as a multiplexer, but instead it needs to say that we use Verilog HDL to **describe** or **model** a logic circuit, meaning that the logic circuit is already existed or designed and we just describe or model it in Verilog HDL.

```
Running PRESTO HDLC
Compiling source file /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:18: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:19: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:20: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:21: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:24: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:25: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:26: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:27: The construct 'declaration initial assignment' is not supported in synthesis; it is ignored. (VER-708)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:37: signed to unsigned conversion occurs. (VER-318)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:45: signed to unsigned conversion occurs. (VER-318)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:53: signed to unsigned conversion occurs. (VER-318)
Warning:  /project/dr159/pw75/pw7504/teacher/src/LED_blinker.v:61: signed to unsigned conversion occurs. (VER-318)
```

Figure 4: The warning messages form Design Compiler.

With explicit reset operation.

To facilitate simulation and make all synthesis tools happy to accept it, an explicit reset input is employed to clear all counters and flip-flops at the beginning. Except for this modification, the basic idea and structure of this module are identical to that of the original one.

```
// an example of using the software programming style --- version 2
// it consumes much more hardware then hard-mind style
module LED_blinker2
       (input i_clock,i_enable, i_switch_1,
        input i_switch_2, reset_n,
        output o_led_drive);
```

```
// Constants (parameters) to create the frequencies needed:
// Input clock is 25 kHz, chosen arbitrarily.
// Formula is: (25 kHz / 100 Hz * 50% duty cycle)
// So for 100 Hz: 25,000 / 100 * 0.5 = 125
parameter c_CNT_100HZ = 125;
parameter c_CNT_50HZ  = 250;
parameter c_CNT_10HZ  = 1250;
parameter c_CNT_1HZ   = 12500;

// These signals will be the counters:
reg [31:0] r_CNT_100HZ;
reg [31:0] r_CNT_50HZ;
reg [31:0] r_CNT_10HZ;
reg [31:0] r_CNT_1HZ;

// These signals will toggle at the frequencies needed:
reg        r_TOGGLE_100HZ;
reg        r_TOGGLE_50HZ;
reg        r_TOGGLE_10HZ;
reg        r_TOGGLE_1HZ;

// One bit select
reg        r_LED_SELECT;

// All always blocks toggle a specific signal at a different frequency.
// They all run continuously even if the switches are
// not selecting their particular output.

always @ (posedge i_clock or negedge reset_n) begin
    if (!reset_n) begin    // add reset_n to enable simulation
        r_CNT_100HZ <= 0;
        r_TOGGLE_100HZ <= 0; end
    else if (r_CNT_100HZ == c_CNT_100HZ-1) begin
            r_TOGGLE_100HZ <= !r_TOGGLE_100HZ; // should use ~
            r_CNT_100HZ <= 0;
         end else
            r_CNT_100HZ <= r_CNT_100HZ + 1;
end

always @ (posedge i_clock or negedge reset_n) begin
    if (!reset_n) begin  // add reset_n to enable simulation
        r_CNT_50HZ <= 0;
        r_TOGGLE_50HZ <= 0; end
    else if (r_CNT_50HZ == c_CNT_50HZ-1) begin
            r_TOGGLE_50HZ <= !r_TOGGLE_50HZ; // should use ~
            r_CNT_50HZ <= 0;
```

```
        end else
            r_CNT_50HZ <= r_CNT_50HZ + 1;
end

always @ (posedge i_clock or negedge reset_n) begin
    if (!reset_n) begin  // add reset_n to enable simulation
        r_CNT_10HZ <= 0;
        r_TOGGLE_10HZ <= 0; end
    else if (r_CNT_10HZ == c_CNT_10HZ-1) begin
            r_TOGGLE_10HZ <= !r_TOGGLE_10HZ; // should use ~
            r_CNT_10HZ <= 0;
        end else
            r_CNT_10HZ <= r_CNT_10HZ + 1;
end

always @ (posedge i_clock or negedge reset_n) begin
    if (!reset_n) begin  // add reset_n to enable simulation
        r_CNT_1HZ <= 0;
        r_TOGGLE_1HZ <= 0; end
    else if (r_CNT_1HZ == c_CNT_1HZ-1) begin
            r_TOGGLE_1HZ <= !r_TOGGLE_1HZ;  // should use ~
            r_CNT_1HZ <= 0;
        end else
            r_CNT_1HZ <= r_CNT_1HZ + 1;
end

// Create a multiplexer based on switch inputs
always @(*) begin
    case ({i_switch_1, i_switch_2})         // concatenation Operator { }
       2'b11: r_LED_SELECT = r_TOGGLE_1HZ;  // should use = rather than <=
       2'b10: r_LED_SELECT = r_TOGGLE_10HZ;
       2'b01: r_LED_SELECT = r_TOGGLE_50HZ;
       2'b00: r_LED_SELECT = r_TOGGLE_100HZ;
    endcase
end

assign o_led_drive = r_LED_SELECT & i_enable;
endmodule
```

The salient features of the above Verilog HDL module are as follows:

1. It is clear that the author does not understand the exact meaning of HDL.
2. It is apparent that the four counters do not need 32 bits.
3. The misuse of blocking (=) and nonblocking (<=) operators appears in the 4-to-1 multiplexer.
4. Variables and counters are initialized with constants 0s along with their declarations. Despite that this is fine in most FPGA implementations, it is not generally acceptable by

synthesis tools (such as design compiler), especially as the constants are not 0s, and in the other implementations.

5. The module is written entirely based on a software mind. The author has no concrete idea about the underlying hardware. The most important point is that he/she does not know well about the properties of counters and registers.
6. Actually, it does not need four equal-length (32-bit) counters. It only requires to cascade together a modulo-250, a modulo-2, a modulo-5, and a modulo-10 counter.

Concluding Remarks:

The author misuses *the beauty of Hardware Design and concurrency.* In spite of the above two modules satisfy the modularity, it is poor in regularity and reusability. In addition, to evaluate whether a hardware module is designed well or not, the following performance merits must be considered:

- Performance — speed, operating frequency, propagation delays, and so on.
- Cost — hardware costs (number of LUTs/LEs), human costs, and so forth.
- Power consumption.

Other important factors include reliability, maintainability, robustness, and so on.

Although initialization of **reg** variables with net declaration assignments can be accepted in FPGA implementations due to their special hardware features, it is not acceptable in general by the other types of implementations and synthesis tools.

The use of declarations to initialize **reg** variables is basically a software programming concept. It cannot be generally realized in hardware. In hardware, to cleart a sequential element, an explicit reset input must be used.

In a digital system, it is vital to reset the system to a known state after the power-on as well as any time needed thereafter. As a consequence, in writing an HDL module to describe a hardware design, an explicit reset input must be provided for real-world operation and for verification as well.

My hardware-mind program I:

From the hardware-minded point, the *beauty of Hardware Design and concurrency* can be achieved in an alternate way. To design a hardware circuit, we need to get ourselves involved into the hardware world. Everything we want to do must be in the hardware world. In addition, we need always to keep in mind the four essential elements of digital systems (VLSI and FPGA) design: **modularity, regularity, locality,** and **reusability.** Consequently, as shown in Figure 5, we may design the desired hardware module in a way that four counters are cascaded together. These four counters are a modulo-250, a modulo-2, a modulo-5, and a modulo-10. The output of the modulo-250 counter is 100 Hz, the modulo-2 counter is 50 Hz, the modulo-5 counter is 10 Hz, and the modulo-10 counter is 1 Hz.

In the first **description** of the hardware module, four **always** statements are used each for each counter. Please note that each **always** block corresponds to a piece of logic.

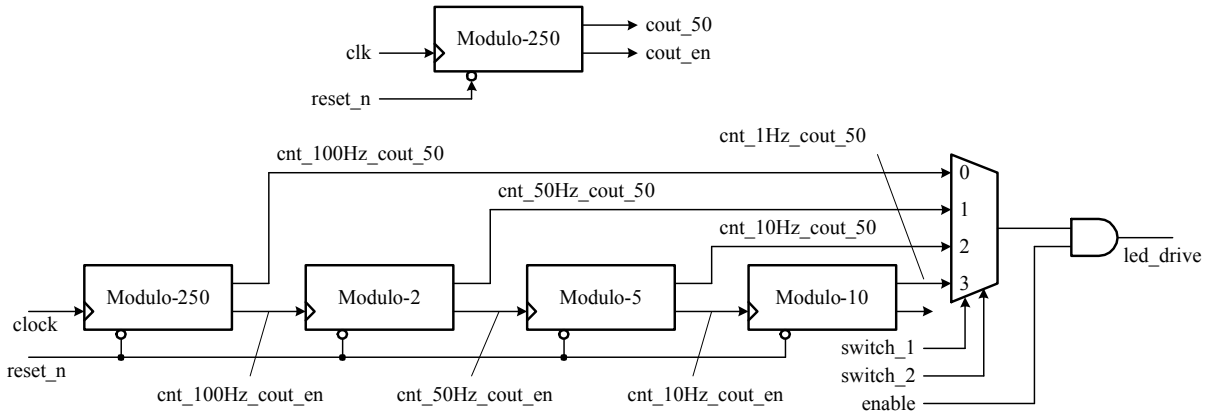

Figure 5: The hardware-minded block diagram of the running example.

```
// an example of hardware-mind style --- version 1
module LED_blinker_my
      (input clock, enable, switch_1, switch_2, reset_n,
       output led_drive);

// input clock is 25 kHz
parameter cnt_100Hz_m = 250;  // modulus for 100 Hz
parameter cnt_50Hz_m  = 2;    // modulus for 50 Hz
parameter cnt_10Hz_m  = 5;    // modulus for 10 Hz
parameter cnt_1Hz_m   = 10;   // modulus for 1 Hz

reg [7:0] cnt_100Hz;  // modulo-250 counter
reg [0:0] cnt_50Hz;   // modulo-2 counter
reg [2:0] cnt_10Hz;   // modulo-5 counter
reg [3:0] cnt_1Hz;    // modulo-10 counter

wire cnt_100Hz_cout_50;
wire cnt_50Hz_cout_50;
wire cnt_10Hz_cout_50;
wire cnt_1Hz_cout_50;

 // One bit select
reg LED_select;
wire cnt_100Hz_cout_en, cnt_50Hz_cout_en, cnt_10Hz_cout_en, cnt_1Hz_cout_en;

// modulo-250 counter --- generate 100-Hz output
assign cnt_100Hz_cout_50 = (cnt_100Hz >= cnt_100Hz_m >> 1);
assign cnt_100Hz_cout_en = (cnt_100Hz == cnt_100Hz_m - 1);
always @(posedge clock or negedge reset_n) begin
    if (!reset_n) cnt_100Hz <= 0;
    else if (cnt_100Hz_cout_en) cnt_100Hz <= 0;
    else cnt_100Hz <= cnt_100Hz + 1;
end
```

```
// modulo-2 counter --- generate 50-Hz output
assign cnt_50Hz_cout_50 = (cnt_50Hz >= cnt_50Hz_m >> 1);
assign cnt_50Hz_cout_en = (cnt_50Hz == cnt_50Hz_m - 1);
always @(negedge cnt_100Hz_cout_en or negedge reset_n) begin
    if (!reset_n) cnt_50Hz <= 0;
    else if (cnt_50Hz_cout_en) cnt_50Hz <= 0;
    else cnt_50Hz <= cnt_50Hz + 1;
end

// modulo-5 counter --- generate 10-Hz output
assign cnt_10Hz_cout_50 = (cnt_10Hz >= cnt_10Hz_m >> 1);
assign cnt_10Hz_cout_en = (cnt_10Hz == cnt_10Hz_m - 1);
always @(negedge cnt_50Hz_cout_en or negedge reset_n) begin
    if (!reset_n) cnt_10Hz <= 0;
    else if (cnt_10Hz_cout_en) cnt_10Hz <= 0;
    else cnt_10Hz <= cnt_10Hz + 1;
end

// modulo-10 counter --- generate 1-Hz output
assign cnt_1Hz_cout_50 = (cnt_1Hz >= cnt_1Hz_m >> 1);
assign cnt_1Hz_cout_en = (cnt_1Hz == cnt_1Hz_m - 1);
always @(negedge cnt_10Hz_cout_en or negedge reset_n) begin
    if (!reset_n) cnt_1Hz <= 0;
    else if (cnt_1Hz_cout_en) cnt_1Hz <= 0;
    else cnt_1Hz <= cnt_1Hz + 1;
end

 // create a multiplexer based on switch inputs
always @(*) begin
    case ({switch_1, switch_2}) // concatenation Operator { }
      2'b00: LED_select = cnt_100Hz_cout_50;
      2'b01: LED_select = cnt_50Hz_cout_50;
      2'b10: LED_select = cnt_10Hz_cout_50;
      2'b11: LED_select = cnt_1Hz_cout_50;
    endcase
end

assign led_drive = LED_select & enable;

endmodule
```

The test bench used to verify the above module and the simulation results in both behavior and timing are as follows. This test bench can be used to test the other modules, such as LED_-blinker2 and LED_blinker_my2, as well except for that the module name in the instantiation statement must be changed accordingly.

```
// a test bench for the register_ce_8bits module
`timescale 1ns / 1ps
module LED_blinker_my_tb;
```

```
// internal signals declarations
parameter clock_period = 40;
reg clock, enable;
reg switch_1, switch_2, reset_n;
wire led_drive;
integer i;
// Unit Under Test port map
LED_blinker_my UUT (
        .clock(clock), .enable(enable),
        .switch_1(switch_1), .switch_2(switch_2), .reset_n(reset_n),
        .led_drive(led_drive));
// generate a 25-kHz clock signal
initial clock <= 1'b0;
always begin
   #(clock_period/2) clock <= 1'b0;
   #(clock_period/2) clock <= 1'b1;
end
initial begin
   enable <=1'b0; reset_n = 1'b0;
   repeat (5) @(negedge clock);
   reset_n = 1'b1;
   {switch_1,switch_2} = 2'b00;
   repeat (5) @(negedge clock) enable <= 1'b0;
   for (i = 0; i < 50; i = i + 1) begin
       enable <=1'b1;
       {switch_1,switch_2} <= i[1:0];
       repeat (12000) @(negedge clock);
       enable <=1'b0;
       repeat (10) @(negedge clock);
    end
end
initial #6000000 $finish;
initial
   $monitor($realtime,"ns %h %h %h %h %h %h \n",
            clock, enable, switch_1, switch_2, reset_n, led_drive);
endmodule
```

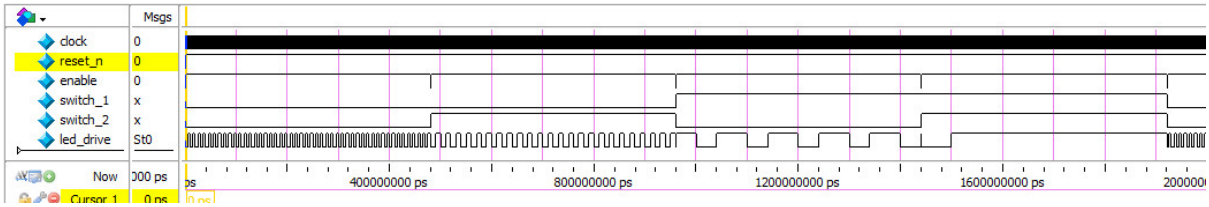

Figure 6: The behavioral simulation result of the LED_blinker_my module.

My hardware-mind program II:

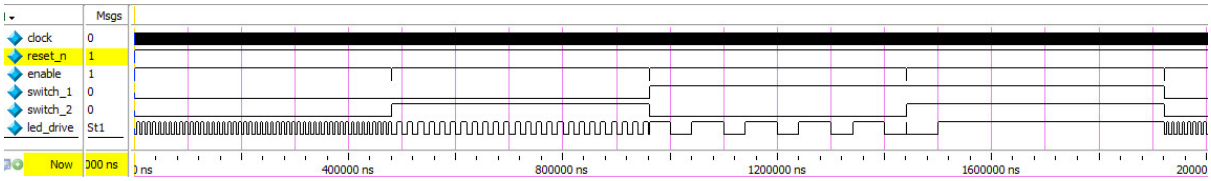

Figure 7: The timing simulation result of the LED_blinker_my module.

To explore the modularity, regularity, and reusability, we may design a universal counter module and then use it to realize the four desired counters. The resulting module is as follows.

```
// an example of hardware-mind style --- version 2
// to explore the module resuability
module LED_blinker_my
  (input clock, enable, switch_1, switch_2, reset_n,
   output led_drive);

// input clock is 25 kHz, all counters are cascaded in the order
// shown below
parameter cnt_100Hz_m = 250;  // modulus for 100 Hz
parameter cnt_50Hz_m  = 2;    // modulus for 50 Hz
parameter cnt_10Hz_m  = 5;    // modulus for 10 Hz
parameter cnt_1Hz_m   = 10;   // modulus for 1 Hz

wire [7:0] cnt_100Hz;  // modulo-250 counter
wire       cnt_50Hz;   // modulo-2 counter
wire [2:0] cnt_10Hz;   // modulo-5 counter
wire [3:0] cnt_1Hz;    // modulo-10 counter

 // one bit select
reg LED_select;

// instantiate the modulo_r_counter four times and cascade them together
// to form the derired counters
wire cnt_100Hz_cout_en, cnt_50Hz_cout_en, cnt_10Hz_cout_en, count_en_out2;
modulo_r_counter #(cnt_100Hz_m, 8) mod_250_cnt
         (.clk(clock), .reset_n(reset_n), .cout_50(cnt_100Hz_cout_50),
          .cout_en(cnt_100Hz_cout_en), .qout(cnt_100Hz));
modulo_r_counter #(cnt_50Hz_m, 1) mod_2_cnt
         (.clk(cnt_100Hz_cout_en), .reset_n(reset_n),
          .cout_50(cnt_50Hz_cout_50),
          .cout_en(cnt_50Hz_cout_en), .qout(cnt_50Hz));
modulo_r_counter #(cnt_10Hz_m, 3) mod_5_cnt
         (.clk(cnt_50Hz_cout_en), .reset_n(reset_n),
          .cout_50(cnt_10Hz_cout_50),
          .cout_en(cnt_10Hz_cout_en), .qout(cnt_10Hz));
modulo_r_counter #(cnt_1Hz_m, 4) mod_10_cnt
         (.clk(cnt_10Hz_cout_en), .reset_n(reset_n),
          .cout_50(cnt_1Hz_cout_50),
```

```
          .cout_en(count_en_out2), .qout(cnt_1Hz));

 // create a multiplexer based on switch inputs
always @(*) begin
    case ({switch_1, switch_2}) // concatenation Operator { }
      2'b00: LED_select = cnt_100Hz_cout_50;
      2'b01: LED_select = cnt_50Hz_cout_50;
      2'b10: LED_select = cnt_10Hz_cout_50;
      2'b11: LED_select = cnt_1Hz_cout_50;
    endcase
end

assign led_drive = LED_select & enable;

endmodule

// a modulo-R binary counter with asynchronous reset and
// enable control. The output is a 50% duty cycle
module modulo_r_counter
       #(parameter R = 10,  // default modulus
         parameter N = 4)(  // N = log2 R
         input  clk, reset_n,
         output cout_50,  // 50% duty-cycle output
         output cout_en,  // carry-out to enable the next stage
         output reg [N-1:0] qout);

// the body of the modulo r binary counter
assign cout_50 = (qout >= R >> 1);
assign cout_en = (qout == R - 1);
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= 0;
   else if (cout_en) qout <= 0;
        else qout <= qout + 1;
endmodule
```

Performance comparison:

The hardware cost of the above four versions implemented with FPGA and cell-based devices are listed in Tables 1 and 2, respectively. By observing the data from these two tables, we may conclude and reemphasize the importance of the design paradigm:

Design $\longrightarrow$ **Entry** (schematic or HDL) $\longrightarrow$ **Verification** $\longrightarrow$ **Real-world test**.

The entry methods can be schematic or HDL (Verilog HDL, SystemVerilog, and VHDL) in general. However, *without a design, it is nonsense to talk about the entry method* because nothing can be input into a CAD system for further processing.

It is **a fallacy** that a module is said to be well-designed when it not only can be simulated in behavior but also can be synthesized. A **well-designed module** must be

Table 1: Comparison of hardware cost of the LED blinker implemented with FPGA devices.

	Quartus II (EP4CE115F29C7)			Xilinx ISE (XC3S200A)		
	LEs	FFs	Delay (ns)	LUTs	FFs	Delay (ns)
Original module	200	130	7.767	291	132	9.767
Original module (with initialization)	200	130	7.767	163	132	9.767
My version 1	28	16	6.152	35	16	7.831
My version 2	28	16	6.152	35	16	7.831

Table 2: Comparison of hardware cost of the LED blinker implemented with TSMC 0.18-μm cell-based library (reported from Design compiler).

	Original (software-minded)		My version	
	With initialization	With explicit reset	Version 1	Version 2
Combinational cells	178	306	42	46
Sequential cells	132	132	16	18
Buffers/inverters	4	4	2	7
Total cells	314	442	59	69
Combinational area	5441.990493	7224.940932	818.294410	848.232011
Noncombinational area	7451.136185	9220.780884	1170.892822	1150.934402
Buffer/inverter area	26.611200	26.611200	13.305600	46.569601
Total cell area	12893.126678	16445.721816	1989.187232	1999.166414

the one that uses minimal hardware resource, consumes least power, and has smallest propagation delays on condition that it satisfies the desired specifications.

To achieve a well-designed hardware module, one should dive oneself into the hardware world (*When in Rome, do as the Romans do.*) and familiarize with the gate-level, register-transfer-level (RTL), even system-level components and their related characteristics. By further incorporating the algorithmic-state chart (ASM) along with the datapath-and-controller paradigm, a complex digital system can then be designed. After this, the design can be entered a CAD system to synthesize into a real-world device, such as an FPGA or ASIC device.

It should be noted that the fundamental differences of the meanings of modularity, regularity, and reusability in between the software world and hardware world. In the software world, only one copy of a module is needed when it is reused in many places as desired while in the hardware world as many copies of a module must be created as the places to use it. It is encouraged for the student to try to differentiate the rationale behind them.

Conclusion:

While writing an HDL module, one should always think it in hardware mind since it describes a hardware design. Probably, this is the most difficult problem for the newcomer who uses Verilog HDL to describe the design of a digital system. To help readers overcome such an obstacle, in this course, especially in the term project, we intend to first instruct the reader to learn how

to design and input logic circuits schematically and then verify them and optionally test the logic circuits on a real-world FPGA device. After this, Verilog HDL modules corresponding to these logic circuits are written, verified, and tested.