1st Lab Assignment | Overview Report

ECE333 | Digital Systems Laboratory – Professor: Christos Sotiriou – Dimitris Garyfallou

Lab Instructor(s): Nikos Chatzivangelis, Dimitris Tsalapatas, Katerina Tsilingiri, Nikolaos Zazatis, Anastasis Vagenas

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module (*Golden Rule #1*)
  - Combinational vs Sequential always blocks (*Golden Rule #2*)
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic (*Golden Rule #4*)
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Golden Rule #1: Dataflow Design

- Design **Dataflow** ***prior the implementation process***
  - Dataflow:
    - **must be strictly equivalent *(1-1)*** to the instantiations of top-level module
    - in case of multiple levels in hierarchy,
      - you can provide more detailed info
        - either within the same schematic
        - or with a separate dataflow of the internal hierarchy

! Do not implement dataflow on hand, use a drawing tool like drawio

Golden Rule #2: Procedural Block Assignments

- **Sequential vs Combinational** `always@` block
  - Sequential:
    - *always @(posedge clock or …)*
      - *posedge clock*, implies the **instantiation of flip-flop*(s)***
      - every *other signal in sensitivity list*, implies an **asynchronous control** signal for the flip-flop*(s)*
      - any signal *not in the sensitivity list* and *used for multiplexing* within the block, implies a **synchronous control** signal for the flip-flop*(s)*
        - *Rule of thumb:* ***Non-Blocking Assignments (<=)***
  - Combinational:
    - *always @(signal_1 or … or signal_x)*
      - sensitivity list must contain **all associated input** signals
        - *Rule of thumb:* ***Blocking Assignments (=)***

Minimal Implementation with *Sequential Elements*

- ❑ always reconsider whether you can eliminate some of the instantiated sequential logic

- ❖ **ALWAYS SEPARATE** combinational and sequential logic blocks
- ❖ **Do NOT MIX** blocking and non-blocking assignments *in the same* `always` *block*
- ❖ **Do NOT ASSIGN** the same variable *from more than one* `always` *block*
- ❖ **Do NOT ASSIGN** as output and **USE** as input a variable *in the same* `always` *block*

Which one is better?

```
1
2  module anodes_mux();
3
4      // multiplexer for anodes
5      always @(posedge clk or posedge reset)
6      begin
7          case (counter_digit)
8              4'b0010: begin // digit 0
9                  an = 4'b1110;
10             end
11             4'b0110: begin // digit 1
12                 an = 4'b1101;
13             end
14             4'b1010: begin // digit 2
15                 an = 4'b1011;
16             end
17             4'b1110: begin // digit 3
18                 an = 4'b0111;
19             end
20             default: begin // default all digits are off
21                 an = 4'b1111;
22             end
23         endcase
24     end
25
26 endmodule
```

```
1  module anodes_mux();
2
3      // multiplexer for anodes
4      always @(counter_digit)
5      begin
6          case (counter_digit)
7              4'b0010: begin // digit 0
8                  an = 4'b1110;
9              end
10             4'b0110: begin // digit 1
11                 an = 4'b1101;
12             end
13             4'b1010: begin // digit 2
14                 an = 4'b1011;
15             end
16             4'b1110: begin // digit 3
17                 an = 4'b0111;
18             end
19             default: begin // default all digits are off
20                 an = 4'b1111;
21             end
22         endcase
23     end
24
25 endmodule
```

Which one is better?

```
1
2  module anodes_mux();
3
4      // multiplexer for anodes
5      always @(posedge clk or posedge reset)
6      begin
7          case (counter_digit)
8              4'b0010: begin // digit 0
9                  an = 4'b1110;
10             end
11             4'b0110: begin // digit 1
12                 an = 4'b1101;
13             end
14             4'b1010: begin // digit 2
15                 an = 4'b1011;
16             end
17             4'b1110: begin // digit 3
18                 an = 4'b0111;
19             end
20             default: begin // default all digits are off
21                 an = 4'b1111;
22             end
23         endcase
24     end
25
26 endmodule
```

```
1  module anodes_mux();
2
3      // multiplexer for anodes
4      always @(counter_digit)
5      begin
6          case (counter_digit)
7              4'b0010: begin // digit 0
8                  an = 4'b1110;
9              end
10             4'b0110: begin // digit 1
11                 an = 4'b1101;
12             end
13             4'b1010: begin // digit 2
14                 an = 4'b1011;
15             end
16             4'b1110: begin // digit 3
17                 an = 4'b0111;
18             end
19             default: begin // default all digits are off
20                 an = 4'b1111;
21             end
22         endcase
23     end
24
25 endmodule
```

Golden Rule #4: Reset & Clock Logic

- **NEVER put logic** on the ***reset*** or ***clock***
  - **NEVER MIX** reset types
    - **asynchronous** vs **synchronous** reset
  - **NEVER** create clock from custom combinational logic
    - **In case you think you should a custom combinational signal**
      - ***probably that signal should be a*** `clock enable`
  - **NEVER** create clock domain crossings ***(studied in more advanced courses)***
    - Advanced designs have more than one clock
      - *which they usually run on different frequencies*

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**G**olden **R**ule **#1**)*
  - Combinational vs Sequential always blocks *(**G**olden **R**ule **#2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**G**olden **R**ule **#4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Hierarchical Design, Coding Style & Comments

- Top-module must contain only module instantiations
  - **NO RTL in top-level**
    - direct wire assignments are allowed
      - *e.g.:* `assign signal_X = signal_Y;`
- Every module should be in a separate file
  - module header comment segment with a basic functional description
  - I/O ports description
  - utilise parameters or definitions
    - ***parameter*** vs ***`define***
  - for each instantiation/implementation add header comment
    - describe explicit corner cases
- Follow a uniform coding style across all files
  - comments
  - indentations
  - `begin-end` usage and format
  - *etc.*

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Combinational Logic & Reset

- Does combinational logic require reset logic?
  - as described below:

```
always @(signal_1 or … or signal_x)
    begin
        if (reset)
            begin
                signal_1 = 1'b0;
                …
                signal_x = 1'b1;
            end // if (reset) //
        else
            begin
                …
            end // else //
    end // always @(signal_1 or … or signal_x) //
```

Correct Implementation of reset in combinational blocks

```
1  always @(signal_1 or signal_2)
2  begin
3      if(enable == 1'b0)
4      begin
5          signal_1 = 1'b0;
6          signal_2 = 1'b0;
7      end
8      else
9      begin
10         ....
11     end
12 end
```

```
1  always @(posedge clk or posedge reset)
2  begin
3      if(reset == 1'b1)
4      begin
5          enable <= 1'b0;
6      end
7      else
8      begin
9          enable <= 1'b1;
10     end
11 end
```

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Anode Driver Refresh Period

- 9.1 Seven-Segment Display - Nexys A7 Reference Manual

To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Nexys A7 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..7 and the CA..G/DP signals are driven low when active.

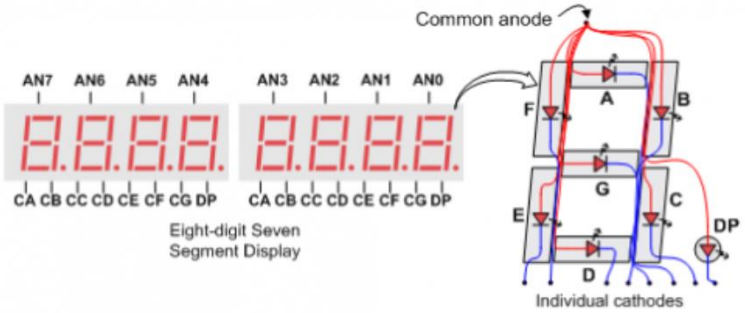


Figure 9.1.2 Common Anode Circuit Node

For each of the four digits to appear bright and continuously illuminated, all eight digits should be driven once every 1 to 16ms, for a refresh frequency of about 1 KHz to 60Hz. For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/8 of the refresh cycle, or 2ms. The controller must drive low the cathodes with the correct pattern when the corresponding anode signal is driven high. To illustrate the process, if

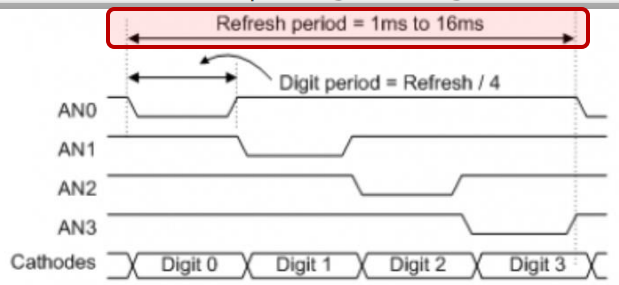


Figure 9.1.3 Four Digit Scanning Display Controller Timing Diagram

Anode Driver Refresh Period

- 9.1 Seven-Segment Display - Nexys A7 Reference Manual

To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Nexys A7 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..7 and the CA..G/DP signals are driven low when a

Figure 9.1.2 Common Anode Circuit Node

Everything you need to know, **especially for provided Xilinx/Digilent IPs and functionality**, is documented in related manuals

For each of the four digits to appear bright and continuo illuminated, all eight digits should be driven once every 1 to 16ms, for a refresh frequency of about 1 KHz to 60Hz. For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/8 of the refresh cycle, or 2ms. The controller must drive low the cathodes with the correct pattern when the corresponding anode signal is driven high. To illustrate the process, if

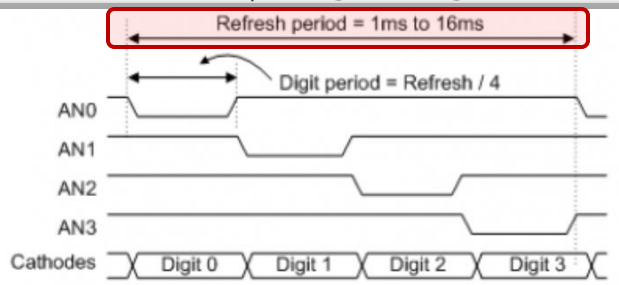


Figure 9.1.3 Four Digit Scanning Display Controller Timing Diagram

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Debouncer & Reset Signal

- How can you simultaneously
  - count bouncing glitches, and
  - reset utilised counter & synchroniser flip-flop(s)
- using the same signal?

```
// 2 Stage Pipeline Synchroniser with Asynchronous Reset //
always @(posedge clock or posedge reset)
    begin
        if (reset)
            begin
                sync_pipe_stage1 <= 1'b1;
                reset_sync <= 1'b1;
            end // if (reset) //
        else
            begin
                sync_pipe_stage1 <= reset;
                reset_sync <= sync_pipe_stage1;
            end // else //
    end // always @(posedge clock or posedge reset) //
```

Debouncer & Reset Signal

- How can you simultaneously
  - count bouncing glitches, and
  - reset utilised counter & synchroniser flip-flop(s)
- using the same signal?

```
// 2 Stage Pipeline Synchroniser with Asynchronous Reset //
always @(posedge clock or posedge reset)
    begin
        if (reset)
            begin
                sync_pipe_stage1 = 1'b1;
                reset_sync = 1'b1;
            end // if (reset) //
        else
            begin
                sync_pipe_stage1 <= reset;
                reset_sync <= sync_pipe_stage1;
            end // else //
    end // always @(posedge clock or posedge reset) //
```

Debouncer & Logic Comparison in Time

- What is the duration of bouncing effect for a button on FPGA?
  - about ≈5-10ms
    - **determined after a 5min search on the internet**
- How do we compare logic signals *(= or ≠)*?

XOR

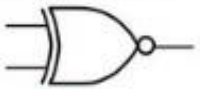
X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

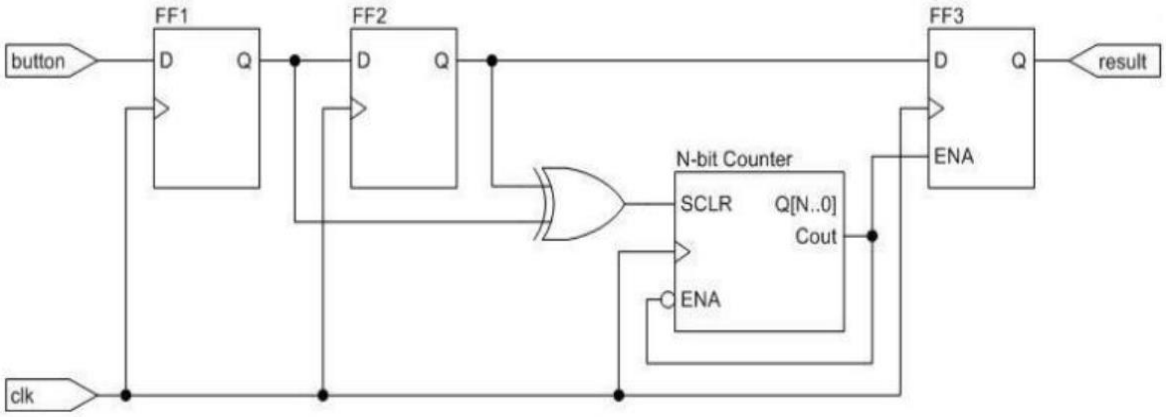
X	Y	$\overline{X \oplus Y}$
0	0	1
0	1	0
1	0	0
1	1	1

XOR Symbol

XNOR Symbol



- **The XNOR is also denoted as equivalence**



- *some useful sources:*
  - 11 Myths about Switch Bounce/Debounce | www.electronicdesign.com
  - Switch Debounce Using TMR2 | microchipdeveloper.com

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

MMCM Reset

- LOCKED - Status and Data Outputs
  - 7 Series FPGAs Clocking Resources - User Guide

LOCKED

An output from the MMCM/PLL used to indicate when the MMCM/PLL have achieved phase and frequency alignment of the reference clock and the feedback clock at the input pins. Phase alignment is within a predefined window and frequency matching within a predefined PPM range. The MMCM automatically locks after power on, no extra reset is required. LOCKED will be deasserted within one PFD clock cycle if the input clock stops, the phase alignment is violated (for example, input clock phase shift) or the frequency has changed. The MMCM/PLL must be reset when LOCKED is deasserted. The clock outputs should not be used prior to the assertion of LOCKED.

- What is the purpose of LOCKED signal?

MMCM Reset

- LOCKED - Status and Data Outputs
  - 7 Series FPGAs Clocking Resources - User Guide

LOCKED

An output from the MMCM/PLL used to indicate when the MMCM/PLL have achieved phase and frequency alignment of the reference clock and the feedback clock at the input pins. Phase alignment is within a predefined window and frequency matching within a predefined PPM range. The MMCM automatically locks after power on, no extra reset is required. LOCKED will be deasserted within one PFD clock cycle if the input clock stops, the phase alignment is violated (for example, input clock phase shift) or the frequency has changed. The MMCM/PLL must be reset when LOCKED is deasserted. The clock outputs should not be used prior to the assertion of LOCKED.

- If you used the synchronised-debounced reset for `MMCM`
  - you should consider `LOCKED` signal
    - as **a reset?**
      - or
    - as **a clock enable?**
      - *for the rest of your design*

MMCM Reset

- LOCKED - Status and Data Outputs
  - 7 Series FPGAs Clocking Resources - User Guide

LOCKED

An output from the MMCM/PLL used to indicate when the MMCM/PLL have achieved phase and frequency alignment of the reference clock and the feedback clock at the input pins. Phase alignment is within a predefined window and frequency matching within a predefined PPM range. The MMCM automatically locks after power on, no extra reset is required. LOCKED will be deasserted within one PFD clock cycle if the input clock stops, the phase alignment is violated (for example, input clock phase shift) or the frequency has changed. The MMCM/PLL must be reset when LOCKED is deasserted. The clock outputs should not be used prior to the assertion of LOCKED.

- If you used the synchronised-debounced reset for `MMCM`
  - you should consider `LOCKED` signal
    - as ~~a reset~~
      - or
    - as **a clock enable**
      - *for the rest of your design*

For the purposes of this course, you must follow *Golden Rule #4*.

When you become advanced designers, *you may consider the implementation as reset logic.*

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

MMCM Feedback

- CLKFBIN - Feedback Clock Input
  - 7 Series FPGAs Clocking Resources - User Guide

CLKFBIN – Feedback Clock Input

Must be connected either directly to the CLKFBOUT for internal feedback or IBUFG (through a clock-capable pin for external deskew), BUFG, BUFH, or interconnect (not recommended). For external clock alignment, the feedback path clock buffer type should match the forward clock buffer type with the exception of BUFR. BUFR cannot be compensated for.

MMCM Feedback

- CLKFBIN - Feedback Clock Input
  - 7 Series FPGAs Clocking Resources - User Guide

CLKFBIN – Feedback Clock Input

Must be connected either directly to the CLKFBOUT for internal feedback or IBUFG (through a clock-capable pin for external deskew), BUFG, BUFH, or interconnect (not recommended). For external clock alignment, the feedback path clock buffer type should match the forward clock buffer type with the exception of BUFR. BUFR cannot be compensated for.

Everything you need to know, **especially for provided Xilinx/Digilent IPs and functionality**, is documented in related manuals

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

MMCM VCO Minimum Frequency

- Voltage-Controlled Oscillator (VCO)
  - VCO operates @600MHz by default, meaning
    - for default parameters, and
    - system clock @100MHz as input clock
  - Thus, all provided output clocks operate @600MHz
    - with default parameters

$$F_{VCO} = F_{CLKIN} \times \frac{M}{D}$$

$$F_{OUT} = F_{CLKIN} \times \frac{M}{D \times O}$$

Which is the minimum achievable frequency?

Did you consider

VCO frequency?

OR

System Clock's frequency?

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Registers' Memory vs FPGA ROM

```
reg [3:0] message [0:15];
```

- Delivered code for required memory
  - Which are correct?

```
assign message[0]  = CHAR_0;
assign message[1]  = CHAR_1;
assign message[2]  = CHAR_2;
assign message[3]  = CHAR_3;
assign message[4]  = CHAR_4;
assign message[5]  = CHAR_5;
assign message[6]  = CHAR_6;
assign message[7]  = CHAR_7;
assign message[8]  = CHAR_8;
assign message[9]  = CHAR_9;
assign message[10] = CHAR_A;
assign message[11] = CHAR_B;
assign message[12] = CHAR_C;
assign message[13] = CHAR_D;
assign message[14] = CHAR_E;
assign message[15] = CHAR_F;

assign char = message[address];
```

```
always @(posedge clk)
begin
  if (reset)
    begin
      message[0]  = CHAR_0;
      message[1]  = CHAR_1;
      message[2]  = CHAR_2;
      message[3]  = CHAR_3;
      message[4]  = CHAR_4;
      message[5]  = CHAR_5;
      message[6]  = CHAR_6;
      message[7]  = CHAR_7;
      message[8]  = CHAR_8;
      message[9]  = CHAR_9;
      message[10] = CHAR_A;
      message[11] = CHAR_B;
      message[12] = CHAR_C;
      message[13] = CHAR_D;
      message[14] = CHAR_E;
      message[15] = CHAR_F;
    end
end
```

```
always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

```
always @(posedge clk or posedge reset)
begin
  if (reset)
    begin
      message[0]  = CHAR_0;
      message[1]  = CHAR_1;
      message[2]  = CHAR_2;
      message[3]  = CHAR_3;
      message[4]  = CHAR_4;
      message[5]  = CHAR_5;
      message[6]  = CHAR_6;
      message[7]  = CHAR_7;
      message[8]  = CHAR_8;
      message[9]  = CHAR_9;
      message[10] = CHAR_A;
      message[11] = CHAR_B;
      message[12] = CHAR_C;
      message[13] = CHAR_D;
      message[14] = CHAR_E;
      message[15] = CHAR_F;
    end
end
```

```
always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

```
always @(posedge clk or posedge reset)
begin
  if (reset)
    begin
      message[0]  <= CHAR_0;
      message[1]  <= CHAR_1;
      message[2]  <= CHAR_2;
      message[3]  <= CHAR_3;
      message[4]  <= CHAR_4;
      message[5]  <= CHAR_5;
      message[6]  <= CHAR_6;
      message[7]  <= CHAR_7;
      message[8]  <= CHAR_8;
      message[9]  <= CHAR_9;
      message[10] <= CHAR_A;
      message[11] <= CHAR_B;
      message[12] <= CHAR_C;
      message[13] <= CHAR_D;
      message[14] <= CHAR_E;
      message[15] <= CHAR_F;
    end
  else
    begin
      char <= message[address];
    end
end
```

```
initial
begin
  message[0]  = CHAR_0;
  message[1]  = CHAR_1;
  message[2]  = CHAR_2;
  message[3]  = CHAR_3;
  message[4]  = CHAR_4;
  message[5]  = CHAR_5;
  message[6]  = CHAR_6;
  message[7]  = CHAR_7;
  message[8]  = CHAR_8;
  message[9]  = CHAR_9;
  message[10] = CHAR_A;
  message[11] = CHAR_B;
  message[12] = CHAR_C;
  message[13] = CHAR_D;
  message[14] = CHAR_E;
  message[15] = CHAR_F;
end

always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

Registers' Memory vs FPGA ROM

```
reg [3:0] message [0:15];
```

- Delivered code for required memory
  - Which are correct?

```
assign message[0]  = CHAR_0;
assign message[1]  = CHAR_1;
assign message[2]  = CHAR_2;
assign message[3]  = CHAR_3;
assign message[4]  = CHAR_4;
assign message[5]  = CHAR_5;
assign message[6]  = CHAR_6;
assign message[7]  = CHAR_7;
assign message[8]  = CHAR_8;
assign message[9]  = CHAR_9;
assign message[10] = CHAR_A;
assign message[11] = CHAR_B;
assign message[12] = CHAR_C;
assign message[13] = CHAR_D;
assign message[14] = CHAR_E;
assign message[15] = CHAR_F;

assign char = message[address];
```

```
always @(posedge clk)
begin
  if (reset)
    begin
      message[0]  = CHAR_0;
      message[1]  = CHAR_1;
      message[2]  = CHAR_2;
      message[3]  = CHAR_3;
      message[4]  = CHAR_4;
      message[5]  = CHAR_5;
      message[6]  = CHAR_6;
      message[7]  = CHAR_7;
      message[8]  = CHAR_8;
      message[9]  = CHAR_9;
      message[10] = CHAR_A;
      message[11] = CHAR_B;
      message[12] = CHAR_C;
      message[13] = CHAR_D;
      message[14] = CHAR_E;
      message[15] = CHAR_F;
    end
end
```

```
always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

```
always @(posedge clk or posedge reset)
begin
  if (reset)
    begin
      message[0]  = CHAR_0;
      message[1]  = CHAR_1;
      message[2]  = CHAR_2;
      message[3]  = CHAR_3;
      message[4]  = CHAR_4;
      message[5]  = CHAR_5;
      message[6]  = CHAR_6;
      message[7]  = CHAR_7;
      message[8]  = CHAR_8;
      message[9]  = CHAR_9;
      message[10] = CHAR_A;
      message[11] = CHAR_B;
      message[12] = CHAR_C;
      message[13] = CHAR_D;
      message[14] = CHAR_E;
      message[15] = CHAR_F;
    end
end
```

```
always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

```
always @(posedge clk or posedge reset)
begin
  if (reset)
    begin
      message[0]  <= CHAR_0;
      message[1]  <= CHAR_1;
      message[2]  <= CHAR_2;
      message[3]  <= CHAR_3;
      message[4]  <= CHAR_4;
      message[5]  <= CHAR_5;
      message[6]  <= CHAR_6;
      message[7]  <= CHAR_7;
      message[8]  <= CHAR_8;
      message[9]  <= CHAR_9;
      message[10] <= CHAR_A;
      message[11] <= CHAR_B;
      message[12] <= CHAR_C;
      message[13] <= CHAR_D;
      message[14] <= CHAR_E;
      message[15] <= CHAR_F;
    end
  else
    begin
      char <= message[address];
    end
end
```

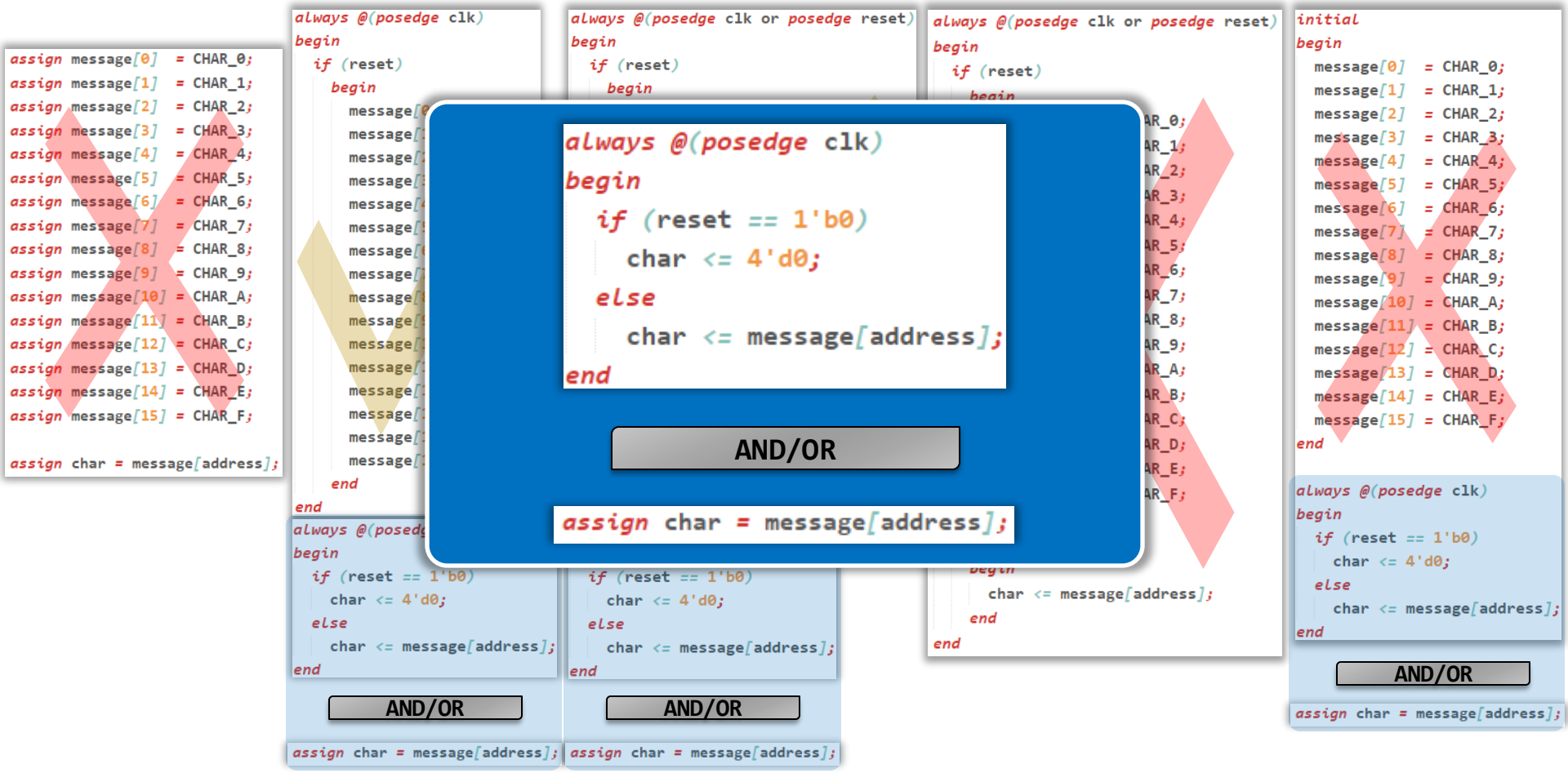
```
initial
begin
  message[0]  = CHAR_0;
  message[1]  = CHAR_1;
  message[2]  = CHAR_2;
  message[3]  = CHAR_3;
  message[4]  = CHAR_4;
  message[5]  = CHAR_5;
  message[6]  = CHAR_6;
  message[7]  = CHAR_7;
  message[8]  = CHAR_8;
  message[9]  = CHAR_9;
  message[10] = CHAR_A;
  message[11] = CHAR_B;
  message[12] = CHAR_C;
  message[13] = CHAR_D;
  message[14] = CHAR_E;
  message[15] = CHAR_F;
end

always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

Registers' Memory vs FPGA ROM

```
reg [3:0] message [0:15];
```

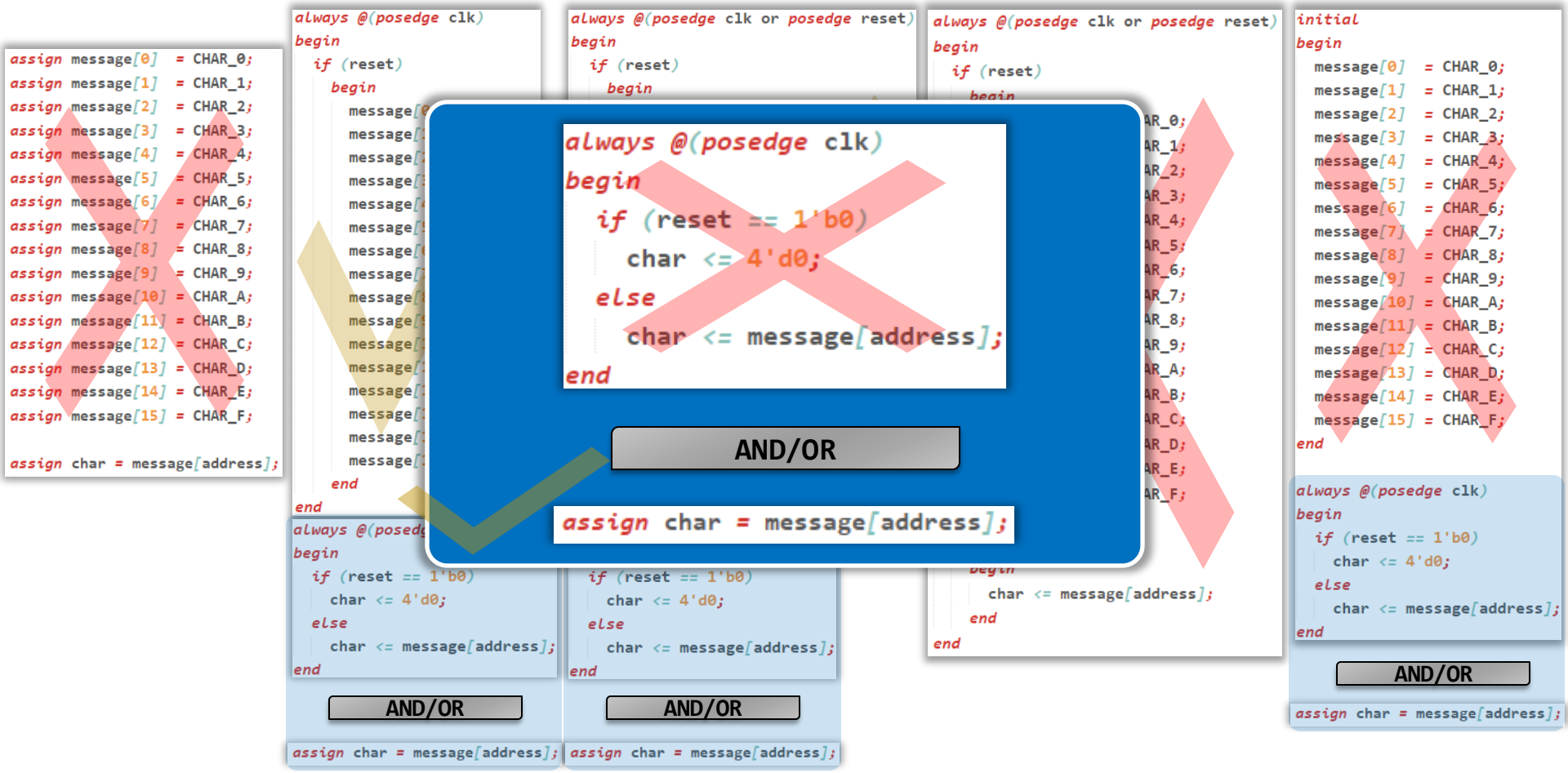
- Delivered code for required memory
  - Which are correct?



Registers' Memory vs FPGA ROM

```
reg [3:0] message [0:15];
```

- Delivered code for required memory
  - Which are correct?



Registers' Memory vs FPGA ROM

```
reg [3:0] message [0:15];
```

- Delivered code for required memory
  - Which are correct?

```
assign message[0]  = CHAR_0;
assign message[1]  = CHAR_1;
assign message[2]  = CHAR_2;
assign message[3]  = CHAR_3;
assign message[4]  = CHAR_4;
assign message[5]  = CHAR_5;
assign message[6]  = CHAR_6;
assign message[7]  = CHAR_7;
assign message[8]  = CHAR_8;
assign message[9]  = CHAR_9;
assign message[10] = CHAR_A;
assign message[11] = CHAR_B;
assign message[12] = CHAR_C;
assign message[13] = CHAR_D;
assign message[14] = CHAR_E;
assign message[15] = CHAR_F;

assign char = message[address];
```

```
always @(posedge clk)
begin
  if (reset)
    begin
      message[0]  = CHAR_0;
      message[1]  = CHAR_1;
      message[2]  = CHAR_2;
      message[3]  = CHAR_3;
      message[4]  = CHAR_4;
      message[5]  = CHAR_5;
      message[6]  = CHAR_6;
      message[7]  = CHAR_7;
      message[8]  = CHAR_8;
      message[9]  = CHAR_9;
      message[10] = CHAR_A;
      message[11] = CHAR_B;
      message[12] = CHAR_C;
      message[13] = CHAR_D;
      message[14] = CHAR_E;
      message[15] = CHAR_F;
    end
end
```

```
always @(posedge clk or posedge reset)
begin
  if (reset)
    begin
      message[0]  = CHAR_0;
      message[1]  = CHAR_1;
      message[2]  = CHAR_2;
      message[3]  = CHAR_3;
      message[4]  = CHAR_4;
      message[5]  = CHAR_5;
      message[6]  = CHAR_6;
      message[7]  = CHAR_7;
      message[8]  = CHAR_8;
      message[9]  = CHAR_9;
      message[10] = CHAR_A;
      message[11] = CHAR_B;
      message[12] = CHAR_C;
      message[13] = CHAR_D;
      message[14] = CHAR_E;
      message[15] = CHAR_F;
    end
end
```

```
always @(posedge clk or posedge reset)
begin
  if (reset)
    begin
      message[0]  <= CHAR_0;
      message[1]  <= CHAR_1;
      message[2]  <= CHAR_2;
      message[3]  <= CHAR_3;
      message[4]  <= CHAR_4;
      message[5]  <= CHAR_5;
      message[6]  <= CHAR_6;
      message[7]  <= CHAR_7;
      message[8]  <= CHAR_8;
      message[9]  <= CHAR_9;
      message[10] <= CHAR_A;
      message[11] <= CHAR_B;
      message[12] <= CHAR_C;
      message[13] <= CHAR_D;
      message[14] <= CHAR_E;
      message[15] <= CHAR_F;
    end
  else
    begin
      char <= message[address];
    end
end
```

```
initial
begin
  message[0]  = CHAR_0;
  message[1]  = CHAR_1;
  message[2]  = CHAR_2;
  message[3]  = CHAR_3;
  message[4]  = CHAR_4;
  message[5]  = CHAR_5;
  message[6]  = CHAR_6;
  message[7]  = CHAR_7;
  message[8]  = CHAR_8;
  message[9]  = CHAR_9;
  message[10] = CHAR_A;
  message[11] = CHAR_B;
  message[12] = CHAR_C;
  message[13] = CHAR_D;
  message[14] = CHAR_E;
  message[15] = CHAR_F;
end

always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
always @(posedge clk)
begin
  if (reset == 1'b0)
    char <= 4'd0;
  else
    char <= message[address];
end
```

```
assign char = message[address];
```

```
assign char = message[address];
```

```
assign char = message[address];
```

Memory/Register Initialisation

```
reg [3:0] message [0:15];
```

We can use an ***initial*** block only for registers & some ROMs/RAMs **only in FPGA design**, **but it is NOT ASIC-proven**.

Basically, we **JUST** define *their initial value when the core turns on.*

```
initial
begin
  message[0]  = CHAR_0;
  message[1]  = CHAR_1;
  message[2]  = CHAR_2;
  message[3]  = CHAR_3;
  message[4]  = CHAR_4;
  message[5]  = CHAR_5;
  message[6]  = CHAR_6;
  message[7]  = CHAR_7;
  message[8]  = CHAR_8;
  message[9]  = CHAR_9;
  message[10] = CHAR_A;
  message[11] = CHAR_B;
  message[12] = CHAR_C;
  message[13] = CHAR_D;
  message[14] = CHAR_E;
  message[15] = CHAR_F;
end
```

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**Golden Rule #1**)*
  - Combinational vs Sequential always blocks *(**Golden Rule #2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**Golden Rule #4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

XDC Constrains Usage & Justification

- **Warning-Free vs Ignoring Warning*(s)***
  - you have to justify both
  - both choices are equally important
    - Meaning we expect from you to resolve upcoming issues
      - **with Critical Thinking**

- So, whenever you ignore or add/implement something,
  - You should be able to explain
    - *Why?*
    - *How it works?*
      - i.*e.,* **`VOLTAGE_CONFIG`**

Wrong Initialisation on variables

```
module test_counter(clk, reset, count)
    input clk, reset;
    output [3:0] count;
    reg [3:0] count;

    wire enable = 1'b1;
    reg var_1 = 1'b1;
```

It is **not valid** to initialise registers and wires like this!
It is not like C that we can set initial values on variables!

```
module test_counter(clk, reset, count)
    input clk, reset;
    output [3:0] count;
    reg [3:0] count;

    wire enable;
    reg var_1;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
        begin
            var_1 <= 1'b1;
            count <= 4'b0000;
        end
        else
            count <= count + 1'b1;
    end

    always @(var_1)
    begin
        if (var_1)
            enable = 1'b1;
        else
            enable = 1'b0;
    end
```

All registers should get initial value on reset!

All wires should get value according to register's value

Overview

- Fundamental Acquired Skills after Completing Lab Assignment 1
  - Dataflow & Top-level Module *(**G**olden **R**ule **#1**)*
  - Combinational vs Sequential always blocks *(**G**olden **R**ule **#2**)*
    - Minimal Implementation with Sequential Elements
  - Reset & Clock Logic *(**G**olden **R**ule **#4**)*
- Review Implementation Faults
  - Hierarchical Design, Coding Style & Comments
  - Combinational Logic Reset
  - Anode Driver Refresh Period
  - Debouncer
  - MMCM Reset
  - MMCM Feedback
  - MMCM VCO Minimum Frequency
  - Registers' Memory vs FPGA ROM
  - XDC Constraints Usage & Justification
- Fundamental Acquired Skills after Completing Lab Assignment 2
  - FSMs
  - Testbench
  - Satisfy Timing Requirements

Questions