FPGA Programming: Lab Tasks

Last update : 05.02.2019

Prof. Dr. Mehdi Tahoori / Arun Vijayan, Dennis Gnad, Ahmet Erozan

CDNC - ITEC - KIT

Contents

4-bit Full-Adder and Simple ALU

First step: create a 4-bit adder and show the output result using the LEDs. In your program, you should have two 4-bit input sets that will be added together. Think about the overflow, your output should be 5-bits long. Design your code such that whenever an input is changed the task is performed, e.g.

```
// define module entity
module four_bit_adder( A_in, B_in, Sum_out );
//define inputs and outputs
input [3:0] A_in;
input [3:0] B_in;
output [4:0] Sum_out;
reg [4:0] Sum_out;
// do the addition whenever A_in or B_in change states
always@( A_in or B_in ) begin
....write code to add A_in and B_in and assign it to output
end
endmodule
```

Hint: Compile your project once through menu (Processing ⇒ Start Compilation) or just press small play button on the toolbar. Next, open pin planner through menu Assignments → Pin Planner. Note: If you don't compile your program before assigning pins, you will not be able to see any listings on the Node Name and Direction columns in Pin Planner window.

The node name refers to your input and output signal names from the program. Now assign, input signals to switches and output signals to LEDs. On page 35 of Terasic DE2-115 User Manual, you can find pin names and locations. Assign corresponding pin locations to your signals and compile the program again. Then you can use the programmer to run your code onto the FPGA module.

Next step: In this step, you will create a simple ALU as an extension of your earlier exercise. Now, you add 2-bit control sequence as input in addition to your earlier inputs. Instead of just performing addition, now the control sequence decides which operation is to be performed, refer to table 1. Again, show your output using LEDs.

Control Sequence	Operation
00	Adder
01	OR
10	AND
11	XOR

Simple sequential counting circuit

In the last task you have been introduced to combinational logic, without a memory. That means, state-free logic that only implements boolean functions. You can also understand it as a directed acyclic graph of boolean operations. In this way, you can only implement functions that are independent and do not depend on previous states, as the examples of the last task. In this task, in order to perform sequential operations, registers (i.e. *flip-flops*) are required, to save the intermediate states. In terms of graph theory, that means we allow cycles in the graph. An example of graph representations are shown in 1, and another example is given as an implementation view, as a schematic block diagram in 2.

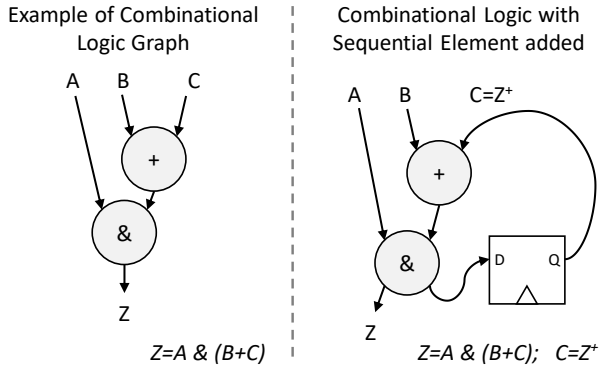

Figure 1: Graph Representation of Combinational and Sequential Logic

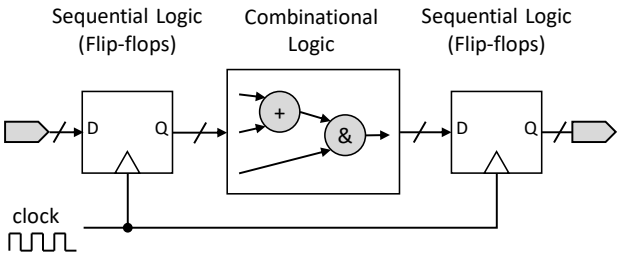

Figure 2: Block Layout Representation of Combinational and Sequential Logic

In this task you will now implement a simple counter that toggles a LED on and off. You should use the internal clock of the FPGA (50 MHz) to count seconds and blink one of the LEDs each second. Create a state register that holds a high or low value and assign it to an LED output variable. You should think about the time the state register is either high or low, so that the blinking is observable. Start off with 1 second of period between each blinks, which means the LED will be high for 0.5 seconds and low for 0.5 seconds as well.

Hint: clock is an input to your program but it is triggered internally through an on-board oscillator. You just have to declare the clock as an input signal variable to your module and assign it to the corresponding location through pin planner, e.g PIN_Y2 is a 50 MHz clock source input. You can find clocks and pin

locations in the DE2-115 user manual.

For a basic counter that increments every clock cycle, the following always-block is already sufficient:

```
always@(posedge clock) begin
 counter <= counter + 1;
end
```

In this case, in one second, the counter will be incremented 50000000 times due to the 50MHz clock. Now please extend it to toggle a LED every 0.5 seconds. The toggling can be done using an XOR-Operation, i.e.

```
LED <= LED ^ 1b'1;
```

(be aware of potential copy-and-paste issues from PDF with quoting-characters)

Simple state machine

A very commonly used model to implement sequential logic is through finite state machines. Please implement such a state machine that automatically transitions from state to state after a few seconds, and has at least three states. In each state it should show a different LED output pattern. Here we already provide a template for such a state machine that still needs a counter that generates a 'time_elapsed' signal, and in which you still need to fill-in a few gaps:

```
module state_machine( clk, LED_out );
input clk;
output [7:0] LED_out;
reg [1:0] state;
parameter S0=0, S1=1, RST=2;

// counting logic that generates a synchronous time_elapsed signal
// it can be used to transition between states after a certain time
always@(..
 TODO
end

// state transition logic
always@(posedge clock) begin
```

```
  case(state)
    RST : begin
                if (time_elapsed) begin
                    state <= S0;
                end
          end
     S0 : begin
            TODO
          end
     S1 : begin
            TODO
          end
    default : begin
                    state <= RST;
              end
  endcase
end

// combinational output logic
always@(*) begin
  case(state)
    RST : begin
                 LED_out <= 8'b00001111;
          end
    S0 : TODO
    S1 : TODO
    default : TODO
  endcase
end
```

Input Sampling and Basic Sequence Detector

In this task, you will design a sequence detector which detects a predefined sequence. Since input sequence should be entered by using switches on the FPGA board, sampling the switch input is an important issue in this task.

First step: You need to sample one switch ($SW0$) with a button ($BT0$). After selecting the input bit (change the SW0 to desired value), BT0 is changed to 1 and 0 sequentially by pressing it, sampling the value

from SW0. To verify that you successfully sample the input, you can use a LED on the board.

Hint: You should be able to capture that BT0 signal is rising. To do that, BT0 is sampled (BT0_r) and the level of BT0 and BT0_r is compared (for example: when BT0 becomes 1, BT0_r is 0 for one clock cycle). Please keep in mind that you should not use the direct BT0 from the input pin, but also sample it in a clock-synchronous register to prevent glitches, i.e.:

```
always@(posedge clock) begin
 BT0 <= BT0_pin;
 BT0_r <= BT0;
end
```

Second step: In this step, design a Finite State Machine that detects the pattern "101" in the last three inputs and shows a '1' for each detection. For instance, if the input sequence is "0000101000010101000", the output should be "0000000100000101000". To verify, you can use a LED on the board.

Using Counters and 7-segment Display

First step: Please start with the counter implemented in .

Second step: In the second step, you will create a counter that shows on the 7-segment displays and goes from starting value 0 to 9 and repeats the pattern. Look at the DE2-115 manual on how to use 7-segment displays in your program. Remember, each segment is turned on when it has a low value. Hint: In your program, you can use as many *always@* modules as you want. Try to seperate functions like this for clearness. A simple 4-bit counter is sufficient to count up to 9. The counter updates in each second like the previous simple counter. Now, create a second counter inside it. You can then, for instance, update your 0-9 counter with a conditional assignment, in such a way:

```
BCD <= (BCD==4'h9 ? 4'h0 : BCD+4'h1)
```

Here, BCD is a 4-bit register that holds values from 0 to 9. You can use the value that BCD holds from a seperate always@ block and assign it to output 7-segments.

Third step: If you have successfully completed the previous task, you are now ready to create a full-fledged digital clock. It is just an extension of the previous task, except that you have now counters for seconds, minutes and hours. Update each of these counters each second from your main *always@* block. Use another *always@(*)* block with a bunch of case statements, which changes the state of 7-segment display units based on current values of each timer counter. After having the digital clock ready, add more features to it, for

example incrementing and decrementing minutes and hours counters through pushbuttons on the FPGA board.

Automatically Glowing and Fading LED

In this exercise, the task is to make one of the LEDs glow starting from least possible intensity ("low always") to maximum possible intensity ("high always") by appropriately driving the LED output.

Figure 3: Sample intensity changes.

Look at Figure 3 to see how your led intensity must change from low (all white in the picture) to high (all dark in the picture). The square boxes represent the state of a led in time, in terms of its intensity. The first box, corresponding to all white means that green or red LED in your FPGA board is turned off. Subsequently, the boxes increasing in darkness towards the right in Figure 3 speak for the increase in intensity, i.e. its color whether its green or red. Again, the last box at the far right filled with all black corresponds to a led at its highest possible intensity, i.e. it is lit up all the time (it is always high!). So simply as:

Case 1:

```
assign LED_out = '1';
```

If you are confused with the idea of a LED having different intensity from low to high, start out by assigning a value 1 (high) to a LED all the time, and then toggle the value in each clock cycle, i.e. for 1 clock cycle it will have a high value and the next cycle a low value, for example like the following:

Case 2:

```
reg LED_state = 1;
// ...

@always( posedge clk ) begin
 LED_state = ~LED_state;
end

assign LED_out = LED_state;
```

In **Case 1**, with a high value for *LED_state* all the time, i.e. without using clock triggered *@always* block, you should observe the LED at its highest possible intensity, thus should be bright RED or bright GREEN color. On the other hand, with the **Case 2** approach of toggling the state, you will observe intensity reduced by half! The fact is, any frequency change above 100Hz is usually too quick for the human eye to observe, and it averages the intensity.

So, considering the 50 MHz clock frequency of Altera DE2-115 FPGAs used in the lab, the LED changes its intensity from high to low with frequency 25MHz, too fast for the human eye to observe the changes. That means, it **averages** the highest (max. brightness, = '1') and lowest (dark off state, = '0'), so you should observe $\frac{1+0}{2} = 0.5 \Rightarrow 50\%$ of the intensity compared to **Case 1**. This observation should be the basis for automatic glowing and fading of LEDs in your current exercise.

Hint: By carefully taking the overflow of an addition of 3-bit registers you can have 8 intensity levels, with 4-bit registers 16 levels, and so on. Remember, the intensity must change in time from low to high and back to low automatically, and this depends on how fast the state of your registers change. Keep clock frequency in mind when designing such a system, to control how fast it glows and fades back.

Working with Embedded Memory Blocks

In this experiment, it is required that a simple memory block be created through quartus IP Catalog. You are going to create **RAM: 1-Port IP Core** for use in your project. The purpose of this RAM module is to store pattern of bit sequences that can be used to drive certain LEDs on the FPGA in a pre-defined manner. For this exercise, you need to create a memory file with extension .HEX or .MIF that has pre-defined pattern of bit sequences. Refer to appendix on how to create one such file using Quartus II.

Once you have either .MIF or .HEX memory file ready. Follow the following steps to create a RAM IP Core module and integrate it into your project. This project simulates the behaviour of a RAM hardware.

The RAM will hold X words (number of different on-off patterns) of Y bits to use a certain number of LEDs on the bottom edge of the FPGA board (so you could choose 26 to have a full LED pattern saved in one word). The number of words will define how many different patterns of bit streams can be stored in the RAM. Find the ALTERA Embedded Memory user guide for detailed specifications before following the next steps.

1. Invoke IP Catalog (Tools → IP Catalog) after creating a new project. Then, expand the "On Chip Memory" menu list, and double click on RAM: 1-Port, see Figure 4. Enter IP variation file name, and select Verilog as the file type. Click OK
2. You should have a MegaWizard window open now, like the one shown in Figure 5. Select the appro-

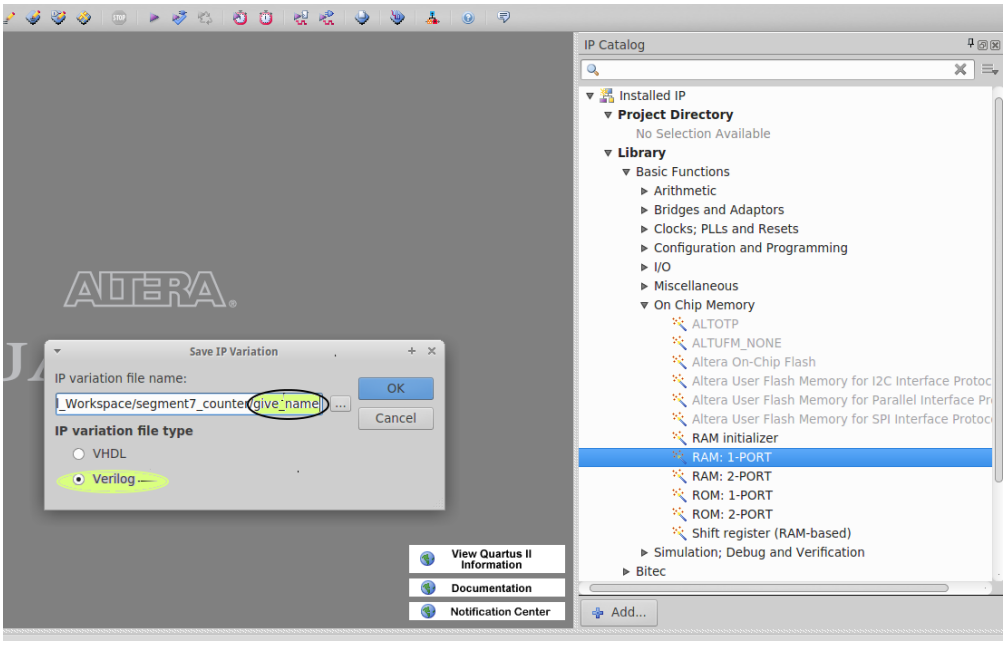

Figure 4: Accessing IP catalog on Quartus II.

priate number of bits and words corresponding to the memory file created. The one given to you has 26 bit output (uses all the LEDs on the bottom of the FPGA) and 256 words (256 different patterns of 26 bits). Leave everything else as it is, Click Next.

3. In the current MegaWizard page 2 of 6, Check read enable signal, this signal indicates to the RAM module if read operation is to be performed. There is already write enable signal (wren) created by default, which signals if the write operation is to be performed on the memory module. However, we do not care about this for now since we have already created a memory file and we just need to perform read operations. Click Next. Select Old Data from the drop-down menu on the next page about Single Port Read-During-Write Option. Click Next.

4. In this step, MegaWizard page 4 of 6, select the option to set initial contents of the memory using the file created, see appendix. Specify the path to the file. Click Next. On MegaWizard page 5 of 6, click Next.

5. The last page, i.e. 6 of 6 on MegaWizard, make sure the last two options are checked as shown in Figure 6. Click Finish. After successfully creating the memory block you will be asked if you would like to add it to your project, click YES and you are ready. Now you just need to instantiate the module in your main Verilog file with valid signals.

6. Now, add a new Verilog file to your project. Define input and output signals. Output signal will be the length of the word connected to LEDs. Input will be the Clock signal. Moreover, you need to connect

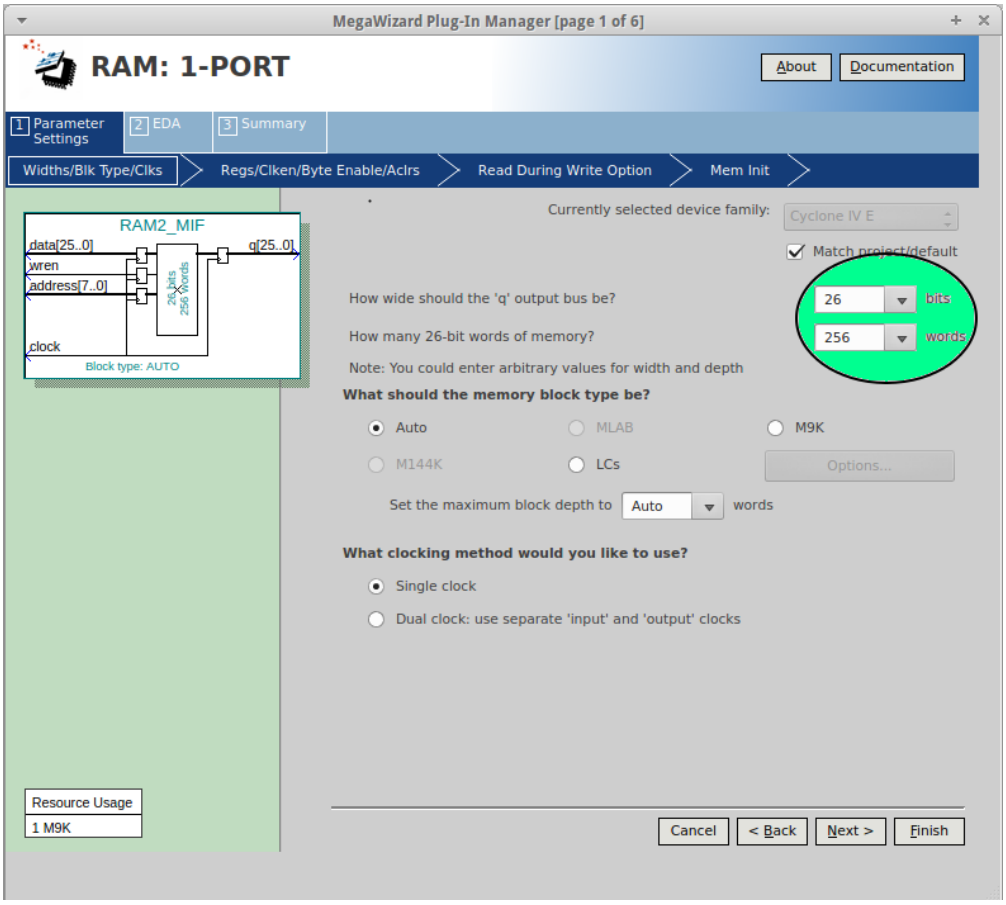

Figure 5: Megawizard RAM: 1-PORT Configuration.

appropriate signals to the RAM module that has been created. Remember, the RAM module created is a hardware block, it has following signals you need to think about:

address: This input signal to RAM module holds the value of the address for the data you are about to read or write from. If you have 256 words in total, it would be an 8-bit long sequence going from 0-255.

clock: This is the input clock of the RAM module. You can connect the internal clock of the FPGA to the clock signal of RAM module. An example instantiation of the RAM module with all the signals is given below.

data: This is an input signal to the RAM module. If you were to write data to RAM block, you change this signal and corresponding address.

wren: This is also an input signal to the RAM module. This signals if the data is to be written to memory, thus called write enable. It is one-bit long, true (1) or false (0).

rden: Similar to write enable, this signal indicates read enable. It must be true when reading data from certain RAM address.

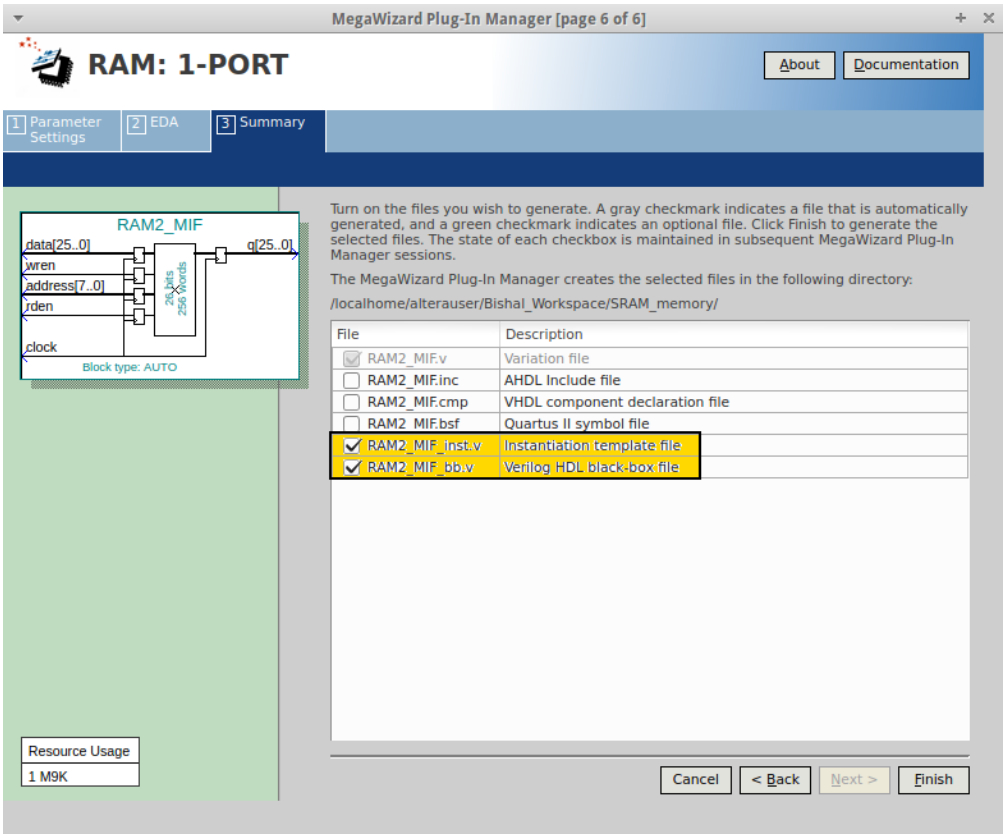

Figure 6: Megawizard RAM: 1-PORT Configuration Final Step.

q: This is an output signal of the RAM block and which holds output data. Based on address signal and read enable signal it holds data corresponding to certain address from the memory. This signal's data should be used to drive the LEDs output.

Example instantiation of the memory block in your main Verilog file,

```
RAM1 my_ram(
.address (addr),
.clock (CLOCK_50),
.data (led_bit_streams),
.wren (write_enable),
.rden (read_enable)
.q (data_temp)
);
```

In the above example, RAM1 is the variation name of the memory block and my_ram is an instance of this block. The signals of the main module: addr, CLOCK_50, led_bit_streams, write_enable, read_enable, data_temp are connected to the signals of the memory block: address, clock, data, wren, rden, and q. Try

varying the speed of read operation, i.e. the frequency of change of signal addr from the main Verilog file, corresponding to above example instantiation.

Hint: It is possible to create a .HEX or .MIF file using Quartus II. Follow the next link to start out and refer to the figures below.

`http://quartushelp.altera.com/14.0/mergedProjects/design/med/med_pro_med_files.htm`

Task: You are free to use the memory in any reasonable way. Please show us any kind of use of the memory module. For example, display a new LED pattern from the memory each second.

Requirements for the following two tasks

For the following two tasks we will be more strict. Please follow the following points:

- Before starting to implement, discuss with us if you understood the task correctly. The advanced sequence detector can be tricky.
- Make sure to not infer latches by accident. Use of latches are an advanced topic and can have many issues. They are also just rarely required in standard synchronous digital design. To check that you did not introduce any latches by accident, check in the Compilation Report → Analysis & Synthesis → Messages. Filter the output for "latch", there should not be any match! Otherwise look at the affected code regions. Do not use posedge on anything but the clock from the board, and always consider all 'else' and 'default' cases.
- Please keep an eye on the "Total used logic elements" in the Compilation Report. Typically, both of the two following tasks should not require more than 300 each. When you are using "SignalTap" for debugging, it could be more. In that case, please check your design without SignalTap, after it is working.

Vending Machine using FSM

In this task, a coffee/tea vending machine will be designed by using an FSM. First, the machine should get coins (switches) and display the total amount (7-segment). Then, coffee or tea is selected and display the selection (LEDs) and display the balance (7-segment).

- Modes: 1)Coffee 2)Tea

- Take 5,3, 2,1 coins as inputs.
- Display the amount already inside the machine.
- Deliver a coffee at $15 (light an LED) if coffee is selected.
- Deliver a tea at $10 (light another LED) if tea is selected
- Return the balance (display the balance)
- Use a switch or time interval to reset the machine.
- Be careful about some corner cases:
  - Do not allow inserting more coins during reduction of the balance (not in the same clock cycle), because in rare cases that could lead to an invalid balance
  - Prevent any potential cheating attempt
  - Do not allow serving coffee and tea at the same time
  - Do only allow valid balances in the machine. That means, do only allow as many coins as the machine can reasonably count - if you want to simplify, just allow coins inserted when coffee or tea is already selected. Then, the maximum is 19.

Design of an Advanced Sequence Detector

Detect an 8 bit binary sequence when entered one bit at a time. For each matching bit, the bit pattern entered should be displayed on seven-segment display. The 8 bit pattern should be user-defined.

For example, let us consider the input sequence to be detected as "01100100". Initially, the user should be able to define/set this sequence to be detected using switches. Afterwards, the user should be able to enter continuous input sequences using push buttons (i.e., user should be able to give either a 0 or a 1 as input bit at any instant). In this example, if the user enters in the order 011 then 011 should be displayed on the seven-segment display. If the fourth input is 1, then the red led should glow, rejecting a match (display should be cleared). Once the whole pattern "01100100" is entered, a green LED should indicate the match of 8 bit sequence.

On the other hand, in this example, if the user enters in the order 011001, 011001 should be displayed because it is valid. If the seventh input is 1, then the display should change from 011001 to 011 instead of being cleared, because these are the last three bits that were entered and are the valid first three bits of our example sequence "01100100".

After a full valid match, consider either resetting to a state where new matches can happen on the same initial sequence or consider following new matches based on the already entered bits.

Additional Tasks

These are for those who finish other tasks. Before you start, please discuss with one of the lab advisors.