EE 2420 Lab Guide 4: Unipolar Stepper Motor State Machine

Written By: Grant Seligman, Gabe Garves, James Starks

Example Overview:

Design a circuit that will be capable of controlling a 4-phase unipolar stepper motor by generating outputs in the following sequence:

ABCD => 0011 0110 1100 1001 0011 …

Notice A and C are opposites. Whenever A is 1, C is 0, and vice versa. The same is true for B and D. This will simplify things and reduce the amount of hardware needed. Below is the circuit diagram for the counter:

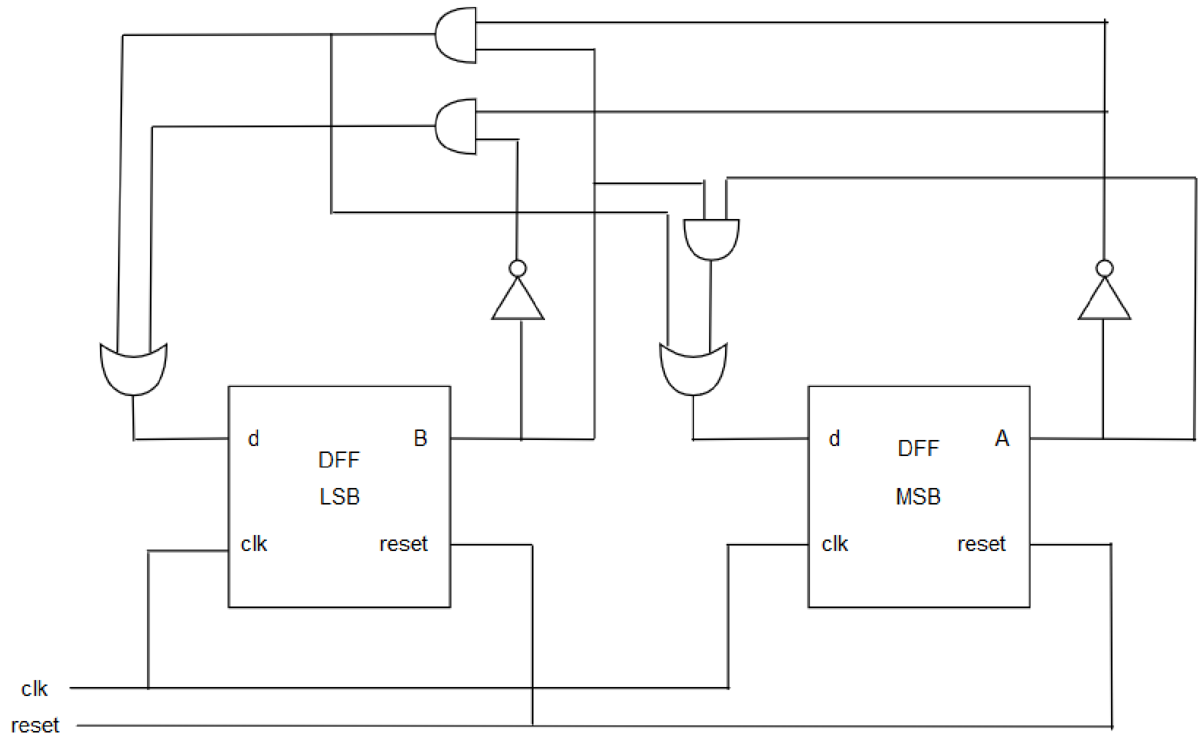
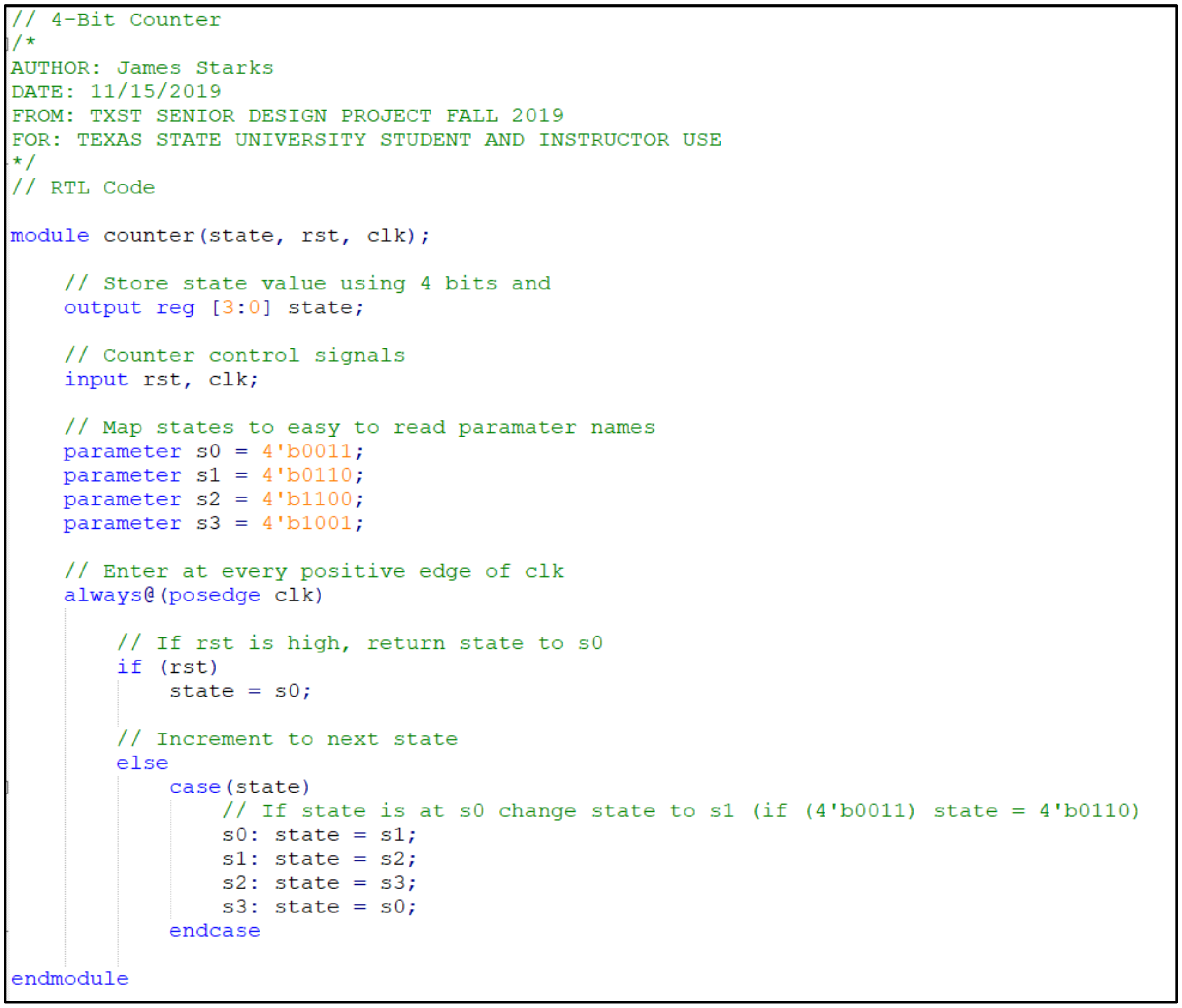


Figure 1

Verilog Code Breakdown:

Option 1: For this lab, it is possible to write a D-flip flop module and a counter module to call the DFFs separately at the structural level. But what makes verilog useful is that this counter can be created using RTL level to simplify the code/module structure. At this level one can write the counter as a basic state machine using case statements as shown in the code below. Also a new primitive shown is the parameter. It is used to hold a constant data value locally to the module. In this case, these parameters hold the actual state values for the counter. It isn't necessary to pass the counter values as parameters but it allows one to name data for ease of use.



```
// 4-Bit Counter
/*
AUTHOR: James Starks
DATE: 11/15/2019
FROM: TXST SENIOR DESIGN PROJECT FALL 2019
FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTOR USE
*/
// RTL Code

module counter(state, rst, clk);

    // Store state value using 4 bits and
    output reg [3:0] state;

    // Counter control signals
    input rst, clk;

    // Map states to easy to read paramater names
    parameter s0 = 4'b0011;
    parameter s1 = 4'b0110;
    parameter s2 = 4'b1100;
    parameter s3 = 4'b1001;

    // Enter at every positive edge of clk
    always@(posedge clk)

        // If rst is high, return state to s0
        if (rst)
            state = s0;

        // Increment to next state
        else
            case(state)
                // If state is at s0 change state to s1 (if (4'b0011) state = 4'b0110)
                s0: state = s1;
                s1: state = s2;
                s2: state = s3;
                s3: state = s0;
            endcase

endmodule
```

Figure 2

Every time clock goes high rst condition is checked. If rst is high the current state is reset back to state zero - else, the case statement is entered. Depending on the current state, the correct case is triggered and the state changes to the next desired state.

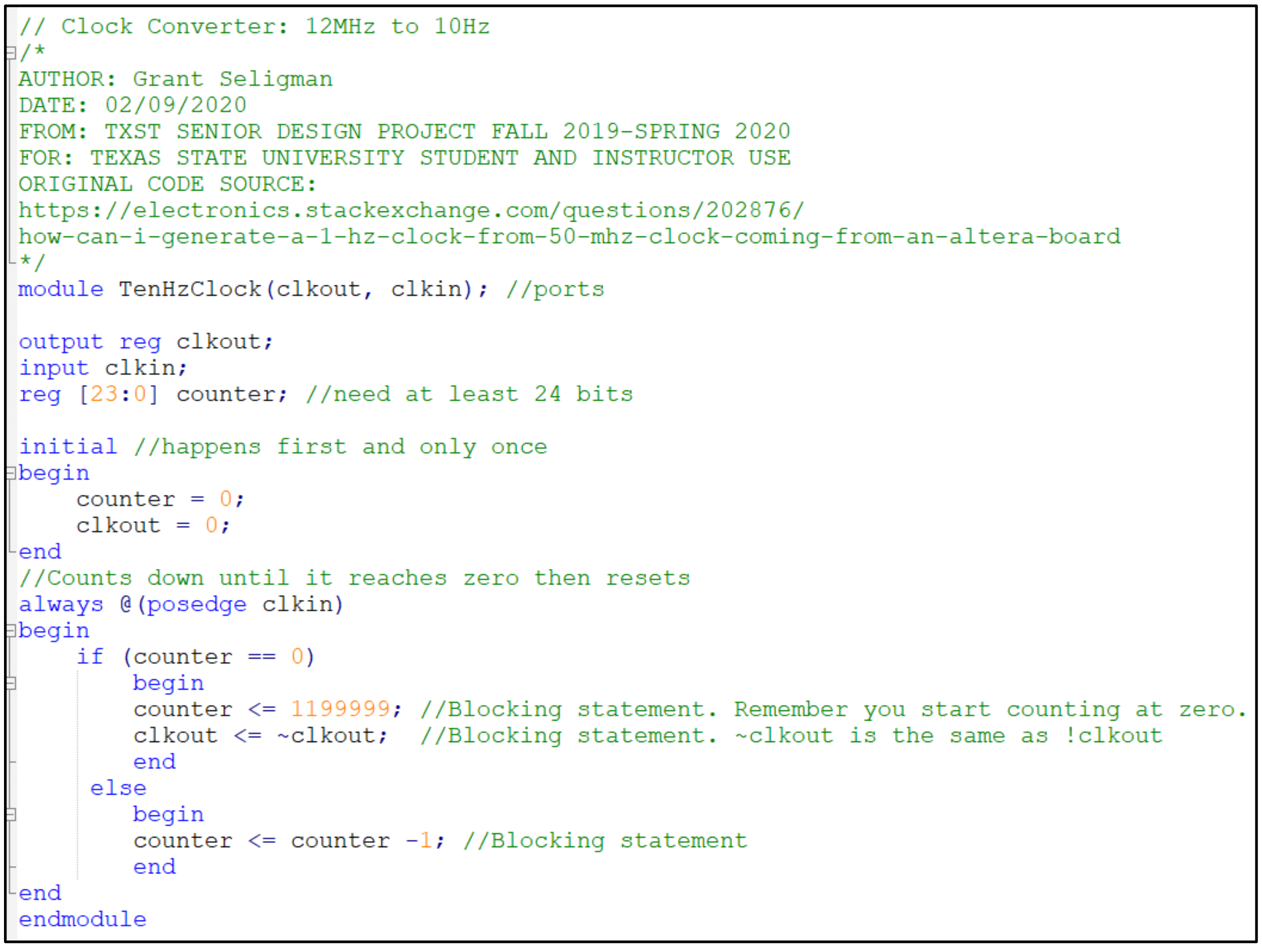
Option 2: This option will introduce you into working with more than one block module in Quartus. Given that most ASIC designs have quite a bit of modules, it’s good to get familiar with this process as well as writing modules to take in clock inputs such as this one shown in **Figure 3.** In order to get the LEDs to blink, one needs a clock signal to run the counter module. 12Mhz is too fast for the human eye to perceive so one needs to step the clock down to a much slower rate. For this example, 10Hz was chosen but the code can be easily modified to one’s liking. First, it’s important to know the amount of bits in binary it will take to handle 12 million. From digital logic, all one has to do is divide the log of the clock signal by the log of the base. Make sure to round up to the nearest bit value:

$$log(12E6)\ /\ log(2)\ \approx\ 24\ bits$$

Then in order to get 10 clock cycles out of 12 Million:

$$12E6\ /\ 10\ =\ 1.2E6\ counts\ per\ cycle$$

Once both these values are known, one can build another counter that counts down 1.2E6 bits before changing the new clock output. This could be done with an up counter but the code shown in **Figure 3** is a down counter. Another thing to mention is the blocking statement. In Verilog, <= represents a blocking statement. Usually verilog code is executed in parallel but with a blocking statement, the arguments are executed from top-down which is seen similarly in C/C++. This can be useful with specific variables or statements that depend on the actions of another. From the code, the counter is initially set to zero, then is set to 1199999. Once the counter reaches zero again, the clock flips. If one wishes to go slower, let's say 1Hz, change the 1199999 to 11999999.



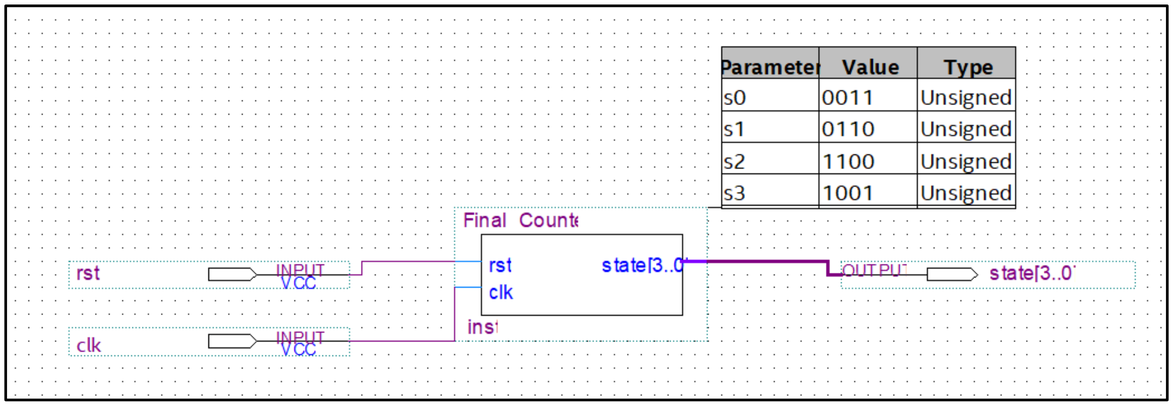
```
// Clock Converter: 12MHz to 10Hz
/*
AUTHOR: Grant Seligman
DATE: 02/09/2020
FROM: TXST SENIOR DESIGN PROJECT FALL 2019-SPRING 2020
FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTOR USE
ORIGINAL CODE SOURCE:
https://electronics.stackexchange.com/questions/202876/
how-can-i-generate-a-1-hz-clock-from-50-mhz-clock-coming-from-an-altera-board
*/
module TenHzClock(clkout, clkin); //ports

output reg clkout;
input clkin;
reg [23:0] counter; //need at least 24 bits

initial //happens first and only once
begin
    counter = 0;
    clkout = 0;
end
//Counts down until it reaches zero then resets
always @(posedge clkin)
begin
    if (counter == 0)
        begin
        counter <= 1199999; //Blocking statement. Remember you start counting at zero.
        clkout <= ~clkout;  //Blocking statement. ~clkout is the same as !clkout
        end
     else
        begin
        counter <= counter -1; //Blocking statement
        end
end
endmodule
```

Figure 3

The original code for the counter was sourced from a stackexchange webpage. [1] *The module was modified by Grant Seligman.*

FPGA Implementation:

Option 1: Using your knowledge from the previous labs, implement your verilog code into a **symbol** for each module. The **rst** will be connected to a switch and the **clk** will be connected to a function generator that is outputting a **0V to 3.0V 10 Hertz** square wave. Which both pins can be set to any of the orange pins shown in **Figure 9.** The output will be the LEDs on the Max1000 board and the LED pinout will be listed on **Table 1.** The symbol file should look similar to **Figure 4. Now all pins given the table are suggestions and can be readjusted.**



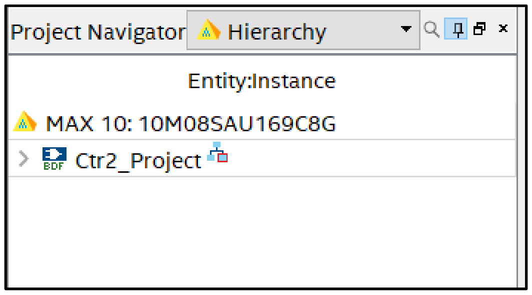
Parameter	Value	Type
s0	0011	Unsigned
s1	0110	Unsigned
s2	1100	Unsigned
s3	1001	Unsigned

Figure 4

Table 1

Node Name	Direction	Location	I/O Standard
rst	Input	PIN_L12	3.3-V LVTTL
clk	Input	PIN_J12	3.3-V LVTTL
state[3]	Output	PIN_B10	3.3-V LVTTL
state[2]	Output	PIN_C9	3.3-V LVTTL
state[1]	Output	PIN_C10	3.3-V LVTTL
state[0]	Output	PIN_D8	3.3-V LVTTL

Option 2: If a function generator is not available, you can use the on-board 12Mhz clock and the Clock Converter module to produce the 10 Hz clock signal and connect the output to the **clk** of the counter module. **It is important that if you plan to do it this way, make sure to set the finished .bdf file as your Top Level Entity before compiling and then setting the pins. This sets the entire block chain as one entity and allows Quartus to set up the pin planner correctly. Follow Figures 5 - 7 to set the Top Level Entity.**



Entity:Instance
MAX 10: 10M08SAU169C8G
Ctr2_Project

Figure 5

Go to the **Project Navigator** and use the drop down menu and select **Files.** This will show you all the current files within your Quartus Project.

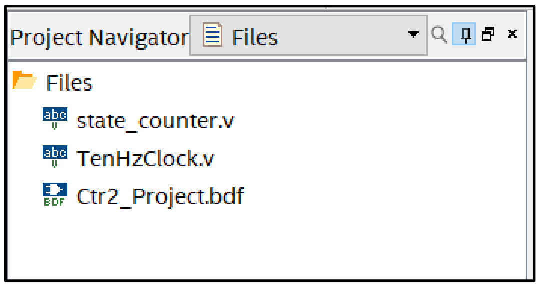


Figure 6

Right-click on the **.bdf** file of your project and select **Set as Top-Level Entity.**

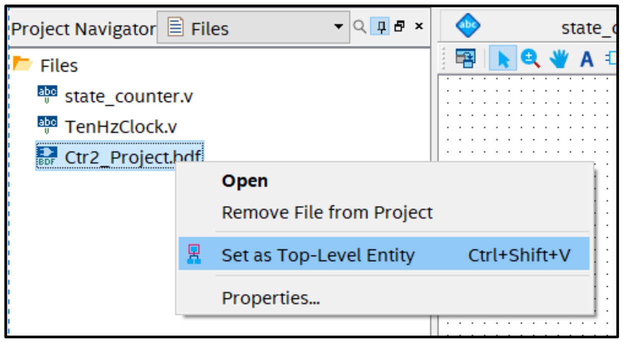
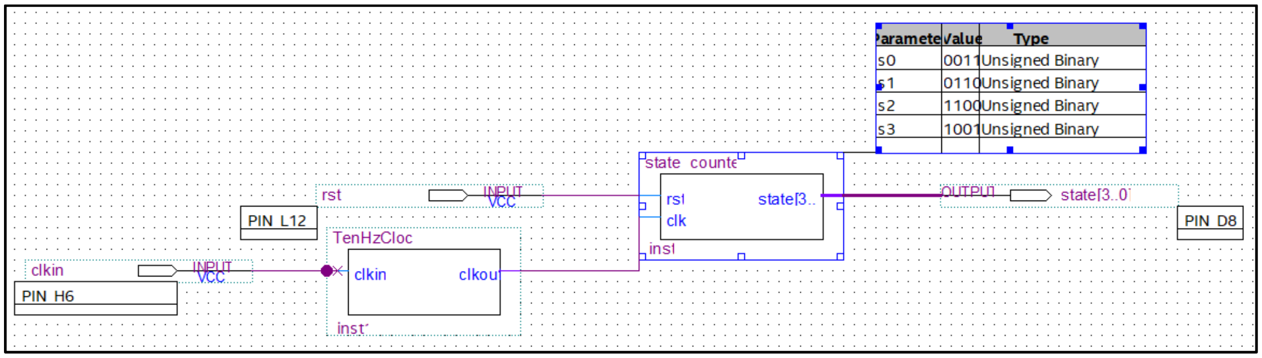


Figure 7

The entire block diagram should look similar to this. Once you have set your pins, compile the program again, then you are set to program the FPGA. Once the program is loaded, switch the reset on then off to start the program. **Table 2** shows the proper pin setup. **PIN_H6** is the on-board 12MHz clock. This is useful later if one plans to program Phase Lock Loops (PLLs).



aramete	Value	Type
s0	0011	Unsigned Binary
s1	0110	Unsigned Binary
s2	1100	Unsigned Binary
s3	1001	Unsigned Binary

Figure 8

As a note: The name of the block modules may be different depending on what you name your .v file when you create/import it into Quartus.

Table 2

Node Name	Direction	Location	I/O Standard
rst	Input	PIN_E6	3.3-V LVTTL
clkin	Input	PIN_H6	3.3-V LVTTL
state[3]	Output	PIN_B10	3.3-V LVTTL
state[2]	Output	PIN_C9	3.3-V LVTTL
state[1]	Output	PIN_C10	3.3-V LVTTL
state[0]	Output	PIN_D8	3.3-V LVTTL

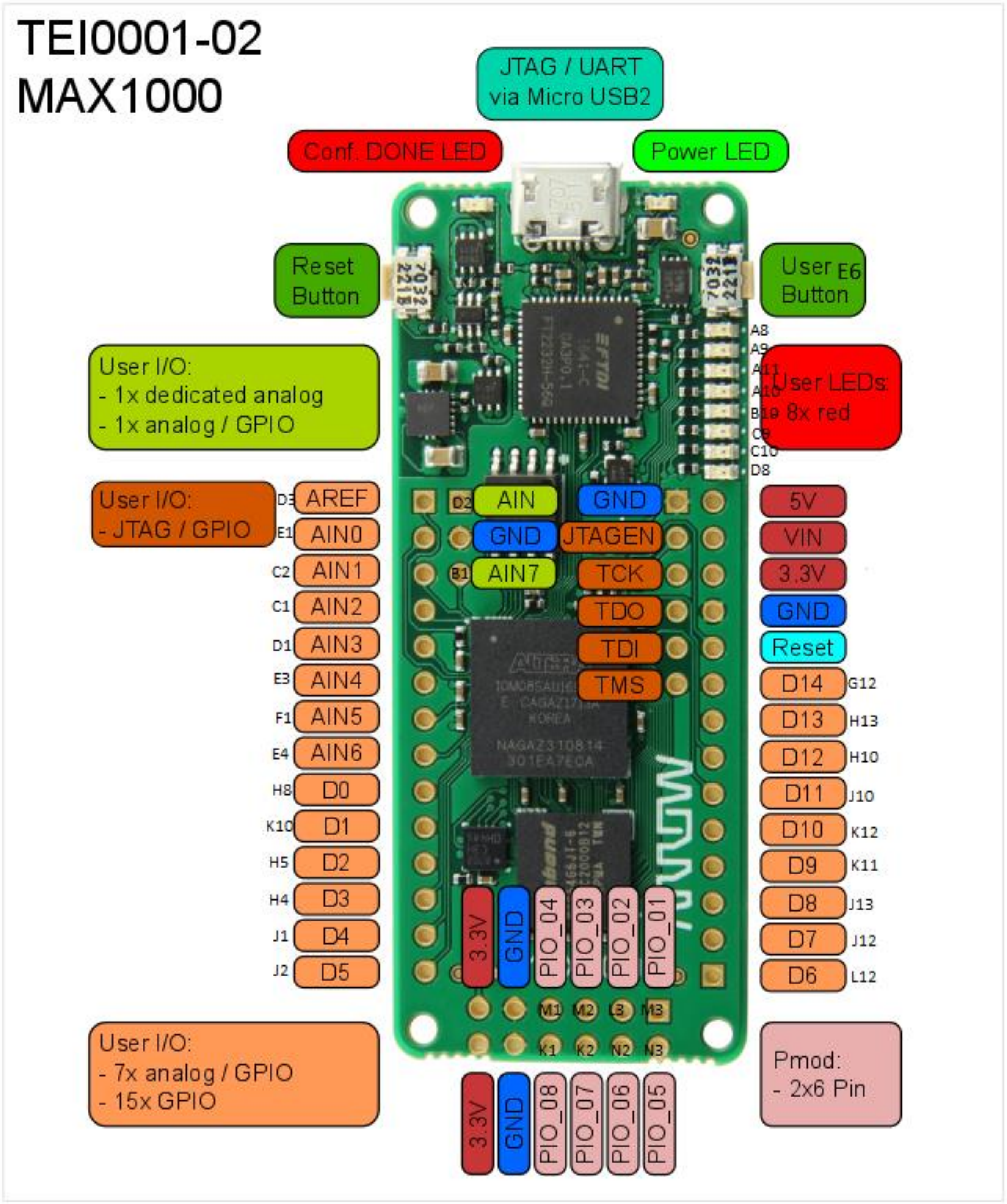

TEI0001-02
MAX1000
JTAG / UART via Micro USB2
Conf. DONE LED
Power LED
Reset Button
User E6 Button
User I/O: - 1x dedicated analog - 1x analog / GPIO
User LEDs: 8x red
A8
A9
A11
A10
B10
C9
C10
D8
User I/O: - JTAG / GPIO
D3 AREF
E1 AIN0
C2 AIN1
C1 AIN2
D1 AIN3
E3 AIN4
F1 AIN5
E4 AIN6
H8 D0
K10 D1
H5 D2
H4 D3
J1 D4
J2 D5
D2 AIN
GND
B1 AIN7
GND
JTAGEN
TCK
TDO
TDI
TMS
5V
VIN
3.3V
GND
Reset
D14 G12
D13 H13
D12 H10
D11 J10
D10 K12
D9 K11
D8 J13
D7 J12
D6 L12
3.3V
GND
PIO_04
PIO_03
PIO_02
PIO_01
M1
M2
L3
M3
K1
K2
N2
N3
3.3V
GND
PIO_08
PIO_07
PIO_06
PIO_05
User I/O: - 7x analog / GPIO - 15x GPIO
Pmod: - 2x6 Pin

Figure 9

Working with the Peripheral Board

Depending on what option you choose, you will need a dip switch and possibly a function generator. Shown below is the **Option 2** layout in reset state. Just flip the reset switch on then off to start the counter in either case. You should see the four leftmost LEDs light up in the correct counting order.

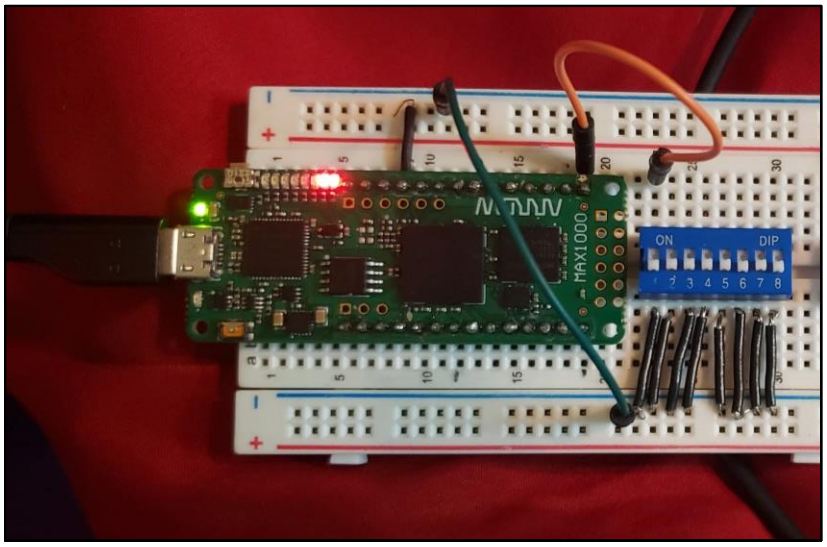


Figure 10

REFERENCES SECTION

[1] “verilog - How can I generate a 1 Hz clock from 50 MHz clock coming from an Altera board?,” *Electrical Engineering Stack Exchange*. [Online]. Available: https://electronics.stackexchange.com/questions/202876/how-can-i-generate-a-1-hz-clock-from-50-mhz-clock-coming-from-an-altera-board. [Accessed: 10-Feb-2020].