Lab 2 Stopwatch

Deadlines & Grading

- **Tutorial B & C:** 5 points, Monday-Wednesday, Sept 16-18, **individually**
- **Prelab 2A:** 5 points, Monday, Sept 23, Noon, **individually**
- **Lab 2A:** 10 points, Monday-Wednesday, Sept 23-25, **in groups of 2-3**
- **Prelab 2B:** 20 points, Monday, Sept 30, Noon, **individually**
- **Lab 2B:** 35 points, Monday-Wednesday, Sept 30- Oct 2, **in groups of 2-3**
- **Report**: 25 points, Thursday, Oct 10, 11 pm, **in groups of 2-3**

Section I: Overview

So far, the logic you have worked with in lab is known as *combinational logic*, where the outputs only depend on the current inputs. However, much of the power of digital design is in its ability to retain *state*, some form of memory. By having state, we can design circuits that can perform much more complex operations, since we can break down these operations into steps and create the circuit such that it remembers what step it is currently in for the operation. We call this *sequential logic*, since we're following a sequence of steps. One simple example of sequential logic is a counter, which just keeps tally of which step it is currently in. In this lab, we will be building a stopwatch, which is in effect nothing more than a counter that counts time.

Section II: Background

Digital clocks and stopwatches have become omnipresent. Timers and clocks play a critical role in a wide variety of areas, from watches to microwaves, and even to nuclear reactors, and are ubiquitous enough that we often take them for granted. A large number of athletic records would effectively not be trackable if it were not for stopwatches. Patented in 1869,[1] stopwatches (which at the time were analog devices with a dial) have undergone massive changes. Presently, the majority of stopwatches are digital, like the one shown in Figure 1, and are expected to come with once-luxurious features such as time and date displays.

Figure 1. Digital stopwatch.[2]

The sports world has benefitted immensely from better timing technology. The practice of timing races started as early as 1860.[1] The Stockholm Olympics in 1912 were the first popular event to use a hand-cranked camera to time athletes crossing the finishing line.[1] Even so, the precision of the results was not high enough. It

Figure 2. Seiko Photo Beam.[5]

was only in 1948, during the London Summer Games, that a photo finish camera was used to time the men's 100m finals, recording times with a precision of up to 0.01 seconds.[3]

Advanced stopwatches are no longer constrained to being started by the press of a button; upon detecting a trigger, they can start counting automatically. One such trigger is an audio signal,[4] which can be used to signal athletes at the same time. A more sophisticated design is the Seiko Photo Beam Unit (Figure 2), which records the precise instant an athlete crosses a certain point by observing when an infrared beam is broken.[5]

New timing technology has significantly altered training in sports. Athletic training can be improved by taking into account the robust nature of the human body, stemming from the adaptation of cells to strain and relaxation patterns.[6] For optimal performance, a training ratio (the ratio of load to recovery)[6] is calculated with the use of timers and knowledge of an athlete's body composition, in order to guide the training process. Set Starter introduced the world's smallest timer in 2012, shown fitting around a finger in Figure 3. They utilize touch-sensitive controls, along with features ranging from LEDs to alarms,[7] to enable athletes to focus more on their training and less on their monitoring equipment.

Figure 3. Set Starter.[8]

Section III: Tutorial B and C

It can be extremely time consuming to build circuits using integrated circuit (IC) chips and wires. As we saw, building and debugging a five gate circuit took quite some time. For the remaining labs this semester, we will use Quartus to help make building these circuits much simpler. Before you read any further, install Quartus II on your computer following *Tutorial B: Installing Quartus II*. *Don't underestimate the time this will take; do it as soon as possible.* You can then get familiar with how to use Quartus, and learn how to use *Verilog*. Verilog is a programming language, but unlike typical languages (such as C, Python, or Java), which have you list a series of steps to perform in a fixed order, Verilog instead describes logic gates and wires. Because of this, it is termed a *Hardware Description Language*. This means that **operations that you write in Verilog, no matter where they are in the file, are usually designed to occur in parallel as separate hardware modules**.

ASSIGNMENT 1

After installing Quartus II, read *Tutorial C: Introduction to Verilog and Quartus II* and complete **Section II** of the tutorial**.** This section will teach

you how to create a simple combinational circuit in Verilog, as well as how to compile and simulate your circuit in Quartus II.

1. Download the tutorial, as well as **lab2.zip**. Unzip the contents of the ZIP file to a folder named **lab2** on your computer. Read **NOTE 1** to make sure you have properly unzipped the file.
2. Use the **lab2** folder to complete up to and including Section II of the tutorial. You will only need to modify **comb.v** from the folder. **Do not delete this folder when you are finished – you will need it for the rest of the lab.**

NOTE 1 For our labs, you should not simply open a compressed folder (a ZIP file) without unzipping it. Quartus (and most software) cannot see inside the ZIP file. In Windows/Ubuntu, you can unzip by opening the ZIP file and clicking on *Extract all files*, which will allow you to create a new folder.

Section IV: Prelab 2A: Sequential Logic (T Flip Flop)

A T flip-flop is a circuit that takesthe stored value and *toggles* it (i.e., a high will become a low, and a low will become a high). As with all storage circuits, we have no idea what the initial value is inside the circuit when we turn the power on. As a result, we use a 1-bit *synchronous* input **RESET**, *which overrides all other inputs*. On the rising edge of the **CLK** input, if **RESET** is high, the value inside of the flip-flop should be set to 0. When **RESET** is low, we will then rely on the 1-bit input **T**. If **T** is low at the rising edge of **CLK**, then the 1-bit value stored in the flip-flop (**Q**) stays the same; if **T** is high, then we will toggle the value. We also need to make sure that we are outputting **Q** so we can observe the value inside the flip-flop.

ASSIGNMENT 2 Build a module called **tffp** that implements a one-bit T flip-flop.

1. Click on *File* → *Open Project…* and open the project **tffp**, which should be in the same **lab2** folder.
2. We have not provided the Verilog file **tffp.v**. You must build this from scratch. Click *File* → *New*. In the window that pops up, select *Verilog HDL File* under *Design File* and click OK.
3. Save this file in the **lab2** folder as **tffp.v** (don't forget **NOTE 5 from Tutorial C** – the filename is case sensitive).
4. Write up the **tffp** module. Use the input and output names we have mentioned above. Again, don't forget **NOTE 5** – assume that everything is case sensitive, and make sure you copy the names for your inputs and outputs exactly as written above.
5. Save the file and compile the design (*Processing* → *Start Compilation*).

6. Again, we have provided you with a test bench, this time called **tffp_test.v**. Read this file (similar to **Assignment 6 in Tutorial C)**, and then follow the steps in **Assignment 7 in Tutorial C** to verify that your module is right.

DELIVERABLE 1: prelab2a.zip

Submit the Verilog file **tffp.v** from the ZIP file given to you, and a text file **readme.txt**, all as a ZIP file named **prelab2a.zip**. The **readme.txt** file should include your name, your NetID, and any pertinent information for the graders. Your Verilog must compile and contain no inferred latches for full credit. Submit the file through CMS (http://cms.csuglab.cornell.edu).

NOTE 2 If you have created any sub-modules that you are using inside any of the required files, please remember to include the **.v** files for those as well. When in doubt, just submit all **.v** files inside your directory. You will get zero credit for missing files.

Section V: Lab 2A: T Flip-Flop Register

We highly recommend that you get a head start on this lab beforehand. If you can get through part B, then you have a higher chance of finishing before the end of your lab session.

A. Instantiating Sub-Modules

So far, everything that you've done has put the entire functionality of your project inside a single Verilog file. As you can imagine, this gets extremely messy for complex circuits. Fortunately, Verilog allows us to add *instances* of modules inside of other modules. Think of it this way: each module is like a black box. We know what it does, and know that it needs certain connections to be plugged into and out of it. As long as we properly connect these inputs and outputs, nothing stops us from putting that black box inside another black box.

Using a Previous Module Inside a New Module as a Sub-Circuit

When we are building more complex circuits, there are times that we want to include a smaller module that we have already built as part of the circuit. When we create a module, think of it as a cookie cutter – we can use it to create several identical copies. In order to distinguish between these different copies, or *instances*, we need to give them each a unique name.

Code Block VIII. Declaring module instances as sub-circuits.

```
/*  1 */  module big_module(a, B, c, x);
/*  2 */    input  [1:0] a;
/*  3 */    input        B;
/*  4 */    input        c;
/*  5 */
```

```
/*  6 */    output [1:0] x;
/*  7 */
/*  8 */    empty_module kitty (
/*  9 */      .A(a[0]),
/* 10 */      .B(B),
/* 11 */      .X(x[0])
/* 12 */    );
/* 13 */
/* 14 */    empty_module puppy (
/* 15 */      .A(a[1]),
/* 16 */      .B(c),
/* 17 */      .X(x[1])
/* 18 */    );
/* 19 */
/* 20 */  endmodule
```

Code Block VIII shows how we can hook up two instances of `empty_module` (from Code Block I) into a larger module, which we have called `big_module`. Our Verilog file needs to explicitly tell the compiler which wires inside our larger module hook up to the inputs and outputs of the smaller module instances that are inside. We use syntax similar to declaring a module. First, we enter the module name, then give this particular instance its unique name, and then list the wiring connections in parentheses. For example, on Lines 8 and 14, you'll see that we have used distinct names for each of the two `empty_module` instances.

Lines 9-11 show an example of how these connections are listed. The syntax of a connection is `.<wire_name_in_inner_module>(<wire_name_in_outer_module>)`. **It is extremely important that you start each connection with a period**; otherwise, the compiler will throw an error. Like inputs and outputs, these connections are separated by commas. The last connection in the list should **not** be followed by a comma. However, after the closing parenthesis, you should make sure to include a semicolon.

NOTE 2 When you create a circuit module, *it must be saved as its own Verilog file*, and the filename must be identical to the name of the module. For example, the module in Code Block VIII must be saved as **big_module.v** (remember – assume everything is case-sensitive).

B. Creating a 4-Bit Register of T Flip-Flops

In Section III, the T flip-flop we created can store a single bit. However, it can often be useful to combine multiple flip-flops into a *register* so we can store larger pieces of data at the same time. In a register, while each flip-flop maintains its own input and output wire, they each share a single input clock **CLK** and a single **RESET** input (as before, both are 1-bit inputs). This allows them to operate on different pieces of data at the same time. For the register that you will create, you will have a 4-bit input **IN** and a 4-bit output **OUT**, of which each bit will connect to a different **tffp** instance. (You should make sure to connect **IN[0]** and **OUT[0]** to the same instance, **IN[1]** and **OUT[1]** to the same instance, etc.) If done correctly, your register shouldn't need to consist of any internal logic other than instances.

ASSIGNMENT 3 Build a module called **treg4bit** that implements a four-bit T register.

1. Click on *File* → *Open Project…* and open the project **treg4bit**, which should be in the same **lab2** folder.
2. Again, we have not provided the Verilog file **treg4bit.v**. You must build this from scratch. Click *File* → *New*. In the window that pops up, select *Verilog HDL File* under *Design File* and click OK.
3. Save this file in the **lab2** folder as **treg4bit.v** (don't forget **NOTE 5** – the filename is case sensitive).
4. Write up the **treg4bit** module, using Code Block VIII as an example. Use the input and output names we have mentioned above. Again, don't forget **NOTE 5** – assume that everything is case sensitive, and make sure you copy the names for your inputs and outputs exactly as written above.
5. Save the file and compile the design (*Processing* → *Start Compilation*).
6. Again, we have provided you with a test bench, this time called **treg4bit_test.v**. Read the contents of the file, and then follow the steps in **Assignment 6 and 7** to verify that your module is right.

When you are debugging the circuit to determine what needs to be fixed, it is useful to examine the inside of the circuit. While ModelSim shows you the values of input and output wires by default, it does not show you internal variables or the connections to sub-circuit instances. However, we can in fact see what they are doing from inside ModelSim.

ASSIGNMENT 4 Run the simulation and examine internal module wires.

1. If you have closed ModelSim, click on *Tools* → *Run Simulation Tool* → *RTL Simulation*.
2. As previously, you will see the outputs and waveforms from before when the simulation completes. If the waves do not show up, make sure you have the Wave window shown (check *View* → *Wave*), and that there are no errors in the Transcript window.
3. Under the *View* menu, make sure that the following are checked: *Library*, *Objects*, *Transcript*, and *Wave*.
4. In the *Library* window, click on the *Sim* tab (circled in Figure 4). Here, you will see a list of modules, including **UUT**. Click on the plus sign next to UUT to show the list of modules inside your circuit.
5. If you click on one of the modules, the *Objects* window (Figure 4) will show a list of all input and output signals for that particular module, as well as all internal variables. If you want to debug by examining these signals, right-click the signal name, then click on *Add* → *To Wave* → *Selected Signals* (Or *Add Wave* in ModelSim 10.3d). The signal will now show up in the bottom of the *Wave* window, and will say *No Data*.

6. Now, we must re-run the simulation to see the internal signals. Click on *Simulate* → *Run* → *Restart…* (or *Simulate* → *Restart…)*. In the pop-up box, make sure everything is checked, and click *OK*. The *Wave* window will go blank.
7. Click on *Simulate* → *Run* → *Run -All*. Your internal signals will now show their proper values.
8. Repeat Steps 4 through 7 to examine more signals. When you are finished, close ModelSim. Again, if you make changes to your circuit, close ModelSim, re-compile your design, and repeat Assignment 6.

Another useful trick is changing the *radix* of multi-bit signals. Often, these buses will contain a number that may be easier to interpret in another base, such as decimal. ModelSim offers us a quick way of switching between bases from within the simulator. In the *Wave* window, simply right-click on the name of signal, go to *Radix*, and then choose which base you want to see the data in (keep in mind that single-bit connections can only be shown as a binary wave).

We will distribute the Altera DE0-CV boards during your lab sessions on Monday-Wednesday, Sept 17-19.

C. Preparing Your 4-bit Register for the DE0-CV FPGA Board

Before you download the design on to the FPGA board, you must first setup a few parameters for compiling a design.

For this lab, the input pins have already been assigned for you. They are mapped as follows:
- The **RESET** input uses button **FPGA_RESET**.
- The **CLK_SEL** input uses toggle switches **SW9** (MSB) through **SW7**. Set this to 000_2 to execute your logic at 1 Hz (allowing us to observe one change per second).
- Input **IN** uses toggle switch **SW3** (MSB) through **SW0 (LSB)**.

The circuit outputs are assigned as follows:
- Output **OUT** is shown on **LED3** through **LED0**. Each LED represents the value of a bit in the 4-bit register (LED ON = 1b'1, LED OFF = 1b'0).

If you want to check where the pin assignments are located:
1. Open the **treg4bit** project in Quartus II.
2. Go to *Assignments* → *Devices…* and select the Cyclone V family. Under *Available Devices*, select *Specific device selected*, and choose the *5CEBA4F23C7*. Click *OK*.
3. Check pin assignments by going to *Assignment* → *Assignment Editor*. This will show you a list of all pins. For this lab, the pins should have already been set for you.

Compile your design. This will create the file **treg4bit.sof**, which we use to program the FPGA.

D. Testing Your Design

First, make sure your system is properly setup to program the board:

1. Take the DE0-CV board and connect one end of the USB cable to the leftmost port (USB Blaster Port) and other end to your system.
2. Make sure that the red power switch on the left is pushed in to turn the board on. You will see the board light up and various things displayed on the 7-segment displays and the LEDs. This indicates the board is ready to be programed.
3. When you plug in the cable, if you are on Windows, it should tell you that the **Altera USB-Blaster** device was installed properly while if you are on Quartus-VM, you need to go the Devices -> USB Devices in the Virtual-Box menu and select Altera USB-Blaster device.
4. Make sure the **RUN/PROG** switch on the left of the board is set to the **RUN** position.

ASSIGNMENT 5 Program the DE0-CV board with your 4-bit register.

1. Inside Quartus, go to *Tools* → *Programmer*. Under *Hardware Setup*, select *USB-Blaster*. If you only see *No Hardware* you likely have an issue with the USB-Blaster driver.
2. Inside the programmer tool window, you will see your **treg4bit.sof** file listed. Make sure that the check box under *Program/Configure* is checked, and click *Start*. Once this completes, your design will now be downloaded to your FPGA.

You can control the toggle inputs (T) to each bit in the 4-bit register using the switches **SW3** through **SW0**. When you set to toggle switch to **ON**, the corresponding bit toggles which can be observed by the toggling of the corresponding LED. For example, if **SW2** is set to **ON**, **LED2** keeps toggling between **ON** and **OFF** state. During the lab check-off, you will demonstrate a complete 4-bit register on the DE0-CV FPGA board. The TAs will also ask you questions on the lab and ask you to demonstrate debugging using ModelSim and test benches.

Section VI: Prelab 2B: Counter

A. Building a Counter

A counter does more than a simple register. While a register just stores data that is fed into it, a counter must be able to *increment* the value stored inside it. The counter must also have a way of incrementing the value only when it is *enabled*. Lastly, there must be some way to *reset* this counter back to zero.

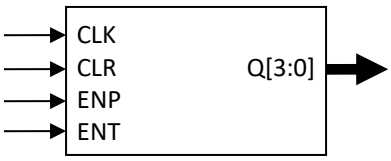

Figure 4. Counter block diagram.

Figure 4 shows you a block-level diagram for the counter, which you will create in the file **tcounter.v**. The one-bit input **CLK** is used to tell the counter when to increment. The one-bit *active low* input signal **CLR** resets the counter when it has the value 0. The counter also has two one-bit enable inputs, **ENP** and **ENT**. When *both* **ENP** and **ENT** are high, the counter will

increment whenever it sees a *rising edge* (a transition from low to high) on **CLK**. Whenever either **ENP** or **ENT** is low, the counter should *hold* the value currently stored within its register. When **CLR** is low *and* both **ENP** and **ENT** are high, the counter should *still* reset. The counter also has a four-bit output **Q**, which shows the current value stored inside the flip flops.

When you create the counter, you should use your **treg4bit** register to store the data. The **tcounter** module should have an *instance* of **treg4bit**, and you must build logic inside **tcounter** that determines what inputs to send to the **treg4bit** instance. Refer to *Tutorial C* on what instances are and how to create them. Remember: for a T flip-flop, the value stored changes when the input is high. Therefore, you must make sure that each bit not only increments when it is supposed to, but that it only increments when **ENP** and **ENT** are high. Also keep in mind that each bit of the **treg4bit** module will have different logic for when it should increment.

ASSIGNMENT 6 Build a module called **tcounter** that implements a counter.

7. Open the project file **lab2.qpf** (in the **lab2** folder from **Assignment 1**) inside Quartus.
8. Click *File* → *New*. In the window that pops up, select *Verilog HDL File* under *Design File* and click OK.
9. Save this file as **tcounter.v**, making sure that it is also inside the **lab2** folder.
10. Write up the counter module, using the code blocks in *Tutorial C* as examples. Remember that your input and output names should look identical to the names we described above, and are **case sensitive**. Feel free to draw out your logic on scrap paper before you start writing Verilog.
11. Save the file, and then compile your project. Correct any syntax errors.

B. Top-Level Assembly & ModelSim Testing

Once you have completed the **tcounter** module, you must now hook it up so we can test it out. Inside the **lab2** folder, you will find a file called **lab2.v**. This is what we will be running on the DE0-CV for this lab. Therefore, if you want to test your **tcounter** module, you must create an instance of it inside **lab2.v** and connect it to the inputs and outputs inside there.

For this part, connect your **tcounter** instance as follows:
- The **lab2** input **RESET** should be *inverted*, and then connected to **CLR** on **tcounter**.
- The **lab2** input **ENABLE** should be connected to **ENP** on **tcounter**.
- Connect **ENT** on **tcounter** to the hard-wired value 1'b1 for now.
- The **lab2** input **CLK** should be connected to **CLK** on **tcounter**.
- The **lab2** output **CENTISEC** should be connected to **Q** on **tcounter**.
- You can leave all other inputs and outputs unconnected for now.

ASSIGNMENT 7 Add your **tcounter** module to the **lab2** module.

1. Click *File* → *Open File*. Open **lab2.v**.
2. Create an instance of **tcounter** inside **lab2.v**, and wire it up as described above.
3. Save the file, compile your project, and test the circuit using the test bench **lab2_test.v**.
4. Verify that your **tcounter** module is working as expected. The testbench provided includes test cases for 2 parts; part A is counter module (Section V) and part B is full stopwatch module (Section VI). For prelab 2B, you only need to verify that the cases below the line 'CENTISEC TEST CASES (can be ignored for Part B)' are all correct.

DELIVERABLE 2: prelab2b.zip
Submit the Verilog file you created "**tcounter.v**", and a text file **readme.txt**, all as a ZIP file named **prelab2b.zip**. The **readme.txt** file should include your name, your NetID, and any pertinent information for the graders. Your Verilog must compile and contain no inferred latches for full credit. Submit the file through CMS (http://cms.csuglab.cornell.edu).

C. Stopwatch Digits

The counter that you created will allow you to count from 0 to 15, since it contains four bits. After it reaches 15, it will increment again, but since it cannot carry the one over (since there is no bit to the left), the counter will reset to 0, and will start counting from there again.

While all this is nice, it is of little use for a real stopwatch. After all, a stopwatch has digits that only count from 0 through 9. In fact, not all digits even go all the way up to 9 (since there are only 60 seconds in a minute). We are going to build a stopwatch that can count minutes (**M**), seconds (**S**), and hundredths of seconds (**h**), which will be displayed in the five-digit format $\mathbf{M}:\mathbf{S_1S_0}:\mathbf{h_1h_0}$. To do this, we must understand how to adapt our **tcounter** module from Section IV so that it can properly represent each of the digits. For this part, you must figure out the range for each digit, as well as what conditions must be met to *(a)* increment and *(b)* reset each digit back to 0.

ASSIGNMENT 8 For each digit of the stopwatch, write down the range that each digit can take.

ASSIGNMENT 9 For each of the digits, write down when the digit should increment. (Hint: you can use the values of other digits.) For example, if a two-digit counter is going from 09 to 10, the upper digit should know to increment when the lower digit is 9, not 0, since increments occur in lockstep at the rising

clock edge. Inside an `always` block, keep in mind that all flip-flop values on the right hand side of the assignment operator (<=) are the values *just before the rising edge*.

ASSIGNMENT 10 Write down what the reset conditions are for each of the digits.

DELIVERABLE 3: prelab2b.pdf
Submit answers for **Assignments 8-10** in PDF format as a part of Prelab 2B. This file must contain the range, increment conditions, and reset conditions for each digit, with the digit clearly labeled. Make sure that the PDF includes the name and NetID of all group members. Submit the file through CMS (http://cms.csuglab.cornell.edu).

D. Preparing Your Counter for the DE0-CV FPGA Board

Before you download the design on to the FPGA board, you must first setup a few parameters for compiling a design.

For this lab, the input pins have already been assigned for you. They are mapped as follows:
- The **RESET** input uses button **FPGA_RESET**.
- The **CLK_SEL** input uses toggle switches **SW9** (MSB) through **SW7**. Set this to 000_2 to execute your logic at 1 Hz (allowing us to observe one change per second).
- Input **ENABLE** (which is connected to **ENP** on the counter) uses toggle switch **SW0**.

The circuit outputs are assigned as follows:
- Output **CENTISEC** is shown on **HEX0**.

If you want to check where the pin assignments are located:
4. Open the **lab2** project in Quartus II.
5. Go to *Assignments* → *Devices*… and select the Cyclone V family. Under *Available Devices*, select *Specific device selected*, and choose the *5CEBA4F23C7*. Click *OK*.
6. Check pin assignments by going to *Assignment* → *Assignment Editor*. This will show you a list of all pins. For this lab, the pins should have already been set for you.

Compile your design. This will create the file **lab2.sof**, which we use to program the FPGA. Use the instructions in **Section IV-D** of this document to program your DE0-CV board. You should see the rightmost 7-segment display count upwards in hexadecimal.

Section VII: Lab 2B: Complete Stopwatch

You will use the code developed in Section V to build a complete stopwatch. The work that you did in **Assignments 8-10** should allow you to determine what logic you need when connecting

multiple **tcounter** instances together. During the lab check-off, you will demonstrate a complete stopwatch on the DE0-CV FPGA board. The TAs will also ask you questions on the lab and ask you to demonstrate debugging using ModelSim and test benches.

A. Top-Level Assembly & Testing

Once you have completed the **tcounter** module, you must now hook it up so you can test it out. Inside the **lab2** folder, you will find a file called **lab2.v**. This is what we will be running on the DE0-CV for this lab. Therefore, if you want to test your **tcounter** module, you must create an instance of it inside **lab2.v** and connect it to the inputs and outputs inside there.

For this part, connect your **tcounter** instance as follows:

- The **lab2** input **RESET** should be connected, along with the reset logic for each **tcounter** digit (you can use Boolean logic gates to connect these together), to the **CLR** input of each **tcounter** instance. Don't forget that **CLR** is *active low*, so you may need to invert your final input.
- The **lab2** input **ENABLE** should be connected to **ENP** on *all* **tcounter** instances.
- Use the **ENT** input on each **tcounter** instance to control when incrementing should occur.
- The **lab2** input **CLK** should be connected to **CLK** on *all* **tcounter** instances.
- The **lab2** output **CENTISEC** should be connected to **Q** on the **tcounter** instance for digit $\mathbf{h_0}$.
- The **lab2** output **DECISEC** should be connected to **Q** on the **tcounter** instance for digit $\mathbf{h_1}$.
- The **lab2** output **SEC** should be connected to **Q** on the **tcounter** instance for digit $\mathbf{S_0}$.
- The **lab2** output **TENSEC** should be connected to **Q** on the **tcounter** instance for digit $\mathbf{S_1}$.
- The **lab2** output **MIN** should be connected to **Q** on the **tcounter** instance for digit **M**.

ASSIGNMENT 11 Modify the **lab2.v** file to connect several **tcounter** instances together, using your work from **Assignments 5-7** to determine what logic needs to be added. Remember to compile and test your design using ModelSim. Inside **lab2_test.v**, you will need to delete the two lines in the test bench that say "FOR PART B, DELETE THIS LINE ONLY" before starting your testing.

B. Preparing Your Stopwatch for the DE0-CV Board

We must first set up the following parameters before compiling a design. We will do so in a file called **lab2_top.v**:

For this lab, the input pins have already been assigned for you. They are mapped as follows:

- The **RESET** input uses button **FPGA_RESET (**signal name **RESET_N)**.
- The **CLK_SEL** input uses toggle switches **SW9** (MSB) through **SW7**. For this part of the lab, set **CLK_SEL** to 010_2 to execute your logic at 100 Hz (allowing the stopwatch to operate at its intended speed).
- Input **ENABLE** (which is connected to **ENP** on the counter) uses toggle switch **SW0**.

Circuit outputs are assigned as follows:

- Output **CENTISEC** is shown on **HEX0**.

- Output **DECISEC** is shown on **HEX1**.
- Output **SEC** is shown on **HEX2**.
- Output **TENSEC** is shown on **HEX3**.
- Output **MIN** is shown on **HEX4**.

Finally, use the instructions in **Section IV-D** of this document to program your DE0-CV board.

Section VIII: Lab Report

DELIVERABLE 4: lab2report.pdf
You will submit a single report for your group. **Each partner must contribute to the writing of the report.** Please refer to the Lab Guidelines for general information on what to include. The report must be a PDF file, named **lab2report.pdf**, and should be submitted through CMS (http://cms.csuglab.cornell.edu).

For this particular lab report, you should include the following sections:

- **Design and Implementation:**
  - General overview of your counter and stopwatch designs including their inputs and outputs, the function they perform.
  - Provide details of how the counter and stopwatch were implemented. For the counter, explain how the counter was implemented using registers. For the stopwatch, you need to include how multiple counters were used.
  - A block diagram showing connections between different **tcounter** instances (and its inputs/outputs) in the stopwatch would be helpful. You can also show the connections of various modules used inside the **tcounter** module.
  - Discuss whether you could implement the counter without using sequential logic.

- **Testing:**
  - Describe how your circuit was tested and debugged. Explain the test cases in the "lab2_test.v" file.

- **Work Distribution:**
  - Compare your prelab design with your partner's. If they differ significantly, please explain why, as well as the reasons you selected one over the other for implementation.

References

[1] Ross, S. (2008). *Higher, Further, Faster: Is Technology Improving Sport?* (p. 271). Chichester, England: Wiley/Dana Centre.

[2] *Stoppuhr digital* (2008, February 2). Wikimedia Commons. Retrieved February 13, 2014, from https://commons.wikimedia.org/wiki/File:Stoppuhr_digital.jpg

[3] *A History of Technology in Sports* (2012, August 10). +Republic. Retrieved October 19, 2012, from http://plusrepublic.com/a-history-of-technology-in-sports/

[4] Dabnichki, P. (2008). "Motion Analysis in Water Sports." *Computers in Sport* (p. 235). Southampton: WIT Press.

[5] *Harmony with SEIKO – Athletics: Track*. Seiko Holdings Corp. Retrieved October 19, 2012, from http://www.seiko.co.jp/en/harmony/sports_timing/timing_system/track.php

[6] *Training Theory*. International Association of Athletics Federations. Retrieved October 19, 2012, from http://www.coachr.org/training_theory.htm

[7] *Tiny Timer Helps Athletes Monitor Rest Between Exercise Sets to Improve Fitness* (2012, August 31). Yahoo! News. Retrieved October 19, 2012, from http://news.yahoo.com/tiny-timer-helps-athletes-monitor-rest-between-exercise-020310056.html

[8] *Set Starter Interval Training Timer Fits on Your Thumb* (2012, July 10). Set Starter. LLC. Retrieved on February 14, 2014, from http://www.prweb.com/releases/setstarter/intervaltraining/prweb9675425.htm