ELE32EDA Project

Clock/Stop Watch/Timer

Jarrod Crivelli, 02571155
Robert Ross, 02578565
Tony de Souza-Daw, 02575383
Wade Tregaskis, 02557793

1st of November 2004

Contents

Abstract

The fundamental functionality of this project is to simulate a stopwatch, timer and clock. There is global reset that will reset the clock, stopwatch and timer to default values. The device will be controlled using two push-buttons, to perform operations such as change mode, start/stop the stopwatch, set the time, etc.

Stopwatch

This aim of this mode is to mimic the operation of a typical consumer stopwatch. The time will be displayed using the two 7-segment displays on the main board, and the six 7-segment displays on the expansion board. The stopwatch has a start/stop button which, intuitively enough, starts and stops the counting. It also has a reset button, which sets the time back to 0.

The stopwatch will continue to count even if the system is in another mode.

Timer

The aim of this mode is similar to the stopwatch, with two key differences – it counts down instead of up, starting from a user-defined time, and it has an alarm which alerts the user when the timer has expired. The time remaining is shown using all eight 7-segment displays, as with the stopwatch. One of the push-buttons toggles between normal and setting mode. In the latter mode, the two push-buttons may be used to cycle between the four fields (hours, minutes, seconds & centiseconds) and modify them incrementally. In normal mode, the timer can be enabled or disabled via one of the push buttons. When enabled it will count down until it reaches 0, at which point it flashes all eight 7-segment displays to indicate that the alarm has been triggered.

When the alarm is de-activated (via a push-button) it resets to the value originally set by the user, to facilitate easy repeated timings.

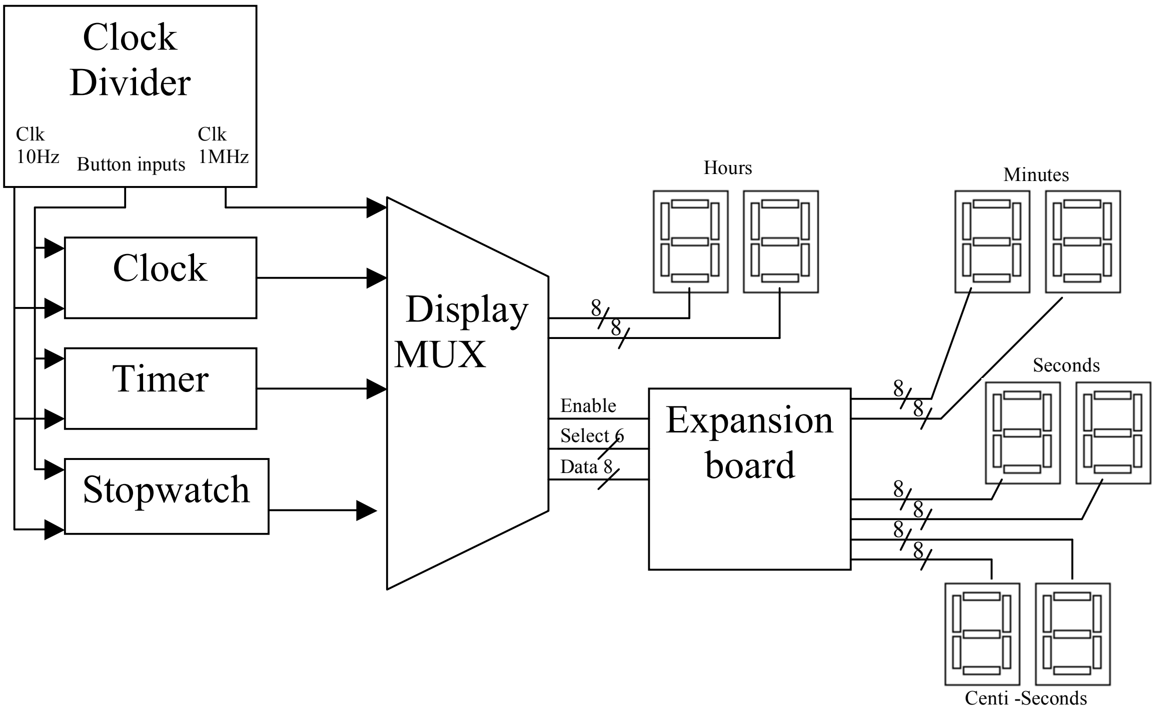
Clock

The aim of this mode is to provide an accurate, settable 12-hour clock. It displays the current time on the eight 7-segment displays, as with the stopwatch and the timer. The time can be set using the push-buttons, in the same way as for the timer.

The clock will continue to keep time even if the system is in another mode.

Specification & Design

Top-Level Functional Breakdown



Functional Description

The design of this project with all the required functionality was developed using only two push-button switches. The exact operation these two push-buttons perform is specified in the four tables below. The left-most push-button on the board is denoted as SW1, the other as SW2.

Table 1 - Push-Button Functionality Overview

Switch Name	Function
SW1	Mode button – cycles through the three modes; clock, stopwatch & timer.
SW2	Action button – Starts/stops the stopwatch & timer, increments selected field in 'setting' mode, etc.

Table 2 - Push-Button Functionality in Stopwatch Mode

Switch Name	Function
SW1	On Press: The mode switches to the next mode in sequence (Clock). On Hold (for more than 2 seconds) and in stop mode: Reset to 00:00:00:00.
SW2	On Press: The stopwatch toggles between running and paused.

Table 3 - Push-Button Functionality in Timer Mode

Switch Name	Function
SW1	On Press when in normal mode: The mode switches to the next mode in sequence (Stopwatch). On Press when in set mode: Cycles the current selection through the four fields (hours, minutes, seconds & centiseconds).

	On Hold (for more than 2 seconds) when in set mode: Go into normal mode.
SW2	On Press when in normal mode: Toggles operation of the timer, from running to paused or vice versa. On Press when in set mode: Increments the selected field by 1. On Hold (for more than 2 seconds) when in normal mode: Resets the timer to 00:00:00:00 (without triggering an alarm). On Hold (for more than 2 seconds) when in set mode: Increments the selected field at a rate of 3 units per second (while the button is held down).

Table 4 - Push-Button Functionality in Clock Mode

Switch Name	Function
SW1	On Press when in normal mode: The mode switches to the next mode in sequence (Timer). On Press when in set mode: Cycles the current selection through the four fields (hours, minutes, seconds & centiseconds). On Hold (for more than 2 seconds) when in normal mode: Go into set mode. On Hold (for more than 2 seconds) when in set mode: Go into normal mode.
SW2	On Press when in set mode: Increments the selected field by 1 unit. On Hold (for more than 2 seconds) when in set mode: Increments the selected field at a rate of 3 units per second (while the button is held down).

Special note: when SW1 & SW2 are both held down for 2 seconds or longer, everything is reset. This occurs regardless.

In addition to these normal functions, three DIP switches are used to aid debugging, by adjusting the global clock multiplier. These function as a 3-bit binary integer which selects the order of the multiplier, from normal speed (010) to 10^2 times slower (000), to 10^5 times faster (111).

This functionality is of course only for debugging – as a hypothetical final product the clock multiplier would be permanently fixed to normal speed, and the DIP switches not provided.

Component Descriptions

Clock

selectionToggle
clock

The basis of the clock is the clockField component. The clock contains four instances of this module, for centiseconds, seconds, minutes & hours. The clock's primary function is simply to connect these clockField's together with the relevant glue logic to provide the specific functionality required for the clock. It's two primary functions are:

1. To maintain a time value (in hours, minutes, seconds & centiseconds) that automatically increments in real-time.
2. To provide a mechanism for changing the current time value.

It's interface is defined largely by the controller component which drives it. This controller provides various digital signals indicating user inputs, in addition to handling the mode selection and so forth. The clock provides various outputs to the controller, which are then multiplexed or otherwise handled appropriately – the details of which are beyond the clock's scope. It simply counts time, and manages relevant user input. It's interface is as shown below:

```
clk : IN std_logic; -- The clock who's rising edge triggers the Clock
selectionToggle : IN std_logic; -- If '1' on a rising clock edge (and 'active' is high, and 'mode' is
high), changes the selected field
inc : IN std_logic; -- If '1' on a rising clock edge (and 'active' is high, and 'mode' is high)
indicates the selected field should be incremented
reset : IN std_logic; -- If '1' (and 'active' is high) resets the Clock
mode : IN std_logic; -- If '1' the Clock operates in set mode, otherwise it operates in "normal" mode
active : IN std_logic; -- If '1' indicates the Clock should respond to it's inputs
('selectionToggle', 'inc' & 'reset')

-- All the following are 7-segment display outputs.  They are all wired directly to the outputs for
each field.

csecLSD : BUFFER integer range 0 to 9; -- msecLSD
csecMSD : BUFFER integer range 0 to 9; -- msecMSD
secLSD : BUFFER integer range 0 to 9;
secMSD : BUFFER integer range 0 to 9;
minLSD : BUFFER integer range 0 to 9;
minMSD : BUFFER integer range 0 to 9;
hourLSD : BUFFER integer range 0 to 9;
hourMSD : BUFFER integer range 0 to 9;

enable : OUT std_logic_vector(7 downto 0); -- Represents the 8 7-segment displays, where a '1'
indicates the display should be active, a '0' indicates it should be off
flash : BUFFER std_logic_vector (7 DOWNTO 0)); -- As above, but '1' indicates the display should
flash on and off, '0' indicates "normal" behaviour
```

The clock operates synchronously, based on the "clk" input. It expects this clock to be exactly 100Hz. Together with the synchronous reset signal these control the basic operation of the clock. In addition, the user control over the clock is implemented as the "selectionToggle", "inc" and "mode" inputs. The first two indicate when the user has indicated they wish to change the selected field or increment the selected field's value (respectively). The "mode" input indicates which local mode the clock should be operating in – whether 'normal' or 'setting'.

In addition to these basic inputs, an "active" input is provided to help determined when these inputs are valid. Since the clock is not the only function of the device, it should generally only pay attention to user inputs if it is active. Consequently, all users inputs are logically anded with the "active" input prior to usage.

The outputs are of course the 8 digits of the clock itself, as well as indicators as to whether those digits should be displayed and, if so, whether they should flash – "enable" and "flash" respectively. Enable is always tied active, since the clock always uses all 8 digits, although flash is dependant on whether the local mode is 'setting' and which field is selected.

The implementation itself is relatively simple. The four fields are defined as clockField instances, and various simple combinational logic defined to join them together (tying the carry of the centiseconds to the seconds, for example). In addition to this asynchronous combinational logic, there is one process which handles non-exclusive logic – that is, logic which cannot easily be handled asynchronously and combinatorial, in this case because not all states are valid, and must be detected as such. For example, which field is currently selected is determined by the presence of an appropriate ‘1’ in a 4-bit vector. This vector is validated each clock cycle (if in ‘setting’ mode) and the “flash” output derived from it.

In retrospect this process could have been converted to combinational logic without too much difficulty, but it would require fairly verbose “error” checking of the aforementioned 4-bit vector. Additionally, the process form is more intuitive – at least to those with a computer science background, such as myself.

clockField

clk
reset
inc
direction
minLSD
minMSD
maxLSD
maxMSD
lsd
msd
carry
clockField

The clockField component represents a generic two-digit counting number, such as you would find for the hours, minutes or seconds on a digital clock. It has three primary functions:

1. To store a time value.
2. To increment or decrement that value on instruction, with appropriate roll-over between minimum and maximum values (as taken from inputs), including an overflow output.
3. To reset the time value as appropriate, when instructed.

To these ends, it has the interface as defined below. It’s implementation is relatively trivial, being a simple process that if…else’s it’s way through the various conditions as set by the inputs. It has no state beyond the current value of it’s two digits.

```
clk : IN std_logic; -- System clock; clockField is triggered by rising edges on this clock
reset : IN std_logic; -- If '1' resets the values to either 'minLSD' and 'minMSD' (if 'direction' is
high) or 'maxLSD' and 'maxMSD' (if 'direction' is low)
inc : IN std_logic; -- If '1' on a rising clock edge, increments or decrements the current value (as
dependant on 'direction')
direction : IN std_logic; -- '1' == up, '0' == down

-- These min & max values define the two numbers between which the field is permitted to operate.  They
apply only in pairs, i.e. if minLSD = 1 and minMSD = 2, the field will never go below '21'.  This
doesn't stop the LSD counting about 1; it will always count from 0 to 9.

minLSD : IN integer RANGE 0 to 9; -- The minimum value of the LSD (only applies if the MSD is equal to
'minMSD'
minMSD : IN integer RANGE 0 to 9; -- The minimum value of the MSD
maxLSD : IN integer RANGE 0 to 9; -- The maximum value of the LSD (only applies if the MSD is equal to
'maxMSD'
maxMSD : IN integer RANGE 0 to 9; -- The maximum value of the MSD

lsd : BUFFER integer range 0 to 9; -- The 7-segment display on which to put the LSD
msd : BUFFER integer range 0 to 9; -- The 7-segment display on which to put the MSD
carry : BUFFER std_logic); -- Will be high for one clock cycle if the field overflows
```

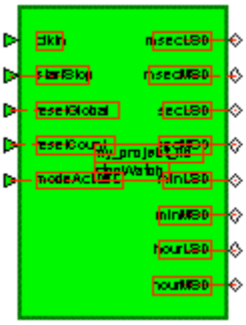
Since the clockField operates synchronously it must be clocked, thus the “clk” input. Additionally, it must include a reset facility (for obvious reasons), thus the synchronous “reset” input. The “inc” input signals when the field should increment or decrement, as determined by the input “direction”.

Since the clockField is designed to be as versatile as possible (within reason), it does not make any assumptions about what it’s minimum or maximum values are. These are provided as inputs (as seen in the above interface listing). When either of these limits is exceeded, the field takes on the value of the opposite limit – i.e. if minMSD & minLSD = 01, maxMSD & maxLSD = 12, and the field is incremented while containing the value 11, it will roll over to 01.

The roll-over behaviour can be essentially forced by asserting the reset signal – depending on the direction input, this sets the current value to either the minimum or the maximum value. It is also guaranteed to reset any other internal state machinery (although the current implementation does not have any).

The final elements are the all-important outputs. In addition to the two BCD digits, there is a carry signal which is asserted for one clock cycle each time the field overflows. This is of course vital for chaining multiple fields together.

Stop Watch



The stopwatch module operates in a manner similar to the clock, with added functionality of allowing the counting to be stopped and started at will, and a reset which allows the count to be set back to 00:00:00:00 at any time.

The start/stop button has been implemented to be fast reacting (with a small amount of de-bounce before the signal is activated and then maintaining de-bounce for a further short time). This means that timing will be more accurate and will give the user greater control over the operations of the stopwatch.

Fundamentally the stopwatch operates on a nested counter in turn incrementing successively each of the next significant digits as one digit rolls over. The MSD for seconds, minutes don’t technically have to go up to 9, and could be bounded at six, but to ensure compatibility with our display driver functions, which by default take a integer with size 0 to 9.

The stopwatch also has two different reset states. If the global reset is applied (both buttons held down for more than 2 seconds) independent of the current mode the stopwatch is reset. If the mode button is held down for more than two seconds (when in stopwatch mode) and when the count is stopped a local reset will apply, resetting the stopwatch count.

The entity for the stopwatch is shown below, showing the inputs and outputs to the stopwatch component.

```
        clkIn: in std_logic;            --100Hz clk signal
        startStop: in std_logic;        --toggles start/stop
        resetGlobal: in std_logic;      --global reset for board
        resetCount: in std_logic;       --resets the count when in stop mode to 00:00:00:00
        modeActiveL: in std_logic;      --if 1 system is in stopwatch mode
--integer outputs to display driver functions
        msecLSD: buffer integer range 0 to 9;
        msecMSD: buffer integer range 0 to 9;
        secLSD: buffer integer range 0 to 9;
        secMSD: buffer integer range 0 to 9;
        minLSD: buffer integer range 0 to 9;
        minMSD: buffer integer range 0 to 9;
        hourLSD: buffer integer range 0 to 9;
        hourMSD: buffer integer range 0 to 9);
```

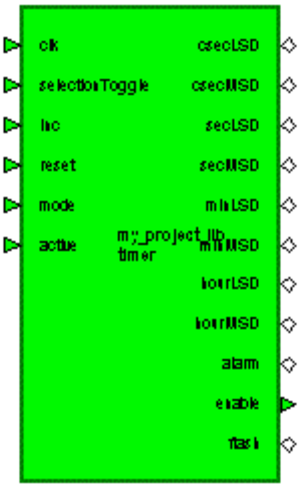
The clkIn input is a 100Hz clock input which synchronises all counting within the stopwatch function. The startStop input toggles the stopwatch from counting mode and stopped mode. The resetGlobal and resetCount inputs reset the stopwatch as described above. The mode active input indicates to the stopwatch if it is currently the selected mode. If it is the selected mode this input will supply 1.

The outputs display the most and least significant digits for all the required outputs to be displayed.

The main process within the component, ‘countUp’ handles the counting and the reset of the stopwatch. This process is synchronous with the 100Hz clkIn signal and continually counts on each clock edge when in counting mode and is not being reset.

The startStopSettings process handles changing the counting mode from start to stop state, handling startup and reset conditions.

Timer



The timer operates very similarly to the clock, with only a few notable exceptions. The most obvious of course is that it counts down instead of up. It also does so only when locally "active" – that is, the user can pause & resume the counting at will.

It also has a global alarm state, which causes focus to shift to the timer regardless of what else the user was doing – it is reasoned that the alarm would be significantly reduced in reliability and practicality if it only functioned while the timer were selected.

Finally, it stores a "set" value in addition to the current time. This enables to it to quickly reset to the previously-set value whenever the alarm is reset.

The interface for the timer component is as shown below:

```
clk : IN std_logic; -- The clock who's rising edge triggers the Clock
selectionToggle  : IN std_logic; -- If '1' on a rising clock edge (and 'active' is high, and 'mode' is
high), changes the selected field
inc : IN std_logic; -- If '1' on a rising clock edge (and 'active' is high, and 'mode' is high)
indicates the selected field should be incremented
reset  : IN std_logic; -- If '1' (and 'active' is high) resets the Clock
mode : IN std_logic; -- If '1' the Clock operates in set mode, otherwise it operates in "normal" mode
active : IN std_logic; -- If '1' indicates the Clock should respond to it's inputs
('selectionToggle', 'inc' & 'reset')

-- All the following are 7-segment display outputs.  They are all wired directly to the outputs for
each field.

csecLSD : BUFFER integer range 0 to 9; -- msecLSD
csecMSD : BUFFER integer range 0 to 9; -- msecMSD
secLSD : BUFFER integer range 0 to 9;
secMSD : BUFFER integer range 0 to 9;
minLSD : BUFFER integer range 0 to 9;
minMSD : BUFFER integer range 0 to 9;
hourLSD : BUFFER integer range 0 to 9;
hourMSD : BUFFER integer range 0 to 9;

alarm  : BUFFER std_logic; -- If '1', indicates the alarm is going off

enable : OUT std_logic_vector(7 downto 0); -- Represents the 8 7-segment displays, where a '1'
indicates the display should be active, a '0' indicates it should be off
flash : BUFFER std_logic_vector (7 DOWNTO 0)); -- As above, but '1' indicates the display should
flash on and off, '0' indicates "normal" behaviour
```

This is identical to that for the clock (save the "alarm" output, covered below), and as such you are referred to the interface part of the clock sub-section for more information.

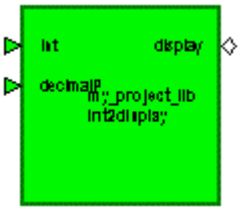
The "alarm" output is asserted whenever the timer expires. It is used to provide an indication of the alarm state even when the timer is not active – ideally this would be through a dedicated LED and speaker, but on the development board's used it instead changes the mode automatically to the timer.

As with the clock, a good portion of the implementation is in defining the four clockField instances, and the combinational logic that glues them together. The additional requirements of the timer, however, means two processes are used instead of the one for the clock.

The first process, "minResetter", is relatively trivial – it responds to an internal "resetTime" signal which is asserted whenever the previously set time should be restored to the current value – i.e. when the user turns the alarm off. It's job is slightly above trivial in that it does this by controlling the minimum & maximum value inputs to each of the four clockField instances. Since there is no mechanism for setting the field values explicitly, it sets them indirectly by setting the appropriate minimum/maximum values. When a reset signal is then sent to the clockField's, they will adopt the desired value.

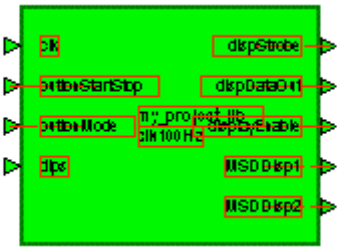
The second process, "timerer", is the timer's equivalent of the clock's "clocker" process. In addition to handling the selected field, however, it also saves the current field values (if in 'setting' mode), and also manages the global alarm output (i.e. determining when to assert the alarm, and when to

Int2Display



The Int2Display Component is a very simple component which when supplied with a integer (between 0 and 10) and a decimal point signal, outputs a 8 bit std logic vector containing the output mapping to the 7-segment displays. If a input value of 10 is supplied then the display will be blank – this blanking was used as it seems to provide better integration and less critical execution paths than switching the individual displays on and off.

clk100Hz

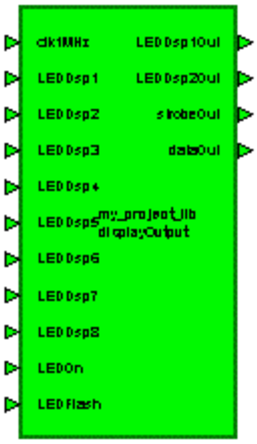


The clk100Hz module works basically as an interfacing module between all the other main modules, directly calling the stopwatch, timer, clock and display components.

The inputs for this component are clk (the 25.125MHz clock), buttonStartStop (the start/stop push button), buttonMode (the mode push button, dips (the dips switches to set test modes).

The outputs from this component are dispStrobe (set to update a register on the expansion board), dispDataOut (data lines for the display lines), displayEnable (enable digit on board), MSDDisp1 and MSDDisp2 (the signals mapped to the two hours digits).

displayOutput



The displayOutput component maps all the digit outputs onto the output displays. It takes in each of the digits including flashing options and on/off options to adjust the displays.

The inputs for the component are clk1MHz (a 1MHz clock to update the display), LEDDsp1 – LEDDsp8 (the integers for each of the digits to be output), LEDOn (a standard logic vector specifying which displays should be switched on), LEDFlash (a standard logic vector specifying which displays should be flashing and which should be static).

The outputs for the component are LEDDsp1Out and LEDDsp2Out (the values being output to the hours digits), stobeOut (the strobe signals to update individual digits), dataOut (the actual data being output which is read in according to which strobe signal is high).

Testing

Extensive testing is a requirement for any engineering project. Testing aims not only to uncover flaws in the tested system, but also conversely to provide assurances of quality, reliability and stability. To emphasise the importance of testing, two of the four group members were devoted entirely to it, from the start of the project to the end. Their systematic and rigorous approach has ensured from the very start that each process, entity and module is operating correctly, and can then be integrated with the rest of the system.

Test Design and Methodology

The testing methodology adopted involved standard module, integration and regression testing. Both simulation and hardware test cases were developed and executed. Revisions were made where necessary, and testing repeated from scratch.

The software simulations were produced with ModelSim and MaxPlus. The hardware tests were performed on ALTERA UP2 Education Board and Altera-Display Board, by either Jarrod Crivelli or Tony de Souza-Daw.

Note that the two developers, Robert Ross and Wade Tregaskis, invariably also performed various tests as they worked. These have been explicitly excluded from this report, on the principle that the very worst people to test any product are those who made it – clearly they would only be looking for expected functionality, and would most likely not test the system in the way an end user would.

Test plan

All individual modules (component) that could be tested independently were examined first, against specially design white and black box test cases. For the discrete components such as the int2display, stopwatch, clockfield and clock test drivers were developed and simulated. Modules (containing integrated-components) were fundamentally their own test bench, simulating with Max-Plus changing the signals default values and providing a clock. This was because there were difficulties encountered with ModuleSim simulating modules with embedded components. The clock divider and the decimal clock mainframe were added and re-tested in conjunction with our integration testing methodologies. Finally the debouce button was combined and tested. These test cases were simulated in software and tested on hardware. A regression test of each user test cases was performed to ensure consistency in functionality as code was further developed . The last test is the most destructive test, exploring unprecedented actions by the user. This anomaly includes turning the power off and pressing more than one button down at the same time. If the system passes these tests, then the finished product is deemed to have passed our total quality standards regime and is ready to be demonstrated.

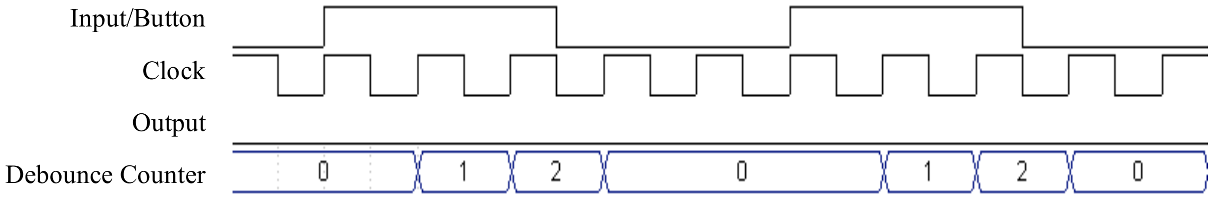
- **Module Testing -** Each component (that does not contain sub-component) was individually tested and evaluated by using ModelSim.
- **Integration Testing -** The modules that require more than one component were tested with MaxPlus and onboard. This was because ModelSim could simulate embedded components properly.
- **Implementation Testing -** The software was tested in both hardware and simulation. The overall system was downloaded each and every user actions were mimic to identify any unwanted results. The tests reveal some unwanted behaviour, for example the clock looped back to 5 instead of zero. All code portions were thoroughly tested and re-tested to guarantee a fully-functional system in step with our regressive testing plans.

Simulation Test Results

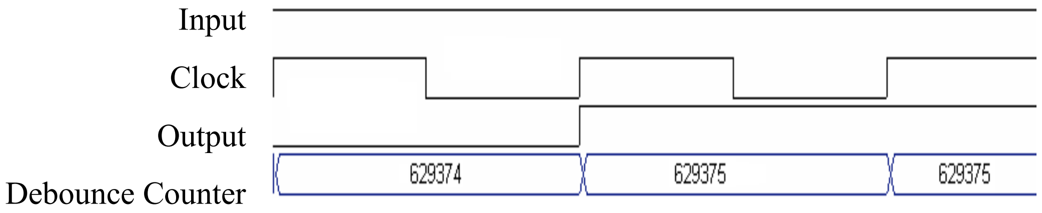
All self-supportive components (does not require additional components) were simulated. The following are the results of the test benches (also see the Appendix for the actual VHDL test benches);-

Component: Debouncer

The following tests whether the Deboucer smooths the input signal.



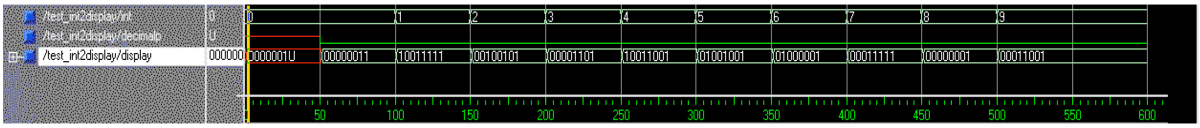
This waveform identifies that the Debounce Counter starts counting on the rising edge clock, when the input is high (button is press) and reset the counts when the input is low and the desire Debounce Counter value was not reached.



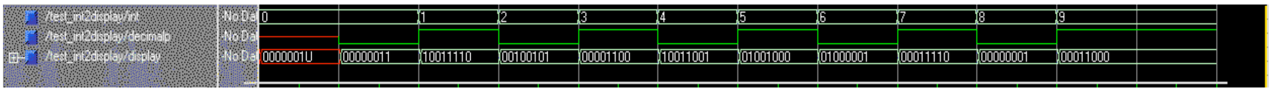
After the input has been high for approximately 25ms, the counter stops and the output is activate on the rising clock edge. Hence the debouncer operates correctly.

Test bench: test_int2display

Aim: To observe the changes in the *display* while changing the *int* variable.
Inputs: *int*
Method: Set the *int* value to count from 0 to 9.
Expected output: The display variable should reflect the binary code needed to display the *int* variable digit.
Results: As expected



Aim: To observe the changes in the display while changing the *int* and *decimalP* variable.
Inputs: *decimalP, int*
Method: Set the *decimalP* to on and off, while setting the *int* value from 0 to 9.
Expected output: The display variable should reflect the binary code needed to display the *int* variable digit and *decimalP*. When *decimalP* is a high then bit 0 of *display* should be a '0' which turns the decimal point on.
Results: As expected.



Test bench: test_stopwatch

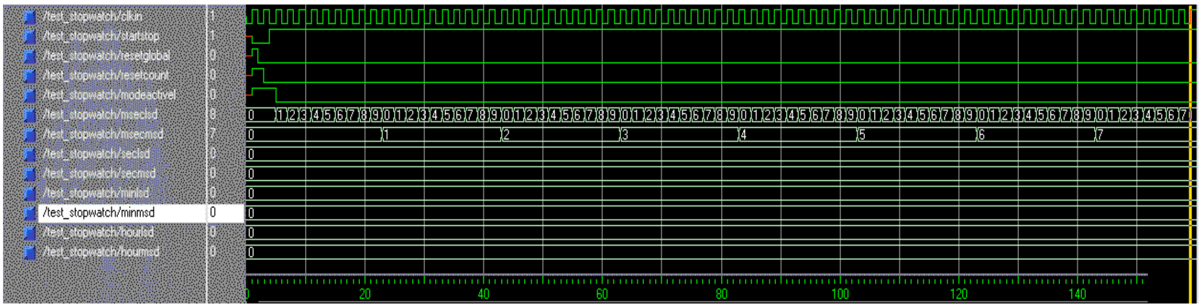
Aim: To observe the changes in the timing variables (minutes, seconds, etc) when counting.

Inputs: *clkin, startStop, resetGlobal, resetCount*

Method: Start the counter and observe the results. The clock is changing every nanosecond for testing purposes only.

Expected output: It should start counting when startStop goes low and resetGlobal goes low. The other variables should not effect the timing variables.

Results: As expected. The hours, minutes and seconds variables can not be tested due to the software limitation. Hence it is extrapolated assume that the hours, minutes and seconds will change accordingly.


/test_stopwatch/clkin 1
/test_stopwatch/startstop 1
/test_stopwatch/resetglobal 0
/test_stopwatch/resetcount 0
/test_stopwatch/modeactivel 0
/test_stopwatch/mseclsd 8
/test_stopwatch/msecmsd 7
/test_stopwatch/seclsd 0
/test_stopwatch/secmsd 0
/test_stopwatch/minlsd 0
/test_stopwatch/minmsd 0
/test_stopwatch/hourlsd 0
/test_stopwatch/hourmsd 0
0
20
40
60
80
100
120
140

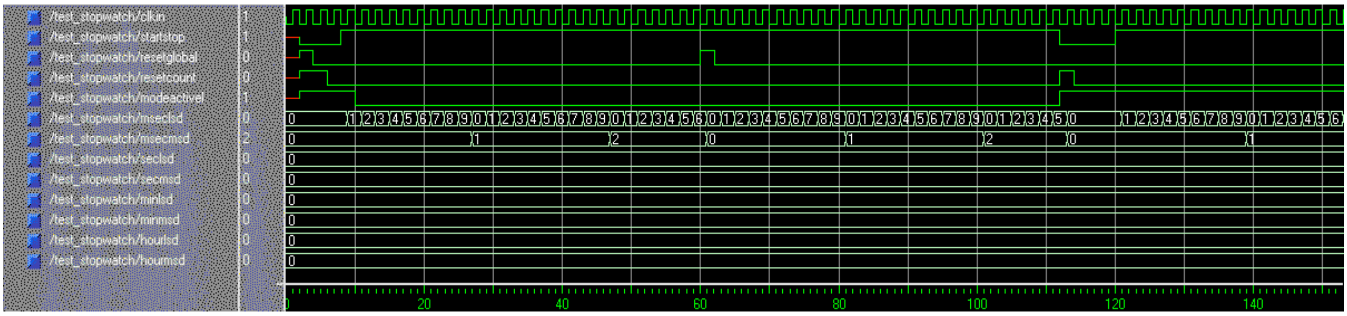
Aim: To observe the changes in the timing variables (minutes, seconds, etc) and also checking the reset conditions.

Inputs: *clkin, startStop, resetGlobal, resetCount*

Method: Start the counter and observe the results. After 60ns, set *resetGlobal* high. After 113ns set resetCount = '1' AND startStop = '0' AND modeActiveL = '1'.

Expected output: It should start counting when startStop goes low and resetGlobal goes low. The other variables should not effect the timing variables (unless resetCount = '1' AND startStop = '0' AND modeActiveL = '1' is satisfied, in which case the timing variables are reset.) The counters should reset when resetGlobal goes high and when the other reset condition is satisfied.

Results: As expected


/test_stopwatch/clkin 1
/test_stopwatch/startstop 1
/test_stopwatch/resetglobal 0
/test_stopwatch/resetcount 0
/test_stopwatch/modeactivel 1
/test_stopwatch/mseclsd 0
/test_stopwatch/msecmsd 2
/test_stopwatch/seclsd 0
/test_stopwatch/secmsd 0
/test_stopwatch/minlsd 0
/test_stopwatch/minmsd 0
/test_stopwatch/hourlsd 0
/test_stopwatch/hourmsd 0
0
20
40
60
80
100
120
140

Test bench: test_clockfield

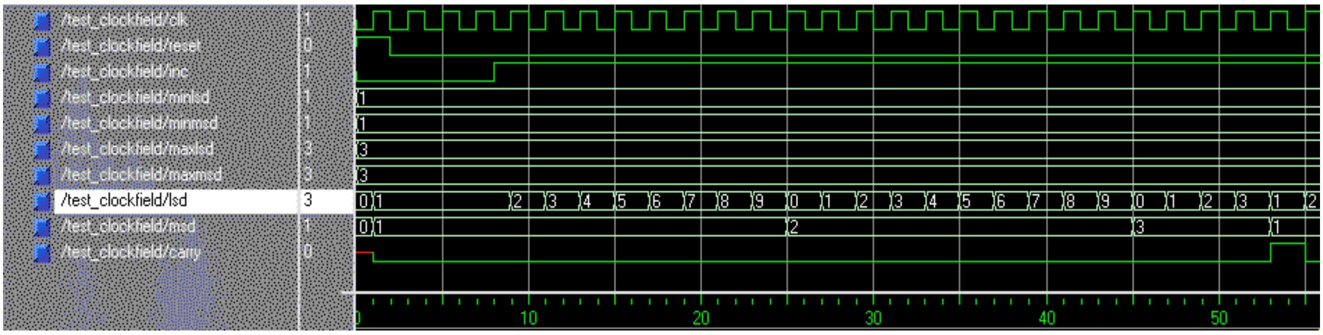
Aim: To test that *carry* goes high when *lsd* = *maxlsd* and *msd* = *maxmsd.*

Inputs: *clk, reset, inc, minlsd, minmsd, maxlsd, maxmsd.*

Method: Set minlsd = 1, minmsd = 1, maslsd = 3, minlsd = 3 and observe the results.

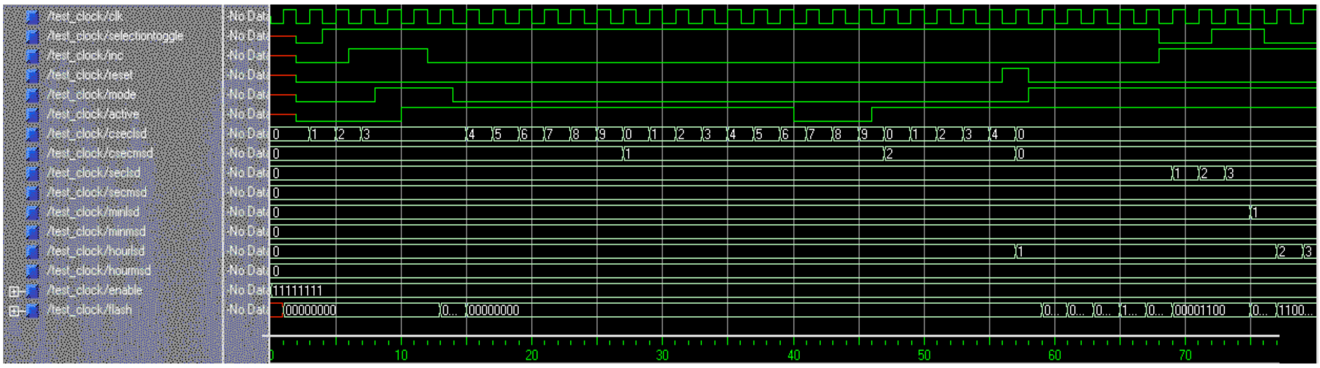
Expected output: Carry should go high when the counter reaches 33 (when lsd = 3 and msd = 3). When this happens it should reset back to 11. Also it should not count while *inc* is low.

Results: As expected. The changeover occurs at 54 ns.

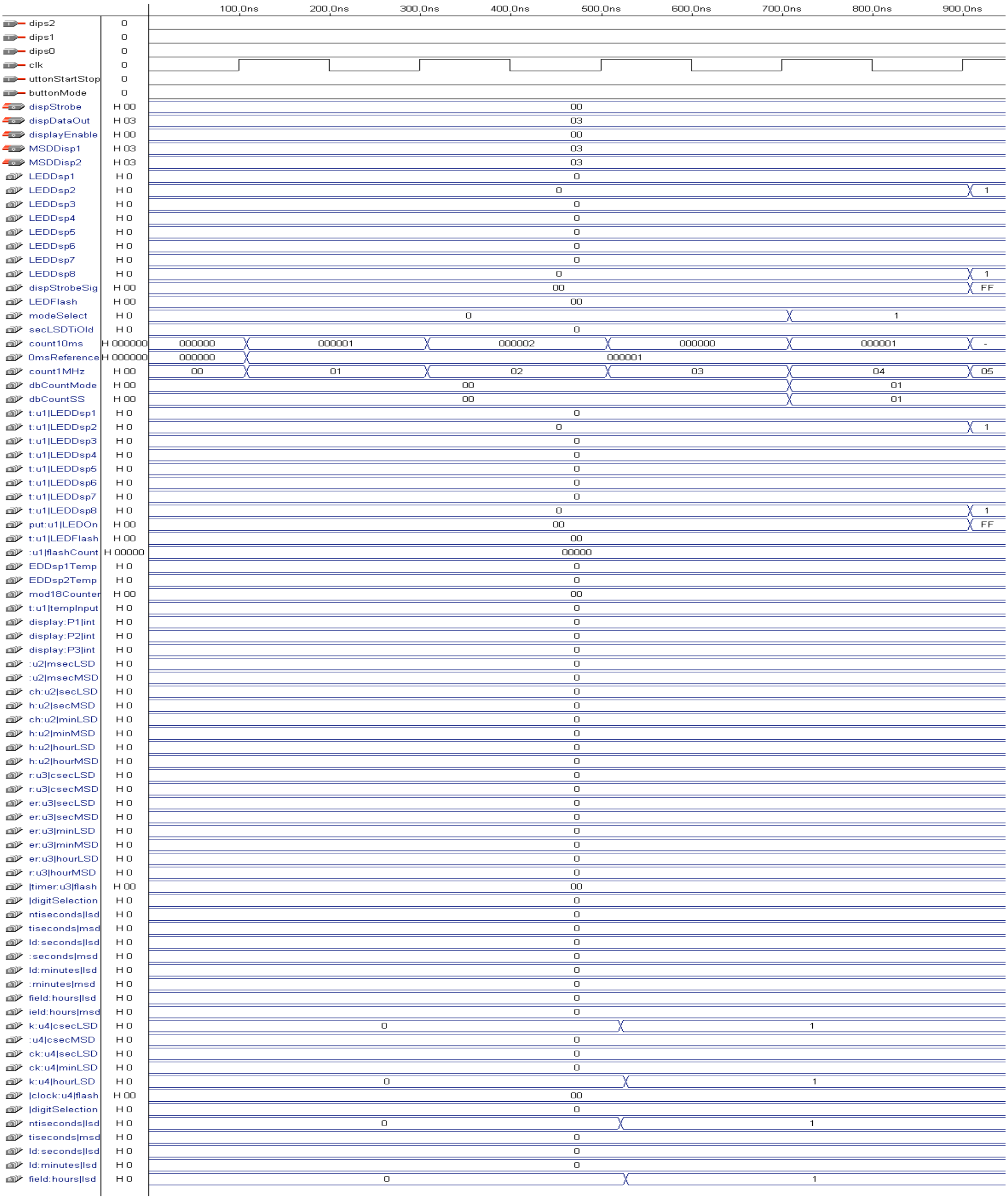


Test bench: test_clock

Aim: To test that *reset* sets the clock to 01:00:00:00 and observe other scenarios.
Inputs: *clk, selectionToggle, reset, inc, mode, active.*
Method: Change inputs to high or low on different time intervals and observe the results.
Expected output: When *mode* goes high, it should stop counting. When reset goes high it should be set to 01:00:00:00. When active goes low, flash should be "00000000".
Results: As expected.



Component: Integrated System



The above demonstrates that the compiled system immediately starts counting the clock and displays the current mode correctly.

Hardware Test Results

The system was downloaded onto the board and rigorously tested. Modules that had embedded component were tested first. Then the fully integrated operation was examined. What follows are the final results from these tests – previous test executions are not included, as they were numerous and, while very important during development, are not interesting within the scope of this document.

User-Based Testing

Test ID	Testing	Inputs/Actions	Results
7	Setting the clock to watch the transition from 00:00:00:00 to 00:00:00:01	Setting of the clock	The transition was as expected.
8	Setting the clock to watch the transition from 11:99:99:99 to 12:00:00:00	Setting of the clock	The transition was as expected.
	Setting the clock to watch the transition from 12:99:99:99 to 01:00:00:00	Setting of the clock	The transition was as expected.
	Test that the system can change modes	Pressing the mode push button	The displayed changed signifying a mode changed.
9	Starting and stoping the stopwatch	Pressing the start/stop push button.	The display changed accordingly.
10	Resetting the stopwatch	Pressing and holding down the start/stop watch	The displayed changed to 00:00:00:00
11	The timer was set to 00:00:10:00 to observe the counting down and alarm sequence	Setting the timer	The timer counted down and until it reached 00:00:00:00. Where it began to the seven segment displays flashed until the stop button was pressed.
12	Setting the timer, such that the display looped past 99:59:59:99	Advance button	The transition looped back to 00:00:00:00.

Anomaly Testing

Test ID	Testing	Inputs/Actions	Results
11	When in set mode does the dip switches alter the operation.	Dip switches and the set mode	The dip-switch will only have an effect when test is off.
12	Testing when the advance switch has an effect.	Pressing the advance switch	The advance switch only has an effect under the set condition.
13	Does the system maintain operations when the power is turned off.	Disconnections of mains	The system looses all functionality.
14	Testing for unstable and unpredictable actions.	Pushing two buttons at the same time	The system responded to the first button pressed.

Personal Reflections

Jarrod Crivelli

The ELE32EDA project helped me develop my problem solving skills and also taught me a valuable lesson of working in a team environment. I was able to see the project come together piece by piece until a final product was reached.

This was a particularly large project and splitting it up evenly between the four members of the group was a real bonus. It showed me that the best way to get a job done is by working together to reach your goals. As a result of this project I have learnt valuable lessons in communication skills, negotiation skills and practical skills.

Being a test engineer I was able to understand the importance of testing when designing and building a product. I learnt how to run simulations in ModelSim and how beneficial they can be. At first I did not have a great understanding of writing test benches and found it very difficult and time consuming. However I progressed from writing simple test benches and eventually writing bigger, more difficult ones to test some of the larger VHDL files.

When the project was complete I felt that everyone had done a terrific job and the final product was flawless when operating. It was a credit to some good planning, designing, progress meetings and communication that we were able to come to a finished product that I was very satisfied with. All things considered I think this project was very challenging and also very rewarding.

Robert Ross

The Electronic Design and Automation subject has further honed my problem solving and time management skills. As with any project work, careful planning must be undertaken to ensure success in delivering a timely and well-developed product. In addition to learning skills in designing and implementing the project end product, communication and group management skills were further developed. This is a real strength of this subject that it allows students to work together on a small-to-medium size project, using their combined skills come together in an effective team.

In terms of the developer environment that we used, we found FPGA Advantage and Max Plus to be quite unstable and was one of the main factors which slowed down the progress on the project. As a design decision we chose to exclusively use Max Plus as our development environment. This decision I believe paid off in the long run in terms of an integrated product and also in productivity. For coding purposes the Max Plus environment was quite efficient. In terms of consulting with our test engineers, they found several difficulties that they ended up working around (using Model Sim as a test environment). This seemed due in part to component style layouts and also trying to start simulations in Max Plus which is quite deficient in simulation capabilities.

In terms of improving the project there are a number of specific areas that could be targeted to improve productivity and therefore the finished product. Clearer descriptions of assessable material should be provided (eg. In terms of project meetings and progress reporting). All the development boards should be tested as some clearly do not work and some have had their display expansion board removed and replaced with a different expansion board that is of no use to EDA students. It would also be advantageous to students if external access (eg. via SFTP/Telnet) could be provided. This could be modelled of a similar set-up to the Computer Science departments external Latcs server, where students can work from home utilizing maximum use of their time. Possibly this could be used then to employ some sort of a CVS (Code Revision System) to accommodate maximum testability and coding access.

Tony de Souza-Daw

I found that the electronic design automation project enhanced my problem solving, organization, team work, communication skills, negotiation skills and practical skills. To see the project worked through the production process to the end was a real insight in how such projects are created. Also the project gave me a real appreciation of how large projects are managed.

FPGA and Max Plus proved to be difficult to use due to instability and excessive functionality. This created simulation issues and test drivers difficulties, resulting in a very time-consuming exercise. Improvements to the project, would see a demonstration simulation of FPGA run from scratch in the lectures (not screen shots – running program). A clearer description of what is to be expected in the progress meetings and the progress report. In addition to lecture on examination preparation. Also copies of the code should be emailed, rather than submission on disk.

All things taken into consideration the electronic project was a beneficial learning experience.

Wade Tregaskis

This design & implementation project was very interesting to work on. As a double-degree student, this semester is the first in which I have used VHDL (also studying CDP in parallel), and I've found it to be a very interesting area, one I hope to pursue further next year and beyond.

This project was also the first in which I had really worked in a team. While first-year EDP is done in pairs, there was little need for interaction between partners as the design could be broken into very distinct blocks. This was not the case with this project, where components I worked on had to be used by other components Rob worked on, and consequently had to have well-defined interfaces and behaviours. All this emphasised strongly the need for good communication, which although was not always the case, did improve as the project went on.

In terms of the implementation itself, I'm largely happy with it. In hindsight there are some things I would do differently – as there usually is in any project – but I think some good decisions and – most importantly - the right compromises were made. It was an interesting and a very important experience to work with Rob on the VHDL code, as it highlighted alternate modes of thought and different solutions available to any particular problem. While Rob & I didn't always see things the same way, we respected each other's decisions and responsibilities, and there was very little friction between us, resulting in very easy integration and a very consistent and reliable final product.

The last point I should make is that the use of various VHDL & FPGA design software packages was very educational, although in many ways for quite the wrong reasons. I am rather disappointed with the software available to us – whether MaxPLUS II, ModelSim or FPGA Advantage – as every team member had huge trouble getting them to work reliably and correctly – sometimes due to poor documentation resulting in inappropriate assumptions or incorrect expectations, but very often due to significant software bugs. This is unfortunate, as it provided a major distraction to getting useful work done. Nonetheless, it was good to use these programs now and remove many [as it turns out] unfounded expectations, which should reduce surprises and difficulties in future. It was immensely useful to develop with both MaxPLUS and Cypress Warp (in CDA) concurrently, as it highlighted very different VHDL implementations (particularly regarding the supposedly standard IEEE libraries).

All in all, I'm very happy with this project. It was, for better or worse, contrasted against the other major project this semester – CSE32PRO – and invariably come out far better in terms of quality of the final product, ease of development, and – most importantly – it's educational and entertainment value. I now greatly look forward to 3rd year electronics project (next year) and the 4th year project (2 years from now).

Appendix A - Test Benches

test_int2display

```
entity test_int2display is -- top level entity of the test bench
end test_int2display; -- has no ports

architecture stimulus of test_int2display is
        component int2display -- declare Unit Under Test (UUT)
                port(
                int      :       in integer range 0 to 10;
                decimalP:        in std_logic;    --on if 1
                display  :       buffer std_logic_vector(7 downto 0));
        end component;

        signal int       : integer range 0 to 10;
        signal decimalP:  std_logic;      --on if 1
        signal display   : std_logic_vector(7 downto 0);

begin
        -- create an instance of the int2display circuit.
        INT_2: int2display port map(int => int, decimalP => decimalP, display => display);


        t: process
        begin
                wait for 50 ns;
                int <= 0;
                decimalP <= '0';

                wait for 50 ns;
                int <= 1;
                decimalP <= '1';

                wait for 50 ns;
                int <= 2;
                decimalP <= '0';

                wait for 50 ns;
                int <= 3;
                decimalP <= '1';

                wait for 50 ns;
                int <= 4;
                decimalP <= '0';

                wait for 50 ns;
                int <= 5;
                decimalP <= '1';

                wait for 50 ns;
                int <= 6;
                decimalP <= '0';

                wait for 50 ns;
                int <= 7;
                decimalP <= '1';

                wait for 50 ns;
                int <= 8;
                decimalP <= '0';

                wait for 50 ns;
                int <= 9;
                decimalP <= '1';

                wait for 100 ns;
        end process t;
end;
```

test_clockfield

```
entity test_clockfield is -- top level entity of the test bench
end test_clockfield; -- has no ports

architecture stimulus of test_clockfield  is
        component clockfield  -- declare Unit Under Test (UUT)
                PORT(
                clk              : IN std_logic;
                reset            : IN std_logic;
                inc              : IN std_logic;
                minLSD   : IN integer RANGE 0 to 9;
                minMSD   : IN integer RANGE 0 to 9;
                maxLSD   : IN integer RANGE 0 to 9;
                maxMSD   : IN integer RANGE 0 to 9;
                lsd              : BUFFER integer range 0 to 9;
                msd              : BUFFER integer range 0 to 9;
                carry            : BUFFER std_logic);
        end component;
        signal   reset            :  std_logic;
        signal   inc              :  std_logic;
        signal   minLSD           :  integer RANGE 0 to 9;
        signal   minMSD           :  integer RANGE 0 to 9;
        signal   maxLSD           :  integer RANGE 0 to 9;
        signal   maxMSD           :  integer RANGE 0 to 9;
        signal   lsd              :  integer range 0 to 9;
        signal   msd              :  integer range 0 to 9;
        signal   carry            :  std_logic;

begin
        -- create an instance of the int2display circuit.
        CLKFIELD: clockfield port map(clk => clk, reset => reset, inc => inc, minLSD => minLSD, minMSD => minMSD, maxLSD
        => maxLSD,
        maxMSD => maxMSD, lsd => lsd, msd => msd, carry => carry);


        clk_gen:process
        begin
                clk <= '0';
                wait for 1 ns;
                clk <= '1';
                wait for 1 ns;
        end process;


        t: process
        begin
                reset <= '1';
                inc <= '0';
                minLSD <= 1;
                minMSD <= 1;
                maxLSD <= 3;
                maxMSD <= 3;

                wait for 2 ns;
                reset <= '0';

                wait for 6 ns;
                inc <= '1';

                wait for 50 ns;

        end process t;
end;
```

test_stopwatch

```
entity test_stopwatch is -- top level entity of the test bench
end test_stopwatch; -- has no ports

architecture stimulus of test_stopwatch is
        component stopwatch -- declare Unit Under Test (UUT)
                port(
                clkIn: in std_logic; --100Hz
                startStop: in std_logic; --toggles start/stop
                resetGlobal: in std_logic; --global reset for board
                resetCount: in std_logic; --resets the count when in stop mode to 00:00:00:00
                modeActiveL: in std_logic; --if 1 this is active and the buttons apply, if 0 is inactive and startStop
        doesn't work
                msecLSD: buffer integer range 0 to 9;
                msecMSD: buffer integer range 0 to 9;
                secLSD: buffer integer range 0 to 9;
                secMSD: buffer integer range 0 to 9;
                minLSD: buffer integer range 0 to 9;
                minMSD: buffer integer range 0 to 9;
                hourLSD: buffer integer range 0 to 9;
                hourMSD: buffer integer range 0 to 9);
        end component;

        signal clkIn:  std_logic;
        signal startStop:  std_logic; --toggles start/stop
        signal resetGlobal:  std_logic;     --global reset for board
        signal   resetCount:  std_logic; --resets the count when in stop mode to 00:00:00:00
        signal   modeActiveL:  std_logic;  --if 1 this is active and the buttons apply, if 0 is inactive and startStop
        doesn't work
        signal   msecLSD:  integer range 0 to 9;
        signal   msecMSD:  integer range 0 to 9;
        signal   secLSD:  integer range 0 to 9;
        signal   secMSD:  integer range 0 to 9;
        signal   minLSD:  integer range 0 to 9;
        signal   minMSD:  integer range 0 to 9;
        signal   hourLSD:  integer range 0 to 9;
        signal   hourMSD:  integer range 0 to 9;

begin
        -- create an instance of the int2display circuit.
        STOPW: stopwatch port map(clkIn => clkIn, startStop => startStop, resetGlobal => resetGlobal, resetCount =>
        resetCount,
        modeActiveL => modeActiveL, msecLSD => msecLSD, msecMSD => msecMSD, secLSD => secLSD, secMSD => secMSD, minLSD
        => minLSD,
        minMSD => minMSD, hourLSD => hourLSD, hourMSD => hourMSD);

        clk_gen: process
        begin
```

```
                wait for 1 ns;
                clkin <= '1';
                wait for 1 ns;
        end process;

        t: process
        begin

                wait for 2 ns; --reset everything
                resetGlobal <= '1';
                resetCount <= '1';
                startStop <= '0';
                modeActiveL <= '1';

                wait for 2 ns;
                resetGlobal <= '0';

                wait for 2 ns;
                resetCount <= '0';

                wait for 2 ns; --start counting
                startStop <= '1';

                wait for 2 ns;
                modeActiveL <= '0';

                wait for 50 ns; --reset
                resetGlobal <= '1';

                wait for 2 ns;
                resetGlobal <= '0';

                wait for 50 ns;
                resetCount <= '1'; --reset
                startStop <= '0';
                modeActiveL <= '1';

                wait for 2 ns;
                resetCount <= '0'; --not reset

                wait for 6 ns;
                startStop <= '1'; --start counting

                wait for 100 ns;
        end process t;
end;
```

test_clock

```
entity test_clock is -- top level entity of the test bench
end test_clock; -- has no ports

architecture stimulus of test_clock  is
        component clock  -- declare Unit Under Test (UUT)
                PORT(
                clk                         : IN std_logic; -- clkIn
                selectionToggle                     : IN std_logic; -- modeToggle
                inc                         : IN std_logic; -- buttonstartStop
                reset                       : IN std_logic; -- resetGlobal/resetCount
                mode                        : IN std_logic; -- setMode
                active                      : IN std_logic; -- modeActive

                csecLSD                     : BUFFER integer range 0 to 9; -- msecLSD
                csecMSD                     : BUFFER integer range 0 to 9; -- msecMSD
                secLSD                      : BUFFER integer range 0 to 9;
                secMSD                      : BUFFER integer range 0 to 9;
                minLSD                      : BUFFER integer range 0 to 9;
                minMSD                      : BUFFER integer range 0 to 9;
                hourLSD                     : BUFFER integer range 0 to 9;
                hourMSD                     : BUFFER integer range 0 to 9;

                enable                      : OUT std_logic_vector(7 downto 0); -- dispStrobeSig
                flash                       : BUFFER std_logic_vector (7 DOWNTO 0)); -- LEDFlash
        end component;

        signal   clk                        :  std_logic; -- clkIn
        signal   selectionToggle                    :  std_logic; -- modeToggle
        signal   inc                        : std_logic; -- buttonstartStop
        signal   reset                      : std_logic; -- resetGlobal/resetCount
        signal   mode                       :  std_logic; -- setMode
        signal   active                     : std_logic; -- modeActive
        signal   csecLSD                    : integer range 0 to 9; -- msecLSD
        signal   csecMSD                    : integer range 0 to 9; -- msecMSD
        signal   secLSD                     : integer range 0 to 9;
        signal   secMSD                     :  integer range 0 to 9;
        signal   minLSD                     :  integer range 0 to 9;
        signal   minMSD                     :  integer range 0 to 9;
        signal   hourLSD                    : integer range 0 to 9;
        signal   hourMSD                    :  integer range 0 to 9;
        signal   enable                     :  std_logic_vector(7 downto 0); -- dispStrobeSig
        signal   flash                      :  std_logic_vector (7 DOWNTO 0); -- LEDFlash

begin
        -- create an instance of the int2display circuit.
        CLCK: clock port map(clk => clk, reset => reset, inc => inc, minLSD => minLSD, minMSD => minMSD, selectionToggle
        mode => mode, active => active, csecLSD => csecLSD, csecMSD => csecMSD, secLSD => secLSD, secMSD => secMSD,
        hourLSD => hourLSD,
        hourMSD => hourMSD, enable => enable, flash => flash);


        clk_gen:process
        begin
                clk <= '0';
                wait for 1 ns;
                clk <= '1';
                wait for 1 ns;
        end process;


        t: process
        begin
                wait for 2 ns;
                selectionToggle <= '0';
                inc <= '0';
                reset <= '0';
                mode <= '0';
                active <= '0';

                wait for 2 ns;
                selectionToggle <= '1';

                wait for 2 ns;
                inc <= '1';

                wait for 2 ns;
                mode <= '1';

                wait for 2 ns;
                active <= '1';

                wait for 2 ns;
                inc <= '0';

                wait for 2 ns;
                mode <= '0';

                wait for 20 ns;
                active <= '1';

                wait for 6 ns;
                active <= '0';

                wait for 6 ns;
                active <= '1';

                wait for 10 ns;
                reset <= '1';

                wait for 2 ns;
                reset <= '0';
                mode <= '1';

                wait for 10 ns;
                inc <= '1';
                selectionToggle <= '0';

                wait for 4 ns;
                selectionToggle <= '1';

                wait for 4 ns;
                selectionToggle <= '0';

                wait for 4 ns;
                selectionToggle <= '1';


                wait for 2 ns;
        end process t;
end stimulus;
```