# Introducing the Lab Environment

EE/CSE 371 Lab 1

University of Washington



| Name | Student ID |
|---|---|
| Andrique Liu | 1365608 |
| Emraj Sidhu | 1329919 |
| Nikhil Grover | 1435083 |

**TABLE OF CONTENTS**

## ABSTRACT

This report includes a comprehensive lab report about Lab 1: Introducing the Lab Environment. This report has a relevant introduction section which serves to introduce the content of Lab 1, as well as design details found in the Design Specification, Software Implementation, and Hardware Implementation sections. Then, the testing and results sections follow. Finally, this report will present an analysis of errors, a summary of the report, and a conclusion. An appendices section is included at the end of the report, containing requirements documents, relevant screenshots, diagrams, and subsequent documentation paralleling design implementation.

## INTRODUCTION

Lab 1 was used for several purposes, mostly involving refreshing our usage of tools as well as introducing new tools. Firstly, Lab 1 refreshed our usage with the Altera DE1-SoC board and the Quartus II development environment, In addition, we refreshed our working knowledge of the Verilog HDL as well as three of the four modeling levels supported by Verilog. Next, we increased our proficiency in the use of Icarus Verilog, a means to model Verilog constructs. Signal Tap is a logic analyzer included in the Quartus IDE, and was also covered in Lab 1. Finally, we used the C language to write and debug a simple C program. Using these tools, we constructed four simple counters.
Tools used included: the DE1-SoC board, the Quartus II IDE and its logic analyzer, iVerilog, the GCC compiler, simple text editors, and more.

## DESIGN SPECIFICATION

### CurrConverter.c Design Specification

#### Abstract

This report includes the design specification for the CurrConverter.c C program. This design specification includes an introduction, inputs and outputs to the program, as well as the major functions used in the program.

#### Introduction

The final C program is contained in one c file, CurrConverter.c. This program is intended to print out user-requested exchange rates, and then calculate equivalent currency using user-defined amounts. The methods used in this program included C programming tools like the standarad library, etc.

#### Inputs to Program

This program takes the following inputs: whether the user would like to exchange from USD to a foreign currency or vice versa, a desired currency, and an amount to exchange. There is also an option to define a custom currency value.

#### Outputs to Program

CurrConverter.c outputs the an exchange rate and also a final currency converstion based upon its inputs. Relevant information is printed to the console for the user.

#### Major Functions

CurrConverter.c includes the standard input and output library to access basic input and output functions. The main program is relatively abstracted, such that implementation details are hidden from the main function, and the behavior of the program is then captured in the program's functions.
The printOptions function prints out available currency options, and also prompts if the user would like a custom currency. The USDtoF and FtoUSD functions provide a USD to foreign currency conversion and a foreign currency to USD conversion, respectively. The findCurrency function matches the user-defined currency with an appropriate currency and its value.

**Counters Design Specification**
Each Counter is listed below, alongside a description of what design specifications were implemented for each associated counter. Additionally, the DFlipFlop model given in the lab assignment was used in each counter. This had the effect of giving each counter an active-low reset.

**Ripple Down Counter: Gate Level**
The ripple down counter was designed so that on reset, the output bit value would be 4'b0000, and would then underflow to 4'b1111, and continue counting down normally until another underflow occured, continuing the above pattern. Each piece of wire and logic within the module was connected using gate implementation (i.e. Not(out, in) ). The counter module included purely combinational logic, as the dflipflop module provided the necessary sequential code. A schematic that was used for the code implementation can be viewed below in the appendix section, under "Ripple Down Counter."
**Synchronous Down Counter**
The synchronous down counter was designed to start at the bit value 4'b1111 on reset, and once reaching 4'b0000, an overflow would occur and the next clock cycle output would be 4'b1111. The module implementation followed a dataflow level module, where every piece of logic was created by using mathematical operators. Additionally, the synchronous down counter used the Dflipflop module given in the lab. A block diagram of the synchronous down counter can be seen below in the appendix, under "Synchronous Down Counter."
**Synchronous Johnson Counter**
The synchronous johnson counter was the only counter that did not include the dflipflop module given in the lab document. To implement the counter, a FSM was created with enough states and associated output values to behave according to the textbook johnson counter definition. A diagram of the finite state machine for the johnson counter is available in the appendix under, "Johnson Counter State Machine". A behavioral design specification was used to implement the FSM for the johnson counter, resulting in a module being written that told the johnson counter exactly how to behave (including delivering correct outputs for each state).
**Synchronous Down Counter (Schematic Level)**
This synchronous down counter was designed with the exact same behaviour in mind as the above jonhson counter, however, a schematic level implementation was used instead of a dataflow level model. To implement the schematic level model, the circuit was drawn using Quartus II's CAD and schematic tool, resulting in the schematic which can be viewed in the appendix under "Synchronous Down Counter." After the schematic was built, quartus was used

to export the schematic design to verilog code, which was then used to connect to the associated hardware inputs and outputs, sufficiently completing the counter specifications.

## SOFTWARE IMPLEMENTATION

The software portion of this lab is found in CurrConverter.c, a program that acts as an exchange rate and foreign currency calculator.
From a high level perspective, this program prompts the user for what they would like to do: convert from USD to foreign currency, or vice versa. Then, the user is asked to define a foreign currency, and is also allowed to define a custom currency amount. The exchange rate is printed using this data, and the program then prompts for the desired input amount to exchange. The program then prints out the output, specifying the currency yielded by the inputted currency.
Refining this view, the program is defined as several functions and other constructs to produce the final program. The main program accesses all major functions in an abstracted manner, leaving implementation details inside these major functions.
The printOptions, USDtoF and FtoUSD functions take in no inputs other than user-inputted data using scanf and getchar. These three functions also do not return any data. The fourth function, printOptions takes a char and a char array. The char corresponds to the user input, and the currency array holds the relevant name of currency.
The printOptions function simply prints out possible default currencies.
USDtoF allows the user to convert from a reference of 1.00 USD to the desired foreign currency. This function will then print the appropriate exchange rate, and will ask the user for the amount of USD they will like to exchange for the foreign currency. Then, the function will print out the converted amount. FtoUSD functions similarly, except converting from a reference of 1.00 of foreign currency, and asks the user for the amount of foreign currency they wish to exchange for USD.
In each of these functions, an array of size three is initialized, providing space for abbreviated currency names. Upon the appropriate user input, the function will eventually print out this currency name. To copy from arrays, the strcpy function is used.
The findCurrency function takes in a char uinput and and a char array currency. Char input keeps track of the user input, and the array currency provides space for currency names to be written and printed. A double foreignVal is also declared, which stores the appropriate currency value relative to USD. A switch statement is used in this function to monitor the user's input and determine the correct currency. If none of the options listed in printOptions are selected, the user is notified that they have entered invalid input. findCurrency also returns this foreignVal variable for printing/indication purposes.
To read in user input, scanf and getchar are used. Scanf is used to read user input and place user input into a variable given the variable's address. Getchar is used to clear the newline from the user-inputted line.
Regarding the flow of control, the program will place the flow of control between the program and the user when prompted. The user is prompted through scanf and getchar functions.
Potential complications are also present when using this program. For instance, negative values entered into the program could result in invalid results, or results that simply do not make sense. This may also be the case when the user inputs 0 for custom currency values. Additionally, this program assumes that the user will enter input without extra whitespace. In this program, getchar

accounts only for the newline inputted by the user while entering inputs, assuming other whitespace is not included.

**Pseudocode**

The complete behavior of the program is captured in the following pseudocode:

```
int main()
{
        while (user has not quit program) {
                Welcome, what would user like to do?;
                F to USD or USD to F?;
                Use appropriate conversion function;
                Find appropriate currency abbreviation;
                Print out exchange rates;
                Amount to exchange?;
                Print out conversion;
        }
}
```

## HARDWARE IMPLEMENTATION

Two families of counters were implemented using iverilog, a synchronous down counter and a ripple down counter. The lab required our group to implement each counter using different top-level design decisions, however, each family of counter shared a schematic. The Synchronous Up Counter block diagram and the Ripple down counter block diagram can be observed in the appendix. In addition to these two families of counters, a johnson counter was also implemented. The johnson counter was implemented using behavioral programming, specifically, by designing a state machine that handled each subsequent state in the correct fashion to behave like a johnson counter. This implementation design choice was completely different than the manner in which the synchronous up counter and ripple counters were implemented, as the respectively mentioned counters were designed with their schematic and gate implementations in mind. Refer to the appendix to view the state machine used to implement the johnson counter.

Note on metastability: Due to the lack of an input signal (other than the clock) used for the counters implemented in verilog, it was deemed unnecessary to include additional DFFs to control metastability..

## FAILURE MODES ANALYSIS

In this lab, failure modes analyses were also conducted on our first three counters. Our analysis methods examined the effects of SA0 and SA1, or signal stuck and 0 and signal stuck at 1, respectively. The signals we examined were each counter's inputs, outputs, and internal signals. In our analysis, we discussed the effects on the signal in question and the operation of the counter.

First, we analyzed the effects of SA0 on the synchronous down counter. With SA0 of the clock signal, the counter would never operate, as there would be no clock edges for the counter to

operate from. Similarly, for the reset signal, the counter would never operate. This is due to the counter's reset operating on active low, such that reset would constantly be enabled each clock cycle. For the outputs, the counter would simply output 4 bits low. SA0 on internal signals also results in buggy behavior. For instance, a 0 output from one DFF to another could delay or hasten the counting sequence.
The synchronous down counter also experiences buggy behavior under SA1. If the clock were to only see high, the counter would not progress, as the sequential logic operates on rising clock edges. However, for the reset, the counter would function normally due to its active low nature. For the output, the counter would simply output four bits high. SA1 on internal signals could also result in buggy behavior. For instance, if the wrong signal is sent from one DFF to another, the counting sequence could be impeded or hastened.

Second, we analyzed the effects of SA0 on the ripple down counter. For the reset signal, the counter will not decrement as the bit pattern will stall at 0. In regards to the outputs, the LEDs will not light up as the counter would output 4 bits low. Lastly for the clock, the counter would never progress as the circuit will never see the next rising clock edge in order to change its values.

Then, we analyzed the effects of SA1 on the ripple down counter. For the reset signal, the counter will continue to work and decrement the bit pattern. However, this could result in some faulty behaviour as the counter will not stop decrementing even if the user presses the reset key. For the outputs, the LEDs will light up and remain on as the counter would output 4 bits high. The counter, however, would not work if the clock signal is SA1 as it will never see the next rising clock edge in order to change its values.

Second, we analyzed the effects of SA0 on the Johnson down counter. For the reset signal, the counter will stall at 0 and not decrement. Similarly,if the clock signal is SA0, the counter will not work as there will be no positive edge of the clock and hence, would not be able to change bit patterns In regards to the outputs, the counter would ouput four bits low. This can be issue as the LEDs will always remain on even in instances where they should be off according to corresponding bit pattern.

Then, we analyzed the effects of SA1 on the Johnson down counter. If the reset signal was to only see high, the counter would continuously decrement even if the user is pressing the reset key. If the clock signal was to only see high, the counter would not work as there would be no positive clock edge for it to operate from. Lastly, for the outputs if they were SA1, the counter would output four bits high and the LEDs would remain on, even in cases where they should be off depending on the corresponding bit pattern.

## DESIGN TESTING

### CurrConverter.c Design Testing

Testing for the currency converter program was conducted by selecting values to input into the program, and then verifying correct calculations by observing the outputs.

To test, we selected numerical currency values to input into the program, and also selected from the default list of currencies. Then, we noted whether the correct exchange rate and conversion were printed.
As an example, if option 2 is selected to convert from foreign to USD, along with Australian Dollars (‘A’/AUD), the exchange rate is 0.73 USD per 1.00 USD. If the amount of AUD to exchange is 8.00, the program outputs 5.84 USD, as expected.
Design testing also accounted for potentially buggy cases, such as when the user inputs invalid input such as negative values, 0s, values with trailing whitespace, and more. For instance, trailing whitespace can “delay” the program, such that a getchar will remove a space, but a newline remains in the buffer for the program to read.

**PRESENTATION AND RESULTS, ANALYSIS OF RESULTS**

Testing of CurrConverter.c procured expected results. As per design specifications and the requirements, the program successfully converts to and from foreign currency, printing out exchange rates and the desired conversion.
Hardware testing of the counters developed in the lab procured expected results.
As per design specifications, the following counters had the respective results:

**Ripple Down Counter: Gate Level:** The ripple down counter behaved as intended- at the first clock cycle (or on reset) the DE1-SOC’s LEDR[3:0] were all off, and the following clock cycle each light was on. The lights acted as stored bit values and displayed the bit values in the order expected from the counter.
Given this analysis, the ripple down counter worked as expected.
**Synchronous Down Counter:** The synchronous down counter was connected to the same DE1-SOC hardware as the ripple down counter, and in accordance with the design spec given in the lab report, behaved as intended. The counter counts up, starting at 4’b1111, and overflows at 4’b0000 back to 4’b1111.
Given this analysis, the synchronous down counter behaves as intended and fulfills the design specification requirements.
**Synchronous Johnson Counter**
The johnson counter module was connected to the LEDR outputs of the board, and SW[0] was used as the reset for the counter. A slow enough clock was used to watch the counter increment in the correct pattern. After connecting the johnson counter module to the DE1-SOC board, the counter performed as expected, and additionally reset as intended.
Due to the performance of the the johnson counter on hardware, our group is led to believe that the johnson counter adequately completes the lab requirements and specifications.
**Synchronous Down Counter (Schematic Level)**
The schematic level down counter was tested in the exact same fashion as the dataflow level up counter module. It performed in the same manner, and displayed the correct bit sequence per clock cycle.
Due to the down counters performance, our group came to the conclusion that the counter adequately fulfills the design specification.
**The Signal Tap II Logic Analyzer:**

The displayed signal outputs are basically the same as the ideal ones. However, since in the signal tap II, we directly used the built-in 50 MHz clock, state changes of the reset switch became extremely sensitive, which caused the output signal being reseted more than once during the measurement. This is possibly due to the switch contact bouncing after being switched on. In the future, perhaps debouncing should be introduced for sensitive applications. But as shown in the Figures 7~9, the signal outputs behaves as we expected as soon as the reset signal becomes stable.

## ANALYSIS OF ERRORS

### CurrConverter.c Errors

As mentioned, CurrConverter.c encounters errors when the user enters in invalid input. The program operates as expected assuming the user enters valid input, but not all error cases are caught in the program- additionally, the scope of errors to catch may prove to be large. For example, adding a space to the end of an input will result in a "delay" to the entire program, such that the input buffer will have a newline- or more- that has yet to be cleared by getchar.
When the user inputs 0 when converting from foreign currency to USD, the program will output "infinite" for USD per 1.00 custom currency. Also, entering negative values may simply result in outputs that do not make sense.
However, when the user enters inputs that the program expects, the program encounters no errors during operation.

### Counter Errors

Although our counters encountered no errors, it is easy to simulate cases where the counters may have run into issues. A first example that springs to mind would be issues with hardware on the DE1-SOC. Even if simulations showed the counting applications working correctly, inherent hardware difficulties could potentially cause issues (such as fault LED display lights), and would potentially be mis-diagnosed during troubleshooting. In this situation, running additional simulations to rule out hardware failure would be a likely solution to pinpoint the faulty device.

On other hand, a counter that was not simulating as it should could likely be troubleshot as having faulty/mismanaged code. A strict debugging session could potentially be a first step towards alleviating the counter issues. If this did not fix the problem, a next step that may be considered would be a redesign of the project. For example, ModelSim may be used as a preliminary design tool to ensure that our design theoretically should work. On the other hand, Signaltap may be used when modeled behavior does not align with real-world behavior, or if we require a real, physical measurement of our design.

Although our exceptionally behaved code did not present itself with the opportunity to practice our error-handling skills, it is still beneficial to consider potential errors that may arise, and how to deal with such issues.

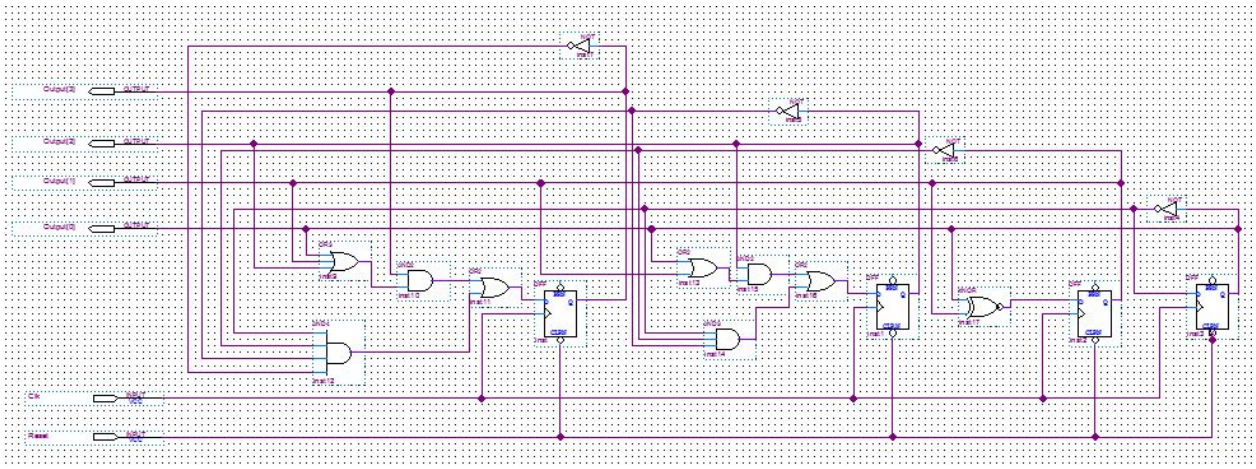
## SUMMARY

This lab report has included a comprehensive overview of Lab 1's problems, our approaches to these problems, and the tools involved in the project. Lab 1: Introducing the Lab Environment introduced several tools which will be used for the remainder of the course, such as: the SignalTap logic analyzer, Quartus as a development environment and means of programming our board, the DE1-SoC board, simple C programming, Verilog as an HDL to model circuits, and more. The Design Specification in this report served to analyze the characteristics of the software and hardware portions of the lab, providing the criteria our designs were required to meet. The software and hardware implementation portions of the report discussed implementation details of our designs, going into further detail by explaining our design choices and how they functioned. A testing section was covered, which discussed testing our design as well as potential test cases. Finally, a concluding portion of the report was covered, which served to analyze our results and errors, as well as conclude this lab report. After this section of the report is an Appendices section which holds relevant screenshots, waveforms, and the like.

## CONCLUSION

This section serves to conclude the lab report of Lab 1: Introducing the Lab Environment. This lab project was effective in teaching us the tools necessary for future projects as well as further establishing the fundamental skills used in this course. As far as suggestions and recommendations go, we feel it is best that future lab specifications mention the use of Verilog and SystemVerilog in this course, specifying that either of the two may be used in future lab projects. Additionally, a ModelSim portion should be included so as to refresh students with ModelSim techniques.

## APPENDICES



Block Diagram 1: Synchronous Down Counter

```
C:\EE371\Lab1\iVerilog>vvp rippleDownTop
VCD info: dumpfile rippleDown.vcd opened for output.
                   5qOut = 0000
                  45qOut = 1111
                  55qOut = 1110
                  65qOut = 1101
                  75qOut = 1100
                  85qOut = 1011
                  95qOut = 1010
                 105qOut = 0000
                 125qOut = 1111
                 135qOut = 1110
                 145qOut = 1101
                 155qOut = 1100
                 165qOut = 1011
                 175qOut = 1010
                 185qOut = 1001
                 195qOut = 1000
                 205qOut = 0111
                 215qOut = 0110
                 225qOut = 0101
                 235qOut = 0100
                 245qOut = 0011
                 255qOut = 0010
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 265 ticks.
```

Figure 1. Simulation of Ripple Down Counter on iVerilog

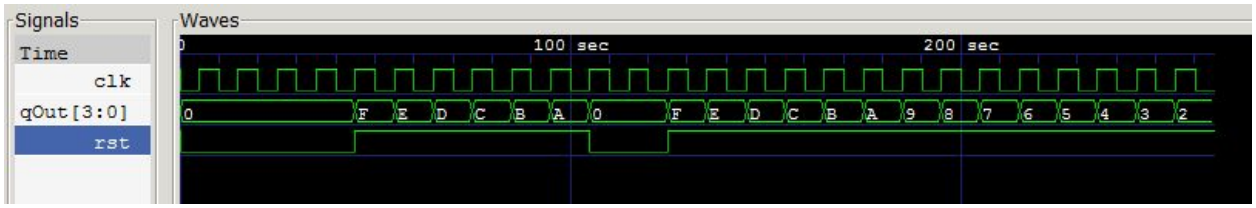



Figure 2. I/O Waveform of Ripple Down Counter

```
C:\EE371\Lab1\iVerilog>vvp syncDownTop
VCD info: dumpfile syncDown.vcd opened for output.
                   5qOut = 0000
                  45qOut = 1111
                  55qOut = 1110
                  65qOut = 1101
                  75qOut = 1100
                  85qOut = 1011
                  95qOut = 1010
                 105qOut = 1001
                 115qOut = 1000
                 125qOut = 0111
                 135qOut = 0110
                 145qOut = 0000
                 165qOut = 1111
                 175qOut = 1110
                 185qOut = 1101
                 195qOut = 1100
                 205qOut = 1011
                 215qOut = 1010
                 225qOut = 1001
                 235qOut = 1000
                 245qOut = 0111
                 255qOut = 0110
                 265qOut = 0101
                 275qOut = 0100
                 285qOut = 0011
                 295qOut = 0010
                 305qOut = 0001
                 315qOut = 0000
                 325qOut = 1111
                 335qOut = 1110
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 345 ticks.
```

Figure 3. Simulation of Synchronous Down Counter on iVerilog

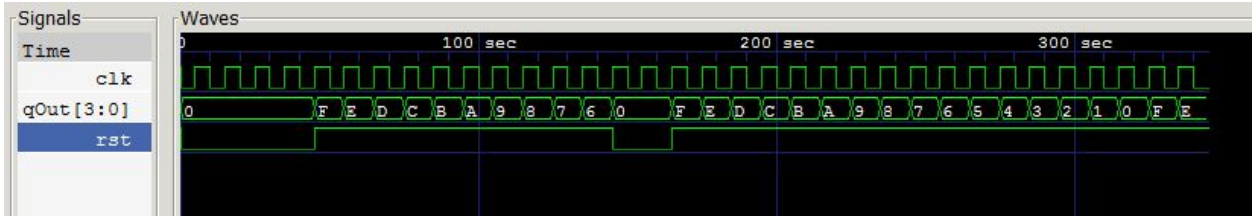



Figure 4. I/O Waveform of Synchronous Down Counter

```
C:\EE371\Lab1\iVerilog>vvp JohnsonCounterTop
VCD info: dumpfile johnsonCounter.vcd opened for output.
                   5qOut = 1111
                  45qOut = 0111
                  55qOut = 0011
                  65qOut = 0001
                  75qOut = 0000
                  85qOut = 1000
                  95qOut = 1100
                 105qOut = 1111
                 125qOut = 0111
                 135qOut = 0011
                 145qOut = 0001
                 155qOut = 0000
                 165qOut = 1000
                 175qOut = 1100
                 185qOut = 1110
                 195qOut = 1111
                 205qOut = 0111
                 215qOut = 0011
                 225qOut = 0001
                 235qOut = 0000
                 245qOut = 1000
                 255qOut = 1100
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 265 ticks.
```

Figure 5. Simulation of Johnson Counter on iVerilog

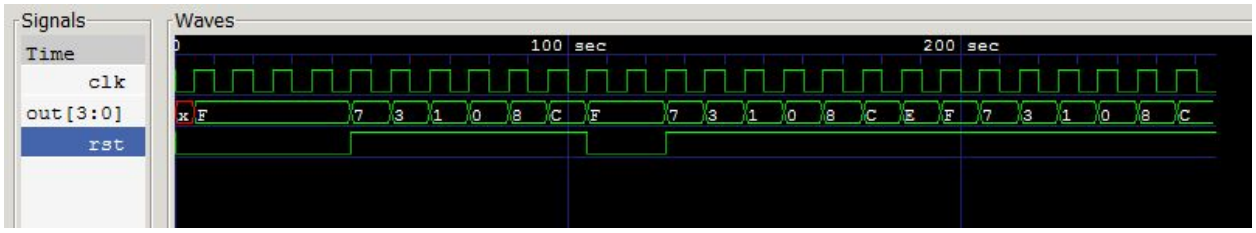



Figure 6. I/O Waveform of Johnson Counter

## SignalTap Logic Analyzer Outputs

In this lab, we used the SignalTap logic analyzer tool to study our designs. In each of the logic analyzer outputs, we triggered the logic analyzer to capture the output at the rising edge of the reset signal (active low). Then, we examined the output; in each of our captured outputs, the output shows counter output around the third state and beyond.

We noticed several characteristics of our design using SignalTap. For instance, the switch input seems very sensitive, as we observed significant button bouncing. Additionally, we noticed that using the 50 MHz clock, the counter output was prone to randomly resetting despite there being no reset signal.


Figure 7 SignalTap Logic Analyzer Results of Ripple Down Counter

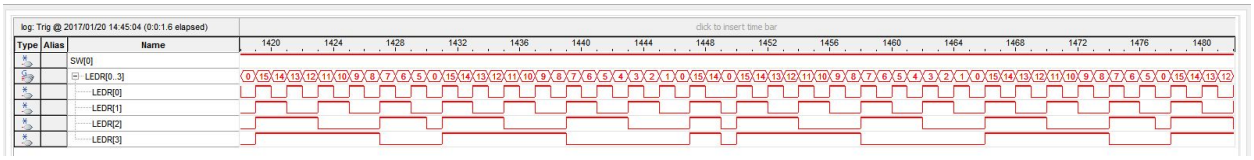



Figure 8 SignalTap Logic Analyzer Results of Synchronous Down Counter


Figure 9 SignalTap Logic Analyzer Results of Johnson Down Counter


Figure 10 SignalTap Logic Analyzer Results of Synchronous Down Counter (Schematic)

## CurrConverter.c Requirements Document

### Abstract

This report includes the requirements document for the CurrConverter.c C program. This requirements document includes an introduction, as well as the requirements of the inputs, outputs, and major functions of the program.

### Introduction

The final C program is contained in one c file, CurrConverter.c. This program is required to convert from dollars to a foreign currency, as well as convert from a foreign currency to dollars. The methods used in this program require C programming tools like the standard library functions, loops, and more.

**Requirements**

CurrConverter.c satisfies the following requirements: prompt the user for whether they would like to convert to or from foreign currency, which currency, and the amount they would like to exchange. Additionally, options are printed at the user's request. The user may also use to quit the program by entering "Q" when prompted.

**Inputs to Program**

This program takes the following inputs: whether the user would like to exchange from USD to a foreign currency or vice versa, a desired currency, and an amount to exchange. There is also an option to define a custom currency value.

**Outputs to Program**

CurrConverter.c outputs the an exchange rate and also a final currency converstion based upon its inputs. Relevant information is printed to the console for the user.

**Major Functions**

CurrConverter.c includes the standard input and output library to access basic input and output functions. The main program is relatively abstracted, such that implementation details are hidden from the main function, and the behavior of the program is then captured in the program's functions.

The printOptions function prints out available currency options, and also prompts if the user would like a custom currency. The USDtoF and FtoUSD functions provide a USD to foreign currency conversion and a foreign currency to USD conversion, respectively. The findCurrency function matches the user-defined currency with an appropriate currency and its value.

**Verilog Code**

```
// DE1_SoC.v
// EE 371 Lab 1 Project
//		DE1_SoC is the highest level module of this project, and connects
// board peripherals, I/O, and more
//		Authors: Nikhil Grover, Emraj Sidhu, Andrique Liu
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR,
SW);
	input 		CLOCK_50; //50MHz clock
	output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
	output [9:0] LEDR;
	input  [3:0] KEY;
	input  [9:0] SW;

	// Generate clk off of CLOCK_50, whichClock picks rate
	wire [31:0] clk;
	parameter whichClock = 24;
	clock_divider cdiv (CLOCK_50, clk);
```

```
	// Hook up FSM inputs and outputs
	wire reset;
	wire [3:0] out1, out2;
	assign reset = SW[0];		// Reset when SW[0] is switched on

	// Note: CLOCK_50 is used for signal tap, but divided clock
	// with whichClock = 24 is used for live demo
	RippleDown Counter1 (.qOut(out2), .clk(CLOCK_50), .rst(reset));	// Ripple down
counter
//	SyncDown Counter2 (.qOut(out2), .clk(CLOCK_50), .rst(reset));	// Sync down
counter... fix
//	JohnsonCounter Counter3 (.out(out2), .clk(CLOCK_50), .rst(reset));	// Sync Johnson
counter
//	DownCounter_Schematic Counter4 (.Clk(CLOCK_50), .Output(out2), .Reset(reset));

	// Show signals on LEDRs
	// Eight LEDs are enabled during demo to for faster demo
	// Four LEDs are enabled during signal tap to test individual counters
	assign LEDR [7:4] = out1;
	assign LEDR [3:0] = out2;

endmodule

// clock_divider is used to divide the clock, slowing down the main clock
// down so that the LEDs are visible.
// divided_clocks[0] = 25MHz, [1] = 12.5MHz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75 Hz, ...
module clock_divider (clock, divided_clocks);
	input 				clock;
	output reg [31:0]	divided_clocks;

	initial
		divided_clocks <= 32'b0;

	always@(posedge clock)
		divided_clocks <= divided_clocks + 1;
endmodule

// JohnsonCounter.v
// EE 371 Lab 1 Project
//	JohnsonCounter controls a counter operating in a Johnson pattern
//	Authors: Nikhil Grover, Emraj Sidhu, Andrique Liu
module JohnsonCounter(out,clk, rst);
	input clk, rst;
	output[3:0] out;
```

```
reg [3:0] out;

parameter state0 = 4'b1111;
parameter state1 = 4'b0111;
parameter state2 = 4'b0011;
parameter state3 = 4'b0001;
parameter state4 = 4'b0000;
parameter state5 = 4'b1000;
parameter state6 = 4'b1100;
parameter state7 = 4'b1110;

always @(posedge clk)
begin
if (~rst)
begin
out = state0;
end

else
begin
case(out)

state0:
out = 4'b0111;
state1:
out = 4'b0011;
state2:
out = 4'b0001;
state3:
out = 4'b0000;
state4:
out = 4'b1000;
state5:
out = 4'b1100;
state6:
out = 4'b1110;
state7:
out = 4'b1111;
endcase
end

end

endmodule
```

```
// RippleDown.v
// EE 371 Lab 1 Project
//	RippleDown controls a counter operating in a ripple down pattern
//	Authors: Nikhil Grover, Emraj Sidhu, Andrique Liu
module RippleDown(qOut, clk, rst);

	input clk, rst;
	output [3:0] qOut;
	wire [3:0] Qbar;

	DFlipFlop ff0 (.q(qOut[0]), .qBar(Qbar[0]), .D(Qbar[0]), .clk(clk), .rst(rst));
	DFlipFlop ff1 (.q(qOut[1]), .qBar(Qbar[1]), .D(Qbar[1]), .clk(qOut[0]), .rst(rst));
	DFlipFlop ff2 (.q(qOut[2]), .qBar(Qbar[2]), .D(Qbar[2]), .clk(qOut[1]), .rst(rst));
	DFlipFlop ff3 (.q(qOut[3]), .qBar(Qbar[3]), .D(Qbar[3]), .clk(qOut[2]), .rst(rst));

endmodule

// SyncDown.v
// EE 371 Lab 1 Project
//	SyncDown controls a counter operating in a synchronous down pattern
//	Authors: Nikhil Grover, Emraj Sidhu, Andrique Liu
module SyncDown(qOut, clk, rst);

	input clk, rst;
	output [3:0] qOut;
	wire d0, d1, d2, d3;
	wire [3:0] Qbar;

	assign d0 = Qbar[0];
	assign d1 = ~(qOut[1] ^ qOut[0]);
	assign d2 = ~(qOut[2] ^ (qOut[1] | qOut[0]));
	assign d3 = ~(qOut[3] ^ (qOut[2] | qOut[1] | qOut[0]));

	DFlipFlop ff0 (.q(qOut[0]), .qBar(Qbar[0]), .D(d0), .clk(clk), .rst(rst));
	DFlipFlop ff1 (.q(qOut[1]), .qBar(Qbar[1]), .D(d1), .clk(clk), .rst(rst));
	DFlipFlop ff2 (.q(qOut[2]), .qBar(Qbar[2]), .D(d2), .clk(clk), .rst(rst));
	DFlipFlop ff3 (.q(qOut[3]), .qBar(Qbar[3]), .D(d3), .clk(clk), .rst(rst));

endmodule

// DFlipFlop acts as a DFF
// This module is sourced from class code files
module DFlipFlop(q, qBar, D, clk, rst);
	input D, clk, rst;
	output q, qBar;
```

```
	reg q;
	not n1 (qBar, q);
	always@ (negedge rst or posedge clk)
		begin
		if(!rst)
		q = 0;
		else
	q = D;
	end
endmodule

// Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, the Altera Quartus II License Agreement,
// the Altera MegaCore Function License Agreement, or other
// applicable license agreement, including, without limitation,
// that your use is for the sole purpose of programming logic
// devices manufactured by Altera and sold by Altera or its
// authorized distributors.  Please refer to the applicable
// agreement for further details.

// PROGRAM		"Quartus II 64-Bit"
// VERSION		"Version 14.0.0 Build 200 06/17/2014 SJ Web Edition"
// CREATED		"Thu Jan 19 21:22:56 2017"

module DownCounter_Schematic(
	Clk,
	Reset,
	Output
);


input wire	Clk;
input wire	Reset;
output reg	[3:0] Output;

wire	SYNTHESIZED_WIRE_0;
wire	SYNTHESIZED_WIRE_1;
reg	SYNTHESIZED_WIRE_17;
wire	SYNTHESIZED_WIRE_2;
```

```
wire    SYNTHESIZED_WIRE_3;
wire    SYNTHESIZED_WIRE_4;
wire    SYNTHESIZED_WIRE_18;
wire    SYNTHESIZED_WIRE_19;
wire    SYNTHESIZED_WIRE_20;
wire    SYNTHESIZED_WIRE_8;
reg     SYNTHESIZED_WIRE_21;
reg     SYNTHESIZED_WIRE_22;
reg     SYNTHESIZED_WIRE_23;
wire    SYNTHESIZED_WIRE_12;
wire    SYNTHESIZED_WIRE_13;
wire    SYNTHESIZED_WIRE_14;
wire    SYNTHESIZED_WIRE_15;




always@(posedge Clk or negedge Reset)
begin
if (!Reset)
        begin
        SYNTHESIZED_WIRE_17 <= 0;
        end
else
        begin
        SYNTHESIZED_WIRE_17 <= SYNTHESIZED_WIRE_0;
        end
end


always@(posedge Clk or negedge Reset)
begin
if (!Reset)
        begin
        SYNTHESIZED_WIRE_23 <= 0;
        end
else
        begin
        SYNTHESIZED_WIRE_23 <= SYNTHESIZED_WIRE_1;
        end
end

assign  SYNTHESIZED_WIRE_4 = SYNTHESIZED_WIRE_17 & SYNTHESIZED_WIRE_2;
```

```
assign  SYNTHESIZED_WIRE_0 = SYNTHESIZED_WIRE_3 | SYNTHESIZED_WIRE_4;

assign  SYNTHESIZED_WIRE_3 = SYNTHESIZED_WIRE_18 & SYNTHESIZED_WIRE_19
& SYNTHESIZED_WIRE_20 & SYNTHESIZED_WIRE_8;

assign  SYNTHESIZED_WIRE_12 = SYNTHESIZED_WIRE_21 |
SYNTHESIZED_WIRE_22;

assign  SYNTHESIZED_WIRE_13 = SYNTHESIZED_WIRE_18 &
SYNTHESIZED_WIRE_19 & SYNTHESIZED_WIRE_20;

assign  SYNTHESIZED_WIRE_14 = SYNTHESIZED_WIRE_23 &
SYNTHESIZED_WIRE_12;

assign  SYNTHESIZED_WIRE_1 = SYNTHESIZED_WIRE_13 | SYNTHESIZED_WIRE_14;

assign  SYNTHESIZED_WIRE_15 = SYNTHESIZED_WIRE_22 ~^
SYNTHESIZED_WIRE_21;


always@(posedge Clk or negedge Reset)
begin
if (!Reset)
        begin
        SYNTHESIZED_WIRE_21 <= 0;
        end
else
        begin
        SYNTHESIZED_WIRE_21 <= SYNTHESIZED_WIRE_15;
        end
end


always@(posedge Clk or negedge Reset)
begin
if (!Reset)
        begin
        SYNTHESIZED_WIRE_22 <= 0;
        end
else
        begin
        SYNTHESIZED_WIRE_22 <= SYNTHESIZED_WIRE_18;
        end
end
```

```
assign  SYNTHESIZED_WIRE_18 =  ~SYNTHESIZED_WIRE_22;

assign  SYNTHESIZED_WIRE_20 =  ~SYNTHESIZED_WIRE_23;

assign  SYNTHESIZED_WIRE_19 =  ~SYNTHESIZED_WIRE_21;

assign  SYNTHESIZED_WIRE_8 =  ~SYNTHESIZED_WIRE_17;

assign  SYNTHESIZED_WIRE_2 = SYNTHESIZED_WIRE_21 | SYNTHESIZED_WIRE_23 |
SYNTHESIZED_WIRE_22;


endmodule
```

**C Code**

```
// Currency Conversion Module
// EE 371, Lab 1
//
// Note: Currency values are current as of Jan. 6 2017 on
// the currency authority www.xe.com
//
// This program prompts the user for desired foreign currency,
// printing exchange rates and calculating foreign currency.

#include <stdio.h>  // Standard I/O
#include <string.h> // For string operations (copy, compare, etc.)

// Forward declarations
void printOptions(void);
void USDtoF(void);
void FtoUSD(void);
double findCurrency(char uinput, char currency[]);

char uinput;   // User input
double amount; // User-defined amount to exchange

int main()
{
   printf("Convert from USD to foreign (1) or foreign to USD (2)?\n");
   printf("Press 'Q' to quit\n");
   scanf("%c", &uinput);
   getchar();
   if (uinput == '1') {
      USDtoF();
```

```
    } else if (uinput == '2') {
        FtoUSD();
    } else if (uinput == 'Q') {
        printf("Quitting program...\n");
        return 0;
    } else {
        printf("Invalid input\n");
        return -1; // Invalid
    }
    return 0;
}

// printOptions prints out available currency options
// This function also allows for custom currency input
void printOptions(void)
{
    printf("Euro (E)\n");
    printf("British Pound (G)\n");
    printf("Pakistani Rupee (P)\n");
    printf("Canadian Dollar (C)\n");
    printf("Australian Dollar (A)\n");
    printf("Custom currency ($)\n");
}

// USDtoF converts USD currency to a foreign currency rate
// Outputs to console
void USDtoF(void)
{
    char currency[3];
    double foreignVal;

    while (uinput != 'Q') {
        printf("Enter foreign currency\n");
        printf("0 for options, Q to quit program\n");
        scanf("%c", &uinput);
        getchar();

        if (uinput == '0') {
            printOptions();
        } else if (uinput == 'Q') {
            printf("Quitting program...\n");
        } else {
            foreignVal = findCurrency(uinput, currency);
        }
        // If there is a valid currency input
```

```
        if (strcmp(currency,"") > 0) {
            printf("%.2lf %s per 1.00 USD\n", foreignVal, currency);
            printf("Enter amount to exchange:\n");
            scanf("%lf", &amount);
            getchar();
            printf("%.2lf %s per %.2lf USD\n", (foreignVal * amount), currency, amount);
        }
        strcpy(currency,"");
    }
}

// FtoUSD converts a foreign currency rate to USD currency
// Outputs to console
void FtoUSD(void)
{
    char currency[3];
    double foreignVal;
    double finalVal;

    while (uinput != 'Q') {
        printf("Enter foreign currency\n");
        printf("0 for options, Q to quit program\n");
        scanf("%c", &uinput);
        getchar();

        if (uinput == '0') {
            printOptions();
        } else if (uinput == 'Q') {
            printf("Quitting program...\n");
        } else {
            foreignVal = findCurrency(uinput, currency);
        }
        // If there is a valid currency input
        if (strcmp(currency,"") > 0) {
            finalVal = 1.00 / foreignVal;
            printf("%.2lf USD per 1.00 %s\n", finalVal, currency);
            printf("Enter amount to exchange:\n");
            scanf("%lf", &amount);
            getchar();
            printf("%.2lf USD per %.2lf %s\n", (finalVal * amount), amount, currency);
        }
        strcpy(currency,"");
    }
}
```

```
// findCurrency finds the appropriate currency and returns the
// desired currency value
// findCurrency takes in user input and an array of chars, currency
// this array is modified to reflect the appropriate currency
double findCurrency(char uinput, char currency[])
{
   double foreignVal;

   switch (uinput) {
      case 'E':
      strcpy(currency, "EUR");
      foreignVal = 0.949420;
      break;
      case 'G':
      strcpy(currency, "GPB");
      foreignVal = 0.813981;
      break;
      case 'P':
      strcpy(currency, "PKR");
      foreignVal = 104.800;
      break;
      case 'C':
      strcpy(currency, "CAD");
      foreignVal = 1.32403;
      break;
      case 'A':
      strcpy(currency, "AUD");
      foreignVal = 1.37030;
      break;
      case '$':
      strcpy(currency, "CTM");
      printf("Enter custom currency value: ");
      scanf("%lf", &foreignVal);
      // scanf needs to read in a double pointer type
      getchar();
      break;
      default:
      printf("Invalid input\n");
      break;
   }
   return foreignVal;
}
```

**CONTRIBUTION**

Andrique Liu

- C program
- SignalTap

Nikhil Grover

- Counter Design
- iVerilog implementation and testing

Emraj Sidhu

- SignalTap
- Synchronous Down Counter schematic

**SIGNATURES**

Andrique Liu

Emraj Sidhu

Nikhil Grover