EE 371 Project 2

Designing a Simple System, Timing Analysis, and Working with the C Language

University of Washington - Department of Electrical Engineering

James K. Peckol

Introduction:

In this project, we will specify and design a lock system …no, not that kind of lock …..between canals of different levels in the city of olde Venice, implement and model the design in Verilog, then test the model using the iVerilog environment. We will then synthesize and port the design to the Cyclone V FPGA where we will test and verify its operation using the DE1-SoC board switches and LEDs to model the I/O, control logic, and display system status.

Next, we will continue to grow our knowledge of the C language by working with variables and their addresses.

Prerequisites:

Familiarity with the Quartus II development environment and the *Signal Tap* (still ain't *Spinal Tap*) logic analyzer. A continued willingness to learn and to explore. No beer until the project is completed and you can turn on a several LEDs. Hey, these are all still good.

Cautions and Warnings:

Never take a deipnosophist and a mangle wurtzel to a fancy French restaurant….unless, of course, you are going to have agneau. Never have a philosophical debate with a curmudgeonly mangle wurtzel that can bench press more that you. Don't try to play tetras with LED sheep (check it out).

Never disconnect the power supply wires from your circuit and leave them lying on your bench. The current will drain out of the power supply and the supply will no longer be usable…. Unless, of course, you can manage to clean up the current and somehow put it back in. You may need to get a current pump from Bill in stores to help with the cleanup.

If you leave the current laying on your bench, it will no longer be current, but, passed.

Background:

For this project, you should understand and know how to:

1. Start from a textual description of the behaviour of a system,
2. Extract the behavioural requirements from the textual description,
3. Express the requirements in a state diagram,
4. Utilize the Verilog HDL to formulate a physical model of the state diagram as a finite state machine (FSM) taking into consideration real-world constraints,
5. Use an FSM for controlling the behavior of a digital system,
6. Use registers and to enter and retrieve data from them.
7. Work with peripheral I/O devices such as switches and LEDs.

8. Test the model to ensure concurrence with the specified requirements,
9. Synthesize the model and program it onto an FPGA
10. Test the physical implementation to ensure concurrence with the specified requirements.
11. Be comfortable working with and utilizing the Signal Tap logic analyzer.

Objectives:

The major objectives of this project include:

- Use knowledge and understanding of FSMs and HDL modeling to execute the design and test of a basic real-world system.
- Continue to work with simple I/O and selection and routing logic to model the source and destination of signals into and out of a digital system.
- Learn and work with the basic concepts of a time base and intra system timing and timing constraints.
- Learn and work with basic C variables and their addresses.

Introduction to the Development Cycle

The formal development cycle comprises a number of steps as we move from an idea to the physical expression of that idea in hardware, software, or both. We start from a glint in the eye as we endeavour to capture *what* the desired entity really is.

It's important to remember, the development cycle is not a one-time process. Often we will iterate through several times (NOTE: this does not say keep guessing) until we are satisfied with the quality and reliability of our design and have ensured that it meets all of our specifications.

Requirements Definition

We begin the development cycle by trying to capture an understanding what we are supposed to be doing…more specifically, identifying the needs of all interested parties. We then document these needs as written definitions and descriptions. It is important, at this stage, that we focus is on *what* problem the system is intended to solve not *how*. The *how* part comes later.

Our goal with the *requirements definition* step is to capture and express a purely external view of the system. We refer to that view as the *public interface* of the system – sounds just like object centered design…and is. We identify *what* needs to be done starting from the user's requirements.

Establishing the Design Specifications

After we have identified the requirements, we move from a qualitative description as we formalize those requirements into a formal, hard, verifiable, quantified specification that we can design and test to. As we do so, we are still in the *what* phase of the development cycle,

Our goal with the *design specification* step is to capture and express, in a formal way, the specific values, tolerances, timing, protocols, intent, and constraints on every signal coming into or leaving the system.

Formulating a Functional Decomposition

As we noted, the system requirements and specification documentation says *what* we are to be developing. The *functional decomposition* step now begins to establish the *how* of the design. We now express the design in the designer’s language and from the designer’s point of view.

In formulating a functional decomposition, we do not consider schematics, code, or parts lists except under exceptional and limited circumstances.

Our goal is to try to identify the major pieces of functionality, divide these into smaller and more detailed pieces, and ultimately find an appropriate internal architecture for the system that explains the *how* the requirements are implemented. It is important that the decomposition be technology-independent.

From the functional system description, which is based on a functional structure and the behavior of each function, we then decide what hardware, software, or combination thereof we will use to implement each piece of functionality.

The Problem

A system is needed to manage and control the movement of gondolas and other boats along the canals and rivers in and around the olde city of Venice that may be at different levels. The system must support the following functionality for arriving or departing boats.

<u>The Lock</u>

The lock system in common use today is called a *pound lock* – not because of the amount of the toll in British locks. The lock comprises a chamber in which the water level can be raised or lowered as necessary and bidirectional gates on either end that can be opened to let a boat enter or leave or closed to allow the water level in the pound to be raised or lowered.

A gate can only be opened if the water level difference across the gate to be opened (arriving or departing) is less than 0.3 feet. The water level in the pound can be raised or lowered by up to 5 feet.

<u>On Arrival</u>

A gondola must signal the lock 5 minutes before arrival.

If the pound is empty, the signal from the gondola should cause inner and outer gates of the lock to be closed and sealed.

After the gates are closed and sealed, the water level difference between the pound interior and the level of the canal from which the boat is arriving shall be managed and controlled to less than 0.3 feet.

When the water level difference between the interior and exterior of the pound reaches less than 0.3 feet, the arrival gate shall be opened. Managing the water level in the pound takes up to 7 minutes to lower the level and 8 minutes to raise it.

When the outer gate is fully opened, gondola can enter the pound. When the gondola is fully inside the pound, the arrival gate is closed and sealed.

On Departure

After the gates are closed and sealed, the water level difference between the pound interior and the departure gate shall be managed and controlled to less than 0.3 feet.

When the water level difference between the pound and the exterior of the departure gate reaches less than 0.3 feet, the departure gate shall be opened. Managing the water level in the pound takes up to 7 minutes to lower the level and 8 minutes to raise it.

When the departure gate is opened, the gondola can exit the lock.

Developing the Specification

Based upon the description of the required behaviour of the lock system given above, produce the following:

1. A short *Requirements Document* for the lock system that will capture those requirements in a formal document. The document should have the following sections:
   - ✓ Abstract
   - ✓ Introduction
   - ✓ Inputs to the system
   - ✓ Outputs from the system
   - ✓ Major Functions

   The each input and output should have a brief textual description of what it is and its purpose.

   Each major function should have a brief textual description of what it is and its behavior. In this context, the major functions specify the behavior seen by the user.

   The purpose of the Requirements Document is to capture and document *what* the customer wants the system to do, not how. It's totally an outside view of the system.
2. A *Design Specification* for the lock system that will capture the requirements in a formal document. This document is different from the previous even though, it looks much the same. The organization follows the same general flow with similar sections; the content of those sections is slightly different and more detailed.

   While our system requirements have not changed, the information that we require will. We now need to specify those changes. The document should have the following sections:
   - ✓ Abstract
   - ✓ Introduction
   - ✓ Inputs to the system
   - ✓ Outputs from the system
   - ✓ Major Functions

Yes, they look the same as those for the Requirements Document. Now, however, each input and output must now also include as part of its description the range of legal values and a brief textual description of what possible errors might occur and their consequences. In this context, the major functions specify the behavior seen by the user.

3. A *Functional Decomposition* for the lock. Identify and briefly describe the top-level functions for the design and then give a functional diagram. In this context, the functions refer to those seen by the engineer. These comprise those seen by the user and those seen solely by the engineers.
4. Based upon your specification and the functions that you have defined, give a high level block diagram for the design.
5. Implement your design model at the behavioural level in iVerilog, design a test suite and test bench.
6. Test the lock control system to ensure that it is functioning properly.
7. Synthesize the HDL implementation of the lock control system, download it to the DE1-SoC board, and verify its operation there using signal tap.

   Use the following switches, keys, and LEDs to model the system inputs and outputs.

   SW0, LED0 – Gondola Arriving
   SW1, LED1 – Gondola Departing
   SW2, LED2 – Open and Close outer port
   SW3, LED3 – Open and Close inner port
   Key 0 – Reset
   Key 1 – Increase water level
   Key 2 – Decrease water level

Learning the C Language – The Next Steps:

We will conclude this project by introducing and working with pointer variables in the C language.

Working with C in the PC Environment and Some of the Tools

Part 1 – Getting to Know Pointers

In a language such as C, when we make a *declaration* or more properly a *definition* such as:

```
int x = 9;
```

we are taking, in this case, and integer and putting (or storing) it in memory. The location where we store it has an address.

In C or C++, we can find out what that address is by using the address of operator '&'. If we write:

```
&x;
```

We can get the address of x in memory. If we assign that address to some other variable, we can use it at some later time to retrieve the value of the variable at that memory address or location. The variable to which we assign the address *points* to that specific location in memory. Hence, very cleverly, we call such a variable a *pointer*.

When we use a pointer variable, we distinguish it from another type of variable by writing it with a '*' preceding it.

Thus,

```
int* xPtr;
```

is a variable that is used to contain or hold a memory address. Specifically, if we read each portion of the expression above from right to left we can see exactly what is going on. Let's go, from the right hand side, the variable called *xPtr* (the identifier thingy) is a *pointer* (the * thingy) to an *integer* (the int or type thingy).

We could also write,

```
float* yPtr;
```

Once again, if we read from right to left, the variable called *yPtr* (the identifier duder) is a *pointer* (the * duder again) to a *float* (the float or type duder).

Each such pointer variable above gets assigned the address of the thing that it points to by taking the address of the thing to be referred (using the address of operator, &) to then assigning the address value to the pointer. Gees, what did you just say? That's confusing. Let's take another look at this.

Ok, let's assume that we first declare a variable. In the first case, it's an int called x and in the second a float called y.

```
int x = 3;          //  declare a variable of type int called x and initialize it to 3
```

or

```
float y = 3.2;      //  declare a variable of type float called y and initialize it to 3.2
```

Now, let’s declare some pointer variables. The first is a pointer to an int called xPtr and the second a pointer to a float called yPtr. Observe that in selecting the pointer names, we add the suffix Ptr to distinguish the variable as a pointer. It just helps to make our code more readable. We could have called it george if we had wanted to, but which name provides more information?

```
int* xPtr ;             //  declare a variable of type pointer to integer
```
or
```
float* yPtr;            //  declare a variable of type pointer to float
```

Finally, let’s make the two pointers point to the two variables, x and y. We’ll use the address of operator to find the addresses of the two variables then make the assignments

```
xPtr = &x;              //  then assign the address of x to it
```
or
```
yPtr = &y;              //  then assign the address of y to it
```

If we wish to use the value at the memory address referred to by the pointer variable, then we perform an operation called *dereferencing*. That proceeds as follows…

```
int z = *xPtr;          //  go to the address in memory identified by the variable xPtr
                        //  get the value at that address and assign it to the variable z
```
or
```
float w = *yPtr;        //  go to the address in memory identified by the variable yPtr
                        //  get the value at that address and assign it to the variable w
```

What do you think would happen if we wrote the following instead…

```
int z = *yPtr;          //  go to the address in memory identified by the variable yPtr
                        //  get the value at that address and assign it to the variable z
```
or
```
float w = *xPtr;        //  go to the address in memory identified by the variable xPtr
                        //  get the value at that address and assign it to the variable w
```

As our first step in working with pointers, let’s just explore.

1. Declare several variables of the following types: two variables of type int, two variables of type float, and two variables of type char.
2. Declare the following pointer type variables: one pointer to an int, one pointer to a float, and one pointer to a char.
3. Assign the address of one of the integer variables to the pointer to int. Print out the value of that integer.
4. Repeat with the second integer.
5. Repeat steps 3 and 4 with the two floats and then the two chars.

Part 2 – Working with Pointer Variables

Now that we have a little experience with variables and addresses in C, let's put that knowledge to work.

Let's repeat the problem that we solved as the last part of an earlier project. This time, let's work at the C level and with pointers.

Once again, as a first step, declare the following variables of type integer. Initialize each to the values indicated.

A = 22
B = 17
C = 6
D = 4
E = 9

Declare one more variable, *result*, of type integer.

Next, declare and define five variables of type pointer to integer and let each refer to one of the variables.

Finally, perform the computation:

result = ((A – B)*(C+D))/E

only instead of using the variables directly, refer to each through its pointer. Of note, the * above is the multiplier operator, not a pointer thingy.

Deliverables:

A lab demo showing…

1. The Lock Control System design and working implementation that meets the specified requirements.
2. A working C program that meets the specified requirements.

A lab report containing

1. The annotated Verilog and C source code for all applications both on the DE1-SoC board and on the PC.
2. Representative screen shots showing the results of testing the various designs.
3. Representative screen shots showing the Signal Tap logic analyzer outputs for the various designs.
4. Your Requirements Document, Design Specification, and Functional Decomposition for your Lock Control System design. Put these in an Appendix.
5. Answers to any questions above.
6. Other things that you deem to be important.
7. Anything that we haven't thought of.