University of Minnesota
School of Physics and Astronomy

Physics 4051 Lab Manual

Fall 2009

Table of Contents

Table of Contents

Preface

Introduction

This manual consists of 14 individual chapters, each representing about one week of laboratory work. Although we have tried to spread the workload equally among these chapters, you will find that some chapters require more work than others. For example, chapters 2 and 4 require considerably more work than any other chapter; it is to your advantage to study the material in these chapters ahead of time.

Before doing the exercises in the manual, you should do the assigned reading in Horowitz and Hill or Diefenderfer and Holton or Polnus. Try to predict the behavior of a circuit before building it. These exercises are designed to accompany the material covered in class lectures.

Troubleshooting

The exercises outlined in this manual require a three step process: first, building the circuit, second testing, and finally, troubleshooting or fixing the circuit. Steps one and two are always required and you will quickly discover that step three is required most of the time. You will also learn that you will spend most of your lab time on troubleshooting. Do not regard trouble shooting as a waste of time. It will help you obtain a more complete understanding of the circuit.

Here are some suggestions on how to make step three as efficient as possible.

Understand the Circuit

It is almost impossible, and to say the least -- frustrating, to fix something that you do not understand. Instead of spending an endless amount of time exchanging components and checking wires, go back to your textbook (or TA) and make sure you understand the material. Sometimes the circuit does work, yet since the student does not understand what the circuit is supposed to do he or she may spend needless time and effort modifying it.

Check the Wiring

It is a good idea for one person to build the circuit and for the lab partner to check the wiring. Some tips on wiring:

-- color code the wires (RED for positive supply voltages, GREEN for ground and BLACK for negative supply voltages).

-- use the correct type of test leads; it greatly reduces noise and errors.

-- check the ground wires; no ground wires should be left hanging or unattached.

-- use as few wires as possible; beginners tend to use far too many connecting wires.

Check the Components

Sometimes components get mixed up in the bins. This is particularly true for resistors and capacitors as they are often returned to the wrong bin. Make sure you have the correct component. If it does not seem to work, try another one. After you have tried three of them, you can be certain that it must be something else in your circuit that is causing the problem. If you find faulty components, do not return them to the bins; throw them in the "dead-components" shoebox or trash.

Check the Supply Voltages

Use a scope or a voltmeter to check the supply voltages. Check these voltages at the point where they are connected to your circuit. Often you will find that for one reason or another you forgot to power your circuit.

Check the Voltages inside the Circuit

After verifying that power is indeed applied to the circuit follow the current path and measure voltages at a few easy to calculate points.

Typos

While we can not claim that there are no mistakes or errors in this manual, there are definitely no deliberate errors in this manual. If we find any mistakes or errors, we will post them on the blackboard in room 65. Also, the circuits in this manual have been tested and in use for the last 10 years and they do work!

Rules of Thumb

Throughout this manual you will find various "rules of thumb." They are approximations and help you remember what you should look for first when dealing with a particular component or instrument. You should completely understand and memorize these rules and you should also understand the limitations to them.

Acknowledgments

This lab manual is based on the first edition of the lab manual by Horowitz & Robinson. Many of the exercises were copied from that manual and adapted to meet our specific needs. Some of the digital exercises were developed by Prof. Zimmermann and Prof. Shupe here at the University of Minnesota. Prof. Ruddick contributed some of the exercises and also spent considerable time and effort making the manual readable. Thanks to Prof. Ganz, Prof. Rusack, Prof. Weyhmann, James Flaten, Michael Krueger, Jens Henrickson and Yaroslav Lutsyshyn for their suggestions and proofreading and to Marty Stevens, Jon Huber, Andrew Stewart and Kienan Trandem for some of the drawings.

Kurt Wick
Summer 1999 / 2009

Chapter 1

Simple DC & AC Circuits

Write-up Format: Short

Reading:

(This) Lab Manual: Appendix A and F
Horowitz&Hill: Sections 1.01 - 1.05
Horowitz&Hill: Sections 1.06 - 1.11
Horowitz&Hill: Appendix A and C

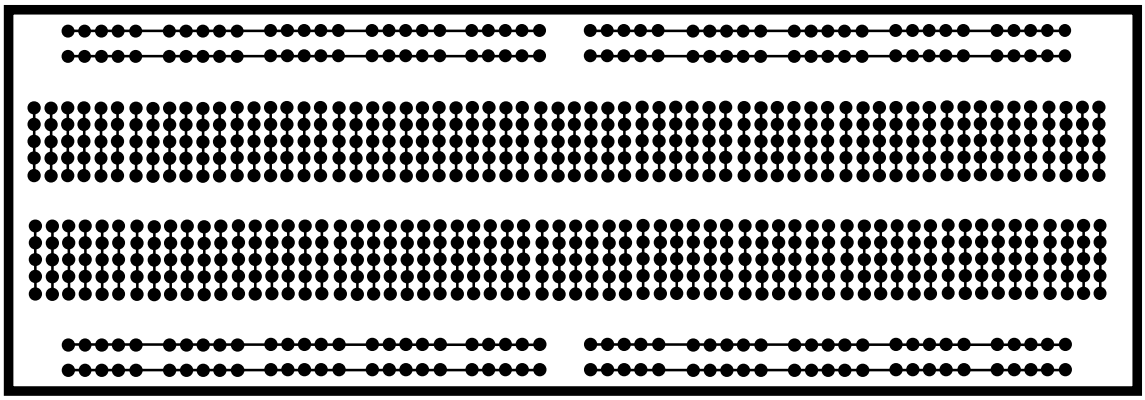
or:

Diefenderfer&Holton: Sections 1-1 - 1.13 (pages 1 to 19)
Diefenderfer&Holton: Sections 6-6 - 6-7 (pages 111 to 115)
Plonus: Sections 1.3 – 1.7 (pages 2 to 37)

Introduction Hardware:

This first chapter introduces you to the lab and its equipment. In the first lab session, you and your lab partner will obtain a "bread board" from your TA. For the rest of this quarter, you will "plug" various components into the breadboard to assemble and to test your circuits. Therefore, you will keep your breadboard until the end of the semester; at the end of each lab session, store it in one of the drawers in lab and clearly mark the label on the drawer with your names.

Study the diagram below and make sure you understand how the wires are connected inside the breadboard!



- Figure 1.1. Top-view of a "breadboard" and its internal connections.

On these breadboards, the two outer strips are usually used for ground and supply voltages. Note, these outer strips are disconnected at the center of the board.

To assemble and to connect your circuits you will need wires, cables, test clips and various connectors. Have your TA show you where to find various cables and how to use them.

Chapter 13

FPGA & VERILOG Part I

Write-up Format: Short

Reading:

This chapter is (mostly) self contained. It contains a lot of information about the Verilog language. You should definitely read it before coming to the lab.

- A short introduction to the Verilog language by Prof. Mans can be found in Appendix K.
- Information about the Xilinx ISE Webpack and the Digilent Export facility is given in Appendix J.
- Finally, a detailed description of the BASYS board (including pin-outs) is given in Appendix L.

In this and the following chapter we will return to digital logic circuits. No new digital logic concepts will be introduced; instead, we will revisit some of the ideas from chapters 8 and 9. This time however, instead of using simple logic gates connected with wires, we apply the concepts in a new, more powerful way. The tools that we will use are:

a) a programmable logic board by Digilent Inc. called BASYS which uses a Field Programmable Gate Array (FPGA);
b) a hardware description (programming) language (HDL) called Verilog.

With these two tools we are able to turn your knowledge of digital circuits into powerful and (hopefully) useful applications. They will allow you to efficiently implement high speed, large scale digital logic designs that would be far too tedious to build using individual chips!

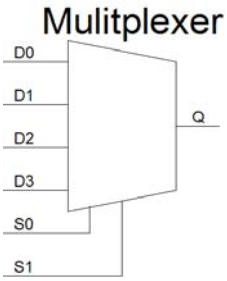
Finally, the reason for putting these exercises at the end of the course, instead of directly after the digital section, is that the syntax of Verilog language is based on the C-language. In other words, we wanted you to familiarize yourself sufficiently with C before having to learn Verilog.

Introduction to the BASYS FPGA Board and Verilog

While it is possible to use either specific gate chips or universal gates such as the NAND, most modern systems have replaced combinations of "fixed logic" chips with programmable logic. The "Basic Systems Board" or **"BASYS"** board which you have used for the frequency counter in chapter 8 is an example of such a programmable logic system. The large chip in the center labeled "Xilinx" is the programmable logic chip called an FPGA (Field-Programmable Gate Array) – as a rough estimate, it can contain up to 100,000 gates-equivalent of logic. Besides the FPGA, most of the rest of the board consists of connectors, displays, switches, and the support machinery to let you program the FPGA from your computer. You can find the full documentation for BASYS in Appendix I.

Inside the FPGA, there are a large number of memory elements or "lookup tables". Each lookup table or LUT has a number of inputs and one output. The input values select an item in a list of

memory elements. It behaves essentially like a multiplexer, familiar to you from section 7.3 and 7.4. The diagram of 4 bit multiplexer and its truth table are shown below.



S1	S0	Q
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Let's consider a simple case with two inputs and one output. If both inputs S0 and S1 are low, then the value in the first memory cell, i.e., input D0, is selected as the output Q. If the S0 is high and S1 is low, the value in the second memory, i.e., input D1, is put on the output. Similarly for S0 low and S1 being high, Q corresponds to D2 and so forth. The lookup table behaves as an arbitrary logic function generator: any logic function of two inputs and one output can be implemented using it. Besides the LUTs, most of the rest of the FPGA consists of registers (Flip-Flops) and multiplexers (MUX) which let us connect external pins to the inputs and outputs of the LUTs.

The FPGA is very flexible – we just have to program it! The program defines the contents of the LUTs and the multiplexer settings. We could do it by manually specifying all the LUTs and mux values by hand, but that would be very slow and prone to error. Instead, we will use a computer-aided design (CAD) process where we describe the behavior we want the FPGA to perform, and then we use a (free) software tool from Xilinx called "Webpack" to convert that into the form needed for the programming of the FPGA. We will use a language called **"Verilog"** to define the behavior of the system. There is a compact guide to Verilog in Appendix K.

In the next section you will learn how to program the BASYS board and how to specify how to get signals into and out of the FPGA. Next you will learn how to implement basic combinational and sequential logic.

13.1. Digital I/O With the BASYS FPGA

Verilog Overview: Modules

The Verilog programming language is based on C language syntax. Similarly to the C-syntax which relies heavily on functions, Verilog uses modules. A very simple Verilog module named MyModule is shown below:

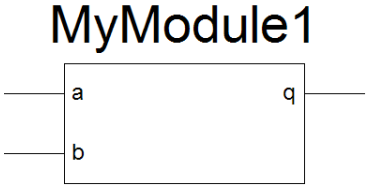
```
module MyModule1(a, b, q);
    input a;
    input b;
    output q;

    assign q = a & b;
endmodule
```

As you can see, the structure of a module is very similar to a C-function. Nevertheless, notice the following similarities / differences:

- There are no curly brackets in Verilog; instead keywords like *endmodule* (or *begin / end*) are used. Note though that semicolons were retained!
- Each module begins with the keyword *module* and ends with the keyword *endmodule*.

- The variables *a, b, q* look like function arguments but they are now called “inputs” or “outputs; they are then further defined below the (module) header.
- Similar to C, once the function and its inputs and outputs have been declared, the statements that will do “something” useful follow them. For right now you do not have to understand them. (In case you are curious, in the above example, *a* and *b* perform a simple AND operation whose result is then assigned to the output *q*.)
- Finally, unlike C, there is no “special” module that needs to be named “main” and which calls all the other modules in your program. Instead, in the Xilinx compiler, you can select any one of your modules and assign it to be the “Top Module” which then functions the same as a “main” module.

For illustrative purposes, a graphical representation of the module MyModule1 is shown below. By convention, inputs are drawn on the left side and the outputs are on the right.



Pin Assignment

A Verilog module is only useful if we are able to communicate with its inputs and outputs. This communication is accomplished by using the appropriate pins on the actual FPGA chip. Before we can make use of them they must be linked, i.e., assigned, to the corresponding inputs and outputs using the *(* LOC = pin number *)* command.

For example, in the code shown below we assign the pin labeled *P69* (which happens to be connected to the rightmost push button labeled BTN0) to some input named *a*:

```
        (* LOC = "P69" *) input a;      //NOTE the LOC statement is enclosed in parentheses!
```

Though we may name our inputs and outputs anything that pleases us, at least as long as it is not a reserved Verilog keyword, how do we find the actual pin numbers that we want them to link to? They are listed in Appendix L on the last page and they can also be obtained directly from DigilentInc’s Website: http://www.digilentinc.com/Data/Products/BASYS/BASYS_E_RM.pdf. Check and see if you can confirm that *P69* indeed is connected to the right most pushbutton labeled BTN0.

(Warning: Note that there is a bug in the DigilentInc documentation concerning all pins with pin numbers less than 10. In the documentation these pins are listed with a leading 0, as, for example, *P06*. For Verilog to compile correctly, they must be entered without the 0, i.e., as *P6*!)

Here are some comments regarding the pins of the FPGA. The chip in the center of the BASYS board, a Spartan 3E, has 144 pins. They are all powered by LVTTL signals, i.e., 0 to 3.3V. Some of its pins have already been assigned by the manufacturer of the BASYS board for a specific task. For example, some pins provide the supply voltage and ground to the FPGA. We cannot and do not need to assign these in our Verilog modules.

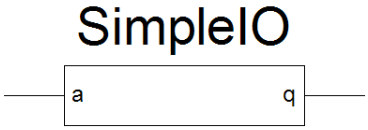
A second group of pins has been connected to various inputs (switches, push buttons and clock signal) and outputs (LEDs, Display, VGA connector.) We will make use of these. Finally, a third group of pins has been connected to connectors that can act either as inputs or outputs; there are 4 six-pin connectors at the top (labeled JA, JB, JC and JD) and pins 1 through 4 can be assigned to be either outputs or inputs. We will use these later.

Exercise 1: Simple I/O Verilog Module

To illustrate the topics covered so far, study the module *SimpleIO* shown below:

```
module SimpleIO(a, q);
        (* LOC = "P69" *) input a;   //use the push button labeled BTN0 at pin P69 as input a
        (* LOC = "P15" *) output q;  //use the LED labeled LD0 at pin P15 for the output q

        assign q = a;           //assign whatever value input a has to output q

endmodule
```

Again, a graphical representation of the module is given below:



The module has one input, *a*, and one output *q*. The input and output have been assigned to the FPGA pins connected to the right most push button labeled BTN0 and the rightmost LED labeled LD0, respectively. Furthermore, the input has been (in the Verilog module through the *LOC* statement) connected to the output so pushing BTN0 will illuminate LD0.

Let's implement this module. For your reference, we have collected the series of steps necessary to create a new project in the programming software in Appendix J. Using the steps in Appendix J.1, create a new project in your personal area called "SimpleIO". Then, use the steps in Appendix J.2 to make a new source file. Modify the generated skeleton source code (including the "module" statement) and add the statements above. Finally use the steps in Appendix J.3 to synthesize your design and download the bit-file to the FPGA using the Digilent ExPort facility. This will be your first assignment. Check that it indeed works and hand in your code.

Write-up

13.1.1. Hand in the Verilog source code needed to turn on LD0 by pushing BTN0. For this and all the subsequent exercises, you must hand in the documented code for all the modules that you used. (Documented means you must add comment statements explaining the purpose of each module and the purpose of the various inputs and outputs as well as the wires and registers within the module.)

13.2. Combinational Logic: 4 Bit Full Adder

In this section you will build a full 4 bit adder. You will learn how to implement combinational logic in Verilog, how to define wires and buses and how to instantiate modules.

Combinational Logic Introduction

The output of a combinational logic circuit depends only on its current input. Therefore, it involves only logic gates that have no memory, i.e., they are amnesic. Typical combinational logic gates are AND, NAND, NOR, OR, NOR and XOR gates. In this exercise you will learn how to use Verilog operators to implement these gates to rebuild your full one bit adder from section 7.3.

Combinational Logic: *assign*-Statement

Combinational logic is implemented in Verilog by using the *assign* keyword and the blocking assignment operator represented by an equal sign. Assignments often use Verilog bit operators. They are identical to the ones used in C and are listed below:

Bit Operator	**Notation**
AND	&
OR	|
NOT	! (or ~)
XOR	^
Is Equal	==
Is Not Equal	!=
Right Shift (by n bits)	>> n
Left Shift (by n bits)	<< n

Table 13.1.

For example, an AND operation between *a* and *b* whose result would be assigned to *q* would be written in Verilog as:

```
assign q = a & b;
```

Similar to C, multiple operations can be combined in one statement like in the example shown below:

```
assign p = (a & b) ^ !c;
```

An if-statement can be implemented in combinational logic as a conditional assignment using the following notation:

```
assign q = c ? a : b;
```

In this case, if *c* is true, *a* will be assigned to *q*; else, *b* will be assigned to *q*. (Note: this is a one bit multiplexer.)

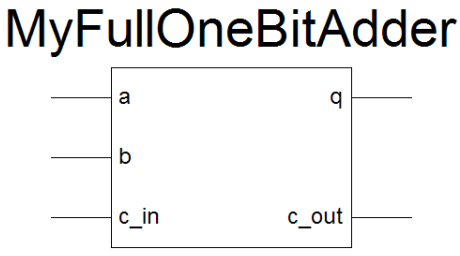
Another form of this construct that you may encounter is shown below:

```
assign q = (a_input == 3) ? x : y;
```

In this case, *x* will be assigned to *q* if *a_input* is 3; else *y* is assigned to *q*.

Exercise 1: One Bit Full Adder

Throughout this section you will be building a very basic calculator that ultimately is capable of adding 4 bit numbers. Though the Verilog syntax contains basic mathematical operators, such as addition, subtraction and multiplication, in this exercise you will not make use of them. Instead you will design the adder circuit using the basic logic gates or bit wise operators described earlier. (In case you wondered, this is also how Verilog implements its mathematical operators but it is done in the background, completely hidden from the programmer.)



Create a new Verilog project with a module called MyFullOneBitAdder. The graphical representation of the module is shown in the previous diagram and the table below lists the relevant inputs and outputs and their functions. In the last column of the table below, fill in the corresponding pin numbers from the DigilientInc documentation in Appendix L. Use this information in your Verilog module to assign the inputs and outputs to the appropriate pins using (* LOC = … *) statements.

Name	Function	BASYS Control	Pin
a	Input	Switch SW0	
b	Input	Switch SW4	
c_in	Carry-in Input	Switch SW7	
q	Output, i.e., a+b+c_in	LED LD0	
c_out	Carry-out Output	LED LD1	

Table 13.2.

Now that you have all the inputs and outputs defined and assigned to the appropriate pins, you need to add the combinational logic statements for the one bit full adder. Find your write-up from section 7.3. Specifically, use the *assign* keyword and convert the logic expression you obtained for 7.3.4., into Verilog syntax using the Verilog bit operators shown in Table 13.1.

Compile your code and download the bit file into the FPGA. Test your one bit adder by operating switches SW0, SW4 and SW7 and check the corresponding LEDs. If everything works, hand in your code.

Write-up

13.2.1. Hand in the Verilog source code for your full one bit adder.

Instantiating a Module

Instantiating a module is similar to C function calls: once you have a (functioning) module you can use it repeatedly in your code by "instantiating" it. For example, in the previous section you built a working one bit full adder. By "instantiating" it multiple times we can extend it easily into a 4 or even 128-bit full adder!

If that still sounds too theoretical, think of your original module as a blueprint; "instantiating" it creates a working "copy" of the blueprint.

To illustrate this concept, let's instantiate the previously built one bit full adder twice, i.e., create a two bit full adder that adds the 2 bit word composed of a_1a_0 to the two bit word composed of b_1b_0. (By convention the index 0 indicates the least significant bit of the word.) The result of the addition is then assigned to the three bit output *c_out* q_1q_0. Such a circuit would be able to add the decimal values, for example, of 3 and 2. See its graphical representation below.

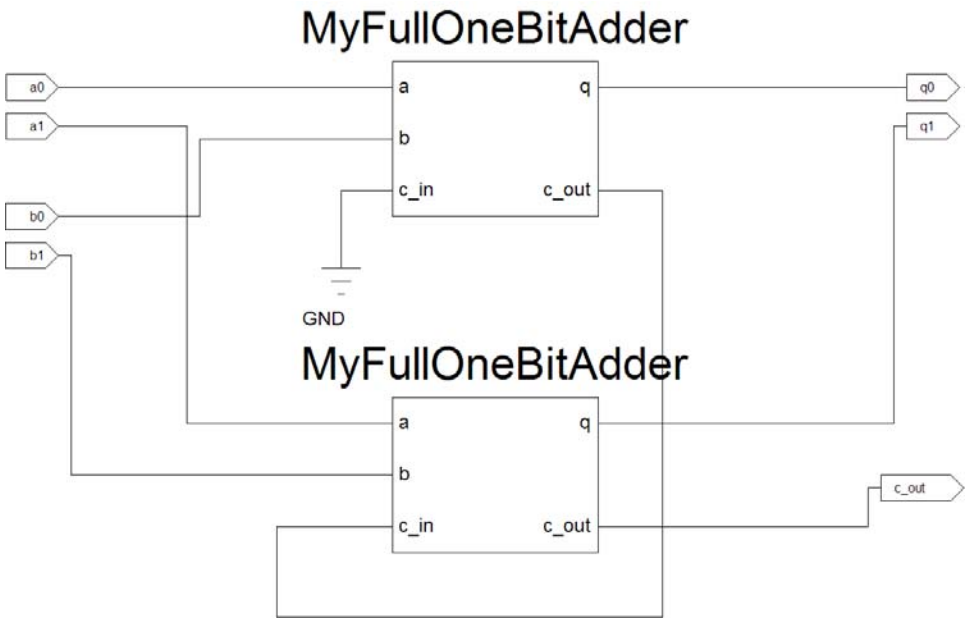


Figure 13.1. Graphical representation of a full 2 bit adder. The input c_in for the least significant bit(top module) has been permanently tied to ground. Why?

The corresponding Verilog statements (minus the pin assignment statements) are:

```
module MyFullTwoBitAdder(a0, a1, b0, b1, q0, q1, c_out);
    input a0, a1, b0, b1;  //Note: all pin assignments were omitted for sake of brevity!
    output q0, q1, c_out;


    //Instantiate the full one bit adders:
    MyFullOneBitAdder FullOneBitAdder0(a0, b0, 0,      q0, c_out0);
    MyFullOneBitAdder FullOneBitAdder1(a1, b1, c_out0, q1, c_out);

endmodule
```

The statements performing the instantiation of the module *MyFullOneBitAdder* are shown in bold print and it uses the following syntax:

```
    OriginalModuleName InstantiatedModuleName(inputs..., outputs...);
```

Similar to C, the names of the inputs and outputs do not have to be identical to the ones used in the original module but their order, number and type must be preserved! Also, inputs and outputs can be implicitly connected by appropriately naming the inputs and outputs. For example, the *c_out* output in the *FullOneBitAdddder0* (instantiation) has been implicitly connected to the *c_in* input of the *FullOneBitAdddder1* (instantiation) through the "variable" *c_out0*. (Note: "wire" would be the proper term here, not "variable;" since we have not yet defined what constitutes a wire in Verilog we call them - at least for now - "variables.")

Also note that the module *MyFullOneBitAdder* was instantiated twice. It could have been instantiated many more times; the only requirement is that each new instantiation has a new, unique name.

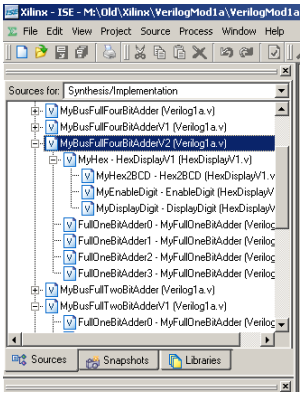
A practical issue: how does the compiler know where to locate the module that is to be instantiated? Two different solutions to tackle this issue exist:

a) You can combine all your modules in a single file, one listed below the other. If you choose this approach then you will have to inform the Xilinx ISE program which one you want to use

as your *Top Module*: in the upper right panel of the Xilinx ISE Program "Sources" window find the module that you want to be the Top Module; right click on it and select "*Set as Top Module*."

b) If you prefer, you can keep each module in a separate file. In this case each file must be added to the project by selecting in the menu bar: *Project / Add Source*. Again, with multiple modules you need to specify which one represents your Top Module. How this is done has been explained in the instructions in the previous paragraph.

Once a module has been instantiated, Xilinx ISE Webpack adds a plus sign to the Top Module in the project Sources window. See the example below.



Clicking on the plus sign lists the instantiated modules in a tree like fashion. Double clicking on a module name will display its code.

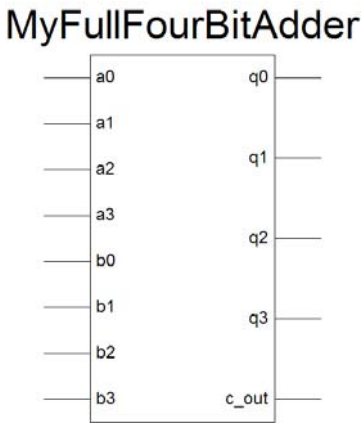
A final comment on the instantiation process concerns the pin assignment information using the *LOC* statement discussed in the previous section: avoid placing any pin assignments in modules that will be instantiated! Instead, always place all your pin assignment statements in the Top Module!

The reason for this is as follows: each time a module is instantiated a new set of inputs and outputs are created. Therefore, had the inputs and outputs already been assigned to a specific set of pins, instantiating such a module multiple times, which is completely reasonable, would tie the inputs or outputs of each instantiated module to the same pins. This clearly would not be work. (If you're still not convinced try to visualize what would happen to the outputs of a module instantiated multiple times with all of the outputs connected to the same pin. Clearly the outputs would short each other out.)

Therefore, *always* place your pin assignment statements in the Top Module! To avoid confusion, it is a good practice to remove all *LOC* statements from modules that will be instantiated. Warning: failing to remove the *LOC* statements from (sub) modules does not produce any error messages by the Xilinx compiler when it generates the program files and it usually results in code that simply will not work!

Exercise 2: 4-Bit Full Adder

Use the 2 bit adder example and build a complete 4 bit full adder in Verilog by instantiating your previously build full one bit adder module multiple times. See the graphical representation of the 4 bit adder below.



Consider a_0 and b_0 to be the LSB and a_3 and b_3 the MSB of each 4 bit term. You may permanently set *c_in* to 0 using either an *assign* statement or by setting corresponding input of the instantiated module to 0. Use the table below to assign the inputs and outputs to the correct pins.

Name	Function	BASYS Control	Pin
a0	Input: LSB of a	Switch SW0	
a1	Input: 1st bit of a	Switch SW1	
a2	Input: 2nd bit of a	Switch SW2	
a3	Input: MSB of a	Switch SW3	
b0	Input: LSB of b	Switch SW4	
b1	Input: 1st bit of b	Switch SW5	
b2	Input: 2nd bit of b	Switch SW6	
b3	Input: MSB bit of b	Switch SW7	
q0	Result: LSB bit of q	LED LD0	
q1	Result: 1st bit of q	LED LD1	
q2	Result: 2nd bit of q	LED LD2	
q3	Result: 3rd bit of q	LED LD3	
c_out	Carry-out and MSB of Result	LED LD4	

Table 13.2.

Write your code and download it to the BASYS board. Test it. Note that the 4 right most switches represent term *a* while the 4 left most switches represent *b*. The LEDs should then directly correspond to the result that the addition of the two terms produces.

Write-up

13.2.2. Hand in the Verilog source code for your full four bit adder.

Wires and Buses

For the final exercise in this section you will make two improvements to your previous project. First, you will use the 7-Segement LED display to show the addition results directly in decimal notation. This will require the instantiation of a display module provided to you, courtesy MXP labs. The second improvement concerns the large number of inputs and outputs that our designs are growing into. Clearly, if we were to expand our 4 bit full adder into a 16 or 64 bit adder, the current approach of specifying each input and output wire is inefficient. This problem is remedied by the introduction of the Verilog concept of buses.

The definition of a bus, also sometimes called a vector, is that it represents a collection of wires. (Again an analogy exists with C in the concept of an array being a collection of variables.) But this leads us to ask, what exactly do we mean by wires?

In Verilog, as in reality, wires do not maintain information. They only transmit it from an output to another input. Wires can be explicitly declared in Verilog through the *wire* keyword. (We will discuss this in more detail below.) However, when inputs or outputs are used without further specifications, Verilog implicitly assumes that they are wires. For example, in the *FullTwoBitAdder* module shown previously the wire *c_out0* was used as an input and output. Because it was not explicitly declared, Verilog assumed (correctly) it to be a wire and not something else.

If a wire is not part of an input or output then it must be *explicitly* declared, using the keyword *wire,* before it can be used in an *assign* operation. See the example below.

```
        wire a, b;
        wire wLO = 0;   //declared and assigned to a permanent value at the same time.
        assign a = b;   // assign b to a
```

You will find that declaring wires, even when it is not required, will make your code more readable.

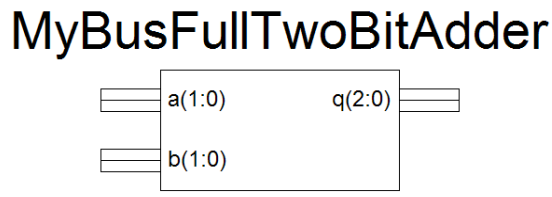
Buses, on the other hand, must always be declared. (If you neglect to declare the bus (size), Verilog assumes it to be a wire!) See the bus declaration examples below:

```
        wire [3:0] a;           //a 4 wire bus consisting of a[0], a[1], a[2] and a[3].
        wire [7:0] x = 255;     //an 8 bit wire bus permanently assigned to decimal 255 or
                                // binary 1111 1111
```

When a bus is used as an input or output, the declaration can often be combined with the input / output declaration as shown below in a modified version of the previous MyFullTwoBitAdder module:

```
module MyBusFullTwoBitAdder(a, b, q);
    input [1:0] a, b;           //2 bit input buses
    output [2:0] q;             //3 bit output bus

         MyFullOneBitAdder FullOneBitAdder0(a[0], b[0], 0,      q[0], c_out0);
         MyFullOneBitAdder FullOneBitAdder1(a[1], b[1], c_out0, q[1], c_out1);

        assign q[2] = c_out1;
endmodule
```

This module shows how individual wires within a bus are accessed by using the bus name and a bracket, similar to the C-notation for array elements. The graphical representation of the module is given below. Note, that to distinguish the buses from wires, buses are drawn with wide or bold lines.

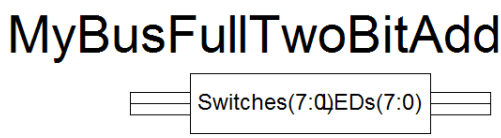


Wires and other buses can also be combined into larger buses as shown in the example below. Note the required curly brackets required when combining, i.e., concatenating, bits!

```
 wire a, b;                     //one bit wires a and b
 wire [4:0] c, d;               //5 bit buses c and d
 wire [7:0] mybus1, mybus2;     //8 bit buses called mybus1 and mybus2
```

```
 assign mybus1 = { 0, c, a, b}; //combine these to form: 0 c[4] c[3] c[2] c[1] c[0] a b
 assign mybus2 = { d[5:2], c }; //combine to: d[5] d[4] d[3] d[2] c[4] c[3] c[2] c[1] c[0]
```

A lot of new material has been covered. Before we go on to the next exercise, let's see how we can simplify our two bit adder design some more by using a bus called Switches, to control the inputs, and a second bus called LEDs, to control the outputs. See its graphical representations below:

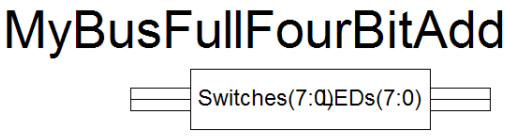


Its implementation, including the pin assignment statements, is shown in the module below. Note how the *LOC* statement has been used to assign pins to an entire bus.

```
module MyBusFullTwoBitAdderV1(Switches, LEDs);
        //control numerical inputs
        (* LOC = "P6 P10 P12 P18 P24 P29 P36 P38" *) input [7:0] Switches;

        //display result of the addition
        (* LOC = "P2 P3 P4 P5 P7 P8 P14 P15" *) output [7:0] LEDs;

         wire [1:0] a, b;       //2 bit input buses
         wire [2:0] q;          //3 bit output bus

         assign a = Switches[1:0];      //input a
         assign b = Switches[5:4];      //input b
                                        // Note Switches 2, 3, 6 and 7 are not being used
                                        // in this particular exercise.

         assign LEDs = {5'b00000, q};   //send result to LED (NOTE: the notation 5'b00000
                                        //stands for five (binary) bits all set to 0)
                                        //It will be explained more in the next section.

         MyFullOneBitAdder FullOneBitAdder0(a[0], b[0], 0, q[0], c_out0);
         MyFullOneBitAdder FullOneBitAdder1(a[1], b[1], c_out0, q[1], c_out1);

         assign q[2] = c_out1;
endmodule
```

Exercise 3: 4 Bit Full Adder Using Buses

Use the 2 bit adder example above and build a similar 4 bit full adder in Verilog using buses. See the graphical representation of the 4 bit adder shown below.



Use a bus for the input Switches and one for the output LEDs. Use the *LOC* statements shown in the module above.

Declare 4 wire buses for *a* and *b*; assign the 4 right most switches to *a* and the remaining ones to *b*. Declare a 5 wire bus for the result, *q,* and assign it to the LEDs output bus.

Write-up

13.2.3. Hand in the Verilog source code for your full four bit adder with buses.

Exercise 4: 4-Bit Full Adder Using Buses and 7 Segment Display

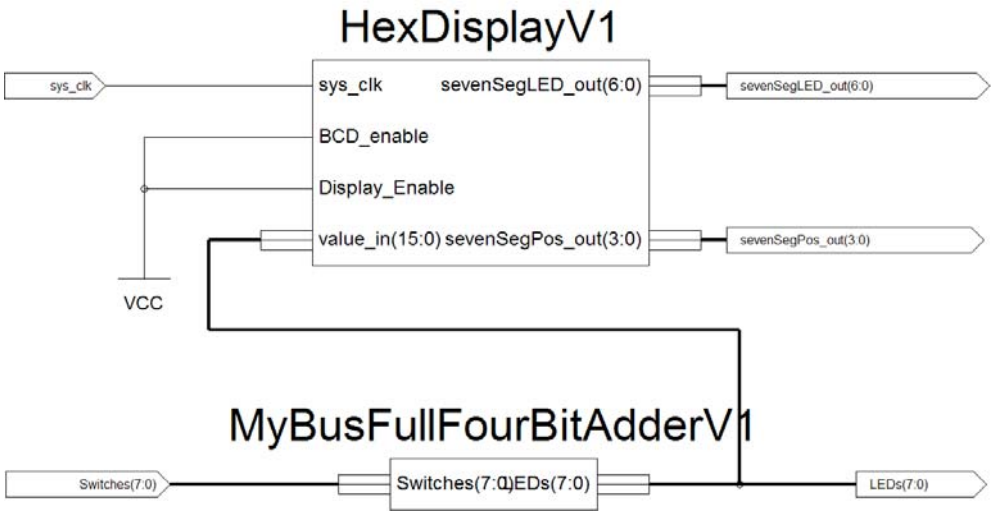
For the last exercise in this section, we turn your 4 bit adder design into a crude calculator. We will instantiate an already written module that will display the result of the addition in decimal notation on the 7 segment displays.

Copy the file HexDisplayV1.v from the folder U:\pub\Verilog\HexDisplay to your current working Xilinx project directory. To do so, go to the menu bar and select *Project \ Add Source* and then select the HexDisplay1.v file and copy / paste it. (Note: you must copy the file into your working directory; linking to the (read-only) file in the pub directory will not work!)

The file should appear in the "Sources" window. Double click on it to open it. You will notice that it contains multiple modules. The one that is of interest to us is the very first one, called *HexDisplayV1*. (The other modules are support modules that will be instantiated by HexDisplayV1.) It has the following inputs and outputs:

Name	**I/O**	**Bus Size (Bits)**	**Purpose**
sys_clk	Input	1	25 MHz BASYS board system clock; required for updating display.
value_in	Input	16	Binary value to be displayed.
BCD_enable	Input	1	If this is set to 1, the display is in decimal format, else, it is in hexadecimal.
Display_Enable	Input	1	If it is 1, the display is turned on or lit; else it is turned off.
sevenSegLED_out	Output	7	Output signal to the actual 7 segements of each display.
sevenSegPos_out	Output	4	Output signal to turn one of the 4 displays on.

Modify a copy of your latest version of the 4 bit full adder program so that it represents the graphical representation shown below.



Tips:

The final module has 2 inputs and 3 outputs. Of the inputs, only sys_clk is new and it will provide the 25 MHz clock signal to the instantiated *HexDisplayV1* module. Add two new buses to the existing output: one for sevenSegLED_out and sevenSegPOs_out. These two buses are directly connected to the pins of the seven segment display. Pin assignments for the three new buses / signals can be found in Appendix K or by studying the first few lines of code in the HexDisplayV1 module.

Compile your code and download the bit file into the FPGA. Test your crude calculator bit operating switches SW0 through SW7 and check the result in the display.

Write-up

13.2.4. Hand in the Verilog source code for your full one bit adder calculator with the HEX display.

.

Additional Comments: Conditional Assignments Using Buses

Before we race to the next Verilog topic, here are some comments concerning some topics that might be of interest to you in future projects.

In the previous section, we briefly covered the conditional assignment statement using 1 bit wires. With buses, more complex logic conditions, similar to a C if – if else statements, can be implemented this way:

```
wire [11:0] y;
wire [3:0] q;
wire [1:0] c;
assign q = (c ==  2'b00 ) ? y[3:0]:  //NOTE Notation: 2'b10
           (c ==  2'b01 ) ? y[7:4]:  //2 stands for the number of bits to be assigned
           (c ==  2'b10 ) ? y[11:8]: //'b stands for binary notation
            4'b1111;                 //10 represents that actual assignment, i.e., 2
```

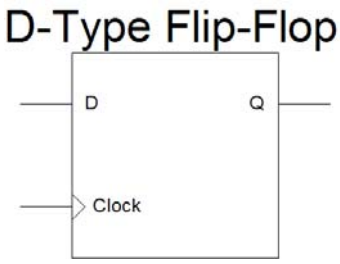
In this example, when *c* equals 0, *q* is set equal to the four least significant bits of (bus) *y*. When *c* is 1, the 4 next larger bits are assigned to *q*, etc. If none of the conditions happens to be met, a default condition, the last statement, is executed and *q* will be assigned a (decimal) value of 15.

(Note the Verilog notation for assigning values, such as 4'b1111. The first number, 4, represents the number of bits that will be assigned. The value after the single quote stands for the notation of the values following, with 'b indicating binary, ('h hex and 'd for decimal.) Finally, the four ones represent the actual bit values.)

13.3. Sequential Logic: Registers

In sequential logic circuits, the output depends not only on the present input but also on the history of the input. Digital logic applications would be very limited if they were restricted only to combinational logic devices. It is the joining of combinational logic devices with devices that retain their memory that has made digital logic so versatile and ubiquitous in our lives.

The most familiar sequential logic device is the D-type Flip-Flop, a one bit memory device, shown below.



It has two inputs. The *D*-input is for the data while the *clock* input instructs the device when to load and subsequently retain the data. The stored data is accessible at all times through the *Q* output.

The Verilog equivalent of a D-Type Flip-Flop is a **register**. It is declared with the ***reg*** keyword. Similar to wire buses, a collection of registers can be declared to extend its size from 1 bit to multiple bits. The size limit of these collections is only limited by the available memory in the FPGA. With the currently used FPGA on the BASYS board the sum of all registers cannot exceed 8000 (bits.)

Below are two examples that show how to declare and initialize registers. The first line declares a single bit register while the second one declares a 16 bit collections and then initializes them all to 0.

```
reg little_r;           //one bit registry
reg [15:0] big_r = 0;   //16 bit registry, all initialized to 0
```

The clock function of a Flip-Flop is executed in Verilog through the *always* @ keyword and the statements directly following it, also sometimes called the always block. See the example below:

```
reg q = 0;
always @(posedge clk_in) begin
        q <= ~q;
        end
```

In this code segment the always block consists of three lines, highlighted in bold type. The first line of the always block specifies that all statements between "*begin"* and "*end"* will be executed at every positive edge of the clock signal called *clk_in*. In this particular example, the one bit register named *q* is negated each time, i.e., it is being toggled.

Note that the clock input signal must always be preceded by the *posedge* or *negedge* keyword. Completely omitting these edge descriptors implies dual triggering, i.e., triggering on both the positive as well as the negative edge of the input clock signal. This should be avoided because the FPGA used on the BASYS boards is not capable of dual triggering. Therefore, always include either the positive or the negative edge descriptor with the clock signal!

If you carefully studied the example above, you may have noticed a new Verilog operator: **<=**. It is called a non-blocking assignment operator - as opposed to the blocking assignment operator previously encountered in the combinational logic section. The non-blocking operator is used when assigning something to a register in an always block.

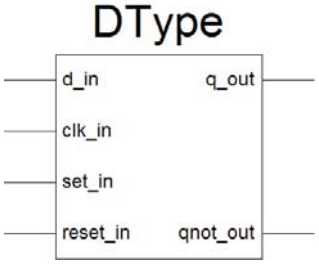
For now do not worry if you feel confused about the blocking and non-blocking operators. We will have more to say about that topic at the end of this chapter. For now try to think of it this way:

- if you use combinational logic, for example if you assign something to a wire, then you use the *assign* keyword and the blocking operator, i.e., the equal sign;
- if you use sequential logic, for example if you assign something to a register in an always block, then you should use the non-blocking assignment operator, i.e., <=; in this case also omit the keyword *assign*.

Let's look at another example. The module below mimics the simple positive edge triggered D-type Flip-Flop discussed earlier. See if you can understand it entirely.

```
module DTypeSimple( d_in, clk_in, q_out);
        input d_in, clk_in;
        output reg q_out;       //Note how the output and the register specification has been
                                //combined into one statement.
        always@(posedge clk_in) begin
                q_out <= d_in;
                end
endmodule
```

The next and also last example of this section illustrates a more complex D-type Flip-Flop. (See the figure below.)



In addition to the inputs and outputs previously discussed, this D-Type Flip-Flop has negative edge triggered asynchronous *set* and reset *inputs* and complimentary q and !q outputs. Study its Verilog code shown below.

```
module DType( d_in, clk_in, set_in, reset_in, q_out, qnot_out);
        input d_in, clk_in, set_in, reset_in;
        output q_out, qnot_out;
        reg q_out;

        assign qnot_out = !q_out;

        always@(posedge clk_in or negedge set_in or negedge reset_in)begin
                if( set_in == 0) begin
                        q_out <= 1;
                        end
                else if( reset_in == 0) begin
                        q_out <= 0;
                        end
                else begin
                        q_out <= d_in;
                        end
                end

endmodule
```

Notice the *clock*, *set* and *reset* inputs: *clk_in*, *set_in* and *reset_in*. They are asynchronous, i.e., they occur independently of each other. Therefore, each of them must be individually listed in the always block's clock signal list, separated by the *or* keyword.

Since every one of them is able to activate the always block, conditional statements in the form of if-statements are used to discern which input signal has activated the always block and also what actions are to be taken.

Again compare the sequential logic's branching instructions with those of the combinational logic. While *if / else if* statements can be used in sequential logic statements, in combinational logic you must use the conditional assignment operator *c ? a : b* discussed earlier.

Finally, note that the *begin* and *end* keywords function just like the curly brackets in C and, similarly, are optional if they enclose only one statement.

13.3.1. Exercise 1: Binary Counter

In addition to storing information, sequential logic devices provide the core of counters and shift registers. Lack of time prevents us to study the use of shift registers which can be found in parallel to serial converters, mathematical logic units and random number generators, to mention a few. Instead, in the following exercises, you will start using the Verilog registers to build various counter applications.

Verilog makes the design of multi-bit counters easy. It includes an addition (or subtraction) operator that increment (or decrements) a register at the transition of a clock input. In the example below, the counter is incremented each time at the rising edge of the *clk* input.

```
reg [3:0] count;  //4 bit register declaration

always @(posedge clk) begin
     count<=count+1;
end
```

In the Xilinx ISE Webpack, create a complete 4 bit binary counter module. Assign the counter outputs to the LEDs LD0 (LSB) through LD3 (MSB.) Use Switch SW0 as your clock input to increment the counter. By operating SW0 you should observe that the counter increments in binary: 0000, 0001, 0010, 0011, 0100 and so on.

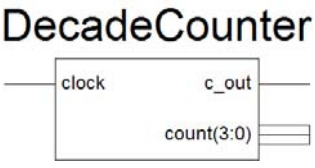
Now see what happens to your counter if you use a push button instead of the switch button as your clock signal. Change the clock input in your Verilog program to one of the push buttons on the BASYS board, for example, pin 69. Press the push button and keep track of the counts displayed on by the LEDs. Does your counter occasionally seem to skip some digits? What you are observing is the effect of a push button which has not been properly "debounced." In other words, each time you press the push button, instead of a single, clean, low to high transition, multiple such transitions are created as the switch "makes" or "breaks" contact. Various methods to remove this extremely annoying (and very common) effect exist; for a discussion see H&H pages 576 to 577.

Write-up

13.3.1. Hand in the Verilog source code for your 4 bit counter.

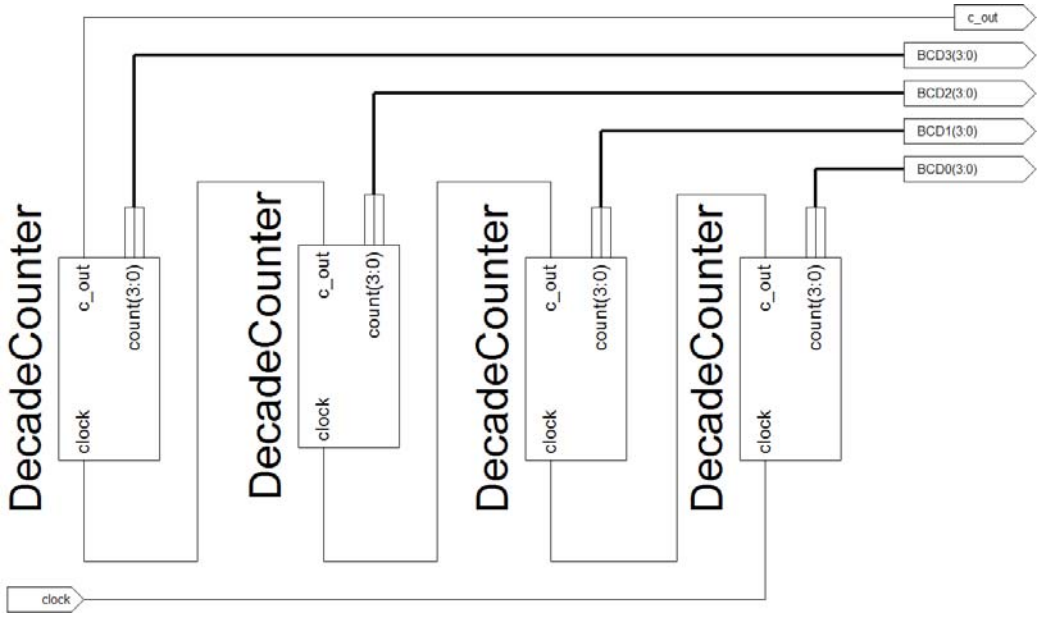
13.3.2. Exercise II: Decade Counter (Optional)

An n-bit binary (up) counter will count up to 2^n-1 and then starts the cycle at 0 again. On the other hand, a decade counter resets itself after 10 cycles. It counts up to 9 and then returns to 0. In other words, it never reaches 10, or for that matter, 11 through 15.



Such a counter module is useful. For example, if we were to connect multiple instances together (as shown below) we would have a counter that counts in decimal, or rather in binary coded

decimals, BCD, instead of binary. Hence, we no longer would need a binary to decimal conversion module.



Build a (single) decade counter with a clock input, a 4 bit count output and a c_out output. The c_out output goes high for one cycle when the counter is reset to 0. It can be used as clock input for the next higher digit, as shown in the picture above. (Hint: To reset the counter and to set c_out HI, you will need an if-statement somewhere in the always bock.)

Instantiate the decade counter in a separate module and use SW0 as the clock input. Display the count and c_out output using LEDs LD0 through LD4. Check that it works and that it resets itself always after 10 consecutive clock cycles.

Write-up

13.3.2. Hand in the code for the decade counter.

13.3.3. Exercise III: Seconds Clock

Besides its obvious function to keep track of counts, counters are also used to generate timing signals that are slower than their clock input. These timing signals then drive other devices at the new slower frequency.

(In terms of circuit design, you may note the similarity between the counter and the (resistive) voltage divider's usage. While one operates in the voltage domain, the other operates in the time (or frequency) domain. Both circuits are used to generate signals that are fractions of a reference signal.)

In this exercise you will use the BASYS board's 25 MHz clock to build a 1 second reference timer. For now, this 1 second timer will be used to turn an LED on every other second for 1second. In the next chapter, it will be used as a reference timer for a radiation counter monitor.

Use the 25 MHz signal from Pin 54 on the BASYS board and feed it into a n-bit counter. Design the counter so that each time the counter reaches a value corresponding to an elapsed time interval of 1 second, it resets itself to 0. Simultaneously, on each reset, it toggles an output signal, *q_out*. Assign this output signal to an LED and check that your design works, i.e., every other second, the LED should be lit for exactly one second.

Hints: First “calculate” the number of 25 MHz clock cycles that elapse in a second. Since your counter will have to count up to this number, determine the number of bits required to hold such a large number. Declare the counter register with the required number of bits. Next, in your always block, use if-statements to determine if the counter has reached its 1 second limit and needs to be reset or if it should be incremented further. On each reset, toggle another, separate register which serves to control the LED.

Write-up

13.3.3. Hand in the code for your working second reference clock.

13.4. Radiation Monitor / Counter

In this section we will “recycle” some of the previous modules to build a “pretty good” radiation survey instrument that displays the number of ionizing events measured in a one second time interval.

13.4.1. Radiation Detectors

The radiation detector, a Model RM 60, was manufactured by Aware Electronics. It utilizes a “cigar shaped” (as opposed to a pan-cake shaped) Geiger Mueller (GM) tube to detect ionizing particles. Each time the gas in the GM tube has been ionized by a particle passing through it, the unit generates a 50 µsec negative going LVTTL pulse that is being sent to the BASYS board. It is these pulses that we want to count when they trigger our radiation counter.

Up to four RM 60 units can be powered and interfaced to the BASYS board through the edge connectors JA through JD as shown below.

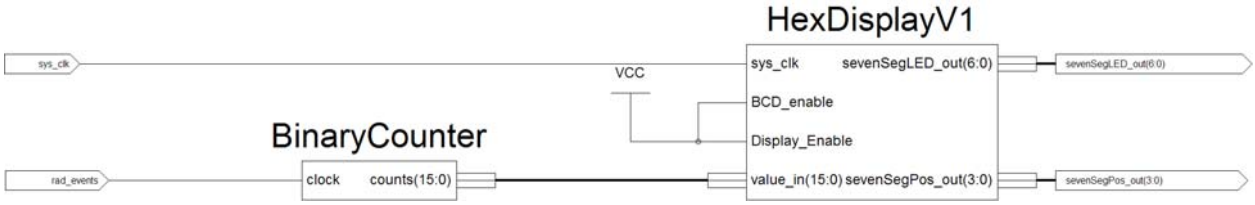
Figure 13.4. RM 60 Radiation Monitor with Thorium Oxide gas mantel and BASYS board. The RM 60 has been connected to pin 1 of the BASYS board's JA edge connector which corresponds to pin P81 on the FPGA.

The labels on top of the connector jacks indicate the pin number of the edge connector receiving the pulse signal. For this exercise, you will use only one detector. Connect it as shown in the picture above and place it near a gas mantle. (Please do not unpack the gas mantels.) These gas mantels contain Thorium Oxide. The Thorium itself emits a very low level of radiation as it decays through along decay chain into lead. It is these decays that we will measure.

13.4.2. Step 1: Basic Continuous Counter

In this exercise you will build on the Verilog code that has already been written and tested by you in previous sections. For this first step, you will start out with a very primitive radiation monitor. In future steps, you will adjust or amend your code to improve your design.

Start a new project and implement the simple counting circuit with the 7 segment display shown below.



Before you instantiate your binary counter module from exercise 13.3.1., you probably want to expand its size from a 4 bit counter to a 16 bit counter. Once you have done that, connect it to an instance of the HexDisplayV1 module which you have used previously in exercise 4, section 13.2.

Once everything works, you should be able to observe the ionization events as the counter keeps continuously incrementing. (Of course you must connect the radiation modules to the BASYS board to observe anything.) From looking at the count rate, what do you estimate the count rate per second to be? (Do not use detailed calculations; instead, just look at your counter and notice which digits are updated about every second.)

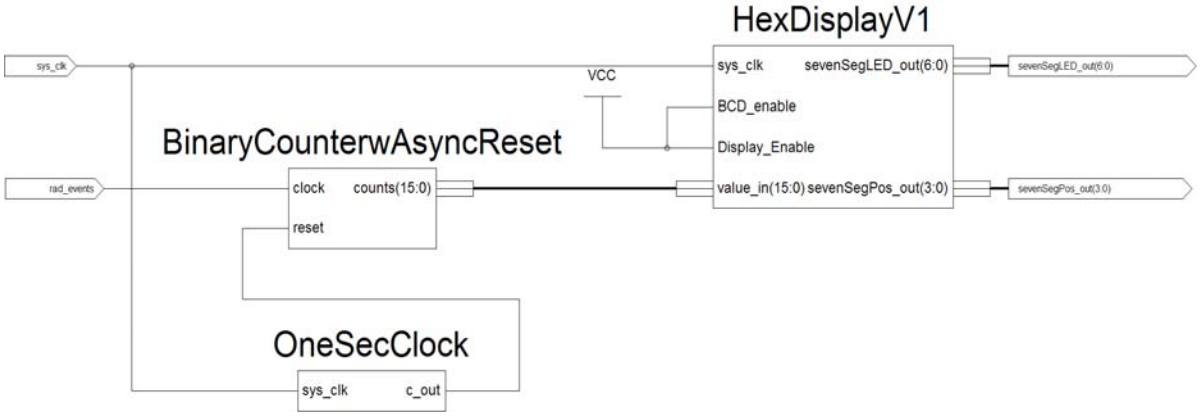
13.4.3. Step 2: Binary Counter with Asynchronous Reset

The previous version of the counter was displaying a continuous count of ionization events. What we really would like to observe is the number of counts in a given time period, for example, counts per second.

This can be achieved with two modifications / additions:

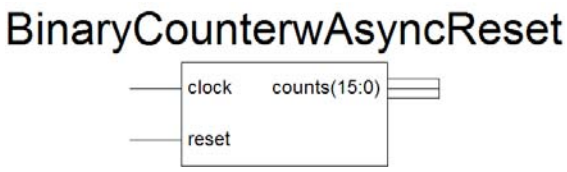
a) Add a reference time source that sends out a "trigger" signal every second.
b) Reset your counter at the beginning of each reference time period. Since the reset and clock signals are independent of each other, this is considered an asynchronous reset. (A synchronous reset would occur always at the same time as the clock signal.)

For a graphical representation on how to implement these ideas, see below.



Implementing part a) is trivial since you have already built a one second reference timer in exercise 13.3.3. You will instantiate in your code and use its output signal to reset your radiation events counter.

Part b) can be realized by modifying your existing binary counter that keeps track of the radiation events. First, add a *reset* input to the existing module.



Second, since the *reset* input is asynchronous it needs to be added to the list of clock signals in the *always* @ block. See the example below for a positive edge *reset* signal.

```
    always@(posedge clock or posedge reset ) begin
```

Third, since there are now two "clock" signals, *clock* and *reset*, an if-statement is required to discern the source of that activated the always block. Specifically, if *reset* is HI, then the counter is reset to 0, else the counter is incremented by one. Add these changes to your binary counter module.

Instantiate your new binary counter and the one second clock module and connect them as shown in the graphical representation above. Compile your code, load it into the BASYS board and watch its operation. (If it doesn't work correctly, for debugging, you may want to connect the 1 second clock output to LED LD0 on the BASYS board.) Is the maximum count value close to what you estimated in the previous section?

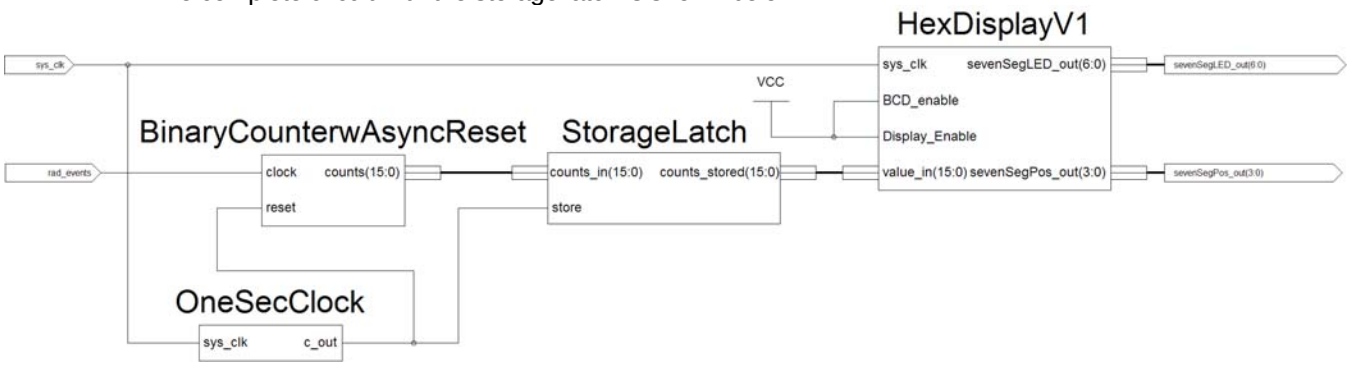
13.4.4. Step 3: Binary Counter with Asynchronous Reset and Storage Latch

The design still suffers from one flaw: though the counter now resets every second, it still displays a continuously incrementing display. What we are really interested in is only the count value right before it is reset. In other words, we need to store the count value in memory at the very same instant as the counter is reset.

You may add the code for the latching memory module directly to your existing module, or, you may create a stand-alone module (shown below) and then instantiate it in your code. The choice is yours.

The storage latch module consist of an always block that stores the *counts_in* input in a register of equal size (named *counts_out)* at each positive edge of the store signal. (Warning: it is tempting to add the code for assigning *counts_out* to *counts_in* directly to the binary counter's module always block, specifically, to the section that handles the asynchronous reset. This will not work! You must create a separate always block (as shown above) with only one clock source that handles the store process.)

The complete circuit with the storage latch is shown below:



Test your new circuit. Are the displayed values similar to the previously observed maximum values?

13.4.5. Step 4: Radiation Monitor Version with Improved "Dead Time"

The time interval during which an instrument is not able to measure or process its input signal is called "dead time." Dead time occurs when an instrument is busy with other (necessary) tasks. Examples of such tasks are the storing or processing of the acquired data, processing user input or updating its display.

The dead time is usually specified (in percents) as the ratio of dead time to the sum of dead time and "live" time. ("Live" time represents the time during which the instrument is measuring something.)

Percentage dead time = $t_{dead} / (t_{live} + t_{dead})$

A "good" instrument is one that has a low dead time ratio. A low dead time ratio assures:

a) That one obtains the same amount of data in a shorter overall time interval than if one were to use an instrument with a large dead time ratio;
b) That you will not miss any important events that may occur during the dead time interval.

Let's apply these concepts to your radiation counter. Note that it only counts radiation events every other second. Can you figure out why? (Hint: make a truth table for your binary counter with the asynchronous reset. Specifically, check what happens when *reset* remains HI.) So what is the percentage dead time of your current circuit?

We can decrease the dead time (ratio) of the radiation counter to almost 0 by decreasing the time interval that one second timer's *reset* signal stays HI. Change the design of the one second timer. Instead of toggling q_out and keeping it high for an entire second, keep it HI for only one 25 MHz cycle when the second counter is being reset. (After doing that, you may also have to adjust the 1 second maximum count value by one. Why?) Implement these changes and check that your counter still works. If it does, it should now be "a pretty good" radiation counter.

Write-up

13.4.1. Hand in the documented code for all your modules that you used to create the counter in step 4. (Documented means you must add comment statements explaining the purpose of each module and the purpose of the various inputs and outputs as well as the wires and registers within the module.)
13.4.2. What count rate did you observe with the gas mantle positioned close by the detector?
13.4.3. What is the dead time ratio of the radiation counter in step 3? What is it for the design in step 4 after decreasing the reset HI output to one 25 MHz clock cycle?
13.4.4. What is the cause of the dead time for both step 3 and 4. Explain specifically what (signal) causes it and what its effect is on the counter.
13.4.5. Dead time can also affect the results from your data analysis and, therefore, needs to be accounted for. For example, if you summed the displayed counter values over a fixed time period for the counters from step 3 and 4 you would observe a significant difference in the final sum. How does dead time affect it and how would you account for it to obtain the actual values of ionizing events?
13.4.6. What are the similarities / differences of this radiation monitor (use the design of step 4) to the previously build frequency counter in section 8.3.? Which major components are similar, which are different in their functionality?

13.5. More Comments and Summary on Verilog

This section contains no exercises (or write-ups) but tries to sum up some topics briefly touched on previously.

13.5.1. Verilog Summary

Here is a very brief summary on what has been covered so far:

To assign a (single) input (or output) to a specific pin use:

```
(* LOC = "P54" *) input sys_clk_in;
```

To assign an entire bus of inputs and outputs to specific pins (on the BAYS board) use:

```
(* LOC = "P6 P10 P12 P18 P24 P29 P36 P38" *) input [7:0] Switches;
(* LOC = "P41 P47 P48 P69" *) input [3:0] Buttons;
(* LOC = "P83 P17 P20 P21 P23 P16 P25" *) output [6:0] sevenSegLED_out;
(* LOC = "P26  P32 P33 P34" *) output [3:0] sevenSegPos_out;
(* LOC = "P2 P3 P4 P5 P7 P8 P14 P15" *)output [7:0] LEDs;
```

Note that the rightmost pin corresponds to the LSB of the bus and the leftmost to the MSB.

	Combinational Logic	**Sequential Logic**
Definition:	Output depends only on the present inputs. Has no memory.	Output depends not only on the present input but also on the history of the input. Has Memory

Main Elements:	Uses Wires
	Examples:
	`wire one_wire;`
	`wire [15:0] many_wires;`
		Uses Registers
		Examples:
		`reg one_register;`
		`reg [15:0] many_registers;`

Assignment:	Blocking Assignment Operator with *assign* keyword: Example: `assign one_wire = 1;`	Non-Blocking Assignment Operator without assign keyword: Example: `one_register <= 1;` Note: assignment must be done in an *always* @ block.

Branching Statements:	`assign one_wire = cond ? a : b;`	`if( cond == 1 )`
		`  one_register <= a;`
		`else`
		`  one_register <=b;`

13.5.2. Blocking vs. Non-Blocking Assignment Operators

The topic of blocking vs. non-blocking assignment operators has been named as one of the most misunderstood and confusing of Verilog. There is probably some truth to that and so, if you already feel overloaded with all the Verilog information presented so far, you may skip this section and return to it later when you feel more sure about Verilog's foundations.

Previously, it was stated that register assignments should always be done using the non-blocking assignment operator, the <=. (Note the keyword 'should.') The statement is correct and you may want to use the non-blocking assignment operator most of the time; nevertheless, you are also allowed to use the blocking assignment operator, the simple equal sign. If you prefer to use the blocking assignment with a register, you should not use the *assign* keyword. The only limitation on

using the blocking and non-blocking operator is that you can never mix the two when assigning values to the same register, for example, in a branching instruction.

(Just to clarify things: you can never use the non-blocking operator with wires! For wires, you always have to use the blocking operator with the *assign* keyword.)

So what's the difference between the blocking and the non-blocking operators? The difference between these two operators is how the timing of the actual assignment operation is carried out.

Since this discussion is only about registers, it follows that these assignment operations will at all times be part of an always block operation!

Let's look at an example using the blocking assignment operators shown below. As its name implies, a blocking assignment operator blocks the execution of the statement(s) following it until it has completed its assignment. In that sense, its behavior is identical to what we are used to in standard computer languages such as, for example C or BASIC.

```
always @ (posedge clock) begin
        b = a;
        c = b;
        d = c;
        end
```

In the example shown here, the assignments will be executed line by line. First, *a* will be assigned to *b*; next *b* (which now contains the value of *a*) to *c* and finally *c* (which also contains the value of *a*) to *d*. The final result, after the always event had been triggered, is that the value of *a* has been assigned to all the other variables in the always block. Also note that if we were to rearrange the order of the three assignment statements, the outcome would be different!

Now let's see what happens if we use the non-blocking assignment operator as shown in the example below. As the name implies, an assignment is no longer blocked until its previously listed assignment has been completed. Instead, all assignment are carried out concurrently and at the very instance the always block is executed.

```
always @ (posedge clock) begin
        b <= a;
        c <= b;
        d <= c;
        end
```

How is that accomplished? Try to think of it as a two step process: the first step occurs prior to the always event becoming true. In this step all assignment operations are calculated and stored in a temporary memory location. The next step happens at the very instant the always block is activated, i.e., at edge of the clock signal. At this point all the previously stored values are assigned simultaneously to the registers.

Here is yet another way to think of this two step process: in the first step, only the right hand side of the assignments has been completed. In the second step, the left hand side of the assignment is carried out.

Now let's look at the example. In the first step, prior to the always block being activated, the values of *a*, *b* and *c* are stored in a temporary location. If we indicate the temporary registers with a prime then:

a → b'
b → c'
c → d'

At the next step, when the always event has been activated, the previously stored values are assigned to the actual registers.

b' → b
c' → c
d' → d

This ultimately results in:

b contains the value of what *a* had been previously;
c contains the value of what *b* had been previously;
d contains the value of what *c* had been previously.

With every further clock period the values *of a, b, c* and d are shifted "down," i.e., it acts like a shift register.

It has been stated that non-blocking assignment operations are all carried out at the same instant. The result of this is that the ordering of the non-blocking assignment statements within an always block has no effect on the final result!

A more detailed paper on the assignment operators can also be found at:
http://csg.csail.mit.edu/6.375/papers/cummings-nonblocking-snug99.pdf

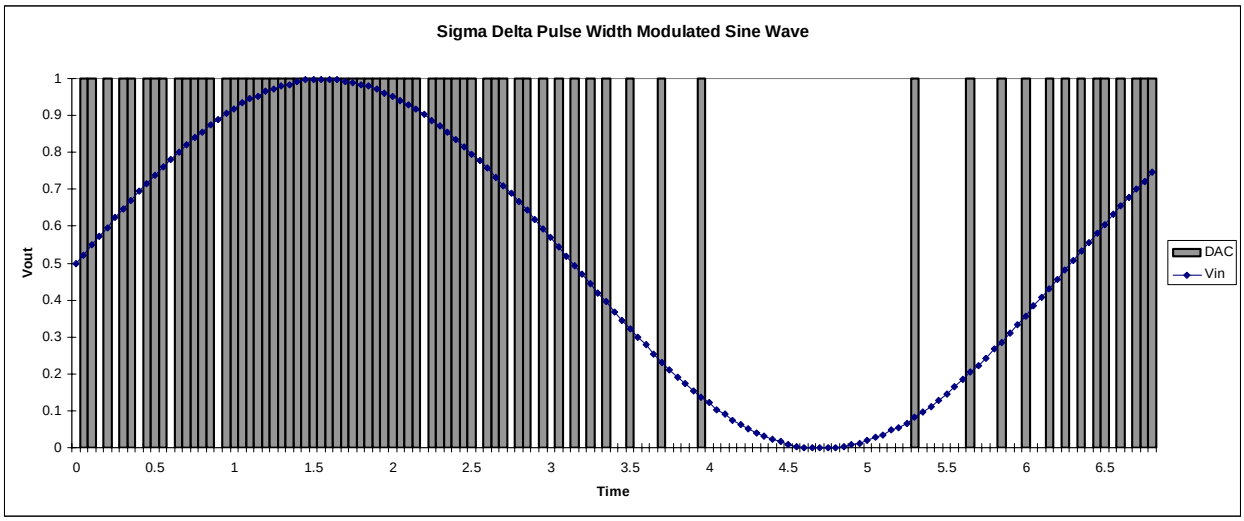
Chapter 14

FPGA and VERILOG Part II:

Write-up Format: Long

Introduction: Pulse Width Modulation

The exercises in this chapter will guide you towards the final project where you will be building a 16 bit audio player. The audio signal itself will be generated using a pulse width modulation (PMW) technique. Therefore, most of this chapter is devoted to this technique which is used in DC to DC (voltage) conversions and digital to analog applications. It is widely used because it is energy efficient and uses few electronic components.



Shown above is a graph of an analog Sine wave and its (Sigma Delta) pulse width modulated equivalent output.

14.1. Simple Pulse-Width Modulation (PMW) Technique

Introduction: Light (LED) Dimmer

In this exercise you will use the BASYS board to build a simple PWM circuit to control the brightness of an LED. Switches SW0 through SW7 will be used to adjust the intensity of the LED on the board.

To illustrate the concept of PWM, consider the following circuit: an LED is connected through a switch to a fixed (DC) supply voltage, V_{On}.

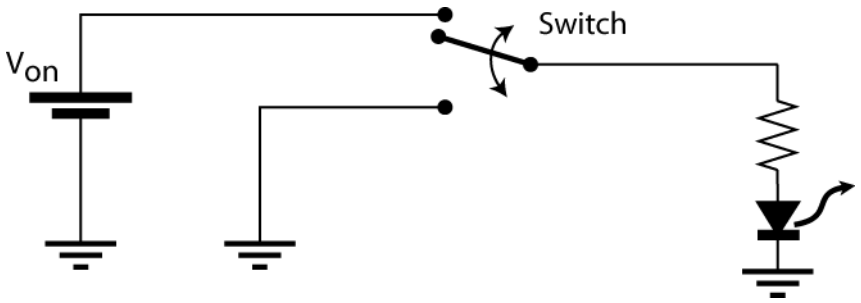


Figure 14.1.1. LED with manually operated switch.

The time it takes for entire switching cycle, τ_{swc}, consists of the time that the LED is on, τ_{on}, and off, τ_{off}:

$$\tau_{swc} \equiv \tau_{on} + \tau_{off} \qquad (14.1.1.)$$

The switching (cycle) rate or speed, f_{swc}, is defined as:

$$f_{swc} = 1/\tau_{swc} \qquad (14.1.2.)$$

At first, let us operate the switch by hand. As long as the τ_{on} and τ_{off} are on the order of seconds our eyes will observe the LED either at its full brightness or off.

Next consider what happens when we connect the switch control to a function generator. Assume for now that the time interval that the light remains on, τ_{on}, is equivalent to the time it is off, τ_{off}.

At a low switching rate, at a few Hertz, our eyes observe the familiar full brightness / off behavior. As the switching speed is increased we begin to notice the blinking LED. As the switching is speed is further increased, we reach a frequency where the response of our eyes, specifically the retina retention, is too slow to register the LED's distinct on and off states. Instead, at this frequency and beyond it, our eyes will act like an "averager" or low pass filter. We perceive an average intensity that is somewhere between the LED's full brightness and it being completely turned off. In the current case, with $\tau_{on} = \tau_{off}$, we probably perceive the LED at half its maximum brightness.

How does the ratio of τ_{on} and τ_{off} affect the "average" brightness perceived? Assume that the switching rate, f_{swc}, is kept constant and remains at a frequency exceeding our eyes' response time. As mentioned previously, at this switching rate, the perceived intensity will be an average based on how long the LED is on - or off - during each switching cycle. For example, if $\tau_{on} \gg \tau_{off}$, the LED is on most of the time and a bright LED will be seen; for $\tau_{on} \ll \tau_{off}$, the LED is mostly off and a dimly lit LED will be observed. It follows that by controlling the ratio of τ_{on} and τ_{off} we can obtain any desired level of intensity.

Let us determine mathematical relationship between τ_{on} and τ_{off} and the "perceived" average intensity. An analogous model of the eye and its response can be represented by the circuit shown in Figure 14.1.2. In this model, V_{on}, would be analogous to the maximum light intensity; the RC low pass filter acts like an averager, representing our eyes; finally, $\langle V_{out} \rangle$ is the average output voltage and it corresponds to the perceived intensity. Therefore, we would like to calculate $\langle V_{out} (RC, \tau_{on}, \tau_{off}) \rangle$.

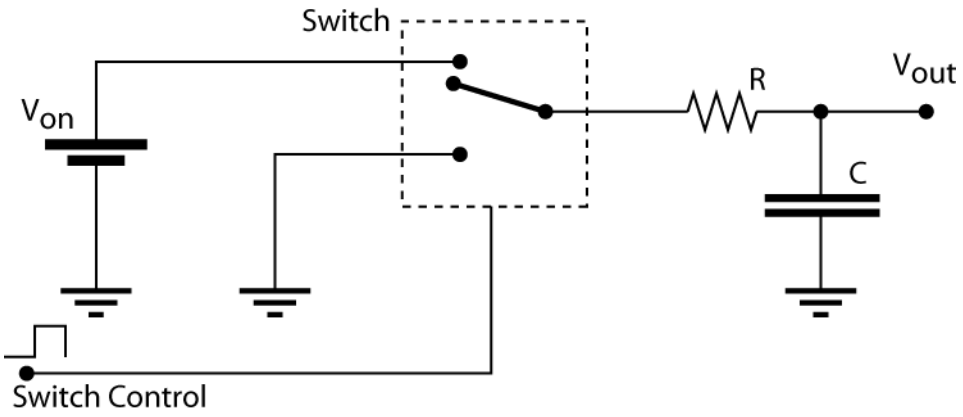


Figure 14.1.2. Pulse Width Modulation circuit with an external (electronic) switch control to turn the switch on or off.

As will be calculated in write-up 14.1.1, as long as RC is much larger then the time for one switching cycle, i.e., RC » τ_{swc}, $\langle V_{out} \rangle$ is:

$$\langle V_{out} \rangle = \tau_{on} V_{on} / (\tau_{on} + \tau_{off}) \qquad (14.1.3.a.)$$

$$= \tau_{on} V_{on} / \tau_{swc} \qquad (14.1.3.b.)$$

Before we proceed to implementing the PWM circuit in Verilog let's consider the implications of the results above.

First, these results show that with the aid of a timing signal controlling a switch, a fixed (DC) voltage can be converted into a voltage that it less than its maximum input voltage, V_{on}. In other words, for RC » τ_{swc}, the circuit acts very much like a voltage divider! Unlike the "normal" 2 resistor voltage divider, this circuit has the advantage that (ideally) no power is wasted in the regulation.

Second, if the system to which the PWM is applied responds much slower than the switching speed, then the low pass filter that provides the averaging is no longer required. For example, when using the LED and our eyes, the eyes will provide the low pass filtering. This is true for many other cases. For example, in "power" applications where the PWM technique is used to control speakers, motors, solenoids or other actuators, as long as the response time of the device being powered is much slower than τ_{swc} the low pass filter is omitted because the slow response of the system that is being controlled provides it indirectly.

Combining these two observation leads to a very important implication: since the PWM circuit acts like a voltage divider and if the low pass filter is part of the "load" then all the power will be absorbed by the load and none will be wasted by the regulating circuitry! In other words, a PWM circuit can be far more efficient in regulating power than a regular two resistor voltage divider where a lot of power is being wasted in the regulating resistors. (See Exercise 14.1.3.) This is one of the reasons that PWM are so popular in many low power applications such as portable electronics devices.

Verilog Implementation of a Simple PWM Technique

A critical element in the PWM technique is the switch control algorithm or module; it controls the time interval that the switch stays on, τ_{on}, or off, τ_{off}. (See the diagram in Figure 14.1.3.) This module has two inputs. An external clock input, f_{clock}, provides a trigger signal for the algorithm and also acts a reference clock signal. The *x_in* input specifies the number of reference clock cycles

that the switch stays on, or off. In other words, *x_in* will be directly proportional to τ_{on} and, hence, $<V_{out}>$ Its output, *PWM_out*, is the pulse width modulated signal. It can be used directly to control low power applications, such as dimming an LED but generally this signal is fed into some sort of physical switch, for example a transistor or relay, to provide a power amplification, as shown below.

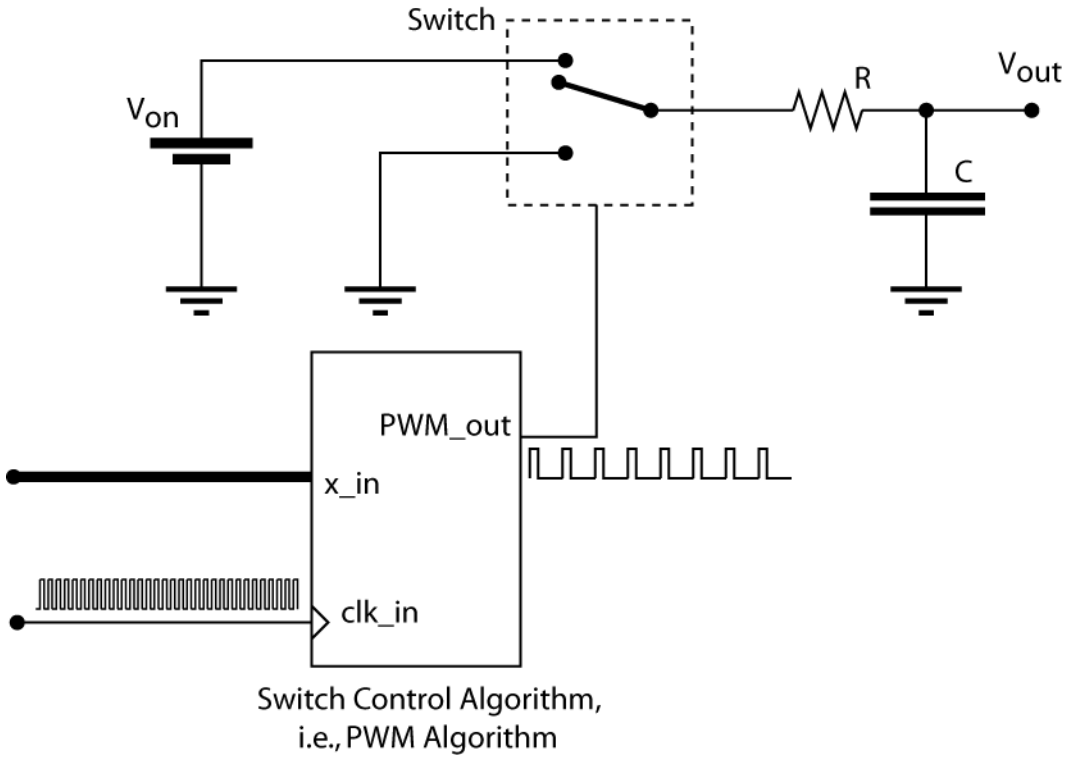


Figure 14.1.3. The complete PWM circuit. The actual switching element can be a physical switch, such as an electrical relay. Typically though transistors are used for switching, most commonly in the configuration of silicon controlled rectifier, i.e., an SCR. The low pass filter is not required if the system to which the PWM signal is applied to responds much slower than one complete switching cycle.

Now let's look at the actual implementation of the switch control algorithm. After this lengthy discussion it may come as a pleasant surprise to discover just how straightforward its implementation is in Verilog. The complete module is shown below.

```
module SimplePWM(clk_in, x_in, PWM_out);
        parameter MAXBITS = 8;   //maximum number of bits for input value and counter

        input clk_in;   //clock for counter
        input [MAXBITS-1:0] x_in;       //control value that defines pulse width
        output reg PWM_out = 1; //PWM signal out

        reg [MAXBITS-1:0] counter = 0;

        always@ (posedge clk_in )begin
                if ( counter < x_in )
                        PWM_out <= 1;
                else
                        PWM_out <= 0;
                counter <= counter+1;
                end
endmodule
```

Module 14.1.1. Simple PWM Technique. (You do not have to type this code in; instead it can be copied from the file PWM_Modules.v from the folder U:\pub\Verilog\PWM to your current project directory.)

Except for the "parameter" statement, all other Verilog commands should look familiar to you. The parameter statement is similar to a preprocessor or constant in C. It allows presetting values such as register size so that, at a later time, they can easily be adjusted throughout the code.

The main components of the PWM technique consist of a counter and a digital comparator. Whenever the continuous count value exceeds the input, *x*, the module outputs a LO, else it is HI. In other words, its output, *PWM_out,* acts like the switch output from the previous discussion.

The input, *x*, is a user selectable value. It controls how long the output remains HI or LO, i.e., how long the "switch" stays "on" and "off." Thereby, it determines the length of τ_{on} and τ_{off} . The range of *x* must never exceed the largest count value of the counter. Therefore, its range depends on the size of the counter specified in the module listed by the parameter *MAXBITS*. For an n-bit counter, its range is: $0 \leq x < 2^n$

Note that the counter runs continuously. Once it reaches its maximum value, it starts over again at 0. Since the counter is incremented at a frequency, $f_{clock,}$ the duration of an entire switching cycle for an n-bit counter is:

$$\tau_{swc} = 2^n / f_{clock} \qquad (14.1.4.)$$

From this it follows that the switch remains "on" for the following time interval:

$$\tau_{on} = x / f_{clock} \qquad (14.1.5.)$$

If the output from the module were connected to a low pass filter with a large RC time constant, (see figure 14.1.2.) then we would find that $<V_{out}>$ is:

$$<V_{out}> = x\, V_{on} / 2^n \qquad (14.1.6.)$$

In other words, the averaged output voltage is directly proportional to *x*. Since *x* is a digital value and $<V_{out}>$ an analog value, we see that the PWM circuit acts like a digital to analog converter, i.e., a D2A!

Exercise

In this exercise you will measure the response time of your eyes by connecting an LED to the output of a PWM circuit. You will adjust the PWM module's clock frequency, f_{clock}, until you are no longer able to observe the discrete blinking of the LED.

We do not want to use external components. Instead, we will implement a circuit (in Verilog) that generates an adjustable clock frequency, f_{clock} from signals available on the BASYS board. The method we will use is to drive an n-bit counter with the BASYS board's 25 MHz system clock. The LSB of the counter will then correspond to a frequency of 25 MHz / 2; the next higher bit has a frequency of 25 MHz / 4 and correspondingly, the n^{th} bit of the counter will have a frequency of 25 MHz/$2^{(n+1)}$. (Note the LSB corresponds to n = 0.) Therefore, by selecting the n^{th} counter bit and assigning it to the output *clk_out* will produce a discretely adjustable clock frequency, f_{clock}, to clock the PWM module.

"Build" the entire PMW circuit using the switches, clock and an LED on the BASYS board using Verilog. (Except for the BNC connectors to observe the output signal, you will not need any external components.) See the diagram below.

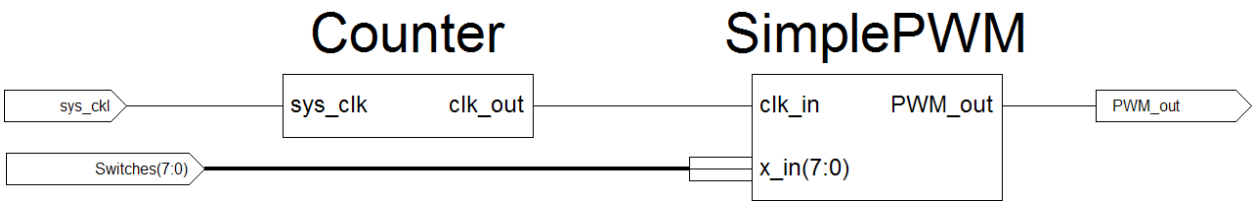


Figure 14.1.4. Simple PWM circuit with counter module to divide the 25 MHz system clock.

Use a 20-bit counter as a frequency divider to control the frequency of f_{clock}. To start with, assign the 12th bit to *clk_out*.

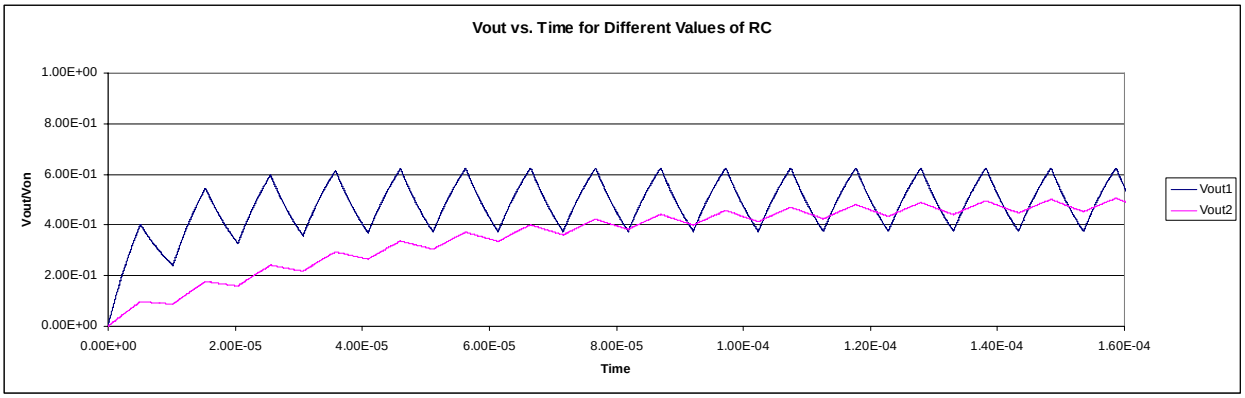
Instantiate the module called *SimplePWM* shown in module 14.1.1. Use Switches SW7 (MSB) through SW0 (LSB) to create the 8 bit control value, *x*, for the PWM module. Select any one of the LEDs LD0 to LD7 as the output. Additionally, also connect the *PWM_out* to pin 1 and *clk_out* to pin 2 of connector JA to observe the output signal and f_{clock} on the scope.

Upload the compiled code into the BASYS board. When you operate switches SW0 through SW7 can you control the blinking LED's brightness? While observing the LED, also study the PWM output on the scope.

Next assign *clk_out* to the counter's next lower bit, i.e., n = 11. Decrease the value of n until you can barely discern the blinking of the LED when the switches are set to provide the LED with half its intensity. The rate at which the LED blinks is the response time of your eye, or its f_{-3dB}. Calculate or measure it from the *clk_out* signal.

Write-Up

14.1.1. For RC » τ_{swc}, calculate $\langle V_{out}(RC, \tau_{on}, \tau_{off})\rangle$ using the circuit shown in Figure 14.1.2. (Hint: find a (differential) equation for the change in voltage, *dV*, across the capacitor during $dt = \tau_{on}$ and $dt = \tau_{off}$. Note that V_{in} during τ_{on} is V_{on} and it is 0 during τ_{off}. Finally, for the output to remain around a fixed (steady state) value, $\langle V_{out}\rangle$, *dV* during the on time must be equal to *–dV* during the off time.)



14.1.2. The graph above shows V_{out}/V_{on} as a function of time for two different values of RC for the circuit in 14.4.3. As you can see, adjusting the value of RC has two major effects on the output: how quickly the output responds and the size of the ripple. Describe how increasing (or decreasing) the value of RC affects these two properties.

14.1.3. To conserve energy, a person decides to dim the light in the house by adding a regulating resistor (in series) to a lamp. For now assume that the resistance of the regulating resistor is 5 times the resistance of the lamp. How much power overall was saved by the entire circuit? How much (more) power is absorbed by the regulating resistor then by the lamp? Why would a PWM circuit be a better choice?

14.1.4. What was the measured response time of your eye, or its f_{-3dB}.
14.1.5. What's the resolution of the simple PWM digital to analog converter?

14.2. Sigma Delta Algorithm to Produce Audio Clock Signal

Introduction

In this section, we study the Sigma Delta algorithm and then use it to generate the 44.1 kHz clock signal for our audio player. In the next section, the Sigma-Delta algorithm will be used to improve the simple PWM technique from the previous section.

Clock Divider Algorithms

Most digital circuits rely on multiple clock signals. Generally, a master clock signal, like the BASYS board's 25 MHz system clock signal, is selected as a reference signal. All other clock signals are then generated by "slowing" down the reference clock signal to the particular frequency of interest by using some sort of counter circuit.

You have already used two different methods for generating clock signals from a (faster) reference signal:

- In the previous section, the MSB of a continuous n-bit counter was used to divide the reference frequency by 2^n; this method is extremely straightforward but limits the range of new frequencies to: $f_{reference} / 2^n$.
- In section 13.3.3, a counter was reset each time a particular time interval (a second) was reached; this method works fine as long as the reference frequency is an integer multiple of the desired clock frequency.

When the reference frequency is <u>not</u> an integer multiple of the desired clock frequency then these methods cannot be used generate an accurate clock signal. Here is an example: assume that you have a reference clock signal of 256 Hz and that you need to generate a new clock signal, arbitrarily chosen, at 6 Hz. The ratio of these frequencies is 42.667. Since the flip-flops can only be triggered an integer number of clock cycles, the new output signal can either have 42 or 43 reference clock cycles. What is needed is some algorithm that corrects itself. For example, two cycles with 43 reference cycles will always be followed by a cycle with 42 clock cycles. Over time, the cycles will average out to 42.667 cycles, i.e., a frequency of 6 Hz. The Sigma Delta algorithm, shown below, achieves exactly that. (Its name originates from the fact that it continuously adds (Sigma) and subtracts (Delta.))

Here is a (C-style) version of the Sigma Delta algorithm with line numbers. It will be executed at each clock cycle at a frequency f_0. Its input values are *y* and *z*. The range of *z* and *y* is: $0 < y < z$.

```
1. Sigma = Sigma + y;
2. if ( Sigma >= z )
3.     {
4.     Delta = z;
5.     Out = 1;
6.     }
7. else
8.     {
9.     Delta = 0;
10.    Out = 0;
11.    }
12.Sigma = Sigma – Delta;
```

Figure 14.2.1. Sigma Delta algorithm.

A graph of *Sigma* (at line 12) and *Out* as function of clock cycles is shown below.

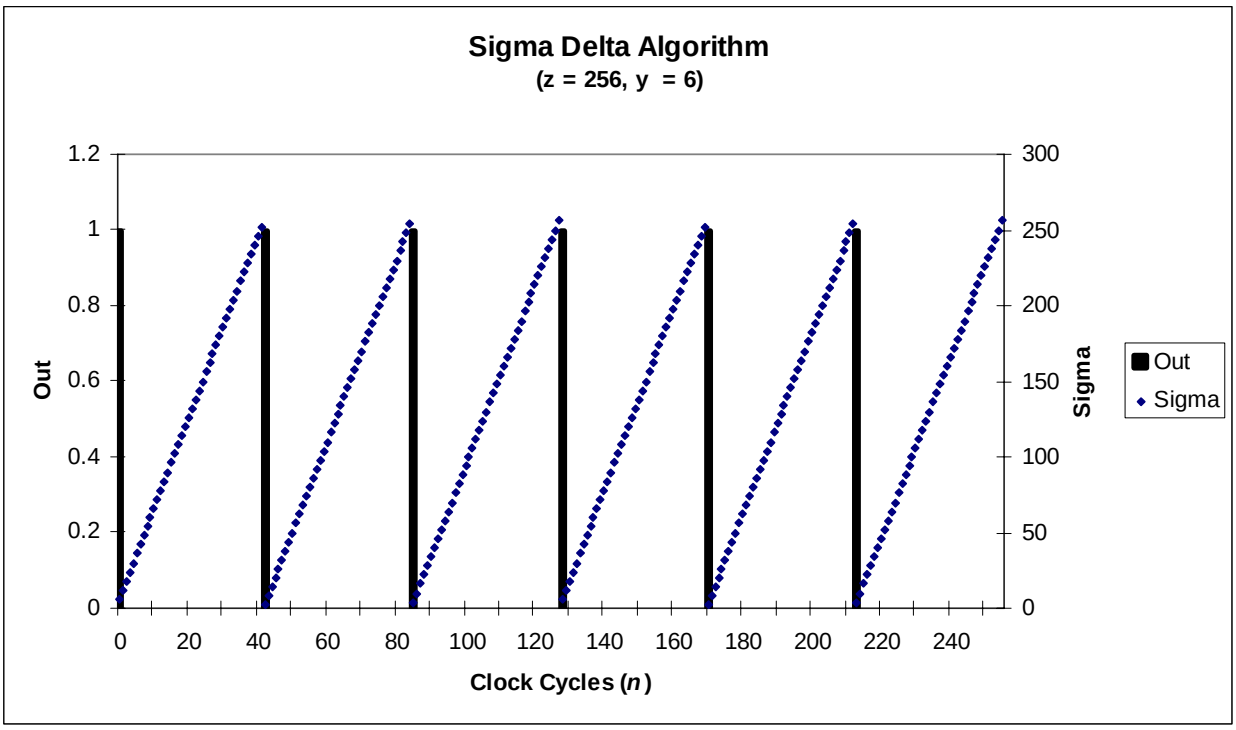


Figure 14.2.2. Graph of the variables "Out" and" Sigma" during 256 consecutive clock cycles.

The graph shows that the 6 *Out* signal pulses were (more or less) evenly distributed within the 256 clock cycles.

As you will show in write-up 14.2.1., if the algorithm is executed at a frequency f_o, then the average frequency for the Out pulses, $\langle f_{out}(f_o, y, z)\rangle$, with $0 < y < z$, is:

$$\langle f_{out}\rangle = y\, f_o / z \qquad (14.2.1.)$$

If the value for z is substituted by the actual clock frequency, f_o, then this becomes

$$\langle f_{out}\rangle = y \qquad (14.2.2.)$$

This trick comes in very handy if we want to generate some arbitrary clock frequency and, therefore, this property is often applied in voltage-to-frequency conversion schemes.

To sum up, given a reference clock signal, f_o, the Sigma Delta algorithm allows us:
a) to generate a number of pulses that are evenly distributed in time;
b) to generate a new clock signal with an average frequency y when $z = f_o$.
Property a) will be very useful in the next section where it will be applied towards a much improved PWM technique while property b) will be used in the subsequent exercise.

Exercise

Use the Sigma Delta algorithm and write a Verilog module to generate the 44.1 kHz audio clock signal using the BASYS board's 25.0 MHz system clock as its input. The module will output a positive going pulse for one system clock cycle (on the average) every 44.1 kHz.

Figure 14.2.3. Schematic of AudioClock module.

Hints: Non-blocking assignment operators can cause havoc with statements that must execute in a well defined sequence such as the ones shown in the (C-based) Sigma Delta algorithm. It is easier to translate the algorithm into Verilog using blocking assignment operators, =, instead of the non-blocking Verilog operators, <=. (Review section 13.5.2. on the difference between the two operators.) Nevertheless, if you want to challenge your Verilog skills, it is possible to implement the code with non-blocking operators and you might want to give it a try.

When you are implementing the code, check that the register size for Sigma and Delta is adequate to hold their maximum values. (Recall that $z = f_o$ = 25'000'000 and y = 41000.) In other words, calculate the bit size required for these registers! (If you want to implement a more efficient version of your code, note that as long as the ratio of *z* and *y* remains constant the code works the same; therefore, dividing *z* and *y* by 1000 will not change its behavior but will substantially lower the number of bits required!)

Test your module with the scope. Check that your module produces a positive going pulse at a repetition rate of 44.1 kHz. (Note: The 25 MHz system clock is only accurate to one or two percents of its nominal value. So don't be alarmed if you are off by one or two percents from the expected frequency.)

Write-Up

14.2.1. Derive equation 14.2.1. (Hint: one approach would be to calculate the average number of *Out* signals, n_{out}, generated during some arbitrary time interval, τ. The ratio of n_{out} / τ then corresponds to the average frequency of the Out pulses, $\langle f_{out} \rangle$. Assume that the time interval, τ, lasts *n* clock cycles, $\tau = n / f_{clock}$. At each clock cycle Sigma is incremented by *y*. From this you can calculate by how much Sigma was incremented over the entire time interval, Σ_{Tot}. Each time Sigma reaches or exceeds *z*, *z* is subtracted from it and an *Out* pulse is generated. Since over the entire time interval, τ, Sigma was incremented by Σ_{Tot} you should be able from this to calculate the number of Out pulses, n_{out}, generated during the time interval, τ.)
14.2.2. The 44.1 kHz rate for sampling and playing audio signal is an industry standard. Can you explain why this frequency was selected, i.e., what does it imply about the range of human hearing?

14.3. Sigma Delta PWM

Introduction

In this section we will use the Sigma Delta algorithm to improve the simple PWM technique used in the first section. The resulting circuit will then be used for the audio player's A2D converter.

Audio Player Specifications

Before we continue building our audio player, let's look at the specifications that a "decent" audio player must meet. First, it must be able to cover the audible frequency range which is: 20 Hz < $f_{audible}$ < 20 kHz. Second, the human hearing is able to distinguish changes in the audible intensity on the order of 1:10000. In other words, its resolution is $1:2^{14}$ or better; we will aim for a $1:2^{16}$ resolution. We now want to apply these specifications our simple PWM circuit from section 14.1. and see if they can be met using the BASYS board and its 25 MHz system clock.

Resolution and Frequency Response of the Simple PWM Circuit

The resolution of a D2A converter is defined as the smallest voltage change that the converter is able to output. In other words, it represents the "step size" in its output voltage when its input changes by the least significant bit, i.e., $\Delta x_{in} = \pm 1$. We prefer these steps to be small. This provides for "smoother" transitions between values and thereby reduces the noise associated with large steps. See the graph below for a Sine wave digitized at different resolutions.

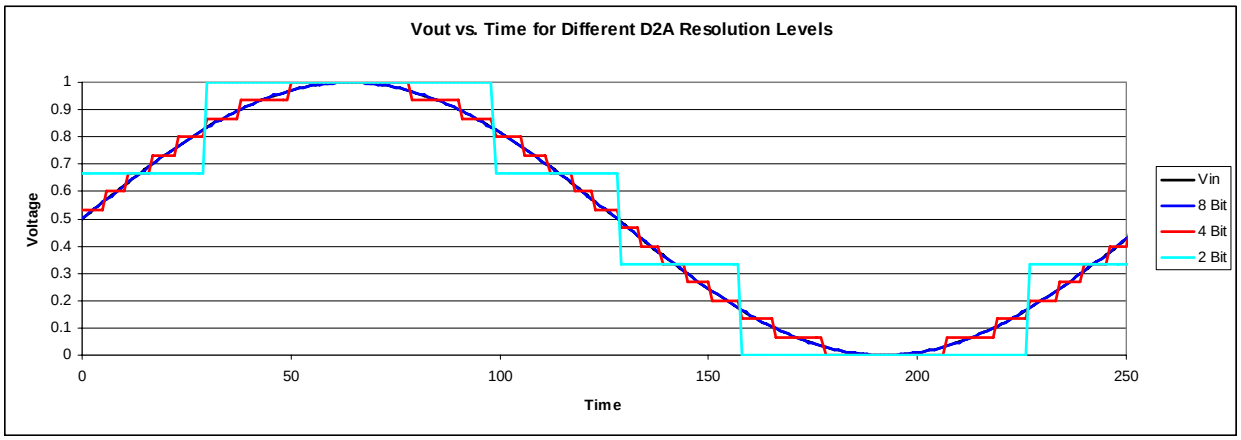


Figure 14.3.1. V_{out} for different resolution levels.

According to equation 14.1.6., the simple PWM D2A converter from the previous section is capable of producing 2^n distinct (averaged) output voltage levels ranging from 0 to V_{on}. Therefore, its resolution is $V_{on} / 2^n$. For a specific output voltage range, its resolution is entirely defined by *n*, i.e., the (bit) size of its internal counter. By adjusting the size of the counter register into a 16, 32 or even 64 bit counter we can easily adjust its resolution to obtain a more precise and smoother output. Achieving the specified 16 bit resolution certainly presents no problem for this design.

Next we need to determine the minimum time during which the input, x_{in}, to the PWM circuit must be maintained to obtain $<V_{out}>$ as specified by equation 14.1.3. Recall that $<V_{out}>$ must be averaged over at least one full switching cycle, τ_{swc}; changing x_{in} before at least one switching cycle has elapsed could lead to unpredictable results because we are no longer able to average over an entire cycle. From this it follows that the shortest time interval over which x_{in} is allowed to change is, τ_{swc}; applying equation 14.1.4. it follows that the maximum input frequency for simple PWM circuit is:

$$f_{MAX\ Simple\ PWM} = 1 / \tau_{swc} = f_{clock} / 2^n \qquad (14.3.1.)$$

For a 16 bit counter with a 25 MHz clock, f_{MAX} is a paltry 400 Hz. Clearly, it does not meet the audio frequency range specified previously.

Why is f_{MAX} is so slow? Because it takes 2^{16}, i.e. approximately 65000, clock cycles to complete just one switching cycle. Even at 25 MHz, this takes a long time. It is tempting to reduce the number of clock cycles for a given switching cycle but this can only be accomplished at the cost of

reducing the A2D's resolution. What is needed is some method to reduce the cycle time without affecting the resolution!

Resolution and Frequency Response of the Sigma Delta PWM Circuit

The Sigma Delta algorithm allows us to spread a signal over some arbitrary time interval by breaking it up into evenly spaced pulses. We will use this feature to reduce the time it takes for one switching cycle to repeat itself.

The key to understanding the Sigma Delta PWM technique lays in the following observation: the averaged voltage of a PWM system, $<V_{out}>$ only depends on the actual number of clock cycles that the switch remains on, or off, during one switching cycle τ_{swc}. (See equation 14.1.3b.) Since we are observing an averaged value, and averages are computed by addition which is commutative, it does not matter in what order the switch was turned on or off as long as the number of on and off cycles is preserved. To illustrate this idea, consider the examples shown below.

First, let us revisit the simple PWM technique from the previous section, see Figure 14.1.3. A reference clock with frequency f_{clock} triggers the switching algorithm, shown in module 14.1.1. The algorithm, based on its input value *x_in*, then determines at each clock cycle whether the switch remains in the open or closed position, resulting in V_{out} (before the low pass filter) either being V_{on} or 0 V.

For example, let us assume (arbitrarily) that the algorithm is fed an input value of *x_in* = 4 and that it uses a 4 bit counter. (The module in 14.1.1. uses an 8 bit counter but such a timing diagram would be too lengthy to draw.) The timing diagram of V_{out} (before the low pass filter) is shown below. Specifically, for the conditions listed, the switch output would remain HI (or on) for 4 reference clock cycles and then LO (or off) for 12 cycles. As long as the input value *x_in* does not change the process repeats itself after these 16 reference clock cycles. In other words, the time it takes for the switching cycle to repeat itself, τ_{swc}, would be 16 (reference) clock cycles. To obtain a meaningful $<V_{out}>$, we would have to average V_{out} for over at least one (or more) complete switching cycles. Since on the average, V_{out} is HI (or on) for 4 cycles and then off for 12, , $<V_{out}> = 4\ V_{on}/12 = V_{on}/3$. (See equation 14.1.3b.)

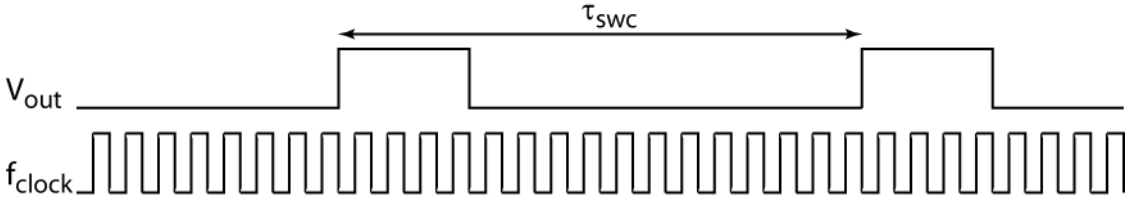


Figure 14.3.2a. V_{out} (before the low pass filter) vs. time for the circuit in Figure 14.1.2.

Let's consider the timing diagram in Figure 14.3.2b. The timing intervals were randomly rearranged within the switching cycle. Nevertheless, the number of (reference) clock cycles during which the switch remained closed , or open, is identical to the previous case: during a complete switching cycle, the switch still remains on for 4 cycles and off for 12 and, therefore, $<V_{out}>$ remains $V_{on}/3$, just as in the previous case.

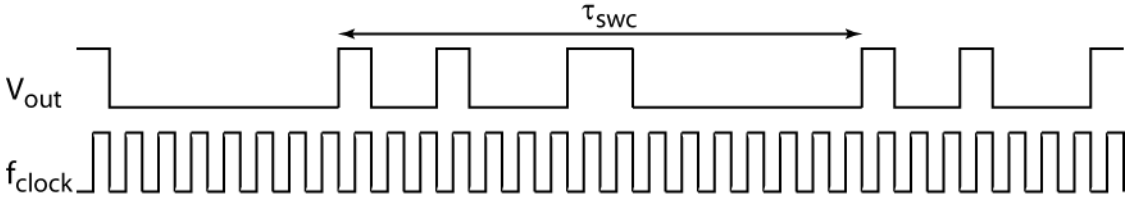


Figure 14.3.2b.

Since the arrangement of the on and off cycles does not matter as long as their ratio is conserved, is there any arrangement that is superior over another one? Consider the time diagram below.

While the number of on and off cycles is identical to the previous cases, note that the switching cycle repeats itself twice within the original switching cycle, τ_{swc}, resulting in an effective switching cycle, $\tau_{swc\ effective}$, of only 8 reference clock cycles.

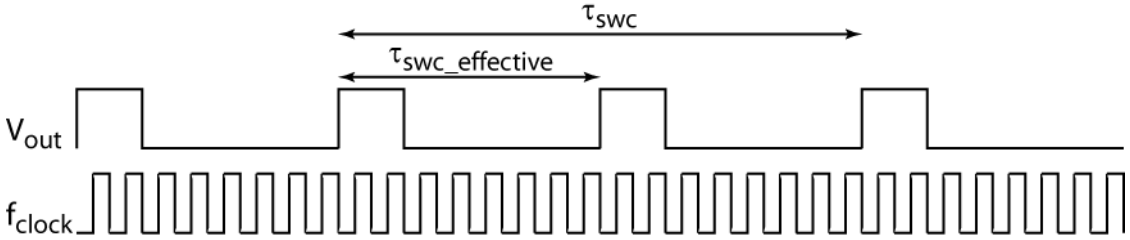


Figure 14.3.2c.

Can we do better than that? Specifically, can we find some algorithm that redistributes the on and off cycles in such a manner that we always obtain the smallest effective switching time while maintaining the ratio of on and off cycles? What method will give us a timing diagram as the one shown below?

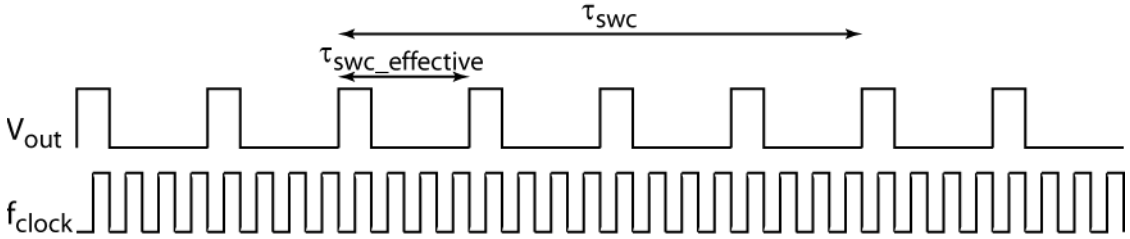


Figure 14.3.2d.

As you probably guessed, applying the Sigma Delta algorithm to the switching circuit of Figure 14.1.3. will accomplish that. The Sigma Delta algorithm will provide us with an effective switching cycle time, $\tau_{swc\_effective}$, that is "usually" far shorter than τ_{swc}. Reducing the switching cycle time allows us to shorten the time interval over which we need to average and, hence, allows us to update *x_in* at a much faster rate.

How much shorter is $\tau_{swc\_effective}$ than τ_{swc}? In the example above, we only saved a factor of 4. That does not seem very impressive but keep in mind that we only had a maximum of 16 clock cycles to begin with because of the 4 bit counter used in the example. Being able to reduce the repetition rate down to just a few clock cycles when we start out with 2^8 or 2^{16} clock cycles for the 8 and 16 bit counters will have a profound effect of reducing τ_{swc}. It is beyond the scope of this discussion but you may assume that for an 8 bit counter 95% of all input values have a $\tau_{swc\_effective}$ of fewer than 10 clock cycles. In other words, the switching cycle was reduced by a factor of 25! To put this in perspective, when we use the system clock of the BASYS board, f_{MAX} can now easily exceed the 20 kHz audio specifications most of the time even in the 16 bit case.

The Sigma Delta technique performs as poorly as the simple PWM for the extreme low or high input values, *x_in*. At these values, the $\tau_{swc\_effective}$ from Sigma Delta algorithm will approach τ_{swc} of the simple PWM switching technique. (For example, for an input value of 1 the switch stays on for only one reference clock cycle; you don't have much freedom in arranging the pulses anymore.) This does not have to concern us as we will place the bulk of the power of our audio signal into the center of the input range. For most of the input range, f_{MAX} far exceeds the audio frequency range specifications listed earlier.

Summary

Compared to the simple PWM from the previous section, the Sigma Delta algorithm preserves the ratio of the time that the switch stays on vs. remains off and, therefore, it will not affect the value of

<V_{out}> or its resolution. Using the Sigma Delta algorithm from section 14.2. and $z = f_{clock}$, if follows that:

$$<V_{out}> = y\ V_{on} / f_{clock} \qquad (14.3.2.)$$

On the other hand, by spreading out the on and off cycles over one entire switching cycle we have decreased the time over which a switching cycles repeats itself.

Exercises

14.3.1. Use the code from exercise 14.1. to dim the LED, including the n-bit counter to divide the 25 MHz system clock frequency by 2^n. See below.

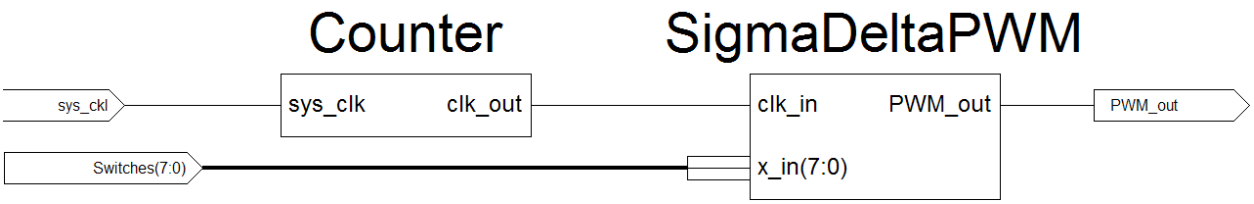


Figure 14.3.4. Dimming the LED using the Sigma Delta PWM technique.

Apply the following modification to your original code: replace the instantiation of the *SimplePWM* module with the *SigmaDeltaPWM* module shown below. (You do not have to type this code in; instead copy the file PWM_Modules.v from the folder U:\pub\Verilog\PWM to your current project directory.)

```
module SigmaDeltaPWM(clk_in, x_in, PWM_out);
        parameter MAXBITS = 8;          //maximum number of bits for input value

        input clk_in;                   //clock for counter
        input [MAXBITS-1:0] x_in;       //control value that defines # of output pulses
        output reg PWM_out = 1;         //PWM signal out

        reg  [MAXBITS:0] Sigma = 0;     //Summing Register, i.e., Sigma

        always@ (posedge clk_in) begin
                Sigma = Sigma + x_in;   //Sigma
                PWM_out = Sigma[MAXBITS];
                Sigma[MAXBITS] = 0;     //Delta
        end
endmodule
```

Code 14.3.1. Note blocking assignment operators were used for the registers in the always block. The code will not work if you use non-blocking assignment operators!

The code in 14.3.1. may look unfamiliar to you but it is just a different implementation of the algorithm shown in Figure 14.2.2. It has been optimized to work for cases where *z* is a power of 2 and, therefore, will work fine with our 8 bit counter. Again use the switches to control the control variable *x_in*. Use the *Counter* module to clock the *SigmaDeltaPWM* module again with the frequency for which you previously observed blinking for the *SimplePWM* module. Do you still observe blinking when you set your switches so their input value is near the center of their 8 bit range, i.e. 128? How about if you change the control value to a very small or large value? Now see to what frequency you can reduce *clk_in* by increasing the value of *n* in the Counter module until you do not observe blinking for an input value of 128. What happens when you use different values for the *x_in*? Change your switch settings and try *x_in* = 1, 2, 3 and 4.

14.3.2. Assign the 8th counter bit to clk_out and observe *PWM_out* on the scope. Describe what happens when your input control value y is: 128, 64, 32 and 127.

Write-up

14.3.1. At what f_{clock} were able to observe the blinking LED when *x_in* = 128? How does it compare to the result obtained in the previous write-up 14.1.4?
14.3.2. Draw the graphs of *PWM_out* vs. *t* for *y* = 128, 64, 32 and 127.

14.4. Audio Player Using Sigma Delta PWM Technique

We are now ready to apply the Sigma Delta PWM technique to build a basic audio player. The audio information has been stored in both 8 and 16 bit words on 2 Megabyte flash memory cards that plug directly into the BASYS board. Since it is easier to work with 8 bit data, we will first implement an 8 bit word audio player. Once it works, it will be straightforward to convert the code to accommodate the flash memory cards with the 16 bit audio data. Using both the 8 and 16 bits cards will allow you to compare the sound quality between the two.

The hardware consists of:

- 2 MB flash memory cards that contains the audio data;
- the BASYS board which will provide PWM circuit;
- an audio amplifier to boost the power of the audio signal;
- a speaker.

These components are already assembled and can be plugged into the BASYS board's 6 pin edge connectors; see below.

14.4.1. Picture on the left: Digilent PModSF 2 Megabyte flash memory module; Picture on the right: Digilent PmodAMP1 Speaker/Headphone amplifier module to boost the audio signal strength.

The Verilog code can be broken down into the following components:

- The 44.1 kHz audio clock module, named *AudioClock* in Figure 14.4.2., provides the trigger signal for reading the audio data from the flash memory modules; the trigger signal also increments the address counter.
- The module named *AddressCounter* specifies the current flash memory address to be read.
- The module *ReadOneByte* reads one (buffered) byte of digital audio data from the *PModSF* flash memory module shown in Figure 14.4.1. The reading is triggered by the enable input and the byte is read from the memory location specified by the address input.
- The module *SigmaDeltaPWM* uses the Sigma Delta PWM technique to convert the digital audio data into an analog output signal that is then be sent to the PModAMP1 audio amplifier module shown in Figure 14.4.1.

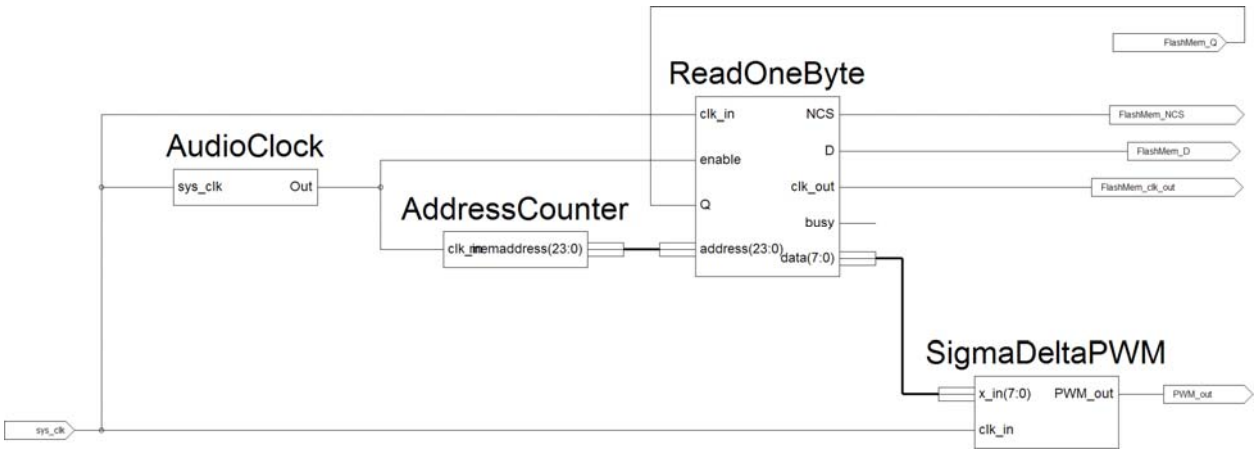


Figure 14.4.2. Complete 8-bit audio player diagram.

The modules for the audio clock and the Sigma Delta PWM technique have already been written or supplied to you in sections 14.2. and 14.3.

The module for reading the audio data from the flash memory, *ReadOneByte,* has already been written for you. It is found in the file FlashMemModulesV6.v. which must be copied from the folder U:\pub\Verilog\FlashMemory to your current project directory. Add this file to your project (after copying it to your project folder) by using the menu bar and selecting *Project* \ *Add Source* and then choosing the FlashMemModulesV6.v file.

The only new module that you will supply is the AddressCounter module. It consists of a simple 24 bit (register) that is incremented at each positive edge of its input signal. Additionally, the counter is reset to zero each time it exceeds the address range of the flash memory, i.e., at Hex 1FFFFFF or, in Verilog notation, 24'h1f_ffff.

After you get the 8-bit version working, make these changes to your code for the 16 bit version:

- Increment the address counter by 2 (instead of 1) at each positive edge audio clock input to account for the two bytes you are now reading at each audio clock cycle.
- Replace the module *ReadOneByte* with the module *ReadTwoBytes*, also located in the FlashMemModulesV6.v file.
- In the *SigmaDeltaPWM* module, change "*parameter MAXBITS = 8;*" to "*parameter MAXBITS = 16;*" to allow for the 16 bit input signals.

Enjoy! (For your own curiosity you may want to see if the previously stated concern about bandwidth was justified and that we indeed needed the Sigma Delta PWM module instead of the Simple PWM. Replace the SigmaDelta PWM module with the Simple PWM from section 14.1 for both the 8 and 16 bit version and see if you can tell the difference.)

What follows is a more detailed description of some of the hardware and software components.

Details

Audio Data: the audio data was created from 8 and 16 bit (mono) WAV files sampled at 44.1 kHz. WAV files are easy to work with for data manipulation because they have not been compressed; the audio information is simply stored in a sequential binary file which can easily be read by a C-program. Unfortunately, since the information has not been compressed, it uses a lot of memory. This is the reason why we are able to play such short audio pieces. (Of course, it would far more rewarding if we were able to use a MP3 compression / decompression algorithm for our data. Unfortunately, these codes are copyrighted and not easily obtainable.)

Flash Memory: after the binary data has been read by the C-program it is offset to the middle of its dynamic range (to compensate for signed values) and then it has been stored (via an RS232 interface and the BASYS board) in the flash memory.

The flash memory can contain up to 2 Megabytes of data and it communicates with the BASYS board through a serial peripheral interface (SPI) using the following 4 pins:

Signal Name	Direction	Purpose	6 Pin Edge Connector J*X* Pin Assignment
Q	Input	Read Data From Memory Line	3
D	Output	Send Data to Memory Line	2
NCS	Output	Chip Select Line (Negative Logic)	1
CLK_OUT	Output	Clock Line	4

Though it does not matter which of the four 6-pin edge connectors (JA through JD) you will be using for the flash memory cards, it is crucial, that once you have selected one, you assign the signals above to the appropriated pins. (See also page 169 in lab manual for BASYS board pin numbers.)

***ReadOneByte or ReadTwoBytes* Modules**: these modules will read either one or two consecutive bytes of data from the flash memory at the address specified at its input. The modules use the following ports:

Direction	Name	Size (Bits)	Description
Input	clk_in	1	Serial interface clock for transmitting data; use the 25 MHz system clock.
Input	enable	1	Hardware Trigger: at the positive edge, the new data is retrieved and subsequently stored in registers "*data."*
Input	address	24	Address where data is retrieved from in the flash memory.
Input	Q	1	Data read from memory is transmitted through this line.
Output	NCS	1	(Not) Chip-Select line.
Output	D	1	Data sent to the memory is transmitted through this line.
Output	clk_out	1	Clock signal from memory; required when receiving data.
Output	busy	1	While HI, flash memory is busy retrieving data; when it is LO again, data has been stored in *"data;"* in our application, this output can be ignored.
Output	data	8 or 16	Depending on version, either 8 or 16 bit register holding the data value read from memory.

Audio Hardware: The Digilent PmodAMP1 Speaker/Headphone Amplifier amplifies low power audio signals to drive either a stereo headphone or a monophonic speaker. The speaker is driven from the left stereo input and works only when plugged into the left input, i.e., the input closest to the round turn pot! The PmodAMP1 module is connected to the BASYS board through any of the four 6 pin edge connectors JA through JD. Use pin 1 for the PWM signal. The round turn pot on the PmodAMP1 module can be used to adjust the audio volume; turning it all the way counter clock wise produces the maximum output power.

Additional information regarding the flash memory and audio amplifier modules can be found here:

Memory:
http://www.digilentinc.com/Data/Products/PMOD-SF/PMod_SF_%20rm.pdf

Audio:
http://www.digilentinc.com/Data/Products/PMOD-AMP1/PmodAMP1_rm_RevB.pdf

Write-Up

14.4.1. Describe the major components (and their function) of the audio player.
14.4.2. Hand in a well documented version of your Verilog code for both the 8 and 16 bit audio player.
14.4.3. What does the difference in audio quality between the 8 and 16 bit samples imply about the human hearing?

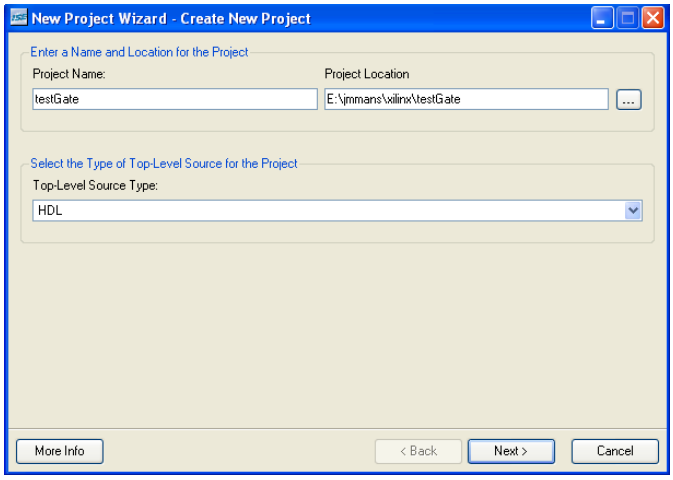
Appendix

J

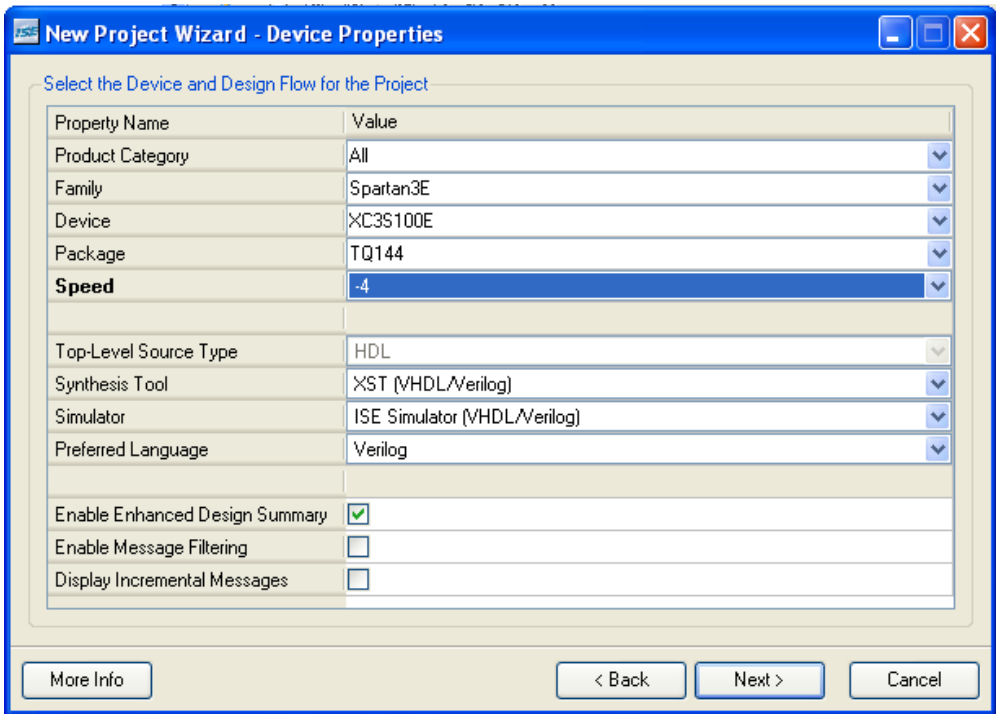
Step-by-Step Guide to Working with the BASYS Board

J.1. Creating a New Project

1. From the Start Menu, open “Progams” → “Xilinx ISE” → “Project Navigator”.
2. When the Project Navigator opens, select “New Project” from the “File” menu.
3. You should create the project in your personal space in a new folder in “My Documents” and give the project a name. Then click “Next”.



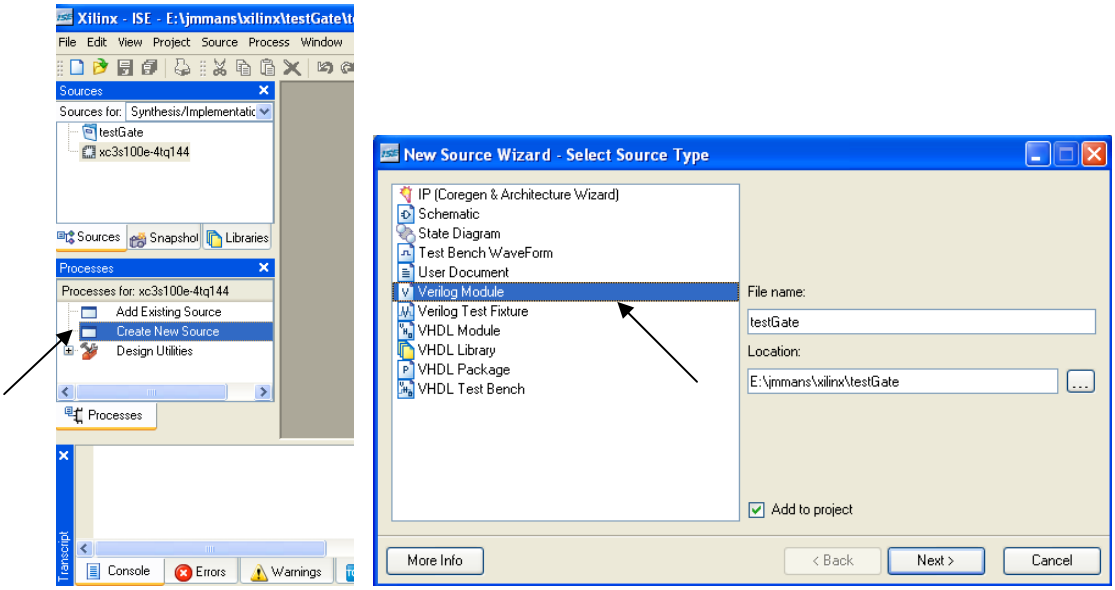
4. The project parameters must be correctly set for the FPGA which you are using. Check to make sure that the “Device Properties” dialog looks like the one below. If you make a mistake, you will not be able to download your program. Press “Next >” after checking this dialog and making any necessary changes.



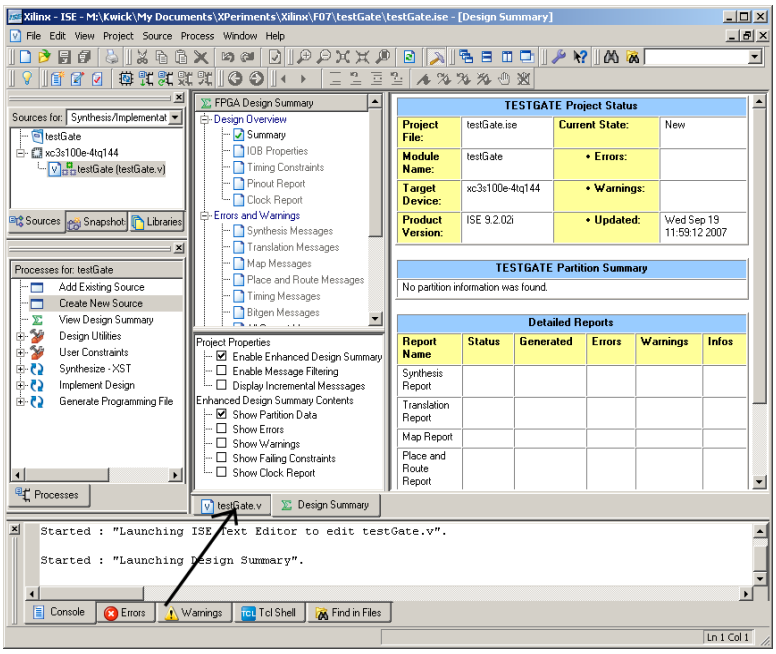
5. Press “Next > “ to skip “Create New Source” and again to skip “Add Existing Source”. Click “Finish” to complete the creating the project.

J.2. Creating a New Source

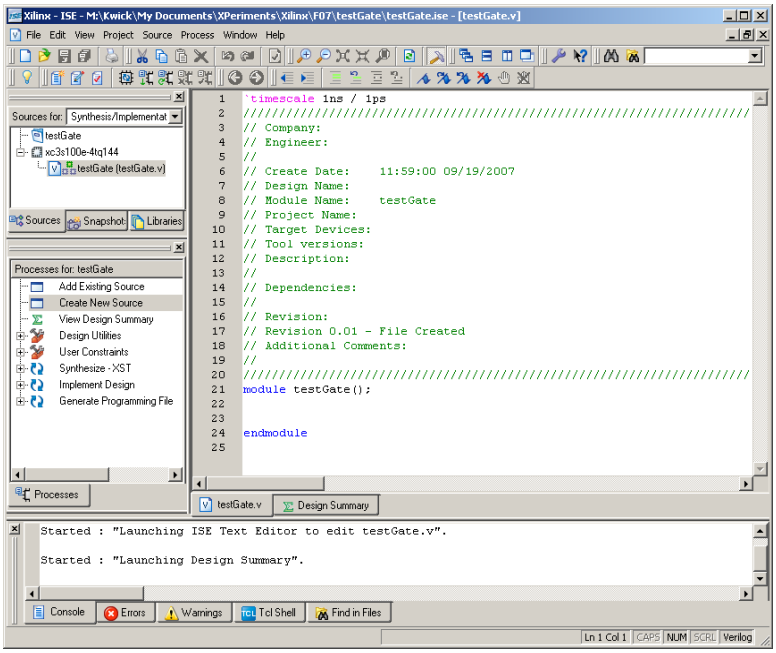
1. To create a new source, double click on “Create New Source” in the process window as seen in the left screenshot below.



2. When the above-right dialog appears choose “Verilog Module” as the type of source to create and enter a file name (which will also be used as the module name). Do not put spaces in the filename. Press “Next > “ to continue.
3. If you wish, you can enter the input and output ports in the “Define Module” dialog. You can always add, remove, or adjust ports afterwards by editing the Verilog file directly. Press “Next >” to continue and “Finish >” to complete the process and you should see a display similar to the one shown in the picture below providing you with a design summary.



4. The previous process generated a “skeleton” source file. To see it click on the tab the arrow points to. See the picture below.



You should now see the source code window on the right. Do not worry if you do not (yet) understand most of the statements.

Since we have not yet specified what exactly the project should accomplish, only an "outline" or "skeleton" source code has been generated to which you then will add the appropriate Verilog statements so that it will do whatever you intend it to do, hopefully "something" useful.

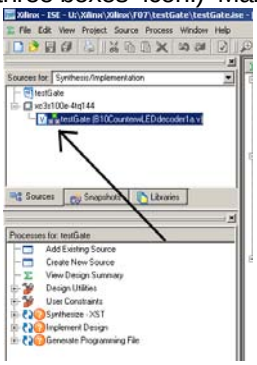
Specifically, note the two commands "module testGate() " and "endmodule" which in the file above are to be found on lines 21 to 24: **it is between these two statements where you will place most of your own code** (at least for now) **and which determines how the FPGA behaves!**

J.3. Synthesis and Downloading of a Project

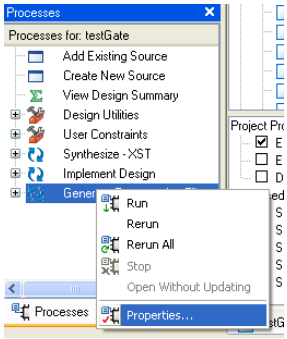
The synthesis process takes awhile, so please be patient! During this very involved process, the software first checks your code for mistakes; next, it finds a way to implement the purpose of your code with the appropriate LUT and muxs and it determines ways to "route" your signal from the various input and output pins in and out of the various FPGA cells. After optimizing it, it will create a "map" or "bit" file which is specific to the particular FPGA employed. Finally, you can then "transmit" or upload this bit file into the FPGA where it will remain until it is overwritten again or the board is turned off or has been reset. Therefore, do not forget to save your code file before you start the synthesis process. System crashes do (sometimes) occur!

1. **Important**: Before beginning this process, you must check the "Programming File" properties.

In the Sources window, shown below, verify that the Top Most module has been selected. (The Top Most module is the one with the "three boxes" icon.) Make sure it is highlighted.

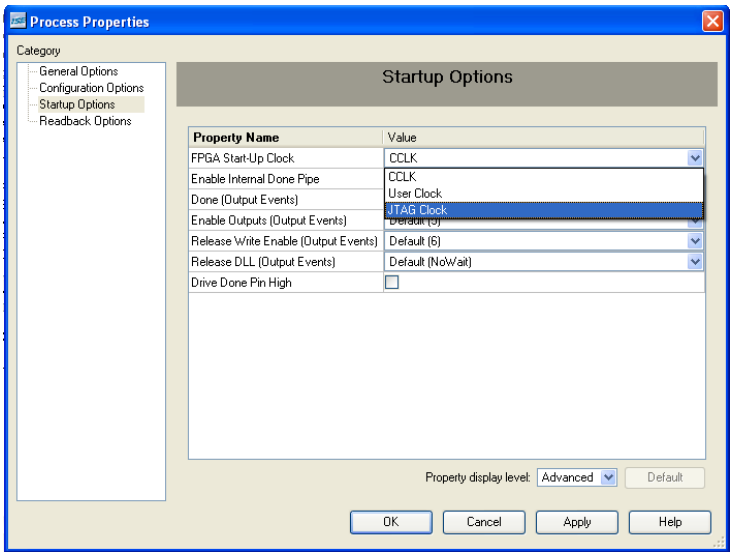


Directly below the "Sources" window, in the "Processes" window, right-click on "*Generate Programming File*" and choose "*Properties*".

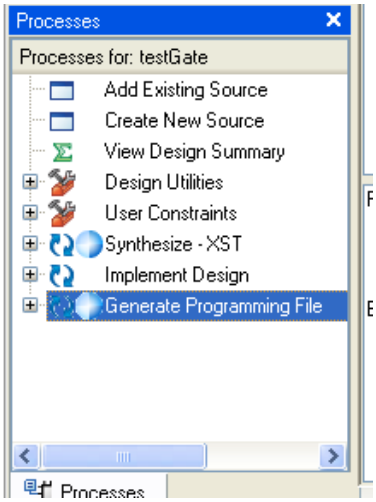


(Note: if the "*Generate Programming File*" option is not listed, then you probably have not selected the Top Module as described earlier.)

2. Choose “Startup Options” and set “FPGA StartUp Clock” to “JTAG Clock”. Click “OK” to save. These two steps will only need to be done once per project.



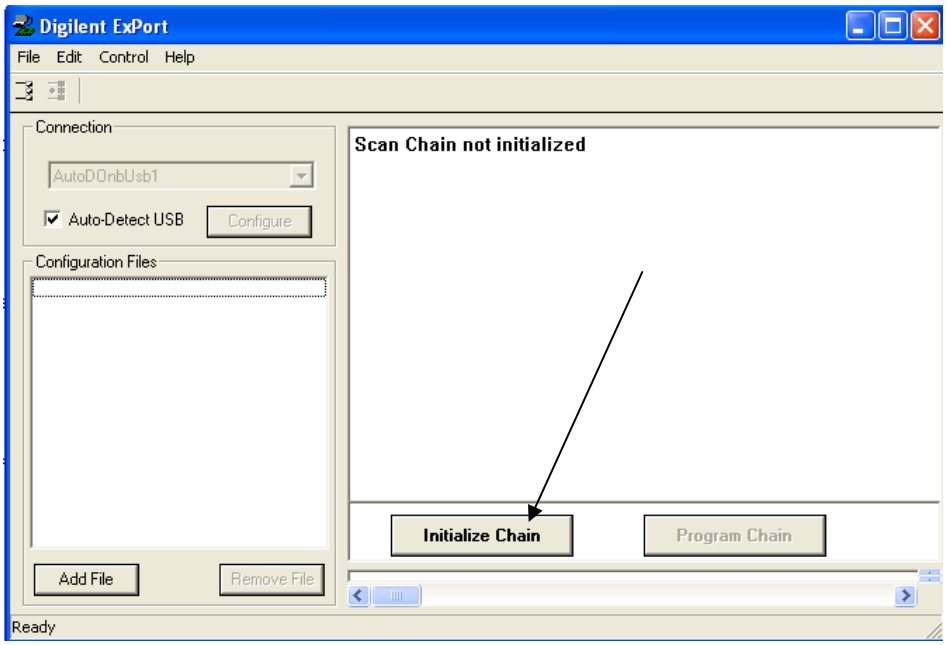
3. Double-click on “Generate Programming File” to start the synthesis process. If there are any errors (such as syntax errors or “typos”), they will appear in the box at the bottom of the screen. Synthesis, implementation, and generation of the programming file may take some time (a few minutes).



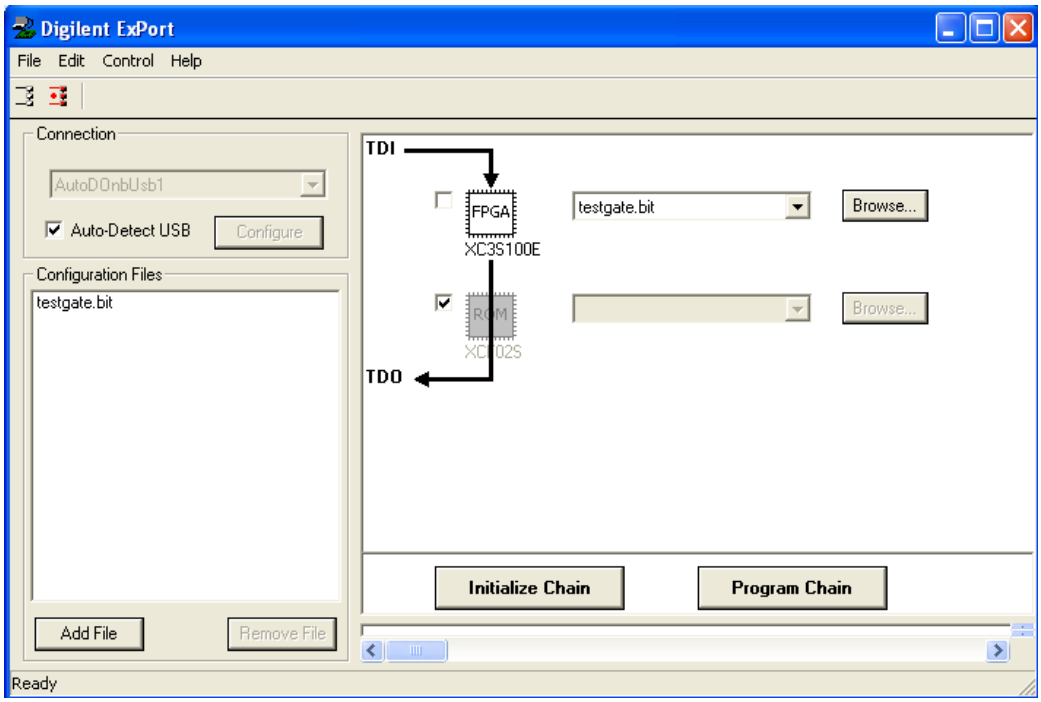
Once it completes successfully, a Xilinx Webtalk dialog window may open. This is way for the manufacturer to collect information on how their products are used. You may decline participating in this by clicking the “Decline” option.

If everything worked ok, you should see the following message in the status window, at the bottom of the Xilinx Navigator: “Process Generate Programming File completed successfully.” Congratulations, you are almost done!

4. When the generation process completes successfully, from the Start menu, open “Programs” → “Digilent” → “Adept” → “ExPort”.
5. Make sure you have connected the USB cable from the BASYS to the computer – the red “power” LED should be on. In ExPort, click “Initialize Chain,” which will connect you to the board.



6. Click on the “check box” to bypass the “ROM”. For the “FPGA”, you should find the “.bit” file which was created when you generated the programming file in Step 4. Press “Program Chain” to download your program!



ExPort will remember your recently used files, which can be very convenient.

J.4. Uploading Program Code into Volatile or Non-Volatile Memory

The previous instructions explained how to upload your bit file to the BASYS board's volatile memory, i.e., to its RAM (Random Access Memory.) However, this implies that as soon as the board is powered off, or reset, the contents of the bit file will be lost. (Of course, the code still exists in your computer account and can be downloaded again through the ExPort facility.) This approach is fine as long as we are learning to program and we do not care to store the programs indefinitely on the boards.

Nevertheless, once you have mastered the basics of Verilog you may want to write code and have it stored in the board's non-volatile ROM (Read Only Memory.) It will reside there indefinitely or until is overwritten by new code. Each time the board is powered up (or reset) it will remember the stored code in the ROM and will begin executing it immediately.

The hardware setting that controls the RAM vs. ROM mode is controlled by the BASYS board's ROM / JTAG jumper. If you want to store the code in RAM set the jumper to "JTAG;" if you want to store the code indefinitely in the ROM, set it to "ROM."

In addition, you must also compile your code for the appropriate setting. Specifically, you must identify the appropriate "Startup Clock." In the previous section, step 2, you were told to set the "Startup Options" for the "FPGA StartUp Clock" to "JTAG Clock." This is the correct setting for storing your code in RAM. However, if you want to store it in ROM you must recompile your code with following setting: in the "Startup Options" set the "FPGA StartUp Clock" to "CCLK." See the table below for a summary.

	Volatile (RAM)	**Non-Volatile (ROM)**
BASYS Board Jumper Setting	JTAG	ROM
Startup Options / FPGA Startup Clock	JTAG Clock	CCLK

J.5. Books on Verilog

Here are some books that provide a more detailed description of the Verilog language:

- M. Ciletti, *Advanced Digital Design with the Verilog(TM) HDL*. (Prentice Hall, Upper Saddle River, NJ, 2002).
- S. Palnitkar, *Verilog HDL,* 2nd ed. (SunSoft Press, Upper Saddle River, NJ , c2003).
- S Lee, *Advanced Digital Logic Design Using Verilog, State Machines, and Synthesis for FPGA's.* (Thomson, Toronto, Ont., 2006).

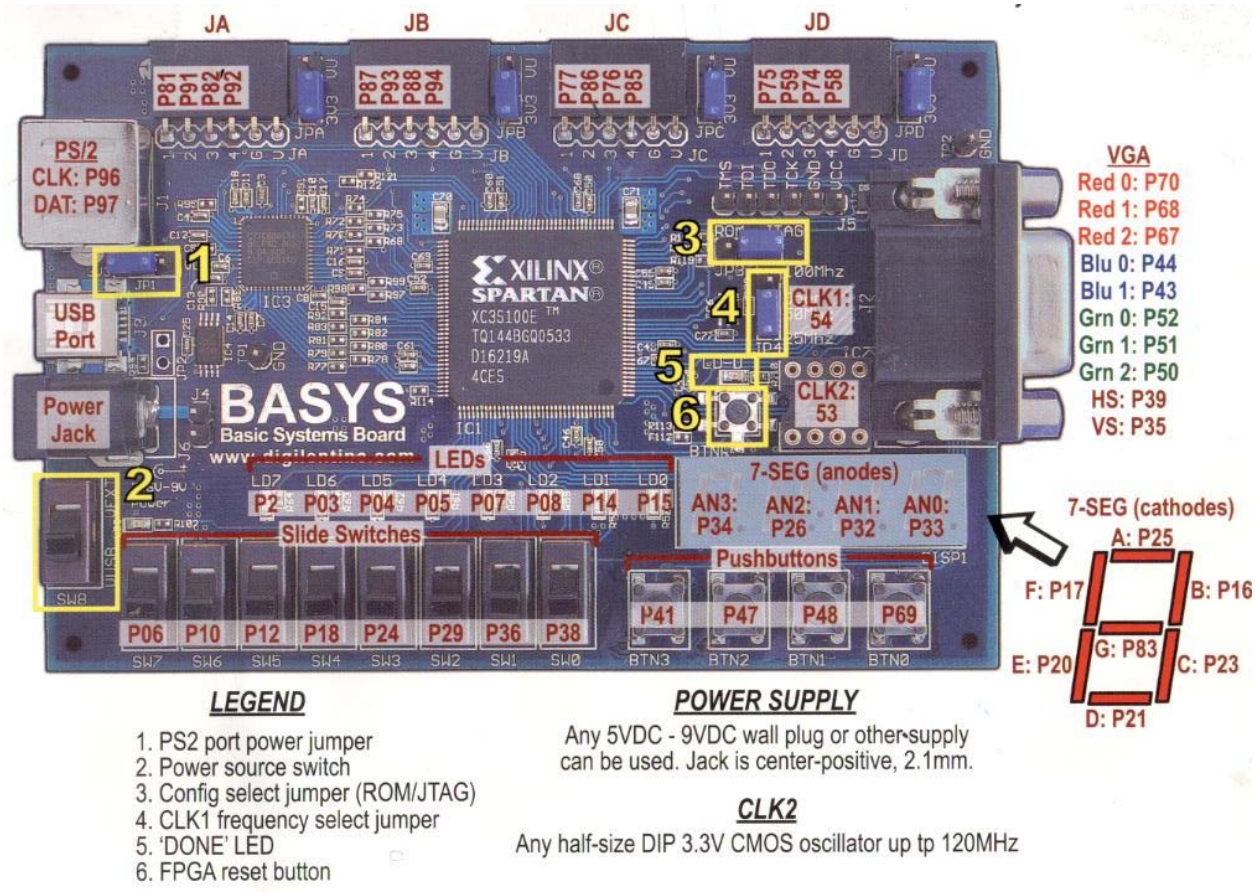
Appendix

L

Digilent BASYS Board Reference Manual

Introduction

The following reference manual and copy right permissions were obtained from DigilentInc at http://www.digilentinc.com/index.cfm



LEGEND

1. PS2 port power jumper
2. Power source switch
3. Config select jumper (ROM/JTAG)
4. CLK1 frequency select jumper
5. 'DONE' LED
6. FPGA reset button

POWER SUPPLY

Any 5VDC - 9VDC wall plug or other supply can be used. Jack is center-positive, 2.1mm.

CLK2

Any half-size DIP 3.3V CMOS oscillator up tp 120MHz