

## EE 469 Lab 1

### Introducing the Lab Environment

*University of Washington - Department of Electrical Engineering*  
*James K. Peckol*

---

#### **Introduction:**

In this first lab project, we have a number of things that we'd like to cover. The first is to introduce (or to refresh our knowledge about) the Altera DE1-SoC board and the Quartus II development environment and tools that we'll be using. The second is to refresh and to extend our working knowledge of the Verilog HDL and three of the four modeling levels supported by the language. The third is to learn the iVerilog modeling tool. The fourth is to learn and to work with an important and very useful tool, the logic analyzer, that is included in the Quartus II IDE, and finally, for now, to begin learning a little about the C language.

We will need to understand C later in the quarter when we develop applications for a processor that we will be designing and implementing over the course of the quarter. Our target platform will be the Cyclone V FPGA and the development environment will comprise the Quartus II IDE and the iVerilog modeling tools.

#### **Prerequisites:**

Familiarity with developing digital circuit models including the understanding of real-world affects and how to incorporate such affects into such models the using the Verilog HDL. A working knowledge of primitive logic devices such as gates, flip-flops, and latches, the logic design of combinational and sequential circuits, the design of finite state machines and the ability to model them using the Verilog HDL, basic software data types and operations, as well as standard program design, development, and debugging techniques. A willingness to learn and to explore. No beer until the project is completed and no food in the lab. I mean it this time.

#### **Cautions and Warnings:**

Never try to run your circuit with the power turned off or the input voltage at a maximum in case you run out. Under such circumstances, the results are generally less than satisfying.

Since current is  $dq/dt$ , if you are running low on current, raise the DE1-SoC board to about the same level as the power supply and use short leads – not short the leads. This has the effect of reducing the  $dt$  in the denominator and giving you more current.

If your circuit is turning on too slowly, raise the power supply so that it is significantly above your breadboard. This enables the charge to get a running start before coming into your board. It will now turn on much faster.

Throwing your completed but malfunctioning implementation on the floor, stomping on it, and screaming 'work you stupid fool, work' is typically not the most effective debugging technique although it is perhaps one of the more satisfying. Clearly it didn't work if you took EE 371...it's not going to work here either.

As noted earlier, when you are debugging you circuit, wiring it, taking it apart, and rewiring again several dozen times (or in the case of the FPGA, recompiling the code and reuploading it) does little to fix what may be a design error. Such an approach is not highly recommended, but, can keep you entertained for hours....particularly if you can convince your partner to do it.

Sometimes - but only in the most dire of situations – sacrificing small animals to the smoke demon living in your circuit does work. Reports from the field from former graduates suggest some success with such an approach...”wanted to let you know that I’ve sacrificed many small animals to get many reference designs running”. However, these are not included in your lab kit and must be purchased separately from an outside vendor. Also, be aware that code gremlins are not affected by such sacrifices.

Alternately, blaming your lab partner can work for a short time...until everyone finds out that you are really to blame.

Always keep only a single copy of your Verilog or C source code. This ensures that you will always have a maximum amount of disk space available for games, email, and some interesting pictures. If a code eating gremlin happens to destroy your only copy, not to worry, you can always retype and debug it again.

### **Debug Tips:**

Adapted from Robin Panda.

One of the most important skills you will learn and practice in this class is how to debug your design. As mentioned above, completely disassembling and reassembling your circuit isn’t helpful. It’s entirely possible that you will repeat the same mistake that your design is incorrect, or your power supply is off, or you’ve misread a chip’s part number and installed the wrong one. The ability to isolate your own problem will only pay off more later with more advanced designs when your instructors know nothing about the details of your design.

The most useful question that you can ask yourself is, “**What DOES Work?**” You’re already asking yourself, “What doesn’t work?”. If that question were easy to answer, you’d be done. This means a number of things....

Check the easiest things to check (and common mistakes) first. Do not assume things work unless you have at least some evidence so. Does the power supply actually put out the proper voltage (at your circuit)? Is the scope set to the right input, voltage, and mode? Did you get any warnings or errors when you compiled? Does your circuit always get really hot in the same place? Are there wires or semicolons laying about that shouldn’t be? Is your circuit actually a burrito and not your circuit? (don’t bring food in lab)

Ask yourself, When did it work last? If the answer is never and you just built a huge nest of wires with a book full of code and are expecting it to work correctly, you are risking pain and you’d better be lucky. Once you have this answer, you can temporarily assume that the old stuff that hasn’t changed is still working and now you have a much smaller circuit (or piece of code) to debug.

What is the smallest suspect (chip, module, component, statement, etc) that is closest to the known good bits? Stick a probe (scope, print statement, dump) on the output of this and verify. If it looks what you expect, add it to your list of probably good components and

repeat. ***This is a key point***...In order to see if what you see matches what you expect...you must know what you are expecting to see...sounds simple, but many people skip this part. If it doesn't, then maybe you've isolated the problem. Verify that the inputs to this piece are correct to be sure. If you can't verify the inputs, you chose the wrong component to test. Sometimes you need to make test code and/or test circuits to simplify the behavior to be able to verify. Always test closest to the inputs and known good bits. This means your Verilog needs to be correct before you worry about your circuit or claim your FPGA is broken.

Once you've got something, it doesn't mean that that component is broken. You also could be using it wrong, or chosen the wrong one. Make sure the component isn't it backwards. Make sure that you have connected power and ground. Understand the code your partner wrote. Double-check the datasheet to make sure it's specifications are what you expect. See above on expecting things...Look for weaknesses: Is there a maximum output current that could explain why you're stuck at 0? Is some signal active-low?

With experience, you will add more simple issues to the list of first checks and often find the problem before needing to be formal about your debugging. Eventually, your experience or intuition will fail you and you will need to check everything. I hope it's clear that the reason you're debugging is that one of your assumptions is wrong; you need to be a skeptic and think critically. Sometimes what was previously good got messed up and you need to revert to the old circuit/code to check. Sometimes you missed the part of the datasheet that says gravity pulls up. Sometimes that warning that you "always get" says something slightly different than you thought it did. Sometimes your TA or lab report gave you dodgy advice that makes no sense...

## **Background:**

In this class, we will continue to work with the Altera tools and development environment on the DE1-SoC platform that you started in EE 271 or EE 371. If you took EE 271 before we introduced the DE1-SoC board, not to worry, it's quite similar to the olde DE1. The platform is new and slightly different from what we've used previously. We've learned a lot for our work in earlier classes, however, there is still much to learn and we hope to take several more steps in that direction this quarter. We also want to continue to grow our knowledge about the formal development process that we will continue to apply to our projects in the quarters ahead. Learning new components and tools is exciting and challenging. This is the fun of engineering.

At the same time, sometimes it can be frustrating when the hardware or the software doesn't behave the way we want it to, or when, unfortunately, it behaves exactly the way that we have told it to, or when we can't immediately find the answer to our questions and problems.

At this stage in learning engineering and about the engineering process, playing with the toys often seems to be the most exciting part...the documentation, formal design, and so on, is, well, often seen as rather boring. *Why do I have to go through all this...why can't I just go to the Internet and find something like the design and make a few modifications and be done?*

In the real-world, the documentation and formal design are absolutely critical and essential parts of all phases of the engineering development process. Doing it and doing it right can mean the difference between success or failure of a project, the malfunction of the system following delivery to the customer, or the possible loss of life while it is being used.

It's also very important to recognize and to remember that the answers to most interesting real-world engineering problems originate in our brains, discovered as we use our imaginations and knowledge to creatively apply the underlying theory; they are reached through our persistent hard work and diligence. The solutions to challenging problems are not sitting, ready, and waiting for us on the Internet. The Vikings didn't discover North America by searching some ancient version of the Internet...they took risks...they explored and challenged. We didn't find the solution for making the first successful airplane flight, to putting someone on the moon, to making the first soft landing on Mars, or to programming the first microprocessors, or the possible discovery of the Higgs boson on the Internet...we won't find the answers to many of today's problems or the labs this quarter there either. We challenge you to explore, to think, and to make those discoveries.

Sometimes your instructor or TA will have the answers to your questions and sometimes they won't. As we said in the opening, bear in mind that we're continuing to learn the environment and tools too – this platform and development environment are complex and are a challenge (and opportunity) all of us. If we all work together, to identify and solve problems as they occur, everyone benefits and we help to build for the next classes. The answers are there somewhere. Let's all work to find them.

Regardless of the specific platform/environment used, systems development requires at least the following:

1. An environment in which to develop the design – hardware or software.
2. A target environment on or in which to execute the design – this can take many different forms.
3. A mechanism for moving the design from the development environment onto the target environment.

*Target platform* or *environment* are terms used loosely in industry to mean the actual piece of hardware that performs the computation and/or control – the place where some application will ultimately run. Here, we will be utilizing pieces of hardware on a piece of hardware – a Cyclone V FPGA (Field Programmable Gate Array) and a collection of supporting peripheral devices. Our target platform will be the FPGA...our hardware development will parallel the methodologies that we use for software development.

At the end of the day, a target platform can be a simple a single chip (i.e. an Arduino or PIC microprocessor), or as complicated as a feature-rich single board computer (built around a multicore high performance processor and possibly several programmable logic devices)...or we could be designing that multicore system as the heart of a larger system. Here, we'll work with one such programmable device. Building upon and extending what we learned in EE 271, we'll find that we can still do some really cool stuff with it. Just watch.

Despite the differences in physical characteristics, the target platform's purpose is the same: to support the design created by the developer that is intended to do some job. Here, and for the typical PLD (programmable logic device) based application, the hardware development/modeling language will be Verilog or VHDL, the application software may be written in the C language, in an assembly language (which implements the C constructs), or the machine language of our machine. The target will be a machine of our design, implemented on an FPGA.

In this context, we will use the Verilog HDL to *model* the hardware functionality of our system in the Quartus II environment. We will then *synthesize* the HDL model into the hardware pieces necessary to affect that functionality on the FPGA. As we implement our target processor, we will follow the same hardware development process, then, we will download a program to run on that processor which we put onto the FPGA.

A final piece of advice from John Fluke Sr. – founder of the Fluke Corporation to take with you. He built an international billion dollar corporation around these words.

Always give the customer a little more than they expected.

It's a good philosophy no matter what you are working on or where you are working.

### **Laboratory Environment:**

Our laboratory environment comprises the DE1-SoC development board and the Quartus II IDE. The combination is a feature-rich set of tools that includes, on the hardware side:

- The Cyclone V FPGA,
- Several DDR RAMs,
- A synchronous dynamic RAM (SDRAM),
- A Flash memory
- SD Card socket
- VGA support
- CODEC
- PS/2 mouse / keyboard I/O
- User I/O in the form of switches and LEDs
- A hard ARM processor and support for a soft NIOS II processor
- Three different oscillators for clock development

and, on the software side, a development environment that supports, among other things

- Graphical, HDL, or mixed design entry
- Simulation capability
- Synthesis of the HDL model for the target environment
- Multiple different views of the design
- Timing constraint analysis
- Logic analyzer capability
- A rich library of parts
- Support for user design components

In this class, we will use most of these hardware and software features as we design and develop the various subsystems in our machine; the details regarding each of the components will be given as we need them.

Using the *Quartus II Development Environment*, Verilog models and C applications can be developed on a standard PC and then downloaded onto the target platform over a USB serial connection.

## Objectives:

The major objectives of this project include:

- Introduce the laboratory environment: the development environment, the target platform, host PC, equipment, etc.
- Begin to work with the various Verilog modeling levels through the design, test, synthesis, download and execution of several different counter implementations.
- Learn the Verilog modeling tool *iVerilog*.
- Learn and use the Quartus II *Signal Tap* logic analyzer to examine the output waveforms from the various counters.
- Learn a bit (more) about the Cyclone V FPGA.
- Begin to learn and work with a formal approach to design beginning with identifying the requirements and then formulating specifications.
- Begin to work with the C language, simple C programs, and the PC development environment.
- Learn to design, compile, and debug a C program on a host PC.

## Designing and Building Verilog HDL Applications:

For the first part of the lab, we will design, build, test, and synthesize several different counting circuits that we will then program onto and run on the Cyclone V FPGA. For each of the following designs, we will use Verilog as the top level. In a later part of this lab, these will serve as our signal source as we begin to learn and to work the Quartus II, Signal Tap logic analyzer.

The four counters are specified as follows. Each of the counters should reset to the all 0's state. Following the first clock, each down counter will move to the all 1's state and continue counting down from there.

1. A four stage (4 bit) ripple down counter, with active low reset, using a *gate level model* and the following D flip-flop model.
2. A four stage (4 bit) synchronous up counter, with active low reset, using a *dataflow level model* and the D flip-flop model given in part 1.
3. A four stage (4 bit) synchronous Johnson counter, with active low reset, using a *behavioural level model*.
4. A four stage (4 bit) synchronous up counter, with active low reset, using *schematic level entry*.

Refer to Appendices B and C in the Brown and Vranesic text, 2<sup>nd</sup> ed. On the class web page under documentation.

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

## Working with iVerilog and gtkwave

You can download and install Icarus Verilog on your personal machines. If you do, make certain that the path to the installation does not contain any spaces. The same is true for any file names.

Design and test each of the first three counters using iVerilog. Implement each design as two simple files, one containing only the counter module and a second, top-level file, containing the test bench and tester for that counter. Include the counter module into the top-level file using the directive

```
`include "fileName.v"
```

where *fileName.v* is your counter file and ` is the back tic, not the apostrophe '.

Verify the proper operation of each design using both the state and waveform (via gtkwave) outputs.

Working with iVerilog and gtkwave as design tools is rather straight forward. The following steps will get you up and running in short order.

Let's work with the simple system modeled by the multifile Verilog programs, *andorTop0.v* and *andOr0.v*, that follow.

```
// andorTop0.v

`include "andOr0.v"

module testBench;

    // connect the two modules
    wire [1:0]    X, Y;
    wire [1:0]    XandY, XorY;

    // declare an instance of the AND module
    AndOr  myAndOr (XandY[1:0], XorY[1:0], X[1:0], Y[1:0]);

    // declare an instance of the testIt module
    Tester  aTester (X[1:0], Y[1:0], XandY[1:0], XorY[1:0]);

    // file for gtkwave

    initial
    begin
        $dumpfile("andor0.vcd");
        $dumpvars(1,myAndOr);
    end

endmodule
```

```
// andorTop0.v cont.

module Tester (xOut, yOut, XandYin, XorYin);

    input  [1:0]    XandYin, XorYin;
    output [1:0]    xOut, yOut;
    reg  [1:0]      xOut, yOut;

    parameter stimDelay = 20;

    initial // Response
    begin
        $display("\t\t xOut yOut \t XandYin XorYin \t Time ");
        $monitor("\t\t %b\t %b \t %b \t %b", xOut, yOut, XandYin,
                                                    XorYin, $time );
    end

    initial // Stimulus
    begin
        xOut = 'b00; yOut = 'b10;
        #stimDelay xOut = 'b10;
        #stimDelay yOut = 'b01;
        #stimDelay xOut = 'b11;

        #(2*stimDelay);           // needed to see END of simulation
        $finish;                  // finish simulation
    end

endmodule
```

```
// andOr0.v

// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
output [1:0]  AandB, AorB;
input [1:0]   A, B;

    and    myAnd [1:0] (AandB[1:0], A[1:0], B[1:0]);
    or     myOr [1:0] (AorB[1:0], A[1:0], B[1:0]);
endmodule
```

1. Create a Verilog source file for the circuit or system that you are modeling using Notepad++ or your favorite editor. Above we have the files *andorTop0.v* and *andOr0.v*. As above, to provide the data for the gtwave tool, you must structure your test bench as...

```
module testBench;

    // connect the two modules
    wire inputs;
    wire outputs;

    // declare an instance of the MyDesign module
    MyDesign    myDesign(outputs, inputs);

    // declare an instance of the Tester module
    Tester      myTester (outputs, inputs);

    // file specifications for gtwave
    initial
    begin
        // dump file is for dumping all the variables in a simulation
        $dumpfile("gfxFile.vcd");

        // dumps all the variables in module myDesign and below
        // but not modules instantiated in myDesign into the dump
        // file.
        $dumpvars(1,myDesign);
    end

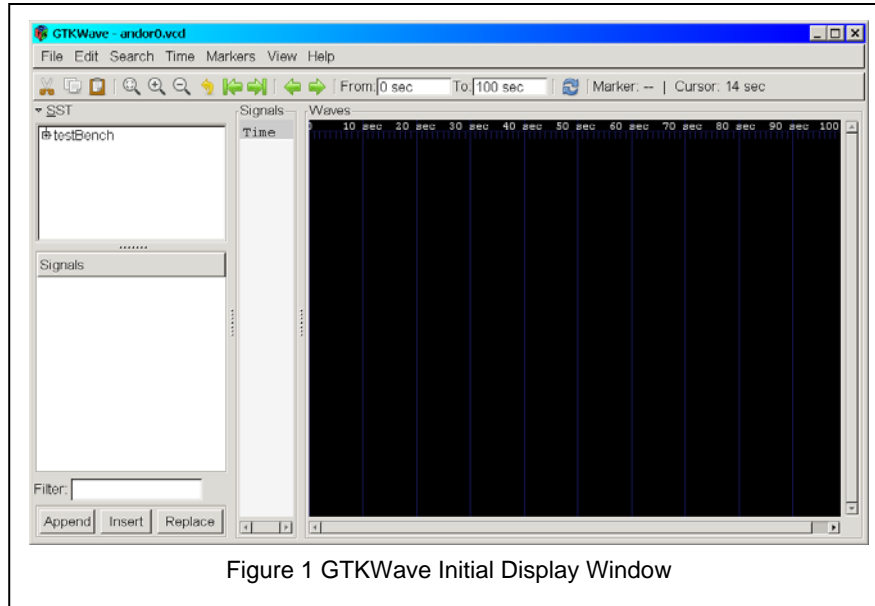
endmodule
```

1. On your PC, go to the *Start* button and select *run*. Enter *cmd* and navigate to where your source file is.
2. Create an executable by typing  
`iverilog andorTop0.v`  
 which produces a file named *a.out*  
 or specify the name of the output file by typing  
`iverilog -o andorTop0 andorTop0.v`  
 which produces a file named *andorTop0*
3. Run the executable by typing either  
`vvp a.out`  
 or  
`vvp andorTop0`



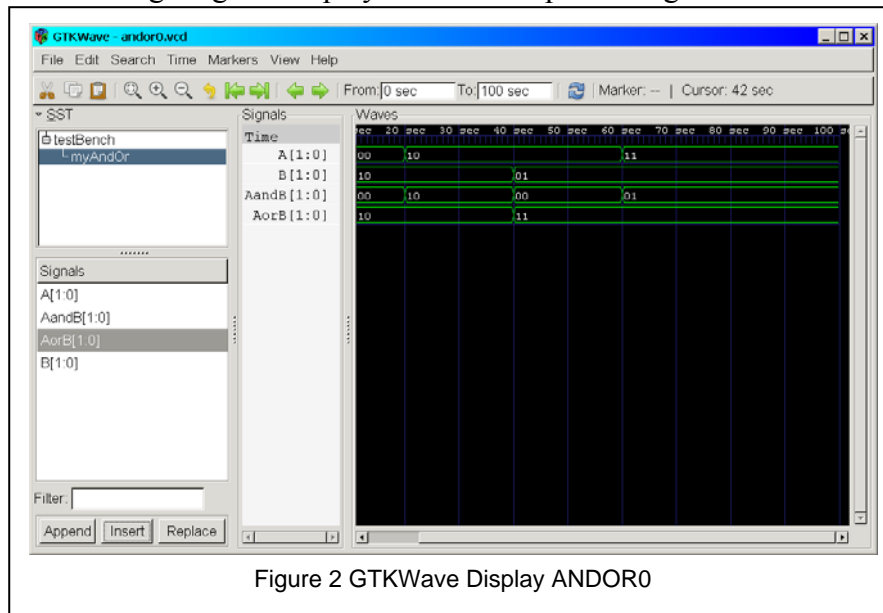
as appropriate. In addition to executing your program and producing a state output, the vvp tool will also produce a file with a .vcd extension that we use as the input to gtkwave. If we named the dumpfile above *andor0.vcd*, this is the file that the vvp tool will produce.

4. To run the gtkwave tool to get the timing diagram, type  
`gtkwave andor0.vcd`  
to create the initial display window,



Select and expand *testBench* then *myAndOr* to provide a list of signals that can be displayed. Choose then click on a displayed signal then select *Insert* to add that signal waveform to the timing diagram.

The following diagram displays all of the top-level signals for the module instance



*myAndOr.*

### Working with Quartus II and the DE1-SoC Board

We will now move to the DE1-SOC board and work with the Quartus II environment. As a refresher on these tools, Appendices B and C in the Brown and Vranesic 2<sup>nd</sup> edition text from EE 271 provide an excellent getting started guide. There is also some very good introductory material from Altera on the class webpage under the documentation/QuartusDocumentation subdirectory (Class Documentation hot button on the front of the class web page).

For this phase of the project, use the LEDs on the DE1-SoC board to display and verify the counter outputs and one of the switches to generate the reset input.

To develop a clock for each counter, start with the 50 MHz oscillator - ***Do not use one of the switches on the DE 1 board as the clock to your counters*** - and divide the signal down sufficiently that you will be able to see the counter outputs change.

The following Verilog code fragment illustrates one way to do this.

```
module MyModule(....., iClk, .....);
    input iClk;                                // connect to system 50 MHz clock
    reg [21:0] tBase;                          // system time base
    always@(posedge iClk) tBase <= tBase + 1'b1;
    .....
    always@ (posedge tBase[20])                // select a frequency
    .....
endmodule
```

Create separate projects using the Quartus II IDE for each of the first three counters. Add the counter module Verilog file to the respective projects, synthesize, and then download each onto the DE1-SoC board. Verify the proper operation of each using the LEDs on the DE1-SoC board.

Create the fourth counter using a schematic entry project using the Quartus II IDE. Make the top level module a Verilog module. Verify the proper operation of the design using the LEDs on the DE1-SoC board.

### Learning the Tools – The Signal Tap II Logic Analyzer:

In this portion of the lab we will work with a very handy tool for testing, debugging, and troubleshooting complex digital circuits and systems called a *logic analyzer*. It gives us the ability to simultaneously examine the extended temporal behavior of a large number of signals. Such a task would be impossible using traditional tools such as an oscilloscope or voltmeter or would require some very fancy probe work.

In the Quartus II environment, the logic analyzer is available as a software tool. Typically, however, it appears alongside our other stimulus and measurement tools as a piece of hardware. None-the-less, we will still be able to control and make many of the same measurements that we would using the hardware instrument.

There is a very good introductory tutorial from Altera on the Signal Tap II on the class webpage under the documentation/QuartusDocumentation subdirectory (Class Documentation hot button on the front of the class web page).

1. In the Quartus II IDE, setup and configure the Signal Tap II logic analyzer to display the inputs and outputs of each of the four counters you designed in part 1 as they are running on the DE1-SoC board. Configure the Signal Tap II to trigger on the trailing (rising) edge of the input reset signal.
2. Configure the Signal Tap II to trigger on, capture, and display data following the third state in each of your counters.
3. How do the displayed signal outputs compare with the ideal signals?

### Learning the C Language – The First Steps:

We will conclude this lab project by beginning to learn the first steps of the formal engineering design process and about working with the C language and the Visual Studio development environment on a PC. Alternately, you can download and install the open source Code Blocks environment. If you elect to use Code Blocks, download and run the file *codeblocks-12.11mingw-setup.exe* from the *binary release* download page. This will bring in the GCC compiler and GDB debugger. This IDE works well and is straight forward to use.

We will first work with the development environment. To do so, we will start with a known good C program. Then, we will work through the first steps of the development cycle with a program that we write.

### Working with C in the PC Environment

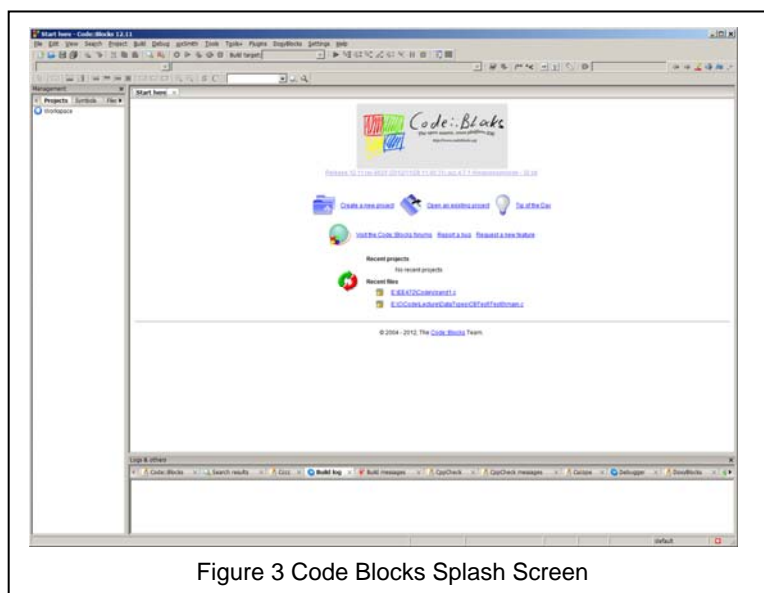
#### Building a Program

1. Using CodeBlocks on your PC, create a project then add the C file project0.c. You can find project0.c in the Lab1 folder on the class web page.

Select

- File →New Project

2.



In the pop-up window select:

- Next

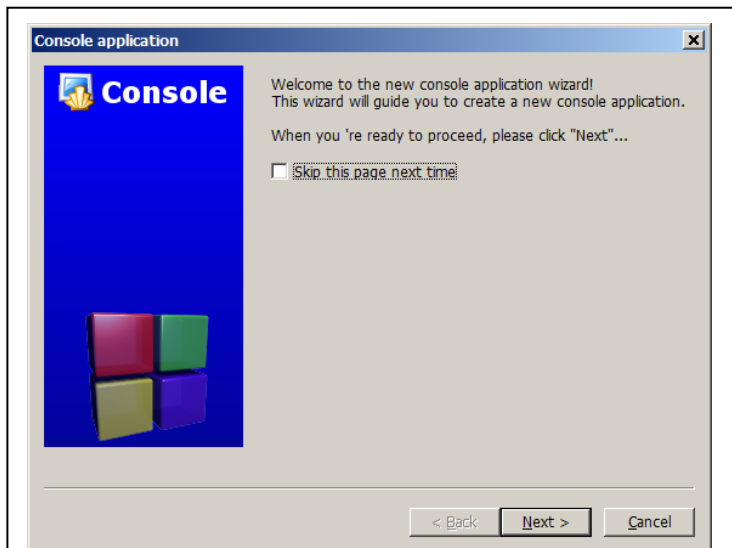


Figure 4 Console Application

3. In the pop-up window select:

- Console Application

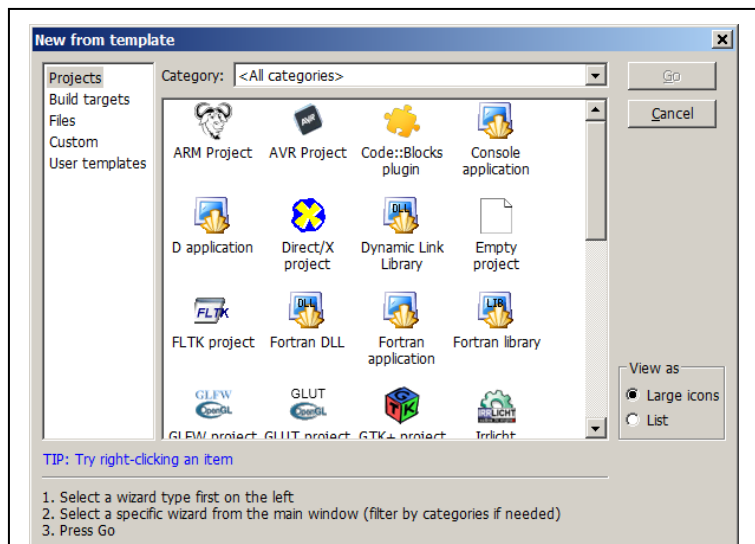


Figure 5 Project Type Selection

4. In the pop-up window select:

- C then Next

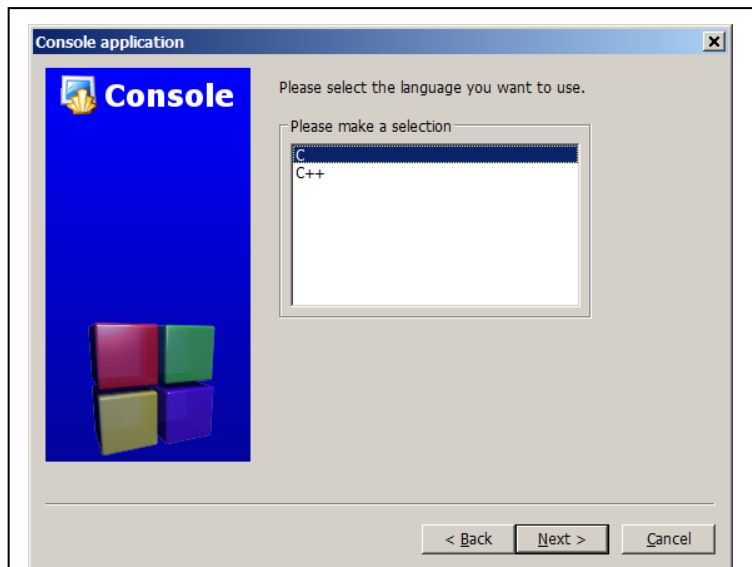


Figure 6 Implementation Language Selection

5. In the pop-up window

- Give the project a Title, Location, and Name
- Select Next then Finish

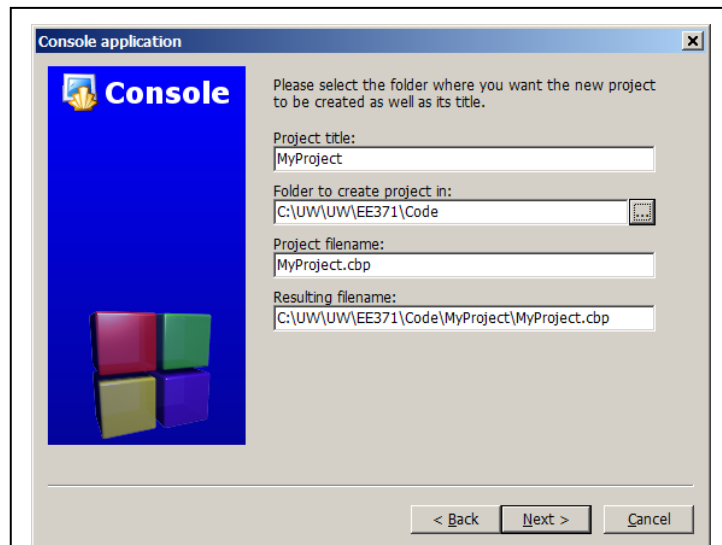


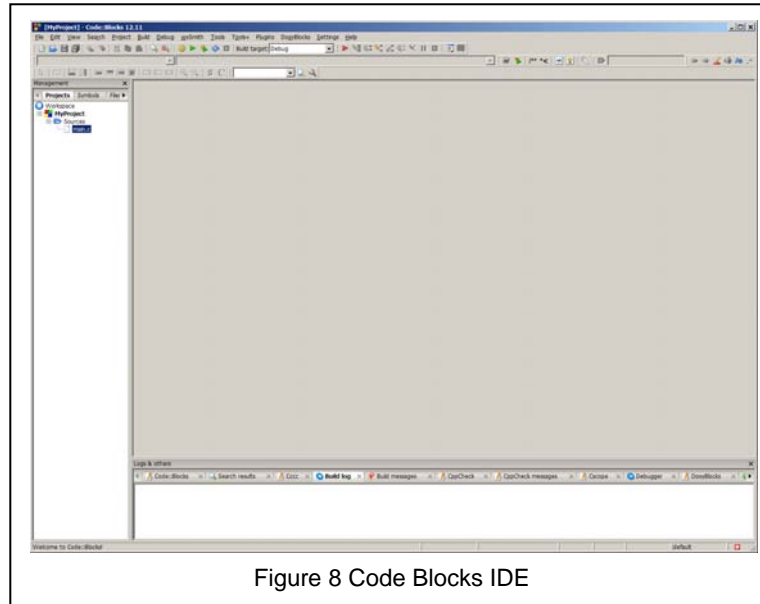
Figure 7 Project Directory Selection

6. In the in the left hand pane of the workspace window that pops up

- Select main.c

Using the right mouse button, select

- Remove this file from project



7. Using the right mouse button, select MyProject

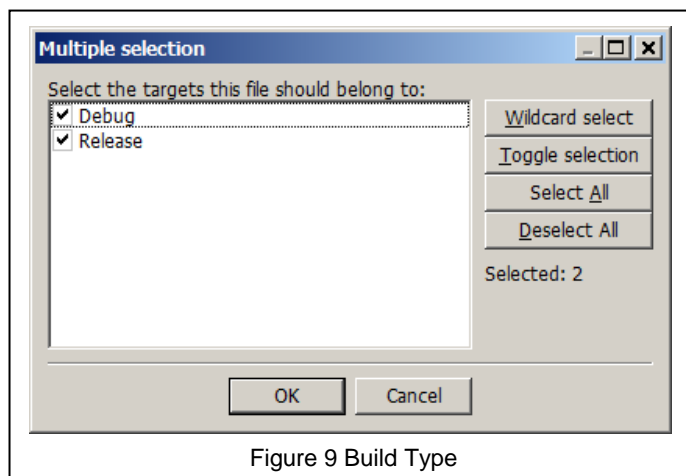
- Select Add Files

8. In the window that pops up, browse to where you have saved project0.c.

- Select the file and select Open to add it to the project.

9. In the window that pops up,

- Select OK



10. If you double click the file project0.c, you should now have something like...

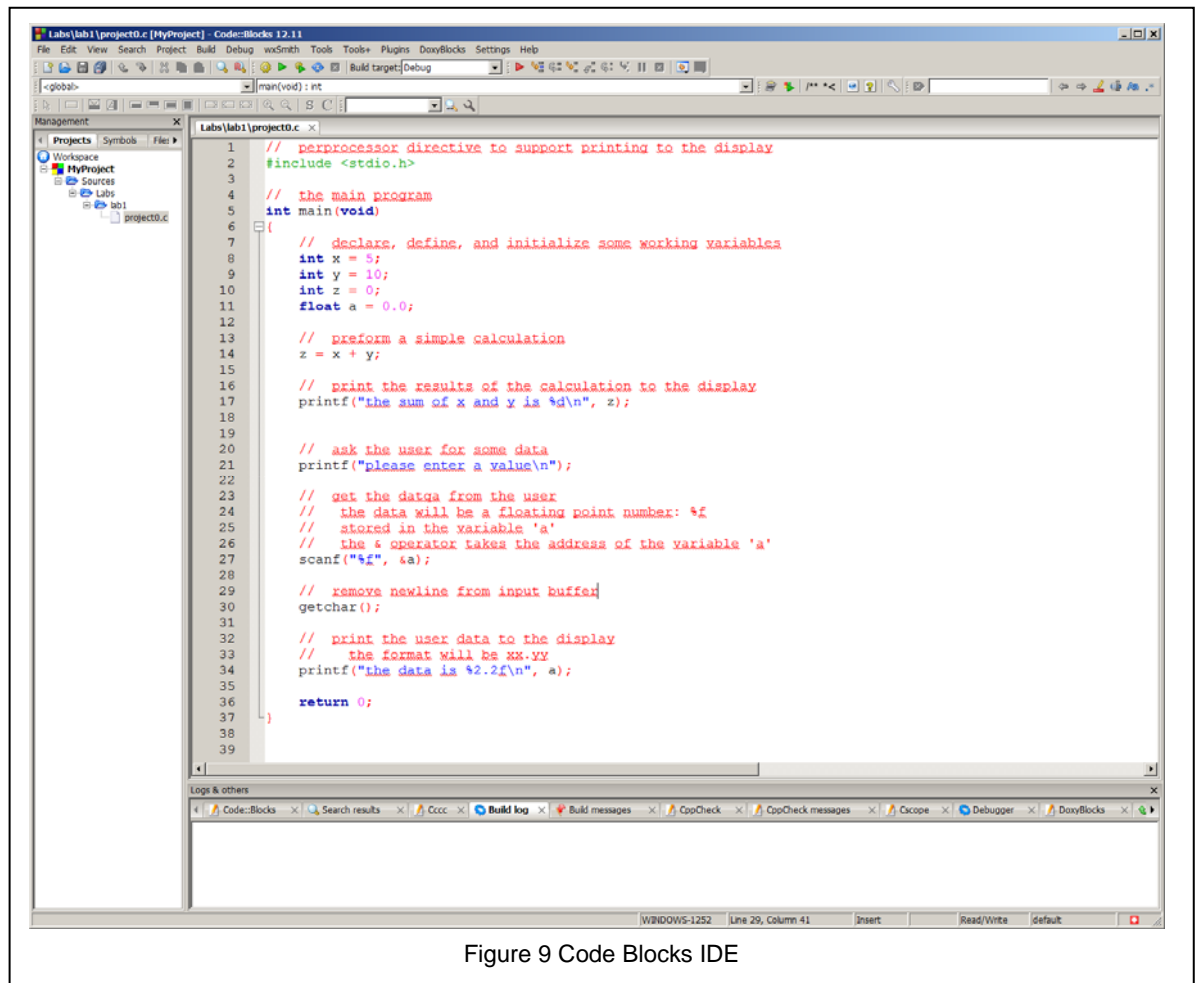


Figure 9 Code Blocks IDE

11. Now compile and execute the program.

- Select Build → Build or press Ctrl-F9

There should be no errors – if there are, then bad copying.

To run the program, press Ctrl F10

The program should print the first two lines and halt waiting for input from the user.  
The results printed for the value of z may be different for you project.

Here the number 23.4 is entered and the remaining lines are then printed.

```
the sum of x and y is 15
please enter a value
23.4
the data is 23.40
Press any key to continue
```

## Creating Your Own Design

It's that time of year again and you are out looking for a job once you graduate. When you start working, in addition to your federal income tax, you may have to pay state income tax as well. So that you can compare your net possible income based upon offers from companies in different states, you decide to write a simple net income program.

Some of the things that you decide to include in your program are: prompt for your starting salary offer and, if the state has a state income, prompt for the state income tax rate as a percentage of your gross.

You have determined the following information from the IRS publications: you will have to pay 10.3% of the first \$65,000 of your income in social security before taxes; in your potential tax bracket, your federal tax will \$3,500 plus 28% of all income over \$30,000.

When executing a formal design, we typically go through what is called a *development cycle* during which we produce, among other things, the following:

1. A short requirements document for your program. The document should have the following sections:

- ✓ Abstract
- ✓ Introduction
- ✓ Inputs to your program or system
- ✓ Outputs from your program or system
- ✓ Major Functions

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

Each major function should have a brief textual description of what it is and its behavior.

2. A design specification for your net income program. The document should have the following sections:

- ✓ Abstract
- ✓ Introduction
- ✓ Inputs to your program
- ✓ Outputs from your program
- ✓ Major Functions

Each input and output should now also include, as part of its description, the range of legal values and a brief textual description of what possible errors might occur and their consequences.

3. In our case, a C program that implements all of the features and capabilities specified in your design specification for your program.



**Deliverables:**

A list of ten uses for a computer...a 25 point bonus for the most creative use of a computer as judged by your TAs and your instructor.

A lab report containing,

1. The annotated Verilog and C source code for all applications both on the DE1-SoC board and on the PC.
2. Representative screen shots showing the results of testing the various counter designs.
3. Representative screen shots showing the logic analyzer outputs for the various counter designs.
4. Your requirements document and design specification for your net income program.
5. Representative screen shots showing the results of testing your net income program.
6. Answers to any questions above.
7. Other things that you deem to be important.
8. Anything that we haven't thought of.