Verilog & FPGA Fundamentals

ENEE 245: Digital Circuits and Systems Laboratory
Lab 5

Objectives

The objectives of this laboratory are the following:

- To learn how to represent your schematics in Verilog
- To learn how to program an FGPA
- To start to understand the benefits of designing hardware via software

Verilog is a powerful language, and writing code that produces hardware makes design and debugging of that hardware far simpler than using a breadboard. As a result, you will find that it is easy to design circuits of incredible complexity. Software design of hardware (i.e., CAD, or computer-aided design) is the only way that modern computer chips could be built. To try to design and test them by hand would be nothing short of impossible.

In this lab you will design a simple 3-bit ALU in Verilog, for the Xilinx FPGA board, and interface it with the manual switches and LED lights as output. You will also design a simple 1-bit full adder so that you can do an in-depth analysis of the efficiency of synthesized circuits.

Verilog & FPGA Overview

A hardware description language (HDL) is a method to describe hardware using software. An HDL representation of any hardware block is a software file, which adheres to a specific syntactical format. We will use a tool called Xilinx Integrated Software Environment (Xilinx ISE) which will help us to convert Verilog codes to fully functional designs on the Xilinx series of Field Programmable Gate Arrays (FPGAs). In this lab, you will design a full adder and ALU using Verilog, on a Nexys2 board (from Digilent), which contains a Spartan 3E FPGA (from Xilinx). You will also use the I/Os on the Nexys2 board to read in input bits and display the output.

Verilog

Throughout the semester, you will build increasingly complex designs using Verilog, a hardware description language (HDL) widely used to model digital systems. The language supports the design, verification, and implementation of digital circuits at various levels of abstraction. The language differs from a conventional programming language such as C in that the execution of *statements* is not strictly sequential.

A Verilog design can consist of a hierarchy of *modules*. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. *Concurrent* statements define the behavior of the module by defining the relationships between the ports, wires, and registers. *Register Assignment* statements are placed inside a begin/end block and come in two flavors: *Blocking* and *Non-Blocking:*

```
regA = output;          // blocking assignment
regA <= output;         // non-blocking assignment
```

Blocking assignments are executed in series within the block, much like a C-language program. This is not how real hardware works, so these statements are synthesized by the design tools into small sequential state machines that eat up power, consume physical resources, reduce the overall performance, and in general are a bad idea. On the other hand, *Non-Blocking* assignments are executed in parallel within the block and represent how actual hardware works. All *Concurrent* statements and all *begin/end* blocks in the design are executed in parallel. This parallel activity (on a chip-wide scale) is the key difference between Verilog and standard programming languages. A module can also contain one or more instances of another module to define hierarchy.

Only a subset of statements in the language is *synthesizable*. If the modules in a design contain only *synthesizable* statements, software tools like Xilinx ISE can be used to *synthesize* the design into a gate-level *netlist* that describes the basic components and connections to be implemented in hardware. The *synthesized* netlist may then be transformed into a *bit-stream* for any programmable logic devices like FPGAs. Note that this enables a significant improvement in designer productivity: a designer writes hardware behavior in synthesizable Verilog and the ISE (or similar) tool realizes this circuit on a hardware platform such as an FPGA. Verilog designs can be written in two forms:

1) **Structural Verilog**: This is a Verilog coding style in which an exact gate-level netlist is used to describe explicit connections between various components, which are explicitly declared (instantiated) in the Verilog code. Structural Verilog is described below, as this lab uses structural Verilog.
2) **Behavioral Verilog**: In this format, Verilog code is written to describe the function of the hardware, without making explicit references to connections and components. A logic synthesis tool is required in this case to convert this Verilog code into gate-level netlists. Usually, a combined coding style is used where part of the hardware is described in structural format and part of the hardware is described in behavioral format according to convenience.

Wires

Wires in structural Verilog are analogous to wires in a circuit you build by hand: they are used to transmit values between inputs and outputs. Wires should be declared before they are used:

```
wire a;
wire b, c;  // declare multiple wires using commas
```

The wires above are scalar (i.e. represent 1 bit). They can also be vectors (i.e., busses):

```
wire [7:0]   d;  // 8-bit wire declaration
wire [31:0]  e;  // 32-bit wire declaration
```

Wires can be assigned to other wires, concatenated, and indexed:

```
wire [31:0] f;
assign f = {d,e[23:0]}; // concatenate d with lower 24 bits of e
```

In the line above, the brackets [] are used to index a 24-bit range of e and the braces {} concatenate comma-separated wires.

Gates (Structural Primitives)

In this lab, you may use the following primitives: and, or, xor, not, nand, nor, xnor. In general, the syntax is:

```
operator (output, input1, input2);
```

For example, the following Verilog statement implements the Boolean equation `F = a OR b`:

```
wire a, b, F;
/* … some code that assigns values to a and b */
```

```
      or (F, a, b);
```

Complex logic functions can be implemented using intermediate wires between these primitive gates.

Modules

Modules provide a means of abstraction and encapsulation for your design. They consist of a port declaration and Verilog code to implement the desired functionality. For example, consider a module that computes `y = (a + b)(c + d)`:

```
      module example_module(a, b, c, d, y);

          // Port and wire declarations:
          input wire a, b, c, d;
          output wire y;
          wire a_or_b, c_or_d;

          // Logic:
          or   (a_or_b, a, b);
          or   (c_or_d, c, d);
          and  (y, a_or_b, c_or_d);

      endmodule
```

There are a few things to note from this example:

1. The ports must be declared as input or output wires, but can be thought of as wires within the module.
2. Wires declared within a module (such as a_or_b) are limited in scope to that module.
3. Modules should be created in a Verilog file (.v) where the filename matches the module name (so the above example should be located in example_module.v).

Then, after creating a module, you can instantiate it in other modules:

```
      example_module unique_name(

          .a(a), .b(b), .c(c), .d(d), .y(result));
```

(Assuming a, b, c, d, and result are valid wires in the module that this instantiation occurs in, and unique_name is globally unique.)

The syntax .<input/output>(<wire>) is used to explicitly hook up wires to the correct input/outputs of a module. You can also write

```
      example_module unique_name(a, b, c, d, result); // correct order
```

which, while perfectly valid, is not recommended since it is possible to mix up the order of the wires. The first form is also easier to read.

```
      example_module unique_name(result, a, b, c, d); // wrong order!
```

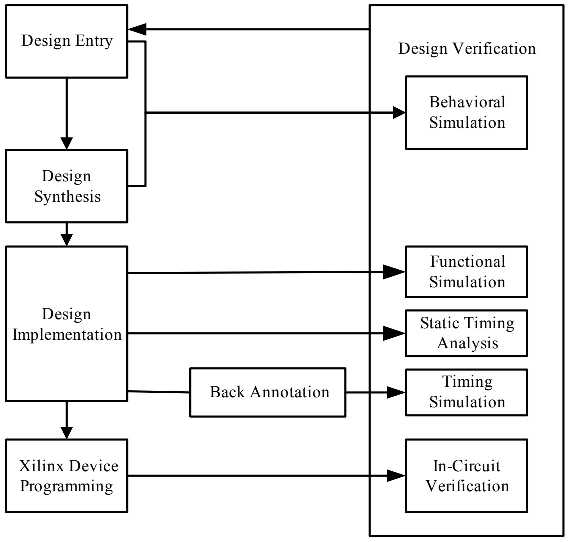
Field Programmable Gate Arrays (FPGA)

A field-programmable gate array is a semiconductor device containing programmable logic components called "logic blocks," and programmable interconnects. Logic blocks can be programmed to perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions such as decoders or mathematical functions. In most FPGAs, the logic blocks also include memory elements like flip-flops. A hierarchy of programmable interconnects allows logic blocks to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. Logic blocks and interconnects can be programmed in the field by the customer or designer (after the FPGA is manufactured) to implement any logical function as and when required hence the name *field programmable logic arrays*.

Realizing a circuit design on an FPGA board consists of three steps, which are performed using a software tool like Xilinx ISE, a tool from Xilinx which integrates various stages of the FPGA design cycle into one software tool:

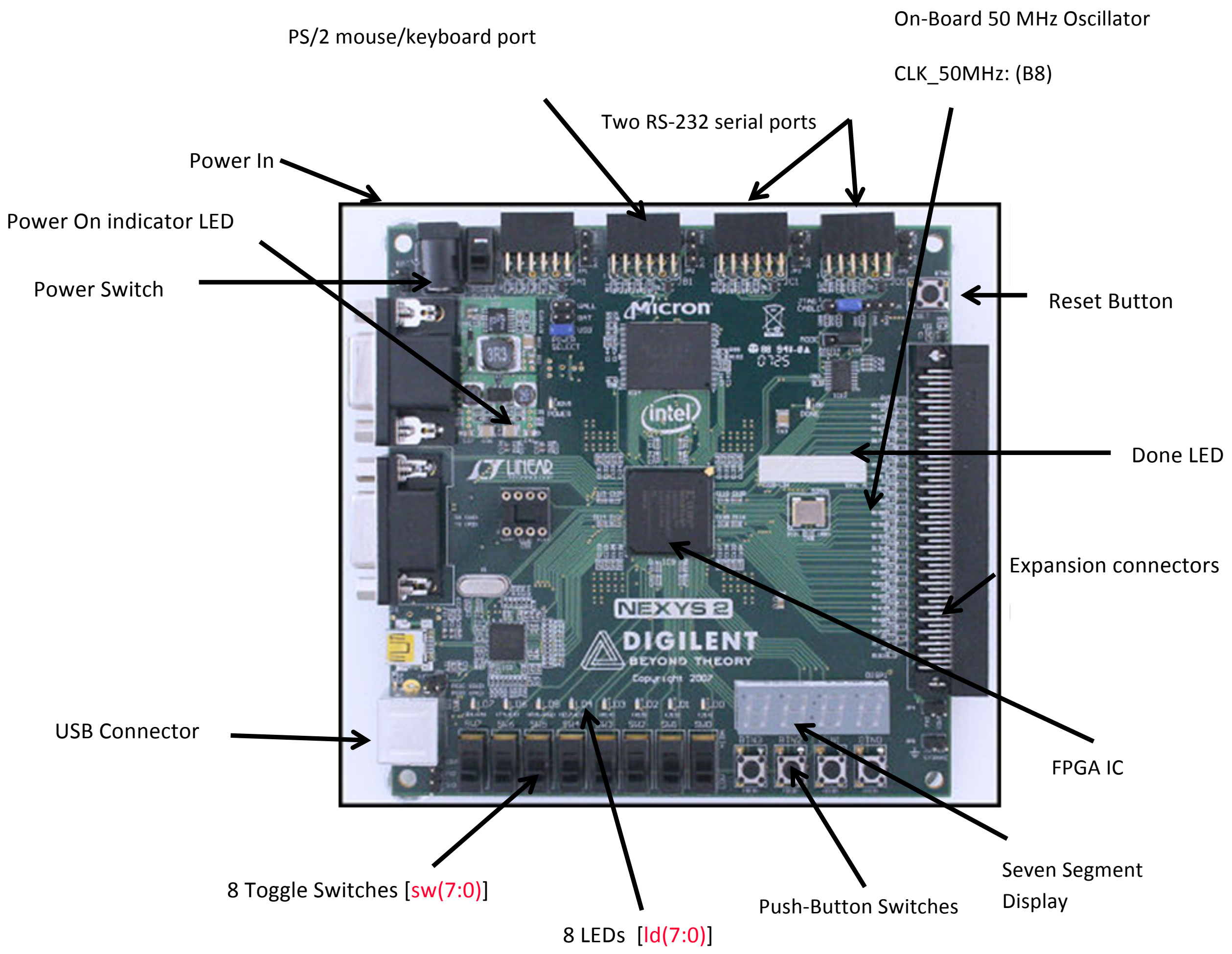
1) **Synthesis**: This is the process of converting a Verilog description into a primitive gate-level netlist. The final product of the design partitioning phase is a netlist file, a text file that contains a list of all the instances of primitive components in the translated circuit and a description of how they are connected.
2) **Implementation**:
    a. **Translation**: The translate step takes all of the netlists and design constraints information and outputs a Xilinx NGD (native generic database) file.
    b. **Mapping**: The mapping step maps the above NGD file to the technology-specific components on the FPGA and generates an NCD (native circuit description) file. This is necessary because different FPGAs have different architectures, resources, and components. Among other tasks, it is responsible for the process of transforming the primitive gates and flip-flops in the netlist into LUTs (lookup tables) and other primitive FPGA elements. For example, if you described a circuit composed of many gates, but ultimately of 6 inputs and 1 output, the circuit will be mapped down to a single 6-LUT. Likewise, if you described a flip-flop it will be mapped to a specific type of flip-flop that actually exists on the FPGA.
    c. **Placement**: This step places the mapped components in a manner that minimizes wiring, delay etc. Placement takes a mapped design and determines the specific location of each component in the design on the FPGA.
    d. **Routing**: This step configures the programmable interconnects (wires) so as to wire the components in the design. Because the number of possible paths for a given signal is very large, and there are many signals, this is typically the most time-consuming part.
3) **Programming the FPGA Device**: In this step, the placed and routed design is converted to a bit-stream using the Xilinx ISE tool. The bit-stream generated by the tool (as a .bit file) is loaded on to the FPGA. This bit-stream file programs the logic and interconnects of the FPGA in such a way that the design gets implemented.

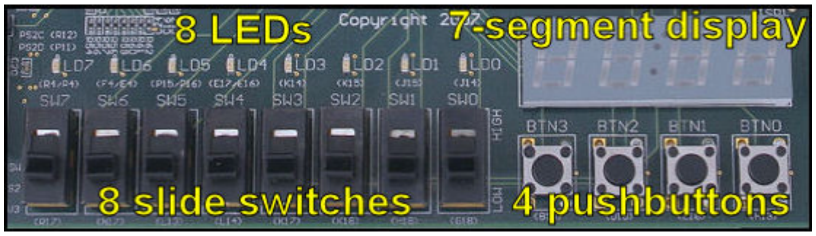
The figure below illustrates the design flow described above.



Digilent Nexys2 Development Board

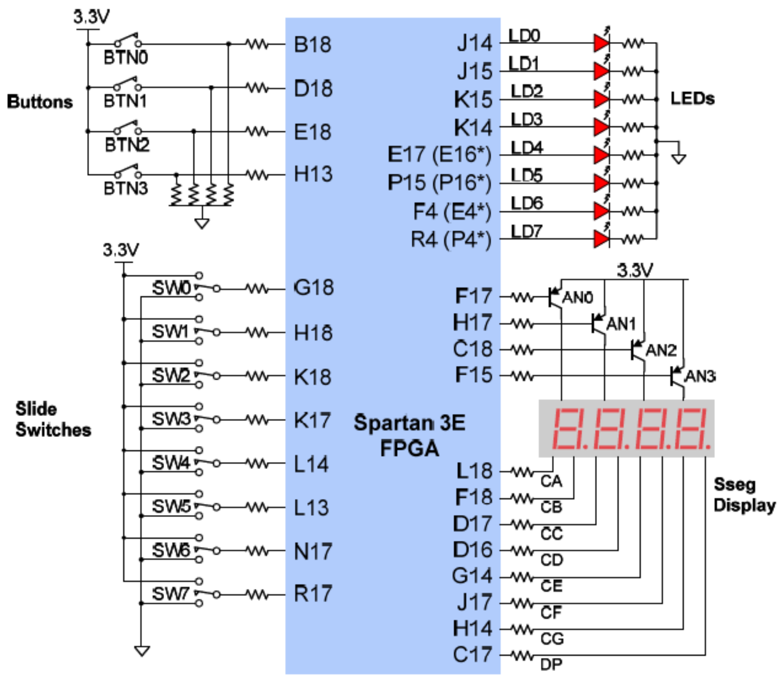
You will build several digital circuits this semester, ranging in complexity from a simple half adder to a simple digital calculator. Digilent's Nexys2 board is the vehicle you will use to implement these circuits. The Nexys2 board is a powerful digital system design platform built around the Xilinx Spartan 3E series of FPGAs. The board has the facility to program the FPGA using a USB connection to your PC. The board provides programmable interfaces to a global reset, four push buttons, a rotational knob, four on/off switches, eight LEDs, clock, memories and the 7-segment displays, as shown in the figure below. The key features and their location on the board is listed below, with a few of the I/O devices (LEDs, 7-segment display, switches and buttons) highlighted:





Inputs: Slide Switches and Pushbuttons

The Nexys2 board includes several input devices, output devices, and data ports, allowing many designs to be implemented without the need for any other components. Four pushbuttons and eight slide switches are provided for circuit inputs. Pushbutton inputs are normally low, and they are driven high only when the pushbutton is pressed. Slide switches generate constant high or low inputs depending on their position. Pushbutton and slide switch inputs use a series resistor for protection against short circuits (a short circuit would occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output).



Nexys2 I/O devices and circuits

Please refer the reference manual for any additional information:

Digilent Nexys2 Board Reference Manual http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf

Pre-Lab Preparation

Full Adder

Design a 1-bit full adder, using either structural Verilog or behavioral Verilog or any combination of the two. As you should remember, the 1-bit full adder has the following inputs and outputs:

- **A** (a 1-bit input operand)
- **B** (a 1-bit input operand)
- **C_in** (a 1-bit input operand)
- **Sum** (a 1-bit output)

- **C_out** (a 1-bit output)

Sum is the 1-bit sum of the three input operands, and C_out (carry-out) indicates the overflow case in which more than one of the input operands is a 1.

The inputs should be tied to the switch inputs of the FPGA (these are called "sw<i>" in the user constraint file). The outputs should be tied to the LED outputs of the FPGA (these are called "Led<i>" in the user constraint file).

Design a test harness to test every possible input to your adder and verify that, for each, it produces correct output.

3-bit ALU

Design the following Verilog modules, using behavioral code:

- a 3-bit adder (has a 3-bit output and 1-bit overflow indicator)
- a 3-bit subtractor (has a 3-bit output and 1-bit overflow indicator)
- a 3-bit bitwise ANDer (has a 3-bit output)
- a 3-bit bitwise ORer (has a 3-bit output)
- an ALU that instantiates the preceding four modules and selects one for output based upon the input function code

Your ALU should have the following inputs:

- a 2-bit function code: 00 = ADD; 01 = SUB; 10 = AND; 11 = OR
- two 3-bit inputs A and B

These inputs should be tied to the switch inputs of the FPGA (these are called "sw<i>" in the user constraint file).

Your ALU should have the following output:

- a 3-bit result
- a 1-bit overflow bit that only lights up if the adder or subtractor overflows

These outputs should be tied to the LED outputs of the FPGA (these are called "Led<i>" in the user constraint file).

The ALU should use the 2-bit function input to choose which module's output gets connected to the ALU's output.

Note that the 1-bit overflow can simply be the topmost bit of a 4-bit sum/difference value, but it should not stay lit when switching from an add/subtract to an AND/OR function.

Write a test harness that checks every possible input and output combination and verifies the design's correctness. Simulate your code and bring a pre-lab write-up showing the code and simulation results.

In-Lab Procedure

Bring flash drives to store your data.

Ask the TA questions regarding any procedures about which you are uncertain.

Complete the following tasks:

- **Program your full adder onto the FPGA board.**
- Test every possible input and verify correctness. Demonstrate your working adder to the TA.
- Look at the detailed RTL schematics produced by the software; save it for your post-lab report.
- Look at the timing report that gives the pin-to-pin delays for input/output combinations of every pin. Save these reports and tables for your post-lab report.
- **Program your ALU onto the FPGA board.**
- Test every possible input and verify correctness. Demonstrate your working ALU to the TA.
- Look at the detailed RTL schematics produced by the software; save it for your post-lab report.
- Look at the timing report that gives the pin-to-pin delays for input/output combinations of every pin in every module. Save these reports and tables for your post-lab report.

Post-Lab Report

Write up your code, schematics, and lab procedures. Demonstrate the correctness of your designs through your pre-lab simulations and note any differences between what you simulated and how the circuits behaved in the lab.

Full Adder

Regarding the RTL schematic produced by the software—how did the design software synthesis your code? Where did it choose poorly, and how could it have done better? Could you have better specified your design to get more efficient results? Given the simplicity of the design, and the many possible ways of generating the circuit, there is likely to be much you can say about this aspect. Moreover, because it is a small design, you should be able to do an extremely thorough analysis. In general, what can you say about synthesis of high-level code?

Regarding the timing report that gives the pin-to-pin delays for input/output combinations of every pin—what does the design software say for the timing? How fast is it?

3-bit ALU

Regarding the RTL schematic produced by the software—how did the design software synthesis your code? Where did it choose poorly, and how could it have done better? Could you have better specified your design to get more efficient results?

Regarding the timing report that gives the pin-to-pin delays for input/output combination of every module—what does the design software say for the timing of each component? How fast is your design? How fast is each component, and how fast is the overall ALU? Calculate the delay through the MUX at the end of the ALU.