Designing with VHDL

Introduction

When a new digital system is designed, the first step is *capturing* that design, whether it be on paper or in a computer. Schematic capture, or block-diagram editing, works well for both paper-based design and CAD (computer-aided design), and it is often very intuitive. Like the real physical implementation, it clearly shows individual components, connected by lines that represent wires and carry voltages which are equivalent to logic levels. One example of this is the logic system that models the way two light switches at opposite ends of a hall can control the same light, shown in Figure 1. Here, the two switches sw4 and sw5 both control the light d3. The other two outputs simply represent indicators of the current position of each switch.

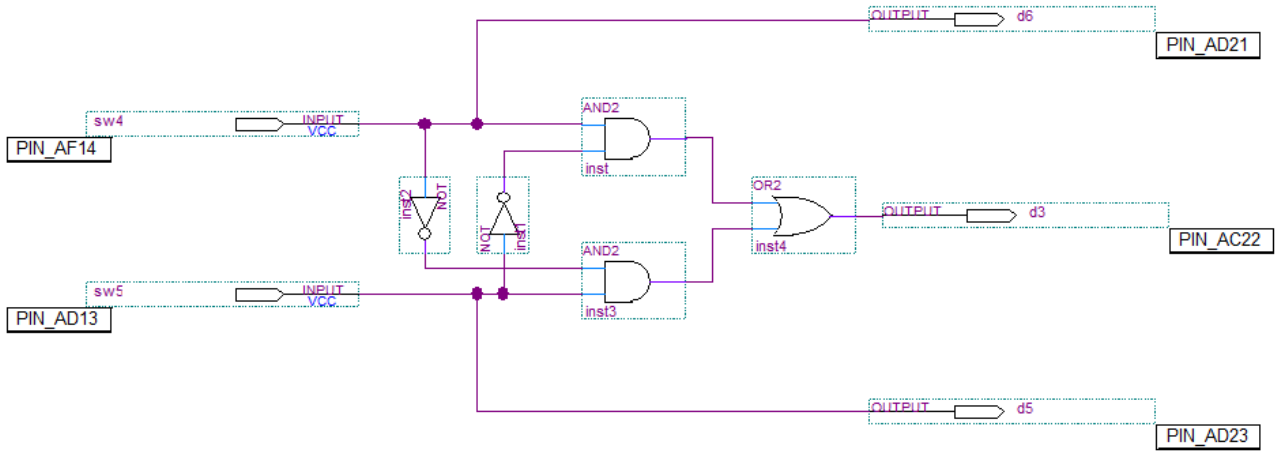

Figure 1: The Quartus schematic capture of a light switch controller example.

No matter how complicated circuits become, it is critical to NEVER lose sight of the fact that they are always composed of components connected by wires or other electrical conductors. The topic of this document is not how to build these circuits, but rather how to describe them – how to capture them. Still, before embarking on another way to describe them, it is important to ground ourselves in the reality of how they are built. The circuit in Figure 1 could have been built with as few as two 14-pin integrated circuits (like that shown in Figure 2), two switches, and one suitable light indicator (assuming that d5 and d6 are not implemented). These could have been soldered to a circuit board or plugged into a prototyping board.

However, when implemented on the DE2 boards used in the laboratory, all of the logic resides in the Cyclone II FPGA highlighted with a yellow box in Figure 3. To be more specific, it actually requires less than 0.01% of the gates and other devices that are inside that FPGA.

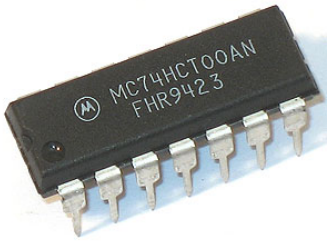

Figure 2. A typical digital logic integrated circuit, specifically one that could be used to implement most of the logic in the previous figure.

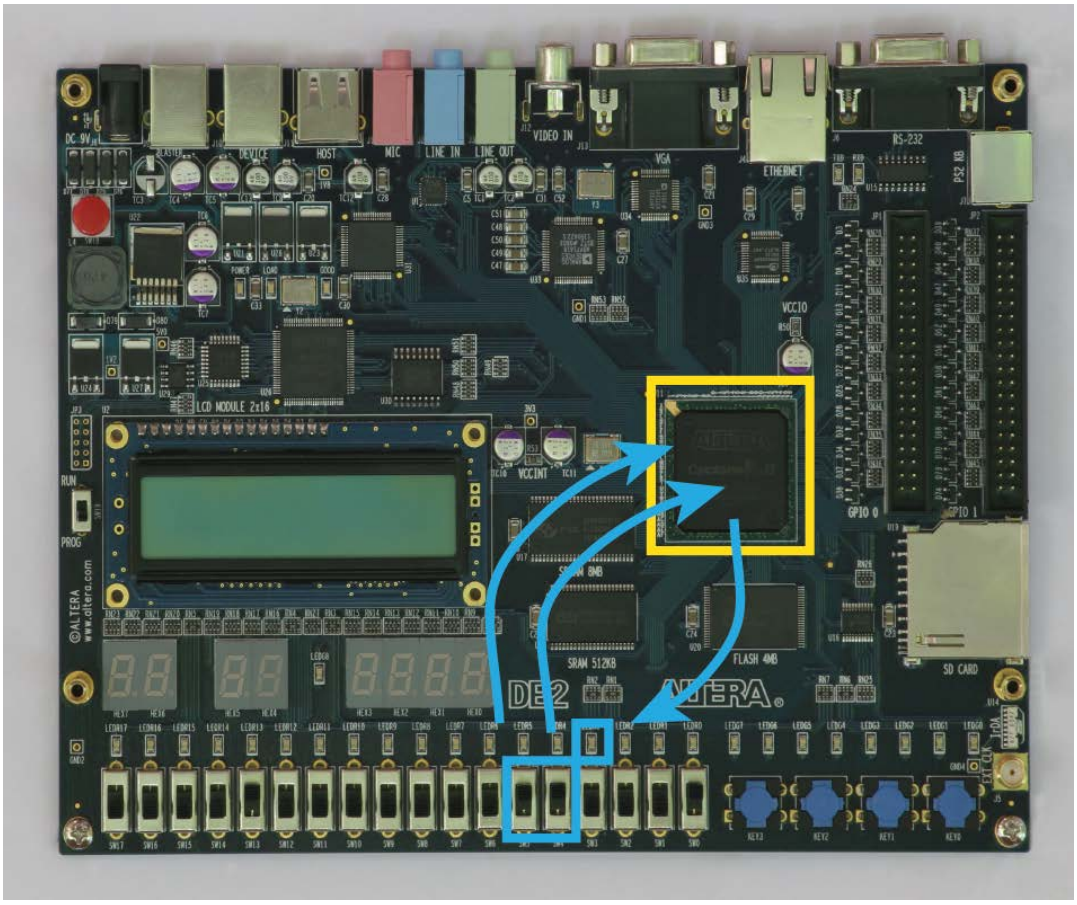

Figure 3. The implementation of the circuit on the DE2, once "programmed," utilizes only a very small part of the board.

The DE2 implementation of that circuit also requires two switches and one LED, such as the ones outlined in blue in Figure 3. Part of the convenience of the DE2 is that other switches and LEDs could easily have been used instead. There are a great many other devices on the DE2 that are simply not needed. Students often think that their computer is needed to operate these circuits or that other devices on the DE2 are needed. This is simply not true. Once the FPGA is "programmed" (a confusing term), the DE2 can be disconnected from the computer. As long as it has power available, and for this particular circuit of Figure 1, the two switches will provide logic levels (voltages "near 0V" or "near the positive supply voltage") to the FPGA, and the FPGA will provide a voltage back to the LED.

And finally, to belabor what may be obvious to some readers, the gates and inverters in this sample circuit RUN ALL OF THE TIME. They do not take turns, they do not function when a computer tells them to compute their output, and they certainly are not "running a program." They are not that smart. They simply have one or more logic levels coming into them, and they produce logic levels according to their function (NOT, AND, or OR in the specific case of Figure 1).

Motivation

Given that schematics are intuitive, a reasonable question might be why they are not sufficient for all digital design needs. As circuits get more complicated, drawing them inevitably becomes more difficult. Lines cross over lines, and it often becomes necessary to keep shifting devices around to make room in some part of the schematic for other devices. There are several techniques for alleviating these problems:

- Give signals unique names where they first appear as outputs from a device, and use those names elsewhere in the schematic. This eliminates some wires (lines), as shown in Figure 4, where RESETN is created as an output of the altpll0 device, and used elsewhere as the input to the SCOMP device.
- Split designs across multiple pages, taking advantage of the named-signal technique above to "run wires between pages."
- When signals have associated meaning, such as a group of eight logic levels that represent an 8-bit number, draw them as a thick line, with clear notation of the underlying meaning as a vector, usually called a "bus" in digital electronics. Figure 4 shows examples including the 8-bit IO_ADDR bus and the 16-bit IO_DATA bus.
- When a useful device is created, give it its own unique symbol, and use the symbol (with none of the internal details) wherever the device is needed. This is analogous to modular programming techniques in software, and it is often referred to as hierarchical design, since a device can be composed of several levels of increasing detail – a hierarchy. Figure 5 shows an example of this, which is exactly what would be produced if the schematic of Figure 1 were embedded in a single block for use in other designs.

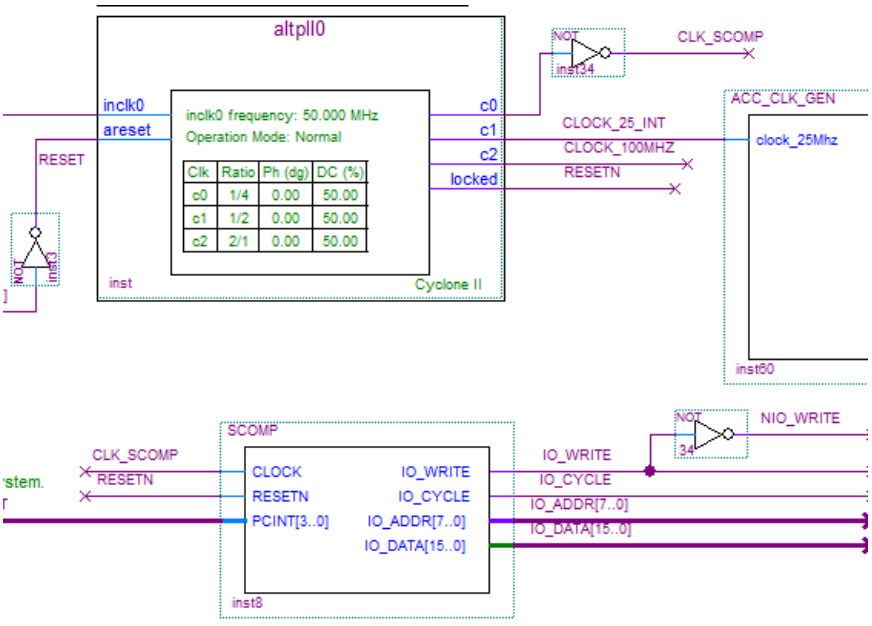

Figure 4. Excerpt from a complex block diagram that illustrates some techniques for extending the utility of schematic capture.

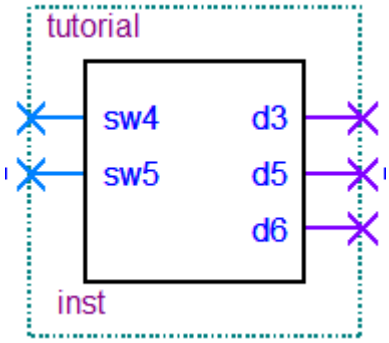

Figure 5. A concise symbol representation of Figure 1, suitable for inclusion in a higher-level schematic.

Still, those techniques are not enough for complex designs. They definitely extend the usefulness of schematics, often as the best way to convey the highest levels of the design hierarchy. But as the only means of describing circuits, schematics can be difficult. For this reason, as digital circuit designs routinely began exceeding tens of thousands of gates in the 1980's, the United States Department of Defense sponsored efforts to establish an open text-based language to describe circuits. The largest international organization of electrical and electronic engineers, IEEE, joined the effort to create a standard, and VHDL, the VHSIC Hardware Description Language, was born. (VHSIC is another acronym – Very High Speed Integrated Circuit, a related U.S. government program.)

VHDL structure

One of the techniques of the previous section, hierarchical design, applies just as well to text-based hardware description languages. For our purposes, a VHDL text file is equivalent to a device. It can be arbitrarily complex, and it can refer to other devices (because of the hierarchical nature), but there is a one-to-one relationship between a VHDL file and a corresponding device that has specific inputs and outputs. Often, instead of the word "device," VHDL textbooks will use the words "entity" or "component." But since these two words are VHDL keywords with very specific meanings, the word "device" will be favored here.

Furthermore, each of our VHDL design files will be divided into two major sections, the *entity* declaration and *architecture* body. Strictly speaking, each of these is a single VHDL statement, and all VHDL statements end with a semicolon, so these two statements, which make up every VHDL file, are generically of this form:

```
entity  device_name is … end  device_name;
architecture arch_name of device_name is begin … end arch_name;
```

The italicized words are names that the designer makes up, so technically all that has to be done to the above two lines to make it a valid VHDL file is 1) make up two names, and 2) fill in the parts denoted by ellipses (…). (Aside: If you are wondering why the second statement has a **begin** and an **end**, while the first only has an **end**, it is simply the way that the syntax was defined.)

It turns out that, as in other languages, there are additional options for including or cross-referencing other files, so the two statements above are usually preceded with clauses that accomplish this. For our purposes, these will usually include the following two clauses, which will be explained later:

```
library  ieee;
use      ieee.std_logic_1164.all;
```

Combining the four statements above into a single group, and making use of the fact that VHDL allows (and encourages) use of separate lines for long statements, our generic VHDL file looks like this:

```
library  ieee;
use      ieee.std_logic_1164.all;

entity  dev_name is
. . .
end  dev_name;

architecture arch_name of dev_name is
begin
. . .
end arch_name;
```

This file is composed of four statements: **library**, **use**, **entity**, and **architecture**. Each one ends with a semicolon. Some of them, when completed to create an actual device description, will have additional statements before their **end** keyword, because it is allowed to have statements within statements. Remember this basic structure. From a design standpoint, it is equivalent to the illustration of Figure 6, where two external input signals have been added (X and Y), along with one output signal (Z), and one internal signal (W).

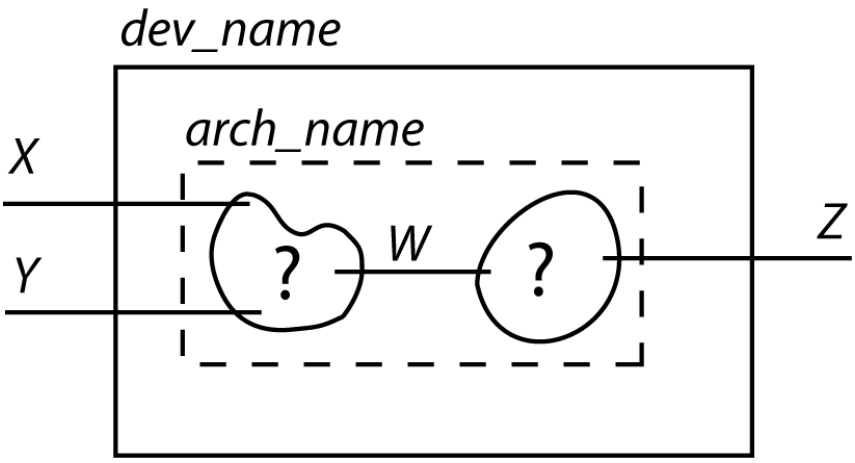

Figure 6. The structure of a VHDL file corresponds to a boundary with inputs and outputs, described as an "entity" with a name, such as *dev_name*, and an internal "architecture," also with a name (*arch_name* here).

The declaration of inputs, outputs, and internal signals like those of Figure 6 is accomplished with the **port** statement (which always must be within an **entity** statement, because it defines externally visible signals), and a **signal** statement (which must always be within an **architecture** statement, because it defines internal signals). Those can be added to the earlier VHDL code, producing an almost complete file that is specific to the example of Figure 5:

```
library  ieee;
use      ieee.std_logic_1164.all;

entity  dev_name is
      port
      (
            X, Y    : in  type;
            Z       : out type
      );
end  dev_name;

architecture arch_name of dev_name is
      signal       W      : type;
begin
      . . .
end arch_name;
```

The designer still has to give the type of each signal X, Y, Z, and W, because VHDL is a strongly-typed language that does not automatically assume an intended type. This will be discussed soon.

Another feature of note is that VHDL is NOT case-sensitive. The examples here will generally use lowercase for VHDL keywords, *italics* for items to be filled in by the designer, and UPPERCASE or MixedCase for signal names and other names created by the designer. But there may be exceptions, and VHDL does not "care" or make any distinction between names like d3 and D3 or PORT and port.

The only other thing missing in the code above are the statements still marked with ellipses (…) between **begin** and **end** in the **architecture** body. These correspond to the parts of Figure 5 where there are question marks. This can be any logical relationship, for example.

With only minor additions and some changes to the input and output names, the generic structure above can become a valid VHDL description of the circuit in Figure 1.

```
-- VHDL equivalent of the tutorial
library ieee;
use ieee.std_logic_1164.all;

entity Tutorial is
	port
	(
		Sw4	: in std_logic;
		Sw5	: in std_logic;
		D3	: out std_logic;
		D5	: out std_logic;
		D6	: out std_logic
	);
end entity;

architecture A of Tutorial is
begin
	D3 <= (Sw4 and (not Sw5)) or (Sw5 and (not Sw4));
	D5 <= Sw5;
	D6 <= Sw4;
end A;
```

Here, no internal signals were required, so there are no **signal** declaration statements. All inputs and outputs are of type **std_logic** (standard logic), which for our current purposes describes signals that can take on the values of high (logic 1) or low (logic 0), along with some other possibilities to be considered later.

Of most significance are the three assignment statements added to the **architecture** body. The pair of symbols "<=" is intended to resemble a left-pointing arrow. It indicates that the expression on the right-hand side produces a value that drives, or "is assigned to" the signal on the left-hand side. In the example above Sw5 drives D5, so they are essentially the same signal — two points along the same wire.

The expression driving D3 introduces some standard VHDL operators, which are self-explanatory. It is possible to use **and**, **or**, and **not** with their normal meanings. There is an established order of precedence among these operators and others, but it is good practice to group terms with parentheses to emphasize the intended order of evaluation.

Another addition to the code above is the use of one comment. Comments can appear anywhere, always following a pair of hyphens, and extending to the end of the current line.

Concurrency in VHDL

Consider an alternate way to implement the example, where two signal names (P and Q) are created, one for each of the two outputs of the AND gates in Figure 1, producing something like the schematic in Figure 7.

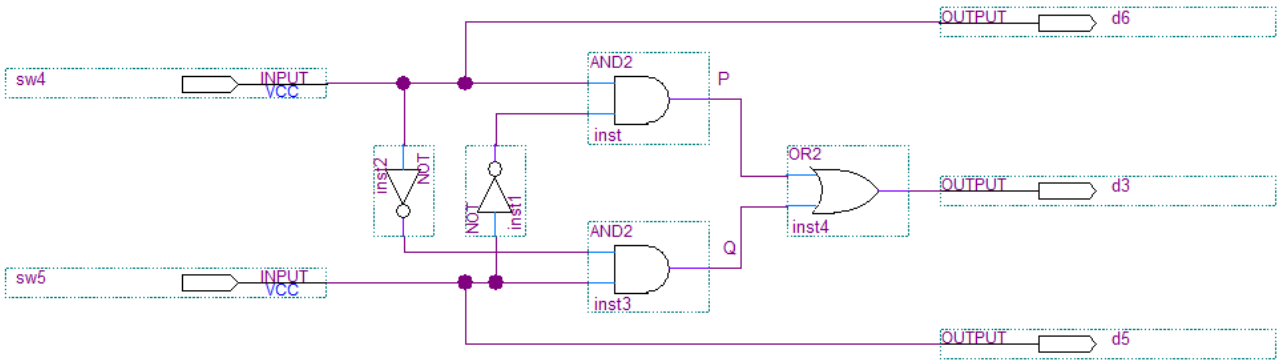

Figure 7. Signal names P and Q have been added to Figure 1, changing nothing in the actual outputs.

P is the output of one gate, and Q is the output of the other. Then, D3 is obviously P OR'ed with Q. This is easily described in VHDL, and the result is equivalent to the earlier example, but with two signal declarations and different assignment statements:

```
architecture A of Tutorial is
      signal P : std_logic;
      signal Q : std_logic;
begin
      P <= (Sw4 and (not Sw5));
      Q <= (Sw5 and (not Sw4));
      D3 <= P or Q;
      D5 <= Sw5;
      D6 <= Sw4;
end A;
```

The one thing that is different, and it may be a little bothersome, is that some of the signal names (the new ones, P and Q) appear on both sides of different assignment statements. They are on the left-hand side of the first two assignment statements, but they both appear on the right-hand side of the third assignment. This is an opportunity to introduce a key feature of VHDL: just like a schematic, all of the hardware described in the VHDL device operates all of the time, concurrently. Recall from the Introduction what was said about logic gates in Figure 1:

> *They do not take turns, they do not function when a computer tells them to compute their output, and they certainly are not "running a program." They are not that smart. They simply have one or more logic levels coming into them, and they produce logic levels according to their function (NOT, AND, or OR in the specific case of Figure 1).*

Nothing has changed in the VHDL implementation versus the schematic. There is no implied order in which the gates (here, the different assignment statements) determine their outputs. They ALWAYS determine their outputs based on their inputs. When one of their inputs changes, their output may change, and that may

propagate to another assignment statement – possibly one that was above it or below it in the **architecture** body.

In the particular example above, there are at least five things happening in parallel (concurrently), represented by the five assignment statements. There are more things happening in parallel when you consider that the **not** operations are also concurrent, even though they are buried within their respective lines of VHDL code. There would be no effect to the circuit if the five lines were shuffled, or if the P assignment were changed to swap the order of the two terms that are and'ed. It would mean nothing more than if we had gone back to the schematic and dragged a different gate toward the upper left-hand corner of the page, as long as we didn't disconnect or reconnect any wires. Physical location of a concurrent VHDL statement on the page of text does not mean anything more than physical location of a gate on a schematic page. (There are some VHDL statements which are not concurrent, but we have not gotten to them yet.)

Assignments and signals vs. variables

Once the concept of concurrency is fully understood, it will no longer be tempting to think of VHDL **architecture** bodies as being lines of code that a computer executes one after the other. Furthermore, it will not be tempting to write VHDL code like this:

```
architecture A of Tutorial is
      signal P : std_logic;
      signal Q : std_logic;
      signal Temp: std_logic;
begin
      Temp <= not Sw5;        --  This code will NOT compile!
      P <= (Sw4 and Temp);
      Temp <= not Sw4;
      Q <= (Sw5 and Temp);
      D3 <= P or Q;
      D5 <= Sw5;
      D6 <= Sw4;
end A;
```

At a glance, this may seem to produce the same result as the previous version, where we have simply introduced a new signal Temp, and then used it as part of the expression for P and part of the expression for Q. And this might have worked if this WERE a computer program being executed sequentially in some computer language that looks like VHDL. But there is a key difference between a VHDL assignment to a signal and a computer programming language assignment to a variable. Always think of assignment to a signal as being completely equivalent to using a gate or some other device to drive that line with a voltage, and it will be clear that the code above is equivalent to Figure 8. By defining Temp as **not Sw5**, it becomes the output of an inverter (NOT gate). Then, by defining Temp as **not Sw4**, it becomes the output of a different inverter. The compiler will produce an error along the lines of "Temp cannot be driven by more than one output." This makes perfect sense, because it cannot be two things at once. That's what concurrency means – multiple things happening at once, and the seven lines in the architecture body are happening at once.

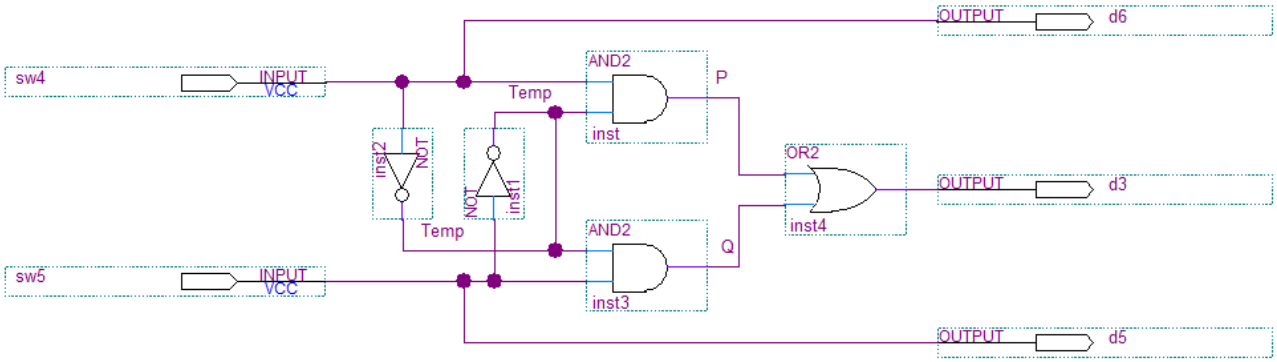

Figure 8. Using the same signal name for two different wires is equivalent to connecting them. Often that is what the designer intends, but two outputs cannot drive the same signal as shown here.

Another way to look at this is that signals are not convenient places to tuck away a value, the way that variables are used as names for memory locations. Signals are not variables in VHDL. (There is such a thing as a variable in VHDL, but we will avoid using variables until we know exactly how they can be used. Signals are all we need for most purposes.) All that being said, there is nothing that prevents us from having a signal, or a group of bussed signals (a vector of signals), being driven by a device that has memory. That is where memory, or state information, comes from in a digital system, and is the next topic.

The D Flip-Flop and process statements

NOTE: Students taking ECE2020 concurrently and freshman ECE students who have not taken ECE2020 yet will not be able to fully appreciate the remainder of this reading until after they have covered flip-flops. There are ways to remedy this: 1) wait a week or two, if necessary (only works for the concurrent ECE2020 students), or 2) read Chapter 3 of Harris & Harris, up to and including section 3.3. Do one or both of these before continuing. Even students who have previously taken a logic course may find this reading in Harris & Harris useful.

As mentioned briefly in Lecture 4, this course will rely on D flip-flops as the memory in devices such as state machines. The D flip-flop (or DFF) presents a puzzle in terms of its implementation in VHDL. If every line in an **architecture** body is running concurrently, and if each of those lines represents combinational logic, then there is no way to represent an event that captures and stores a single bit of data, as a DFF does. This will require a new statement in VHDL, the **process** statement.

In its simplest form, a **process** statement is a sequence of statements that are evaluated sequentially as the result of the change of one or more signals, in order to determine one or more other signals. It is possible to make a **process** that implements a simple AND gate, although it is needlessly complicated. This is shown in the code example below, and the **process** is just an extra set of encompassing statements around the line that actually implements the assignment. The two parameters X and Y in the line **process (X, Y)** are called the *sensitivity list* of the **process**. Like any other statement in the architecture body of a VHDL file, this **process** will constantly produce a value, but that value will change ONLY when one of the signals in the sensitivity list changes.

```
-- AND gate implemented with a process statement
library ieee;
use ieee.std_logic_1164.all;

entity AndGate is
	port
	(
		X	: in std_logic;
		Y	: in std_logic;
		Z	: out std_logic
	);
end entity;

architecture A of AndGate is
begin
	process( X, Y )
	begin
		Z <= X and Y;
	end process;
end A;
```

So, if instead of the above, the implementation of AndGate were as follows, the result would be different.

```
architecture A of AndGate is
	begin
		process( X )
		begin
			Z <= X and Y;
		end process;
	end A;
```

This version above would still compile, and it could be implemented in an FPGA just as well as the previous version of AndGate. The problem is that a NEW value of Z is only produced when X changes. So, if X and Y both became 1 (True), then the **process** will produce the value of 1 for Z, because ”1 and 1” is 1. But, if Y were then to become 0 (False), the **process** would not change its value for Z, even though “1 and 0” is 0. This is because Y is not in the sensitivity list, and it would require a change in X for the value of Z to change.

This does not mean that the **process** is not still running concurrently and producing a value for Z after Y changes. It definitely still continues to produce a value – the old value. The next time X changes, it will cause a new update in the value of Z that takes into account both the changed X value and any change in Y that may have been ignored.

So, aside from producing AND gates that may not behave as they should, what is a **process** good for? Remember what a DFF does. The Q output depends on the D input, but the D input can change between 0 and 1 many times and never change the Q output. The CLOCK input of the DFF has to change in order to capture a new value of D and pass it to the Q output.

The code below is very close to what is needed. It has the correct inputs, D and CLOCK, and the correct output Q, but there is a slight problem.

```
-- A D flip-flop, almost
library ieee;
use ieee.std_logic_1164.all;

entity DFF is
      port
      (
            D        : in std_logic;
            CLOCK    : in std_logic;
            Q        : out std_logic
      );
end entity;

architecture A of DFF is
begin
      process( CLOCK )
      begin
            Q <= D;
      end process;
end A;
```

This example above correctly uses the sensitivity list to make the DFF ignore changes of the D input, in terms of producing a new Q output. The value of Q stays constant until CLOCK changes, and only then does Q change to become the current value of the D input. So, this is correct except for one minor detail. The DFFs that are commonly used are *positive-edge-triggered* DFFs. They only change their value when their CLOCK input changes from low to high. (There are also negative-edge-triggered DFFs, but they will not be of interest.) Fortunately, VHDL includes ways to specify exactly what sort of change will trigger a new output. One of these is the **rising_edge()** function:

```
architecture A of DFF is
begin
      process( CLOCK )
      begin
            if rising_edge(CLOCK) then
                  Q <= D;
            end if;
      end process;
end A;
```

In order to make use of this new function, it is necessary to add a conditional aspect to VHDL, an **if** statement. This implementation of **if… then…end if** is similar to implementations in other languages, and it does allow for an “else” possibility that is not needed here. Since **rising_edge(CLOCK)** is only true when a positive (rising)

edge occurs on CLOCK, then that is the only time that the Q value is updated, and this correctly implements a positive-edge-triggered DFF.

It is worthwhile to demonstrate that a **process** statement defines a device (and thus a concurrent activity) just as surely as the assignment statements earlier were defining concurrent gates. Suppose that instead of using the D5 output to simply be the same as the input Sw5, we wished to make it show an updated value of Sw5 every tenth of a second. Assuming there is a clock running at 10 Hz, called clk_10Hz, this could be done with the schematic shown in Figure 9. (The PRN and CLRN inputs of this DFF are not being used, but if they were, they could force it to a high or low state upon reset, power-up, or some other event.)

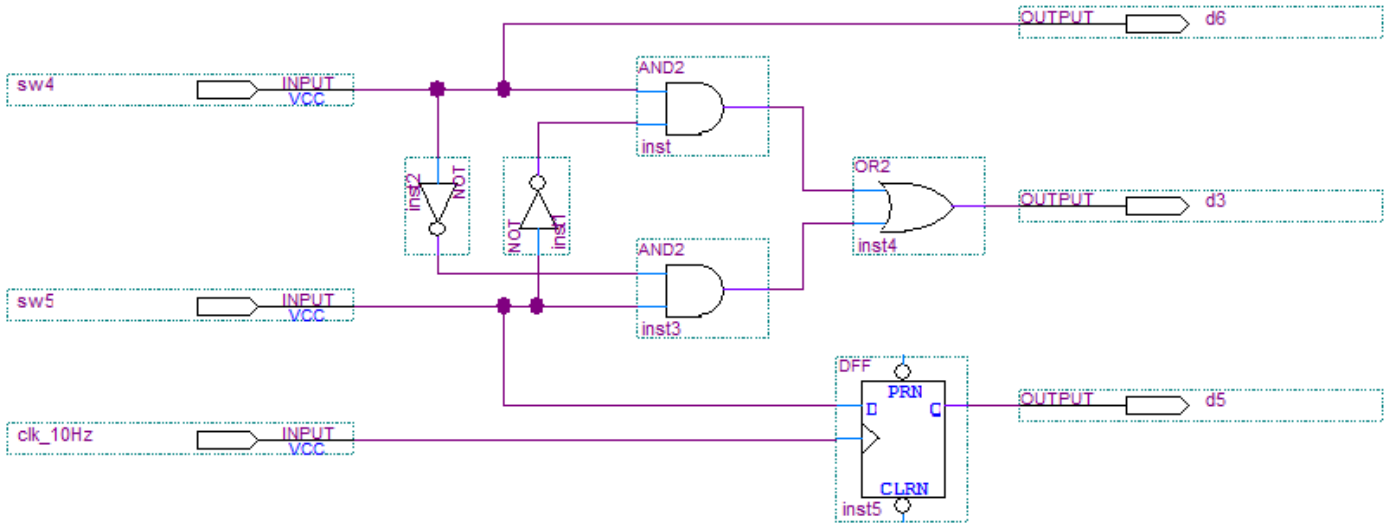

Figure 9. A DFF can be used to sample the value of Sw5 every tenth of a second. Now, D5 is only updated on the rising edge of the 10 Hz clock.

And finally, to bring all of this together in VHDL, including the original lightswitch tutorial combined with the DFF, we have the code below, which is completely equivalent to Figure 8. Do not forget that the **process** statement (all six lines in the example below), is a single concurrent entity, along with the line that defines D3 and the line that defines D6. So, the **process** statement can be moved, in its entirety, up higher in the **architecture** body without changing the operation of the device. It is no different than moving the DFF symbol around in Figure 9.

```
library ieee;
use ieee.std_logic_1164.all;

entity tutorial is
	port
	(
		Sw4	: in std_logic;
		Sw5	: in std_logic;
		Clk_10Hz : in std_logic;
		D3	: out std_logic;
		D5	: out std_logic;
		D6	: out std_logic
	);
end entity;

architecture a of tutorial is
begin
	D3 <= (Sw4 and (not Sw5)) or (Sw5 and (not Sw4));
	D6 <= Sw4;
	process( Clk_10Hz )
	begin
		if rising_edge( Clk_10Hz ) then
			D5 <= Sw5;
		end if;
	end process;
end a;
```

Major concepts

VHDL is an alternate means of capturing a designer's intent. It is arguably "better" as designs get more complicated, but in theory there is nothing that can be done with VHDL that cannot be done with a schematic.

Digital hardware is inherently parallel in nature. Whether creating schematics or writing VHDL, a designer must remain aware of the power and pitfalls of devices running concurrently.

The primary components of a VHDL file defining the operation of a single device (our primary emphasis) is a series of one or more clauses for including external content, followed by a single **entity** statement (with an embedded **port** statement) and a single **architecture** statement. Because the **architecture** statement is usually moderately lengthy, it is usually the dominant portion of the VHDL device file.

Purely combinational logic circuits can be implemented as a series of assignment statements in VHDL, all within the **architecture** body.

Sequential logic circuits (including counters and anything involving D flip-flops) require the use of the **process** statement to surround statements that have a sequential dependency on a triggering event such as a clock edge.

For further reading

If most of the preceding material made sense, there is no need to review other sources at this time. But, if you are confused, continue here. Sometimes, reading about the same material from a different perspective can be helpful.

A thorough primer in VHDL is at http://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html . It is recommended that students go no further than the section called *Concurrency* (i.e., stop at *Structural Description*). Even before this, there is some discussion of behavioral vs. structural VHDL, and that is possibly a confusing topic, since we will rely almost exclusively on behavioral VHDL in the weeks ahead.

The series of slides at http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/599_2008_94_2797.pdf quickly gets to the point of discussing some details and additional features not covered here, but it is generally readable and not too confusing. You may gain insight from slides all the way to the examples at the end.

Be very careful about seeking out other tutorials on VHDL. There are several common pitfalls, and most treatments of the material fall into one or more of these categories:

- Assuming a prior knowledge of electronics and/or advanced digital logic theory
- Covering only a specific vendor's use of VHDL (specific compilers, specific FPGAs)
- Needlessly going into arcane details early in the discussion
- Emphasizing simulation over design
- Trying to find a way to make writing VHDL be more like programming a computer
- Being so structured in the presentation that it reads too much like a reference manual

Exercise 1

This extended exercise has three main learning objectives:

1. Installing Quartus on your own home computer,
2. Repeating the process of creating a project, with minimal specific instructions.
3. Creating a project with a different target FPGA (the one on your project hardware) and a different design capture method (VHDL instead of schematic).

Quartus installation

Instead of installing the version of Quartus recommended for ECE2031 students (Quartus version 9.1), install a newer one that should be more suited to the newer hardware that you use. Go to this location: https://www.altera.com/downloads/download-center.html . If necessary, create a free "myAltera" account, and if asked, note that you are a student.

Find version 13.0, Service Pack 1, as shown in Figure 10. This is NOT the newest version. Do not install any other version. Download and install the free Web Edition, ignoring any later suggestions to install the Subscription Edition. If you run into any problems, post a note on Piazza.

Optionally, after Quartus is installed, you can repeat either of the two schematic entry projects that you completed earlier on Quartus 9.1. You can choose the same target hardware, even though you will not need

to ever program the DE2 with this design file. You should not encounter any major differences, aside from simulation. The steps for simulation are slightly different, and you should just avoid simulation at this time.

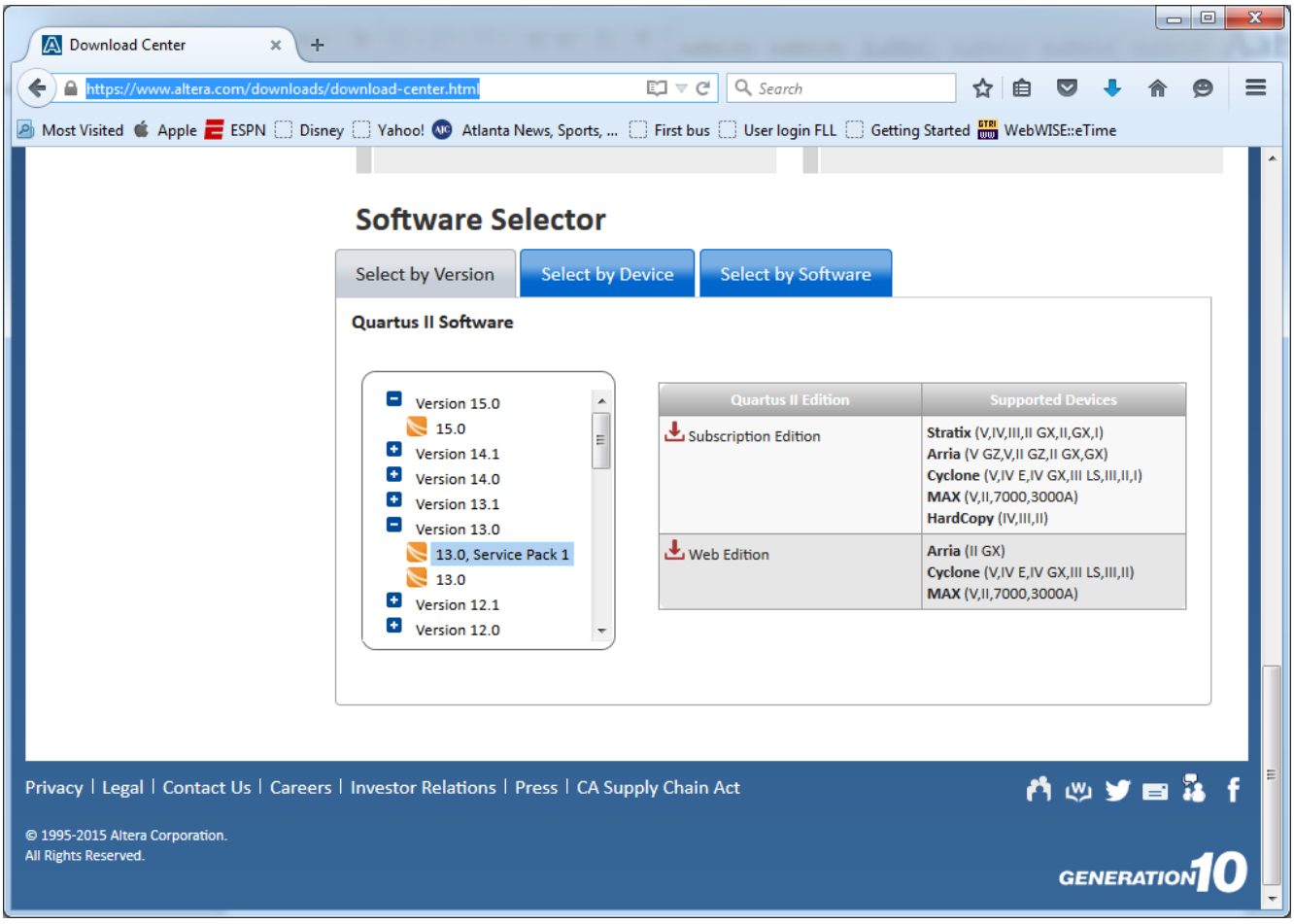

Figure 10. Download screen at Altera web site. Do not install the newest version!

Project creation with a different target

Each ECE2883HP design team will have their own DE2-115 development board (http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=502). This board has a Cyclone IV FPGA, which is more powerful than the Cyclone II on the lab boards.

Using your newly installed Quartus 13.0, create a new project on your own computer, which you can call "Lab4" or anything else that you choose (but you will see "Lab4" used in the remaining instructions here. When you select a target device, choose the EP4CE115F29C8, as shown in Figure 11. Stop before creating any design files, and the steps should be similar to those followed earlier with Quartus 9.1 in the lab.

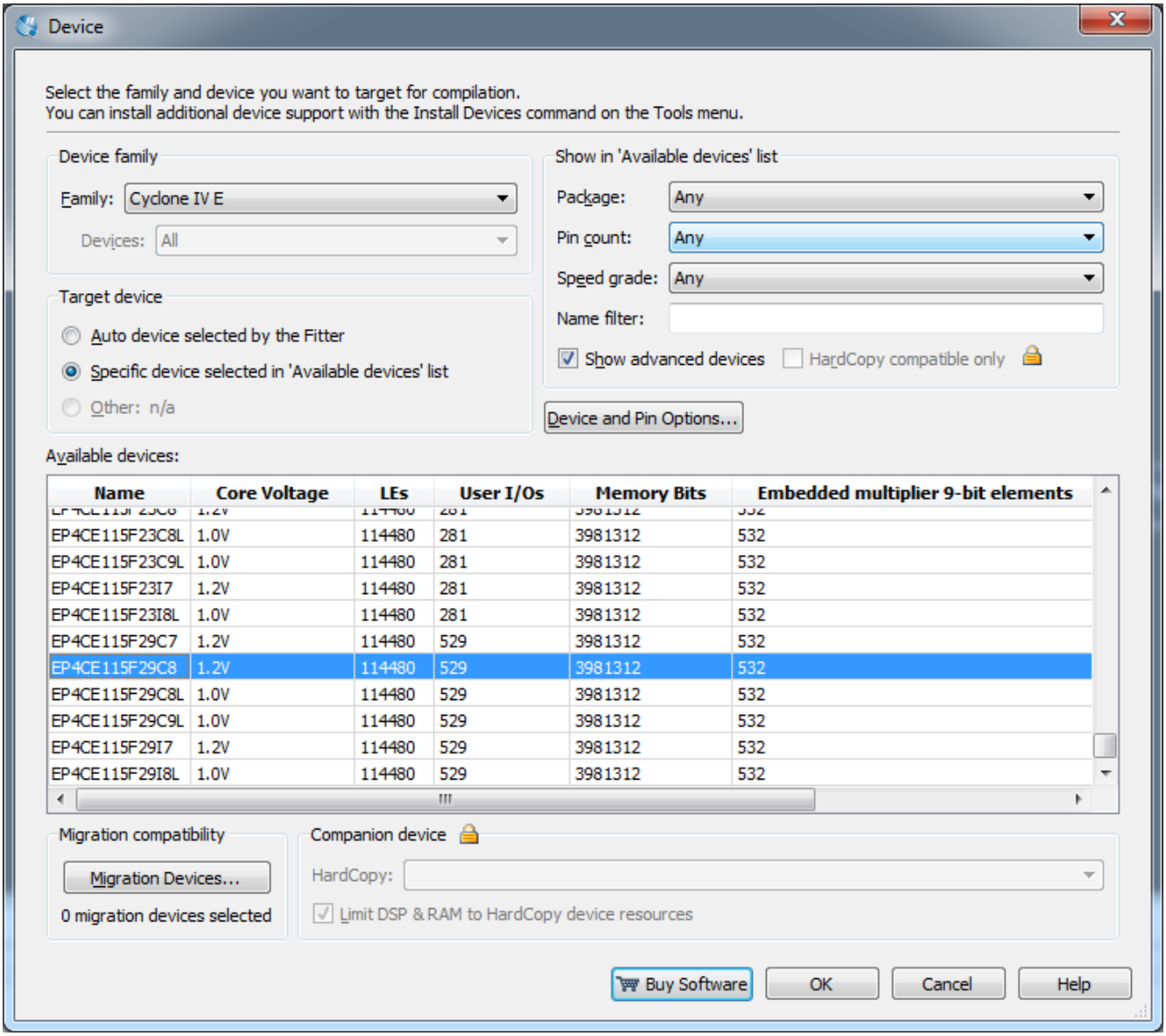

Figure 11. Device selection screen, with new target highlighted.

Capturing a new design based on VHDL

In this final step, you will create a version of the Lab 3 project (based on a Karnaugh map), but this one will use VHDL as the design entry method.

While still in Quartus with your project open, select File, New…, and choose “VHDL” as the type of file, as shown in Figure 12. You will be presented with a blank window for editing. In that window, enter the text to represent your K-map equation, which will resemble that in Figure 13, with the main difference being that you will not have the question marks.

Save your VHDL file. The file name has to match the name given to your VHDL entity, and the file should be saved in your current project directory. Quartus should make it your top-level design file automatically.

Compile your project. Pin assignment do not matter, as long as you are able to successfully compile. If you encounter problems, post on Piazza.

By September 24 at 4:30 PM (seminar time) submit the text of your VHDL file. You may do this on T-Square, or bring a hardcopy to class.

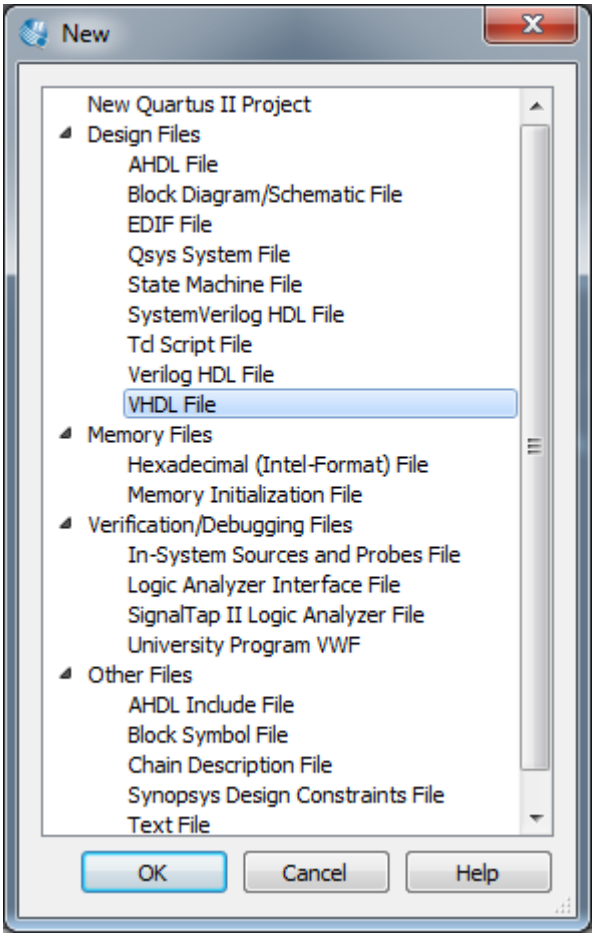

Figure 12. Selection of "VHDL File," lower on the list than "Block Diagram/Schematic File."

```
-- Lab 4 assignment

library ieee;
use ieee.std_logic_1164.all;

entity Lab4 is

	port
	(
		A, B, C, D	: in std_logic;
		Y		: out std_logic
	);

end entity;

architecture rtl of Lab4 is
begin

	Y <= ????

end rtl;
```

Figure 13. Representative code, with question marks for the actual Boolean equation.