Objectives

- Understand the design flow used in this course
- Walk through a completed design and understand how a hierarchical block diagram is made
- Gain familiarity with using Xilinx tools to create and simulate VHDL models of digital circuits
- Learn how to use the following in VHDL:
  - Comments
  - Entity declaration
  - STD_LOGIC and STD_LOGIC_VECTOR data types
  - Architecture body
  - Signals
  - Concurrent assignment
  - Component instantiation
- Learn how to test a design
  - Manually forcing values and examining outputs
  - Using a simple self-checking VHDL test bench

Functional Specification

- In this lab you will design, build and test an adder/subtractor with the following specifications:
  - I/O
    - A: First operand, 4-bit two’s complement
    - B: Second operand, 4-bit two’s complement
    - SUBTRACT: Control signal to select add or subtract
    - SUM: Sum of difference of A and B, 4-bit two’s complement
    - OVERFLOW: Overflow flag
  - Operation
    - The adder/subtractor is a combinational circuit that computes the sum or difference of two numbers. Both inputs and the output are 4-bit two’s complement numbers. The add/subtract control signal is asserted high (1 = subtract). The overflow flag is asserted high when the sum cannot be represented in four bits.
  - Testing
    - Full adder component is tested exhaustively
    - Ripple-carry adder and top-level adder/subtractor are tested with a subset of cases

Discussion

In this lab you will go through the design flow for the adder/subtractor except for the last step, test in hardware. The design flow used in this course consists of the following steps:

- Specification
- Design
  - Algorithms
  - Block Diagrams
  - State Diagrams
  - State Tables
- Translate to VHDL
- Test in Simulator
- Test in Hardware
The specifications for the adder/subtractor are given above. Specifications will usually be given to you in labs. You will write the specifications for your project later in the course. The complete design for the adder/subtractor has been done for you. This serves as a detailed example of a hierarchical block diagram and allows you to quickly begin writing VHDL code. You will do more of the design in later labs and all of the design for your project. The rest of this section describes the design. Read it carefully, you are expected to understand it as if you designed it yourself and you may see parts of it on an exam.

The first step in design is to develop an algorithm or algorithms to meet the specification. Briefly, the algorithm for this design is as follows:

- We know how to make a ripple-carry adder (RCA) so we will start with that
- Note that \( A - B = A + (-B) \), so subtraction can be performed using an adder by negating the second operand before adding
- A two’s complement number can be negated by complementing each bit and adding ‘1’ to the least-significant bit (LSB)
- When the add/subtract control signal is ‘1’, complement each bit of B and add ‘1’
- To detect overflow, note that
  - When adding two numbers A and B:
    - If either one is positive and the other is negative, the sum cannot overflow
    - If both are positive or both are negative, the sum might overflow
  - When subtracting B from A:
    - Same rules as for addition, except the sign of B is negated, so
    - If both are positive or both are negative, the sum cannot overflow
    - If either one is positive and the other is negative, the sum might overflow
  - The most-significant bits (MSBs) of the input operands and the output sum are examined in conjunction with the add/subtract control bit. If the sign of the sum is not consistent with the signs of the operands and the operation, there is overflow. For example, if the signs of the operands are both positive and the operation is add, the sign of the sum must be positive or there is overflow.

The second step in design is to make block diagrams of hardware to implement the algorithm(s). Completed block diagrams are provided for you. **Follow this description and make new block diagrams on paper, using the supplied block diagrams as little as possible.** Here is a description of how to make block diagrams from the algorithm:

- Use a top-down approach. This means start with the “Top Level” block diagram, which includes the main components. Next, recursively drill down into every component, making block diagrams for each, until you get down to truth tables or gates.
- Draw a large dotted rectangle for the top-level block. Leave room to add a few blocks inside the rectangle. Label it `AdderSubtractor_4bit`. Write the input and output signal names outside the rectangle (A, B, SUBTRACT, SUM, OVERFLOW) and indicate bit indices for signals that are wider than one bit.
- We are starting with an RCA, so start by putting an RCA inside the top-level component.
  - Draw a solid rectangle for the RCA block
  - Label all inputs and outputs inside the rectangle. We will make it a component (so we can reuse it later) and instantiate a copy, so label the block using the form `<instancename>:<componentname>`
  - Draw arrows pointing into the block for inputs and arrows pointing out of the block for outputs
Both operands and the sum are 4-bit signals, indicate that with a slash and the number ‘4’ on each line.

To subtract, the B operand is complemented. To add, B is unchanged. Add a block to the diagram that does this, call it InvertOrPass.

- Label the inputs and outputs. This will just be a block, (we will implement it directly in the top level, not make a component) so label the block using the form <componentname>.
- The RCA uses the output of the block, so draw an arrow from the block to the B input of the RCA. This is an internal signal so give it a name.
- Draw an arrow on the diagram pointing into the B input of the InvertOrPass block.
- The block needs to know if the operation is add or subtract, so draw an arrow into it and use the SUBTRACT signal.
- For subtract operations, ‘1’ has to be added in addition to complementing the second operand. We can use the carry-in to the RCA to add one when the operation is subtract or add zero when it is add. If we bring the SUBTRACT signal into the carry-in, it will do exactly what we want, so add that connection to the diagram.

Add a block to detect overflow. The algorithm uses the add/subtract control signal, the MSBs of the operands and the MSB of the sum to determine if there is overflow.

- This will just be a block, so label the block using the form <componentname>.
- Add arrows into the block for the two operand MSBs. These come from the A and B inputs. Since the MSBs are only one bit from a 4-bit signal, indicate which bit is being used by labeling the arrows as A(3) and B(3) respectively.
- Add an arrow for the MSB of the SUM signal. Since SUM is an output of the top level that is being used as an input to an internal component, a separate signal must be used for the internal connection. Label the SUM internal signal as SUM_int, which becomes SUM when it leaves the top level. Label the Sum MSB into the overflow detect block as SUM_int(3).
- Add an arrow to bring the SUBTRACT signal into the block.
- Add an arrow for the output, the overflow signal.

Drill into the RCA and design it. Create a new block diagram for it.

- Draw four full adders, one for each bit. A full adder takes three input bits, a, b and a carry-in. It adds them to produce a sum bit and a carry-out bit.
- Each full adder is an instance of a component you create once. Name the component FullAdder. Give each instance a unique name, such as FA3, FA2, FA1, and FA0. This is indicated inside the full adder block as FA3:FullAdder, FA2:FullAdder, etc.
- Connect the carry-out of each full adder to the carry-in of the next. Bring the carry-out of the most significant full adder out of the RCA. We are not using it in the top level, but many other designs will use it and we want to be able to re-use the RCA as much as possible to save work later. Likewise, include a carry-in input to the RCA and wire it to the least significant full adder. We need it for this design and it is also often used in other designs.
- The carries are internal signals, so name them. Instead of making each a different STD_LOGIC signal, use a single STD_LOGIC_VECTOR. It is more convenient here, and will be a lot more convenient later.

Drill into the InvertOrPass block that complements B when the operation is subtract.

- Each bit of B must be inverted when the operation is subtract and left unchanged when the operation is add. Four XOR gates work well for this.
- Since we don’t expect to use this in another design and it is easy to code in VHDL, we will not make a component out of this block. Therefore, we don’t need to give it an instance name, so just label the block as InvertOrPass.
• Drill into the overflow detect block.
  o When the gates needed to make a logic function are not obvious, start by making a truth table.
  o There are four inputs to the truth table, SUBTRACT, A(3), B(3) and SUM(3), so the table has $2^4 = 16$ entries. The entries go from “0000” to “1111”
  o Instead of trying to go from the inputs straight to the overflow output, add some intermediate columns to make it easier
    ▪ Add a column for the operation, where you can write “addition” or “subtraction”
    ▪ Add columns for the sign of A, B and SUM where you can write “positive”, “negative”, “+”, or “-” as you wish.
    ▪ Add a column for the output, OVERFLOW
  ▪ Complete the truth table line by line using the algorithm to guide you. For example, the first row of the table in the block diagram provided to you is “0000” which corresponds to subtract, A is positive, B is positive and SUM is positive. When two positive numbers are added, the sign of the sum must be consistent. In this case it is positive so there is no overflow. The second row is the same, except the sum is negative. This is not consistent, so there is overflow. For the third through sixth rows, one operand is positive and the other is negative, so there is no overflow regardless of the sign of the sum.
  o We don’t expect to use this block again (if we do there is a better way to do overflow detection anyway) so we will not make a component for it. Therefore it doesn’t need an instance name, so just label the block as OverflowDetect.

• When you think you have finished the block diagram review it carefully. Any mistakes made in the block diagram will be mistakes in the translated VHDL. You can either fix them now, or you can code them→test and find mistakes→fix the block diagram→code them again. The choice is yours. Here are some things to look for:
  o You may want to change some signal names for several reasons
    ▪ to be more descriptive
    ▪ because a better name is apparent after the full design is made
    ▪ because two names are easily confused with each other
    ▪ etc.
  o Make sure the number of bits is correct for each signal. Any signals that are two or more bits should be indicated with a slash and the number of bits.
  o If a signal is split or concatenated with another signal, label each wire involved with the resulting signal, using indices as needed.
  o Every block should have its name indicated. If it is a component, the instance name should also be indicated as $<\text{instancename}>:<\text{componentname}>$. This will correspond nicely to the VHDL code that instantiates the component. If the block is not an instantiation of a component, such as OverflowDetect, the name is not needed for the VHDL code. However, it should be given in the comment for the code that implements it.
  o Every block, starting with the top level and working down to truth tables or logic gates, should have another block diagram that shows its internal structure
  o Make sure that all the ports of a component are shown on the block diagram that drills into that component. Make sure the port names and the bitwidths are the same.
  o Make sure that every internal signal has a name.
  o If an output of a block is used as an input to a sub-component internally, make sure to have an internal signal as well. Give it the same name as the output, but with “_int” appended to it.
Make sure that all inputs to a component have exactly one driver, even if it is a constant one or zero. A driver is a signal that is an input to the top level or the output of another component.

An output of a component can go to more than one component input. If an output is not used, connect it to “open.” This will tell most VHDL compilers that you know the signal is unused so it should not generate a warning. Unfortunately, the current version of the Xilinx compiler still generates a warning.

The third step in design is to make state diagrams for all finite-state machines (FSMs). The fourth step is to make state tables for the FSMs. This is a combinational circuit, so there are no FSMs.

The block diagrams are provided as Microsoft Visio files. You can use them as templates for block diagrams you create later. Update the title block on each drawing. This includes team members name and user ID in the TEAM block, and the user ID and date of the person who checked off the drawing in the CHECKED block. The DRAWN BY block is for the person who drew the majority of the drawing, so do not change it in this case. If you use this drawing as a template in future labs, you will update the DRAWN BY block appropriately.

After the design is completed, it is translated to VHDL. If you make changes to the design during the translation process, update the master design documents so they are consistent. You will use them for debugging. Any differences between the design documentation and the VHDL code will be confusing.

**Lab Exercises**

1) Create a directory somewhere on your P: drive named CMPEN371. Create a sub-directory named Lab01 in the CMPEN371 directory.

2) Start Xilinx ISE. You can find it under Engineering Programs > Xilinx ISE Design Suite 14.7 > ISE Design Suite 14.7 > ISE Design Tools > Project Navigator. You may want to create a shortcut for your desktop.

3) Create a new project. Select File > New Project in Xilinx ISE
   a. Name it Lab01_xyz123_abc456, replacing xyz123 and abc456 with each team members email ID.
   b. For location, select the .\CMPEN371\Lab01 directory you created on your P: drive.
   c. Select HDL for Top level source type then select “Next” to continue.
   d. Family is Artix7 (Nexys-4 or Nexys-4 DDR)
   e. Device is XC7A100T (Nexys-4 or Nexys-4 DDR)
   f. Package is CSG324 (Nexys-4 or Nexys-4 DDR)
   g. Speed is -1 (Nexys-4 or Nexys-4 DDR)
   h. Synthesis tool is XST
   i. Simulator is ISim
   j. Preferred language is VHDL

4) Start by translating and testing the smallest components. Create a new VHDL file for your FullAdder component.
   a. Select Project > New Source
   b. Type is VHDL Module
   c. Name it FullAdder
d. Make sure add to project is checked

e. You can enter the port signals using the wizard or type them manually in your VHDL file after it is created. Work off of your block diagrams to use the right signal names.

f. Replace the auto generated header comment with the template provided on ANGEL (Resources folder, then VHDL folder). Complete the comment with your information.

g. The auto generated line comments above the entity declaration are not needed, remove them.

h. Complete the port signals in the entity declaration if needed. Modify the port signal list to comply with the VHDL coding standards used in this course (posted on ANGEL, Resources folder, then VHDL folder). This includes using only one signal per line, lining up the colons and the signal type, etc.

i. The FullAdder will use concurrent assignments, so it is a Dataflow model instead of a behavioral model. Modify the architecture accordingly (replace Behavioral with Dataflow in two places).

5) Translate your full adder to VHDL.

a. Declare signals for every wire that is not an entity port signal. All of these signals should be named on your block diagram. If not, update your diagram with the name.

b. Use concurrent assignment statements to drive the wires and the full adder outputs.

c. At the top of the Design panel in ISE (where the hierarchy of the FPGA target and your VHDL files are shown in a tree) select the Simulation button (next to the Implementation button, ask for help if you cannot find it). In the tree, click your VHDL file once to select it. Go to the window below it and expand the ISim Simulator entry. Double-click on Behavioral Check Syntax. This will do a basic syntax check on your design. Fix any errors it reports before going to the next step.

6) Simulate manually in ISim.

a. In order to test a circuit, you must know the expected outputs. Make a truth table for the full adder.
   i. Inputs are A, B and C_in
   ii. Outputs are C_out and SUM
   iii. Together, C_out and SUM are a 2-bit binary number and the value is the number of inputs that are ‘1’. This is why a full adder is also called a 3,2 counter; it counts the number of ‘1’ s in three inputs and outputs a 2-bit total.

b. Double-click Simulate Behavioral Model under ISim Simulator. This will start ISim (be patient, it may take some time to start).

c. You should see all port signals and internal signals. Note that they have the value ‘U’, which means uninitialized.

d. In the waveform window, right-click on the port input signal “A”, select force constant, and force it to 0. Repeat this for B and C_in. Note that they are shown in lowercase; VHDL is case-insensitive so they are the same signals.

e. Use the tool bar to run the simulation for the time specified in the toolbar. Note how the signals and port outputs now have values of ‘0’. Compare the output signals to your truth table, are they correct?

f. Force A to ‘1’, leaving B and C_in at ‘0’. Are the output signals correct?

g. You could repeat this for all input combinations, but by now you are probably thinking there has to be a better way. The better way is to make a test bench.

7) Make a VHDL test bench and simulate in ISim

a. Start with the full adder test bench provided on ANGEL. Download a copy and put it in your project directory. Select Project > Add Source, then select the test bench file.

b. Open the test bench and note the following:
i. The entity has no port signals. This is because it is a self-contained model used only for simulation. It does not need any inputs and does not drive any outputs.

ii. The component to be tested, FullAdder, is declared. Such a component is called the Unit Under Test (UUT) by most digital designers.

iii. Signals are declared for the UUT inputs and outputs.
   1. Inputs are initialized to ‘0’. Otherwise, they would begin as ‘U’ (uninitialized)
   2. Signals can have the same names as component port names, and they often do.

iv. The FullAdder is instantiated in the architecture body, mapping the ports to signals with the same name. This is called named association.

v. The rest of the architecture body is a VHDL process. You will learn more about processes later. This process is used like a “normal” programming language in that each line is executed sequentially.
   1. The first line, “wait for 100 ns,” waits for 100 ns (surprised?). The Xilinx simulator will simulate a global reset on the FPGA during this time.
   2. The next line assigns ‘0’ to each of the three inputs. Note that this may be the only place in this course where it is acceptable to put more than one statement on a line.
   3. The next line waits for 10 ns, this gives the logic gates time to transition and set the circuit outputs.
   4. The next three lines are actually one statement, broken up for readability.
      a. The assert statement tells the simulator to check for expected values. In this case, check that C_out = ‘0’ and SUM = ‘1’.
      b. The report part of the assert statement tells the simulator the text to display if the assertion is false.
      c. The severity part of the assert statement tells the simulator how to classify an assertion failure. Possible values are note, warning, error, and failure. A severity of failure tells the simulator to halt.
   5. This code (assign inputs, wait, assert) is repeated for each test. The first two are given, you will add the rest
   6. The final wait statement causes the simulator to finish the simulation.

c. Double-click Behavioral Check Syntax then Simulate Behavioral Model. If ISim is still open you will get an error. If so, close it and try again.
   i. The simulator will automatically run the test bench, and it will fail on the second test (unless you did something both strange and lucky in your full adder to make it pass).
   ii. The console window at the bottom of ISim will report the failure and the message in the assert statement. The simulation will also halt because we used ‘failure’ for severity.
   iii. Click the Zoom to Full View icon on the toolbar (it is one of the magnifying glass icons) to see the full simulation. In the waveform window, drag the vertical yellow line back and forth and note that the value of each signal is indicated for the time at the yellow line.
   iv. Drag the yellow line to the end of the simulation. Look at the inputs and the outputs. Do they match your truth table? If your circuit is correct it should. Go back to the test bench and note that the assertion for the second test is wrong. Fix the assertion and run again. Hopefully it works. If not, go to the next step anyway.

d. Add code to the test bench to do an exhaustive simulation. This means run every possible combination of inputs. When you are done, check the syntax and fix any mistakes. If ISim is still open, you can click the Re-launch button on the toolbar or select Simulation > Relaunch from the menu instead of closing it every time you make a change to your VHDL.
i. If your design fails a test, you want to “traverse the design hierarchy” (see course objectives) to debug the code. So far, the test bench is the top of the hierarchy and it only goes down to the full adder. The hierarchy will be deeper by the end of the lab.

ii. On the left side of the screen you should see the top of the hierarchy, FullAdder_tb.
   1. Click the arrow next to it to expand. You should now see uut, the name of the instance of the full adder, and :stim_proc, the name of the process that does the simulation.
   2. Click on uut, you will notice that the window immediately to the right (but still to the left of the waveform window) has the internal signals used in the full adder. Drag them to the waveform window.
   3. Restart the simulation using the toolbar icon or Simulation > Restart, then zoom to full view. You should now see the values of the internal signals you added.

e. Here are the steps to debug a design:
   i. Move the yellow line to the point where the error occurred, probably the end of the simulation.
   ii. Look at the inputs, and compare the simulated output to the output you would expect. If the simulated output is the same as the expected output, your test bench is wrong. If the simulated output is different than the expected output, the test bench has found a problem for you.
      1. Drill one level into the hierarchy. Add internal signals as needed to trace the incorrect output back to each of the inputs that affect it, then restart the simulation. Your block diagram will help you identify these signals.
      2. Work backwards from the incorrect output. Look at the inputs to the block that produced the output.
         a. If the inputs to the component should produce a different output, then the problem is that component. Drill into it and repeat the process. Also, make a note to yourself that you should have fully tested the component before you used it because it would have saved you this trouble.
         b. If the inputs to the component should produce the same output, then the component is OK (at least for now) and the wrong inputs were applied to the component. You don’t yet know which input is wrong, so for each input:
            i. Work backwards to the component that produced it
            ii. Look at the inputs to that component, check if they should produce it.
                If so, the component is OK. If not, drill into the component and debug.

f. Note: By default, ISim runs for the time shown in the toolbar, 1.00 µs (1000 ns). If your test bench takes longer than this (total of all the waits in your test bench), you can either change the run time in the toolbar, press the “Run for the time specified on the toolbar” icon as many times as needed, or reduce the wait times in your simulator.

g. The basic idea in this course is to make a component, test it extensively so you know it works properly, then re-use it to make larger designs. As you do this, you will accumulate a set of components that always work. You will use these components to make larger designs and only have to debug the top level and any new components. If you do not do this, your larger designs will become exponentially more difficult to debug and get working.

8) Complete and test the design for the adder/subtractor by working from the bottom up.
   a. Create and test the 4-bit ripple-carry adder
      i. Create a new VHDL file. Replace the auto generated header comment with the template provided on ANGEL. Complete the comment with your information. Clean up the entity declaration to meet course coding standards.
ii. This design will instantiate components and wire them together. This is called a Structural model instead of a behavioral model. Modify the architecture accordingly (two places).

iii. Declare the internal signals from the block diagram and comment them.

iv. Instantiate four full adders. Add a comment for the group of them. Don’t forget to declare the component.

v. Make a test bench. Exhaustive simulation would be best, but just create enough test cases to give you confidence it works. Test your component and debug it until it works.

b. Create and test the top level for the adder/subtractor.

i. Create a new VHDL file. Replace the auto generated header comment with the template provided on ANGEL. Complete the comment with your information. Clean up the entity declaration to meet course coding standards.

ii. Although there will be some Dataflow code for small blocks, this is a primarily a Structural model. Modify the architecture accordingly.

iii. Declare the internal signals from the block diagram and comment them.

iv. Instantiate the ripple-carry adder. Add a comment for it.

v. Add logic for the InvertOrPass block. Add a comment for it.

vi. Add logic for the OverflowDetect block. Add a comment for it.

vii. If you do not have comments for everything yet, ask yourself why.

viii. Make a test bench. Exhaustive simulation would be best, but just create enough test cases to give you confidence it works. Test and debug your adder/subtractor.

ix. If you find you need to drill into the ripple-carry to debug the adder/subtractor, you did not test it well enough. In future labs, if your components are not well tested you will probably be overwhelmed by the number of problems in your top level design.

9) After everything works, review your code. Make sure it meets style guidelines. If you are adding comments, fixing indentation, etc. now, you missed the benefits of doing it from the start. Style guidelines are there to make it easier to read and debug your code, which includes you when you are debugging your own designs.

Deliverables (50 total points)

1) Complete the lab exercises.

2) Demonstrate your work at the beginning of the next lab period.

3) Write one lab report per team following the lab guidelines document posted on ANGEL.

   a. Start with the lab report template posted on ANGEL. NOTE: Any text enclosed in braces, { }, is to be replaced or deleted (delete the braces also). Don’t forget to answer the questions at the end of the report.

   b. Turn in one hard copy of the report. The hard copy includes the block diagrams you made while working through the discussion.

   c. Submit one electronic copy via ANGEL drop box. Name it Lab01_xyz123_abc456.docx, replacing xyz123 and abc456 with each team members email ID. Submit all files needed to recreate and test your design, such as all VHDL files, Tcl files, the UCF file, etc. Do not submit the entire Xilinx directory.

4) Deliverables are due at the beginning of your next lab period.