# Overall design flow:[[1]](#footnote-1)

**Big Picture:** In this computer exercise, you will design, write, and test a 32-bit Arithmetic Logic Unit (ALU). As a part of this process, you will design and write a self-checking testbench. Later, you will use this ALU in future assignments as you build a fully-functional MIPS microprocessor.

# Deliverables

1. Abbreviated README containing:
   1. Initial test vectors table (**due COB Lesson 14**)
   2. Additional test vectors
   3. Explanation/rationale of all test cases
   4. Simulation waveform
   5. Explanation of results
   6. Synthesis results (hardware picture and report)
   7. Interpretation of synthesis results
2. VHDL ALU module
3. VHDL testbench

# Background

You should already be familiar with the ALU from Section 5.2.4 of the textbook (Figure 1). The design in this lab will demonstrate the ways in which VHDL makes hardware design more efficient. It is possible to design a 32-bit ALU from 1-bit ALUs (i.e., you could program a 1-bit ALU incorporating your half adder from earlier in the course, combine it to create a full adder, chain four of these together to make a 4-bit ALU, and chain 8 of those together to make a 32-bit ALU). However, it is altogether more efficient, both in time and lines of code, to code it succinctly in VHDL using behavioral modelling.

Figure 1 on the next page shows the schematic for a N-bit ALU along with a table of it functions. Before you write any VHDL code, it is important that you can visualize the hardware you intend to build. Since HDL is designed to *describe* hardware, you can imagine why it would be critical to have a plan for what hardware you want to build. In this sense, creating a schematic of your design is like creating pseudocode for traditional programming languages. Look carefully at the layout/configuration of the hardware in Figure 1 as it will make it easier to implement your design for this exercise.



|  |  |  |
| --- | --- | --- |
| F2:0 | Function |  |
| 000 | A & B |  |
| 001 | A | B |  |
| 010 | A + B |  |
| 011 | not used |  |
| 100 | A & ~B |  |
| 101 | A | ~B |  |
| 110 | A - B |  |
| 111 | SLT |  |

Figure 1 – Schematic and table of functions for a N-bit ALU

# Creating a Test Plan

Before taking on any serious design project you must always ask yourself early on “How will I test my design to verify that it will meet requirements?” No matter how brilliant and elegant your design is, if you are unable to *prove* that it meets requirements through testing then your project will be considered a failure. It is important to note that not all designs are easily testable (or producible)!

Thus, we start this computer exercise getting you thinking about how you will test your ALU. Before writing your ALU module or testbench, it is prudent to think through a set of input vectors (test cases). We need to use an appropriate set of test vectors to convince a reasonable person that your design is probably correct. You want to fully test *all* functions of your ALU. To that end, complete Table 1 on the next page to verify that all seven ALU operations work as they are supposed to. **Note that the A, B, and Y values are expressed in hexadecimal**.

Provide a copy of your completed table on Bitbucket for your instructor to review by COB Lesson 14.

Once your initial table is completed, add in at least three test vectors of your own. Explain why you think *each* of your test cases and each of the provided ones are useful test cases. Think about corner cases and how you may be able to break your design. Identify at least one limitation of your ALU design, and create a test case you can use to verify the limitation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # | Test | F[2:0] | A | B | Y | Zero |
| 1 | ADD 0+0 | 2 | 00000000 | 00000000 | 00000000 | 1 |
| 2 | ADD 0+(-1) |  | 00000000 | FFFFFFFF |  | 0 |
| 3 | ADD 1+(-1) |  | 00000001 |  |  | 1 |
| 4 | ADD 0xFF+1 |  | 000000FF | 00000001 |  |  |
| 5 | SUB 0-0 |  | 00000000 |  |  |  |
| 6 | SUB 0-(-1) |  | 00000000 |  |  |  |
| 7 | SUB 1-1 |  | 00000001 |  |  |  |
| 8 | SUB 0x100-1 |  | 00000100 |  |  |  |
| 9 | SLT 0,0 | 7 | 00000000 |  |  |  |
| 10 | SLT 0,1 |  | 00000000 |  |  |  |
| 11 | SLT 0,-1 |  | 00000000 |  |  |  |
| 12 | SLT 1,0 |  | 00000001 |  |  |  |
| 13 | SLT -1,0 |  | FFFFFFFF |  |  |  |
| 14 | AND 0xFFFFFFFF, 0xFFFFFFFF | 0 | FFFFFFFF |  |  |  |
| 15 | AND 0xFFFFFFFF, 0x12345678 |  | FFFFFFFF |  |  |  |
| 16 | AND 0x12345678, ~0x02040608 | 4 | 12345678 |  |  |  |
| 17 | AND 0xAAAAAAAA, ~0xFFFF0000 |  | 00000000 |  |  |  |
| 18 | OR 0xFFFFFFFF, 0xFFFFFFFF |  | FFFFFFFF |  |  |  |
| 19 | OR 0x12345678, 0x87654321 |  | 12345678 |  |  |  |
| 20 | OR 0x00000000, 0xFFFFFFFF |  | 00000000 |  |  |  |
| 21 | OR 0x00000000, 0x00000000 |  | 00000000 |  |  |  |
| 22 | OR 0x00000000, ~0x00000000 |  | 00000000 |  |  |  |

Table 1 - Test vectors to verify the functionality of the ALU.

# VHDL Design Code

Now that you have a design and test plan, you can begin implementation. Implement the ALU from Figure 1 with behavioral VHDL code. The encoding scheme for arithmetic operations, f, was chosen to make the hardware solution elegant and efficient.

An adder is a relatively expensive piece of hardware. Be sure your design uses *no more than one adder*. Figure 1 should be useful in helping you write your behavioral VHDL code. HDL Example 4.6 in your textbook provides a good example of how to behaviorally describe a multiplexer.

You will need to add the following line to your import statements so that you can perform addition of std\_logic\_vector ports/signals: “**use** ieee**.**std\_logic\_signed**.all;**”.

Name the file alu.vhd. It should have the below entity declaration. (Failure to adhere to either of these two requirements will give you grief on Lab 4). The output zero should be ‘1’ if y is equal to zero.

|  |
| --- |
| **entity** alu **is**  **port(**  a**,** b **:** **in** std\_logic\_vector**(**31 **downto** 0**);**  f **:** **in** std\_logic\_vector**(**2 **downto** 0**);**  y **:** **out** std\_logic\_vector**(**31 **downto** 0**);**  zero **:** **out** std\_logic  **);**  **end** alu**;** |

# VHDL Testbench

**Build a self-checking testbench to test your ALU** **in ISim**. Pages 220-224 of your textbook provide helpful information on building testbenches. In particular, an example of creating a self-checking testbench is provided on page 222. Do not forget to **test both Y and Zero outputs**! Take a screenshot of the simulation console window and of the test vector waveforms once you have a successful test run. The console window should display a message that all of the tests were successful and have no errors or warnings.

You may need more than one screenshot to make the test vector waveforms readable for all tests. Your signals should be displayed using “hexadecimal” radix and in the following order (top to bottom): a, b, f, y, zero. Make the output signals a different color to make help differentiate the signals. Clearly label each test on your simulation (differently colored boxes around each test, associated with test numbers, can very clearly tell a story) for full points. Compare your simulations results to the predictions in your initial test vector table.

# Analysis

Now you need to determine how *efficient* your design was. Open up the “synthesis report” in the Design tab. Look through this log file until you find a section *like* (not identical, as the results of your synthesis will be slightly different) this:

|  |
| --- |
| =========================================================================  HDL Synthesis Report  Macro Statistics  # Adders/Subtractors : 23  16-bit adder : 23  # Multiplexers : 4  16-bit 64-to-1 multiplexer : 4  ========================================================================= |

Take a screenshot of these results for your lab notebook. Also take a screenshot of the RTL (real-time library) and Technology schematics that show how your hardware was synthesized.

The RTL viewer opens a netlist (NGR) file as a gate-level schematic. This schematic is not optimized for your hardware. The Technology view, on the other hand, is optimized for your specific hardware platform (i.e., it shows your design in terms of FPGA logic elements).

On the ISE top menu click on Tools🡪Schematic Viewer and then RTL or Technology. Start from the top level. You can double-click on object to drill down further as desired to show how pieces of your design are synthesized (or right-click to hide details). Compare the schematics to the synthesis report. What do these results mean?

|  |  |
| --- | --- |
| Wrapping Up… |  |

1. Add a Feedback section to the end of your README. Include the following information:

**Number of hours spent on CE3:** \_\_\_\_\_\_\_\_ (no points associated with this unless you leave it blank)

**What did you learn?**

**Suggestions to improve CE3 in future years:**

1. The table below is how the points will be distributed for this assignment:

|  |  |  |
| --- | --- | --- |
| **Item** | **Points** | **Out of** |
| Initial test vectors |  | 15 |
| Additional test vectors |  | 5 |
| Test case rationale |  | 10 |
| ALU module code |  | 10 |
| Test bench code |  | 15 |
| Simulation waveform |  | 20 |
| Explanation of sim results |  | 10 |
| Synthesized H/W pictures |  | 5 |
| Synthesis report picture |  | 5 |
| Synthesis interpretation |  | 5 |
| **Total** |  | **100** |

1. Modeled after a lab provided with instructor notes for Digital Design and Computer Architecture, David Money Harris & Sarah L. Harris, 2nd Edition [↑](#footnote-ref-1)