CMPE 260
Digital System Design II
Laboratory Exercise 3: Arithmetic Logic Unit
Revised: 08-05-2017

This exercise develops an N-bit arithmetic logic unit (ALU) for use with an FPGA. The objective of this exercise is to understand and be able to implement basic arithmetic operations in hardware using generics. An ALU is written and tested in VHDL, then rechecked in hardware.

Background

Many of the operations performed by computers can be broken down into arithmetic or logic operations. It would be difficult, if not impossible, to have to reimplement such operations in software every time a program was written. Thus a dedicated piece of hardware, known as an ALU, was created and included in almost every processor since its inception. This exercise builds off of what you already created in Exercise 1, where you created a more basic ALU.

Prelab Work

- Draw a schematic of the ALU following the specifications laid out below. The internals of each component must also be shown at least once. For example, showing the gates inside one full adder, then drawing a full adder block thereafter is sufficient. This is not necessary for the multiplexers. The connections between components must also be shown.
- Become familiar with the design of a multiplier. Consider what needs to be changed to make it a generic size.
- Review how to use text I/O in VHDL.
- Print the sign-off sheet.

Program Specification

The design must meet the following specifications and use all components listed at least once. Some will be used multiple times. Table 1 contains the required operations of the ALU, along with each control code.

Table 1: The Operations, Instructions, and Controls of the ALU

Operation	Instruction	Control
Addition	ADD	0100
Subtraction	SUB	0101
Multiplication	MUL	0110
Bit-wise or	OR	1000
Bit-wise not	NOT	1001
Bit-wise and	AND	1010
Bit-wise xor	XOR	1011
Shift Left Logical	SLL	1100
Shift Right Logical	SRL	1101
Shift Right Arithmetic	SRA	1110

Ripple Carry Full Adder

The full adder has the following features:

- Functions:
  - Multi-bit addition.
  - Multi-bit subtraction. Thus all multi-bit inputs and outputs are in two's complement.
- Parameters:
  - The number of bits of the two addends. To be called `n`.
- Inputs:
  - `A` (`n` bits): First addend, or minuend if subtracting
  - `B` (`n` bits): Second addend, or subtrahend if subtracting
- Outputs:
  - `Sum` (`n` bits): Sum, or difference if subtracting

The ripple carry full adder must be structural, however, the full adders it is made up of can be behavioral, dataflow, or structural. Use the for generate syntax to instantiate the individual full adders inside of the ripple carry adder.

Logic Unit

The logic unit has the following features:

- Functions:
  - Bitwise XOR.
  - Bitwise AND.
  - Bitwise OR.

  - Bitwise NOT.

- Parameters:
  - The number of bits of the two inputs. To be called `n`.
- Inputs:
  - `A` (`n` bits): First input.
  - `B` (`n` bits): Second input. Ignored for the bitwise NOT operation.
  - `Control` (Implementation defined): To determine which operation is performed. See Table 1.
- Outputs:
  - `Output` (`n` bits): Result of the appropriate bitwise operation.

This should already be mostly completed from Exercise 1. The only additional functionality required here is adding the ability to select which operation the entity will output.

Shifter

The shifter has the following features:

- Functions:
  - Logical left shift: Shift to the left and append zeros.
  - Logical right shift: Shift to the right and append zeros.
  - Arithmetic right shift: Shift to the right and sign extend the MSB.
- Parameters:
  - The length of the value to be shifted, in bits. To be called `n`.
- Inputs:
  - `A` (`n` bits): Value to be shifted.
  - `B` (`n` bits): Number of bits to shift
  - `Control` (Implementation defined): To determine which shift is performed. See Table 1.
- Outputs:
  - `Output` (`n` bits): Result of the appropriate shift.

This should already be mostly completed from Exercise 1. The added functionality needed here is the addition of an Arithmetic Shift Right and the ability to choose which shifting operation is output by the entity.

Carry Save Multiplier

The multiplier has the following features:

- Functions:
  - Multi-bit multiplication.
- Parameters:
  - The output length, in bits. To be called `n`.
- Inputs:
  - `A` (`n`/2 bits): First factor.
  - `B` (`n`/2 bits): Second factor.
- Outputs:
  - `Product` (n bits): Product. Note that this is twice as long as the factors.

The carry save multiplier must be structural, however, the components of the multiplier do not have to be. Figure 1 shows a four-bit carry save multiplier. Note that the rightmost adder could be a half adder, as the carry in is set to zero. This multiplier functions in a similar way to humans performing multiplication with two decimal numbers on paper. In order to implement such a mutliplier, consider using an array of `std_logic_vector`s to hold the intermediate results such as the results of the AND operation, sums, and carry outs. The use of the `+` and `*` operators is not allowed.

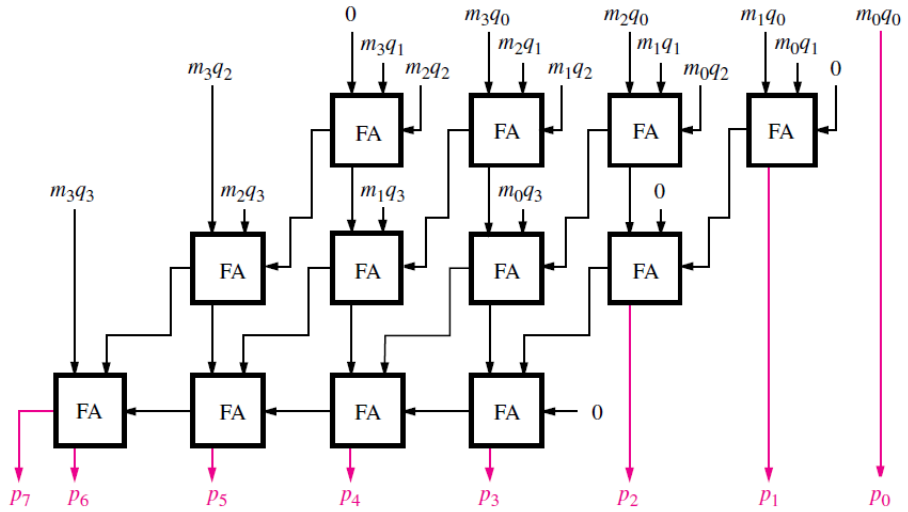

Figure 1: 4-bit Carry Save Multiplier

Multiplexer

Various sized multiplexers will be needed. Be sure to create them with `n` bit wide inputs so that they can properly select among the various operations being performed. It is also acceptable to write behavioral statements with if-else or switch statements in place of the muxes.

ALU

The ALU is made up of all previous components. See Figure 2.

- Inputs:
  - `input1, input2` (16 bits): The two numbers on which to operate.
  - `control` (4 bits): The operation to perform. See Table 1.
- Outputs:
  - `output` (16 bits): The result of the selected operation.

Notes

- All components should use the same value of `n`. `n` should be even, as the multiplier inputs are length `n`/2.
- `Cout`, from the adder, is not used in the ALU and can be safely discarded or ignored.
- `amt`, a shifter input, is the lower ceiling($\log_2$ `n`) bits of `input2`.
- `A` and `B`, the multiplier inputs, are the lower halves of `input1` and `input2`.
- Do not use parameters in the entity of the ALU, however, generics should still be used when instantiating the various components.

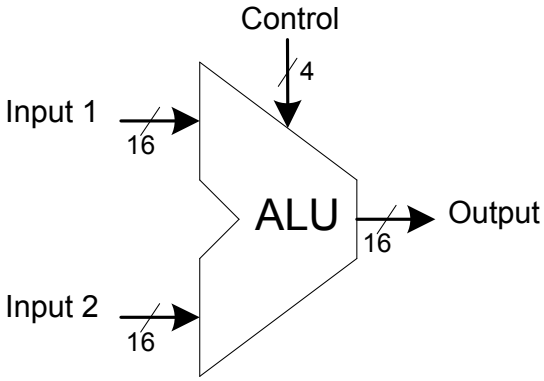

Figure 2: Arithmetic Logic Unit Ports

Test benches

Two test benches are to be created. The first test bench will exhaustively test the multiplier. The second test bench will test the complete ALU.

Multiplier Test bench

Assume that the multiplier inputs are 8 bits each. Create a VHDL file, not a test bench, that writes all possible factors and expected products to a file, in hex. Use the `*` operator to ensure the multiplication is done properly. The file should be 2^{8+8} or 65536 lines long. Each line will contain three values: `A`, `B`, and `Product`. Each of these values shall be separated by a space. Some of these values should be incorrect for use with assert statements in the test bench, meaning that in this file you should have at least 2 lines which are not correct (ex. $2 * 2 \neq 5$). The values can be changed by hand after the file has been initially generated, or in the file generation code itself.

Create a VHDL test bench for the multiplier. It should instantiate the multiplier to have 8-bit inputs and be capable of parsing the hex data created by the previous file. The multiplier should then be tested using the data read in and its output verified using assert statements. The validity of the assert statements will be verified using the incorrect data previously placed in the file. Be sure to account for the delays that will be present in the multiplier.

ALU Test bench

This test bench is not required to be exhaustive, however, it must test all operations. Assert statements are not required as long as the test set is not so large that hand verification is time consuming. Once again, consider the delays that will exist in the ALU following the place and route. Be sure to include tests for the multiplier as well, though they do not need to be extensive or exhaustive.

Lab Procedure

1. Create a new directory and Xilinx ISE project for this exercise.
2. Write a properly commented and properly formatted VHDL program according to the preceding specifications.
3. Run the file generation VHDL program.
4. Test the design using a behavioral simulation. Be sure to simulate with both test benches.
5. Synthesize the design and retest it using a post route simulation. Once again, both test benches must be simulated.
6. Using the generics, resize the design to fit on the board. The switches should be used as inputs. If a larger input size is desired, buttons may be used as well, but this is not required.
7. Prepare the design for hardware, following the procedure laid out in Exercise One.
8. Program the board and verify that it is working.
9. Demonstrate steps 4, 5, and 8 to the instructor, who will sign the grading sheet.
10. Screen shot the appropriate waveforms for the report, making sure that they will be readable.
11. If necessary, modify the code to acquire the timing results. See the section on Lab Reports for details.

12. Submit the source code and report online to the appropriate myCourses dropbox.

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab report format on myCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic.
- A detailed description of how the multiplier was designed, complete with a diagram similar to, **but not the same as**, Figure 1.
- Area used, which is reported via number of occupied slices, FFs used, and LUTs used.
- Timing results (best case achievable). See Exercise Two for further details if required.
- Marked simulation results. This also includes the assert output from the multiplier test bench.