**Project 2 – Combinational Chips**

**Background**

The centerpiece of the computer's architecture is the CPU, or Central Processing Unit, and the centerpiece of the CPU is the ALU, or Arithmetic-Logic Unit. In this project you will gradually build a set of chips, culminating in the construction of the ALU chip of the Hack computer. All the chips built in this project are standard, except for the ALU itself, which differs from one computer architecture to another.

**Objective**

Build all the chips described in Chapter 2 (see list below), leading up to an Arithmetic Logic Unit - the Hack computer's ALU. The only building blocks that you can use are the chips described in chapter 1 and the chips that you will gradually build in this project.

**Chips**

Students may open the chip file from <install directory>/projects/02/

|  |  |  |
| --- | --- | --- |
| Chips Name | File Name | Description |
| **Basic Chips:** |  |  |
| HalfAdder | HalfAdder.hdl | Half Adder |
| FullAdder | FullAdder.hdl | Full Adder |
| Add16 | Add16.hdl | 16-bit Adder |
| Inc16 | Inc16.hdl | 16-bit incrementer |
| **Advanced Chips:** |  |  |
| 4-bit adder | Add4.hdl | 4-bit ripple carry adder |
| 6-bit adder | Add6.hdl | 6-bit ripple carry adder |
| Negation | Negation.hdl | 2’s complement of the input |
| LeftLogicBitShift | LeftLogicBitshift.hdl | 16-bits left bit shifter |
| ALU | ALU-nostat.hdl | Arithmetic Logic Unit (without handling of status outputs) |
| ALU | ALU.hdl | Arithmetic Logic Unit (complete) |

**Proposed Implementation**

There are totally 4 tasks for this project.

1. Build the chips in the order listed above.
2. Use the chips built from task 1 to build a 8-bit negation operator that outputs the 2’s complement of the input.
3. Build a 4-bit ripple carry adder and a 6-bit ripple carry adder. Apply modular construction techniques of building adders as discussed in class.
4. Build a barrel-shifter to perform left logical bit shift by stated amount.
5. Finally, construct an ALU in two incrementally advanced stages.

Below we share with you some additional background on the logic of a shifter and the ALU.

**BIT SHIFTER:**

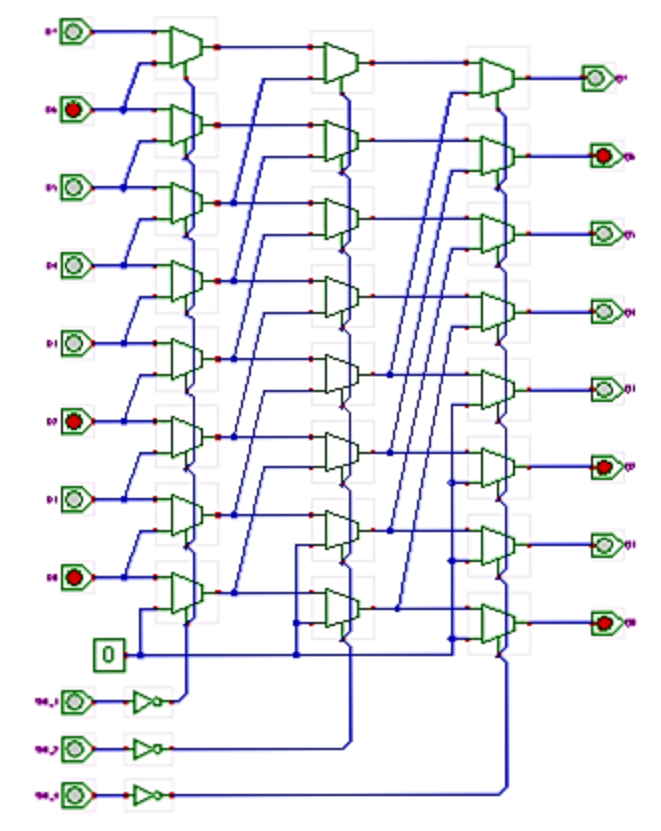


Figure 1 – schematic of a 4-bit barrel shifter. X represents the input (original bits position) and Y represents the output (after bit shifting).

Figure 1 shows a 8-bit barrel-shifter as an example. The circuit allows shifting the input data word left, where the amount of shifting is selected via the control inputs. Several microprocessors include barrel-shifters as part of their ALUs to provide fast shift (and rotate) operations.

The circuit shown in the applet consists of three stages of 2:1 multiplexers, with one multiplexer per bit of the input data (here 8). When all multiplexer select inputs are inactive (low), the input data passes straight through the cascade of the multiplexers and the output data (Q7..Q0) is equal to the input data (D7..D0). When enabled via the SHL\_1 input, the first stage of multiplexers performs a shift-left by one bit operation, due the their interconnection to the next-lower input. A low input value (0) is used for the least significant bit, so that the shifter output becomes (Q6..Q1 0).

Similarly, the second stage of multiplexers performs a shift-left by two bits when enabled via the SHL\_2 control signal. Note that the corresponding multiplexer inputs are connected to their second next-lower input, and two zeroes are required for the lowest bits. Finally, the third stage of multiplexers performs a shift-left by four bits, with four zero bits filled into the lowest bits.

Due to the cascade of three stages, all three shift operations (by one bit, by two bits, and by four bits) can be activated independently from each other. For example, when both SHL\_1 and SHL\_4 are activated, the shifter performs a shift-left by five bits. The generalization to higher word-width (e.g. 16 bits) should be obvious, as should the construction of shift-right or rotate operator. **In your implementation of the barrel shifter, you are operating on a 16-bit data but maximum shift in bits is limited to 8 bits.** In your LeftLogicBitshift.hdl, there are two inputs, x and y. X is the input bits and Y is the number of bits to shift in binary form. For instance, X equals 1100 0000 1010 0101, and Y equals to 0000 0000 0000 0010. It means X needs to be shifted left by 2 bits, which the answer is 0000 0010 1001 0100. The most significant bits are dropped and the least significant bits are replaced by 0.

**The HACK ALU**

The Hack ALU produces two kinds of outputs: a "main" 16-bit output resulting from operating on the two 16-bit inputs, and two 1-bit "status outputs" named 'zr' and 'ng'. We recommend building this functionality in two stages. In stage one, implement an ALU that computes and outputs the 16-bit output only, ignoring the 'zr' and 'ng' status outputs. Once you get this implementation right (that is, once your ALU.hdl code passes the ALU-nostat test), extend your code to handle the two status outputs as well. This way, any problems detected by ALU.tst can be attributed to the incremental code that you've added in stage two. We thank Mark Armbrust for proposing this staged implementation plan, and for supplying the test files to support it.

**Contract**

When loaded into the supplied Hardware Simulator, your chip design (modified .hdl program), tested on the supplied .tst script, should produce the outputs listed in the supplied .cmp file. If that is not the case, the simulator will let you know.

**Resources**

The relevant reading for this project is Chapter 2 and Appendix A. Specifically, all the chips described in Chapter 2 should be implemented in the Hardware Description Language (HDL) specified in Appendix A.

For each chip, we supply a skeletal .hdl file with a missing implementation part. In addition, for each chip we supply a .tst script that instructs the hardware simulator how to test it, and a .cmp ("compare file") containing the correct output that this test should generate. Your job is to complete and test the supplied skeletal .hdl files.

The resources that you need for this project are the supplied Hardware Simulator and the files listed above. If you've downloaded the Nand2Tstris Software Suite, these files are stored in your projects/02 directory.

**Tips**

Use built-in chips: Your HDL programs will most likely include chip parts that you've built in project 1. As a rule, though, we recommend using the built-in versions of these chips instead. The use of built-in chips ensures correct, efficient, and predictable simulation. There is a simple way to accomplish this convention: make sure that your project directory includes only the .hdl files of the chips developed in the current project.

**What to turn-in**

Turn in a zip file “<your name>\_P2” containing HDL files for all 9 chips implemented in the exercise shown in the table on Page1.