CSCE-312 | Fall 2019

Project 2

Combinational Chips

Due Date: Submit on eCampus by **Wed, Oct 2nd, 11:45 PM**

Grading

(A) Project Demo [70%]:

You will be graded for correctness of the chips (hdl) you have designed and coded. So, make sure to test and verify your codes before finally submitting on eCampus.

***Rubric**:* Each circuit needs to pass all its test cases to get the points, else you will receive 0 on that circuit.

(B) Lab Quiz [30%]: Fri, Oct 4th Labs

The questions can involve drawing circuit diagram of randomly selected chips. Should not be difficult for you if you have understood the core inner workings of your project.

Deliverables & Submission

You need to turn in only the HDL files for all the chips implemented. Put your full name in the introductory comment present in each HDL code. Use relevant code comments and indentation in your code. Zip all the required HDL files and the signed cover sheet into a compressed file ***FirstName-LastName-UIN.zip*** Submit this zip file on eCampus.

Late Submission Policy: Refer to the Syllabus

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

The objective of this project is to build all the chips discussed in Chapter 2 and class, 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 Project 1 and the chips that you will gradually build in this project.**

Chips

You may open any given chip file from ***P2Codes.zip***

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

Proposed Implementation

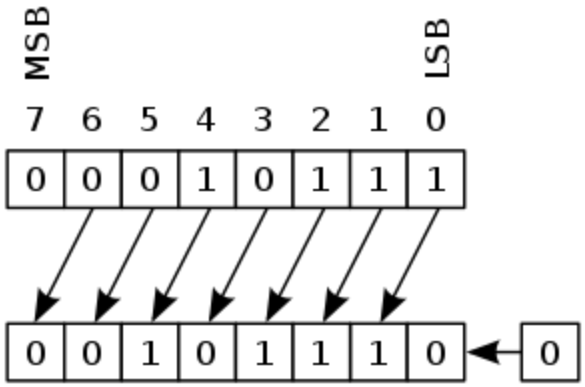
1. Build the basic chips in the order listed above.
2. Build a 4-bit ripple carry adder and a 6-bit ripple carry adder. Apply modular construction techniques of building adders as discussed in class.
3. Build a 16-bit negation operator that outputs the 2's complement of the input.
4. Build a barrel-shifter to perform left logical bit shift by the stated amount.
5. Finally, construct an ALU in two incrementally advanced stages.

For Basic Chips: Refer to class lecture material and the video below

https://www.youtube.com/watch?v=xtcJMxlEFek&index=18&list=PLrDd_kMiAuNmSb-CKWQqq9oBFN_KNMTa

Below we share with you some additional background on the logic of a **Left Logic Bit Shift and the ALU.**

LEFT LOGIC BIT SHIFTER:



For an operand ***N*,** whose binary equivalent is **x[8]**, the 8-bit representation is indexed as 7..0 from left to right.

Left Logic Shifting ***N*** by ***one bit***, represented as *N<<1,* moves each bit of **x** one place on its corresponding left. For instance, in the example above, let **out[8]** be the binary representation of the resulting number after applying Left Logic Bit Shifter on the input **x[8]**. Then, **x[6]** becomes **out[7]**, **x[5]** becomes **out[6]**, and so on. The rightmost bit (LSB), i.e., **out[0]** in the resulting number is "always" simply filled by 0.

Mathematically, what does left logic bit shifting do to a number?
To answer this, let's take the above example where the *x = 00010111* is left shifted by *b=1* bit. Let us start by writing its decimal equivalent before and after shifting. Before shifting, the number is *N*=23. After shifting by one bit, the resulting *out = 00101110*, in decimal representation becomes

$$(1 \times 2^5) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) = 32 + 8 + 4 + 2 = 46$$

46 is nothing but 23×2. This essentially means that left logic shift by 1 bit doubles the number.

Similarly, left logic shift by 2 bits, $N << 2$, results in $N \times 2^2 = N \times 4$, and ,
left logic shift by 3 bits, $N << 3$, results in $N \times 2^3 = N \times 8$

Therefore, **for any given operand/ decimal number *N*, shifting by *b* bits to the left results in a new number,** $N << b = N \times 2^b$

Now, with this background in mind, let's jump into the concept of a barrel shifter. Barrel shifter is simply a digital circuit that implements the left logic bit shifting. Following the input pin convention according to the select pin state (0 or 1) for MUX as shown below, the diagram of the barrel shifter is as follows.

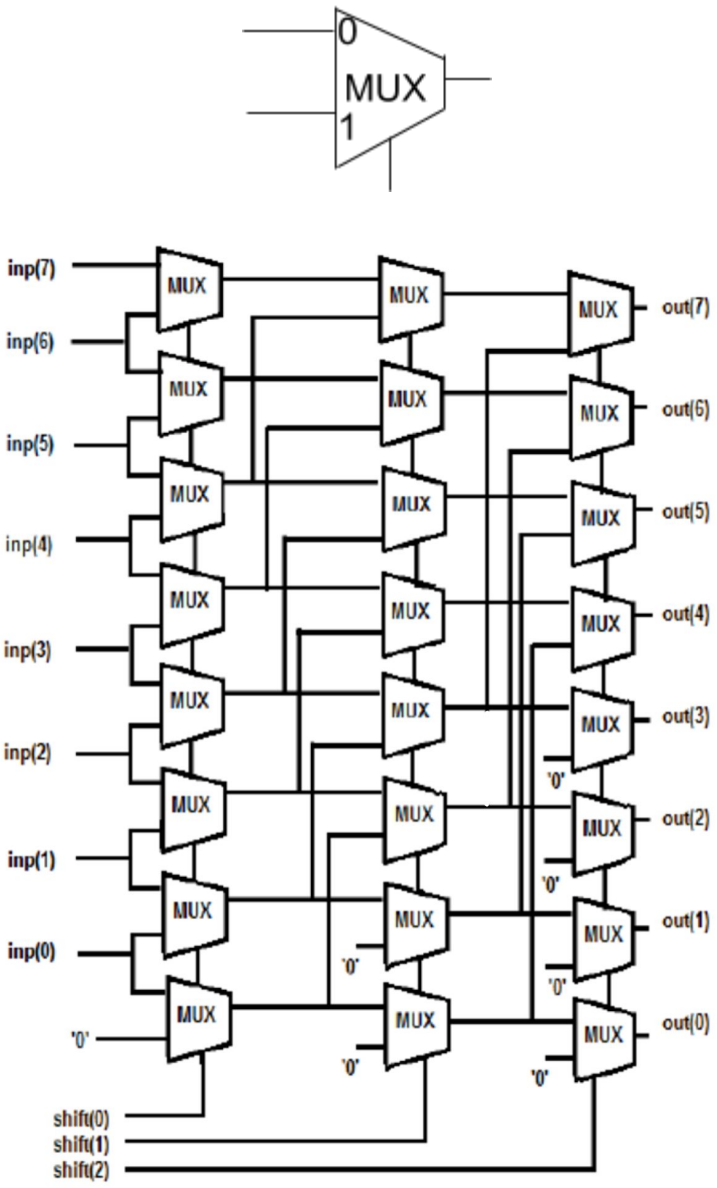


Figure 1 – schematic of a 8-bit barrel shifter. "inp" represents the input (original bits position) and "out" represents the output (after bit shifting).

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

The logic circuit shown in Figure 1 consists of three stages of 2-way MUX, with one multiplexer per bit of the input data ***inp[8]*** which is binary representation for ***N***.

Given, shift input, ***shift(2)..shift(0)*** , which is binary representation for ***b***

1. When all multiplexer select inputs are inactive (low), i.e., *shift=000*, the input data passes straight through the cascade of the multiplexers unchanged (b=0) and the output data ***out(7)..out(0) = inp(7)..inp(0)***

2. On enabling just ***shift(0)***=*1,* i.e., *shift=001*, the first stage of multiplexers performs a shift-left by one bit, i.e. *N<<1* operation (b=1), due to 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 ***out(7)..out(0) = inp(6)..inp(0) 0*** and in this process dropping ***inp(7)***.

3. Similarly, on enabling just ***shift(1)***=*1*, i.e., *shift=010*, the second stage of multiplexers performs a shift-left by two bits, i.e. *N<<2* (b=2). Note that the corresponding multiplexer inputs are connected to their second next-lower input, and two zeros are required for the lowest two bits. ***out(7)..out(0) = inp(5)..inp(0) 0 0***

4. Finally, on enabling just ***shift(2)***=*1*, i.e., *shift=100*, the third stage of multiplexers performs a shift-left by four bits, i.e. *N<<4* (b=4), with four zero bits filled into the lowest bits. ***out(7)..out(0) = inp(3)..inp(0) 0 0 0 0***

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. Hence, one can shift a number left , ***N<< b*** by enabling appropriate *shift(2)..shift(0)*

For example, when both ***shift(0)*** and ***shift(2)*** are activated , i.e. overall *shift* = 101, the barrel shifter performs shift-left by five bits (b=5), i.e. ***N<<5***. This is realized through first stage where ***shift(0)***=1 results in ***N<<1***, and then the resulting shifted number, ***N<<1***, goes through second stage of MUX unchanged as it was, since ***shift(1)***=0. Finally, it exits the barrel shifter circuit through the third stage of MUX where ***shift(2)***=1 results in shift-left by four more bits, i.e., (***N<<1)<<4***). So, overall we get the effect of ***N<<5***.

The generalization to higher word-width (e.g. 16 bits) should be obvious

In Project 2 implementation of the barrel shifter (left-logic bit shift), you are operating on a 16-bit data with maximum left-shift allowed is by *8 bit* positions i.e. N<<b, where, b={0,1,2,..,8}

In your *LeftLogicBitshift.hdl*, there are two inputs, *x[16]* and *y[16],* and an output, *out[16]*.

- ***x[16]*** (*inp[8]* used in above illustration in Figure 1) is the binary representation for number *N* which needs to be shifted left by *b* bits, *N<<b*.
- ***y[16]*** (*shift[3]* used in above illustration in Figure 1) is the binary representation of *b* by which data needs to be shifted left, resulting in
- ***out[16]*** (*out[8]* used in above illustration in Figure 1) is the binary representation of *N<<b*

For instance, $x = 0000\ 0000\ 1010\ 0101$, and $y = 0000\ 0000\ 0000\ 0010$ implies that $N = 165$ needs to be shifted left by $b = 2$ *bits*, which should result in $165<<2 = 165 \times 2^2 = 165 \times 4 = 660$. We can verify this by left-shifting x by *2 bits* and observing $out = 0000\ 0010\ 1001\ 0100$. (The blue *out* bits represent what part of original input x is still retained and shifted left, while red bits represent the newly appended zeros). Here, the higher significant bits, *x(7)x(6)* are dropped as bits are shifted left, and the lowest two bits are assigned value zero, i.e., ***out(1)out(0) = 00***.

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 1, 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 2. We thank Mark Armbrust for proposing this staged implementation plan, and for supplying the test files to support it.

Watch ALU Video:

https://www.youtube.com/watch?v=PEs855FNCOw&list=PLrDd_kMiAuNmSb-CKWQqq9oBFN_KNMTaI&index=17

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, class notes, and Appendix A.

Specifically, all the chips must 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.

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.**