**Project 3: Sequential Chips**

**Background**

The computer's main memory, also called Random Access Memory, or RAM, is an addressable sequence of n-bit registers, each designed to hold an n-bit value. In this project you will gradually build a RAM unit. This involves two main issues: (i) how to use gate logic to store bits persistently, over time, and (ii) how to use gate logic to locate ("address") the memory register on which we wish to operate.

**Objective**

Build all the chips described in the list below, leading up to a Random Access Memory (RAM) unit. **The only building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and chips described in previous chapters.**

**Chips**

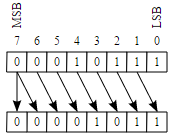
|  |  |  |
| --- | --- | --- |
| Chips Name: | Description | File Name |
| Bit | 1-bit register | Bit.hdl |
| Register | 16-bit register | Register.hdl |
| RAM8 | 8 16-bit register memory | RAM8.hdl |
| RAM64 | 64 16-bit register memory | RAM64.hdl |
| RAM512 | 512 16-bit register memory | RAM512.hdl |
| PC | 16-bit program counter | PC.hdl |
| CaesarCipher | 4-bit counter using D flip flop | CaesarCipher.hdl |
| RightArithmeticBitshift | Bit shifter using D flip flop | RightArithmeticBitshift.hdl |

**Caesar Cipher**

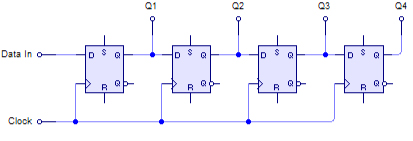
Caesar cipher is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a left shift of 3, D would be replaced by A, E would become B, and so on. The method is named after Julius Caesar, who used it in his private correspondence.

In project 3, the output of Caesar Cipher is equals to the input + value generated from counter, where counter starts from 0000, and increment by 1 every clock cycle.

**Right Arithmetic Bit Shifter**

The two basic types are the arithmetic left shift and the arithmetic right shift. For binary numbers it is a bitwise operation that shifts all of the bits of its operand; every bit in the operand is simply moved a given number of bit positions, and the vacant bit-positions are filled in. Instead of being filled with all 0s, as in logical shift, when shifting to the right, the leftmost bit (usually the sign bit in signed integer representations) is replicated to fill in all the vacant positions (this is a kind of sign extension).

For project 3, we are using the SIPO(Serial In Parallel Out) implementation, which reads in one-bit per one clock cycle and output when desired cycles are reached. The basic structure is similar to the diagram below,



For each cycle, output of a D flip-flop is loaded into its right neighbor (except the rightmost D flip-flop), thus the input bits are shifted right for one bit.

In order to implement arithmetic shift (filling in new bits with MSB bit), you will be given a load signal input, which is set to 1 in our test file when filling in input bits (normally takes 4 cycles). When load signal is set to 0 after we finish reading in, the leftmost D flip-flop should keep outputting its value to its right neighbor.

You will also be given a reset signal, when reset is set to 1, load 0 into each of the D flip flops.

Do not worry about counters, the test file will read the output at the right time and reset the circuit for next test case.

**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 3 and Appendix A.** Specifically, all the chips described in Chapter 3 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. Download your hdl files from ecampus and replace these files to those stored in your projects/03 directory.

**Tips**

The Data Flip-Flop (DFF) gate is considered primitive and thus there is no need to build it: when the simulator encounters a DFF chip part in an HDL program, it automatically invokes the built-in tools/builtInChips/DFF.hdl implementation.

Tools

All the chips mentioned projects 0-5 can be implemented and tested using the supplied Hardware Simulator. Here is a screen shot of testing a built-in RAM8.hdl chip implementation on the Hardware Simulator:

