Project 3: Sequential Chips

Deadline: Section 501 (Oct 3), Section 502 (Oct 4), Sections 504 & 505 (Oct 6) on eCampus by 9am CST

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. In addition, you will build functions that are constructed with combinational and sequential logic design elements.

Objective

Build all the chips described in the list below. **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 (use DFF)	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
Aggie Cipher	4-bit counter using D flip flop	AggieCipher.hdl
RightArithmeticBitshift	Bit shifter using D flip flop	RightArithmeticBitshift.hdl
Fibonacci	Fibonacci Sequence generator	Fibonacci.hdl

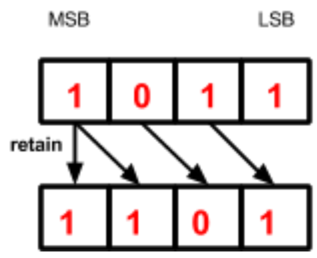
Aggie Cipher

Design a simple cipher logic which generates a code which is equal to a user-provided 4-bit *input + the value generated from a counter*, where counter value starts from 0000, and increments by 1 every clock cycle. The counter wraps to 0000 when it reaches a count of 15. You may use the program counter (PC) designed in prior exercise to implement the Aggie Cipher logic.

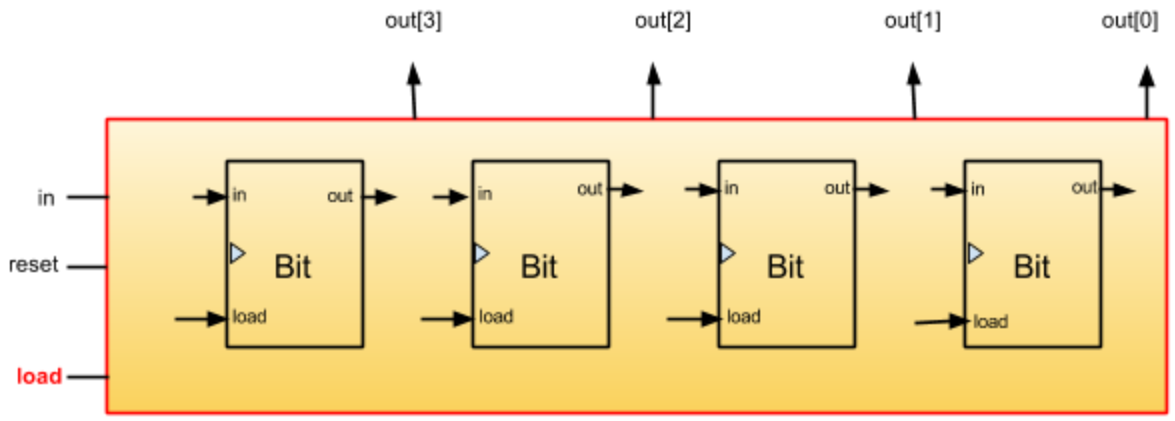
out=in+counter*, where counter=<0,1,2,3,4,5,6,....,15,0,1,2,...>*

Right Arithmetic Bit Shifter

An arithmetic shift requires preservation of the sign bit (MSB). Note the example below where a single arithmetic right shift operation results in the sign bit (MSB) getting **retained** at MSB, and all four bits (including MSB) will respectively shift to the right resulting in the LSB getting dropped.



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



We implement this logic in two steps:

(a) Input "in" is loaded serially into the 4-bit register while the external **load** input is a 1.
(b) Once all the bits are loaded, then perform arithmetic right shift when **load** input is a 0**.**

In order to implement arithmetic shift (filling in new bits with MSB bit), you will be given

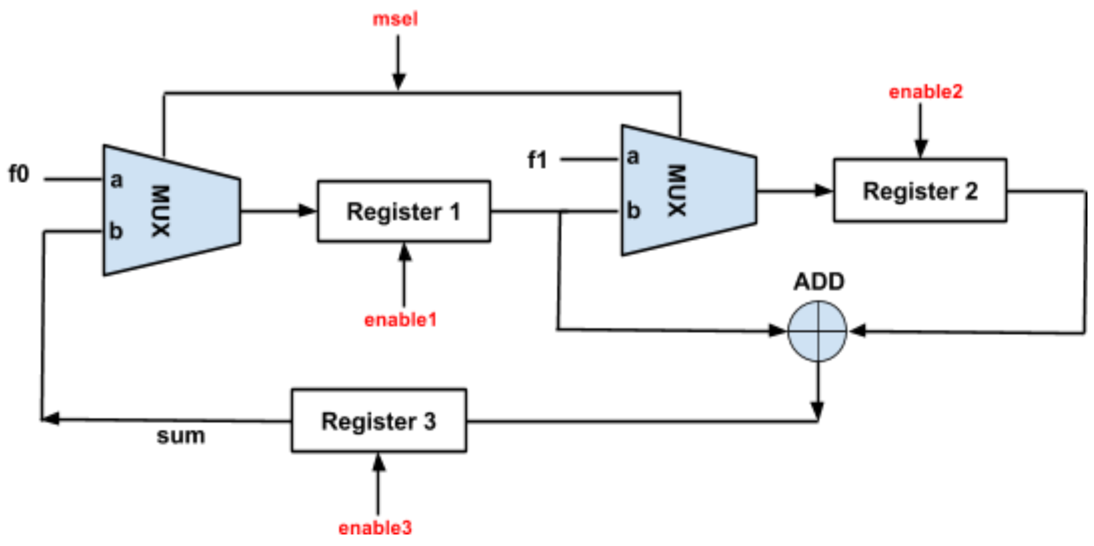
1. A ***load*** signal input. In our test file,
    a. when ***load*=**1, we fill in *in* bits; 1 bit right shift with each clock cycle. In other words, each **Bit** passes its stored value to its next one.
    b. when ***load*** = 0 after we finish reading in, 1 bit right shift occurs. However, **out[3] (MSB) should retain its value while propagating it to the bit register to its right**.

    NOTE: ***load*** input of RightArithmeticBitShifter chip works differently (based on above heuristic) from the usual load pin of Bit chip
2. A *reset* signal. When *reset=1*, fill 0 into each of the four **Bit**s. Otherwise, *reset=0*.

Complete the above circuit of the RightArithmeticBitShifter chip by proper use of logic to implement the above specification. See the RightArithmeticBitshift.cmp and RightArithmeticBitshift.tst files to understand the logic.

Do not worry about clock connection (small triangle in Bit) as the test file will read the output at the right time and reset the circuit for next test case.

Fibonacci Sequence generator:
The general Fibonacci sequence is a sequence that starts with *f0=0* and *f1=1* . The next number in the sequence is the *sum* of previous two numbers. So the Fibonacci number sequence generated in our circuit will be:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…
Here is the circuit you may use. **Please make an effort to understand the working of this circuit and explain it as comments in your HDL file.**



To use this circuit, you have to control these signals, namely, *enable1, enable2, enable3* and *msel*.

- *msel=0* will select the starting values *f0* and *f1* of the Fibonacci Sequence
- *msel=1* will keep running the Fibonacci sequence with *sum(t+1) ← sum(t) + sum(t-1)* for clock cycle *t*
- *enable1=1 or enable2=1* or *enable3=1* activate respective registers by loading the corresponding input values to corresponding register outputs
- *enable1=0 or enable2=0* or *enable3=0* retain the register outputs from the previous cycle

The test file Fibonacci.tst assigns the values to these control signals.
See how output in the Fibonacci.out file changes while changing those signals.

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.

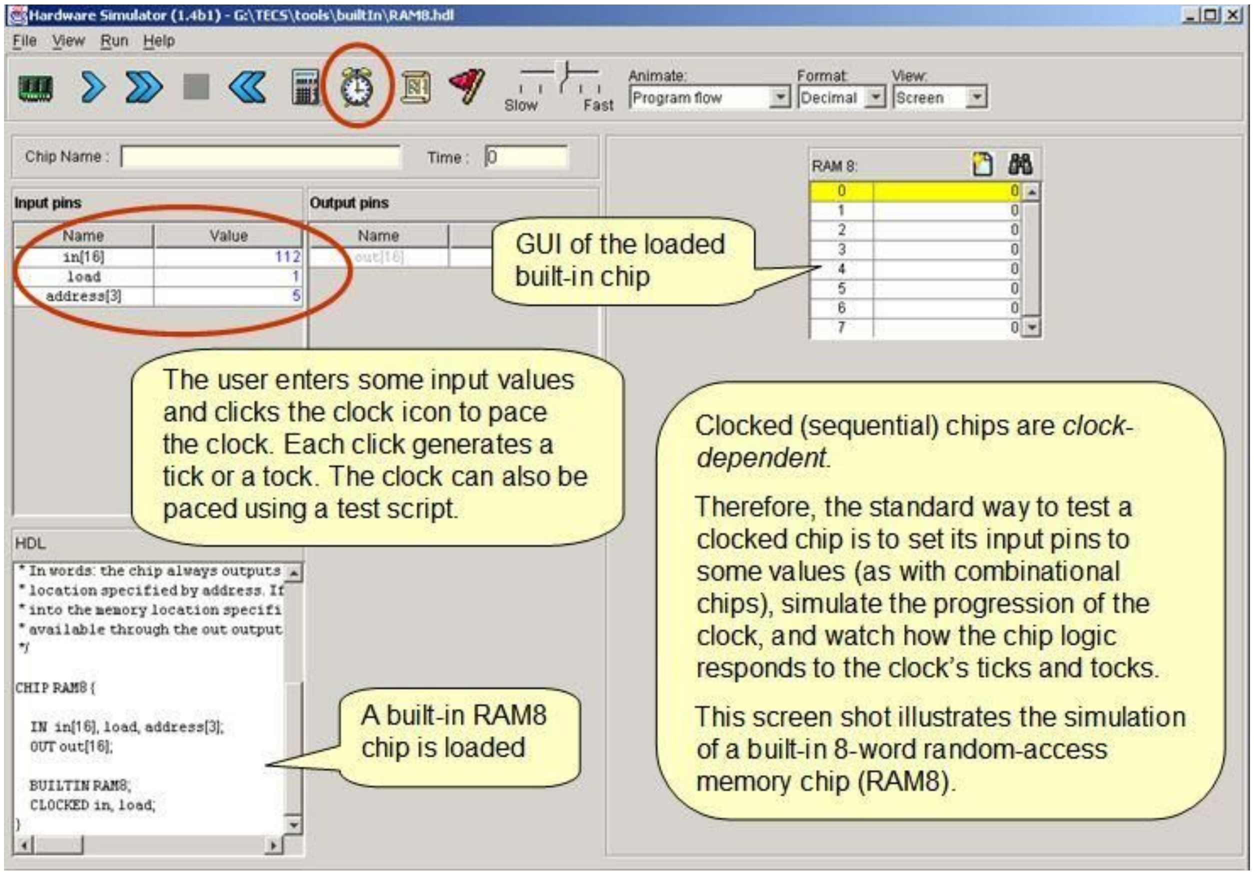
The resources that you need for this project are the supplied Hardware Simulator and the files listed above. Download your hdl files from ecampus and replace these files to those stored in your projects/03 directory.

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 screenshot of testing a built-in RAM8.hdl chip implementation on the Hardware Simulator:



Hardware Simulator (1.4b1) - G:\TECS\tools\builtIn\RAM8.hdl

Animate: Program flow
Format: Decimal
View: Screen

Chip Name :
Time : 0

Input pins

Name	Value
in[16]	112
load	1
address[3]	5

Output pins

Name	Value
out[16]	

RAM 8:

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0

```
* In words: the chip always outputs
* location specified by address. If
* into the memory location specifi
* available through the out output
*/

CHIP RAM8 {

  IN in[16], load, address[3];
  OUT out[16];

  BUILTIN RAM8;
  CLOCKED in, load;
}
```

GUI of the loaded built-in chip

The user enters some input values and clicks the clock icon to pace the clock. Each click generates a tick or a tock. The clock can also be paced using a test script.

Clocked (sequential) chips are *clock-dependent*.

Therefore, the standard way to test a clocked chip is to set its input pins to some values (as with combinational chips), simulate the progression of the clock, and watch how the chip logic responds to the clock's ticks and tocks.

This screen shot illustrates the simulation of a built-in 8-word random-access memory chip (RAM8).

A built-in RAM8 chip is loaded

Rubric (total 50 points):
Bit: 4
Register: 4
RAM8: 5
RAM64: 5
RAM512: 6
PC: 6
AggieCipher: 6
RightArithmeticBitshift: 8
Fibonacci: 6

What to turn-in

Similar to what you submitted for Project one, turn in a zip file in format "FirstName-LastName-UIN" containing HDL files for all 9 chips implemented in the exercise shown in the table on Page1.