CS 252 (Spring 17): Computer Organization

Project #7

Carry Lookahead and Multiplier

due at 5pm, Fri 24 Mar 2017

1 Purpose

It's back to C for this project. In this project, we'll be implementing both a Carry Lookahead Adder, and also Multiplier.

1.1 Required Filenames to Turn in

Name your C file `proj07.c` .

2 Tasks

Like with Project 1, I will provide a header file, which defines the structs and functions that we will use. You will implement these functions in your C file.

2.1 Carry Lookahead Adder

For the Carry Lookahead Adder, I won't provide a struct that you need to modify, and you won't provide an `execute_*` function. Instead, I will call a series of functions that you must implement, which calculate the various values used in a CLA. For each function, I will give you some of the information which you would know in that part of the CLA (some of the bits that we've calculated), and you will write a function which calcuates **one bit** of information. (Every function must return either 0 or 1.)

You will need to implement bit shifting and masking in each function (to read the proper bits from the proper fields). I'm giving you (almost) complete freedom in how you implement your function - provided that you actually perform your calculations using **logic, not addition.** That is, you're allowed to use bit shifting, masking, logical operators, etc.; you can even use `if()` statements. But the one thing you can't do is to simply add the numbers together to get the proper answer.

You must implement the following functions. For each one, you can assume that we are implementing a 16-bit adder.

- `int CLA_calcGenerate(int bitNum, int a,int b)`

  Returns the generate bit for column `bitNum`, which is in the range 0 through 15 (inclusive). `a,b` are the two input numbers (which you can assume will only have the bottom 16 bits set).

- `int CLA_calcPropagate(int bitNum, int a,int b)`

  Returns the propagate bits. See `CLA_calcGenerate()` above.

- `int CLA_calcSuperGenerate(int nibbleNum, int generate,int propagate)`

  Returns the super-generate bit for **nibble** `nibbleNum`, which is in the range 0 through 3 (inclusive). `generate,propagate` are the generate and propagate bits for the various input bits, encoded as 16-bit numbers.

  You may use multiplication in this function, to calculate how much to shift the bits. However, don't use multiplication or addition for anything else.

- `int CLA_calcSuperPropagate(int nibbleNum, int generate,int propagate)`

  Returns the super-propagate bits. See `CLA_calcSuperGenerate()` above.

- `int CLA_calcNibbleCarryIn(int nibbleNum, int superGenerate, int superPropagate, int carryIn)`

  Returns the carry-in bit for nibble `nibbleNum`. `superGenerate,superPropagate` are the super-generate and super-propagate bits, expressed as 4-bit numbers. `carryIn` is the carry-in bit to the **entire adder** (single bit); it will either be 0 or 1.

- `int CLA_calcBitCarryIn(int bitNum, int nibbleCarryIn, int generate, int propagate)`

  Returns the carry-in bit for a **single bit**. `nibbleCarryIn` is the carry-in bit for the current nibble; it will either be 0 or 1. `generate,propagate` are the generate and propagate bits for the various input bits, encoded as 16-bit numbers.

  You are allowed to use division and modulo in this function to figure out which nibble the bit is in, and also its position within the nibble. (If you're a little tricker, you could even accomplish this with masking and bit shifting!)

2.2 Multiplier - One Step

The One Step Multiplier will work a lot like the other hardware simulation tasks you've done in Projects 1,3,5. However, because it represents something which has registers - and these registers change over time - we don't want to use a single struct for both input and output.

Instead, this execute function has **two** parameters, both of the same type. The first is your input; it is the state of the registers at the **beginning** of the clock cycle. The second is your output; it is the state of the registers at the **end** of the clock cycle. You **must not** modify the first struct; you **must** set all of the fields in the second.

You will implement the following function:

```
void execute_multOneStep(MultOneStepData *before, MultOneStepData *after)
```

The fields in the struct are `multiplicand, multiplier, result`. Your `execute` function should model one step of the multiplier; shift both of the fields by one, and add to update the result.

Since this is a multiplier it is **OK to add**. (We'll assume that when you built the real hardware, you had a pre-built adder ready to use.) However, as you might expect, it is **against the rules to multiply or divide.**

NOTE: With both multipliers, assume all values are unsigned.

2.3 Multiplier - MultiCycle

You will implement a multi-cycle multiplier. This multiplier must use the One Step Multiplier, over and over, to perform its work. (Why is it suddenly OK to call `execute_*` multiple times? Because each call represents a different clock cycle!)

This has an `execute` function that's a lot more like the ones you've seen in other Projects; it has a single parameter, and the struct contains both input and output fields.

The fields `a,b` are the inputs; each is a 32-bit value. The field `out` is the output field, which is 64 bits wide. Your function should set up a pair (or more) of `MultOneStepData` structs, and call the one-step function up to 32 times. (If you simply loop 32 times, that's OK. But if you find certain situations where you can terminate the loop early because there's no work left to be done, that's a cool bonus.)

NOTE: With both multipliers, assume all values are unsigned.

3 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

3.1 Testcases

You can find a set of testcases for this project at
`http://www.cs.arizona.edu/classes/cs252/spring17/projects/proj07/`
You can also find them on Lectura at
`/cs/www/classes/cs252/spring17/projects/proj07/`
(this is the backing directory for the website).

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

3.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj07`. Place this script, all of the testcase files (including their `.out` files if assembly language), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

3.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

4 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.