CS 252 (Spring 17): Computer Organization

Project #1

Basics of Binary

due at 5pm, Fri 27 Jan 2017

1 Purpose

This project will use both C and Java[1]. You will practice implementing basic binary operations - but you won't be using the regular operators given by C and Java; instead, you will be implementing them yourself, by hand.

The point of this is to give you practical experience with how the CPU works internally; while you'll be writing code in C or Java, the logic you'll be using is essentially the same as the digital logic implemented in hardware.

You will be implementing some of these algorithms twice: once in C, and once in Java. I'm doing this for several reasons. First, I want you to refresh your knowledge of of both languages, as we'll be using them in later projects. Second, seeing this in two different ways (values as integers, or values as arrays of booleans) should help you understand the link between the "values" we see as programmers, and the electrical wires that carry the bits. Third, repetition helps retention.

2 Tasks

You will be implementing binary (integer) addition twice, once in Java, and once in C. (It's probably easiest to do it in Java first.) You will also implement bitwise AND, OR, and NOT in Java. To practice composition (building complex things from simpler ones), you will implement 2's complement in Java, using the adder and NOT that you wrote.

In Java, I will provide a base class, which has three arrays of `boolean`: two inputs and one output. The base class will ensure that all of the arrays are non-`null` and have exactly 32 elements. You will override some methods in a child class, and fill in all of the appropriate outputs for each method (not all methods will fill in all outputs).

In C, I will provide a header, which gives the declaration of a struct, which is very similar to the Java class - except that instead of arrays of `boolean`, it will have `int`s (which are 32-bit signed integers). You will provide the implementation for one function, which takes a pointer to that struct as a parameter.

[1] We won't be doing anything complex in either language. If you feel a little rusty, feel free to ask questions in class, or come by Office Hours.

2.1 You Can't Do Addition!

Neither version of this project may use the ordinary addition operator - in fact, our grading script will scan your program for that operator, and will cut your grade in half if it finds any! (`++` is OK for your loops - but `+=` is not.)

Instead, you must implement your code entirely with logical operators. For the Java program, the key operators will be:

- `&&` (boolean AND)
- `||` (boolean OR)
- `!=` (boolean XOR)

For the C program, you will need to extract bits from the inputs, and then assemble them back together for the inputs. In addition to the operators above, you will find the following operators useful (later in this spec, I have written up some reminders about how to use them):

- `>>` (right shift)
- `<<` (left shift)
- `&` (bitwise AND - useful for extracting bits out)
- `|` (bitwise OR - useful for adding bits in)

2.2 Base Code

Download the files `Proj01_Base.java` and `proj01.h` from the project directory `http://www.cs.arizona.edu/classes/cs252/spring17/projects/proj01/`.

DO NOT MODIFY THESE FILES. If you turn in a modified version of these files, our grading script will ignore them, and use the standard versions instead. (This will probably make your program fail to compile.)

2.3 Required Filenames to Turn in

Name your Java file `Proj01.java`. Name your C file `proj01.c`.

3 Required Functions

In the base Java code, `a[],b[],out[]` are all arrays of `boolean`; the base class will ensure that all are non-`null`, with exactly 32 elements. In the C code, the matching fields `a,b,out` are all `int`s (32-bit signed integers).

Your Java class must be named `Proj01`, and it must be a child of `Proj01_Base`. It must implement the following member functions:

- `void execute_add()`

  This reads the input arrays `a[],b[]`.

  Treat each input array as a 32-bit signed integer, where `[0]` is the least significant bit, and `[31]` is the most significant bit.

  Calculate the sum (using only logical operations, not Java's addition operator!) and store the result into `out[]`. Set the flags `carryOut,overflow` to `true` or `false` to indicate whether or not those two conditions occurred.

  REMEMBER: Carry-out is a condition of the adder, and does not necessarily mean that an error occurred. Overflow is an error.

- `void execute_AND(), void execute_OR()`

  These methods read the input arrays `a[],b[]`.

  Perform bitwise AND or OR of the input, and store the results in `out[]`.

  Do not change `carryOut,overflow`.

- `void execute_NOT()`

  This method reads the input array `a[]`. It ignores `b[]`.

  Perform bitwise NOT of the input, and store the results in `out[]`.

  Do not change `carryOut,overflow`.

- `void execute_2sComplement()`

  This method reads the input array `a[]`. It ignores `b[]`.

  Calculate the 2's complement of the input, and store the result in `out[]`. You are strongly encouraged (though we won't check this) to implement this by creating two sub-objects of the same class; use one to perform NOT, and another to perform addition.

  Do not change `carryOut,overflow`. (Although addition is part of calculating the 2's complement, and carryOut and overflow can both occur, you should not set them in this method.)

Your C code must implement the following functions:

- `void execute_add(Proj01Data *obj)`

  This reads the inputs `int a,b`.

  Add the two inputs, but do so without using the C addition operator. Instead, use bitwise shifting and masking to extract each bit from each input, then use logical operations to determine the proper value for each output bit. Use logical operations to join the output bits together into a single integer.

  Treat all 32 bits in the same way (no special handling for the sign bit).

  Finally, set the flags `carryOut,overflow` to 1 or 0 to indicate whether or not those two conditions occurred.

4 Bit Shifting and Extraction

I know that some of you aren't terribly familiar with bit shifting and masking. So here's a quick summary of the important operators. (All of these operators exist in both C and Java.):

- `val >> n`

  Shifts the bit pattern `val` to the right (that is, toward the less-significant bits) by `n` bits. Any bits which "fall off the bottom" are simply lost. [2]

  EXAMPLE:

  Suppose that

  $$\texttt{val} == 0101\_1011_2$$

  Now shift it right by 2 bits.

  $$\texttt{val>>2} == 0001\_0110_2$$

- `val << n`

  Shifts the bit pattern `val` to the left. Any bits which fall off the top are lost; zeroes are added at the bottom.

- `val & mask`

  Bitwise AND. Usually used for "masking" - that is, for zeroing out all bits in a number **except** the ones of interest. For instance, the following operation will zero out every bit in `val`, except for bits 2-4 (the 4's through 16's columns):

  ```
  val & 0x1c
  ```

  When used in conjuction with right shift, it allows you to read any individual bit:

  ```
  (val >> n) & 0x1
  ```

- `val | bitsToSet`

  Bitwise OR. Usually used for setting bits in a number. For instance, the following operation will set bits 2-4 to 1 (no matter what they used to be):

  ```
  val | 0x1c
  ```

  When used in conjuction with left shift, it allows you to set any individual bit:

  ```
  val | (0x1 << n)
  ```

[2] What happens at the top? Do we sign extend the value, or just add zeroes? Java has two shift operators, `>>` and `>>>` to solve that problem. In C, there is only one shift operator, and different machines work in different ways. But in our problem, we'll be using `&` to get a single bit after the shift, so it doesn't matter!

5 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!

5.1 Testcases

You can find a set of testcases for this project at
`http://www.cs.arizona.edu/classes/cs252/spring17/projects/proj01/`

In this project, we have C testcases (named `test_*.c`) and Java testcases (named `Test_*.java`). Download all of them, since the grading script will use both sets.

For each of these testcases, we have also provided a `.out` file, which gives **exactly** what your code must print (to `stdout`) when this testcase runs.

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 testcaes of your own, in order to better test your code.** You are also encouraged to share your testcases on Piazza!

5.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj01`. Place this script, all of the testcase files, 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!)

Below, I've provided some example output from the script:

5.2.1 Example: Passed Testcase

When a testcase passes, it looks something like this:

```
*****************************
* Testcase 'Test_05_2sComplement.java' passed
*****************************
```

5.2.2 Example: Failed Testcase

When a testcase fails, it looks something like this:

```
*****************************
* TESTCASE 'Test_03_ADD.java' FAILED
*****************************

     ----- diff OUTPUT ----
4c4
<   00100000100100110010111001111011
---
>   00000000000000010000010101010001
6c6
<   carryOut = 1
---
>   carryOut = 0
     ----- END diff ----
```

5.2.3 Example: If You Use Addition

If the script finds that you used addition:

```
ERROR: The grading script found that you used + or += in your C or Java code -
       your grade will be cut in half.
```

5.2.4 Example: Summary

Look at the end of the output for your score:

```
*****************************************
*            OVERALL REPORT
* attempts: 10
* passed:   10
*
* score:    70
*****************************************
```

5.3 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!

6 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.