CS 252: Computer Organization

Sim #1

Basics of Binary

(see class website for due date)

Don't miss this handy YouTube video:

- C for Java Programmers (sim1.c Overview)
  `https://youtu.be/9vy3AAMw3HA`

1 Purpose

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

1.1 What You'll Be Doing

- You will implement a few individual gates as Java classes; these are very simple, and mostly function as an introduction to how this Java simulation works.
- You will implement a 32-bit adder as another Java class; it will perform every step of the addition (column by column) using logical statements - **never addition.** The purpose of this part of the project is twofold: one, to give you practice expressing complex conditions as logical statements, and two, to fully convince you that "dumb" hardware can do **everything** required to perform addition.
- You will implement another class, which performs 2's complement on a 32-bit input; we'll require that this uses your NOT class to negate the input (you can't do it by hand), and also require that you use your ADD class. The purpose of this one is to practice with hardware abstractions: that is, of using simple components as elements in larger pieces of hardware.
- Your last Java class will be a subtractor - which will be built out of your 2's complement and adder classes.
- You will write a single C function. This function will implement addition, using the same rules as your Java class (that is, you have to do it with logic, not with `+`). But for this function, your inputs and outputs will be integers (instead of arrays of booleans), and thus you will have to use bit shifting and masking to read and write individual bits inside integers.

  This function will also implement subtraction - but using a more streamlined fashion. Instead of performing 2's complement and then adding, you

will actually **mix** the 2's complement operation and addition into a single step.

The purpose of this function is twofold: first, to start to build a connection in your mind between integers and their hardware, **bit-wise** implementation; and second, to teach you the basics of bit shifting and masking.

1.2 Languages

This project will use both C and Java. We won't be doing anything complex in either language. If you feel a little rusty, feel free to ask questions or Discord, or in Office Hours.

You will be implementing the adder 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. Finally, repetition helps retention.

2 Tasks

You must implement a total of 6 different Java classes and 1 C file, representing a gradually increasing level of complexity. I encourage you to implement them in the order listed above, so that you can gradually build on your understanding.

For each of the required files, I've provided a skeleton file to get you started. Pay attention to the comments I've left, as well as any example code I provide.

Don't leave the C function to the last minute. While it is a repetition of the adder that you already implemented in Java (with a few extra features), students often find it more challenging than they would have guessed - since few of you are already familiar with how to read and write individual bits inside an integer.

2.1 You Can't Do Addition!

None of the code that you write for 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 `for()` 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)[1]

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)

2.1.1 Can You Please Regrade My Code?

Sometimes, students think that the grading script is unfair. It will mark them down for uses of `+` that they didn't think should be a problem, and they ask us to overrule the grading script by hand.

There are two common scenarios where this happens. **Pay attention to what I'm about to say here.**

- In Comments

  The grading script is not smart enough to parse your code and realize that your `+` is inside a comment. So, even though you're not using it in your code, you get marked down.

  ANSWER: No, I won't fix this.

  You have the grading script. Run it before you turn in your code! If your comment has a problem, then change the comment before you turn it in.

- In Array Indexing

  Some students write their code in such a way that they need to access element `[i+1]` or `[i-1]` while they iterate through the array. To change this would require that they rethink their algorithm in a non-trivial manner.

  ANSWER: Yes, I'll fix this. But I wish I didn't have to.

  Do your best to devise an algorithm, from the beginning, that doesn't need to do this. But if this is your best answer, I'll allow it.

2.2 Base Code

Start your work by downloading the zipfile from the class website.

You should start by looking at:

[1] Why is `!=` used as the boolean XOR operator? Would it also work as a bitwise XOR operator?

- `Sim1_XOR.java`

  This is an **example,** which has lots and lots of comments to explain what's going on in the Java simulation. You don't need to do anything with this file; it's just for your education.

- `sim1.h`

  This is the "header" file for the C part of the project. **You must not modify this file** - but you should read it, to understand what sorts of inputs I will provide to you, and what sorts of outputs you must produce.

You should fill in the **TODO** marks in the following files:

- `Sim*.java` (except for XOR)

  These represent the various features you must implement. (XOR is an example of how to do composition, you don't have to change that.)

  Start with `AND/OR/NOT`. Then implement `ADD`. Do `2sComplement` next, followed by `SUB`.

  Make sure that you read the entire spec - some of these classes have **extra restrictions** about how you will implement them.

- `sim1.c`

  This contains one function to implement, which performs ADD. (And, when a certain flag is turned on, it also performs SUB - but you will find out that the two are **very** similar.)

  Do this one last, but don't leave it to the last minute. Leave some extra time to complete it.

You **should not modify** the following files:

- `grade_sim1`

  This is the grading script that I've provided, for testing your code. It should work well on CS department machines; it's also possible (no guarantees) that it might work well on your personal Mac or Linux machine.

  (The command `timeout`, which is used by the grading script, is missing from most Macs. But you can Google to find out how to get it installed. I don't recall the details, I'm not a Mac user.)

- `RussWire.java`

  This implements `RussWire`, a necessary utility class for all of the other Java classes (see below).

- `Test*.java`
  `test*.c`

  These are the source files for the testcases.

- `*.out`

  These show the expected output from each testcase.

- `sim1.h`

  This is the header which your C code will use to communicate with each testcase.

3 The `RussWire` Class

In hardware, it is never possible for the same wire to carry two different signals during the same "clock cycle" - that is, during the same unit of time. In this project, every time we run a testcase, we are simulating a single clock cycle - which means that every input and output, from every different object, can only have one value, forever. (Of course, if you create multiple instances of the same object, each instance can have different values.)

However, we don't know all of the values at once. Instead, information must flow from our inputs, into some components, which calculate some tiny part of the answer; that is fed forward, to other components, which calculate a little more, and on through the network, until we finally determine our output. In hardware, this all happens naturally; we just energize the circuits, wait a few picoseconds, and the correct value arrives at the output.

Unfortunately, when we simulate this with code, we have to be a little more careful. We must be careful to execute all of our components in a specific order, and make sure that we set each "wire" within our simulation no more than once. We also need to sanity-check our code, and ensure that we never read a "wire" before it's been set by something else.

The `RussWire` class does this for you. It basically is just a wrapper around a `boolean`, along with a flag that indicates whether or not the flag has been set yet. It has two basic rules:

- If you try to read it before it has been set, it throws an exception. (Sorry, no default values here!)
- If you try to set it a second time, it throws an exception. (No changing the value after you've set it.)

Basically, if you see an exception from this class, it means that your simulation has a bug that needs to get fixed.

3.1 Irritations of the `RussWire` Class

All of the inputs and outputs of your classes must be `RussWire` classes (or arrays of such classes). Unfortunately, this means that things will be slightly more annoying.

First, the constructor for every class must create a `RussWire` object for every bit of input or output. You must also declare these variables as public fields of your class:

```
public class Example
{
    public void execute() { ... }

    public RussWire in, out;

    public Example()
    {
        in  = new RussWire();
        out = new RussWire();
    }
}
```

When you need an array of bits, you need to create the array object first, then fill it in with individual wire objects:

```
RussWire[] arr = new RussWire[32];
for (int i=0; i<32; i++)
    arr[i] = new RussWire();
```

(If you have an array of inputs or outputs, remember that you need to declare the variable in the class, **NOT** inside the constructor function - or else it won't be publicly visible to other objects!)

Finally, the most annoying part: you cannot use `=` to assign values. Instead, call `get()` to read a value from one wire, and `set()` to write to another. (If you need to hard-code a value, you can simply call `set()` with a `boolean` constant.)

```
boolean value = input_wire.get();
output_wire.set(value);
other_hardcoded_output_wire.set(true);
```

4 Rules for Simulating Hardware

Since you are simulating hardware, there are a number of rules about what you can and can't do in `execute()`. I know, these rules sound silly, and if this was just a program, you'd be right. But the whole idea is that we are **pretending** that we are actually wiring little components together with physical wires. These limitations represent the real limitations of hardware.

In the `execute()` method, you **MUST NOT:**

- Create any new objects. Instead, you must do this in the constructor.

  (Creating and using Java primitives - such as `boolean` - is OK. Basically, just don't use the keyword `new` inside your `execute` method.)

- Ever perform addition or any other mathematical operation - except for the `++` operator which is required for a `for()` loop.

  (Remember, logical operators are allowed.)

In the constructor for a Java class, you may create new objects - but save them as public variables of the object, so that you can use them later. (See my XOR code for an example.)

5 Required Functions

Each Java class implements some inputs, an `execute()` method, and some outputs. My skeleton code has already defined the inputs and outputs, and provided an (empty) `execute()` method; you must write the body of `execute()`.

For most classes, you will also need to fill in the constructor for the class - use that to create the `RussWire` objects that you need, or any other sub-components.

For some of the classes, in addition to the input and output wires that I've defined, you will also need to add some more variables (for sub-components). Feel free to add these - just make sure that you initialize them properly in your constructor as well.

The following classes have special limitations you should pay attention to:

5.1 `Sim1_ADD`

This class has two 32-bit inputs, and several outputs. The primary output, `sum`, is a 32-bit array, and it should contain the output from the addition. The `carryOut` wire should indicate whether or not there was carry-out from the MSB, and the `overflow` wire should indicate whether or not overflow occurred.

You should treat both inputs (and the output) as 32-bit signed integers. Remember, the fact that they are signed won't affect how you perform addition, but it **will** affect how you determine whether or not overflow occured.

Element `[0]` in each array is the **least** significiant bit, and element `[31]` is the most.[2]

5.1.1 Challenge: Elegance

Lots of students implement ADD with a long series of `if/else` statements, and that's fine; it will earn full credit if it's correct.

However, challenge yourself to find elegant, one line solutions to key steps. In particular, it is possible (using only AND, OR, NOT, and XOR) to perform each of the following steps in one line of code:

- Calculate the `sum` bit, for one column, using the `a,b` and carry-in values for that column.
- Calculate the carry-out, for one column, using the `a,b` and carry-in values for that column.

[2] Remember how I told you in lecture that we didn't always draw our integers in the normal order? Here's an example. I said that `[0]` is the LSB, which makes sense to me because the LSB is worth 2^0. But if we drew the array as a picture, `[0]` would be the leftmost element in the array!

That's why having precise terminology like LSB/MSB is so useful.

- Calculate the `carryOut` output from the entire adder, once you've calculated the `sum,carryOut` from each column.
- Calculate the `overflow` output from the entire adder, once you've calculated the `sum,carryOut` from each column.

  (This one is the hardest one, but the one most worth working at. It's shocking how simple it is, once you get your head around it.)

5.2 `Sim1_2sComplement`

This Java class simply calculates the 2's complement of a 32-bit input. However, it **must not** do the addition inside its own `execute()` method! Instead, it **must** have sub-components inside it (see my XOR code for an example), and must use those components to actually perform the calculations.

In other words, this `execute()` method must:

- Use your `NOT` class to negate the inputs (but you'll need to negate 32 items - how will you do that?)
- Use your `ADD` class to add 1 to the negated values.

5.3 `Sim1_SUB`

This must perform subtraction. It has (almost) the same inputs and outputs as the `ADD` class - the only difference being that it doesn't have `carryOut` or `overflow` outputs.

However, like your `2sComplement` class, this **must not** do the work itself. Instead, compose it from the `2sComplement` and `ADD` classes.

5.4 C Code: `execute_add(Sim1Data *obj)`

Your C code must implement only a single function, which performs 32-bit signed addition, much like the `Sim1_ADD` class in Java; however, it has a few additional features.

Since C doesn't have classes, the inputs and outputs for this function are all stored in a `struct` which is passed (by pointer) to your C code.

More importantly, the inputs and outputs are not arrays. Instead, they are actual 32-bit integers. None the less, you **must not** use the C addition operator to calculate the result. Instead, you **must** perform bitwise addition, just like the Java class.

To do this, you will need a `for()` loop which iterates over the 32 bits; for each bit, it should perform 4 basic tasks:

- Use bit shifting and masking (see below) to extract one bit from input `a`, and one bit from input `b`.
- Figure out what the sum and carry bits are for the current column.

- Save the carry into a temporary variable of some sort (for use in the next iteration of this loop).
- Write the sum-bit to the output - again, using bit shifting and masking.

5.4.1 Other Flags

The output struct has three additional flags which are not present in the Java version (see `sim1.h`). The fields `aNonNeg,bNonNeg,sumNonNeg` indicate whether or not the two inputs (and the sum) are non-negative[3]. To calculate these values, you **must not** use greater than/less than operators. Instead, you must extract the MSB from each of the three numbers, and then set the three `NonNeg` fields appropriately.

NOTE: When you set the `bNonNeg` field in `execute_add()`, you should set it based on the **original** (not negated) value of the `b` input - even if we are performing subtraction!

5.4.2 Subtraction

As you know, subtraction in a computer is simply a 2's complement of the 2nd term, and then addition. In our Java version of the program, we use your `NOT` and `ADD` classes to perform 2's complement, and then use the ordinary `ADD` class to add the values together.

However, you will be using another method in `execute_add()` - one which is more true to how it works in hardware. Jump ahead to slide deck 5 (ALUs), slides 14-16. You'll see that, inside a computer, we don't perform the 2's complement and add as two separate steps. Instead, we simply perform (bitwise) negation of the 2nd input - and we **also** set the "carry-in" to 1.

The "carry-in" bit, in an adder, is the carry that comes up from the previous column. Normally (that is, when we're adding), the first column has a carry-in which is zero. But, in this case (when we're doing subtraction), we will set it to 1 - meaning that (in essence) we are performing an extra `+1` to our numbers.
In `execute_add()`, you must:

- Read the `isSubtraction` field out of the struct.
- If `isSubtraction` is set to 1, then do a carry-in to the very first column of your adder.
- If `isSubtraction` is set to 1, then **negate** each bit of the second input, as you perform the addition.

Pay attenttion to the following rules:

- You must **NOT** use the `-` operator (arithmetic negation) to simply negate the input. Instead, you **must** use bitwise logic!

[3]Remember, "non-negative" means "positive or zero."

- You must **NOT** split the ADD and SUB features into two different pieces of code - not even two sides of an `if()`. You must have **one** loop, which does all of the work of the adder (for both addition and subtraction).

  Don't worry, this isn't crazy-hard. In fact, turning an adder into a subtractor will only require you to change a few lines.

5.4.3 2's Complement, Special Cases

When we perform 2's complement, there are two special values: 0 and the most-negative number (`0x0000_0000, 0x8000_0000`). In both cases, the 2's complement has the same sign as the original number! This means that it can be a little confusing to decide whether or not overflow has occured.

For that reason, we'll promise to never set `b` to either of these numbers while performing subtraction.

6 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. [4]

  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

[4] 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!

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)
  ```

7 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!

7.1 Testcases

In this project, we have C testcases (named `test_*.c`) and Java testcases (named `Test_*.java`). You can find all of them in the zipfile.

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[5].

[5] **NOTE:** Do not simply cut-n-paste the file, since that will mess up the whitespace in the file! Instead, make sure to **download** the file, so that you get an exact copy.

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.** You are also encouraged to share your testcases on Discord!

7.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_sim1`. 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:

7.2.1 Example: Passed Testcase

When a testcase passes, it looks something like this:

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

7.2.2 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 ----
```

7.2.3 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.
```

7.2.4 Example: Summary

Look at the end of the output for your score:

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

7.3 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 Discord. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

8 Turning in Your Solution

You must turn in your code to GradeScope. Turn in only your program; do not turn in any testcases.