CS-UY 2214 — Project 1

Jeff Epstein, Ratan Dey

1 Introduction

This project represents a substantive programming exercise. Like all work for this class, it is to be completed individually: any form of collaboration is prohibited, as detailed in the syllabus. This project is considered a take-home exam.

Before even reading this assignment, please read the E20 manual thoroughly. Read the provided E20 assembly language examples.

2 Assignment: Assembler

Your task is to write an E20 assembler: a program that will convert E20 assembly language into E20 machine language.

Each E20 assembly language instruction can be expressed as a 16-bit E20 machine language instruction. The rules for converting between the two forms is given in the E20 manual, particularly the chapter on the E20 instruction set.

For example, consider the assembly language instruction `addi $1, $2, 3`. The opcode is `addi`, used for performing addition of an immediate. In this case, we are adding the number 3 to the value currently stored in register `$2`, and storing the result into register `$1`. The corresponding machine code instruction is `0010100010000011` in binary, or 59523 in decimal, or `e883` in hex. Each component of the assembly language instruction maps onto a region of its machine language counterpart. Below, we color-code each field to show how they match:

```
    addi $1, $2, 3     <==>     0010100010000011
```

As you can see, the first (most significant) three bits of the machine code instruction correspond to the opcode; the next three bits indicate the source register; the next three bits indicate the destination register; and the last seven bits store the immediate.

Unlike the E15, in the E20, different instructions have different formats, and your program will need to take that into account.

The purpose of your assembler is to make it easier to write programs for the E20 processor. Eventually, we will execute the machine code generated by your assembler on a simulated E20 processor.

2.1 Input

The input to your assembler will be the name of an E20 assembly language file, given on the command line. By convention, E20 assembly language files have an `.s` suffix.

Your program will read in the contents of the file. You may assume that the file contains well-formed E20 assembly language code. The file may contain comments, which your program should ignore.

You are provided with several examples of valid E20 assembly language files, which you can use to test your assembler.

Here is an example of an E20 assembly language program, in a file named `loop2.s`:

```
    movi $1, 10
beginning:
    jeq $1, $0, done
    addi $1, $1, -1
    j beginning
done:
    halt
```

2.2 Output

Your program should print to stdout the E20 machine code corresponding to its input.

Below is an example invocation of an assembler from Linux's `bash`. In this case, we are assembling the assembly language program given above. Text in italics represents a command typed by the user.

```
user@ubuntu:~/e20$ ./asm.py loop2.s
ram[0] = 16'b0010000010001010;      // movi $1,10
ram[1] = 16'b1100010000000010;      // beginning: jeq $1,$0,done
ram[2] = 16'b0010010011111111;      // addi $1,$1,-1
ram[3] = 16'b0100000000000001;      // j beginning
ram[4] = 16'b0100000000000100;      // done: halt
```

Your assembler should produce output in exactly the format shown above. That is: for each assembly language instruction, print a line of machine code, consisting of the memory address, followed by an equals sign, followed by a 16-bit binary number in Verilog syntax. Your program is responsible for determining the memory address of each instruction. The instructions should be printed in sequential order of their address. You are not responsible for printing the value of memory addresses that are not specified by the input.

In the above listing, each line of machine code includes a comment indicating the corresponding assembly language code. Your solution does *not* need to output the comments. However, outputting comments in this way may be helpful as you debug your program.

Your solution will be checked mechanically, so it is important that your assembler produce machine code output identical to the machine code output above. Please avoid losing points for superficial deviations.

2.3 Testing

Several example assembly code files have been provided for you. Each example file includes, in comments, the expected machine code, as well as the expected execution result. You can use these examples to verify the correctness of your assembler. However, you should not rely exclusively on these examples, as they are not sufficient to exercise every aspect of an assembler. You are therefore expected to develop your own test cases.

2.4 Starter code

You may, but are not required to, use the provided starter code for this assignment, found in the files `asm-starter.cpp` and `asm-starter.py`. Please rename them to `asm.cpp` or `asm.py`, as appropriate.

3 Hints

- In order to run a Python program from the Linux command line, it must first be marked executable. Otherwise, you may get a "permission denied" error message.

  To mark your Python file as executable, use the following command (assuming your file is named `asm.py`) from `bash`:

```
chmod u+x asm.py
```

Also make sure that the first line of the file specifies the path to the Python interpreter: it should be `#!/usr/bin/python3`. See the provided starter code. If you get an "exec format error," the problem is usually that that the first line is wrong.

Alternatively, you can run the program by typing `python3 asm.py`.

- For this assignment, you will have to manipulate individual bits within a number. Many students are tempted to treat binary numbers as strings of `"0"` and `"1"`, however, this approach is problematic and leads to lots of bugs. We will therefore discuss *bitmasks*, a better approach for manipulating numbers.

  First, we have to acknowledge that a binary number is just a number. All integers are stored in binary, regardless of the syntax we use to write them. There is no distinct "binary" type in your programming language. Nevertheless, we sometimes want to access or modify individual bits within an integer. The bit-twiddling operators (`&`, `|`, $\sim$) are perfect for this.

First example: extracting certain bits Let's say I want to get only the two least significant bits from an 8-bit number `n`. If `n` is 42, it would look like this, with the digits I want shown in green:

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

In other words, the desired results is 10_2, or 2_{10}. (In the above diagram, notice that each bit in the 8-bit number `n` is labeled with its significance: the rightmost bit is in the one's place, then the two's place, then four's place, etc.)

A naive approach to solve this problem would be to convert the number into a string and extract its last two characters, then convert back into integer:

```
int(bin(n)[-2:], 2)
# returns 2 decimal
```

A better approach is to AND the number `n` with a value having 1s in the bit positions we want, and every other position 0. Such a value is called a bitmask. Suppose that I decide to AND `n` with the bitmask $3_{10} = 11_2$:

$$\begin{array}{c|cccccccc} & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ \& & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

The bitwise AND operator takes the logical AND of each pair of bits from its two inputs. The result is just what we wanted: 10_2, or 2_{10}. In other words, the bitmask shows us which bits to select from `n`. We can calculate this elegantly in Python like this:

```
n & 3
```

Equivalently, we can express the bitmask in binary, which makes our intent clearer:

```
n & 0b11
```

Second example: constructing values As another example: let's say I want to construct an 8-bit number where the most significant 3 bits are the 3-bit number `a`, and the least significant 3 bits are the 3-bit number `b`, and the other bits are zero. Both `a` and `b` consist of three bits each: a_2, a_1, a_0, and b_2, b_1, b_0 respectively, given in most-to-least significant order. In other words, my final result should look like this:

a_2	a_1	a_0	0	0	b_2	b_1	b_0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

A naive approach would be to convert `a` and `b` into binary strings and concatenate them with two zeros in between, then convert the result back into an integer:

```
int(bin(a) + "00" + bin(b), 2)
```

I propose a better solution. The first step of my approach is to adjust the three bits of `a` so they are in the right place. We need to shift `a` five bits to the left, so that `a`'s most significant bit is in the 2^7 position of our result. In Python:

```
a << 5
```

Now it's enough to combine the left-shifted `a` with `b` in its original position. We can use bitwise OR for this:

$$\begin{array}{c|cccccccc} & a_2 & a_1 & a_0 & 0 & 0 & 0 & 0 & 0 \\ | & 0 & 0 & 0 & 0 & 0 & b_2 & b_1 & b_0 \\ \hline & a_2 & a_1 & a_0 & 0 & 0 & b_2 & b_1 & b_0 \end{array}$$

We can now express the final result in Python:

```
a << 5 | b
```

Note that this above expression works only if `a` and `b` are limited to 3 bits. If they're larger, we may get an unexpected result. Fortunately, there's an easy fix: we can consider only the least-significant three bits of `a` and `b` by ANDing with a bitmask. Here, I'm using the bitmask 7_{10}, which in binary is 111_2, in other words, a bitmask that ignores everything except the three least-significant bits:

```
(a & 7) << 5 | (b & 7)
```

In E20, each machine code instruction is just a 16-bit number where certain bits are assigned certain meaning. I hope you can see how this hint is applicable to your assignment.

- Your assembler should never crash for any valid input. In this case, "valid input" means an E20 assembly language program conforming to the syntax described in the E20 manual. You should read the E20 manual thoroughly before you start coding to make sure you understand the syntax.

4 Rules

Language You should implement this project in Python 3 or in C++.

File names and building If you are using Python 3, you must name your program `asm.py`. If your solution consists of multiple source files, submit them as well. Assume that your program will be invoked by running `asm.py` with a filename as its parameter, using Python 3.6.

If you are using C++, you must name your program's main source file `asm.cpp`. If your solution consists of multiple source files, submit them as well. Assume that your program will be built by gcc 8.3.x using the command `g++ -Wall -o asm *.cpp` and then run by the executable `asm` with a filename as its parameter. If you use C++, your program should compile cleanly (i.e. no errors or warnings) with gcc 8.3.x.

Libraries You are free to make use of all packages of the standard library of your language (that is, all libraries that are installed by default with Python 3 or C++, respectively). Do not use any additional external libraries. Do not use any OS-specific or compiler-specific extensions.

Tools Your program submission will be evaluated by running it under the GNU/Linux operating system, in particular a Debian or Ubuntu distribution. Your grade will therefore reflect the behavior of your project code when executed in such an environment. While you are welcome to develop your project under any operating system you like (such as Windows or Mac OS), you are responsible for any operating system-dependent deviations in program behavior.

Academic integrity You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else. Code may not be developed collaboratively. You may rely on publicly-accessible documentation of the language and its libraries. Please read the syllabus for detailed rules and examples about academic integrity.

Code quality You should adhere to the conventions of quality code:

- Indentation and spacing should be both consistent and appropriate.
- Names of variables, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.
- All functions should have a documenting comment in the appropriate style describing its purpose, behavior, inputs, and outputs. In addition, where appropriate, code should be commented to describe its purpose and method of operation.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in grading your work.

Submission You are obligated to write a `README` file and submit it with your assignment. The `README` should be a plain text file (not a PDF file and certainly not a Word file) containing the following information:

- Your name and NYU email address.
- The state of your work. Did you complete the assignment? If not, what is missing? Be specific. If your assignment is incomplete or has known bugs, I prefer that students let me know, rather than let me discover these deficiencies on my own.
- Any other resources you may have used in developing your program.
- Justify your design decisions. Why did you write your program the way you did? If you feel that your design has notable strengths or weaknesses, discuss them.

Submit your work on Gradescope. Submit all source files necessary to build and run your project. Do not submit external library code. Do not submit binary executable files.