CISC 340 – Project 1

Due dates:
Assembler (Section 4): 11:59 pm, Wednesday, October 10
Simulator (Sections 5&6): 1:35 pm, Monday, October 15
(10 points)

1. Purpose

This project is intended to help you understand the instructions of a very simple assembly language, how to assemble programs into machine language, and how those machine instructions are executed.
You will write procedural C.
You will use gcc on the departmental linux system (hank).
You will do this project with a partner.

2. Problem

This project has three parts:

1. You will write a program to take an assembly-language program and produce the corresponding machine language.
2. You will write a behavioral simulator for the resulting machine code.
3. You will write a short assembly-language program to multiply two numbers.

3. UST-3400 (Rip Van saWinkle) Instruction-Set Architecture

In class this semester, we will be gradually "building" the UST-3400 computer. The UT-3400 is very simple, but it is general enough to solve complex problems. For this project, you will only need to know the instruction set and instruction format of the UST-3400.

The UST-3400 is an 8-register, 32-bit computer. All addresses are **word-addresses** and the memory is aligned in a **big-endian** manner. The UST-3400 has 65536 words of memory. By assembly-language convention, register 0 will always contain 0 (i.e. the machine will not enforce this, but no assembly-language program should ever change register 0 from its initial value of 0).

There are 4 instruction formats/types. Bits 31-25 are unused for all instructions, and should be 0. (Yes, they're wasted – so sue me)

(bit 0 is the least-significant bit)

R-type instructions (add, nand): bits 24-22: opcode bits 21-19: reg A bits 18-16: reg B bits 15-3: unused (should all be 0) bits 2-0: destReg	**J-type instructions** (jalr): bits 24-22: opcode bits 21-19: reg A bits 18-16: reg B bits 15-0: unused (should all be 0)
I-type instructions (lw, sw, beq): bits 24-22: opcode bits 21-19: reg A bits 18-16: reg B bits 15-0: offset_field (a 16-bit, 2's complement number with a range of -32768 to 32767)	**O-type instructions** (halt, noop): bits 24-22: opcode bits 21-0: unused (should all be 0)

Table 1: Description of Machine Instructions

ASM Instruction	Format	Opcode in binary (bits 24-22)	Action
add	R-type	000	Add contents of regA with contents of regB, store results in destReg.
nand	R-type	001	Nand contents of regA with contents of regB, store results in destReg. This is a **bitwise** nand; each bit is treated independently.
lw	I-type	010	Load regA from memory. Memory address is formed by adding offsetField with the contents of regB.
sw	I-type	011	Store regA into memory. Memory address is formed by adding offsetField with the contents of regB.
beq	I-type	100	If the contents of regA and regB are the same, then branch to the address PC+1+offsetField, where PC is the address of the beq instruction.
jalr	J-type	101	First store PC+1 into regA, where PC is the address of the jalr instruction. Then branch to the address contained in regB. **Note:** This explicit ordering implies that if regA is the same as regB, the processor will first store PC+1 into that register, then end up branching to PC+1.
halt	O-type	110	Increment the PC (as will all instructions), then halt the machine (let the simulator notice that the machine halted).
noop	O-type	111	Do nothing except increment the PC.

4. UST-3400 Assembly Language and Assembler (40%)

The first part of this project is to write a program to take an assembly-language program and translate it into machine language (bits). You will translate assembly-language names for instructions, such as ***beq***, into their numeric equivalent (e.g. 100), and you will translate symbolic names for addresses into numeric values. The final output will be a series of numbers representing the machine-code version of the 32-bit instructions.

As discussed in lecture, the format for a line of assembly code is:

label<ws>instruction<ws>field0<ws>field1<ws>field2<ws>comments

where <ws> means any series of whitespace (tabs and/or spaces).

The leftmost field on a line is the label field. Valid labels contain a maximum of 6 characters and can consist of letters and numbers (but must start with a letter). The label is optional (**the white space following the label field is required, even if there is no label!**). Labels make it much easier to write assembly-language programs, since otherwise you would need to modify all address fields each time you added a line to your assembly-language program!

After the optional label is **non-optional** white space. Then follows the instruction field, where the instruction can be any of the assembly-language instruction names listed in the above table. After more white space comes a series of fields. All fields are given as decimal numbers or labels. The number (and interpretation) of fields depends on the instruction, and unused fields should be ignored (treat them like comments).

R-type instructions (add, nand) require 3 fields:
field0 is destReg, field1 is regA, and field2 is regB.

I-type instructions (lw, sw, beq) require 3 fields:
field0 is regA, field1 is regB, and field2 is either a numeric value for offset_field or a symbolic address. Numeric offset_fields can be positive or negative; symbolic addresses are discussed below.

J-type instructions (jalr) require 2 fields:
field0 is regA, and field1 is regB.

O-type instructions (noop and halt) require no fields other than the opcode.

Symbolic addresses refer to labels. For lw or sw instructions, the assembler should compute offset_field to be equal to the address of the label. This could be used with a zero base register to refer to the label, or could be used with a non-zero base register to index into an array starting at the label. For beq instructions, the assembler should translate the label into the numeric offset_field needed to branch to that label.

After the last used field comes more white space, then any comments. The comment field ends at the end of a line. Comments are vital to creating understandable assembly-language programs, because the instructions themselves are rather cryptic. (Thus, the invention of high-level languages…)

In addition to UST-3400 instructions, an assembly-language program may contain directions for the assembler. The only assembler directive we will use is .fill (note the leading period). .fill tells the assembler to put a number into the place where the instruction would normally be stored. .fill directives use one field, which can be either a numeric value or a symbolic address. For example, ".fill 32" puts the value 32 at the address where an instruction in the same slot would normally be stored. .fill with a symbolic address will store the address of the label. In the example below, ".fill start" will store the value 2, because the label "start" is at address 2.

Major Hint: The assembler should make **two passes** over the assembly-language program. In the first pass, it will calculate the address for every symbolic label. **Assume that the first instruction is at address 0.** In the second pass, it will generate a machine-language instruction (output in decimal) for each line of assembly language. For example, on the following page is an assembly-language program (that counts down from 5, stopping when it hits 0):

```
        lw      1       0       five    load reg1 with 5 (symbolic address)
        lw      2       1       3       load reg2 with -1 (numeric address)
start   add     1       2       1       decrement reg1
        beq     0       1       2       goto end of program when reg1==0
        beq     0       0       start   go back to the beginning of the loop
        noop
done    halt                            end of program
five    .fill   5
neg1    .fill   -1
stAdd   .fill   start                   will contain the address of start (2)
```

And here is the corresponding machine language:

```
(address 0):    8912903     (hex 0x00880007)
(address 1):    9502723     (hex 0x00910003)
(address 2):    1114113     (hex 0x00110001)
(address 3):    16842754    (hex 0x01010002)
(address 4):    16842749    (hex 0x0100FFFD)
(address 5):    29360128    (hex 0x01C00000)
(address 6):    25165824    (hex 0x01800000)
(address 7):    5           (hex 0x00000005)
(address 8):    -1          (hex 0xFFFFFFFF)
(address 9):    2           (hex 0x00000002)
```

Since your programs will always start at address 0, your program should only output the contents, not the addresses. Your final output should look like this:

```
8912903
9502723
1114113
16842754
16842749
29360128
25165824
5
-1
2
```

4.1. Running Your Assembler

Write your program to take one or two command line inputs. The first input is the file name where the assembly-language program is stored, and the second input is the file name where the output (the machine-code) is to be written. In the following examples, the assembly-language program to be assembled is "program.as" and the machine-code file generated would be "program.mc":

```
# One input example
$./assembler -i program.as > program.mc
# Two input example
$./assembler -i program.as -o program.mc
```

Your program should store only the list of decimal numbers in the machine-code file, one instruction per line. **Any deviation from this format (e.g. extra spaces or empty lines) will render your machine-code file unusable (and ungradeable! → automatic zero).** Any other output that you want the program to generate (e.g. debugging output) can be printed to standard error.

4.2 Error Checking

Your assembler should catch the following errors in the assembly-language program:

- use of undefined labels (start with a number, include weird characters, too long, etc.)
- duplicate labels
- offset_fields that don't fit in 16 bits
- unrecognized opcodes

Your assembler should exit early and print an error message if it detects an error. Your assembler should **NOT** catch simulation-time errors, i.e. errors that would occur at the time the assembly-language program executes (e.g. branching to address -1, infinite loops, etc.).

4.3 Test Cases

An integral (and graded) part of writing your assembler will be to write a suite of test cases to validate any UST-3400 assembler. This is common practice in the real world--software (and hardware) companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of the project specification and your program, and it will help you a lot as you debug your program.

The test cases for the assembler part of this project will be short assembly-language programs that serve as input to an assembler. You will submit your suite of test cases together with your assembler, and we will grade your test suite according to how thoroughly it exercises an assembler. Each test case may be at most 50 lines long, and your test suite may contain up to 20 test cases. These limits are much (MUCH) larger than needed for full credit (done carefully, test suites composed of 5 test cases, each < 10 lines long can be easily sufficient).

Here is a free test case: Make sure you can handle both kinds of whitespace when parsing an assembly instruction, tabs and spaces. If you only test one and I use the other when I check your program, it will likely crash.

Hints: the example assembly-language program above is a decent case to include in your test suite, though you'll need to write more thorough (and targeted) test cases to get full credit. Besides testing

normal operations, remember to create some test cases that test the ability of an assembler to check for the errors in Section 4.2.

4.4 Assembler Hints

Since offset_field is a 2's complement number, it can only store numbers ranging from -32768 to 32767. For symbolic addresses, your assembler will compute offset_field so that the instruction refers to the correct label. Remember also that offset_field is only a 16-bit 2's complement number. Since most platforms' integers are 32 bits, you'll have to chop off all but the lowest 16 bits for negative values of offset_field.

5. Behavioral Simulator (40%)

The second part of this assignment is to write a program that can simulate any legal UST-3400 machine-code program. The input for this part will be the machine-code files that you created with your assembler. Your program should take the machine-code filename as a command line argument (similar to the assembler's).

The simulator should begin by initializing all registers and the program counter to 0.
The simulator will then simulate the program until the program executes a halt.

The simulator should call print_state() (included below) before executing each instruction and once just before exiting the program. This function prints the current state of the machine (program counter, registers, memory). print_state will print the memory contents for memory locations defined and initialized in the machine-code file (addresses 0-9 in the Section 4 example). The final output is print_stats() (included below), which will print out the total number of executed instructions.

5.1 Test Cases

As with the assembler, you will write a suite of test cases to validate any UST-3400 simulator.

The test cases for the simulator part of this project will be short, valid assembly-language programs that, after being assembled correctly into machine code, serve as input to a simulator. You will submit your suite of test cases together with your simulator, and I will grade your test suite according to how thoroughly it exercises an UST-3400 simulator. Each test case may execute at most 200 instructions on a correct simulator, and your test suite may contain up to 20 test cases. Again, these limits are much (**MUCH**) larger than needed for full credit (carefully selected full test suites can be composed of as little as 2-3 test cases, each executing less than 40 instructions).

5.2. Simulator Hints

Be careful how you handle offset_field for lw, sw, and beq. Remember that it's a 2's complement 16-bit number, so you need to convert a negative offset_field to a negative 32-bit integer by sign extending it. One way to do this is to use the following function (**In other words, you must use this function)**:

```
// convert a 16-bit number into a signed 32-bit integer
int convert_num(int num){
        if (num & (1<<15) ) {
                num -= (1<<16);
        }
        return(num);
}
```

6. Assembly-Language Multiplication (20%)

The third part of this assignment is to write an assembly-language program to multiply two numbers. Input the numbers by reading memory locations called "mcand" and "mplier". The result should be stored in register 7 when the program halts. You may assume that the two input numbers are at most 15 bits and are positive; this ensures that the (positive) result fits in an UST-3400 word. See the multiplication algorithm on page 185 of the textbook. Remember that shifting left by one bit is the same as adding the number to itself (multiplying by 2). Given the UST-3400 instruction set, it's easiest to modify the algorithm so that you avoid the right shift. Submit a program that computes (29562 * 11834).

Your multiplication program must be reasonably efficient--it must be at most 50 lines long and execute at most 1000 instructions for any valid input (this is several times longer and slower than the best solution). To achieve this, you must use a loop and shift algorithm to perform the multiplication; algorithms such as successive addition (e.g. multiplying 5 * 6 by adding 5 six times) will take too long.

7. Grading

I will grade primarily on functionality, including error handling, correctly assembling and simulating all instructions, correctly multiplying, and comprehensiveness of the test suites. I won't be docking you points for program style. That shouldn't prevent you from using good coding practices, however.

The best way to debug your programs is to generate your own test cases, figure out the correct answers by hand (and with the provided solution binary), and compare your program's output to the correct answers. This is also one of the best ways to learn the concepts in the project.

Your suite of test cases for the assembler and simulator parts of this project will be graded according to how thoroughly they test an UST-3400 assembler or simulator. I will judge thoroughness of the test suite by how well it exposes potential bugs in an assembler or simulator. For the simulator test suite, you can assume that the test cases will be assembled correctly. (So you don't need to worry about testing for assembler problems in the simulator test set)

Finally, grading will be performed on the departmental Linux machines (hank). Make sure your programs run correctly on these platforms (they should all be identical), and that **your outputs match EXACTLY the reference assembler/simulator outputs.**

8. Turning in the Project

The entirety of the project will be turned in via canvas. A hard copy of your overview document should also be submitted in class on the day of the due date. The following components (**and structure!**) are necessary for submission:

- Assembler Sub-Folder:
  - code for your assembler
  - a Makefile to compile your code (include a clean rule!)
  - Sub-Folder of assembler test files
  -A README file with instructions and explanations of files.
- Simulator Sub-Folder:
  - code for your simulator
  - a Makefile to compile your code (include a clean rule!)
  - Sub-Folder of assembler test files (assembly code, and machine!)

- A README file with instructions and explanations of files.

• Multiplication assembly code (and machine!)

• Two overview documents (pdf) telling me how your assembler and simulator work, respectively. Each of these documents should also communicate any difficulties you had in developing each program, as well as any shortcomings that remain (e.g., did not finish implementing jalr).

Place all of this stuff inside a directory named "project1_username1_username2" using whatever your login names are. You can then zip this directory using the linux command "zip -r project1_username1_username2.zip project1_username1_username2". Submit the zip file via Canvas. Only one submission per group!
Mac users: creating a zip on a mac includes an unneeded __MACOSX directory. **Do not include this directory.** E.g., do your zipping on a linux or windows machine.

You should hand in the overview documents as hardcopies in class on their respective due dates.

9. C/C++ Programming Tips – Dealing with Bits

Here are a few programming tips for getting programs to manipulate bits:
1) To indicate a hexadecimal constant, precede the number by 0x. (zero x).
For example, 27 decimal is 0x1b in hexadecimal.

2) The value of the expression (a >> b) is the number "a" shifted right by "b" bits.
Neither a nor b are changed. E.g. (25 >> 2) is 6. Note that 25 is 11001 in binary, and 6 is 110 in binary.

3) The value of the expression (a << b) is the number "a" shifted left by "b" bits. Neither a nor b are changed. E.g. (25 << 2) is 100. Note that 25 is 11001 in binary, and 100 is 1100100 in binary.

4) The expression (a & b) performs a logical AND on each bit of a and b (i.e. bit 31 of a ANDed with bit 31 of b, bit 30 of a ANDed with bit 30 of b, etc.). E.g. (25 & 11) is 9, since:

```
11001 (binary)
& 01011 (binary)
---------------------
= 01001 (binary), which is 9 decimal.
```

5) The expression (a | b) performs a logical OR on each bit of a and b (i.e. bit 31 of a ORed with bit 31 of b, bit 30 of a ORed with bit 30 of b, etc.). E.g. (25 | 11) is 27, since:

```
11001 (binary)
& 01011 (binary)
---------------------
= 11011 (binary), which is 27 decimal.
```

6) ~a is the bit-wise complement (inversion) of a (a is not changed).

Use these operations to create and manipulate machine-code. E.g. to look at bit 3 of the variable a, you might do: (a>>3) & 0x1. To look at bits (bits 15-12) of a 16-bit word, you could do: (a>>12) & 0xF. To put a 6 into bits 5-3 and a 3 into bits 2-1, you could do: (6<<3) | (3<<1). If you're not sure what an operation is doing, print some intermediate results to help you debug. **These operations will save you**

enormous amounts of time and energy – make sure you understand how they work. Come ask if you don't!

10. Code Fragment for Simulator

Here is some C code that may help you write the simulator. Again, you should take this merely as a hint. You may have to re-code this to make it do exactly what you want, but this should help you get started.

```
typedef struct state_struct {
        int pc;
        int mem[NUMMEMORY];
        int reg[NUMREGS];
        int num_memory;
} statetype;

void print_state(statetype *stateptr){
        int i;
        printf("\n@@@\nstate:\n");
        printf("\tpc %d\n", stateptr->pc);
        printf("\tmemory:\n");
        for(i = 0; i < stateptr->nummemory; i++){
                printf("\t\tmem[%d]=%d\n", i, stateptr->mem[i]);
        }
        printf("\tregisters:\n");
        for(i = 0; i < NUMREGS; i++){
                printf("\t\treg[%d]=%d\n", i, stateptr->reg[i]);
        }
        printf("end state\n");
}
```

HINT: When calling the print_state() function you can provide the memory address (the pointer) of a statetype like so: print_state(&my_state);

Printing execution statistics will be the final output of the simulator. Use the following code:

```
void print_stats(int n_instrs){
        printf("INSTRUCTIONS: %d\n", n_instrs); // total executed instructions
}
```