

Computer Architecture, Spring 2023

Project 1

Assigned: Friday, 4/14

Due: Monday, 5/8, 11:59 PM

1 Introduction

This project is intended to help you understand the instructions of a very simple assembly language and how to assemble programs into machine language.

2 Problem

This is an individual project and the project has three parts:

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

3 LC-2K Instruction-Set Architecture

In this project, you will be gradually “building” the LC-2K (Little Computer 2000). The LC-2K 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 LC-2K (Not MIPS!!)

The LC-2K is another 32-bit RISC architecture which includes

- 8 x 32-bit registers: reg0¹ to reg7
- Each address is 32-bits and it stores a word
- 65536 words of memory (addressed by word index, not byte)

¹Note that this register will always contain 0, and this is assembly-language convention.

There are 4 instruction formats (bit 0 is the least-significant bit). Bits 31-25 are unused for all instructions, and should always be 0.

R-type instructions (add, nor):

```
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
```

I-type instructions (lw, sw, beq):

```
bits 24-22: opcode
bits 21-19: reg A
bits 18-16: reg B
bits 15-0:  offsetField (a 16-bit value with a range of -32768 to 32767)
```

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)
```

O-type instructions (halt, noop):

```
bits 24-22: opcode
bits 21-0:  unused (should all be 0)
```

4 LC-2K Assembly Language and Assembler (40 pts.)

The first part of this project is to write a program to take an assembly-language program and translate it into machine language. 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 32-bit instructions (instruction bits 31-25 are always 0).

The format for a line of assembly code is :

```
label<white>instruction<white>fld0<white>fld1<white>fld2<white>comments
(<white> means a series of tabs and/or spaces)
```

The leftmost field on a line is label field. Valid labels contain a maximum of 6 characters and can consist of letters and numbers (but must start with a letter). label is optional (the white space following the label field is required). 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!

Assembly language name for instruction	Opcode in binary (bits 24-20)	Action
add (R-type format)	000	Add contents of <code>regA</code> with contents of <code>regB</code> , store results in <code>destReg</code> .
nor (R-type format)	001	Nor contents of <code>regA</code> with contents of <code>regB</code> , store results in <code>destReg</code> . This is a bitwise nor; each bit is treated independently.
lw (I-type format)	010	Load <code>regB</code> from memory. Memory address is formed by adding <code>offsetField</code> with the contents of <code>regA</code> .
sw (I-type format)	011	Store <code>regB</code> into memory. Memory address is formed by adding <code>offsetField</code> with the contents of <code>regA</code> .
beq (I-type format)	100	If the contents of <code>regA</code> and <code>regB</code> are the same, then branch to the address <code>PC+1+offsetField</code> , where <code>PC</code> is the address of this <code>beq</code> instruction.
jalr (J-type format)	101	First store <code>PC+1</code> into <code>regB</code> , where <code>PC</code> is the address of this <code>jalr</code> instruction. Then branch to the address contained in <code>regA</code> . Note that this implies if <code>regA</code> and <code>regB</code> refer to the same register, the net effect will be jumping to <code>PC+1</code> .
halt (O-type format)	110	Increment the <code>PC</code> (as with all instructions), then halt the machine (let the simulator notice that the machine halted).
noop (O-type format)	111	Do nothing.

Table 1: Description of Machine Instructions

After `label` is white space. Then follows `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 of fields depends on the instruction, and unused fields should be ignored (treat them like comments).

R-type instructions (`add`, `nor`) instructions require 3 fields: `field0` is `regA`, `field1` is `regB`, and `field2` is `destReg`.

I-type instructions (`lw`, `sw`, `beq`) require 3 fields: `field0` is `regA`, `field1` is `regB`, and `field2` is either a numeric value for `offsetField` or a symbolic address. Numeric `offsetFields` 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`, `halt`) require no fields.

Symbolic addresses refer to labels. For `lw` or `sw` instructions, the assembler should compute `offsetField` 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 `offsetField` needed to branch to that label.

After the last used field comes more white space, then any comments. `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.

All valid LC-2K programs must have a newline character at the end of each line. Some text editors enforce this for the last line of the file, and some do not. Failing to have a newline on the last line of assembly code can lead to strange bugs that can be quite difficult to find. Thus, be sure to end your assembly language files with newlines. In many editors, you can guarantee this by putting a blank line at the end of the file, and then deleting the blank line. Leaving a blank line in place is not recommended, as this will typically cause assembly to fail with an “unrecognized opcode” error.

In addition to LC-2K 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` instructions use one field, which can be either a numeric value or a symbolic address. For example, `.fill 32` puts the value 32 where the instruction 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. The bounds of the numeric value for `.fill` instructions are -2^{31} to $+2^{31-1}$ (-2147483648 to 2147483647).

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 (in decimal) for each line of assembly language. For example, here is an assembly-language program (that counts down from 5, stopping when it hits 0).

```

                lw      0      1      five    load reg1 with 5 (symbolic address)
                lw      1      2      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
stAddr       .fill     start                will contain the address of start (2)

```

And here is the corresponding machine language:

```

(address 0): 8454151 (hex 0x810007)
(address 1): 9043971 (hex 0x8a0003)
(address 2): 655361 (hex 0xa0001)
(address 3): 16842754 (hex 0x1010002)
(address 4): 16842749 (hex 0x100fffd)
(address 5): 29360128 (hex 0x1c00000)
(address 6): 25165824 (hex 0x1800000)
(address 7): 5 (hex 0x5)
(address 8): -1 (hex 0xffffffff)
(address 9): 2 (hex 0x2)

```

4.1 Running Your Assembler

Write your program to take two command-line arguments. The first argument is the file name where the assembly-language program is stored, and the second argument is the file name where the output (the machine-code) is written. For example, with a program name of “assemble”, an assembly-language program in “program.as”, the following would generate a machine-code file “program.mc”:

```
assemble program.as program.mc
```

Note that the format for running the command must use command-line arguments for the file names (rather than standard input and standard output). Your program should store only the list of decimal numbers in the machine-code file, one instruction per line. The decimal numbers will range from -2^{31} to $+2^{31}-1$ (-2147483648 to 2147483647). Any deviation from this format (e.g. extra spaces or empty lines) will render your machine-code file ungradeable. Any other output that you want the program to generate (e.g. debugging output) can be printed to standard output.

4.2 Error Checking

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

- Use of undefined labels
- Duplicate definition of labels
- `offsetFields` that don't fit in 16 bits
- Unrecognized opcodes
- Non-integer register arguments
- Registers outside the range [0, 7]

Your assembler should `exit(1)` if it detects an error and `exit(0)` if it finishes without detecting any errors. 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 LC-2K assembler. This is common practice in the real world—software 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 5 test cases. These limits are much larger than needed for full credit (the solution test suite is composed of 10 test cases, each up to 20 lines long). See section 7 for how your test suite will be graded.

Hints: the example assembly-language program above is a good case to include in your test suite, though you'll need to write more test cases to get full credit. 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 `offsetField` is a 2's complement number, it can only store numbers ranging from -32768 to 32767. For symbolic addresses, your assembler will compute `offsetField` so that the instruction refers to the correct label.

Remember that `offsetField` is only a 16-bit 2's complement number. Since Linux integers are 32 bits, you'll have to chop off all but the lowest 16 bits for negative values of `offsetField`.

5 Behavioral Simulator (40 pts.)

The second part of this assignment is to write a program that can simulate any legal LC-2K machine-code program. The input for this part will be the machine-code file that you created with your assembler. With a program name of "simulate" and a machine-code file of "program.mc", your program should be run as follows:

```
simulate program.mc > output
```

This directs all `printf()`s to the file "output".

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 `printState()`, 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). `printState()` will print the memory contents for memory locations defined in the machine-code file (addresses 0-9 in section 4 example).

5.1 Test Cases

As with the assembler, you will write a suite of test cases to validate any LC-2K simulator.

The test cases for the simulator part of this project will be short, valid assembly-language programs that, after being assembled into machine code, serve as input to a simulator. You will submit your suite of test cases together with your simulator, and we will grade your test suite according to how thoroughly it exercises an LC-2K simulator. Each test case may execute at most 200 instructions on a correct simulator, and your test suite may contain up to 5 test cases. These limits are much larger than needed for full credit (the solution test suite is composed of a couple test cases, each executing less than 40 instructions). See section 7 for how your test suite will be graded.

5.2 Simulator Hints

Be careful how you handle `offsetField` for `lw`, `sw`, and `beq`. Remember that it's a 2's complement 16-bit number, so you need to convert a negative `offsetField` to a negative 32-bit integer on the Linux workstations (by sign extending it). One way to do this is to use the function:

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

An example run of the simulator (not for the specified task of multiplication) is included at the end of this posting.

6 Assembly-Language Multiplication (20 pts.)

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 1 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 LC-2K word. Remember that shifting left by one bit is the same as adding the number to itself. Given the LC-2K instruction set, it's easiest to modify the algorithm so that you avoid the right shift. Submit a version of the program that computes $(32766 * 12328)$.

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 solution). To achieve this, you are strongly encouraged to consider using 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, Auto-Grading, and Formatting

We will grade primarily on functionality, including error handling, correctly assembling and simulating all instructions, input and output format, method of executing your program, correctly multiplying, and comprehensiveness of the test suites.

You must be careful to follow the exact formatting rules in the project description:

- (assembler) Follow exactly the format for inputting the assembly- language program and outputting the machine-code file.

- (assembler) Call `exit(1)` if you detect errors in the assembly-language program. Call `exit(0)` if you finish without detecting errors.
- (simulator) Don't modify `printState()` or `stateStruct` at all. Download this handout into your program electronically (don't re-type it) to avoid typos.
- (simulator) Call `printState()` exactly once before each instruction executes and once just before the simulator exits. Do not call `printState()` at any other time.
- (simulator) Don't print the sequence `@@@` anywhere (except where the provided `printState()` function prints it).
- (simulator) `state.numMemory` must be equal to the number of lines in the machine-code file.
- (simulator) Initialize all registers to 0.
- (multiplication) Store the result in register 1.
- (multiplication) The two input numbers must be in locations labeled `mcand` and `mplier` (lower-case).

8 Turning in the Project

Submit your files to `hconnect` using `git push` command. Here are the files you should submit for each project part:

1. Assembler

- C program for your assembler (name should end in ".c")
- suite of test cases (each test case is an assembly-language program in a separate file)

Example:

```
assemble.c test1.as test2.as test3.as
```

2. Simulator

- C program for your assembler (name should end in ".c")
- suite of test cases (each test case is an assembly-language program in a separate file)

Example:

```
simulate.c test1.as test2.as
```

3. Multiplication

- assembly program for multiplication (should be located in `./assembler`)

Example:

```
mult.as
```


Your assembler and simulator must each be in a single C file. We will compile your program on a Linux workstation using `gcc program.c -lm`, so your program should not require additional compiler flags or libraries.

The official time of submission for your project will be the time the last file is sent. If you send in anything after the due date, your project will be considered late (and will use up your late days). If you have already used up all of your late days, additional late submissions will not be scored for your project grade.

9 C Programming Tips

Here are a few programming tips for writing C programs to manipulate bits:

1. To indicate a hexadecimal constant in C, precede the number by `0x`. For example, 27 in 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. To find the value of the expression `a & b`, perform a logical AND on each bit of `a` and `b`. E.g., `25 & 11` is 9, since:

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

5. To find the value of the expression `a | b`, perform a logical OR on each bit of `a` and `b`. E.g., `25 | 11` is 27, since:

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

6. `~a` is the bit-wise complement 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.