

CMSC 212
Project #3
Due 03/27/2007 8:00 PM

Background

This assignment builds on what you learned about assembly language and the SPIN instruction set in assignment #1. In this assignment, you will write code to convert assembly language program into machine code (called an assembler). You will also build an interpreter to simulate the execution of instructions in the SPIN machine language.

The Simulated Computer

Recall from assignment #1, that the simulated computer has a memory that contains 2^{16} (65,536) words of memory, each 32 bits long.

In addition to memory, the computer has 16 registers that can be used to hold values. Two of the registers are special. R0 is hardwired to 0 and writing to it is legal, but doesn't change it, but reading from it returns 0. R1 is the "Program Counter" and always contains the address of the next instruction to execute. R1 can be read like a normal register, but can only be modified using special instructions (bal and beq), any other attempt to modify it is an Illegal Instruction (including using it as the register₁ value of bal) R2-R15 are General Purpose Registers, and can be read or written.

Description of instructions

lw <register₁> <register₂> <memory>

Copies the value stored in the memory location <memory> + <register₂> into the register location <register₁>. For example if register₂ is R0 and memory is 42, the data stored in location 42 is loaded into the register. However, if the second operand was R4 (rather than R4) and R4 contained 22, the values would be loaded from memory location $42 + 22 = 64$. (opcode 0)

li <register₁> <number>

Copies the supplied number <number> into the register location <register₁> (opcode 1). The number must be an unsigned integer in the range 0 to 65535 inclusive.

mv <register₁> <register₂>

Copies the value stored in <register₂> into <register₁> (opcode 2)

sw <register₁> <register₂> <memory>

Copies the value stored in <register₁> into memory location <memory> + the contents of <register₂> (opcode 3)

add <register₁> <register₂> <register₃>

Adds the value stored in <register₁> to the value stored in <register₂> and stores the result into <register₃> (opcode 4)

neg <register₁>

Negates the value stored in <register₁> (i.e. 1 becomes -1), (opcode 5)

beq <register₁> <register₂> <memory>

If the value stored in <register₁> is equal to the value stored in <register₂>, change the program counter (next instruction to execute) to execute the instruction stored in <memory> next. If they are not equal the instruction has no effect (opcode 6)

bal <register₁> <register₂> <memory>

Stores the current value of R1 (program counter) into <register₁>, Sets the value of R1 (program counter) to the contents of <register₂> plus the value of the <memory> field. (opcode 7)

read <register₁>

Read an integer from standard input, and store the value in <register₁> (opcode 8)

write <register₁>

Write the integer value stored in <register₁> to standard output. The instruction should print a newline after the integer.
(opcode 9)

halt

Terminates execution of the machine. No operands are used (opcode 10)

In addition, in the assembly language there is a special directive called `.data`. A `.data` directive is used to specify a memory location that contains information other than machine instructions (such as data values). The syntax of this directive is:

```
.data const1
```

Where `const1` is a 32 bit signed integer value to define what should be stored in that location.

Also, before any instruction or `.data` directive can be a label. A label is a symbolic name for that memory location. Labels may be up to 32 characters long and may contain any characters except a space or tab. However, a label may not start with a digit (0-9). Immediately after a label definition is a colon character (:). Here are two sample assembly instructions with labels defined.

```
loop1: add R3 R3 R4  
name: .data 1234567
```

Labels may be used anywhere an instruction takes a memory operand. The value of labels is that it make it easier to read and modify assembly code. For example, if you need to insert a new instruction between two instructions, if memory locations were specified numerically, it would be tedious to make all of the changes. Instructions may define their memory field using either a label or a numeric value. Here are some examples of instructions that use labels.

```
lw R4 name  
beq R3 R5 loop1
```

The Assignment

In this assignment, you will write two functions (and several helper functions).

```
int assemble(char *file, memoryLocation mem[]);
```

This function reads input from the file name passed as the first parameter. Each line consists of assembly code and which should be assembled into instructions in the passed computer memory array mem. The first instruction read from input should be placed at location 0, and the next at location 1, and so on. If the input contains any errors, the return value should be negative and the value of the negative result is the line the first error was encountered. For example, if the first error is at line 4, the return value is -4. If the program contains no errors, the return value should be the number of memory locations filled in.

When assembling instructions, the values of operands that are not used for a given instruction (i.e. operands 2 and 3 for the read instruction) should be set to zero.

Your assembler should check for valid assembly files. Here is a list of the types of errors you should be able to detect. This is not exhaustive, but should get you thinking about the types of errors that can occur in an assembly program.

- Labels that are used with an instruction, but not defined (i.e. label: missing).
- Invalid op codes
- Missing or Invalid operands
- Invalid Memory address specified (out of the range 0-65535).
- Invalid Register numbers specified

Your assembler needs to process any valid input which includes sequences of multiple spaces or tabs in the input to separate operands, labels, or before the operator. Each instruction will be one line. Any input line with more than 256 characters is invalid, and is considered an error.

```
int execute(memoryLocation mem[],
            int startingPC, bool traceFlag);
```

This function simulates the execution of the SPIN architecture. Memory of the computer to simulate is passed in mem. The first memory location to execute is passed as startingPC.

If the traceFlag is true, the simulate program should print out debugging output. The nature of this output is for you to design. Our test programs will only invoke this parameter with the false (no debugging output mode).

The program counter (R1) should be advanced prior to the execution of an instruction. For example, if you are running the instruction at memory location 1, while executing that instruction the PC should be 2.

If the program runs to normal completion (i.e. executes a halt instruction), the execute function should return the number of instructions executed. If an error was encountered (such as an illegal instruction), the execute function should return -1. Even though your assembler should check for many invalid inputs, the execute routine must also check for things like invalid instructions because a program could accidentally (or maliciously) branch to a word specified by a .data statement.

We will supply a main program for this assignment. We will also supply several header files. The file machine.h contains a definition of the machine instructions and memory, and the prototype for execution. The file assembler.h contains the prototypes for the function assemble that you will write. All of your code should be placed in the file named assembler.c and machine.c that we supply. You will obtain the files by typing the UNIX command line:

```
cp ~212files/p3.tar.gz .
```

Hints

To help you to split up each line of assembly code into its parts, we provided a routine called tokenize. This function will turn each group of non-white space characters into a “token”. Since each assembly instruction is made up of tokens, this will be helpful to split the input line into useful parts. The prototype for this function is:

```
int tokenize(char *line, char *token[], int tokenLimit);
```

The first parameter to this function is a line of input (with the new line removed). The second parameter is an array of character pointers to hold the tokens. The third parameter is the maximum number of tokens to use. The return value is the number of tokens found in the line. For example, if the input line is:

```
MyLabel: beq R5 R6 otherLabel
```

The function will return 5 and the passed array will contain the following elements:

```
token[0] = "MyLabel:"  
token[1] = "beq"  
token[2] = "R5"  
token[3] = "R6"  
token[4] = "otherLabel"
```

Note, that tokenize will not test for valid instructions, labels, or the correct number of tokens on a line. That is what your function assemble should do.

You will need to keep a label table to know the memory address where each label was defined. Think about how you will handle the case when labels are defined after they are used (i.e. the first operand to use a label is on a line before the line that defines the label). The easiest way to handle this case is to keep a table of labels and for each label store the memory word where it is defined as well as the memory word(s) where it is used. After you have processed the entire input file, you can go back (called back patching) and fill out the address field of all uses of each label with the appropriate value.