

CMSC 212

Project #1A & B

Part A Due 02/10/2005 8:00 PM

Part B Due 02/24/2005 8:00 PM

Background

Programs run on computers by having the hardware (or system software) execute basic operations called instructions. Many languages (such as Java) represent the program to be executed as byte codes which are very similar to machine instructions. In this assignment, you will build an interpreter for a simple machine language.

At their lowest level, computers operate by manipulating information stored in registers and memory. Registers and memory are like variables in C or Java; in fact inside the computer that is how variables get stored. In addition to data, memory also stores the instructions to execute. The basic operation of a computer is to read an instruction from memory, execute it and then move to the instruction stored in the next memory location. Typical instructions will read one or more values (called operands) from memory and produce a result into another register or memory location. For example, an instruction might add the values stored in two registers, R3 and R4 and then store the result in the register R5. Other instructions might just move data from one place to another (between registers or between a register and a memory location). A final type of instruction, called a branch instruction, is used to change what instruction is executed next (to allow executing if and looping statements).

This assignment will be done in two parts. You will need to turn in each part by its deadline.

The Simulated Computer

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 (Branch and Bnn), any other attempt to modify it is an Illegal Instruction (including as the register₁ value of Branch) R2-R15 are General Purpose Registers, and can be read or written.

Description of instructions

Load <register₁> <memory>

Copies the value stored in the memory location <memory> into the register location <register₁> (opcode 1)

Load <register₁> #<number>

Copies the supplied number #<number> into the register location <register₁> (opcode 1)

Move <register₁> <register₂>

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

Store <register₁> <memory>

Copies the value stored in <register₁> into memory location <memory> (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)

Halt

Terminates execution of the machine. (opcode 5)

Negate <register₁>

Negates the value stored in <register₁> (i.e. 1 becomes -1), (opcode 6)

Branch <register₁> <register₂> <memory>

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

Bnn <register₁> <memory>

If the value stored in <register₁> is non-negative (i.e. ≥ 0), change the program counter (next instruction to execute) to execute the instruction stored in <memory> next (opcode 8)

Input <register₁>

Read an integer from standard input, and store the value in <register₁> (opcode 10)

Output <register₁>

Write the integer value stored in <register₁> to standard output (opcode 11)

Instruction Format

In the computer, instructions are stored in memory locations just like data. However, each bit of the memory location is used to identify different parts of the instruction. The first 4 bits indicate which instruction is stored (called an opcode). For example, a load instruction is op code 1 so 0001_2 would be stored in the 4 bits of the opcode. The next three parts of the instruction store the number of the registers to be used. For example, if

a load instruction was trying to load something into R5, it would have 0101_2 stored in Register₁ field. The final part of the instruction is the Memory location field. This field is 16 bits long and can describe any of the 65,536 memory locations in the computer. Not all instructions use all of the fields. For example, the load instruction does not use Register₂ or Register₃. The following figure shows the layout of a memory location storing an instruction.

Purpose	Opcode	Register ₁	Register ₂	Register ₃	Memory
Size (bits)	4	4	4	4	16

The Assignment – Part A

In the first part of this assignment, you will write a function that, when passed an instruction, will print out the instruction in a format that is easier for humans to read. You will use the switch statement to figure out what instruction to print and you will use the printf function to do the printing. You should not print a newline character. The prototype of the function is:

```
void printInsn(instruction insn);
```

The output format should look like the definitions of the instructions given above. So for example if the instruction contained $0001\ 0011\ 0000\ 0000\ 0000\ 0000\ 0000\ 1110_2$, you would print Load R3 14. Notice there is exactly one space and no comma or other punctuation between the instruction and operands. If the instruction is invalid for any reason, print “Invalid Instruction”.

You will also write a second function that will take the memory as a parameter and print out the instructions from 0 up to passed limit of Limit. The prototype of this function is:

```
void disassemble(memoryLocation mem[], int limit);
```

The disassemble function will for print the address of each memory location followed by a “:” and then the output of your printInsn function. So if the sample instruction above is in memory location 0, the first line of output for the function would be:

```
0: Load R3 14
```

As you may have noticed, there are two variations of the load instruction. This instruction uses the Register₂ field to identify which form of the instruction is used. If the value in the Register₂ field is 0, the instruction is of the first form (the memory field contains an address of a memory location to load). If the field contains a 1, the memory field contains a number that should be loaded into the register. Any other value of Register₂ is and Invalid Instruction.

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

```
cvs co cs212/pla
```

The Assignment – Part B

In the second part of this assignment, you will write a function to “execute” a program stored in the computer, and write the assembler. An assembler is a tool that reads program instructions in a format similar to what you produced for part A and converts them into instructions stored memory locations.

The first part of the assignment is to write an interpreter that executes each instruction in the program. The prototype of the function to do this is:

```
int execute(memoryLocation memory[],
           int startingPC, int traceOn);
```

The first parameter is the memory of the computer. The second parameter is first address to execute, and the third parameter is a flag to indicate if the machine should produce debugging output. If the program executes a Halt instruction, the function should return the number of instructions executed (including the halt instruction). If an error is detected (such as an invalid instruction), the function should return -1.

The second part of the assignment is to write an assembler. The prototype for this function is

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

The first parameter is the name of a file that contains a program to be loaded. The second parameter is the memory of the computer.

The syntax of the assembly file looks very similar to the output of part A of the assignment. However, one addition is the idea of symbolic labels. Symbolic labels may be used to identify an instruction in the program, and then another instruction may refer to this label. For example,

```
TOP:  Add R1, R2, R3
      Bnn R3, TOP
```

In this example, the second instruction would branch to the memory location of the first instruction when the value of R3 is non-negative. A colon always follows symbolic label definitions. The sample files provided with this assignment include additional example programs.

For this part of the assignment, you will create two new files. The file machine.c will contain the code for the function execute, and the file assembler.c will contain the code for the function assemble (and any helper functions you decide to write).

Hints

To read the file in for the assemble function, you can call the routine getFile we provide. This function will read file named in the first parameter into the two dimensional array of characters in the second parameter. Each element of the outer array is a line in the original file.