

**CMSC 212**  
**Project #1**  
Due 9/15/2005 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).

## **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 Invalid Instruction (including as the register<sub>1</sub> value of Branch) R2-R15 are General Purpose Registers, and can be read or written.

## ***Description of instructions***

### **Load <register<sub>1</sub>> <memory>**

Copies the value stored in the memory location <memory> into the register location <register<sub>1</sub>> (opcode 1, register<sub>2</sub> is "0")

### **Load <register<sub>1</sub>> #<number>**

Copies the supplied number #<number> into the register location <register<sub>1</sub>>. For example: "Load R3 #44" copies 44 into R3. (opcode 1, register<sub>2</sub> is "1")

Note that for load instructions, *any value other than 0 or 1 in register<sub>2</sub> makes the instruction invalid.*

**Move <register<sub>1</sub>> <register<sub>2</sub>>**

Copies the value stored in <register<sub>2</sub>> into <register<sub>1</sub>> (opcode 2)

**Store <register<sub>1</sub>> <memory>**

Copies the value stored in <register<sub>1</sub>> into memory location <memory> (opcode 3)

**Add <register<sub>1</sub>> <register<sub>2</sub>> <register<sub>3</sub>>**

Adds the value stored in <register<sub>1</sub>> to the value stored in <register<sub>2</sub>> and stores the result into <register<sub>3</sub>> (opcode 4)

**Halt**

Terminates execution of the machine. (opcode 5)

**Negate <register<sub>1</sub>>**

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

**Branch <register<sub>1</sub>> <register<sub>2</sub>> <memory>**

Stores the current value of R1 (program counter) into <register<sub>1</sub>>, Sets the value of R1 (program counter) to <register<sub>2</sub>> plus the value of the <memory> field. (opcode 7)

**Bnn <register<sub>1</sub>> <memory>**

If the value stored in <register<sub>1</sub>> is non-negative (i.e.  $\geq 0$ ), change the program counter (next instruction to execute) to execute the instruction stored in <memory> next (opcode 8)

**Input <register<sub>1</sub>>**

Read an integer from standard input, and store the value in <register<sub>1</sub>> (opcode 10)

**Output <register<sub>1</sub>>**

Write the integer value stored in <register<sub>1</sub>> 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, listed above after each instruction type). 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<sub>2</sub> stored in Register<sub>1</sub> 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 Register<sub>2</sub> or Register<sub>3</sub>. The following figure shows the layout of a memory location storing an instruction.

Purpose	Opcode	Register <sub>1</sub>	Register <sub>2</sub>	Register <sub>3</sub>	Memory
Size (bits)	4	4	4	4	16

## The Assignment

In the first part of this assignment, you will write a function that, when passed a structure describing an instruction, will print out the instruction in a format that is easier for humans to read.

- 1) Print out the instruction names as listed above.
- 2) Registers are printed as an 'R' followed by the register number.
- 3) Memory addresses are printed in decimal notation.
- 4) Print a single space between each opcode and operand.
- 5) Use a "switch" statement, based on the opcodes listed above, to figure out what instruction to print.
- 6) 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<sub>2</sub>, 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 (but not including) the passed limit of Limit. The prototype of this function is:

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

The disassemble function will print the address (in decimal notation) of each memory location followed by a ":", a single space, the output of your printInsn function, and then a newline. Note that *the disassemble() routine should call the printInsn() routine*. 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
```

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.

Obtain the files by copying p1.tar.gz from ~/212files (set up in lab 2).

Submit via “<http://submit.cs.umd.edu>” or the submit212 script (also set up in lab 2).