

CMSC 212
Project #3
Due 10/14/2005 Midnight

Background

This assignment follows on to project P1. In that project you wrote a disassembler. In this project you will write an *assembler*, and an *interpreter* for the described machine. Refer to the P1's description of the simulated computer and instruction set.

The Interpreter

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 (whatever is useful to *you*, our testing routines will not use it). 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 Assembler

The second part of the assignment is to write an assembler. An assembler is a tool that reads program instructions in a format similar to what you produced for P1 and converts them into instructions stored memory locations. 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 return value should be the number of words assembled (both instructions and data) for a file of valid assembly code. The value “-1” should be returned on error.

The syntax of the assembly file looks very similar to the output of P1. 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.

Place your new code for this assignment in `machine.c`, for the function `execute()`, and `assembler.c` for `assemble()` (and any helper functions you decide to write). `disassemble.c` will contain your disassembler from P1.

NOTE: Only place code in `assemble.c`, `disassemble.c`, or `machine.c`. No other files will be compiled. The one exception is that you may create new header files. However, do NOT modify *old* header files.

Additional details:

- 1) All registers and memory should be initialized to zero at start of a run.
- 2) Correctness will be graded by inspecting return values (correctly flagging invalid code by returning -1, for example), and the contents of memory after code has been run.
- 3) The amount of white space (spaces, tabs) between tokens (opcodes, registers, etc.) is variable.
- 4) Your code should check for extra tokens on the line that should not be there (e.g. “Add R2 R3 R4 R5” should be caught).
- 5) You must use **your** `disassemble.c` in this project. Do not use anyone else’s. If yours still does not work, work w/ the TA to get it going.

Hints

- 1) To read the file in for the `assemble()` function, you can call the routine

```
int getFile(char *fileName, char file[MEM_SIZE][80])
```

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.

- 2) The `strtok()` library call can parse lines with variable amounts of white space into *tokens*. Read the man page (“man strtok” at the command line) for details.
- 3) The example files we provide will *not* exercise every opcode and every error condition. Be sure to write plenty of test files before attempting to release-test.