16.216: ECE Application Programming
Fall 2013

Programming Assignment #9: Instruction Decoding and File I/O
Due **Monday, 12/2/13**, 11:59:59 PM

1. Introduction

In this assignment, you will work with files to handle input and output. Your program will simulate a very simple processor that is controlled using a 32-bit "instruction." The program will decode this instruction to determine what operation should be performed and what values should be used in the computation. Initial register values and instructions will be read from input files, and it prints its results to an output file.

2. Deliverables

Submit your source file directly to Dr. Geiger (Michael_Geiger@uml.edu) as an e-mail attachment. Ensure your source file name is ***prog9_decode.c***. You should submit only the .c file. Failure to meet this specification will reduce your grade, as described in the program grading guidelines.

Note: a "starter" version of the .c file, which essentially contains an outline of what your main program should look like as well as some variables to be used, is available on the website. You may use this file as a starting point for your own code.

3. Specifications

NOTE: See Section 5 for a full breakdown of instruction encoding, possible opcodes, and detailed information about the program inputs.

Variables: Your program should contain, at a minimum, the following variables:

- `unsigned int inst`: The current instruction

- `int regs[32]`: Array of register values. If you have an operation that uses R0, R5, and R22, you'll access regs[0], regs[5], and regs[22].

- `unsigned int opcode`: Instruction field indicating operation to be performed.

- `unsigned int dest`: Instruction field that holds destination register number.

- `unsigned int src1`: Instruction field with number of first source operand.

- `unsigned int src2`: Instruction field with number of second source operand.

- `unsigned int shamt`: Instruction field that holds shift amount; only used for left and right shift operations.

You will also need to add variables to allow you to handle two different input files—one binary file, one text file—and one output file. You may need additional variables to successfully complete the program.

Program outline: The program should perform the following operations:

- Prompt the user to enter the name of a binary input file. This file will contain 32 integer values—open the file and read these values into the `regs[]` array to provide the initial register values.
    - Sample binary files: `invals1.bin`, `invals2.bin`, and `invals3.bin`.
    - Note that these files, as well as the text input file(s), should be placed in the same directory as your source code or executable.
- Prompt the user to enter the names of two text files: one to be used for program input; the other to be used for program output. Open each of these files.
    - The input file will contain a series of "instructions"—32 bit unsigned integers in hexadecimal format.
    - Sample program files: `p1.txt`, `p2.txt`, and `p3.txt`.
- Set up a loop that will repeatedly read a single instruction from the program input file, stopping when it reaches the end of the file.
- For each instruction:
    - Decode the instruction into the appropriate fields. <u>Note: the code to perform the decoding is **not** provided in the starter file—an earlier version of the specification incorrectly specified that it was present.</u>
    - Perform the appropriate operation on the source values. See Section 5 for a list of operations and corresponding opcodes.
    - Print the following output to the open text output file:
        - <u>Line 1:</u> The instruction, printed in the form:
            ```
            INSTRUCTION <#>: <inst>
            ```
          The number printed indicates how many instructions have been read thus far.
        - <u>Line 2:</u> The registers and operation, in the form:
            ```
            <dest> = <src1> <op> <src2>
            ```
        - <u>Line 3:</u> The values used in calculation and the result, in the form:
            ```
            = <val1> <op> <val2> = <result>
            ```
          For example, the first instruction may generate the following output:
            ```
            INSTRUCTION 0: 0x04011000
            R0 = R1 + R2
               = 1 + 2 = 3
            ```
    - Store the instruction result in the appropriate `regs[]` array element.
        - Note: I suggest storing the result after printing the output, to ensure the output is correct. For example, if you have the instruction:
            ```
            R17 = R17 + R17
            ```
          You can only print the correct source values on the next line if you don't overwrite `regs[17]` until after you've printed the output.

Error checking: If your program cannot open any of the three files (binary input file, text input file, text output file), print an error message and repeat the prompt until a correct file name is entered.
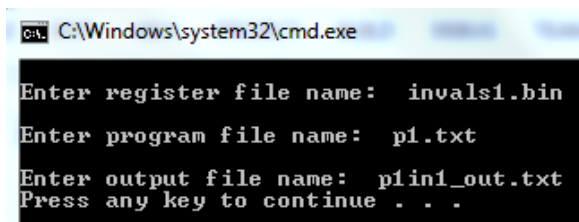
You may assume all input files are formatted correctly and therefore will not generate any errors.

Hints: Most hints for this program can be found in the following sources:
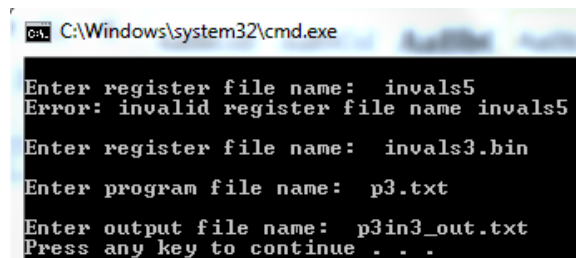
- The starter file, which contains an outline of the program.

- The lecture slides, code, and recording from Wednesday, 11/13, which dealt with file I/O.

- The lecture slides and recordings from Friday, 11/15 and Monday, 11/18, which dealt with bitwise operators. Pay particular attention to the topics at the end, which discuss how to isolate groups of bits from a larger value—those techniques will be particularly helpful for instruction decoding.

- The code presented on the course website, which shows an old assignment that provides further applications of bitwise operators.

4. Test Cases

The screenshots below show the console input and output from two different program runs—one in which all files open successfully, and another in which the user provides an invalid input file name:

The more useful test cases can be found on the web page, in the form of input and output files. Follow the "Program 9 files" link to find sample input and output files, the starter code file, and the example bitwise operator program mentioned above.

Your output files should match these test cases exactly for the given input values. I will use these test cases in grading of your lab, but may also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.

5. Additional Instruction Decoding Details

General description: When C programs are compiled, they are converted to *instructions*—simple operations that processors execute. Most instructions specify an operation to be performed and the data to be used in that operation.

Processors often store data in *registers*—temporary storage locations that are referenced by name or number in the instruction, as shown in the example below. This instruction adds the contents of registers 0 and 1 (the *source operands*) and stores the result in register 2 (the *destination operand*):

ADD R2, R0, R1

In practice, each instruction is encoded as a bit sequence; the processor *decodes* those bits to determine the operation and operands used for each instruction. Each possible operation is assigned a number, or *opcode*—for example, 0 might represent addition. Registers are usually referred to by number.

This program simulates a simple processor with nine operations, which are listed below. The "instruction" that you will input uses a total of 32 bits, as shown below (fields are described in Section 3):

31 26	25 21	20 16	15 11	10 6	5 0
opcode (6 bits)	dest (5 bits)	src1 (5 bits)	src2 (5 bits)	shamt (5 bits)	UNUSED

The example instruction above would be encoded as 0x04400800 =
0000 0100 0100 0000 1000 0000 0000$_2$:

- The first six bits (000001) indicate the operation (add)

- The next five bits indicate the destination register number ($00010_2 = 2 \rightarrow$ R2)

- The next five bits indicate the first source register number ($00000_2 = 0 \rightarrow$ R0)

- The next five bits indicate the second source register number ($00001_2 = 1 \rightarrow$ R1)

- The next five bits, which can hold a shift amount to be used in left and right shift operations, are unused.

- The last six bits are always unused and can be ignored.

<u>Opcode values and operations to be performed</u>. Note that most operations use two source registers, but the left and right shift operations (opcodes 5 and 6) use the shift amount field as the second operand:

Opcode	Operation	Example instruction
1	Add	0x04011000 R0 = R1 + R2
2	Subtract	0x08642800 R3 = R4 - R5
3	Multiply	0x0cc74000 R6 = R7 * R8
4	Divide	0x112a5800 R9 = R10 / R11
5	Left shift	0x158d0200 R12 = R13 << 8
6	Right shift	0x19cf00c0 R14 = R15 >> 3
7	Bitwise AND	0x1e119000 R16 = R17 & R18
8	Bitwise OR	0x2274a800 R19 = R20 | R21
9	Bitwise XOR	0x26d7c000 R22 = R23 ^ R24

<u>Contents of binary input files:</u> Values listed should be loaded directly into `regs[]` array; if read correctly, each element of `regs[]` will start with the value shown:

	invals1.bin	invals2.bin	invals3.bin
regs[0]	0	-65536	5
regs[1]	1	-32768	1785
regs[2]	2	-16384	-333
regs[3]	3	-8192	117
regs[4]	4	-4096	-906
regs[5]	5	-2048	10000
regs[6]	6	-1024	-1978
regs[7]	7	-512	26
regs[8]	8	-256	12
regs[9]	9	-128	-24
regs[10]	10	-64	2007
regs[11]	11	-32	2111
regs[12]	12	-16	-2112
regs[13]	13	-8	718
regs[14]	14	-4	-3
regs[15]	15	-2	-88
regs[16]	16	-1	15
regs[17]	17	0	3
regs[18]	18	1	-916
regs[19]	19	2	-12345
regs[20]	20	4	1000000
regs[21]	21	8	2552
regs[22]	22	16	13
regs[23]	23	32	14
regs[24]	24	64	86
regs[25]	25	128	17
regs[26]	26	256	119
regs[27]	27	512	-890
regs[28]	28	1024	-5152
regs[29]	29	2048	33
regs[30]	30	4096	16
regs[31]	31	8192	8