ECS 40 **Program #1 (50 pts)** **Winter 2016**

Due: Wednesday, January 27th, 11:59 PM. Executable Name: CPU.out

Filenames (case sensitive): registers.c, registers.h, decoder.c, decoder.h, reader.c, reader.h, instruction.h, main.h, main.c, Makefile, and authors.csv.

Format of authors.csv: author1_email,author1_last_name,author1_first_name
author2_email,author2_last_name,author2_first_name
For example: simpson@ucdavis.edu,Simpson,Homer
potter@ucdavis.edu,Potter,Harry

This is the first of several programming assignments in which you will be implementing a CPU simulator that relies on the x86 Intel core for its design. There are many parts to an Intel CPU, among them are: memory cache, decoder, control unit, registers, arithmetic/logic unit (ALU), and floating point unit (FPU). By the middle of February, your program will simulate many of these parts. For this first assignment, you will create Instruction, Reader, Decoder, and Registers typedef structs. Program #3 will have you convert your C program to C++, and add additional functionality.

This first program will read in an assembly language file created using the g++ -S option. The lines in the file will be read into a struct Reader that assigns consecutive simulated memory locations to each line of the instructions, starting with address 100. The main() will then start a traditional fetch-decode-execute loop that will "execute" the assembly language file by displaying the contents of the registers after each assembly language instruction has been "executed".

Further specifications:

1. Registers typedef struct contains an array of four ints named regs.
   1.1. At the heart of an actual Pentium are sixteen 32-bit (int) registers that are used to contain addresses and data that must be manipulated by the CPU. These registers, combined with the ALU and FPU, do all of the "thinking" of the CPU. There are four types of registers: general purpose, segment, index, and status/control.
   1.2. For this assignment, your Register struct will only have one general purpose register (eax), two index registers (ebp, esp), and one status/control register (eip).
   1.3. I have declared an enum in main.h to define the four register names so they may be used to access the array.
   1.4. The eax register is the accumulator, and is used for all arithmetic/logic instructions.
   1.5. The ebp register is the base pointer, and serves to hold addresses in simulated memory.
   1.6. The esp register is the stack pointer, and serves to hold the current address in simulated memory of the bottom of the stack. As items are added to the stack, esp is decremented, so the stack grows down into lower addresses.
   1.7. The eip register is the instruction pointer, and contains the address of the next instruction to be fetched-decoded-executed. This will be automatically incremented by four after fetching the current instruction.
2. Directives begin with a period.
   2.1. Directives affect the way machine code is generated. For this assignment directives will have no effect, and should be ignored.
3. Assembly Language Instructions
   3.1. Each assembly language instruction is composed of an opcode string, followed by either a label, or one or two operands. If it is followed by two operands, then, in most cases, the first is the source, and the second is the destination. These two operands rely on addressing modes to determine their actual values.
   3.2. For this assignment, there are four addressing modes used.
      3.2.1. Immediate Addressing Mode is indicated by a "$" followed by a number, e.g. $8 mean the value 8.
      3.2.2. Direct Addressing Mode is indicated by a "%" followed by a register, e.g. %eax , and means to use the value stored in that register if it is the source register, or place the value in that register if it is a destination register.
      3.2.3. The last two addressing modes assume that an address in memory is stored in a register--think C pointer.
         3.2.3.1. Indirect Addressing Mode is indicated by a "(% reg)", e.g. (%ebp), and means that the register contains the address in memory.
         3.2.3.2. Indirect Indexed Addressing Mode is indicated by a number followed by "(% reg)", e.g. 4(%ebp) means to add four to the address stored in ebp to get the final address.
         3.2.3.3. If the register is the source (first operand), then value stored in that final address is read. If the register is the destination (second operand), then then a value will be written to that final address.
   3.3. Instructions used in Program #1 files:
      3.3.1. The "l" at the end of most opcodes stands for long, i.e. 32 bits.

3.3.2. **addl** *operand1, operand2* : adds *operand1* and *operand2*, and puts the result in *operand2*.
3.3.3. **andl** *operand1, operand2* : bitwise ANDs *operand1* with *operand2*, and puts the result in *operand2*.
3.3.4. **leave** : copies the value in the ebp to esp, then copies the value in the memory specified by esp to ebp, and adds four to esp.
3.3.5. **movl** *operand1, operand2* : copies the information from *operand1* into *operand2*.
3.3.6. **pushl** *operand1* : subtracts four (size of a long) from esp, and then places the *operand1* information at the location specified by the esp. This means that the stack actually grows down into lower addresses!
3.3.7. **ret :** copies the value in the stack specified by esp to the eip and adds four to esp.
3.3.8. **subl** *operand1, operand2* : subtracts *operand1* from *operand2*, and puts the result in *operand2*.

4. Instruction typedef struct has an int named address, and a char* named info.
   4.1. info will hold the address of a dynamically allocated array of char sized to fit the information stored.
5. Reader typedef struct holds the information read from the file. It stores each line along with its simulated memory address.
   5.1. Reader has an array of 1000 Instruction named "lines" that holds each line of the file that takes up simulated memory. Only assembly language instructions use simulated memory space.
      5.1.1. As you read in each line of the file, you should allocate room for it in the lines[pos].info array, and store its simulated memory address. The program starts at address 100
         5.1.1.1. All assembly instructions require four bytes of simulated memory.
6. Decoder typedef struct has one char[20] named opcode, and two int* operand1, and operand2. These will be filled with their appropriate values as you decode each Instruction in the lines array of Reader.
7. Design of the program. All structs should be passed as pointers. Your main() should do the following.
   7.1. Declare a Reader, Decoder, and Registers with the names reader, decoder, and registers.
   7.2. Declare int memory[1001] that will be used to hold the stack. Note that really this should be an array of chars, but it simplifies matters to use ints for now. Thus, memory[992] will hold the int that would actually be stored in memory[992] to memory[995] in real RAM.
   7.3. Call a function to initialize registers.regs[esp] to 1000, registers.regs[ebp] to 996, registers.regs[eip] to 100, registers.regs[eax] to zero, memory[1000] to zero.
   7.4. Call a function to read the file into reader. The name of the file will be passed as the command line parameter.
      7.4.1. Note that many of the lines in the file start with a tab character, and use tabs to separate fields. You should replace these tabs with spaces. You should also remove all '\n' from the lines.
      7.4.2. Ignore all directive lines, and the "main:" line.
   7.5. In a loop do the following
      7.5.1. Call a function to fetch the Instruction from reader that has the address matching the value in eip. This function should also add four to the eip.
      7.5.2. Call a function to parse the current Instruction into decoder. I found strtok() quite handy for parsing.
         7.5.2.1. I have written a function, address(), in main.c, that has as parameters of an operand string, memory, and registers. It will return an int pointer, which will be the final address of the operand based on its addressing mode. This pointer could be the address of one of the registers in the registers variable, or an address in memory. By declaring a static int in address(), it can supply the address of the static int when immediate addressing mode is used (after copying the value of the operand into the static int).
      7.5.3. Call a function to execute the decoder. This function will call functions named for each of the possible assembly instructions, e.g. addl(), pushl() that will manipulate the registers and memory.
      7.5.4. Call a function to print out the decoder opcode and the contents of all of the registers. Your format of this must match mine.
      7.5.5. If the eip becomes zero, then leave the loop. This is why you initialize memory[1000] to zero.
8. Miscellaneous
   8.1. You should place the functions in their appropriate .c files based on the primary struct used. Provide the prototypes and typedef structs in the matching header files. Note that there is not an instruction.c because that struct is not the primary struct for any function.
   8.2. You may assume that all input will be valid, and not require any form of range checking.
   8.3. Your Makefile file must contain four pairs of compiling lines, and one pair of linking lines. You must use g++ with the –g –Wall –ansi options for compiling and linking. You will lose one point for each warning.
   8.4. You will find test1.c, test1.s, main.c, main.h and my own executable in ~ssdavis/40/p1.

```
[ssdavis@lect1 p1]$ cat test1.c
#include <stdio.h>
int main()
{
  int a, b, c;
  a = 7;
  b = 15;
  c = a + b;
  return c;
}
[ssdavis@lect1 p1]$ g++ -S test1.c
[ssdavis@lect1 p1]$ cat test1.s
        .file   "test1.c"
        .text
        .align 2
.globl main
        .type   main,@function
main:
.LFB2:
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
        subl    $24, %esp
.LCFI2:
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $7, -4(%ebp)
        movl    $15, -8(%ebp)
        movl    -8(%ebp), %eax
        addl    -4(%ebp), %eax
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        leave
        ret
.LFE2:
.Lfe1:
        .size   main,.Lfe1-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.2.3 20030502 (Red Hat Linux 3.2.3-49)"
[ssdavis@lect1 p1]$ CPU.out test1.s
pushl %ebp           eip: 104 eax:   0 ebp: 996 esp: 996
movl %esp, %ebp      eip: 108 eax:   0 ebp: 996 esp: 996
subl $24, %esp       eip: 112 eax:   0 ebp: 996 esp: 972
andl $-16, %esp      eip: 116 eax:   0 ebp: 996 esp: 960
movl $0, %eax        eip: 120 eax:   0 ebp: 996 esp: 960
subl %eax, %esp      eip: 124 eax:   0 ebp: 996 esp: 960
movl $7, -4(%ebp)    eip: 128 eax:   0 ebp: 996 esp: 960
movl $15, -8(%ebp)   eip: 132 eax:   0 ebp: 996 esp: 960
movl -8(%ebp), %eax  eip: 136 eax:  15 ebp: 996 esp: 960
addl -4(%ebp), %eax  eip: 140 eax:  22 ebp: 996 esp: 960
movl %eax, -12(%ebp) eip: 144 eax:  22 ebp: 996 esp: 960
movl -12(%ebp), %eax eip: 148 eax:  22 ebp: 996 esp: 960
leave                eip: 152 eax:  22 ebp: 996 esp: 1000
ret                  eip:   0 eax:  22 ebp: 996 esp: 1004
[ssdavis@lect1 p1]$
```