Computer Architecture, Spring 1396 , Project 1
Due 1:00 P.M., 1396/02/10

The goal of this project is to help you understand the functionality of assemblers and how they are constructed. In this project you are asked to write an assembler for a simple computer called Miniature. Followings are the characteristics of Miniature:

a: Miniature is a 32 bit computer (i.e., each word is 4 bytes long).

b: It has 16 32-bit registers, R0 to R15, R0 is always zero, and R1 is the stack pointer.

c: Each addressable unit is a word (viz., each instruction is a word, and therefore PC + 1 points to the next instruction in the instruction stream).

d: Miniature has 65536 words of memory.

e: It has 3 instruction formats and 15 instructions that are

R-type: Instructions: ***add***, ***sub***, ***slt***, ***or***, and ***nand***

```
bits 31-28 unused all zero
bits 27-24 opcode
bits 23-20 rs (source register)
bits 19-16 rt (target register)
bits 15-12 rd (destination register)
bits 11-0  unused (all zero)
```

I-type: Instructions: ***addi***, ***ori***, ***slti***, ***lui***, ***lw***, ***sw***, ***beq*** and ***jalr***

```
bits 31-28 unused all zero
bits 27-24 opcode
bits 23-20 rs (source register)
bits 19-16 rt (target register)
bits 15-0  offset
```

J-type: Instructions: ***j*** and ***halt***

```
bits 31-28 unused all zero
bits 27-24 opcode
bits 23-16 unused and they should be all zero
bits 15-0  target address
```

Note that $rs field of ***lui*** instruction and offset field of ***jalr*** instruction are zero, so as the target address for ***halt*** instruction.

Description of Miniature instructions is given in table 1. Figure 1 shows the memory layout for Miniature.

What you need to do for this project

1. Write an assembler in C# for Miniature. Your assembler should translate assembly-language names for instructions, such as beq, into their numeric equivalent (e.g. 1011), and it will also translate symbolic names for addresses into numeric values. The final output will be a series of 32-bit instructions.
   The format for a line of assembly code is (<white> means a series of tabs and/or spaces):

   ```
   label<white>instruction<white>field0,field1,field2<white>#comments
   ```

   The leftmost field on a line is the label field. Valid labels contain a maximum of 6 characters and can consist of letters and numbers (but must start with a letter). The label is optional (the white space following the label field is required). Labels make it much easier to write assembly-language programs, since otherwise you would need to modify all address fields each time you added a line to your assembly-language program! After the optional label is white space. Then follows the instruction field, where the instruction can be any of the assembly-language instruction names listed in the above table. After more white space there a series of fields. Each register is specified by its number. For example, 3 refers to the 4^{th} register of the CPU.

```
mnemonic               opcode   Description
add  $rd,$rs,$rt       0000     $rd <- $rs + $rt, PC <- PC + 1
sub  $rd,$rs,$rt       0001     $rd <- $rs - $rt, PC <- PC + 1
slt  $rd,$rs,$rt       0010     if ($rs < $rt)  $rd <- 1
                                otherwise       $rd <- 0
                                , PC <- PC + 1
or   $rd,$rs,$rt       0011     $rd <- $rs | $rt, PC <- PC + 1
nand $rd,$rs,$rt       0100     $rd <- $rs & $rt, PC <- PC + 1
addi $rt,$rs,imm       0101     $rt <- $rs + SE(imm), PC <- PC +1
slti $rt,$rs,imm       0110     if ($rs < SE(imm) $rt <- 1
                                otherwise         $rt <- 0
                                , PC <- PC + 1
ori  $rt,$rs,imm       0111     $rt <- $rs | ZE(imm), PC <- PC + 1
lui  $rt,imm           1000     $rt <- imm << 16, PC <- PC + 1
lw   $rt,$rs,offset    1001     $rt <- Mem($rs + SE(offset)), PC <- PC + 1
sw   $rt,$rs,offset    1010     Mem($rs + SE(offset)) <- $rt, PC <- PC + 1
beq  $rt,$rs,offset    1011     if ($rs == $rt) PC <- PC + 1 + SE(offset)
                                else PC <- PC + 1
jalr $rt,$rs           1100     $rt <- PC + 1, PC <- $rs
j    offset            1101     PC <- ZE(offset)
halt                   1110     PC <- PC + 1, then halt the machine
```

Table 1: Miniature Instruction Description

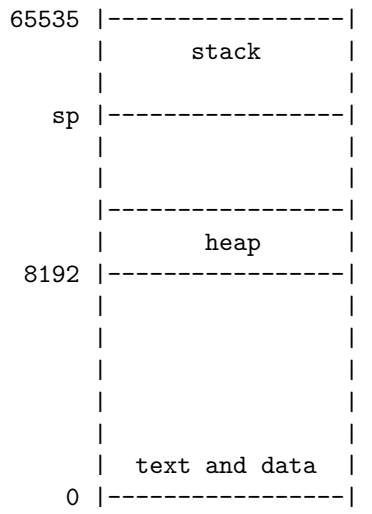

Figure 1: Layout of Memory

The number of fields depends on the instruction, and unused fields should be ignored. For example: R-type instructions require 3 fields: field0 is $rd, field1 is $rs, and field2 is $rt. Symbolic addresses refer to labels. For lw or sw instructions, the assembler should compute offset to be equal to the address of the label. This could be used with a zero base register to refer to the label, or could be used with a non-zero base register to index into an array starting at the label. For beq instructions, the assembler should translate the label into the numeric offset needed to branch to that label.

After the last used field, more white spaces appear and then any comments which should follow a #. The comment field ends at the end of a line. Comments are vital to creating understandable assembly-language programs, because the instructions themselves are rather cryptic.

In addition to Miniature instructions, an assembly-language program may contain directives for the assem-

bler. The only assembler directives we will use are .fill and .space (note the leading period). .fill tells the assembler to put a number into the place where the instruction would normally be stored. .space tells the assembler to fill memory with zero for the given size. .fill and .space instructions use one field, which can be either a numeric value or a symbolic address for .fill and it is the size of memory filled with zero for .space. For example, ".fill 32" puts the value 32 where the instruction would normally be stored. .fill with a symbolic address will store the address of the label. In the example below, ".fill start" will store the value 2, because the label "start" is at address 2; ".space 10" clears 10 words of memory from where the instruction would be stored.
The assembler should make two passes over the assembly-language program. In the first pass, it will calculate the address for every symbolic label. Assume that the first instruction is at address 0. In the second pass, it will generate a machine-language instruction (in decimal) for each line of assembly language. For example, here is an assembly-language program (that counts down from 5, stopping when it hits 0).

```
        lw      1,0,five  #  load reg1 with 5 (symbolic address)
        lw      2,1,2     #  load reg2 with -1 (numeric address)
start   add     1,1,2     #  decrement reg1
        beq     0,1,done  #  goto end of program when reg1==0
        j       start     #  go back to the beginning of the loop
done    halt              #  end of program
five    .fill   5
neg1    .fill   -1
stAddr  .fill   start     #  will contain the address of start (2)
```

And here is the corresponding machine language:

```
(address 0): 151060486 (hex 0x09010006)
(address 1): 152174594 (hex 0x09120002)
(address 2): 1183744 (hex 0x0121000)
(address 3): 184614913 (hex 0x0b010001)
(address 4): 218103810 (hex 0x0d000002)
(address 5): 234881024 (hex 0x0e000000)
(address 6): 5 (hex 0x5)
(address 7): -1 (hex 0xffffffff)
(address 8): 2 (hex 0x2)
```

Be sure you understand how the above assembly-language program got translated to machine language.

Since your programs will always start at address 0, your program should only output the contents, not the addresses.

```
151060486
152174594
1183744
184614913
218103810
234881024
5
-1
2
```

2. ***Running Your Assembler***
Write your program to take two command-line arguments. The first argument is the file name where the assembly-language program is stored, and the second argument is the file name where the output (the machine-code) is written. For example, with a program name of "assemble", an assembly-language program in "program.as", the following would generate a machine-code file "program.mc":

```
assemble program.as program.mc
```

Note that the format for running the command must use command-line arguments for the file names (rather than standard input and standard output). Your program should store only the list of decimal numbers in the machine-code file, one instruction per line. Any deviation from this format (e.g. extra spaces or empty lines) will render your machine-code file ungradeable. Any other output that you want the program to generate (e.g. debugging output) can be printed to standard output.

3. ***Error Checking***
   Your assembler should catch the following errors in the assembly-language program: use of undefined labels, duplicate labels, offset that doesn't fit in 16 bits, and unrecognized opcodes. Your assembler should exit(1) if it detects an error and exit(0) if it finishes without detecting any errors. Your assembler should NOT catch simulation-time errors, i.e. errors that would occur at the time the assembly-language program executes (e.g. branching to address -1, infinite loops, etc.).

4. ***Test Cases***
   An integral (and graded) part of writing your assembler will be to write a suite of test cases to validate any Miniature assembler. This is a common practice in the real world–software companies, they maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of the project specification and your program, and it will help you a lot as you debug your program.

   The test cases for the assembler will be short assembly-language programs that serve as input to an assembler. You will submit your suite of test cases together with your assembler, and we will grade your test suite according to how thoroughly it exercises an assembler. Each test case may be at most 50 lines long, and there may be 20 test suites. These limits are much larger than needed for full credit (the solution test suite is composed of 5 test cases, each < 10 lines long).

   Hints: the example assembly-language program above is a good case to include in your test suite, though you'll need to write more test cases to get full credit. Remember to create some test cases that test the ability of an assembler to check for the errors.

5. ***Assembler Hints***
   Since offset is a 2's complement number, it can only store numbers ranging from -32768 to 32767. For symbolic addresses, your assembler will compute offset so that the instruction refers to the correct label.

   Remember that offset is only an 16-bit 2's complement number. Since Intel integers are 32 bits, you'll have to chop off all but the lowest 16 bits for negative values of offset.