Project 1: MIPS Assembler

CS2506: Introduction to Computer Organization II
Fall 2011
Due Date: 10/31/2011 (Monday, midnight)

300 points

In this project, you will implement an assembler for a subset of the MIPS instruction set. The assembler should be implemented in the C programming language on Linux.

Recall that an assembler translates code written in mnemonic form in assembly language into machine code. Your assembler will take a file written in assembly language as input on the **command line** and should produce an output file containing the MIPS machine code. The input file will be in ASCII text and look similar to the MIPS assembly programming examples that you solved in homework 2. Each line in the input assembly file contains either a mnemonic, a section header (such as .data) or a label (jump or branch target. The maximum length of a line is 256 bytes. Your input file can also contain comments. Any text after a # symbol is a comment and can be ignored. Section headers such as .data and .text will be in a line by themselves with no other assembly mnemonic. Similarly, branch targets such as loop: will be on a line by themselves with no other assembly mnemonic. The input assembly file will only contain one data section and one text section. The first section in the file will be the text section, followed by the data section. Your assembler would be invoked as:

```
assembler <input file> <output file>
e.g. ./assembler add.asm add.txt
```

For example you input file could like:

```
# The following program initializes an array of 10 elements and computes
# a running sum of all elements in the array. The program prints the sum
# of all the entries in the array.

.text
      la $a0, array      # load address of array
      la $a1, array_size # load address of array_size
      lw $a1, 0($a1)     # load value of array_size variable
loop:
      sll  $t1, $t0, 2     # t1 = (i * 4)
      add $t2, $a0, $t1    # t2 contains address of array[i]
      sw $t0, 0($t2)       # array[i] = i
      addi $t0, $t0, 1     # i = i+1
      add $t4, $t4, $t0    # sum($t4) = ($t4 + array[i])
      slt $t3, $t0, $a1    # $t3 = ( i < array_size)
      bne $t3, $zero, loop # if ( i < array_size ) then loop
      la $a0, message     # print sum message
      add $a0, $t4, $zero #load value to print into argument register $a0
      nop                 # done.
```

```
.data
     array: .word  0:10            # array of 10 words
     array_size: .word 10          # size of array
     message: .asciiz "The sum of numbers in array is: "
```

The subset of the MIPS assembly language that you need to implement includes the following:

Data Types in the .data section

.**word:** This is used to hold a one or more 32 bit quantities and initialized the given value. For example,

```
var1: .word 15      # this declaration creates one 32 bit integer called
                    # var1 and initializes it to 15.

array1: .word 2:10  # this declaration creates an array of 10 32 bit
                     # integers and initiazes each element of the array
                     # to the value 2.
```

.**asciiz :** This is used to hold a NULL (0) terminated ascii string. For example,

```
hello_w: .asciiz "hello world"   # this declaration creates a NULL
                                 # terminated string with 12 characters
                                 # including the terminating 0 character
```

Instructions in the .text section

Your assembler needs to support the following instructions in the .text section

Load and Store Instructions

la: this is a **pseudo** instruction that loads the address of a variable into a register. For example

```
la $t0, var1
```

would copy the RAM address of var1 (presumably a variable defined in the .data section) into register $t0. The la pseudo instruction should be translated into two instructions:

```
lui $t0 var1[31-16]
ori $t0 $t0, var1[0-15]
```

The fields var1[31-16] and var1[15-0] represent the upper 16 bits and the lower 16 bits of the var1's address respectively.

lw: This loads a word at the memory address specified by a register and a constant offset. For example

```
lw    $t2, 4($t0)
```

would load the word at ram address $t0 + 4 into $t2.

sw: This stores the word in a register into a memory address specified by a register and a constant offset. For example

```
sw    $t2, 4($t0)
```

would store the word in $t2 at the ram address $t0 + 4.

Note that the constant offset in the load and store instructions (lw and sw) may be positive or negative.

Arithmetic Instructions

add: Adds the contents of two registers and stores the result in a register. For example

```
add $t0 $t1 $t2
```

adds the contents of $t1 and $t2 and stores the result in $t0.

sub: Subtracts one register from another and stores the result in a third register. For example

```
sub $t0 $t1 $t2
```

subtracts the contents of $t2 from $t1 and stores the result in $t0.

addi: Adds a constant to a register and stores the result in a register. Note that the constant may be positive or negative. For example

```
addi $t0 $t1 -15
```

adds the constant -15 to the contents of $t1 and stores the result in $t0.

or: Performs a logical "or" of the contents of two registers and stores the result in a third register. For example

```
or $t0 $t1 $t2
```

performs a logical OR of registers $t1 and $t2 and stores the result in $t0.

and: Performs a logical "and" of the contents of two registers and stores the result in a third register. For example

```
and $t0 $t1 $t2
```

performs a logical AND of registers $t1 and $t2 and stores the result in $t0.

ori: Performs a logical "or" of a constant and a register and stores the result in another register. For example

```
ori $t0 $t1 25
```

performs a logical OR of the constant 25 and the contents of $t1 and stores the result in $t0.

andi: Performs a logical "and" of a constant and a register and stores the result in another register. For example

```
andi $t0 $t1 25
```

performs a logical AND of the constant 25 and the contents of $t1 and stores the result in $t0.

slt: Compares two registers and sets a register to 1 or 0 depending on the result of the comparison. For example

```
slt $t0 $t1 $t2
```

sets the register $t0 to 1 if the contents of $t1 are less than the contents of $t2 or sets $t0 to 0 otherwise.

slti: Compares a register and a signed constant and sets a register to 1 or 0 depending on the result of the comparison. For example

```
slti $t0 $t1 -15
```

sets the register $t0 to 1 if the contents of $t1 are less than the constant -15 or sets $t0 to 0 otherwise.

sll: shifts the contents of a register left by the specified number of bits. For example

```
sll $t0 $t1 3
```

logically shifts the contents of the register $t1 left by 3 bits.

srl: shifts the contents of a register right by the specified number of bits. For example

```
srl $t0 $t1 3
```

logically shifts the contents of the register $t1 right by 3 bits.

Control Instructions

beq: Compares two registers and branches to the specified label if the comparison is equal. For example

```
beq $t0 $t1 L1
```

Compares the contents of registers $t0 and $t1 and if they are equal, jumps to the branch target label L1.

j: unconditionally jumps to the target label. For example

```
j L1
```

Unconditionally jumps to the branch target label L1,

jr: unconditionally jumps to the address contained in the specified register. For example

```
jr $t0
```

Unconditionally jumps to the address contained in the register $t0.

jal: jumps to the branch target specified in the instruction and sets the register $ra to the program counter value of the next instruction. For example

```
jal print
```

Jumps to the branch target print and sets the register $ra to the program counter of the next instruction.

Output

Your assembler should resolve all references to branch targets in the .text section and variables in the .data section and convert the instructions in the .text section into machine code. To convert an instruction into machine code follow the instruction format rules specified in the class textbook. For each format – R format, I format or J format, you should determine the opcode that corresponds to instruction, the values for the register fields and any optional fields such as the function code and shift amount fields for arithmetic instructions (R Format) and immediate values for I format instructions. The output machine code should be saved to an output file specified in the command line. The output file should contain the machine code corresponding to instructions from the .text section followed by a blank line followed by variables from the .data section in human readable binary format i.e., (0s and 1s). For example to represent a decimal number of 40 in binary you would write 0000000000101000. Your output file should match with the machine file generated by the MARS simulator.

How do I verify my output or test my code?

In order to test your assembler you should use the MARS simulator. You will need to do the following steps in the MARS simulator.

1. First, ensure that MARS is configured to start the text section at address 0x00000000.

    i) Open MARS simulator and modify the memory configuration settings through **Settings->Memory Configuration** and select "**Compact, Text at Address 0**" and DO NOT modify any of the remaining addresses on the right. Click "**Apply and Close**" to exit memory configuration settings.

2. Now, in order to generate the actual machine file for the assembly program, you will need to dump the **binary text format** for the text and data sections.

    i) Generate machine code for .text section of your assembly program.

    Open the assembly program and select **File->Dump memory**. Select the "**.text (0x00000000 - 0x00000058)**" in the "**Memory Segment**" and select "**Binary Text**" for dump

format and click on "**dump to a file**" button and specify an output file (say *text_section_of_add_asm.txt*) to dump the machine code of the assembly program. Now you have the machine instructions for your text section of your assembly program.
Note that the actual end address of ".text" section will vary depending on your assembly program. For the sample input file "add.asm", attached with this project, you will find that the text section starts at 0x00000000 and extends up to 0x00000058. However, the end address may vary depending on your input assembly files.

ii) Generate machine code for your .data section of your assembly program.

a) Open the assembly program and select **File->Dump memory** from the MARS IDE.

b) Select the **".data (0x00002000 - 0x00002ffc)**" in the "**Memory Segment**" and select "**Binary Text**" for dump format and click on "**dump to a file**" button. Give an output file (say *data_section_of_add_asm.txt*) to dump the machine code of the assembly program. Now you have the machine instructions for your data section of your assembly program.

Note that the actual end address of ".data" section will vary depending on your assembly program. For the "add.asm" example above, you will find that data section starts at 0x00002000 and extends up to 0x00002ffc. Similar to the text section, the actual size of the data section depends on your input file.

iii) Copy the contents of your "text_section_of_machine_file.txt" and your "data_section_of_machinefile.txt" with a blank line in between them to distinguish the two sections into another file (say add_asm.txt). Now you have your output machine file ready. Your assembler program should produce an output identical to this machine file.

The sample machine file uploaded with this project on scholar is generated using the above approach. Note that the text section should always start at 0x00000000 and your data section should start at 0x00002000.

Extra Credit:
For 50 points of extra credit, augment your assembler to take more than one data section and more than one text section. This is equivalent of creating a single text section from the multiple text sections and a single data section from the multiple data sections.

Please turn in your programs electronically by uploading them to Scholar by midnight 10/31/2011

Happy debugging!