CSI 402 – Systems Programming – Spring 2012

Programming Assignment IV

Date given: Mar. 27, 2012 **Due date:** Apr. 13, 2012
Weightage: 10%

The deadline for this assignment is **11 PM, Friday, April 13, 2012**. With lateness penalty, the program will be accepted until **11 PM, Sunday, April 15, 2012**. The assignment *won't* be accepted after 11 PM, Sunday, April 15, 2012.

Very important: Your source program must consist of two or more C files and you must also have a `makefile`. The C files (with extension "`.c`"), header files (with extension "`.h`") and the `makefile` must be submitted together using the `turnin-csi402` command. Instructions for using `turnin-csi402` and additional specifications for the `makefile` will be included in the `README` file for this assignment.

You are required to implement an assembler for a hypothetical computer called TMIPS. Details regarding the architecture of TMIPS and its assembly language appear elsewhere in this handout. You may choose either a 1-pass design or a 2-pass design for the assembler.

The executable version of your program must be named `p4`. It will be executed using a command line of the following form:

`p4` *infile*

Here, the argument *infile* specifies the name of an input file containing a program written in TMIPS assembly language. Your program must assemble the specified input file. If the assembly language program does not contain any syntax errors, your program must produce two output files, one file containing the object code and the other file containing the symbol table. If there are syntax errors, your program must produce only an error file. For this program, the input and output files are all text files. The naming conventions for the output files and their formats are discussed below.

Naming conventions for output files: For each input file, the names of the output files must be generated as follows. The **root name** of the input file is obtained from the string corresponding to the name of the file by including all the characters up to but *not* including the *last* occurrence of the period (`'.'`); if the source file name does not contain a period, then the root name is the entire source file name. The name of the object code file is obtained by appending the string `".obj"` to the root name. The name of the symbol table file is obtained by appending the string `".sym"` to the root name. Similarly, the name of the error file is obtained by appending the string `".err"` to the root name.

Examples: Suppose the input file is named `prog1.part1.asm`. Here, `prog1.part1` represents the root name. So, the names of the object, symbol table and error files are `prog1.part1.obj`, `prog1.part1.sym` and `prog1.part1.err` respectively. As another example, if the name of the source file is `prog1`, the root name is `prog1` itself. Therefore, the names of the object, symbol table and error files are `prog1.obj`, `prog1.sym` and `prog1.err` respectively.

Format of source file: Each line of the input file has at most 80 characters, including the newline character. Each source line may be a blank line (i.e., a line containing only whitespace characters), a comment line (i.e., a line whose first character is `'#'`) or an assembly language instruction. The assembler should read and ignore the blank lines and comment lines. However, blank and comment lines must be counted for the purposes of generating the error file.

Each line containing an assembly language instruction has an optional label, an opcode, zero or more operands and an optional inline comment (starting with the `'#'` character). These fields are separated by one or more spaces. If a statement has a label, then the label will appear from the first character of the statement. Each label is terminated by a colon (`':'`). The operands themselves are separated by commas. The number and types of operands depends on the opcode. Details concerning the machine opcodes and the pseudo-opcodes (i.e., assembler directives) are given elsewhere in this handout.

Format of object file: Recall that the object file produced by the assembler has the extension `".obj"`. Each line of this file must have two hexadecimal integers separated by spaces or tabs. These hexadecimal integers represent respectively an address and the contents of that address. As a convention, the starting address for the assembly language program must be set to zero.

Format of symbol table file: Recall that the symbol table file produced by the assembler has the extension `".sym"`. Each line of this file has a symbol and its address (LC-value) in decimal; these two quantities must be separated by spaces or tabs.

Format of error file: Recall that the error file produced by the assembler has the extension `".err"`. The first part of this file is a source listing containing each line of the input file preceded by a line number. (The line numbers must start at 1.) Note that blank lines and comment lines must also be output with a line number. The second part of the file must contain the list of errors detected by the assembler. Each error message must indicate the line number where the error occurs and a brief error message. These messages must be in *increasing order* of line numbers. If a source line has multiple errors, it is sufficient to indicate any one of the errors. After listing all the error messages, the error file must contain the list of all multiply defined symbols (if any) and all undefined symbols (if any).

Assembler Conventions:

1. All integers appearing in the in the assembly language source program are in **decimal**. Register numbers are preceded by the `'$'` symbol.
2. For each source program, the object program generated starts at address 0 and instructions are placed in successive memory addresses.
3. The assembler must set all the bits corresponding to unused parts of an instruction to zero.
4. When the assembler assigns memory locations to process the `.resw` directive, those locations must be initialized to zero.

Errors to be Detected: Your program must detect the following syntax errors in the input.

(a) Undefined label.

(b) Multiply defined label.

(c) Illegal machine opcode.

The assembler should *not* stop upon detecting an error. It should continue to detect syntax errors in all the lines of the input file.

In addition to the above errors, your program must also check for the following fatal (command line) errors. In each case, the program must print an appropriate error message to `stderr` and stop.

1. Incorrect number of parameters on the command line.
2. The source file specified on the command line cannot be opened.

Information about `README` file: The `README` file for this assignment will will be available by 10 PM on Monday, April 2, 2012. The name of the file will be `prog4.README` and it will be in the directory `~csi402/public/prog4` on `itsunix.albany.edu`.

Details Regarding the Assembly Language for TMIPS

Each line of an assembly language program for TMIPS may be a blank line, a comment line or an assembly language instruction. A blank line contains only whitespace characters. Each comment line starts with the character `'#'`. An assembly language instruction consists of a sequence of fields in the following order: an optional label (terminated by a colon), an opcode, zero or more operands and an optional inline comment (starting with `'#'`). The number of operands depends on the opcode. The label (if any), the opcode and the operands (if any) are separated by one or more spaces. The operands themselves are separated by commas.

Each valid label starts with a lower case letter followed by zero or more lower/upper case letters or digits or the underscore character. Each valid label has at most 8 characters. If a statement has a label, then the label will start from the first position on that line.

The assembler assigns an address to each instruction. Addresses for instructions are generated successively starting from zero.

The opcode in an instruction may be one of the 36 machine opcodes for TMIPS (discussed elsewhere in this handout) or the following pseudo-opcodes (also called assembler **directives**). Note that each directive starts with a period (`'.'`).

(a) The `.text` directive: This directive indicates the beginning of the text segment (i.e., executable instructions). The directive does not have any operands. You may assume that in any source file, the `.text` directive appears at most once.

(b) The `.data` directive: This directive indicates the beginning of the data segment. You may assume that in any source file, the `.data` directive appears at most once. The directive does not have any operands. In any source file containing both text and data segments, you may assume that the data segment appears *after* the text segment. You may also assume that the data segment contains (in addition to comments and blank lines) only statements with `.resw` and `.word` directives.

(c) The `.resw` directive: The operand for this directive is a positive integer value that determines the number of words of storage needed at that point. Note that the assembler must initialize each of those words to zero.

(d) The `.word` directive: The operand for this directive is a pair of integers separated by a colon (`':'`). The first integer (which may be negative, zero or positive) indicates the initial value and the second integer (which must be positive) gives the number of words that must be initialized to that value.

TMIPS: A Hypothetical Computer

The hypothetical computer TMIPS (for "Tiny MIPS") to be used in this assignment has 65536 (decimal) memory locations, numbered 0 through 65535 (decimal). Thus, 16 bits are used to specify the address of a memory location. The smallest addressable memory unit is a word, and each word is 32 bits long. Thus, successive words have addresses 0, 1, 2, ..., 65535. The numbering of bits within a word is **little endian**. The only data type the machine supports is integer. Each integer is stored in one machine word (32 bits) using 2's complement representation.

The machine has 32 registers, numbered 0 through 31. Each register is 32 bits long. In the assembly language of TMIPS, the registers are denoted by `$0`, `$1`, ..., `$31`.

Each machine instruction for TMIPS is one word (32 bits) long. There are three instruction formats, namely R-format, I-format and J-format. In each instruction, six bits are used for the opcode. Since there are 32 registers, five bits are used to specify each register. Additional details regarding these formats are given below.

Instruction Formats:

(a) R-format:

6	5	5	5	5	6
Opcode	Rs1	Rs2	Rt	sa	Unused

This format is used for arithmetic instructions, bitwise instructions, input/output instructions as well as for the `jr` instruction. (Some instructions may ignore one or more fields in the instruction.) Here, `Rs1` and `Rs2` specify the two source registers and `Rt` specifies the target register. The field `sa` specifies the shift amount as a five bit unsigned integer for `sll`, `srl` and `sra` instructions. The least significant six bits are unused.

(b) I-format:

6	5	5	16
Opcode	Rs	Rt	Immediate Operand

This format is used for arithmetic operations involving immediate operands and for `lwb` and `swb` instructions. (The `lwb` and `swb` instructions use the base + displacement mode of addressing.) Here, `Rs` and `Rt` specify the source and target registers respectively. In I-format, the least significant 16-bits are treated as an integer in 2's complement form, with bit 15 as the sign bit.

Opcode	Mnemonic	Format	Opcode	Mnemonic	Format
0	`hlt`	R	18	`li`	I
1	`add`	R	19	`addi`	I
2	`sub`	R	20	`subi`	I
3	`mul`	R	21	`muli`	I
4	`div`	R	22	`divi`	I
5	`mod`	R	23	`modi`	I
6	`move`	R	24	`lwb`	I
7	`and`	R	25	`swb`	I
8	`or`	R	26	`lwa`	J
9	`xor`	R	27	`swa`	J
10	`com`	R	28	`j`	J
11	`sll`	R	29	`jal`	J
12	`srl`	R	30	`jeq`	J
13	`sra`	R	31	`jne`	J
14	`jr`	R	32	`jlt`	J
15	`rdr`	R	33	`jle`	J
16	`prr`	R	34	`jgt`	J
17	`prh`	R	35	`jge`	J

Table 1: Mnemonic opcodes and their decimal values for TMIPS

(c) J-format:

6	5	5	16
Opcode	Rs	Rt	Address

This format is used for jump instructions as well as for `lwa` and `swa` instructions. All of these instructions use the direct addressing mode. Here, `Rs` and `Rt` specify the source and target registers respectively. (Some instructions use only the target register.) In J-format, the least significant 16-bits are treated as an unsigned integer.

Notes: The list of mnemonic opcodes, their numerical values (in decimal) and their formats are given in Table 1 above. Note that opcodes 0 through 17 use the R format, 18 through 25 use the I format and 26 through 35 use the J format.

Additional Information about the Instructions:

(a) R-format Instructions: (Opcodes 0 through 17)

1. Opcode `hlt` has no operands.
2. Each of the opcodes `add`, `sub`, `mul`, `div`, `mod`, `and`, `or` and `xor` uses three register operands. An example of such an instruction is the following:

```
add   $15,$17,$2
```

In this example, `$15` is the target register (`Rt`), `$17` is the first source register (`Rs1`) and `$2` is the second source register (`Rs2`).

3. Opcodes `move` and `com` use two register operands. An example of such an instruction is the following:

```
move   $9,$17
```

In this example, `$9` is the target register (`Rt`) and `$17` is the first source register (`Rs1`).

4. Opcodes `sll`, `srl` and `sra` use three operands; the first two are registers and the last is a non-negative integer in the range 0 through 31 (decimal). An example of such an instruction is the following:

```
sll   $12,$17,5
```

In this example, `$12` is the target register (`Rt`), `$17` is the first source register (`Rs1`) and the constant `5` is the shift amount (`sa`).

5. Opcodes `jr`, `rdr`, `prr` and `prh` use just one operand, namely the target register (`Rt`). An example of such an instruction is the following:

```
jr   $15
```

(b) I-format Instructions: (Opcodes 18 through 25)

1. Opcode `li` uses two operands, the first of which is the target register (`Rt`) and the second is an integer constant (which may be positive, negative or zero). The following is an example of this instruction.

```
li   $15,-134
```

2. Opcodes `addi`, `subi`, `muli`, `divi` and `modi` use three operands; the first two are registers and the last is an integer (which may be positive, negative or zero). An example of such an instruction is the following:

```
addi   $12,$17,-5
```

In this instruction, `$12` is the target register (`Rt`), `$17` is the source register (`Rs`) and `-5` is the immediate operand.

3. Opcodes `lwb` and `swb` use the base + displacement form of addressing. The following examples use these opcodes.

```
lwb   $12,3($17)
swb   $12,-7($17)
```

In both of the above examples, `$12` is the target register (`Rt`) and `$17` is the source register (`Rs`). The immediate operands in these examples are `3` and `-7` respectively .

(c) J-format Instructions: (Opcodes 26 through 35)

1. Opcodes `lwa` and `swa` have two operands, namely a register and a a symbol. The following examples use these opcodes.

   ```
   lwa   $19,buffer
   swa   $19,total
   ```

   In both of the above examples, `$19` is the target register (`Rt`). The symbols used in the two instructions above are `buffer` and `total` respectively. The assembler should obtain the addresses (i.e., LC values) of these symbols from the symbol table. (For all the J-format instructions, the address is stored in the last 16 bits of the assembled instruction.)

2. Opcodes `j` and `jal` have only one operand, namely a symbol. The following examples use these opcodes.

   ```
   j       loop
   jal     sort
   ```

   The symbols used in the above examples are `loop` and `sort` respectively. The assembler should obtain the address of the symbols from the symbol table.

3. Opcodes `jeq`, `jne`, `jlt`, `jle`, `jgt` and `jge` have three operands. The first two are registers and the last is a symbol. The following is an example.

   ```
   jeq     $17,$13,loop
   ```

   In this example, `$17` is the target register (`Rt`) and `$13` is the source register (`Rs`). The symbol used in the above example is `loop`.