CDA3103 Final Project, rev. 1
MIPS Assembler
Due July 24, 2013 @ 11:59pm

1. INTRODUCTION

In this project, you are asked to write the core part of a simple MIPS assembler in the C programming language. Your assembler will read a text file containing a collection of MIPS assembly instructions and output a separate text file containing the 32-bit integer representation of the provided instructions. You will be provided with a handful of utility functions which may prove useful. You are allowed to implement any additional functions or data structures you deem necessary or helpful so long as you do not (1) add any additional files and (2) modify any files other than `assembler.c` and `assembler.h`.

2. FORMAT OF THE INPUT & OUTPUT FILES

The assembler takes a text file containing formatted assembly instructions as input and outputs a text file containing the decimal representation of the 32-bit binary machine code. An example of an assembly file with its machine code equivalent is shown below.

Input File Contents	Output File Contents

`.ent main`
 	
`main:`
`addi $8, $8, 1`
`addi $9, $8, 2`
`add $t2, $8, $9`
`bne $8, $t1, xyz`
`bne $10, $8, done`
 	
`xyz:`
`sub $8, $8, $t4`
 	
`done:`
`jr $ra`
`.end main`
	`554172417`
	`554237954`
	`17387552`
	`352911361`
	`357040129`
	`17580066`
	`65011720`

You are allowed to make the following assumptions about input:

- every assembler directive is on its own line begins with a period (".")
- every label is on its own line and ends in a colon (":")
- there are no comments
- there are no blank lines (the input file may contain blank lines; these lines will be removed when the file is read)

3. EXECUTING AND COMPILING

There are a variety of ways to compile and execute the assembler. This is a list of several ways in which you may compile and execute.

1. **Command line on a UNIX (Mac, Linux, BSD) system.**

   This is the method under which the project will be tested by the grader.

   To compile: Navigate to the directory containing the source code using the command line. Type '`make assembler`' (sans quotes).

   To execute: While still in the directory, type
   '`./assembler NAME_OF_INPUT_FILE NAME_OF_OUTPUT_FILE`'
   (sans quotes), where `NAME_OF_INPUT_FILE` is the path to your input file and `NAME_OF_OUTPUT_FILE` is the path to your output file.

2. **Command line on a Windows system.**

   First, you must install Cygwin (http://www.cygwin.com/); ensure that during installation you select to also install `make` and `gcc`.

   After installation of Cygwin, you'll have access to `make` and `gcc`. Use the instructions in option (1) to compile and execute your code.

3. **IDE of your choosing.**

   Depending on the IDE, you may need to download and install a C compiler separately. Given that every IDE is different, you'll have to read the documentation related to the IDE on specifically how to compile and provided command line arguments to the program. The arguments needed can be found above in option (1).

4. IMPLEMENTATION

4.1 Included files

You are provided with the following files.

- `assembler.c/.h`: This is where you are to do all of your work. You may not change the definition of `assemble()`, but you do need to populate its body. You may add any functions and data structures you believe will be helpful.
- `main.c`: Contains `main()`. Reads and writes the files. Creates two vectors: one which will hold the MIPS assembly and one to hold your 32-bit machine code.
- `utility.c/.h`: Contains functions to read and write files and trim whitespace from strings. You should not need to use the functions related to reading and writing files, but `string_trim(my_string)` may be useful to you.
- `vector.c/.h`: Contains functions related to vectors (simple dynamically sized arrays). You may use these functions to create and manage vectors.

Do NOT modify ANY of the files other than `assembler.c` and `assembler.h`. You are allowed to implement any additional functions or data structures you deem necessary or helpful so long as you do not (1) add any additional files and (2) modify any files other than `assembler.c` and `assembler.h`.

In the event that you find any bugs in the functions you are provided with, contact John Stone (jestone1990@knights.ucf.edu). Provide the following information:

- name of the function
- description of the bug
- a copy of your `assembler.c` and `assembler.h`
- how you believe you caused the bug to arise

4.2 Functions not to use

Do not use `itoa()`!!! This is a Windows specific function which does not port over to UNIX systems. A better option is to use `sprintf()` (http://www.cplusplus.com/reference/cstdio/sprintf/), which is available on all C compilers.

4.3 Functions to use

The following functions should prove useful to you while working on this project. All of these can be found in the `string.h` library.

- `strcmp`: Compare two strings
- `strstr`: Locate a substring
- `strtok`: Split a string into tokens delimited by some collection of delimiters

Documentation and examples of these functions can be found at http://www.cplusplus.com/reference/cstring/.

4.4 Bitwise operations in C

Given that this project requires you to manipulate substrings of bits, it will be necessary to use some of the bitwise operations that come with C. The ones that you will find to be most useful are and (&), or (|), left shift (<<) and right shift (>>).

- bitwise `and`
  A bitwise `and` compares two strings of bits, one bit at a time, and performs a logical AND on each pair of corresponding bits. The result in each position is 1 if the first bit is 1 and the second bit is also 1. Otherwise, the result is 0.

  C syntax:

  ```
  z = x & y
  ```

  Example:

  ```
    1011
  & 1001
  ------
    1001
  ```

- bitwise `or`
  A bitwise `or` compares two strings of bits, one bit at a time, and performs a logical inclusive OR on each pair of corresponding bits. The result in each position is 1 if either or both of the bits are 1. Otherwise, the result is 0.

C syntax:

```
z = x | y
```

Example:

```
  1011
& 1001
------
  1011
```

- left shift
  A left bit shift physically moves bits to the left by some amount, discarding any bits which "fall off" the end and padding any empty spaces to the right with zeroes. A left bit shift of `n` to integer `x` is the equivalent of $x*2^n$.

  C syntax

  `y = x << n`, where `x` is the number to be shifted, `n` is by how many places the number is to be shifted, and `y` will hold the result of the shift

  Examples

  ```
  11011 << 0 → 11011
  11011 << 1 → 10110
  11011 << 2 → 01100
  11011 << 3 → 11000
  11011 << 4 → 10000
  ```

- right shift
  A right bit shift behaves exactly the same as a left bit shift except that bits are moved to the right instead of the left. A right bit shift of `n` to integer `x` is the equivalent of $x/2^n$.

  C syntax

  `y = x >> n`, where `x` is the number to be shifted, `n` is by how many places the number is to be shifted, and `y` will hold the result of the shift

  Examples

  ```
  11011 >> 0 → 11011
  11011 >> 1 → 01101
  11011 >> 2 → 00110
  11011 >> 3 → 00011
  11011 >> 4 → 00001
  ```

Combining these bitwise options can prove to be powerful. For example, let's assume you have the following MIPS instruction:

```
addi $10, $9, 234
```

The opcode for `addi` is 9 (MIPS binary code: `001001`). The number for register `$10` is 10 (MIPS binary code: `01010`). Let's use the bitwise operations to "merge" these two values into one consecutive bit string such that the first six bits of our binary string is the `opcode` and the next five bits is our destination register `rd`.

Let the variable to hold the result be `q`.

```
unsigned int q = 0; // the resulting binary
int opcode = 9; // addi
int rd = 10; // $10

q = q | (opcode << 26);
// Shifting opcode to the right by 26 means that the first 6 bits
// of opcode will be 001001. or-ing q with opcode sets
// q = 0010 0100 0000 0000 0000 0000 0000 0000.

q = q | (rd << 21);
// Shifting rd to the right by 21 means that the first 6 bits of rd
// will be 0. The next five bits will be 01010. or-ing q with rd sets
// q = 0010 0101 0100 0000 0000 0000 0000 0000.
```

`q` now has a merged copy of `opcode` and `rd`. It should be quite simple from this point to figure out how to merge the various parts of your instructions into a single 32-bit binary integer.

One last trick

If you only want the lowest 16/26 bits of a number (**cough cough** immediate part of i/j-type instructions **cough**), a bit wise `and` should do wonders (for i-type: `q = q | (immediate & 0xFFFF)`; `0xFFFF` is 16 zeros followed by 16 ones).

4.5 Instructions to implement

Below is a table of instructions which are to be implemented. You must implement all of the instructions to get full credit. The exact encoding of these instructions can be found at http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html.

r-type	i-type	j-type
add	addi	j
div	bltz	jal
jr	bne	
mflo	lw	
mult	sw	
sll		
sub		

Hint: If you write your code right, once you get the first r-/i-/j-type instruction working, adding the remaining r-/i-/j-type instructions shouldn't take too much time.

4.6 Error checking

You must check for invalid instructions. Any instruction not listed above should be considered invalid; labels and assembler directives should not be considered invalid. You may assume that if a valid instruction is provided, it is formatted correctly. Valid instructions are only those listed above. Do not make assumptions about the amount of whitespace separating operands, however.

On an invalid instruction, the assembler should display an error message to the screen dictating the instruction which caused the error. The assembler should then immediately terminate.

4.7 Branches and jumps

I was battling with myself on the best way to explain this, so here is an explanation partially copied from http://fog.ccsf.edu/~gboyd/cs270/online/mipsII/if.html:

The label/address encoded in branch and jump instructions are called PC-relative (PC = program counter). The PC holds the address of the next instruction to be executed. By default if you are executing an instruction at address `n`, the PC is set to the address `n + 4` (the next instruction is 4 bytes forward). The label/address encoded in a branch or jump instruction contains a number that indicates how to adjust the PC (an offset). The adjustment is measured in number of words (which also happens to be the number of instructions).

Computing branch offsets

Here is a program which contains a couple branches:

```
   a:
1   add $t0, $t1, $t2
2   bne $t2, $t0, c
3   mult $t2, $t1
   b:
4   addi $t0, $t0, -1
5   bltz $t0, a
6   div $t0, $t3
   c:
7   mflo $t0
8   jr $ra
```

The problem of computing the offset for these branches can be broken into two situations.

Situation 1: The target is before the current instruction.

This situation is shown on the branch on line 5:

```
   a:
1   add $t0, $t1, $t2
2   bne $t2, $t0, c
3   mult $t2, $t1
   b:
4   addi $t0, $t0, -1
5   bltz $t0, a
6   div $t0, $t3
   c:
7   mflo $t0
8   jr $ra
```

The branch on line 5 is requesting a jump to label a, which is line 1. The offset for this branch will be -5. Why -5? Because we're adjusting the program counter by -5 instructions. "But there are only 4 instructions between the branch and the target. Where's the 5th instruction?" Let's take a look at all offsets from the line 5 branch:

offset	code
	a:
-5	1 add $t0, $t1, $t2
-4	2 bne $t2, $t0, c
-3	3 mult $t2, $t1
	b:
-2	4 addi $t0, $t0, -1
-1	5 bltz $t0, a
0	6 div $t0, $t3
	c:
1	7 mflo $t0
2	8 jr $ra

First, the offset is counted from the immediate next instruction (instead of the branch instruction); this has to do with when branch targets are computed in the pipeline. Secondly, we skip labels and directives when counting the number of lines separating the branch from the target; labels and directive are there only for the assembler and will not be part of the final output.

Situation 2: The target is after the current instruction.
This is the situation of line 2's branch:

```
  a:
1  add $t0, $t1, $t2
2  bne $t2, $t0, c
3  mult $t2, $t1
  b:
4  addi $t0, $t0, -1
5  bltz $t0, a
6  div $t0, $t3
  c:
7  mflo $t0
8  jr $ra
```

The offset for this branch will be 4. Here are all of the offsets from the branch:

offset	code
	a:
-2	1 add $t0, $t1, $t2
-1	2 bne $t2, $t0, c
0	3 mult $t2, $t1
	b:
1	4 addi $t0, $t0, -1
2	5 bltz $t0, a
3	6 div $t0, $t3
	c:
4	7 mflo $t0
5	8 jr $ra

What about jumps?
A jump is a branch that is always taken (an "unconditional branch"). The offset for a jump is computed *exactly* the same way as an offset for a branch.

4.8 Directives and labels

This assignment does not require you to create anything other than the code segment of an object file. Therefore, directives can be ignored as directives are used to create items such as the data segment and symbol table.

Labels are not instructions and should not be considered as such. Labels are used to assist the assembler in navigating code; they are simply used to compute PC-relative addresses. Your output file should, therefore, not contain any labels (or spaces for them).

4.9 Documentation

Your project should contain a README file which contains a list of the members in your group and a rough breakdown of the responsibilities of each member; in other words, answer the question, "Who did what?" (be brief).

Every function and `struct` you implement should be commented.

- A function should have a banner comment noting:

  1) a description of what the function does (not how it does it),
  2) a description of each parameter,
  3) a description of what the function returns, and
  4) any specific information that someone using the function should probably be aware of (e.g. if you're writing a function to compute square roots, it would be a good idea to mention that the user shouldn't provide the function with a negative number).

  Not every function will need all of the above; only provide what is needed for that function. When you provide the above, be complete but concise: no more than about 1-3 lines per description or note.

- A `struct` should have a comment indicating the purpose of the `struct`. Each component inside the `struct` should also have a comment indicating what the component is.

5. SUBMITTING

You are to submit 3 files: `assembler.c`, `assembler.h`, and `README`. Compress these files into a ZIP file named `assembler.zip`. Submit `assembler.zip` on WebCourses by the due date.

6. EXTRA CREDIT

If you would like 5 points extra credit, implement the "load immediate" instruction `li`. `li` is actually 2 instructions: `lui` and `ori`. Why is it two instructions?

- http://www.engr.uconn.edu/~jeffm/Classes/CSE240-Spring-2000/Lectures/lecture6/node4.html
- http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/load32.html