CSCI 2500 — Computer Organization
Homework 05 (document version 1.0) — Due November 9, 2021
Can't stop compiling

- This homework is due by the midnight EDT on the above date via a Submitty gradeable.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- Homework assignments are available approximately ten calendar days before they are due. Plan to start each homework early. You can ask questions during office hours, in the Submitty forum, and during your lab session.

This homework will be a thrilling adventure of writing your first compiler that would translate C code into MIPS. Do not worry, it is going to be extremely simplified, and actually only capable of compiling C assignment statements. At the same time, developing this compiler should help you solidify your MIPS knowledge as well as reflect upon performing arithmetic operations.

Overview

You will implement a rudimentary compiler in C that translates one or more arithmetic expressions as assignment statements into valid MIPS code. You can assume that each C variable name is one lowercase letter (e.g., `a`, `b`, `c`, etc.) and of type `int`.

Integer constants may be positive, negative, or zero. Further, you will need to support the addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and mod (`%`) operators. Treat all of these as 32-bit integer operations with each operation requiring two operands.

You will also need to support multiple lines of input, i.e., multiple assignment instructions. To avoid overcomplicating your implementation, you can assume that each assignment statement will consist of a mix of addition and subtraction operations, a mix of multiplication or division operations, or a simple assignment (e.g., `x = 42`). Therefore, you can always parse each line from left to right.

The MIPS code you generate must make proper use of registers `$s0,...,$s7` to correspond to C variables and registers `$t0,...$t9` to correspond to any temporary variables you need. Variables in MIPS should match those in C from left to right. Note that the final result of the last assignment statement will likely not end up in register `$s0`.

You can again assume that you will not need more than the specific MIPS registers listed here, with the use of temporary registers cycling from `$t9` back to `$t0` as necessary.

Required Input and Output

Your code must read the input file specified as the first command-line argument (i.e., `argv[1]`). Translate each line of input into corresponding MIPS instruction(s). In addition, include each original line being translated as a comment. Note that the use of pseudo-instructions is fine.

Below are a few example runs of your program that you can use to better understand how your program should work, how you can test your code, and what output formatting to use for Submitty.

In the first example (i.e., with input `example1.src`), register `$s0` corresponds to C variable `g`, register `$s1` corresponds to `h`, and register `$s2` corresponds to `f`.

```
bash$ cat example1.src
g = 100;
h = 200;
f = g + h - 42;
bash$ ./a.out example1.src
# g = 100;
li $s0,100
# h = 200;
li $s1,200
# f = g + h - 42;
add $t0,$s0,$s1
addi $s2,$t0,-42
bash$ cat example2.src
q = 12;
j = q - 2;
x = q * q / j;
bash$ ./a.out example2.src
# q = 12;
li $s0,12
# j = q - 2;
addi $s1,$s0,-2
# x = q * q / j;
mult $s0,$s0
mflo $t0
div $t0,$s1
mflo $s2
```

Be sure to test your resulting MIPS code, for example by using MARS or QtSpim and checking the values of the relevant registers to verify that your MIPS code properly computes the correct answers and stores them in the appropriate registers.

For multiplication and division, you will need to make use of the `LO` register, as shown in the following example.

```
bash$ cat example3.src
a = 10;
b = 73;
c = a * b / a;
bash$ ./a.out example3.src
# a = 10;
li $s0,10
# b = 73;
li $s1,73
# c = a * b / a;
mult $s0,$s1
mflo $t0
div $t0,$s0
mflo $s2
```

For modulo (%), you will need to make use of the `HI` register, as shown in the following example.

```
bash$ cat example4.src
m = 9;
e = -9;
w = m * e * m % e;
bash$ ./a.out example4.src
# m = 9;
li $s0,9
# e = -9;
li $s1,-9
# w = m * e * m % e;
mult $s0,$s1
mflo $t0
mult $t0,$s0
mflo $t1
div $t1,$s1
mfhi $s2
```

Assumptions

Given the complexity of this assignment, you can make the following assumptions:

- Assume all input files are valid.
- Assume the length of `argv[1]` is at most 128 characters.
- Assume that constants will only appear as the second operand to a valid operator (e.g., `x = 42 / y` is not possible).
- Assume that expressions will never have two constants adjacent to one another (e.g., `x = 42 / 13` is not possible).

Simplifying Multiplication and Division

There are two "simplifications" that you are required to implement. When you multiply or divide involving a constant operand, there are cases in which you should not use `mult` or `div` instructions. Instead, for multiplication, break the constant down into its sum of powers of two, then use a series of `sll` instructions to perform each multiplication, adding the resulting intermediate products.
As an example, to multiply n by constant 45, start by determining sum $2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45$. Next, multiply $n \times 32$, $n \times 8$, and $n \times 4$ using three successive `sll` instructions. Note that there is no need to multiply n by 1 for the last term.
Finally, add each term as you calculate it, thereby accumulating the sum. More specifically, the first intermediate product should be placed in the target register via `move`, while subsequent intermediate products should be added via `add`.
Note that you should use only two temporary registers for this, the first to calculate each required power of 2, the second to accumulate the sum. For example, the `example5.src` input shown below should produce the MIPS code shown.

```
bash$ cat example5.src
n = 100;
b = n * 45;
bash$ ./a.out example5.src
# n = 100;
li $s0,100
# b = n * 45;
sll $t0,$s0,5
move $t1,$t0
sll $t0,$s0,3
add $t1,$t1,$t0
sll $t0,$s0,2
add $t1,$t1,$t0
add $t1,$t1,$s0
move $s1,$t1
```

To multiply by a negative constant (e.g., `n * -45`), replace the last `move` instruction with a subtraction from `$zero`, as in:

```
sub $s1,$zero,$t1
```

And to multiply by given constant 0, treat this as a special case in which the target variable/register simply is assigned to 0 (as shown below). Note that this could be a temporary register if it appears as part of a longer expression.

```
li $s3,0
```

Another special case is multiplying by constant 1 or -1. These cases are shown via the example snippets below:

```
# b = a * 1;
move $t0,$s0
move $s1,$t0

# b = a * -1;
move $t0,$s0
sub $s1,$zero,$t0
```

Division

The relationship status of division is definitely "it's complicated."

For division in this assignment, if the divisor is an exact power of 2 (e.g., 32), we *might* be able to use the right-shift simplification in MIPS. More specifically, we can use this simplification if the first operand is non-negative.

To implement this logic in MIPS, we can use the `bltz` (*Branch if Less Than Zero*) instruction to test whether the first operand is negative. If it is negative, we then branch to the standard `div`/`mflo` approach described earlier. The example below shows generated MIPS code for the above logic (regardless of variable n):

```
bash$ cat example6.src
n = 100;
b = n / 32;
bash$ ./a.out example6.src
# n = 100;
li $s0,100
# b = n / 32;
bltz $s0,L0
srl $s1,$s0,5
j L1
L0:
li $t0,32
div $s0,$t0
mflo $s1
L1:
```

Note that labels should be generated as `L0`, `L1`, `L2`, ..., `L10`, `L11`, `L12`, ..., and be on lines of their own (as shown above).

And to divide by a negative constant (e.g., `n / -32`), after the `srl` instruction, perform a subtraction from `$zero`, i.e., `sub $s1,$zero,$s1`. Further, for the special case of dividing by constant 1 or -1, the following examples show the required behavior:

```
# b = a / 1;
move $s1,$s0

# b = a / -1;
sub $s1,$zero,$s0
```

Submission and Grading Criteria

For this assignment, you will submit your code into the Submitty gradeable. As a reminder, you **must** use C for the coding part of this homework assignment. Your code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04. Grading criteria for this assignment are as follows.

1. Code correctness as determined by standard visible and hidden Submitty autograded test cases: 90%
2. Coding problem TA grading: 10%
    - Code has clear and logical organization and is well structured. For example, functions have well defined responsibilities, there are no huge "God" functions, arguments and return values are properly used to pass data between callers and callees, there are no (or very few) global data items, etc.
    - Code is properly formatted, commented, and consistently follows some C style guidelines (see https://www.gnu.org/prep/standards/html_node/Writing-C.html or http://www.literateprogramming.com/indhill-annot.pdf as examples).