CSCI 2500 — Computer Organization
Homework 4 (document version 1.4)
Compiling C Assignment Statements into MIPS

Overview

- This homework is due by 11:59:59 PM on Friday, November 9, 2018.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully execute on Submitty to obtain full credit.

Homework Specifications

For this individual homework assignment, we revisit Homework 2 and implement a rudimentary compiler in C that translates one or more arithmetic expressions as assignment statements into valid MIPS code. Just as with Homework 2, you can assume that each C variable name is one lowercase letter (e.g., `a`, `b`, `c`, etc.) and of type `int`.

Unlike Homework 2, in this assignment, 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.

As with Homework 2, 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 unlike Homework 2, the final result of the last assignment statement likely will not end up in register `$s0`.

You can again assume that you will not need more than the specific MIPS registers listed here, with 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 instructions. And include as a comment each original line being translated. 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
sub $s2,$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;
sub $s1,$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 `print $N` in `spim` to verify your MIPS code properly computes the correct answers and stores it in the appropriate registers.

(v1.1) For multiplication and division, you will need to make use of the `HI` and `LO` registers, 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
```

(v1.1) 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, but do not assume that `argv[1]` is populated.
- 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 a 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`.

(v1.1) 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 `example4.src` input shown below should produce the MIPS code shown.

```
bash$ cat example4.src
n = 100;
b = n * 45;
bash$ ./a.out example4.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
```

(v1.1) 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
```

(v1.1) 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
```

(v1.3) 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
```

(v1.2) Division

The relationship status of division is definitely "it's complicated." (See the in-class notes for 10/30.)

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 example5.src
n = 100;
b = n / 32;
bash$ ./a.out example5.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`. **(v1.3)** 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
```

Error Checking

Given the complexity of this assignment, you can assume that all input files are valid. Further, you can assume the length of `argv[1]` is at most 128 characters. But do not assume that `argv[1]` is populated. Be sure to verify this via `argc`, displaying an error message if the argument is missing.

In general, if an error occurs, use either `perror()` or `fprintf( stderr, "..." )`, depending on whether the global `errno` is set.

And be sure to return either `EXIT_SUCCESS` or `EXIT_FAILURE` upon program termination.

Submission Instructions

Before you submit your code, be sure that you have clearly commented your code (this should not be an after-thought). Further, your code should have a clear and logical organization.

To submit your assignment (and also perform final testing of your code), please use Submitty.

Note that the test cases for this assignment will be available on Submitty a minimum of three days before the due date and will include hidden test cases.

Also as a reminder, your code **must** successfully execute on Submitty to obtain credit for this assignment.