Computer Architecture

Lab 1

The goals of this lab are to get familiar with the MARS development environment, think a bit about directly using memory addresses in our programs, and to get some experience writing actual assembly code.

You should turn in the source files for the two programs you're asked to write. Both documents should be submitted via the tool available on the course's Sakai site for this lab. These are due by the start of the next class period.

Setting up MARS

We are going to use the MARS simulator for MIPS. It's hosted on the website of Missouri State University: https://courses.missouristate.edu/KenVollmar/mars/index.htm

You can download it on the lab machines and on your personal machines as well.

Download the example1.asm program from Sakai, assemble, and run it to make sure things are working.

Vector Addition

Let's start by writing a MIPS program to add two vectors together.

First, variables are not physical! The only thing that's physical is the data and the labels for that data. We call these labels addresses. So, in low-level programming—that is, programming which has zero abstraction and therefore must directly correspond to what is physical—we must interact with these addresses in a significant way.

Instead of creating a vector using a variable declaration with appropriate initialization, we only have the raw data. In assembly programming, we have `.data` regions where we can place the data needed for a program. Write the following into the MARS simulator:

```
      .data
x:    .word 2, 6, 4, 9, 10, 3, 2, 6, 5, 2, 9, 8
```

Normally, the assembler will convert the instructions you write into 1's and 0's. The `.data` directive tells the assembler to represent raw data, rather than instructions. The `.word` directive tells the assembler to use word-sized locations to store whatever data follows. In this case, we are creating an array of 12 values. The letter `x` to the left of the `.word` directive is a label, and it corresponds to the physical label (i.e., the address) where the array begins.

Make sure you get all that, and then add another two lines:

```
y:    .word 3, 8, 1, 0, 9, 2, 4, 1, 1, 7, 3, 4
z:    .word 1:12
size: .word 12
```

Here we create a second array, with **y** representing the label where it begins, and a single data element that holds the **size** of our arrays. We also set aside 12 word-sized locations, without putting data in them, which we label **z**. This is for the sum vector. These lines set up all the data our program requires.

Note that **x**, **y**, **z**, and **size** here are not variables. They are labels that represent the address of the contents to the right. These are used by the assembler to compute the required address arithmetic for working with the data. As an assembly programmer, we can use them in lieu of knowing the exact memory addresses involved.

Now we're ready to write the program. We use the **.text** directive to tell the assembler to convert the instructions that follow into their 1 and 0 encodings.

Since we can't use **x** and **y** as variables—that is, we can't change them—we can only access the data through the data's addresses. So, the first thing we have to do is load those addresses into registers.

```
    .text
    la $t0, x
    la $t1, y
    la $t2, z
    la $t3, size
```

The **la** instruction loads the address of the label into the destination register. So, these instructions set **$t0** to the address of the first element of **x**, **$t1** to the address of the first element of **y**, **$t2** to the address of the first element of **z**, and **$t3** to the address of **size**. The **.text** directive tells the assembler that instructions follow. (Technically, **la** is a *pseudo-instruction*: that is, a macro which is compiled into two other instructions. But, we'll use it as an instruction for now.)

We can't give the registers **$t0** through **$t3** different names. We can't use identifiers: we just have to remember what the data in each is being used for.

Now, we want the actual size, not just its address. To do this we use the **lw** instruction, which loads the word at a specific address into a register.

```
lw $t4, 0($t3)
```

Now **$t4** is equal to the contents of the memory location given by **$t3**. (Actually, it's equal to the contents of the memory location given by **$t3 + 0**, where we can change that **0** to access different elements. But, we'll most often just use a **0** offset there.)

Again, we can't rename **$t4** so we'll just have to remember that it is holding the size information for our arrays.

In C++, we'd add these arrays together with a **for** loop such as

```
for(int i = 0; i < size; i++)
      z[i] = x[i] + y[i];
```

In assembly, we don't have a **for** instruction. Instead, we have to build up each of the pieces separately.

Whereas in the **.data** section we used labels to indicate the memory addresses of the data items, in the **.text** section we'll use labels to indicate the addresses of our instructions: that is, the addresses of our code. The computer will automatically execute the next instruction in sequence: if we want to change that, say to implement a **for** loop, we need to use a *branch instruction* which requires us to know the address of the start of the loop.

The first thing we want to do in the **for** loop is to access the next elements of our **x** and **y** arrays. We accomplish that with the following:

```
loop: lw $t5, 0($t0)
      lw $t6, 0($t1)
```

The **loop:** is a label so we can use branch instructions to return to the start of the loop. The registers **$t5** and **$t6** now hold the first two elements of **x** and **y**. Make sure you see this.

We now need to sum the array values. This is relatively straightforward:

```
add $t7, $t5, $t6
```

The *destination register* is the one on the far left, and the *source registers* are the ones to the right. So this computes **$t5 + $t6** and stores the result in **$t7**.

We now want to store the sum into the first element of **z**. We do this using **sw** (store word), which works just like **lw**, except in reverse.

```
sw $t7, 0($t2)
```

This stores the value in **$t7** in the memory location contained in **$t2** (which, if you recall, held the location of the first element of **z**.)

Next we need to increment the loop index. We can do that with the following:

```
add $t0, $t0, 4
add $t1, $t1, 4
add $t2, $t2, 4
```

We don't have **i** stored as a variable in a particular location. Instead, we have separate addresses tracking where we are within each array. We have to add 4 to increment through because the word-size is 4 bytes. If we just add 1, then we will be at the second byte of the first element, which is not what we want. (Recall this is exactly what happens in C++ pointer arithmetic: when you add 1 to a pointer you don't really add 1, you add the size of the data type the pointer is pointing to, which is one reason why pointers need to be typed.)

Now we increment **i**, which we accomplish through decrementing **size**:

```
add $t4, $t4, -1
```

You could have a register dedicated to **i** starting at 0 and going up to **size**, but this way works, too, and uses fewer resources.

To evaluate the loop termination criteria **i < size** (or **size > 0** in our case), we need to evaluate a boolean expression. To do this in MIPS assembly, we use the **slt** (*set-if-less-than*) instruction. This will set (i.e., make 1) a destination register if the first source register is less than the second source register. Otherwise, the destination register is cleared (i.e., made 0.) (See p.93 in the text for more details.)

```
slt $t8, $zero, $t4
```

This will set **$t8** if **0 < $t4** (**$zero** is a special register that always holds the constant value 0. It's surprisingly useful!) Once **$t4** reaches 0, **$t8** will be cleared. We can now use a branch instruction to go back to our **loop:** label if **$t8** is set.

```
bne $t8, $zero, loop
```

The **bne** instruction stands for *branch-if-not-equal*. We also have a **beq**, *branch-if-equal*, instruction and the entire loop could be written using complementary logic (reverse the **slt** operands and then use **beq** instead of **bne**.)

This is how **if** statements are evaluated in MIPS assembly. First, you use **slt** to evaluate the boolean expression, and then you use **beq** or **bne** to jump to the required branch. For complex conditions, these instructions may have to be layered carefully.

When the condition is not met, that is, when **$t8 = $zero**, the branch will not be taken and the next instruction in sequence will be executed instead. MIPS has an unconditional branch instruction called **j label** if you want to always goto a given instruction (again, this is actually required to make if statements work.)

OK, that's it! Assemble and run the program. Inspect the Data Segment window to look to see if the addresses for z are appropriately updated. You should see the **.data** values for your arrays and **size** data and then the last 12 should be for the **z** vector. The last two addresses should have the value **c** in them (**c** is **12** is hexadecimal, and most numbers are represented in hex here.) You can also see the various residual values in the registers in the window to the right.

Lab 1 Exercise

Now try one on your own. Use the following data:

```
        .data
x:      .word 3, 4, 8, 2, 9, 3, 6, 7, 10, 4, 12, 5, 7, 1, 2
size:   .word 15
```

Write a program that **(1)** computes the sum of all elements of **x** and stores the result in register **$s0** and **(2)** counts the number of elements of **x** that are greater-than-or-equal-to 5 and store the count in register **$s1**. (Hint: the **slti** instruction may be useful. You can type it into the MARS editor and get info from the IDE on its syntax, or you can look at page 93 of the text.)