CSE 410 22wi - HW1 Supplement

Part 1: Overview of the HW1 Tasks

Homework 1 is mainly about understanding memory addresses, and to a lesser extent about what labels are in assembly programming.

CSE 410 assembler (executable) programs have two memory regions, a “text segment” that contains instructions to be executed and a “data segment” that holds possibly initialized data. In our simulator, each instruction takes one word of memory and each data item takes one word of memory. Memory in each segments is numbered starting at zero. So, in this example

```
                .text
main:           lw      x3, 2(x0)       # offset 0 in text segment
                halt                    # offset 1 in text segment
                .data
                .word   10              # offset 0 in data segment
                .word   11              # offset 1 in data segment
                .word   12              # offset 2 in data segment
                .word   13              # offset 3 in data segment
```

- The lw instruction is at offset 0 in the text segment
- The halt instruction is at offset 1 in the text segment
- The value 10 is at offset 0 in the data segment, 11 at offset 1, 12 at offset 2, and 13 at offset 3
- What’s at offset 4 in the data segment? We don’t know. It’s not initialized, so it could be anything. (Same for offset 2 and beyond in the text segment.) Memory locations always have bits, even if your code hasn’t set them to any particular value.

The lw in this program says to take the contents of x0, which is a special register whose contents are always zero, and add 2 to them, then fetch the word from the data segment at the offset given by the result (2) and copy the bits there into register 3. (“x3” is the name for register 3 when talking to the assembler. That distinguishes it from “3”, which might mean 3.) That is, we put 12 into register 3.

For this program, we know at code time (now) what it will do, because we know what the values of the data segment words are. For hw1, though, you should imagine that the data segment looks more like this:

```
        .data
        .word   ???
        .word   ???
```

For instance, pretend the first thing our program did was read three integers from the keyboard and store them in the first three words of the data segment. Then you wouldn’t know at code time what the value would be that you loaded from memory into a register with a lw instruction. We don’t actually do that because I/O (input-output) is complicated, and at this point the simulator doesn’t support it (and we have no plans to support it). So, while we know at code time what the values are, you have to pretend that we don’t, that they could be anything, and then write instructions that work in that circumstance. One way to think of this is that **we will replace your .data segment** with a different one when grading your program, and your code should print the right result for our data.

Part 2: Labels

It is clumsy and very error prone for the assembly language programmer to count offsets. More substantial programs will use hundreds or thousands of memory locations in the data segment, and will have thousands of instructions. To help, the assembler lets you put labels anywhere you'd like, and then refer to the label instead of having to give an offset as a literal. For instance, here's the example code above re-written using a label:

```
                .text
main:           lw      x3, A
                halt
                .data
                .word   10
                .word   11
 A:             .word   12
                .word   13
```

Three things about this:

- The assembler considers labels to be symbolic constants, kind of like
      final int A = 2;
  in Java. That is, a label is an integer. It is not a variable – you cannot assign a value to it. Its value is known at assembly time, because the assembler figures out what offset it's at.
- The value of "main" above is 0. It's at offset 0 in its segment (the text segment). The value of A is 2. It's at offset two in its segment. Therefore,
  ```
  lw   x3, A(x0)
  lw   x3, 2(x0)
  ```
  are exactly the same. So are
  ```
  lw   x3, A(x2)
  lw   x3, 2(x2)
  ```
- If you just say
  ```
  lw   x3, A
  ```
  the assembler turns it into
  ```
  lw    x3, A(x0)
  ```
  which is
  ```
  lw    x3, 2(x0)
  ```

Part 3: More About Labels

Because the assembler thinks of labels as symbolic constants, you can use them wherever you could use a constant.

```
.text
addi      x1, x0, second
```

```
          addi    x2, x0, first
          sub     x1, x1, x2
          print   x1
          halt
          .data
          .word   1
first:    .word   1
          .word   1
second:   .word   1
```

That program prints 2.

Part 4: Effective Addresses

The "effective address" is the address computed by adding the contents of the base register and the offset. For instance, if x2 contains the value 3, then 10(x2) results in an effective address of 13.

- An "absolute address" is an offset into the data segment. If you knew you wanted word 10 of the data segment, for instance, you'd write
  ```
  lw   x3, 10            # or …
  lw   x3, 10(x0)
  ```
- If you want a word that has a label, say X, then you don't need to know its offset, you just use its label:
  ```
  lw     x3, X   # or
  lw     x3, X(x0)
  ```
- Sometimes you know that the word you want is some distance past where a base register points. For instance, arrays are stored as consecutive words of memory. In many languages, int A(10) reserves 10 words of memory, one right after another. if x2 holds the address of word 0 of array A, then you can load A[3] into x4 with
  ```
  lw    x4, 3[x2]
  ```
  At code time you don't need to know where the array is in memory, just that x2 points at it (i.e., holds the address of A[0]) and that you want the word at offset 3.
- What about A[i]? What if x2 holds the address of A[0] and x3 holds the index i? Then you need an extra instruction:
  ```
  add    x4, x2, x3
  lw     x4, 0(x4)
  ```

Here's some sample code that is array-like:

```
.text
addi    x1, x0, Array    # point x1 at Array
lw      x2, 2(x1)        # put Array[2] in x2
addi    x3, x1, 2        # these two instructions are a slower way to…
lw      x2, 0(x3)        #       load Array[2] into x2
halt
.data
.word   4
```

```
Array:  .word  -1
        .word  -1
        .word  -1
        …
```