Adarsh Solanki as5nr
3/28/12
CS 2150: Lab 8 InLab

Parameter Passing

I first created a C++ function which called a function with the following signature: "`int add ( int x, int y );`" with the goal of examining the passing of the two integer parameters to the subroutine.

The first thing I noticed was the nomenclature of the assembler subroutine corresponding to my add() function. (__Z3addii) I assume the Z3 is some sort of hashed value or perhaps a way to identify the return type. The name 'add' is preserved, and I believe the appended 'ii' represent the two **i**nteger parameters. Also, all of the instructions have an 'l' appended to them, functioning on long 32-bit data values.

I am sort of confused about the couple of seemingly redundant instructions that seem to be inverses of each other. For example, there are many instructional pairs of the form:

```
movl    %eax, -12(%rbp)
movl    -12(%rbp), %eax
```

which seemingly does nothing.

Within the subroutine, the assembler stored the two parameters x and y into edi and esi respectively, but then also stored them into

Next, I slightly altered the signature of the function to be "`int add ( int &x, int y );`", using a pass-by-reference for one of the integer parameters. In this case, rather than using the 'movl' command to use the parameter values in the subroutine, the assembler instead used the LEAQ instruction that Loads the Effective Address rather than the value. This makes sense, as the actual bits of data representing the integer should not be passed to the subroutine as in pass-by-value but rather the bits representing the location in memory of the parameter, hence the pass-by-reference. Also, the name of the subroutine was changed from Z3addii to Z3addRii, with the R standing for reference. I predict that if the second parameter were instead the reference, then the name would probably be Z3addiRi.

When the first parameter was instead changed to a constant reference, the assembler made very few changes. According to the unix diff command, the only differences between the compiled code with a constant pass-by-reference and a normal pass-by-reference is in the nomenclature. The subroutine's name is changed from Z3addRii to Z3addRKii with the K standing for 'constant' probably.

When a character is passed, the mov function is movb instead, moving the byte-sized character rather than the 'long' integer.

To handle a floating point value, the assembler relies on more complex instructions such as "cvttss2si" to translate floating point values into more easily computable signed values.

This lab has enlightened me to the intense 'plumbing' going down underneath the shiny façade of high-level programming languages, as it is impossible to skip the step of assembly to machine code. This strange land of extremely truncated function names and obscure acronyms is behind all of the digital computation that exists, and the importance of a good/efficient assembler is suddenly much more clear to me.