|  |
| --- |
|  |
| **Instruction Set Architecture (Comparison Across Different Platforms)** |
| Brandon Chin |
|  |
| **CSC342 - Instructor: Prof. Izidor Gertner** |
| **3/2/2015** |

|  |
| --- |
|  |

**Objective:**

Instruction set architecture (ISA) refers to part of the interaction between hardware and software. It involves giving basic machine language commands to the CPU, and telling it what to do. We will primarily analyze this though debugging and disassembling examples over three major operating systems; MIPS/MARS, Windows, and Linux. (These examples are taken from the textbook -- *Computer Organization and Design*, 4th Ed. by David A. Patterson and John L. Hennessy)

**MIPS on MARS**

The first example we will look it will evaluate the expression:

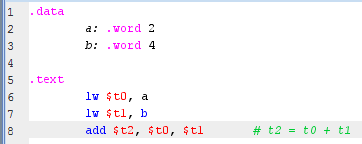
*c = a + b*

In MIPS assembly language, we must enter the following equivalent statement for the computer to understand:

*add c, a, b*

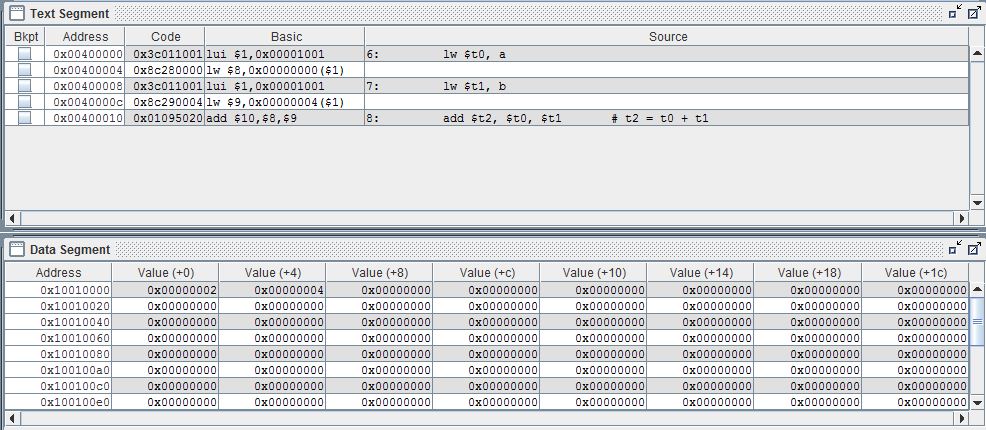
This means we are performing the add operation, storing the result into c, with inputs a and b. In other words, c equals the result of adding a and b.

In order to assemble this into MIPS, we must first initialize values for a and b. Then we can add them, and store the result. This is accomplished in the following way:



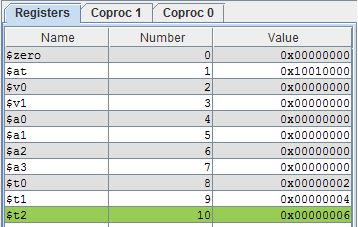
Here, we can see that there are two segments, "data" and "text". Data is where we will initialize our values for a and b. "a" and "b" are the variable names, ".word" indicates the data type, which is a 32-bit word, and "2" and "4" are the values assigned to a and b respectively. Text is where we will represent this data in registers, and perform the operations. We begin by loading "a" into register $t0, and "b" into register $t1. Then we will add the values stored in registers $t0 and $t1, and store the result into register $t2.

Now let's analyze these segments in memory:



Here we can see the text segment above, and the data segment below. The text segment displays the address in which these instructions are stored, as well as the instruction code, and the corresponding source code. Take a moment to notice the offset between memory addresses is 4. In the data segment, we can just see the address and the values for "a" and "b". Once again, we can see the offset between each address is 4.

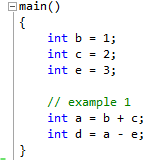
Now let's look at the registers:



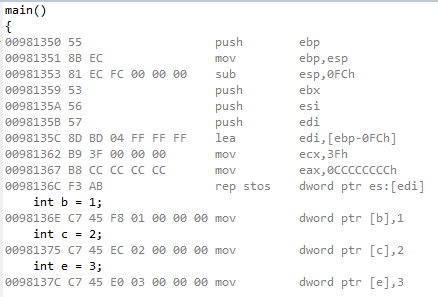
If we recall, we had stored address of the values 2 and 4 in the registers $t0 and $t1 respectively. Looking at the registers window, we can see these values are represented in hexadecimal notation. Finally, our result, 6, is represented in register $t2 (highlighted in green).

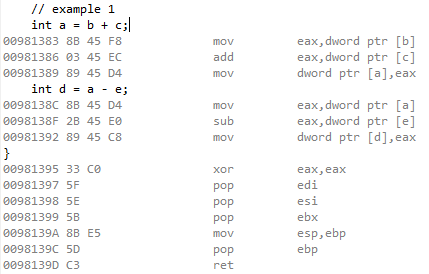
**Windows OS**

For Windows, we will be using the MS Debugger from Microsoft Visual C++ 2010 Express on a 64-bit Intel Core i5 CPU. Let's debug and analyze the following code segment written in C:



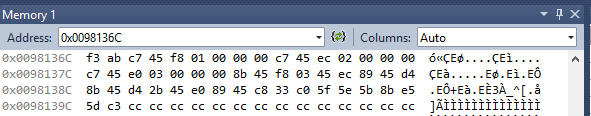
This is a similar example from before, however, this time we are initializing another variable "e", subtracting "e" from "a", and storing the result in "d". Let's look at the disassembly for this:





When the main function begins, a stack must be created in memory. This can be seen in the first line where a base pointer (ebp) is pushed into memory. This marks the beginning of the stack. Next, a stack pointer (esp) is created at the location of the ebp, however, this will mark the top of the stack, and will grow accordingly as data is pushed and popped into and from the stack. If we continue to look down, we will eventually see where our first few C statements are being executed. Values are being assigned to their respective variable names. Then we can see the addition statement, and the subtraction statement, as well as the necessary machine code instructions. Finally, as we approach the end of the main function, we can see that the registers are being cleared and the stack is de-allocated. The base pointer is popped, and the "ret" instruction is called, indicting the return call of main.

Now let's look at the memory addresses:

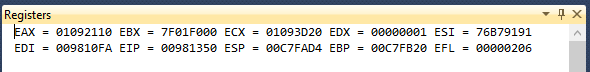


On the left, we have the address of the first instruction, which can be seen by the left most 8 bit hexadecimal number. The following instructions are displayed, however, there addresses are not. Each address on the left is actually 16 addresses apart from the next. For example, the address:

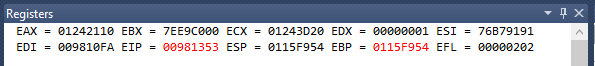
*0x0098136C*

is the address for the instruction *f3*. The address for instruction *ab* is *0x0098136D*, and so on.

Finally, let's look at the registers:



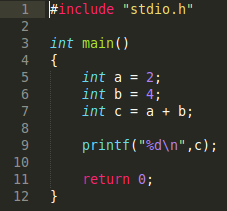
This is taken at the start of the main function.



This is taken after the base pointer and the stack pointer are allocated. We can see that the ebp and the esp store the same memory address location.

**Linux OS**

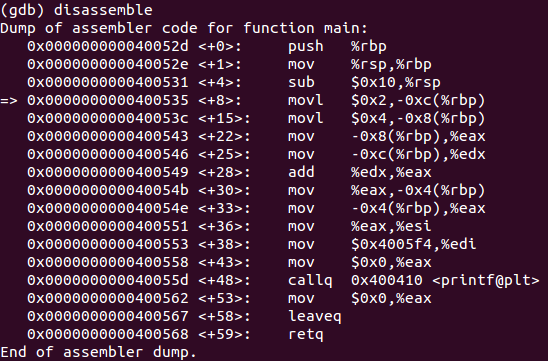
For Linux, we will be using Ubuntu to run GCC and GDB on a 64 bit AMD Phenom II X4 CPU. Let's just look at this code segment, again written in C:



If we compile this program with gcc, and run it, we get the following output:

C:\Users\brandon\Downloads\Screenshot from 2015-03-04 01-11-22.png

If we run gdb on the executable file, break at main, and then disassemble, we can see the following:



Here, from left to right, we can see the address of each instruction, the offset from the first instruction at the start of main, and then the assembly code. Like before, we can also see the allocation of the stack at the beginning of main, the de-allocation of the stack, and the return statement at the end of main.

**Conclusion:**

We have used three different platforms to analyze how code is interpreted by the CPU and stored in memory. We saw how temporary memory must be stored in a stack, that way the stack can be destroyed once its data is no longer in use. This is also the only thing that is being looked at until the stack is destroyed (in the above examples, the main function is allocated on stack, executed, and then destroyed at the end of the function).

The allocation of stack, as well as all of its contents and statements, are translated into machine code by the compiler. By debugging our program, we saw where each machine code instruction was stored in memory. These instructions told the CPU what to do with certain data and registers. Before being processed, data must be represented by a register. Finally, these registers contain the address of the data in memory, rather than the data itself. This summarizes a CPU's instruction set -- the basic commands in the computer's machine language.