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:

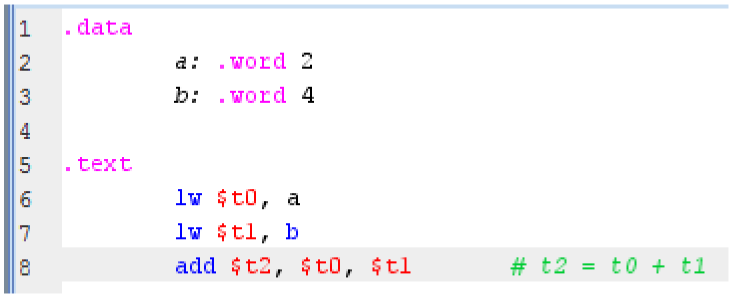
c = a + b

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.

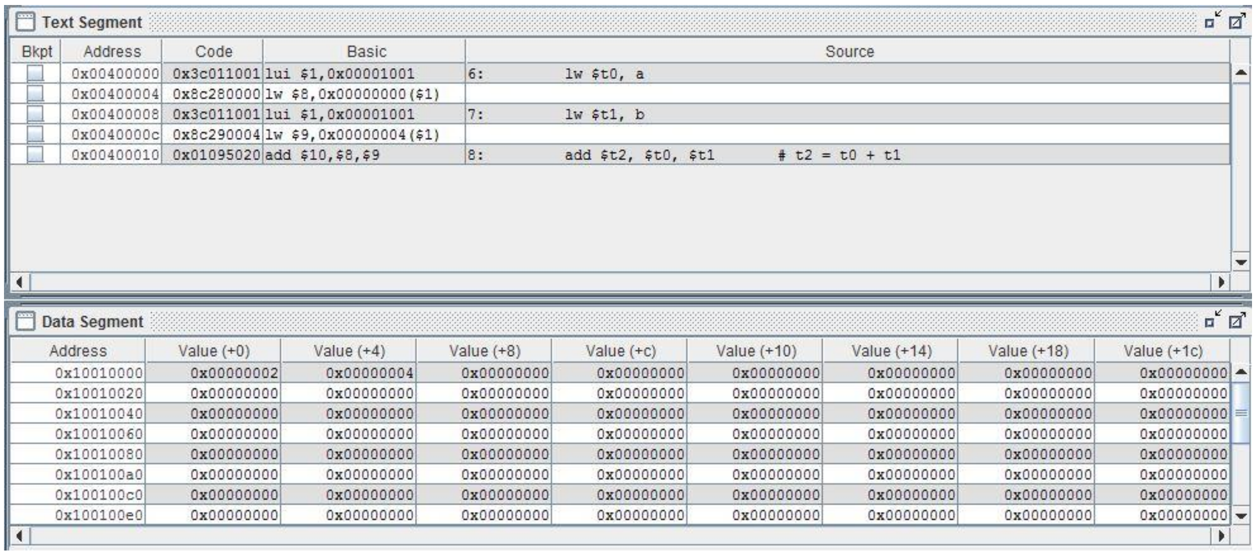
In order to assemble this into MIPS, we must first initialize values for a and b. Then we can add them, and store the result. This is accomplished in the following way:



```
.data
        a: .word 2
        b: .word 4

.text
        lw $t0, a
        lw $t1, b
        add $t2, $t0, $t1       # t2 = t0 + t1
```

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:



Text Segment

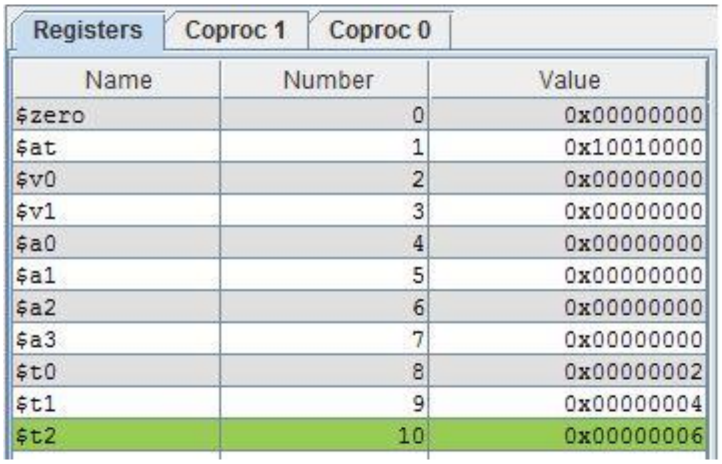
Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	6: lw $t0, a
☐	0x00400004	0x8c280000	lw $8,0x00000000($1)	
☐	0x00400008	0x3c011001	lui $1,0x00001001	7: lw $t1, b
☐	0x0040000c	0x8c290004	lw $9,0x00000004($1)	
☐	0x00400010	0x01095020	add $10,$8,$9	8: add $t2, $t0, $t1 # t2 = t0 + t1

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0x00000004	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

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:

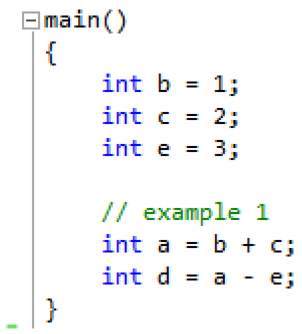


Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000002
$t1	9	0x00000004
$t2	10	0x00000006

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).

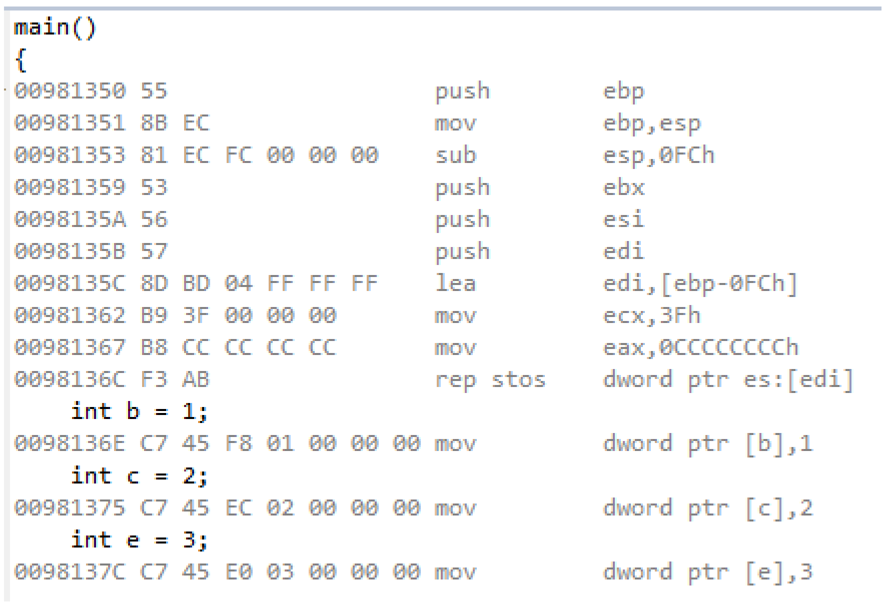
Windows OS

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:

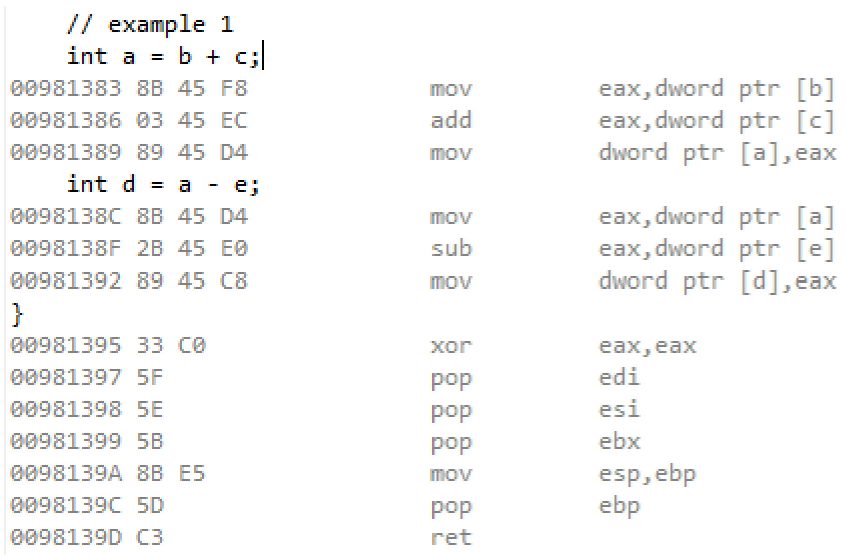


```
main()
{
    int b = 1;
    int c = 2;
    int e = 3;

    // example 1
    int a = b + c;
    int d = a - e;
}
```

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:



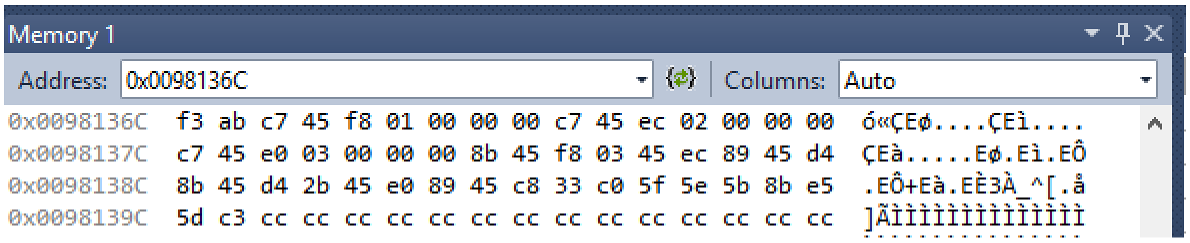
```
main()
{
00981350 55                   push        ebp
00981351 8B EC                mov         ebp,esp
00981353 81 EC FC 00 00 00    sub         esp,0FCh
00981359 53                   push        ebx
0098135A 56                   push        esi
0098135B 57                   push        edi
0098135C 8D BD 04 FF FF FF    lea         edi,[ebp-0FCh]
00981362 B9 3F 00 00 00       mov         ecx,3Fh
00981367 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
0098136C F3 AB                rep stos    dword ptr es:[edi]
    int b = 1;
0098136E C7 45 F8 01 00 00 00 mov         dword ptr [b],1
    int c = 2;
00981375 C7 45 EC 02 00 00 00 mov         dword ptr [c],2
    int e = 3;
0098137C C7 45 E0 03 00 00 00 mov         dword ptr [e],3
```



```
    // example 1
    int a = b + c;
00981383 8B 45 F8             mov         eax,dword ptr [b]
00981386 03 45 EC             add         eax,dword ptr [c]
00981389 89 45 D4             mov         dword ptr [a],eax
    int d = a - e;
0098138C 8B 45 D4             mov         eax,dword ptr [a]
0098138F 2B 45 E0             sub         eax,dword ptr [e]
00981392 89 45 C8             mov         dword ptr [d],eax
}
00981395 33 C0                xor         eax,eax
00981397 5F                   pop         edi
00981398 5E                   pop         esi
00981399 5B                   pop         ebx
0098139A 8B E5                mov         esp,ebp
0098139C 5D                   pop         ebp
0098139D C3                   ret
```

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:



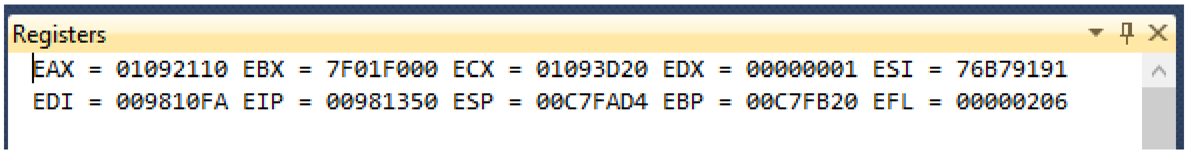
```
Address: 0x0098136C
0x0098136C  f3 ab c7 45 f8 01 00 00 00 c7 45 ec 02 00 00 00  ó«ÇEø....ÇEì....
0x0098137C  c7 45 e0 03 00 00 00 8b 45 f8 03 45 ec 89 45 d4  ÇEà.....Eø.Eì.EÔ
0x0098138C  8b 45 d4 2b 45 e0 89 45 c8 33 c0 5f 5e 5b 8b e5  .EÔ+Eà.EÈ3À_^[.å
0x0098139C  5d c3 cc cc cc cc cc cc cc cc cc cc cc cc cc cc  ]ÃÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
```

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

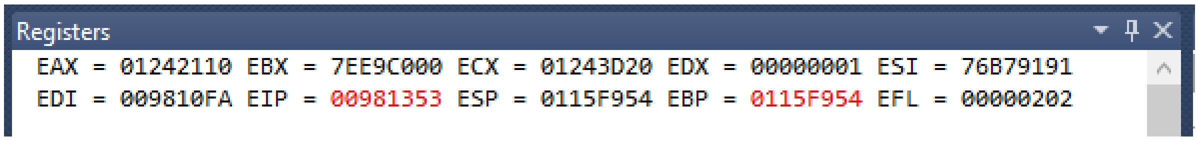
is the address for the instruction *f3*. The address for instruction *ab* is *0x0098136D*, and so on.

Finally, let's look at the registers:



```
EAX = 01092110 EBX = 7F01F000 ECX = 01093D20 EDX = 00000001 ESI = 76B79191
EDI = 009810FA EIP = 00981350 ESP = 00C7FAD4 EBP = 00C7FB20 EFL = 00000206
```

This is taken at the start of the main function.

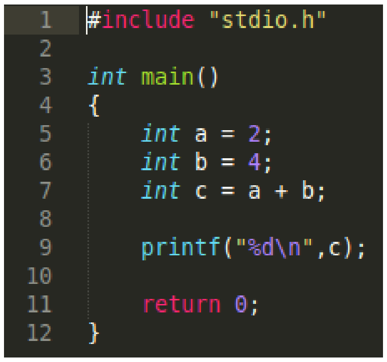


```
EAX = 01242110 EBX = 7EE9C000 ECX = 01243D20 EDX = 00000001 ESI = 76B79191
EDI = 009810FA EIP = 00981353 ESP = 0115F954 EBP = 0115F954 EFL = 00000202
```

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.

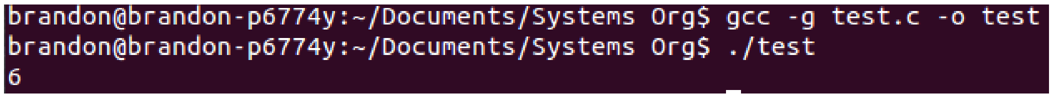
Linux OS

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:



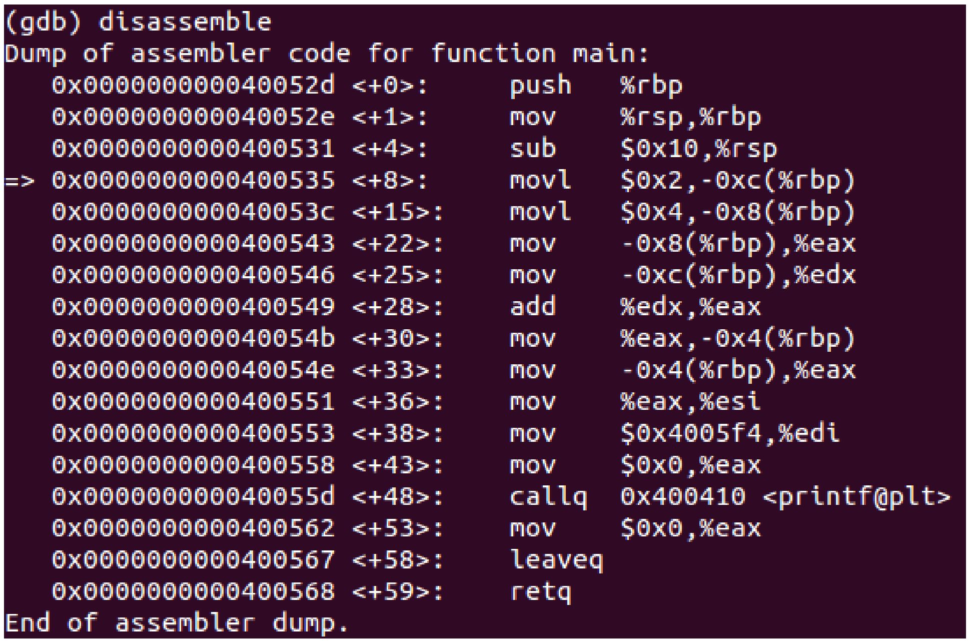
```c
#include "stdio.h"

int main()
{
    int a = 2;
    int b = 4;
    int c = a + b;

    printf("%d\n",c);

    return 0;
}
```

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



```
brandon@brandon-p6774y:~/Documents/Systems Org$ gcc -g test.c -o test
brandon@brandon-p6774y:~/Documents/Systems Org$ ./test
6
```

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



```
(gdb) disassemble
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     sub    $0x10,%rsp
=> 0x0000000000400535 <+8>:     movl   $0x2,-0xc(%rbp)
   0x000000000040053c <+15>:    movl   $0x4,-0x8(%rbp)
   0x0000000000400543 <+22>:    mov    -0x8(%rbp),%eax
   0x0000000000400546 <+25>:    mov    -0xc(%rbp),%edx
   0x0000000000400549 <+28>:    add    %edx,%eax
   0x000000000040054b <+30>:    mov    %eax,-0x4(%rbp)
   0x000000000040054e <+33>:    mov    -0x4(%rbp),%eax
   0x0000000000400551 <+36>:    mov    %eax,%esi
   0x0000000000400553 <+38>:    mov    $0x4005f4,%edi
   0x0000000000400558 <+43>:    mov    $0x0,%eax
   0x000000000040055d <+48>:    callq  0x400410 <printf@plt>
   0x0000000000400562 <+53>:    mov    $0x0,%eax
   0x0000000000400567 <+58>:    leaveq
   0x0000000000400568 <+59>:    retq
End of assembler dump.
```

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.