Lecture Notes – Computer Science 61: Systems Programming & Machine Organization

Justin Meretab, Mark Andrew Yao, Batsheva Moriarty

October 22, 2013

Midterm Statistics:

- **Average:** 75
- **Median:** 76
- **Std. Deviation:** 9
- **High Grade:** 93, indicates exam was too hard
- Overall, class did quite well

At the end of today, will have seen everything we need to get through the problem set. Next problem set is virtual memory, and we will see how system calls actually work.

When variables such as *a* or *b* are declared with no "magic prefixes" (%), they are global variables and most likely ints.

Comparison Functions:

- Do not appear to be modifying anything
- But they are! Condition codes are set by arithmetic and comparison computations

Condition codes – Hidden registers that change when we run comparison functions.

The codes combine many flags:

- **ZF:** zero flag
- **OF:** overflow flag
- **CF:** carry flag
- **SF:** sign flag
- **PF:** parody flag
- And a number more

Some comparison instructions:

- **cmpl** *a*, *b* ➔ **subl** *a*, *b* without changing *b*
- **testl** *a*, *b* ➔ **andl** *a*, *b* without changing *b*

Jump instructions:

- Change the way the instruction pointers move through the code
- Conditional jump instructions, jump if certain condition is met (e.g. jump if greater, jump if less, etc)
- If the condition isn't met, we continue on to the next instruction

- Different jumps are used for unsigned and signed numbers, because comparisons between unsigned integers and between signed integers are not always the same

Some jump instructions:

- **jge** ➔ jump if $b > a$, used for signed
- **ja** ➔ jump if above, used for unsigned
- **jb** ➔ jump if below, used for unsigned
- www.unixwiz.net/techtips/x86-jumps.html has list of jumps, what they mean, and what their flags are

Generally see comparison/test instructions immediately before jumps. This is because we are setting up the comparisons specifically to use the condition codes set when determining whether or not to jump. Occasionally, comparisons or tests will be followed by "sete" (set if equal) or something else that uses condition codes but is not a jump.

Statements with "dots" in front, e.g. ".size" are variables instead of functions

With twos complement representation of signed integers, arithmetic stays the same, but comparisons don't. For example:

0010 > 0001 (true always)
1000 > 0111 (true if unsigned, false if signed)

"movzbl": zero extend a byte quantity into a long quantity

- Move zero byte to long
- L is type of destination
- B is type of source—source is a byte
- Z means zero extend, ignore the sign bit and fill in the destination with 0s

"sete" : Moves the zero flag from the condition codes into a 1 byte destination

- 1 ➔ 00000001
- 0 ➔ 00000000

You can modify **%al** without modifying other parts of the **%eax** register

Assembly Code Practice:
f29.s: NULL is represented by the bit pattern 0 on x86 machines
f30.s: returns *a+x*

Remember: You cannot add two pointers. You can only add an int to a pointer. You *can* subtract a pointer from another pointer. This gives the offset.

f31.s: returns $a + (x << 2)$

a + (*x* << 2) is equivalent to:
char* *a*;
return &*a*[4 * *x*];

or

int* *a*;
return &*a*[*x*];

f32.s: *a* must be a byte because of "movz**b**l"

---BREAK---

Sign Extension:
When we extend signed values, we fill in the rest of the bits with "1"s if the value is negative (like an arithmetic shift). For example -1 = 1111 in 4-bit binary. When extended to 8-bit gives 11111111 = -1. If this were not the case, we would get 00001111 = 15 != -1.

"movzbl" → unsigned values

"movsbl" → signed values

"movsbl" & "movzbl" can inform us about the data types of the source and destination.

f33.s:
movl *a* %eax
movl (%eax) %eax
This is equivalent to returning **a* or *a*[0]

General x86 Assembly Indirection:
off(base, index, size) → dereference address at off + base + (index * size):

- off and index default to 0
- size can only be [1,2,4,8], default is 1
- index and size are useful for arrays, off is useful for structs

You can't tell with just a movl instruction whether a quantity is signed or unsigned.

When you see a jump backwards, it's most likely a loop.

%esp points to the top of the stack when the function enters. It stores the return address here and is 4 bytes long. After this, we have the parameter values, padded out to a minimum of 4 bytes. Parameters that you don't need are invisible to you.

Other Answers to Lecture Code:
f34.s: returns *a*[*x*], where *a* is of type unsigned char*
f35.s: returns *a*[*x*], where *a* is of type int*

f36.s: returns $a[x]$! = 0
f37.s: returns $*x$, where x is of type unsigned short*
f38.s: returns the sum of all the elements in an array with address a and size x
f39.s: identity function
f40.s: adds the first two parameters
f41.s: adds the first two parameters out of 8

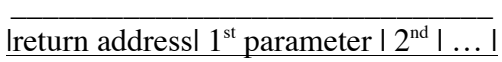
When you compile while loops into assembly, the condition moves to the bottom. By doing so, we only need to include one jump statement instead of two (if we had the condition at the top).

"decl" = decrement by 1

Comparison instructions only set condition codes. Arithmetic operations make a change and as a side effect, can also set condition codes. Other instructions such as "mov" don't change condition codes.

Get used to the idea that while loops can be re-stated as do-while loops. Also, compilers leverage the relationship between pointer arithmetic and array dereferencing depending on which method is faster.

Below is the state of the stack when we enter a function. The very top of the stack is where %esp points and holds the return address (4 bytes long). Other parameters are then laid out in the stack as if in a struct, where every value is padded to a minimum of 4 bytes.



Parameters that you don't need are invisible to you. f41.s has 8 parameters but because it only calls the first two, its assembly code is identical to that of f40.s which only has two parameters.