gcc x86 Assembly Quick Reference ("Cheat Sheet")

Instructions

Mnemonic	Purpose	Examples
mov *src,dest*	Move data between registers, load immediate data into registers, move data between registers and memory.	mov $4,%eax # Load constant into eax mov %eax,%ebx # Copy eax into ebx mov %ebx,123 # Copy ebx to memory address 123
push *src*	Insert a value onto the stack. Useful for passing arguments, saving registers, etc.	push %ebp
pop *dest*	Remove topmost value from the stack. Equivalent to "mov (%esp),*dest*; add $4,%esp"	pop %ebp
call *func*	Push the address of the next instruction and start executing func.	call print_int
ret	Pop the return program counter, and jump there. Ends a subroutine.	ret
add *src,dest*	*dest=dest+src*	add %ebx,%eax # Add ebx to eax
mul *src*	Multiply eax and *src* as unsigned integers, and put the result in eax. High 32 bits of product go into eax.	mul %ebx #Multiply eax by ebx
jmp *label*	Goto the instruction *label*:. Skips anything else in the way.	jmp post_mem mov %eax,0 # Write to NULL! post_mem: # OK here...
cmp *a,b*	Compare two values. Sets flags that are used by the conditional jumps (below). WARNING: compare is relative to *last* argument, so "jl" jumps if *b*<*a*!	cmp $10,%eax
jl *label*	Goto *label* if previous comparison came out as less-than. Other conditionals available are: jle (<=), jeq (==), jge (>=), jg (>), jne (!=), and many others.	jl loop_start # Jump if eax<10

Stack Frame

(example without %ebp or local variables)

Contents	**off esp**
caller's variables	12(%esp)
Argument 2	8(%esp)
Argument 1	4(%esp)
Caller Return Address	0(%esp)

```
my_sub: # Returns first argument
  mov 4(%esp), %eax
  ret
```

(example when using %ebp and two local variables)

Contents	**off ebp**	**off esp**
caller's variables	16(%ebp)	24(%esp)
Argument 2	12(%ebp)	20(%esp)
Argument 1	8(%ebp)	16(%esp)
Caller Return Address	4(%ebp)	12(%esp)
Saved ebp	0(%ebp)	8(%esp)
Local variable 1	-4(%ebp)	4(%esp)
Local variable 2	-8(%ebp)	0(%esp)

```
my_sub2: # Returns first argument
  push %ebp  # Prologue
  mov %esp, %ebp
  mov 8(%ebp), %eax
  mov %ebp, %esp  # Epilogue
  pop %ebp
  ret
```

Constants, Registers, Memory

Constants MUST be preceeded with "$". "$12" means decimal 12; "$0xF0" is hex. "$some_function" is the address of the first instruction of the function. WARNING: a bare "12", "0xF0", or "some_function" dereferences the expression like it was a pointer!
Registers MUST be preceeded with "%". "%eax" means register eax.
Memory access (use register as pointer): "(%esp)". Same as C "*esp".
Memory access with offset (use register + offset as pointer): "4(%esp)". Same as C "*(esp+4)".
Memory access with scaled index (register + another register * scale): "(%eax, %ebx, 4)". Same as C "*(eax+ebx*4)".

Registers

%esp is the stack pointer
%ebp is the stack frame pointer
Return value in %eax
Arguments are on the stack
Free for use (no save needed):
   %eax, %ebx, %ecx, %edx
Must be saved:
   %esp, %ebp, %esi, %edi

Common Errors

Segfault on innocent-looking code.
   Do you need to add "$" in front of a constant?
   Did you clean up the stack properly?
"

The Intel Software Developer's Manuals are incredibly long, boring, and complete--they give all the nitty-gritty details. Volume 1 lists the processor registers in Section 3.4.1. Volume 2 lists all the x86 instructions in Section 3.2. Volume 3 gives the performance monitoring registers. For Linux, the System V ABI gives the calling convention on page 39. Also see the Intel hall of fame for historical info. Sandpile.org has a good opcode table.

O. Lawlor, ffosl@uaf.edu
Up to: Class Site, CS, UAF