nasm x86 Assembly Quick Reference ("Cheat Sheet")

Instructions

Mnemonic	Purpose	Examples

mov *dest,src*	Move data between registers, load immediate data into registers, move data between registers and memory.	mov eax,4 ; Load constant into eax
		mov ebx,eax ; Copy eax into ebx
		mov [123],ebx ; Copy ebx to memory address 123

call *func*	Push the address of the next instruction and start executing func. For local functions, you don't have to say anything special. For functions defined in C/C++, say "extern *func*" first.	call print_int
ret	Pop the return program counter, and jump there. Ends a subroutine.	ret
add *dest,src*	*dest=dest+src*	add eax,ebx ; 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 edx.	mul ebx ; Multiply eax by ebx
imul *dest,src*	*dest=dest*src*	imul ecx,3

idiv *bot*	Divide eax by *bot*. Treats edx as high bits above eax, so set them to zero first!
	top = eax+(edx<<32)
	eax = *top*/*bot*
	edx = *top*%bot
		mov eax,73; top
		mov ecx,10; bot
		mov edx,0
		idiv ecx

jmp *label*	Goto the instruction *label*:. Skips anything else in the way.	jmp post_mem ...

Stack Frame

(example without ebp or local variables)

Contents	**off esp**
caller's variables	[esp+12]
Argument 2	[esp+8]
Argument 1	[esp+4]
Caller Return Address	[esp]

```
my_sub: # Returns first argument
  mov eax,[esp+4]
  ret
```

(example when using ebp and two local variables)

Contents	**off ebp**	**off esp**
caller's variables	[ebp+16]	[esp+24]
Argument 2	[ebp+12]	[esp+20]
Argument 1	[ebp+8]	[esp+16]
Caller Return Address	[ebp+4]	[esp+12]
Saved ebp	[ebp]	[esp+8]
Local variable 1	[ebp-4]	[esp+4]
Local variable 2	[ebp-8]	[esp]

```
my_sub2: # Returns first argument
  push ebp  # Prologue
  mov ebp, esp
  mov eax, [ebp+8]
  mov esp, ebp  # Epilogue
```

		post_mem:
cmp *a,b*	Compare two values. Sets flags that are used by the conditional jumps (below).	cmp eax,10
jl *label*	Goto *label* if previous comparison came out as less-than. Other conditionals available are: jle (<=), je (==), jge (>=), jg (>), jne (!=), and many others. Declare your label with a semicolon beforehand, just like in C/C++: "*label*:".	jl loop_start ; Jump if eax<10
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 *dest,*[esp] add esp,4"	pop ebp

```
  pop ebp
  ret
```

Constants, Registers, Memory

"12" means decimal 12; "0xF0" is hex. "some_function" is the address of the first instruction of a label.
Memory access (use register as pointer): "[esp]". Same as C "*esp".
Memory access with offset (use register + offset as pointer): "[esp+4]". Same as C "*(esp+4)".
Memory access with scaled index (register + another register * scale): "[eax + 4*ebx]". Same as C "*(eax+ebx*4)".

Subroutines are basically just labels. Here's how you declare labels for the linker:

- "extern some_function;" declares some_function as being outside the current file. You'll get a "symbol undefined" compile error if you call or jump to a label you never declare. In C++, be sure to declare the corresponding function as being 'extern "C"'!
- "global my_function;" exposes the label my_function so it can be called from outside. (In MASM, it's "PUBLIC my_function"). Again, your C++ prototype better be 'extern "C"'!

Differences with C:

- "010" means decimal ten in NASM, but *octal* eight in C/C++! Write octal by ending with letter 'o', like "10o".
- In NASM, you can write binary constants by ending with the letter 'b', like "mov eax,00101111b;".
- "1+(7<<13)/15" is evaluated at compile time, and it's a constant. "3+eax" can't be evaluated in NASM--it's not a constant.

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, ecx, edx
Must be saved:
  ebp, esp, esi, edi
ebx must be saved in a shared library, but is otherwise free for use.
8 bit: ah (high 8 bits) and al (low 8 bits)
16 bit: ax
32 bit: eax
64 bit: rax

Pretty much this same syntax is used by NASM (portable x86 assembler for Windows/Linux/whatever), YASM (adds 64-bit support to NASM), MASM (the Microsoft/Macro Assembler), and the official Intel documentation below. See the NASM documentation or MASM documentation for details on constants, labels and macros. Paul Carter has a good x86 assembly tutorial using the Intel syntax. The other, nastier syntax out there is the AT&T/GNU syntax, which I can't recommend. The machine code in all cases is identical.

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 in Section. 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.

Ralph Brown's Interrupt List is the aging but definitive reference for all PC software interrupt functions. See just the BIOS interrupts for interrupt-time code.

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