x86_64 NASM Assembly Quick Reference ("Cheat Sheet")

Here's the full list of ordinary integer x86 registers. The 64 bit registers are shown in red. "Scratch" registers any function is allowed to overwrite, and use for anything you want without asking anybody. "*Preserved*" registers have to be put back ("save" the register) if you use them.

Name	**Notes**	Type	**64-bit long**	**32-bit int**	**16-bit short**	**8-bit char**
rax	Values are returned from functions in this register.	scratch	rax	eax	ax	ah and al
rcx	Typical scratch register. Some instructions also use it as a counter.	scratch	rcx	ecx	cx	ch and cl
rdx	Scratch register.	scratch	rdx	edx	dx	dh and dl
rbx	*Preserved register: don't use it without saving it!*	*preserved*	*rbx*	*ebx*	*bx*	*bh and bl*
rsp	*The stack pointer. Points to the top of the stack (details coming soon!)*	*preserved*	*rsp*	*esp*	*sp*	*spl*
rbp	*Preserved register. Sometimes used to store the old value of the stack pointer, or the "base".*	*preserved*	*rbp*	*ebp*	*bp*	*bpl*
rsi	Scratch register used to pass function argument #2 in 64-bit Linux. In 64-bit Windows, a preserved register.	scratch	rsi	esi	si	sil
rdi	Scratch register and function argument #1 in 64-bit Linux. In 64-bit Windows, a preserved register.	scratch	rdi	edi	di	dil
r8	Scratch register. These were added in 64-bit mode, so they have numbers, not names.	scratch	r8	r8d	r8w	r8b
r9	Scratch register.	scratch	r9	r9d	r9w	r9b
r10	Scratch register.	scratch	r10	r10d	r10w	r10b
r11	Scratch register.	scratch	r11	r11d	r11w	r11b
r12	*Preserved register. You can use it, but you need to save and restore it.*	*preserved*	*r12*	*r12d*	*r12w*	*r12b*
r13	*Preserved register.*	*preserved*	*r13*	*r13d*	*r13w*	*r13b*
r14	*Preserved register.*	*preserved*	*r14*	*r14d*	*r14w*	*r14b*
r15	*Preserved register.*	*preserved*	*r15*	*r15d*	*r15w*	*r15b*

- A 64 bit linux machine passes function parameters in rdi, rsi, rdx, rcx, r8, and r9. Any additional parameters get pushed on the stack. OS X in 64 bit uses the same parameter scheme. Example function call:

  ```
  extern putchar
  mov rdi,'H' ; function parameter: one char to print
  call putchar
  ```

- Windows in 64 bit is quite different:
  - Win64 function parameters go in registers rcx, rdx, r8, and r9.
  - Win64 functions assume you've allocated 32 bytes of stack space to store the four parameter registers, plus another 8 bytes to align the stack to a 16-byte boundary.

    ```
    sub rsp,32+8; parameter area, and stack alignment
    extern putchar
    mov rcx,'H' ; function parameter: one char to print
    call putchar
    add rsp,32+8 ; clean up stack
    ```

  - Win64 treats the registers rdi and rsi as preserved.
  - Some function such as printf only get linked if they're called from C/C++ code, so to call printf from assembly, you need to include at least one call to printf from the C/C++ too.
  - If you use the MASM assembler, memory accesses must include "PTR", like "DWORD PTR [rsp]".
  - See NASM assembly in 64-bit Windows in Visual Studio to make linking work.
- In 32 bit mode, parameters are passed by pushing them onto the stack in reverse order, so the function's first parameter is on top of the stack before making the call. In 32-bit mode Windows and OS X compilers also seem to add an underscore before the name of a user-defined function, so if you call a function foo from C/C++, you need to define it in assembly as "_foo".

You can convert values between different register sizes using different mov instructions:

	Source Size				
	64 bit rcx	32 bit ecx	16 bit cx	8 bit cl	**Notes**

64 bit rax	mov rax,rcx	movsxd rax,ecx	movsx rax,cx	movsx rax,cl	Writes to whole register
32 bit eax	mov eax,ecx	mov eax,ecx	movsx eax,cx	movsx eax,cl	Top half of destination gets zeroed
16 bit ax	mov ax,cx	mov ax,cx	mov ax,cx	movsx ax,cl	Only affects low 16 bits, rest unchanged.
8 bit al	mov al,cl	mov al,cl	mov al,cl	mov al,cl	Only affects low 8 bits, rest unchanged.

Memory access:

C/C++ datatype	**Bits**	**Bytes**	**Register**	**Access memory**	**Allocate memory**
char	8	1	al	BYTE [ptr]	db
short	16	2	ax	WORD [ptr]	dw
int	32	4	eax	DWORD [ptr]	dd
long	64	8	rax	QWORD [ptr]	dq

Instructions (basically identical to 32-bit x86)

For gory instruction set details, read this per-instruction reference, or the full Intel PDFs: part 1 (A-M) and part 2 (N-Z).

Mnemonic	Purpose	Examples
mov *dest,src*	Move data between registers, load immediate data into registers, move data between registers and memory.	mov rax,4 ; Load constant into rax mov rdx,rax ; Copy rax into rdx mov rdx,[123] ; Copy rdx to memory address 123
push *src*	Insert a value onto the stack. Useful for passing arguments, saving registers, etc.	push rbp
pop *dest*	Remove topmost value from the stack. Equivalent to "mov *dest*, [rsp]; add 8,rsp"	pop rbp
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 *dest,src*	*dest=dest+src*	add rax,rdx ; Add rbx to rax
mul *src*	Multiply rax and *src* as unsigned integers, and put the result in rax. High 64 bits of product (usually zero) go into rdx.	mul rdx ; Multiply rax by rdx ; rax=low bits, rdx overflow
div *src*	Divide rax by *src*, and put the ratio into rax, and the remainder into rdx. Bizarrely, on input rdx must be zero, or you get a SIGFPE.	mov rdx,0 ; avoid error div rcx ; compute rax/rcx
shr *val,bits*	Bitshift a value right by a constant, or the low 8 bits of rcx ("cl"). Shift count MUST go in rcx, no other register will do!	add rcx,4 shr rax,cl ; shift by rcx
jmp *label*	Goto the instruction *label*:. Skips anything else in the way.	jmp post_mem mov [0],rax ; Write to NULL! post_mem: ; OK here...
cmp *a,b*	Compare two values. Sets flags that are used by the conditional jumps (below).	cmp rax,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. Also available in unsigned comparisons: jb (<), jbe (<=), ja (>), jae (>=).	jl loop_start ; Jump if rax<10

Constants, Registers, Memory

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

See sandpile.org for an opcode map.

O. Lawlor, lawlor@alaska.edu
Up to: Class Site, CS, UAF