x86-64 Assembly Language Summary

Dr. Orion Lawlor, last update 2019-10-14

These are all the normal x86-64 registers accessible from user code:

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.*	*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. Also used to pass function argument #2 in 64-bit Linux. String instructions treat it as a source pointer.	scratch	rsi	esi	si	sil
rdi	Scratch register. Function argument #1 in 64-bit Linux. String instructions treat it as a destination pointer.	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*

Functions return values in rax. How you pass parameters into functions varies depending on the platform:

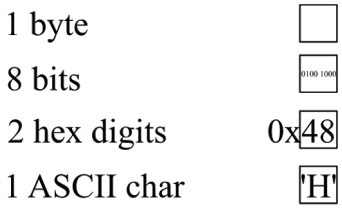
- A 64 bit x86 Linux machine, like NetRun:
  - Call nasm like: nasm -f elf64 yourCode.asm
  - Function parameters go in registers rdi, rsi, rdx, rcx, r8, and r9. Any additional parameters get pushed on the stack. OS X in 64 bit uses the same parameter scheme.
  - Linux 64 floating-point parameters and return values go in xmm0. All xmm0 registers are scratch.
  - If the function takes a variable number of arguments, like printf, rax is the number of floating point arguments (often zero).
  - If the function modifies floating point values, you need to align the stack to a 16-byte boundary before making the call.
  - Example linux 64-bit function call:

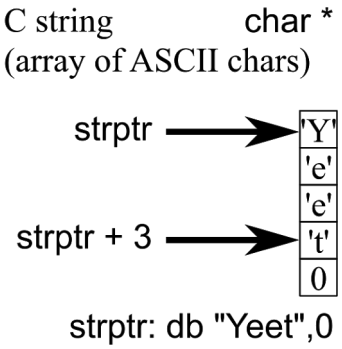
```
extern putchar
mov rdi,'H' ; function parameter: one char to print
call putchar
```

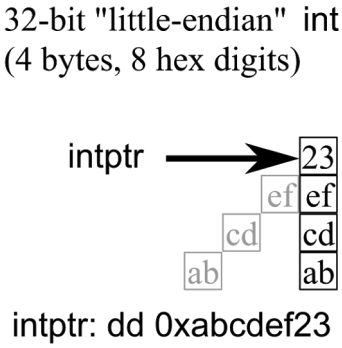
- Windows in 64 bit x86 is quite different:
  - Call nasm like: nasm -f win64 yourCode.asm
  - Win64 function parameters go in registers rcx, rdx, r8, and r9.
  - Win64 treats the registers rdi and rsi as preserved.
  - Win64 floating point registers xmm6-15 are preserved.
  - Win64 functions assume 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.

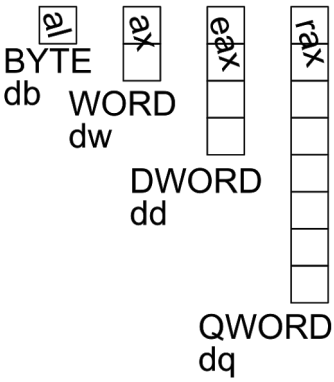
```
sub rsp,32+8; parameter area, and stack alignment
extern putchar
mov rcx,'H' ; function parameter: one char to print
call putchar
add rsp,32+8 ; clean up stack
```

  - 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 Windows MinGW or Visual Studio C++ compiler, "long" is the same size as "int", only 32 bits / 4 bytes even in 64-bit mode. You need to use "long long" to get a 64 bit / 8 byte integer variable on these systems. (Even on Windows, gcc, g++, or WSL makes "long" 64 bits, just like Linux or Mac or Java.) It's probably safest to #include <stdint.h> and refer to int64_t.
  - 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".









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)".

Different C++ datatypes get stored in different sized registers, and need to be accessed differently:

C/C++ datatype	**Bits**	**Bytes**	**Register**	**Access memory *ptr**	**Access Array ptr[idx]**	**Allocate Static Memory**
char	8	1	al	BYTE [ptr]	BYTE [ptr + 1*idx]	db
short	16	2	ax	WORD [ptr]	WORD [ptr + 2*idx]	dw
int	32	4	eax	DWORD [ptr]	DWORD [ptr + 4*idx]	dd
long [1]	64	8	rax	QWORD [ptr]	QWORD [ptr + 8*idx]	dq
float	32	4	xmm0	DWORD [ptr]	DWORD [ptr + 4*idx]	dd
double	64	8	xmm0	QWORD [ptr]	QWORD [ptr + 8*idx]	dq

[1] It's "long long" or "int64_t" on Windows MinGW or Visual Studio; but just "long" everywhere else.

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.

Registers can store either signed or unsigned values.

Signed	**Unsigned**	**Description**
int	unsigned int	In C/C++, int is signed by default.
signed char	unsigned char	In C/C++, char may be signed (default on gcc) or unsigned (default on Windows compilers) by default.

movsxd	movzxd	Assembly, **s**ign e**x**tend or **z**ero e**x**tend to change register sizes.
jo	jc	Assembly, **o**verflow is for signed values, **c**arry for unsigned values.
jg	ja	Assembly, jump **g**reater is signed, jump **a**bove is unsigned.
jl	jb	Assembly, jump **l**ess signed, jump **b**elow unsigned.
imul	mul	Assembly, imul is signed (and more modern), mul is for unsigned (and ancient and horrible!). idiv/div work similarly.

Normally, your assembly code lives in the code section, which can be read but not modified. When you declare static data, you need to put it in section .data for it to be writeable.

Name	**Use**	**Discussion**
section .data	r/w data	This data is initialized, but can be modified.
section .rodata	r/o data	This data can't be modified, which lets it be shared across copies of the program. In C/C++, global "const" or "const static" data is stored in .rodata.
section .bss	r/w space	This is automatically initialized to zero, meaning the contents don't need to be stored explicitly. This saves space in the executable.
section .text	r/o code	This is the program's executable machine code (it's binary data, not plain text--the Microsoft assembler calls this section ".code", a better name).

Before you can call some existing function, you need to declare that the function is "extern":

```
extern puts
call puts
```

If you want to define a function that can be called from outside, you need to declare your function "global":

```
global myGreatFunction
myGreatFunction:
    ret
```

When linking a program that calls functions directly like this, you may need gcc's "-no-pie" option, to disable the position-independent executable support.

Instructions

For gory instruction set details, read this per-instruction reference, or the uselessly huge Intel PDF (4000 pages!).

Instruction	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 [rdi],rdx  ; Copy rdx into the memory that rdi is pointing to`

`push src`	Insert a value onto the stack. Useful for passing arguments, saving registers, etc.	`push rbx`
`pop dest`	Remove topmost value from the stack. Equivalent to "mov *dest*, [rsp]; add 8,rsp"	`pop rbx`
`call func`	Push the address of the next instruction and start executing func.	`call puts`
`ret`	Pop the return program counter, and jump there. Ends a function.	`ret`
`add dest,src`	*dest=dest+src*	`add rax,rdx ; Add rdx to rax`
`imul dest,src`	*dest=dest*src* This is the signed multiply.	`imul rcx,4 ; multiply rcx by 4`
`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`

`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 (!=) Also available in unsigned comparisons: jb (<), jbe (<=), ja (>), jae (>=) And checking for overflow (jo) and carry (jc).	`jl loop_start  ; Jump if rax<10`

Standard Idioms

Looping over array elements, including the first-time test at startup:

```
; rdi: pointer to array.  rsi: number of elements
mov rcx,0 ; i, loop counter
jmp testFirst ; because rsi might be zero
startLoop:
      … work on QWORD[rdi+8*rcx], which is array[i] ...
      add rcx,1 ; i++
      testFirst:
      cmp rcx,rsi ; keep looping while i<n
      jl startLoop
```

(Try this in NetRun now!)

Allocating and deallocating memory:

Memory type	The Stack	The Heap	Static Data

Allocate *nBytes* of memory	`sub rsp,`*nBytes*	`mov rdi,`*nBytes*
		`extern malloc`
		`call malloc`
			`section .data`
			stuff`:`
			`  times` *nBytes* `db 0`

Pointer to the allocated data	`rsp`	`rax`	*stuff* or `lea rdx,[rel` *stuff*`]`

Deallocate the memory	`add rsp,`*nBytes*	`mov rdi,rax`
		`extern free`
		`call free`
			; Not needed

Properties	The stack is only 8 megs on most machines.	Slowest memory allocation: costs at least a half-dozen function calls.	Static data stays allocated until the program exits.

SSE Floating Point Instructions

There are at least three generations of x86 floating point instructions:

- fldpi, the original "floating point register stack", mostly limited to 32-bit machines now.
- addss xmm0,xmm2 the SSE instructions
- vmovss xmm0,xmm1,xmm2 the VEX-coded instructions

The SSE registers are named "xmm0" through "xmm15". The SSE instructions can be coded as shown below, or with a "v" in front for the VEX-coded AVX version, which allows the use of the 32-byte AVX "ymm" registers, and three-operand (destination, source1, source2) instruction format.

	Serial Single-precision (1 float)	**Serial Double-precision (1 double)**	**Parallel Single-precision (4 floats)**	**Parallel Double-precision (2 doubles)**	Comments
add	addss	addsd	addps	addpd	sub, mul, div all work the same way
min	minss	minsd	minps	minpd	max works the same way
sqrt	sqrtss	sqrtsd	sqrtps	sqrtpd	Square root (sqrt), reciprocal (rcp), and reciprocal-square-root (rsqrt) all work the same way
mov	movss	movsd	movaps *(aligned)* movups *(unaligned)*	movapd *(aligned)* movupd *(unaligned)*	Aligned loads are up to 4x faster, but will crash if given an unaligned address! The stack is always 16-byte aligned before calling a function, specifically for this instruction, as described below. Use "align 16" directive for static data.
cvt	cvtss2sd cvtss2si cvttss2si	cvtsd2ss cvtsd2si cvttsd2si	cvtps2pd cvtps2dq cvttps2dq	cvtpd2ps cvtpd2dq cvttpd2dq	Convert to ("2", get it?) Single Integer (si, stored in register like eax) or four DWORDs (dq, stored in xmm register). "cvtt" versions do truncation (round down); "cvt" versions round to nearest.
com	ucomiss	ucomisd	*n/a*	*n/a*	Sets CPU flags like normal x86 "cmp" instruction for unsigned, from SSE registers.
cmp	cmpeqss	cmpeqsd	cmpeqps	cmpeqpd	Compare for equality ("lt", "le", "neq", "nlt", "nle" versions work the same way). Sets all bits of float to zero if false (0.0), or all bits to ones if true (a NaN). Result is used as a bitmask for the bitwise AND and OR operations.
and	*n/a*	*n/a*	andps andnps	andpd andnpd	Bitwise AND operation. "andn" versions are bitwise AND-NOT operations (A=(~A) & B). "or" version works the same way.

The algebra of bitwise operators:

Instruction	C++ Operator	Useful to...
AND	&	Mask out bits (set other bits to zero) • 0=A&0 AND by 0's creates 0's, used to mask out bad stuff • A=A&~0 AND by 1's has no effect
OR	|	Reassemble bit fields • A=A|0 OR by 0's has no effect • ~0=A|~0 OR by 1's creates 1's
XOR	^	Invert selected bits • A=A^0 XOR by zeros has no effect • ~A = A ^ ~0 XOR by 1's inverts all the bits • 0=A^A XOR by yourself creates 0's--used for initialization • A=A^B^B XOR is its own inverse operation--used for cryptography
NOT	~	Invert all the bits in a number • ~0 All bits are set to one • ~A All the bits of A are inverted • A=~~A Inverting twice recovers the bits

Weird Instructions

x86 is ancient, and it has many weird old instructions. The more useful ones include:

div *src*	Unsigned divide rax by *src*, and put the ratio into rax, and the remainder into rdx.
	Bizarrely, on input rdx must be zero (high bits of numerator), or you get a SIGFPE.
		mov rax, 100 ; numerator
		mov rdx,0 ; avoid error
		mov rcx, 3 ; denominator
		div rcx ; compute rax/rcx

idiv *src*	Signed divide rax by the register *src*.
	rdx = rax % src
	rax = rax / src
	Before idiv, rdx must be a sign-extended version of rax, usually using cqo (Convert Quadword rax to Octword rdx:rax).
		mov rax, 100 ; numerator
		cqo ; sign-extend into rdx
		mov rcx, 3 ; denominator
		idiv 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 work!	add rcx,4 shr rax,cl ; shift by rcx

lea *dest*,[*ptr expression*]	Load Effective Address of the pointer into the destination register--doesn't actually access memory, but uses the memory syntax.	lea rcx,[rax + 4*rdx +12]

loop *jumplabel*	Decrement rcx, and if it's not zero, jump to the label.	mov rcx,10
		start:
		  add rax,7
		  loop start

lodsb	Load one char of a string: al = BYTE PTR [rsi++]	
stosb	Store one char of a string: BYTE PTR [rdi++] = al	
movsb	Copy one char of a string: BYTE PTR [rdi++] = BYTE PTR [rsi++]	
scasb	Compare the next char from string with register al: cmp BYTE PTR[rdi++], al	
cmpsb	Compare the next char from each of two strings: cmp BYTE PTR [rdi++], BYTE PTR [rsi++]	

rep *stringinstruction*	Repeat the string instruction rcx times. Only works with string instructions (lods, stos, cmps, scas, cmps, ins, outs)	mov al,'x'
		mov rcx,100
		mov rdi,bufferStart
		rep stosb

repne *stringinstruction*	Repeat the string instruction until the instruction sets the zero flag, or rcx gets decremented down to zero.	mov al,0
		mov rcx,-1
		mov rsi,stringStart
		repne lodsb

Debugging Assembly

error: parser: instruction expected
error: label or instruction expected at start of line

- This means you spelled the instruction name wrong.

error: invalid combination of opcode and operands

- Another way to say this: "there is no instruction taking those arguments". For example, there's a 64-bit "mov rax,rcx", and a 32-bit "mov eax,ecx", but there is no "mov rax,ecx".

It compiles but won't link: "undefined reference to foo**()**" <- note parenthesis!

- The C++ side needs to use "extern **"C"** long foo(void);" because you get this C++-vs-C link error if you leave out the extern "C".

It compiles but won't link: "undefined reference to _foo" <- note underscore!

- The assembly side may need to add underscores to match the compiler's linker names. This seems common on 32-bit machines.

It compiles but won't link: "undefined reference to foo"

- The assembly side needs to say "global foo" to get the linker to see it.

It compiles but then crashes with a SIGSEGV

- This means your code accessed a bad memory location
- Do you access [memory] one too many times? Accessing memory like "mov rax,[rdx]" when rdx is a number instead of a pointer will crash, accessing a low address like 0x0.
- Do you have write access to your [memory]? You may need to put your values into section .data
- Is your stack manipulation (push/pop) OK? It's common to leave extra garbage on the stack, which causes ret to pop the garbage and jump there; or to accidentally remove the return address with an extra pop. Every push must have exactly one pop.
- Is something trashing your pointer or counter registers? For example, if you call a function to print your output, it trashes all scratch registers. The fix is to push/pop scratch registers around function calls, or use preserved registers.
- Are you using 64-bit pointers? Some folks doing 32-bit accesses want to write "mov eax, DWORD [ecx]" which can crash--even if the value you're accessing is 32 bit, all your address arithmetic needs to be 64 bit, so write "mov eax, DWORD [rcx]".

It runs but gives the wrong output

- Do you need to access [thingy] instead of bare thingy? I've had programs that return the value *of* the pointer "mov rax,rdx" instead of the value the pointer is pointing to "mov rax,[rdx]"
- Do your loops run the right number of times? It's always tricky getting the last iteration correct.
- If you access arrays, are you multiplying by the size correctly? It's common to write to DWORD [rdi+1] instead of DWORD[rdi+4], which results in byte slices of the value you're after.

Using a debugger, like gdb, is very handy both for writing new code, and analysing existing programs even if you just have a compiled binary without source code. Here's my GDB reverse engineering cheat sheet.