

A Readers Guide to x86 Assembly

Purpose and Caveats

- This guide should give you enough background to read and understand (most) of the 64bit x86 assembly that gcc is likely to produce.
- x86 is a poorly-designed ISA. It's a mess, but it is the most widely used ISA in the world today.
 - It breaks almost every rule of good ISA design
 - Just because it is popular does not mean it's good
 - Intel and AMD have managed to engineer (at considerable cost) their CPUs so that this ugliness has relatively little impact on their processors' design (more on this later)
- There's a nice example here
 - http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

Registers

16bit	32bit	64bit	Description	Notes
AX	EAX	RAX	The accumulator register	These can be used more or less interchangeably
BX	EBX	RBX	The base register	
CX	ECX	RCX	The counter	
DX	EDX	RDY	The data register	
SP	ESP	RSP	Stack pointer	
BP	EBP	RBP	Points to the base of the stack frame	
	Rn	RnD	(n = 8...15) General purpose registers	
SI	ESI	RSI	Source index for string operations	
DI	EDI	RDI	Destination index for string operations	
IP	EIP	RIP	Instruction Pointer	
FLAGS			Condition codes	

Different names (e.g. ax vs. eax vs. rax) refer to different parts of the same register

Assembly Syntax

- There are two syntaxes for x86 assembly
- We will use the “gnu assembler (gas) syntax”, aka “AT&T syntax”. This different than “Intel Syntax”
- `<instruction> <src1> <src2> <dst>`

Details

Instruction Suffixes		
b	byte	8 bits
s	short	16 bits
w	word	16 bits
l	long	32 bits
q	quad	64 bits

Arguments	
%<reg>	Register
\$nnn	immediate
\$label	Label

MOV and addressing modes

- x86 does have loads and stores. It has mov

Instruction	Meaning
<code>movb \$0x05, %al</code>	$R[al] = 0x05$
<code>movl %eax, -4(%ebp)</code>	$mem[R[ebp] - 4] = R[eax]$
<code>movl -4(%ebp), %eax</code>	$R[eax] = mem[R[ebp] - 4]$
<code>movl \$LC0, (%esp)</code>	$mem[R[esp]] = \$LC0$ (a label)

Arithmetic

Instruction	Meaning
<code>subl \$0x05, %eax</code>	$R[\text{eax}] = R[\text{eax}] - 0x05$
<code>subl %eax, -4(%ebp)</code>	$\text{mem}[R[\text{ebp}] - 4] = \text{mem}[R[\text{ebp}] - 4] - R[\text{eax}]$
<code>subl -4(%ebp), %eax</code>	$R[\text{eax}] = R[\text{eax}] - \text{mem}[R[\text{ebp}] - 4]$

- Note that the amount of work per instruction varies widely depending on the addressing mode.
- A single instruction can include at least 6 additions (for the addressing mode), 2 memory loads, and one memory store.

Branches

- x86 uses condition codes for branches
 - Arithmetic ops set the flags register
 - carry, parity, zero, sign, overflow

Instruction	Meaning
<code>cmpl %eax %ebx</code>	Compute <code>%eax - %ebx</code> , set flags register
<code>jmp <location></code>	Unconditional branch to <code><location></code>
<code>je <location></code>	Jump to <code><location></code> if the equal flag is set (e.g., the two values compared by <code>cmp</code> are equal)
<code>jg, jge, jl, gle, jnz, ...</code>	jump {>, >=, <, <=, != 0,}

Stack Management

Instruction	High-level meaning	Equivalent instructions
<code>pushl %eax</code>	Push <code>%eax</code> onto the stack	<code>subl \$4, %esp;</code> <code>movl %eax, (%esp)</code>
<code>popl %eax</code>	Pop <code>%eax</code> off the stack	<code>movl (%esp), %eax</code> <code>addl \$4, %esp</code>
<code>leave</code>	Restore the callers stack pointer.	<code>movl %ebp, %esp</code> <code>pop %ebp</code>

Function Calls

Instruction	High-level meaning
call <label>	Call the function. Push the return address onto the stack.
ret	Jump to the return address and pop it from the stack.
leave	Restore the callers stack pointer.

- Arguments are passed on the stack
 - Use push to put them there.
- Return value in register A (eax, rax, etc)

```
int foo(int x,  
        int y,  
        int z);  
...  
d = foo(a, b, c);  
  
push c  
push b  
push a  
call foo  
mov  %eax, d
```

Accounting for Work

```
addq -4(%rax), -6(%rbx)
```

```
addq %rax, %rbx
```

```
movl %eax, 4(%ebx)
```

```
t1 = %rax-4
```

```
t2 = mem[t1]
```

```
t3 = %rbx - 6
```

```
t4 = mem[t1]
```

```
t5 = t4 + t2
```

```
mem[t1] = t5
```

```
%rbx=%rbx+%rax
```

```
t1 = %ebx + 4
```

```
mem[t1] = %eax
```

type	count
mem	3
arithmetic	3

type	count
mem	0
arithmetic	1

type	count
mem	1
arithmetic	1