

C Programming Assembler

C Programming & Basic Unix

Before Read: Compilation of C Programming Language

(link: <https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>)

```
----- hello.s
.file "hello.c"
.section .rodata
.LC0:
.string "hello world"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
call puts
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
.section .note.GNU-stack,"",@progbits

----- hello.c
#include <stdio.h>
int main()
{
    printf("hello world\n");
    return 0;
}
```

x86 Assembly

1. Intel Syntax: op dst, src <ex> movl eax, 1 / addl eax, ebx
2. AT&T (GAS) Syntax: op src, dst <ex> movl 1, %eax / addl %eax, %ebx

Intel Architecture Reference Manuals

Intel® 64 and IA-32 Architectures Software Developer Manuals

These manuals describe the architecture and programming environment of the Intel® 64 and IA-32 architectures. Electronic versions of these documents allow you to quickly get to the information

[intel https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html](https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html)



Assembler Directives

`.text`

- Instructions/program code are placed here

`.data`

- Initialized read/write data are defined here

`.section .rodata`

- Initialized read only data are defined here

`.comm symbol, length, alignment`

- Uninitialized data are allocated in the `bss` section

`.local name`

- Makes a name a local symbol
- `.lcomm = .local + .comm`

`.ascii "string" ...`

- Define strings without the terminal zero

`.string "string" ...`

- Define null-terminated strings

`.byte, .int, .long, .quad`

- Define integer numbers

`.double, .float`

- Define floating point numbers

`.align`

- Pad the location counter to a particular storage boundary.

`.size`

- Set the size associated with a symbol

AT & T Assembly Format

General format:

- operation source, destination
- e.g. `movb $0x05, %al`

Operation Suffixes

- Instructions are suffixed with
`b`: byte, `s`: short (2 byte int or 4 byte float), `w`: word (2 byte), `l`: long(4 byte int or 8 byte float), `q`: quad (8 byte), `t`: ten byte (10 byte float)

Prefixes

- `%` for registers, `$` for constant numbers (literals/immediates)

Literals

Integers

- 24, 0b1010, 0x4a, 074

Floating point numbers

- 0.1, 1.2e3

Strings

- "abc\n"

Characters

- 'a', '\n'

Addressing Operand

Syntax

- `segment:displacement(base register, offset register, scalar multiplier)`
- `base register + offset register * scalar multiplier + displacement` (ignoring segment)
- Either or both of numeric parameters can be omitted
- Either of the register parameters can be omitted

Example

- `movl -5(%rbp, %rsi, 4), %eax` #load `[rbp + rsi * 4 - 5]` into `eax`
- `movl -5(%rbp), %eax` #load `[rbp - 5]` into `eax`
- `leaq 8(%rbx, %rcx, 2), %rax` #load `rcx * 2 + rbx + 8` into `rax`

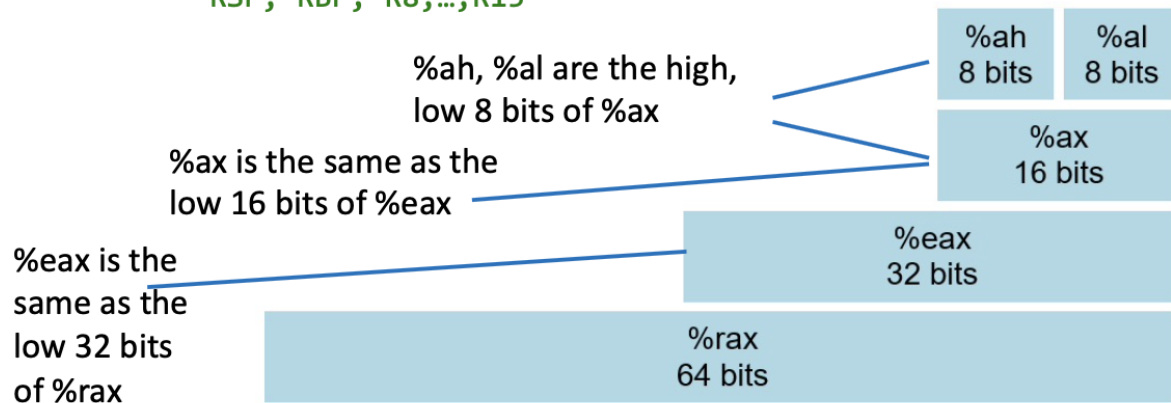
Registers

8 bit: AH, AL, BH, BL, CH, CL, DH, DL, R8B,...,R15B

16 bit: AX, BX, CX, DX, SI, DI, SP, BP, R8W,...,R15W

32 bit: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D,...,R15D

64 bit: RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8,...,R15



Move and Stack Manipulation Instructions

`mov src, dst`

- At least one of src or dst must be a register
- e.g. `movl $0, %eax`

`push src`

- e.g. `pushl %eax`
- eqv: `subq $4, %rsp; movl %eax (%rsp)`

`pop dst`

- e.g. `popq %rax`
- eqv: `movq (%rsp) %rax; addq $8, %rsp`

`leave`

- `movq %rbp, %rsp`
`popq %rbp`

Arithmetic Instructions

add src, dst

- e.g. `addq $2, %rax` # `rax = rax + 2`

sub src, dst

- e.g. `subq %rbx, %rax` # `rax = rax - rbx`

mul arg

- e.g. `mulw %bx` # `bx * ax -> dx` (high 16bits), `ax` (lower 16bits)

div arg

- e.g. `divl %ebx`
`(edx * 2^32 + eax) / ebx -> eax,`
`(edx * 2^32 + eax) mod ebx -> edx`

Logical Instructions

and src, dst

- e.g. `andl $0xf, %eax` # `eax &= 0xf`

or src, dst

- e.g. `orl $0xf, %eax` # `eax |= 0xf`

not dst

- e.g. `notq %rax` # `rax = ~rax`

xor src, dst

- e.g. `xorw %ax, %ax` # `ax = ax xor ax`

Flags

ZF (zero flag)

- Set if the result is 0

SF (sign flag)

- Set if the MSB of the result is 1

OF (overflow flag)

- Set when overflow occurred ($8 + 8 \rightarrow 16$ in 4bit)
- Positive num op Positive num \rightarrow Negative num
- Negative num op Negative num \rightarrow Positive num

Compare and Branch Instructions

`cmp arg1 arg2`

- `cmpq $2, %rax`
ZF = iff `%rax - 2 == 0`
SF = iff MSB of `%rax - 2 == 1`
OF = iff overflow occurs

`test arg1 arg2`

- `testq $5, %rax`
ZF = iff `%rax & 5 == 0`
SF = iff MSB of `%rax & 5 == 1`

`JE, JZ, JNE, JNZ, JG, JGE, JL, JLE`

- `jne label`
jump if `ZF == 0`
- `jg label`
jump if `SF == OF` and `ZF == 0`
 - `cmp -4, 4` \Rightarrow `OF=1, SF=1` on 4bit machines
 - `cmp 2, -1` \Rightarrow `OF=1, SF=0` on 4bit machines

Call Instructions

call label

- e.g. `call 0x1234`
- Equivalent to
 - `pushq %rip`
 - `movq 0x1234, %rip`

ret

- e.g. `ret`
- Equivalent to
 - `popq %rip`