x86-64 Reference Sheet

David Broman, KTH Royal Institute of Technology
Version 0.2, December 1, 2020

Operands how operands work

Forms	Examples	
1. Registers:	rax, edx, di	
2. Immediate operands:		
Decimal form:	-721	
Hexadecimal form:	0x21af	21afh
Binary form:	0b11010111	1101_0111b
3. Memory operands		
[num]	[0x1232]	
[reg]	[rbx]	
[reg + reg*scale]	[rbx + rcx * 4]	
[reg + num]	[rbx + 9]	
[reg1 + reg2*scale + num]	[rbx + rcx * 8 + 2]	scale ∈ {1, 2, 4, 8}

Only 64-bit registers are allowed in memory operations

"sign extension/ extend the sign of a binary number" means:将一个二进制数阔位，在不改变其±、绝对值的情况下。举例：
by padding the left side with ones, a negative number will be successfully extended;
by padding the left side with zeros, a positive number will be successfully extended.

Integer Instructions (subset)

Inst	Operands	Description	Semantics
add	op1, op2	Addition	op1RM := op1RM + op2RMI
and	op1, op2	Bitwise AND	op1RM := op1RM & op2RMI
call	op1	Procedure call	rsp := rsp - 8; [rsp] := PC; PC := op1RMI
cdq		Sign extend 32-bit reg.	edx:eax := signext(eax)
cmovC	op1, op2	Conditional move	op1R := op1RM if condition C
cmp	op1, op2	Compare	op1RM - op2RMI
cqo		Sign extend 64-bit reg.	rdx:rax := signext(rax)
dec	op1	Decrement by 1	op1RM := op1RM - 1
div	op1	Unsigned division	rax (rdx) := rdx:rax / op1RM
idiv	op1	Signed division	quotient(remainer) rax (rdx) := rdx:rax / op1RM
imul	op1	Signed multiplication	rdx:rax := rax * op1RM
inc	op1	Increment by 1	op1RM := op1RM + 1
jC	op1	Conditional jump	PC := op1I if C else next
jmp	op1	Unconditional jump	PC := op1RMI
mov	op1, op2	Move	op1RM := op2RMI
movsx	op1, op2	Move w. sign-extension	op1R := signext(op2RM)
movzx	op1, op2	Move w. zero-extension	op1R := zeroext(op2RM)
mul	op1	Unsigned multiplication	rdx:rax := rax * op1RM
neg	op1	Signed negation	op1RM := 0 - op1RM
not	op1	Bitwise NOT operation	op1RM := ~op1RM
or	op1, op2	Bitwise OR (inclusive)	op1RM := op1RM | op2RMI
pop	op1	Pop from stack	op1RM := [rsp]; rsp := rsp + 8
push	op1	Push on stack	rsp := rsp - 8; [rsp] := op1RMI
ret		Return from procedure	PC := [rsp]; rsp := rsp + 8
sar	op1, op2	Shift arithmetic right	op1RM := op1RM >>> op2RI
setC	op1	Set byte on condition	op1R := 1 if condition C is true, else 0
shl	op1, op2	Shift logical left	op1RM := op1RM << op2RI
shr	op1, op2	Shift logical right	op1RM := op1RM >> op2RI
sub	op1, op2	Subtraction	op1RM := op1RM - op2RMI
xchg	op1, op2	Exchange (swap)	t := op1RM; op1RM = op2RM; op2RM := t (t is temp)
xor	op1, op2	Bitwise XOR (exclusive)	op1RM := op1RM ^ op2RMI

Notes

读作：1号操作数（允许reg、mem、imm的形式）

- Meaning of op1RMI: op1 = the operand, R = can be a register, M = can be a memory operand, and I = can be an immediate op. 读作：1号操作数（允许reg、mem的形式） Example: op1RM allows reg and mem, but not immediate value.
- For shl, sar, sal, op2 is either intermediate or the cl register.
- For jump and call instructions, a label can be used as operand.
- Jump instructions check the flags set by the previous instruction. Key flags are s (sign), z (zero), c (carry), and o (overflow).
- The setC instruction requires op1 to be an 8-bit register.
- For instructions jC, setC, and cmovC, the C options are: e = equal, ne = not equal, l = less than (signed), le = less or equal (signed), g = greater than (signed), ge = greater or equal (signed), a = above (unsigned), b = below (unsigned), above and equal (unsigned), and be = below or equal (unsigned).
  Example: setne al ; set register al if not equal
  jle foo ; jump to label foo if less or equal
- Binary instructions (e.g., add) cannot use two memory operands.
- div and idiv stores the result in rax and the remainder in rdx
- Memory operations may be proceeded by a size specifier.
  Example: mov qword [ecx], 0x7f
  byte = 8 bits, word = 16 bits, dword = 32 bits, and qword = 64 bits

Registers

#	64-bit	32-bit	16-bit	8-bit	Note	Function
0	rax	eax	ax	al	-	Accumulator
3	rbx	ebx	bx	bl	Callee saved	Base
1	rcx	ecx	cx	cl	Param 4	Counter
2	rdx	edx	dx	dl	Param 3	Data
6	rsi	esi	si	sil	Param 2	Source Index
7	rdi	edi	di	dil	Param 1	Destination Index
5	rbp	ebp	bp	bpl	Callee saved	Base Pointer
4	rsp	esp	sp	spl	-	Stack Pointer
8	r8	r8d	r8w	r8b	Param 5	-
9	r9	r9d	r9w	r9b	Param 6	-
10	r10	r10d	r10w	r10b	-	-
11	r11	r11d	r11w	r11b	-	-
12	r12	r12d	r12w	r12b	Callee saved	-
13	r13	r13d	r13w	r13b	Callee saved	-
14	r14	r14d	r14w	r14b	Callee saved	-
15	r15	r15d	r15w	r15b	Callee saved	-

- The # column can be used for numbered registers, e.g. r3 = rbx. Note that in NASM, line `%use altreg` is needed for numbered registers.
- Reg eax represents the least significant bits (LSB) of rax, ax the LSB of eax, and al is the LSB of ax. Same pattern holds for all other registers.

Calling Conventions about registers

- Integer arguments: rdi, rsi, rdx, rcx, r8, r9
- Floating-point arguments: xmm0 - xmm7
- Additional arguments pushed on stack, right to left, removed by caller
- Callee saved registers: rbp, rbx, r12, r13, r14, r15
- Integer return register(s): rax or rdx:rax
- Floating-point number return register(s): xmm0 or xmm1:xmm0

Directives and Comments about registers

.text	Assembly instructions (code section)
.data	Initialized data (data section)
.bss	Uninitialized data (bss section)
.global label	Make the label visible to the outside
.extern label	Assumes that the label is defined elsewhere
; comment	Line comments start with a semicolon

Data Reservation

Reservations in the .data section

db	0x1f	Initializes 1 byte
db	1, 78, 0x2f	Initializes 3 bytes
db	'A', 10	Initializes 2 bytes
db	"Hello", 0	Initializes 6 bytes
dw	0x1821	Initializes a 16-bit word (2 bytes)
dd	0x1782ab12	Initializes a 32-bit word (4 bytes)
dq	0x8, 0x3	Initializes two 64-bit words (16 bytes)

Reservations in the .bss section

resb	88	Reservers 88 uninitialized bytes
resw	10	Reservers 10 uninitialized 16-bit words (20 bytes)
resd	10	Reservers 10 uninitialized 32-bit words (40 bytes)
resq	10	Reservers 10 uninitialized 64-bit words (80 bytes)

Hello world (NASM)

```
        global  main              ; makes main available
        extern  printf            ; assumes printf is defined
                                  ;   in another linked lib
        section .data             ; .data = start of data
str1:   db      "Hello world: %d, %x", 10, 0
        section .text             ; .text = start of code
main:                             ; start of program label
        mov     rdi, str1         ; address of label str1
        mov     rsi, -782         ; second argument in rsi
        mov     rdx, 0x78912ab    ; third argument in rdx
        xor     rax, rax          ; set eax to zero
        call    printf            ; call printf
        ret                       ; exit program
```

Assemble, link, and run under Linux

```
nasm -felf64 hello.asm
gcc -no-pie hello.o
./a.out
```