CoE 163

Computing Architectures and Algorithms

03c: x86 Assembly Reference

SYNTAX

Unlike MIPS, the destination is implied to contain one of the operands to instructions that require at least two operands

x86 has two major syntax flavors: AT&T and Intel.

- AT&T is common in Linux systems.
- Intel is common in Windows systems.

For the rest of the discussion, we will be using Intel syntax - more specifically the NASM flavor.

REGISTERS

8 bytes	rax rbx rcx rdx rsi rdi rbp rsp r8 r9 r10 r11 r12 r13 r14 r15
4 bytes	eax ebx ecx edx esi edi ebp esp r8d r9d r10d r11d r12d r13d r14d r15d
2 bytes	ax bx cx dx si di bp sp r8w r9w r10w r11w r12w r13w r14w r15w
1 byte (upper)	ah bh ch dh
1 byte (lower)	al bl cl dl sil dil bpl spl r8b r9b r10b r11b r12b r13b r14b r15b

REGISTER NAMES

Register	(Historical) Name
`ax`	accumulator
`bx`	base
`cx`	counter
`dx`	data (ax extension)
`si`	source index (strings)
`di`	destination index (strings)
`sp`	stack pointer
`bp`	base pointer

STATIC DATA

Memory data can be initialized in the `.data` section of the code. Each variable should be on a separate line and denoted as follows:

```
<name> <size> <data>
```

Example

```
section .data

var DB 64      ; byte
abc DQ ?       ; uninit qword
x DD 27        ; dword
z DD 1, 2, 3 ; “array” of 3 dwords
```

STATIC DATA

Arrays and strings can also be initialized in the `.data` section.

Example

```
section .data

bytes TIMES 7 DB ?   ; 7 uninit bytes
abx TIMES 25 DQ 0    ; 25 qwords inited to 0
mystr DB ‘hello’, 0  ; null-terminated string
```

MEMORY ADDRESSING

Memory addresses are denoted by square brackets [].

Up to two registers and one signed constant can be added together to form a memory address. One of the registers can be pre-multiplied by 2, 4, or 8.

Example

```
mov [rax], rbx
mov rax, [rsi - 8]  ; added by signed -8
mov [rax + rbx], 12
mov [2 * rbx + rax], FFh
```

MEMORY ADDRESSING

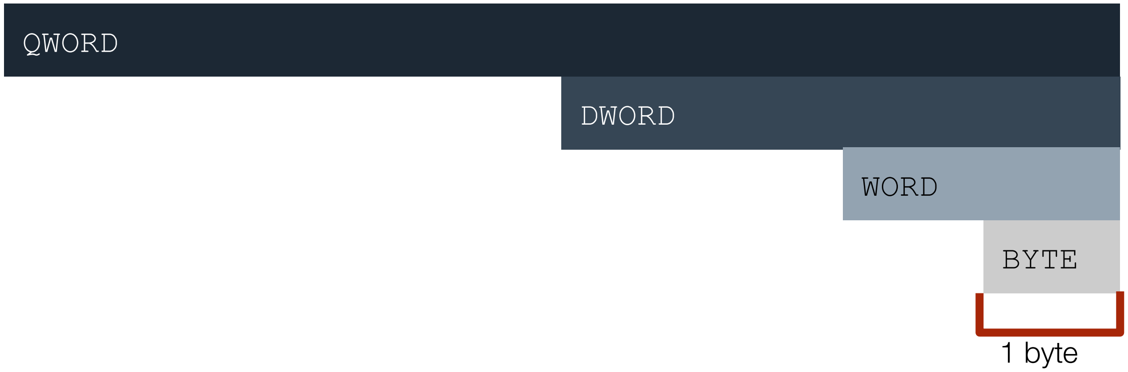
The example below shows some of the incorrect ways to compute memory addresses.

Example

```
mov [rax + rbx + rcx], -FEh
mov rsi, [rbx - rcx]
```

SIZE DIRECTIVES

Size directives are used to label how many bytes of the content should be used for an operation. This is required to infer how many bytes to get from a memory cell.



BASIC INSTRUCTIONS

x86_64 INSTRUCTIONS

```
mov <R/M/I>, <R/M/I>
```

Move contents of B into A.

Note that you cannot move contents directly from one memory cell to another using this instruction. A `mov` to a register should be done first.

Example

```
mov QWORD rax, 5
mov QWORD [rax + Fh], 5
mov DWORD rbx, rax
mov DWORD rbx, [rax + 5]
```

x86_64 INSTRUCTIONS

```
push <R/M/C>
```

Push operand into the stack.

The register `rsp` is decremented by 8 first, and then the *8-byte content* of the operand is moved into the address pointed to by `rsp`.

Example

```
push rax
push [rax + 5]
push 0
```

x86_64 INSTRUCTIONS

```
pop <R/M>
```

Pop something from the stack.

The *8-byte content* at the address pointed to by `rsp` is saved into the operand first, and then the `rsp` is decremented by 8.

Example

```
pop rax
pop [rax + 5]
```

x86_64 INSTRUCTIONS

```
lea <R>, <M>
```

Get memory address of a cell.

This operand loads the *effective address* of the second operand, not its contents. R will contain a “pointer” to the memory cell M.

Example

```
lea QWORD rax, [rdx + 4 * rbx]
lea QWORD rsi, [rax]
```

x86_64 INSTRUCTIONS

```
add <R/M/C>, <R/M/C>
sub <R/M/C>, <R/M/C>
```

Add or subtract operands

These instructions add or subtract the two operands and store the result into first operand. Like `mov`, only one of the operands may be a memory location.

Example

```
add rax, rbx
add WORD [rax + 3], Fh
sub WORD Fh, [rbx]
```

x86_64 INSTRUCTIONS

```
inc <R/M>
dec <R/M>
```

Increment or decrement an operand by 1.

Example

```
inc rdx
inc DWORD [rax]
dec DWORD [rdx + FFh]
```

x86_64 INSTRUCTIONS

```
imul <R>, <R/M>
imul <R>, <R/M>, <C>
```

Multiply two numbers and store the result in the first (register) operand.

The two-operand version multiplies the two operands while in the three-operand version, the second and third operands are multiplied.

Example

```
imul rbx, rax
imul QWORD rax, [rbx + 9]
imul rax, rbx, FFh
imul QWORD rax, [4 * rsi], EFh
```

x86_64 INSTRUCTIONS

```
idiv <R/M>
```

Divide two numbers with the operand as divisor.

The divided should be stored in `rdx` (MSB) and `rax` (LSB). The quotient is stored in `rax` while the remainder is in `rdx`.

Example

```
mov rdx, 0
mov rax, deadbeefh
idiv 0ff1ceh
```

x86_64 INSTRUCTIONS

```
and <R/M/C>, <R/M/C>
or <R/M/C>, <R/M/C>
xor <R/M/C>, <R/M/C>
```

Perform various bitwise operations.

These instructions operate on the two operands and store the result into first operand. Like `mov`, only one of the operands may be a memory location.

Example

```
and rax, rbx
or QWORD [rax], 1h
xor FEE7h, rdx
```

x86_64 INSTRUCTIONS

```
not <R/M>
```

Invert bits of the operand contents.

Example

```
not FEFh
not QWORD [rax + 16]
```

x86_64 INSTRUCTIONS

```
neg <R/M>
```

Negate the operand contents.

The negation works using two’s complement.

Example

```
neg rdx
neg QWORD [rbx + 9]
```

x86_64 INSTRUCTIONS

```
shl <R/M>, <R/C>
shr <R/M>, <R/C>
```

Shift bits of first operand by some amount.

These instructions shift the bits by some 8-bit amount modulo 64 specified in the second operand. This operand can either be an 8-bit constant or the `cl` register.

Example

```
mov cl, 3
shl rax, cl
shr QWORD [rbx], 3
```

CONTROL FLOW INSTRUCTIONS

CONTROL FLOW

Unlike MIPS, branch instructions are merged into jump instructions.

Conditional jumping is checked based on a register named *machine status word* (MSW), which is changed based on the last arithmetic operation among other things.

x86_64 INSTRUCTIONS

```
jmp <L>
```

Move the program counter some memory location.

Example

```
loop: mov rax, Fh
      jmp loop
```

x86_64 INSTRUCTIONS

```
je <L>    ; jump if equal
jne <L>   ; jump if not equal
jz <L>    ; jump if previous is zero
```

Move the program counter some memory location depending on the MSW.

Example

```
mov rax, 1000
mov rbx, 500
cmp rax, rbx
je loop
```

x86_64 INSTRUCTIONS

```
jg <L>    ; jump if greater than
jge <L>   ; jump if greater than or equal to
jl <L>    ; jump if less than
jle <L>   ; jump if less than or equal to
```

Move the program counter some memory location depending on the MSW.

Example

```
mov rax, 1000
mov rbx, 500
cmp rax, rbx
jle loop
```

x86_64 INSTRUCTIONS

```
cmp <R/M>, <R/M/C>
```

Compare contents of two operands.

The MSW is set depending on the result of this comparison, and is equivalent to `sub` with the result discarded. This is usually used before a J-type instruction.

Example

```
      mov rax, 10
loop: dec 1
      cmp rax, 0
      jg loop
```

SUBROUTINE INSTRUCTIONS

SUBROUTINES

Subroutine handling is similar to MIPS, which starts with pushing the current program counter (PC) to the stack and then jumping to the start of the subroutine.

When the subroutine ends, it is imperative to clear the stack so that the top of it contains the previous PC.

x86_64 INSTRUCTIONS

```
call <L>
```

Call subroutine starting at some location

This is similar to `jmp` except that the current PC is `push`ed to the stack and then a `jmp` is performed to the label.

Example

```
       mov rax, 3
       mov rbx, 5
       call addme

addme: add rax, rbx
```

x86_64 INSTRUCTIONS

```
ret
```

Exit from subroutine

This is similar to `jmp` except that the stack is `pop`ed and then a `jmp` is performed to the address pointed by it as if it is the PC.

Example

```
addme:  add rax, rbx
        ret
```

CALLING CONVENTION: SYSTEM V (LINUX)

x86_64 CALLING CONVENTION

- For easier tracking of the conventions, you can write a *prologue* and an *epilogue* around the `call` instruction, and at the start and end of the subroutine.
- Note that the stack grows down, so the SP should decrement when something is pushed into the stack.

x86_64 CALLER RULES

- Push caller-saved register values into the stack - `r8`, `r9`, and any registers used as parameters to a subroutine.
- Place the first six parameters into these registers in order: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`.
- The seventh and other parameters should be pushed into the stack starting from the last one.
- Once everything has been prepared, use `call` to go to the subroutine.

x86_64 CALLER RULES

- Once the subroutine returns, the return value should be in `rax`
- Next, pop any parameters from the stack.
- Finally, restore caller-saved registers values by popping from the stack.

x86_64 SUBROUTINE RULES

- Allocate space for local variables by decrementing the stack pointer (SP) by the size needed.
- Push callee-saved register values into the stack - `rbx`, `rbp`, `r12-r15`.
- Once everything has been prepared, the subroutine can now proceed.
- Once the subroutine is finished, the return value should be saved in `rax`.

x86_64 SUBROUTINE RULES

- Next, restore callee-saved registers values by popping from the stack.
- Deallocate space for local variables by incrementing the stack pointer to its original value before the subroutine started.
- Finally, use `ret` to exit the subroutine.

x86_64 SAVED REGISTERS

Caller-saved (call-clobbered)

Scratch registers that the one calling the subroutine should "save" to the stack if the relevant register contents are useful.

```
rax rcx rdx rsi rdi r8
r9 r10 r11
```

Callee-saved (call-preserved)

Preserved registers that the subroutine should either not change or "save" its original contents if they will be used.

The stack pointer `rsp` is saved when a `call` is used.

```
rbx rsp rbp r12 r13
r14 r15
```

x86_64 CALLING CONVENTION

In the example below, only the parameter registers (`rdi`, `rsi`) were edited, so there's no need to write a prologue.

Note that the return value should be in `rax`!

```
section .text

global main
global add_ints

main:
    ; prologue
    mov rdi, 2
    mov rsi, 3

    call add_ints
```

x86_64 CALLING CONVENTION

In the example below, only the parameter registers (`rdi`, `rsi`) were edited, so there's no need to write a prologue.

Note that the return value should be in `rax`!

```
add_ints:
    mov rax, rdi
    add rax, rsi

    ret
```

CALLING CONVENTION: MICROSOFT (WINDOWS)

x86_64 CALLER RULES

- Push caller-saved register values into the stack - `rbx`, `rbp`, and any registers used as parameters to a subroutine.
- The stack pointer should be 16-byte aligned, so a necessary increment to the stack pointer may be required.
- Allocate 32 bytes of free space on the stack for the first four parameters of the subroutine.
- Place the first four parameters into these registers in order: `rcx`, `rdx`, `r8`, `r9`.

x86_64 CALLER RULES

- The fifth and other parameters should be pushed into the stack starting from the last one.
- Once everything has been prepared, use `call` to go to the subroutine.
- Once the subroutine returns, the return value should be in `rax`
- Next, pop any parameters from the stack, including the 32 bytes of free space.
- Finally, restore caller-saved registers values by popping from the stack.

x86_64 SUBROUTINE RULES

- Allocate space for local variables by decrementing the stack pointer (SP) by the size needed.
- Push callee-saved register values into the stack - `rbx`, `rbp`, `r12-r15`.
- Once everything has been prepared, the subroutine can now proceed.
- Once the subroutine is finished, the return value should be saved in `rax`.

x86_64 SUBROUTINE RULES

- Next, restore callee-saved registers values by popping from the stack.
- Deallocate space for local variables by incrementing the stack pointer to its original value before the subroutine started.
- Finally, use `ret` to exit the subroutine.

x86_64 SAVED REGISTERS

Caller-saved (call-clobbered)

Scratch registers that the one calling the subroutine should "save" to the stack if the relevant register contents are useful.

```
rax rcx rdx r8 r9 r10
r11
```

Callee-saved (call-preserved)

Preserved registers that the subroutine should either not change or "save" its original contents if they will be used.

The stack pointer `rsp` is saved when a `call` is used.

```
rbx rbp rdi rsi rsp
r12 r13 r14 r15
```

x86_64 CALLING CONVENTION

The prologue aligns the stack pointer and allocates for the first four parameters in a single instruction.

Example

```
section .text

global main
global add_ints

main:
    ; prologue
    sub rsp, 8 * 5
    mov rcx, 2
    mov rdx, 3

    call add_ints

    ; epilogue
    add rsp, 8 * 5
```

x86_64 CALLING CONVENTION

In the example below, only the parameter registers (`rcx`, `rdx`) were edited, so there's no need to write a prologue.

Note that the return value should be in `rax`!

Example

```
add_ints:
    mov rax, rcx
    add rax, rdx

    ret
```

RESOURCES

- x86 introduction from the University of Washington (US)
- x86 reference from the University of Virginia (US)
- Microsoft x64 calling convention
- NASM documentation

CoE 163

Computing Architectures and Algorithms

03c: x86 Assembly Reference