Basic Assembly Instructions

SE 2XA3

Term I, 2019/20

Outline

Basic instructions

- For a brief description of basic instructions, please see Help, item NASM Cheat Sheet, or at

  `http://www.cas.mcmaster.ca/~franek/courses/cs2xa3/help/cheat_sheet.html`
- For a complete list of instruction, please see Help, item x86 and x86-64 instruction reference , or at

  `https://www.felixcloutier.com/x86/`
- For complete NASM manual, please see Help, item NASM manual, or at

  `https://www.nasm.us/doc/`

Addition, Subtraction, Move

- `add dest, source`
  - **`dest` ← `dest+source`**
  - `dest` is a register or a memory location
  - `source` is a register, a memory location, or immediate
- `sub dest, source`
  - **`dest` ← `dest-source`**
- `mov dest, source`
  - **`dest` ← `source`**
  - `dest` is a register or a memory location
  - `source` is a register, a memory location, or immediate
  - both cannot be a memory location at the same time

Multiplication

- `mul` is for unsigned integers
- `imul` is for signed integers
- 255 x 255 = 65025 if unsigned
  255 x 255 = 1 if signed
- `FFh = 1111|1111`
  as unsigned is 255
  as signed is `1|1111111` = -1
- Two's complement representation
  first bit `1` means -; `0` means +
  flip all the bits, and then add 1

mul

- **`mul source`**
  - **`source`** can be register or memory
  - the other operand is implicit, determined by the size

source	implied operand	result
byte	**`AL`**	**`AX`**
word	**`AX`**	**`DX:AX`**
dword	**`EAX`**	**`EDX:EAX`**
qword	**`RAX`**	**`RDX:RAX`**

imul

- **`imul source`**
  - **`source`** can be register or memory
  - the other operand is implicit
- **`imul source`**
- **`imul source1, source2`**

source	implied operand	result
byte	**`AL`**	**`AX`**
word	**`AX`**	**`DX:AX`**
dword	**`EAX`**	**`EDX:EAX`**
qword	**`RAX`**	**`RDX:RAX`**

Division

- **div** is for unsigned integers
- **idiv** is for signed integers
- both work the same way
- **div source**
  - **source** can be register or memory

source	operation	quotient	remainder
byte	AX/source	AL	AH
word	(DX:AX)/source	AX	DX
dword	(EDX:EAX)/source	EAX	EDX
qword	(RDX:RAX)/source	RAX	RDX

Do not forget to initialize to 0 the remainder !!!

FLAGS register

- Contains various flags
- `cmp a, b`
  - subtracts `a - b`
  - does not store the result
  - sets flags
- For unsigned integers
  - `ZF` so-called **zero flag**
  - `CF` so-called **carry flag**
- For signed integers
  - `ZF` so-called **zero flag**
  - `OF` so-called **overflow flag**; 1 if results overflows
  - `SF` so-called **sign flag**; 1 when the result is negative

- Unsigned integers

cmp a, b		
a-b	ZF	CF
=0	1	0
>0	0	0
<0	0	1

- Signed integers

cmp a, b			
a-b	ZF	OF	SF
=0	1		
>0	0	{0,1}	SF←OF
<0	0	0	1

Jump Instructions

`jump` = transfer execution control

- Unconditional jumps
  - `jmp label`
  - `call label`
- Conditional jumps
  - `jxx label`
  - checks some flags
  - if true, jump to `label`
  - otherwise continue by executing the next statement

forms of conditional jump

First execute an instruction that sets flags such as
`cmp a, b`
then use one of the following forms of **`jxx`**:

mnemonics

For unsigned integers

`je` = jump if equal
`jb` = jump if below
`jbe` = jump if below or equal
`ja` = jump if above
`jae` = jump if above or equal
`jz` = jump if zero

`jne` = jump if not equal
`jnae` = jump if not above or equal
`jna` = jump if not above
`jnbe` = jump if not bellow or equal
`jnb` = jump if not bellow
`jnz` = jump if not zero

forms of conditional jump

First execute an instruction that sets flags such as
`cmp a, b`
then use one of the following forms of **`jxx`**:

mnemonics

For signed integers

`je` = jump if equal
`jl` = jump if less
`jle` = jump if less of equal
`jg` = jump if greater
`jge` = jump if greater or equal
`jz` = jump if zero

`jne` = jump if not equal
`jnge` = jump if not greater or equal
`jng` = jump if not greater
`jnle` = jump if not less or equal
`jnl` = jump if not less
`jnz` = jump if not zero

forms of conditional jump

if	signed	unsigned
`a=b`	`je`	`je`
`a!=b`	`jne`	`jne`
`a<b`	`jl`, `jnge`	`jb`, `jnae`
`a>b`	`jg`, `jnle`	`ja`, `jnbe`
`a>=b`	`jge`, `jnl`	`jae`, `jnb`
`a<=b`	`jle`, `jng`	`jbe`, `jna`

For additional instructions, see the documentation in the Help section

If statements

Consider a Python if statement

```
if <condition>:
    statement_1
    …
    statement_n
```

then-block

If statements

Can be translated as

```
    ;instructions that set flags
    ;according to the <condition>
    ;e.g.  cmp a,b
  jxx end_if
    ;instructions of then-block
 end_if:
```

where `jxx` is a suitable jump instruction

If statements

Consider a Python if statement

```
if <condition>:
    statement_1
    …
    statement_n
else:
    statement_1
    …
    statement_m
```

then-block

else-block

If statements

Can be translated as

```
    ;instructions that set flags
    ;according to the <condition>
    ;e.g.  cmp a,b
  jxx else_block
  ;instructions of then-block
  jmp end_if
else_block:
  ;instructions of else-block
end_if:
```

where `jxx` is a suitable jump instruction

Examples

```
sum=0
i=i-1
if i>0:
    sum=sum+1
```

Can be translated as

```
  ;assume i is in rcx
  mov rax, 0                ;sum=0
  dec rcx                   ;i=i-1
  cmp rcx, qword 0          ;if i > 0
  jbe end_if
  inc rax                   ;sum=sum+1
end_if:
```

Examples

```
if rax>=5:
    rbx=1
else:
    rbx=2
```

Can be translated as

```
cmp rax, qword 5
  jge then_block
  mov rbx, qword 2
  jmp next
then_block:
  mov rbx, qword 1
next:
```

Examples

or as

```
  cmp rax, qword 5
  jnz else_block
  mov rbx, qword 1
  jmp next
else_block:
  mov rbx, qword 2
next:
```

Loop constructs using RCX

`loop` instruction, Example:

```
sum = 0
for x in range(10, -1, -1):
    sum=sum+i
```

Can be translated as

```
  mov rax, dword 0    ;sum=0
  mov rcx, dword 10   ;rcx=10, loop counter
Lstart:
  add rax, rcx        ;sum=sum+i
  loop Lstart         ;decrement rcx
                      ;if rcx!=0, then jump
                      ;to Lstart
```

Loop instructions

loop instruction, Example:

```
sum = 0
for x in range(1,10):
    sum=sum+i
```

Is the following a correct translation?

```
  mov rbx, qword 1
  mov rax, qword 0     ;sum=0
  mov rcx, qword 10    ;rcx=10, loop counter
Lstart:
  add rax, rbx         ;sum=sum+i
  inc rbx
  loop Lstart          ;dec rcx,jump to Lstart
```

No, it is not correct. The python code loops for x from 1 to 9 and the sum is 45. The NASM code loops for rcx from 10 to 0 and the sum is 55

Loop instructions

- **loop Lstart** same as
  - decrement **rcx** by 1
  - if **rcx!=0** goto **Lstart**
- **loope Lstart** the same as **loopz Lstart**
- **loopz Lstart** same as
  - decrement **rcx** by 1
  - if **rcx!=0** and **ZF=1** goto **Lstart**
- **loopne Lstart** the same as **loopnz Lstart**
- **loopnz Lstart** same as
  - decrement **rcx** by 1
  - if **rcx!=0** and **ZF=0** goto **Lstart**

ZF unchanged if **rcx=0**

General loops – while loop

Example

```
while <continuation-condition>:
    statement_1
    …
    statement_n
```

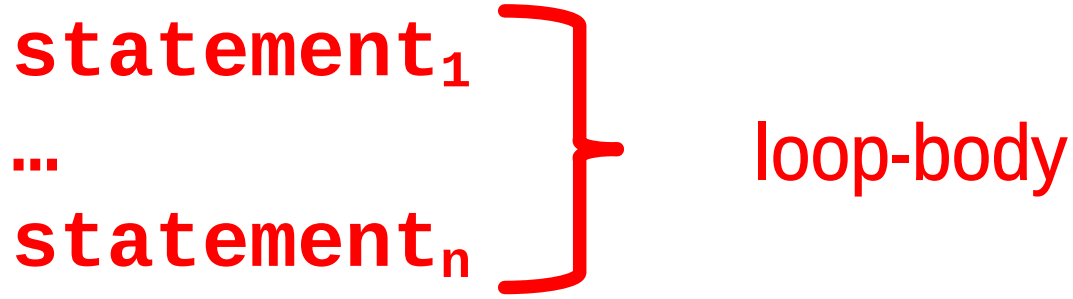
loop-body

Can be translated as

```
while:
    ;code that sets flags
    jxx end_while    ;jump if false
    ;code of loop-body
    jmp while
end_while:
```

General loops – until loop

Example (does not exist in Python)




```
until <termination-condition>
```

Can be translated as

```
until:
    ;code of loop-body
    ;code that sets flags
    jxx end_until    ;jump if true
    jmp until
end_until:
```