Basic Assembly Instructions

CS 2XA3

Term I, 2018/19

Outline

Basic instructions

Addition, Subtraction, Move

Multiplication

Division

FLAGS register

Branch Instructions

Conditional constructs

Loop constructs

Basic instructions

For a brief description of basic instructions, please see Help, NASM Instruction set Reference at

`http://home.myfairpoint.net/fbkotler/nasmdocc.html`

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**

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`

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`

Do not forget to initialize to 0 `DX` or `EDX`

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

cmp

- 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

Branch Instructions

`branch` = **`jump`** = transfer execution control

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

Variants of conditional jump

- `jxx short label`
  - the jump is ± 128 bytes from the current location
  - advantage: the offset is 1 byte
- `jxx near label`
  - the jump is to any location within a segment
  - label is 32 bit
  - default, same as `jxx label`
- `jxx word label`
  - 16-bit label
- `jxx far label`
  - outside a segment

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

`je` = jump if equal

`jl` = jump if less

`jb` = jump if below

`jg` = jump if greater

`ja` = jump if above

`jge` = jump if greater or equal

`jae` = jump if above or equal

`jz` = jump if zero

`jne` = jump if not equal

`jnge` = jump if not greater or equal

`jnae` = jump if not above or equal

`jnle` = jump if not less or equal

`jnbe` = jump if not bellow or equal

`jnl` = jump if not less

`jnb` = jump if not bellow

`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

For additional instructions, see the documentation in the Help section

If statements

Consider a Python if statement

$$\begin{array}{l}\texttt{if <condition>:}\\ \qquad\left.\begin{array}{l}\texttt{statement}_1\\ \texttt{...}\\ \texttt{statement}_n\end{array}\right\}\quad\text{then-block}\end{array}$$

If statements

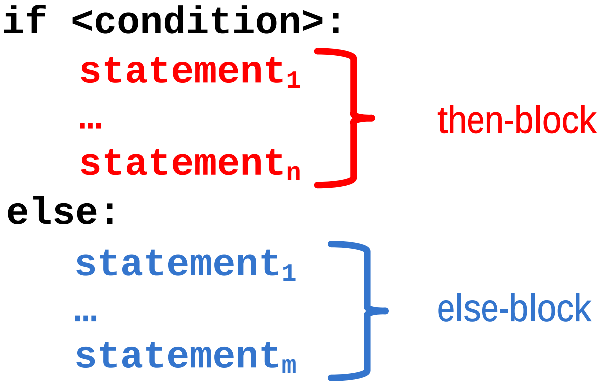
Can be translated as

```
    ;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 branch instruction

If statements

Consider a Python if statement



if <condition>:

    $\texttt{statement}_1$

    …

    $\texttt{statement}_n$

else:

    $\texttt{statement}_1$

    …

    $\texttt{statement}_m$

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 branch instruction

Examples

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

Can be translated as

```
  ;assume i is in ecx
  mov eax, 0          ;sum=0
  dec ecx             ;i=i-1
  cmp ecx, dword 0    ;if i > 0
  jbe end_if
  inc eax             ;sum=sum+1
 end_if:
```

Examples

```
if eax>=5:
   ebx=1
else:
   ebx=2
```

Can be translated as

```
cmp eax, dword 5
  jge then_block
  mov ebx, dword 2
  jmp next
then_block:
  mov ebx, dword 1
next:
```

Examples

or as

```
  cmp eax, dword 5
  jnz else_block
  mov ebx, dword 1
  jmp next
else_block:
  mov ebx, dword 2
next:
```

Loop constructs

`loop` instruction, Example:

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

Can be translated as

```
 mov eax, dword 0    ;sum=0
 mov ecx, dword 10   ;ecx=10, loop counter
Lstart:
 add eax, ecx        ;sum=sum+i
 loop Lstart         ;decrement ecx
                     ;if ecx!=0 goto Lstart
```

Loop instructions

`loop` instruction, Example:

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

Is the following a correct translation?

```
 mov ebx, dword 1
 mov eax, dword 0   ;sum=0
 mov ecx, dword 10  ;ecx=10, loop counter
Lstart:
 add eax, ebx       ;sum=sum+i
 inc ebx
 loop Lstart        ;ecx--, goto 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 ecx from 10 to 0 and the sum is 55

loop

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

ZF unchanged if **ecx=0**

While loops

Example

while <condition>:

    $\texttt{statement}_1$

    …

    $\texttt{statement}_n$

loop-body

Can be translated as

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