Basic Assembly

Signed Operations

Assembly language programming
By xorpd

Objectives

- We will learn about the **NEG** instruction, used to find the two's complement of a number.
- We will learn about different **number extensions**: Signed and Unsigned.
- We will learn about the **Signed versions of MUL and DIV**.
- We will see an example of combining number extension with arithmetic operation.

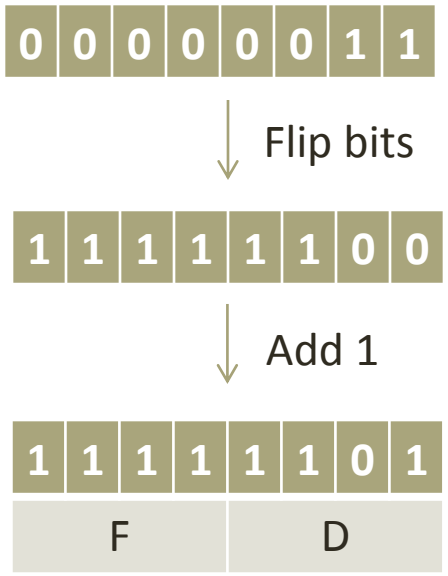
NEG

- NEG – Negate.
- NEG arg
- The NEG instructions allows to change the sign of a number.
  - Using the two's complement method.
  - Flips all the bits, and finally adds 1 to the number.
- Examples:
  - neg eax
    - Changes the sign of eax.
  - neg bl
    - Changes the sign of bl.

NEG - Example

```
mov     al,3d
neg     al

; al == 0xfd
```



Sign extension

- Question: We have a number of size 8 bits, and we want to extend it to 16 bits.
  - How to deal with the sign?
  - Maybe we want to extend from 16 bits to 32 bits, etc.
- Example:
  - 5 is 00000101_2.
  - -5 is 11111011_2 in the two's complement, assuming 8 bits length.
  - Try adding leading zeroes:
    - We get $(00000000\ 11111011)_2$ in base 2, which is 251 according to the two's complement of 16 bits.
    - Not quite what we wanted to get.
  - Try Adding leading ones:
    - We get $(11111111\ 11111011)_2$ in base 2, which is -5 according to the two's complement of 16 bits.
    - This time we got it right.

Sign extension (Cont.)

- Some conclusions:
  - Extending positive numbers: We add leading zeroes.
  - Extending negative numbers: We add leading ones.
- It could be cumbersome (Or inefficient) to do this yourself.

- There are two types of instructions that could help you.
  - Extending “while moving”:
    - MOVSX and MOVZX.
  - Extending “in place”:
    - CBW and CWDE.
    - CWD and CDQ.

MOVZX

- MOVZX – Move with zero extension.
- **MOVZX** destination, source.
- Copies and zero extends source into destination.
- Unsigned extension. (Does not care about the sign).
- Examples:
  - `movzx eax,bl`
    - Extends bl using leading zeroes, and stores the result into eax.
    - Equivalent to:
      - `mov eax,0`
      - `mov al,bl`
  - `movzx esi,cx`
    - Extends cx using leading zeroes, and stores the result into esi.
- Should be used when working with unsigned numbers.

MOVSX

- MOVSX – Move with sign extension.
- **MOVSX** destination, source
- Copies and sign extends source into destination.
- Signed extension (Understands signed numbers).
- Examples:
  - `movsx eax,bl`
    - Extends bl according to bl's sign.
      - If bl's highest bit is 0, extends bl using leading zeroes.
      - If bl's highest bit is 1, extends bl using leading ones.
    - Stores the final extended result into eax.
  - `movsx esi,cx`
    - Sign extends cx, and stores the result into esi.
- Should be used when working with signed numbers.

MOVZX and MOVSX

```
; Unsigned extension:
mov         al,11010000b
movzx       cx,al

; cx == 0000000011010000
```

```
; Signed extension:
mov         al,11010000b
movsx       cx,al

; cx == 1111111111010000
```

```
; Unsigned extension:
mov         al,00100111b
movzx       cx,al

; cx == 0000000000100111
```

Always zero extends.

```
; Signed extension:
mov         al,00100111b
movsx       cx,al

; cx == 0000000000100111
```

Decides extension according to argument's sign.

CBW and CWDE

- **CBW** – Convert Byte to Word.
  - Has no arguments.
  - Sign extends AL to AX.
- **CWDE** – Convert Word to Doubleword.
  - Has no arguments.
  - Sign extends AX to EAX.
- Example:

```
mov      al,10001010b
cbw
; ax  ==                  1111111110001010

cwde
; eax == 11111111111111111111111110001010
```

CWD and CDQ

- Extensions that involve the edx register too.
- **CWD** – Convert Word to Doubleword.
  - Has no arguments.
  - Sign extends AX to DX:AX. (16 bits to 32 bits)
  - Seems to have the same name as CWDE, but does something different.
- **CDQ** – Convert Doubleword to Quadword.
  - Has no arguments.
  - Sign extends EAX to EDX:EAX. (32 bits to 64 bits)
- Examples:

```
mov      ax,0xff12
cwd
; dx == 0xffff, ax == 0xff12.
```

```
mov      eax,0x23c56780
cdq
; edx == 0x00000000
; eax == 0x23c56780
```

```
mov      eax,0xf3c56780
cdq
; edx == 0xffffffff
; eax == 0xf3c56780
```

IMUL and IDIV

- IMUL and IDIV are the sign aware versions of MUL and DIV.
  - Understand the two's complement representation.

- One can think of those instructions as doing the following:
  - Remember the original signs of the operands.
  - Convert all numbers to positive numbers.
  - Invoke Multiplication or Division.
  - Convert the result to negative if necessary.

- Example (In 8 bit two's complement):
  - 0xf8 / 0x4 = ? (Negative / Positive = Negative)
  - $0\text{xf8} = 11111000_2 \rightarrow 00000111_2 + 1 = 00001000_2 = 8_{10}$
  - 8/4 = 2 → 0xf8 / 0x4 = -2
  - $2 = 00000010_2 \rightarrow 11111110_2 = 0xfe$
  - Finally 0xf8 / 0x4 = 0xfe.

IMUL

- IMUL arg
  - Signed multiplication.
  - Basically just like MUL, but understands sign. (Although has some more advanced forms).
  - arg of size 32 bits:
    - $edx{:}\,eax \leftarrow eax \cdot \arg$
  - arg of size 16 bits:
    - $dx{:}\,ax \leftarrow ax \cdot \arg$
  - arg of size 8 bits:
    - $ax \leftarrow al \cdot \arg$
  - IMUL has some more advanced forms.
  - Example:

```
mov     al,0x9c  ; == -0x64
mov     cl,0x19
imul    cl

; ax == 0xf63c ; == -0x9c4
```

IDIV

- IDIV arg
  - Signed division.
  - Just like DIV, but considers the sign of the dividend and the divisor.
  - arg of size 32 bits:
    - $eax \leftarrow edx{:}eax / \arg$; $edx \leftarrow\ edx{:}eax\ \%\ \arg$.
  - arg of size 16 bits:
    - $ax\ \leftarrow dx{:}ax / \arg$; $dx\ \leftarrow dx{:}ax\ \%\ \arg$.
  - arg of size 8 bits:
    - $al\ \leftarrow ax / \arg$; $ah\ \leftarrow ax\ \%\ \arg$.
- Example:

```
mov     ax,0xf63c ; -0x9c4
mov     cl,0x19
idiv    cl

; ah == 0x00, al == 0x9c == -0x64
```

Example (IDIV and CDQ)

- Signed division:

```
; This program divides eax by 3.

mov     esi,3
call    read_hex  ; input
cdq
idiv    esi
call    print_eax ; output
```

- Note the combination of CDQ and IDIV.
  - They usually come together.
- CDQ knows how to extend eax to edx:eax both if eax is a positive or a negative number.
  - If we use `mov edx,0` instead, we will not get correct results for negative inputs.

Summary

- NEG can find the two's complement of a number.
- We can extend numbers using movzx (Unsigned) and movsx (Signed).
- We could extend numbers "in place":
  - using one of CBW,CWDE (If we want to extend inside eax)
  - using one of CWD,CDQ (If we want to extend to edx too).
- IMUL and IDIV are the signed versions of MUL and DIV.
- CDQ and IDIV work well together.

Exercises

- Code Reading.
  - You will see some sign aware instructions that we have just learned about.
- Code Writing.
- Have fun :)