Typically all arithmetic & logical instructions set the condition flags.

But LEAL does not update any flags.

→ intended for address computations.

But Logical operators

AND, OR, XOR only update the ZF or the SF flag

They do not ⇒ OF / CF.

Why? ↘ Set them to zero.

Shift operators

CF → set to the last bit that was switched out

0001 >> 1
ZF → 1

OF → 0 / Always.

① 1001 >> 2

CF → 0 /

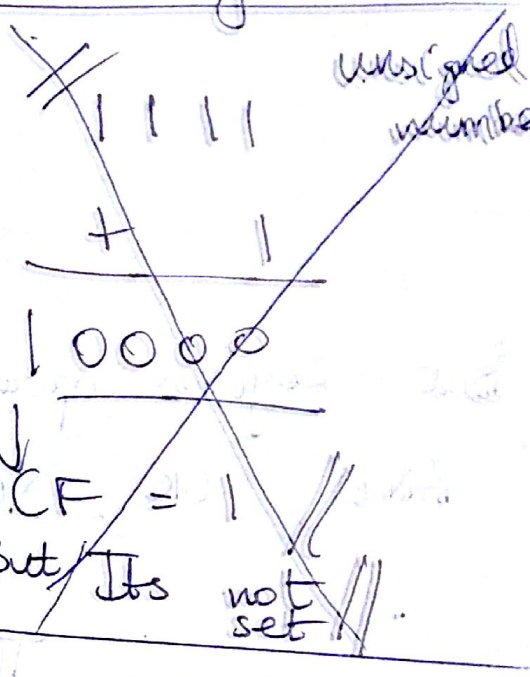
② 1001 << 1

CF → 1

INC / DEC → Set the zero flag / Sign flag / OF.

But they do not set the carry flag //

Out of scope for this class!



Compare Instruction

Syntax → CMP$_{B,W,L}$ S_2, S_1 //

Just like subtraction

Performs the operation → $S_1 - S_2$ //

Difference! It only sets the condition code registers!
It does not change any other registers or memory location.

① ZF is set to 1 ⇒ if they are equal

② Based on how other flags are set, we can do comparisons (We'll see briefly).

TEST instruction

TEST_ S_2, S_1

B W L

It performs the operation → S_1 & S_2.

Except it only modifies the condition flags.

testl %eax, %eax

ZF = 1 => If eax is zero so ZF will be 1

SF = 1 => eax is negative

SF = 0 => eax is positive or zero.

```
 0000
&0000
-----
 0000
-----
 1000
&1000
-----
 1000
-----
```

Accessing the condition codes / flags:

We don't access it directly.

We set a single BYTE to 0 or 1 based on some combo of the condition flags.

SET instruction(s)

① sete D (or) — 1 byte — (D ← ZF)

setz D — 1 byte

set zero. → 0

② setne D (or) setnz D — (D ← ~ZF)

③ sets D — (D ← SF)

④ setns D — (D ← ~SF)

1 byte (0000 0000) – 0

(0000 0001) – 1

Fig 3.11 in CS:APP.

Signed SET instructions → setg, setge, setl, setle.

Unsigned SET instructions → seta, setae, setb, setbe.

↑ The descriptions will make more sense once we look at them in the context of the compare instruction.

For example:

CMPL %eax, %edx If they are equal,

%edx − %eax = 0 ⇒ ZF = 1

Example:

if a < b

a is in %edx
b is in %ebx

Set z (or) sete → set if zero.

sete → set if equal.

Cmpl %ebx, %edx (a − b)

setl %al → al will be set to 0 if a is not lower than or equal to b. and 1 otherwise. if it is 1

movzbl %al, %eax

(Check next lecture's notes)

① cmpl %ebx , %edx

if (a − b == 0), ZF = 1

if (a − b < 0), SF = 1

if (a − b >= 0), SF = 0 //

OF = 1 when a = −8 and b = 4.

OF = 0 when a = 5 and b = 2.

② setl %al ← a single byte register.

Performs the operation => (setl D)

D ← SF ^ OF

SF	OF	XOR D
0	0	0
0	1	1
1	0	1
1	1	0

a = −8 → 1 0 0 0

b = 7 (−) 0 1 1 1

0 0 0 1

SF = 0, OF = 1 & D = 1

a < b.

a = 3 → 0011

b = 4 (−) 0100 (+) 1100

SF = 1 OF = 0 // So D = 1

0 0 1 1

1 1 1 1

We talked about a quick way to check if an signed overflow (OF) happened when ADDING two numbers. // For subtraction,

$-8 - (7)$

$-8 + (-7)$

$-8 \Rightarrow 1000$

$-7 \Rightarrow (+)\ 1001$

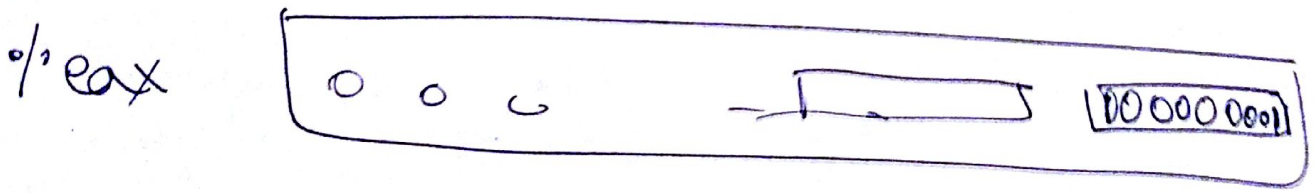
$\times\ 0001$

OF = 1

---

(3)

movzbl %al, %eax



---

(1) %al is already set

(2) So set the remaining most significant 3 bytes of eax to zero

JUMP instruction

Jump ↑

CAN cause the execution to switch to a completely new position in the program.

↳ It depends on some condition.

or sometimes it may not depend on anything

Unconditional Jumps

① Direct Jump

Syntax → JMP Label

Eg

```
0x104 mov . . .
0x106 jmp Label1
0x108 mov . . .
      add . . .
Label1:
      pushl . . .
      add . . .
```

② Indirect Jump

Syntax → JMP *Operand

Operand.

Eg.

jmp * %eax

→ Jumps to the value stored in eax

jump * (%eax)

→ Reads the jump target from the memory address stored in %eax.

Quick Aside

~ → bitwise NOT

! → logical NOT.

~ → simply flips the bits.

~(0000) → 1111

~(0101) → 1010

! → ① ! (Any non-zero number) → ①

! (-5) → 0

! (1) → 0

! (200000) → 0

② ! (0) → 1

!! (-5) → ! (0) → 1

2's complement of some unsigned number.

7 → 0 1 1 1

To find out →

Step 1 → Flip the bits (~)

→ 1 0 0 0

Step 2 → Add 1 to it

→ 1 0 0 1

→ $-1 \times 2^3 + 0 + 0 + 1 \times 2^0$

$= -8 + 0 + 0 + 1$

$= -7$ //