Scaled Indirect with fixed displacement:

Imm(, %RAX, 2)
↑ scale

Memory address is sum of Imm plus product of register and scale

Scaled Indirect with Index Register:

(%RAX, %RBX, 8)
↑ scale
other register ↑ ↑ Index register

Memory address is product of index register and scale added to other register

Scaled Indirect with Index register and fixed displacement:

Imm(%RBP, %RAX, 4)
↑ scale
other register ↑ ↑ Index register

Memory address is sum of Imm plus other register plus the product of index register and scale

07/12

Push and Pop Instruction:

W = word 16 bits

pushw	Reg16	%RSP -= 2;	%Reg16 → (%RSP)
pushw	Mem16	%RSP -= 2;	Mem16 → (%RSP)
pushW	Imm16	%RSP -= 2;	Imm16 → (%RSP)

L = Long 32 bits

pushL	Reg32	%RSP -= 4;	%Reg32 → (%RSP)
pushL	Mem32	%RSP -= 4;	Mem32 → (%RSP)
PushL	Imm32	%RSP -= 4;	Imm32 → (%RSP)

Q = Quadword 64 bits

PushQ	Reg64	%RSP -= 8;	%Reg64 → (%RSP)
PushQ	Mem64	%RSP -= 8;	Mem64 → (%RSP)
PushQ	Imm64	%RSP -= 8;	Imm64 → (%RSP)

Pop W	Reg 16	(%RSP) → %Reg 16 ; %RSP += 2
Pop W	Mem 16	(%RSP) → Mem 16 ; %RSP += 2
Pop L	Reg 32	(%RSP) → %Reg 32; %RSP += 4
Pop L	mem 32	(%RSP) → mem 32; %RSP += 4
Pop Q	Reg 64	(%RSP) → %Reg 64 ; %RSP += 8
Pop Q	mem 64	(%RSP) → mem 64; %RSP += 8

limited to 32 bit

*If you push 2/4/8 bytes, you have to pop 2/4/8 bytes.
* Assembly code → part of bits in memory pattern
* Assembly language ≠ Machine language (bits)
* Every functions set up a stack frame and break down stack frame.

Move Instructions:

Moving (actually copying) stuff

- movB
- movW
- movL
- movQ

R	R
R	m
m	R
I	R
I	m

*Both operands must be same size.

Move smaller Src to larger Dest with Sign Extension

- movSBW
- movs BL
- movs BQ
- movs WL
- movs WQ
- movs LQ

ex) movsWL

AB CD ⇒ FF FF AB CD

2 byte Word → 4 byte Long

*preserve sign ⊖

ex) movsWL

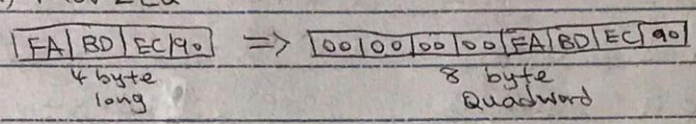
07 89 ⇒ 00 00 07 89

*preserve sign ⊕

Move Smaller src to larger Dest with Zero extension

- movZBW
- movZBL
- movZBQ
- movZWL
- movZWQ
- movZLQ

ex) movZLQ



*Just add 0 ignoring the sign of the highest bit.

Sign Extension for C short, int, long, char

Zero Extension for unsigned char, unsigned int, unsigned short, unsigned long

a really short number (0~255)

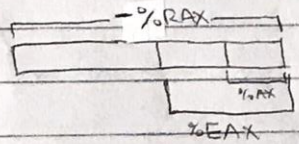
Move Larger src to smaller Dest: (?)

*32 bit pointer (i.e. address) implies 2^{32} memory size.

*64 bit pointer (i.e. address) implies 2^{64} memory size

- R R No; larger registers overlap smaller registers
- R M No; should get the src you want from smaller register
- M R No; should get the memory src you want
- I R No; make immediate the same size as register
- I M No; make immediate the right size

ex)



movW $0, %AX
movL $-1, %EAX %AX == 0XFFFF

movL $0, %EAX
movW $-1, %AX %EAX = 0000FFFF

movB $0XF3, %BH
movW %AX, %BX
movL $0X007FFFFF, (%ECX)
32 bit immediate

movSBL %AH, %EAX
movZWL (%EBX), %EDX

Arithmetic Instructions:

Single operand register or memory

Inc	B
Dec	W
Neg	L
Not	Q

Operand ↖ 8, 16, 32, or 64 bit

16 diff. instructions

Increment by 1	X++
Decrement by 1	X--
Negate	X = -X
Flip bits	X = ~X

Two operands

ADD	B
Sub	W
or	L
and	Q
Xor	

Src, DST

DST += Src	
DST -= Src	
DST |= Src	
DST &= Src	
DST ^= Src	

*Dest. can change BUT! Src can NOT change!

Shift Instructions: ✡ Know this!

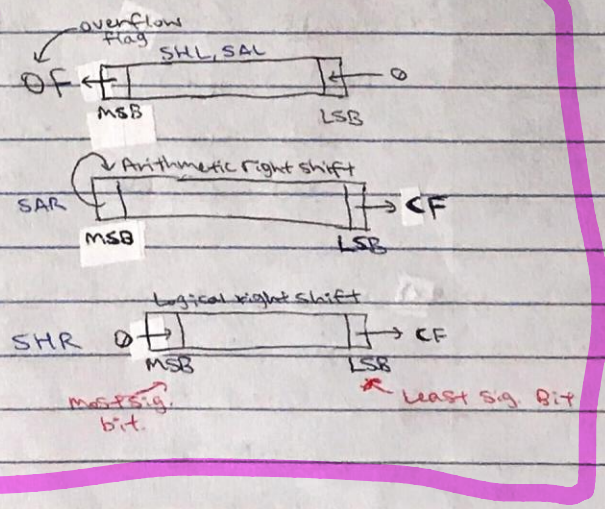
- Two operands Src specifies how many bits
- Src is immediate or %CL

SAL	B
SHL	W
SAR	L
SHR	Q

SAL, SHL } Both are left shift

SAR } Arithmetic, Both are right shift

SHR } Logical



OF = Overflow flag

CF = carry flag

SF = Sign flag

*Logical shifts for unsigned

* Arithmetic " for signed (Arith. shift = Sign preserving)

movL %EAX, %ECX change %CL Register

SHL %EDX, %CL
└ %ECX

Multiplication - Some math

1. The sum of two N-bit numbers may require up to N+1 bits. Intel carry flag CF
2. The product of two N-bit numbers may require up to 2N bits. Proof follows.

Unsigned Ints	2^N-1
$0 \ldots 2^N-1$	$\times\ 2^N-1$
	-2^N+1
	$(2^N)^2-2^N$
	$2^{2N}-2^{N+1}+1$

Signed Ints	2^N-1
$-2^{N-1} \ldots 2^{N-1}-1$	2^N-1
	$-2^{N-1}+1$
	$(2^{N-1})^2-2^{N-1}$
	$2^{2N-2}-2^N+1$

Multiplication Instructions:

One operand, unsigned

Implicit

- I · mulB operand ← is a register or memory
- I · mulW
- I · mulL
- I · mulQ

		16 bit	8 bit	8 bit
ex)	mulB opnd8	%AX ⇐	%AL *	opnd8
	mulW opnd16	%DX:%AX ⇐	%AX *	opnd16
	mulL opnd32	%EDX:%EAX ⇐	%EAX *	opnd32
	mulQ opnd64	%RDX:%EAX ⇐	%RAX *	opnd64

ex)
```
movL $20, %EAX
mulL $30
→%EDX == 0
 %EAX == 600
```

IMUL signed multiplication, two operands

16 bit

- Imul Mem, Reg16 — Reg16 *= mem16
- Imul $Reg16_s$, $Reg16_D$ — $Reg16_D$ *= $Reg16_s$
- Imul Imm8, Reg16 — Reg16 *= Imm8
- Imul Imm16, Reg16 — Reg16 *= Imm16

32 bit	SRC	Dest
IMul	mem, Reg32	Reg32 *= mem
	Reg16, Reg32	Reg32 *= Reg16
	Reg32$_S$, Reg32$_D$	Reg32$_D$ *= Reg32$_S$
	Imm8, Reg32	Reg32 *= Imm8
	Imm16, Reg32	Reg32 *= Imm16
	Imm32, Reg32	Reg32 *= Imm32

*size of the src, determine size of the dest.

*register is always a dest.

07/17 Intel Division Instructions:

Unsigned integers, one operand

*use cltd before idiv

DivB opnd %AL ∈ %AX / opnd Quotient

%AH ∈ %AX % opnd Remainder

DivW opnd %AX ∈ %DX : %AX / opnd

%DX ∈ %DX : %AX % opnd (32 bit register pair)

DivL opnd %EAX ∈ %EDX : %EAX / opnd

%EDX ∈ %EDX : %EAX % opnd (64 bit register pair)

DivQ opnd %RAX ∈ %RDX : %RAX / opnd

%RDX ∈ %RDX : %RAX % opnd (128 bit register pair)

~~For signed integers,~~
- IDivB
- IDivW
- IDivL
- IDivQ

Comments:

- opnd is register or memory (not Immediate)
- Must set high part of register pair even if numerator fits in one register
- Dividing by 1 still gets you results that fit in two registers.
- Implicit use of %AX, %EAX, %RAX registers
- Dividing by 0 stops execution.