Machine-Level Representation of Programs

(1)

* Machine code vs Assembly code.
  - binary
  - (mostly) human readable ☺

* Reverse engineering - find out how a compiler generated Assembly code from a C program.

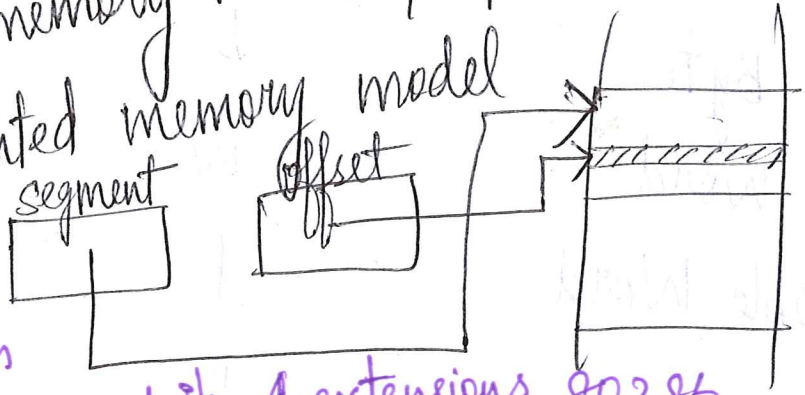
* Machine Languages - IA 32 and X86-64.
  - IA 32 → 32-bit data and addresses. max addr range = 4GB.
  - X86-64 → 64 bit data & addr. Theoretically 2^{64} addresses can be accessed. Practically around 2^{48} bytes or $2^{8} \times 2^{40}$ bytes = 256 TB.

IA32 - Intel Architecture 32-bit.

* Memory models

1. Flat memory model

2. Segmented memory model



used in 8086 and its extensions 80286.

Aside: PAE - Physical Address Extension
- Allows IA32 machines to have a addr space more than 4GB.
  eg. Linux with IA32 + PAE - max of 64GB.

Machine-Level Code

ISA - Instruction Set Architecture.

Processor Registers

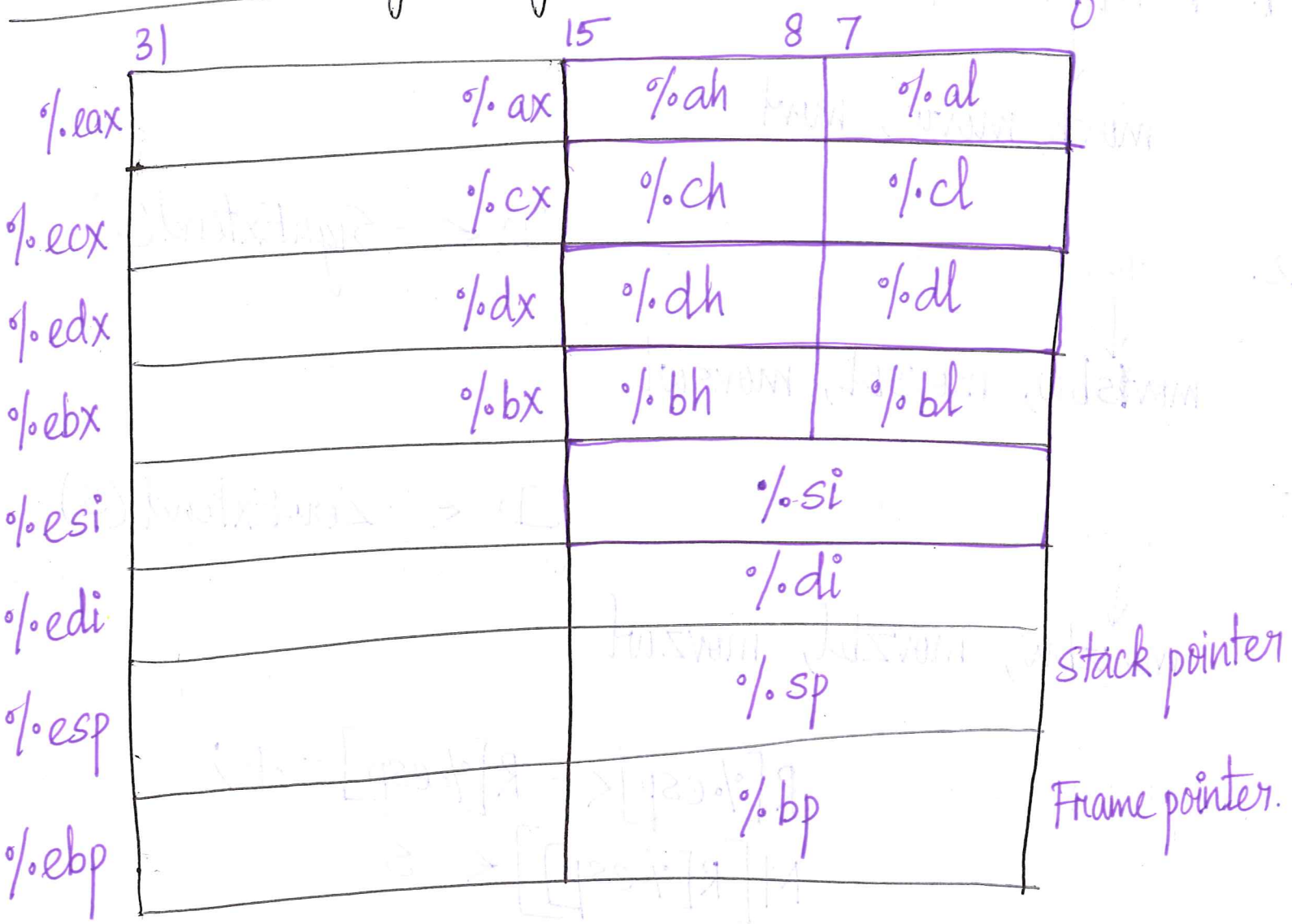
1. Program Counter (PC)
2. Integer register file (8 named 32-bit registers)
3. Condition code registers.
4. Floating-point registers.

(X) SHOW ASSEMBLY CODE EXAMPLES!

* Data Formats

C type	Intel data type	Assembly suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double Word	l	4
long int	"	l	4
long long int	—	—	4
char *	Double Word	l	4

IA 32 Integer Registers



Operand Forms

	Type	Form	Operand Value	Name
1.	Immediate	$Imm	Imm	Immediate
2.	Register	Ea	R[Ea]	Register.
3.	Memory	Imm	M[Imm]	Absolute.
4.	Memory	Imm(Eb, Ei, s)	M[Imm + R[Eb] + R[Ei] · s]	Scaled Indexed.

Data Movement Instructions

1. MOV S, D — D ← S — "Move."

   ↓

   movb, movw, movl

2. MOVS S, D — D ← SignExtend(S)

   ↓

   movsbw, movsbl, movswl — Move with sign extension.

3. MOVZ S, D — D ← ZeroExtend(S)

   ↓

   movzbw, movzbl, movzwl — Move with zero extension.

4. pushl S

   R[%esp] ← R[%esp] - 4;
   M[R[%esp]] ← S

5. popl D

   D ← M[R[%esp]];
   R[%esp] ← R[%esp] + 4;

5 possible combinations (5) of source and dest. operands.

1. Immediate → Register

```
movl $0x4050, %eax
```

2. Register → Register

```
movw %bp, %sp
```

3. Memory → Register

```
movb (%edi, %ecx), %ah
```

4. Immediate → Memory

```
movb $-17, (%esp)
```

5. Register → Memory

```
movl %eax, -12(%ebp)
```

---

* Stack Operation
* Data Movement Example
* Arithmetic and Logical Operations (X)

Arithmetic and Logical Operations (6)

1. Load Effective Address

leal S, D → must be a register

D ← &S

%edx
x

eg. leal 7(%edx, %edx, 4), %eax

⇒ %eax = 7 + x + x·4 = 5x + 7.

S: @memory location. (M[S] is not computed!)

eg.
```
struct Point
{ int xcoord;
  int ycoord;
};
struct Point points[10];
```

① int y = points[i].ycoord;

mov 4(%ebx, %eax, 8), %edx

⇒ loads y in edx.

- %ebx: base of array points[]
- %eax: index i
- 8: each Point is 8 bytes in size.
- 4: ycoord is at offset 4 bytes in the struct.

② int *py = &points[i].ycoord

leal 4(%ebx, %eax, 8), %esi

loads address of y in %esi.

2. Unary Operations.

inc D	D ← D+1
dec D	D ← D−1
neg D	D ← −D
not D	D ← ~D

negate
complement

3. Binary Operations

add S, D	D ← D+S
sub S, D	D ← D−S
imul S, D	D ← D*S
xor S, D	D ← D ^ S
or S, D	D ← D | S
and S, D	D ← D & S

4. Shift Operations

sal K, D	D ← D ≪ K	} Left shift. same.
shl K, D	D ← D ≪ K	
sar K, D	D ← D $\gg_A$ K	Arithmetic Right shift.
shr K, D	D ← D $\gg_L$ K	Logical Right shift.

Control

sequential code vs conditionals, loops, switches "jump"

Condition Code Registers (1 bit)

1. CF - Carry Flag
2. ZF - Zero Flag
3. SF - Sign Flag
4. OF - Overflow Flag

$t = a + b$ $\quad a, b, t \in$ integers.

CF: (unsigned) $t <$ (unsigned) a - Unsigned overflow

ZF: $(t == 0)$ - Zero

SF: $(t < 0)$ - Negative

OF: $(a < 0 == b < 0) \;\&\&\; (t < 0 \;!=\; a < 0)$ - Signed overflow.

Comparison and Test (sets condition codes w/o updating any other registers).

1. CMP S2, S1 $\quad (S1 - S2) \Rightarrow$ condition

↓

cmpb, cmpw, cmpl

2. TEST S_2, S_1

Condition = S_1 & S_2.

eg. testl %eax, %eax

Test to see if %eax is negative, positive, or zero.

testb, testw, testl

Note: leal - does not alter any condition codes.
(∵ intended to be used in address computations).

*Accessing the Condition Codes

SET instructions.

1. sete D (or) setz — D ← ZF
2. setne D (or) setnz — D ← ~ZF
3. sets D — D ← SF
4. setns D — D ← ~SF

See Fig 3.11 in CS:APP 2e for other SET instructions for signed and unsigned comparisons.

eg. setg, setge, setl, setle — signed comparisons
seta, setae, setb, setbe — unsigned comparisons.

eg. $a < b$ a, b are integers.

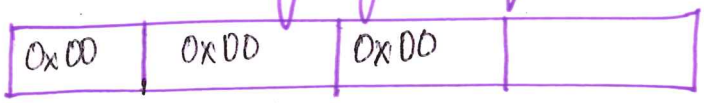
a is in %edx, b is in %eax.

```
cmpl  %eax, %edx
setl  %al
movzbl %al, %eax
```

Compare a : b

Set low order byte of %eax to 0 or 1.

Set remaining bytes of %eax to 0.



1. cmpl %eax, %edx

if $(a-b) = 0$, ZF = 1

else if $(a-b < 0)$, SF = 1

else $(a-b > 0)$, SF = 0

} overflow may or may not have occured.

2. if no overflow (OF = 0) / if overflow (OF = 1)

setl D $\Rightarrow$ D $\leftarrow$ SF $\wedge$ OF

Less (signed <)

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

eg. 4 bits

$a = -8$ 1000 OF=1

$b = 7$ 0111 SF=0

$a-b =$ 0001

eg. $a = 3$ 0011 SF=1

$b = 4$ 0100 OF=0

$a-b =$ 1111

In both these case $a < b$.