

ΗΡΥ 201

ΨΗΦΙΑΚΟΙ ΥΠΟΛΟΓΙΣΤΕΣ

Αρχιτεκτονική Συνόλου Εντολών Intel x86

N. Αλαχιώτης

© Δ. Πνευματικάτος 2013

Σύνοψη χαρακτηριστικών IA32

Evolutionary Design

- Starting in 1978 with 8086 (really 1971 with 4004)

Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats
- Αριθμητικές εντολές μπορούν να διαβάσουν/γράψουν θέσεις μνήμης
- (more) Difficult & (somewhat) Expensive, but Possible

2-address architecture

- Κάθε εντολή ορίζει ένα source & ένα destination
- Add src, dst σημαίνει dst = dst + src (ανάλογα για sub, κλπ)

Σύνθετος τρόπος πρόσβασης μνήμης (addressing modes)

Κωδικοποίηση εντολών:

- Μεταβλητού μήκους, 1-15 bytes

Μια απλή εντολή: movl

Moving Data

`movl Source, Dest:`

- Move 4-byte (“long”) word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
 - Like C constant, but prefixed with ‘\$’
 - E.g., \$0x400, \$-533
 - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
 - But %esp and %ebp reserved for special use
 - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
 - Various “address modes”

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

Register Names

8 καταχωρητές + EIP (PC)

Eax Accumulator

Ebx Base register

Ecx Count register

Edx Double-precision register

Esi Source index register

Edi Destination index register

Ebp Base pointer register

Esp Stack pointer

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

x64 Register Names

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

movl Operand Combinations

	Source	Destination	C Analog
movl	Imm	Reg	movl \$0x4,%eax temp = 0x4;
		Mem	movl \$-147,(%eax) *p = -147;
	Reg	Reg	movl %eax,%edx temp2 = temp1;
		Mem	movl %eax,(%edx) *p = temp;
	Mem	Reg	movl (%eax),%edx temp = *p;

- Cannot do memory-memory transfers with single instruction
- Ομοίως για άλλες εντολές, add, κλπ

Indexed Addressing Modes

Most General Form

D(Rb,Ri,S)

Mem[Reg[Rb]+S*Reg[Ri]+ D]

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **Rb:** Base register: Any of 8 integer registers
- **Ri:** Index register: Any, except for %esp
 - Unlikely you’d use %ebp, either
- **S:** Scale: 1, 2, 4, or 8

Special Cases

(Rb,Ri)

Mem[Reg[Rb]+Reg[Ri]]

D(Rb,Ri)

Mem[Reg[Rb]+Reg[Ri]+D]

(Rb,Ri,S)

Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

Expression	Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

Flags (condition codes)

	VM	RF		NT	IO PL	IO PL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
--	----	----	--	----	----------	----------	----	----	----	----	----	----	--	----	--	----	--	----

CF	Carry Flag	IOPL	I/O Privilege Level (286+ only)
PF	Parity Flag	IOPL	I/O Privilege Level (286+ only)
AF	Auxiliary Flag	NT	Nested Task Flag (286+ only)
ZF	Zero Flag	RF	Resume Flag (386+ only)
SF	Sign Flag	VM	Virtual Mode Flag (386+ only)
TF	Trap Flag (Single Step)		
IF	Interrupt		
DF	Direction Flag		
OF	Overflow flag		

Flag meaning

CF – Carry Flag

- Set by arithmetic instructions. Can be set, inverted and cleared with STC, CLC or CMC instructions respectively

PF – Parity Flag

- Set by most instructions if the least significant eight bits of the destination operand contain an even number of 1 bits.

AF – Auxiliary Flag

- If a carry or borrow from the most significant nibble of the least significant byte – Aids BCD arithmetic

ZF – Zero Flag

- Set by most instructions if the result of the arithmetic operation is zero

SF – Sign Flag

- On signed operands, this tells whether the result is positive or negative

TF – Trace Flag

- Allows single-step through programs. Executes exactly one instruction and generates an internal exception 1 (debug fault)

Flag meaning

IF – Interrupt Flag

- When set, the processor recognizes the external hardware interrupts on INTR pin. On clearing, anyway has not effect on NMI (external non maskable interrupt) pin or internally generated faults, exceptions, traps etc. This flag can be set and cleared using the STI and CLI instructions respectively

DF – Direction Flag

- Specifically for string instructions. DF = 1 increments ESI and EDI, while DF = 0 decrements the same. Set and cleared by STD and CLD instructions

OF – Overflow Flag

- Most arithmetic instructions set this flag to indicate that the result was at least 1 bit too large to fit in the destination

IOPL – Input Output Privilege Level flags (2 bits)

- For protected mode operations – indicates the privilege level, 0 to 3, at which your code must be running to execute any I/O-related instructions

NT – Nested Task Flag

- When set, it indicates that one system task has invoked another through a CALL instruction as opposed to a JMP. For multitasking ...

RF – Resume Flag

- It is related to Debug registers DR6 and DR7. By setting this, you can selectively mask some exceptions while you are debugging code

CISC Architecture Properties

Instruction can reference different operand types

- Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation

- $R_b + S * R_i + D$
- Useful for arithmetic expressions, too

Instructions can have varying lengths

- IA32 instruction size ranges from 1 to 15 bytes

Use Condition Codes

- 9 (!) condition code flags plus more processor status flags

Γενικό (Εγκυκλοπαιδικό) Υλικό

Σύνοψη χαρακτηριστικών IA32

Evolutionary Design

- Starting in 1978 with 8086 (really 1971 with 4004)

Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats
- Difficult & (somewhat) Expensive, but Possible

2-address architecture

- Κάθε εντολή ορίζει ένα source & ένα destination
- Add src, dst σημαίνει dst = dst + src (ανάλογα για sub, κλπ)

Σύνθετος τρόπος πρόσβασης μνήμης (addressing modes)

Κωδικοποίηση εντολών:

- Μεταβλητού μήκους, 1-15 bytes

x86 history

Name	Date	Transistors
4004	1971	2.3K
<ul style="list-style-type: none">■ 4-bit processor. First 1-chip microprocessor■ Didn't even have interrupts!		
8008	1972	3.3K
<ul style="list-style-type: none">■ Like 4004, but with 8-bit ALU		
8080	1974	6K
<ul style="list-style-type: none">■ Compatible at source level with 8008■ Processor in first “kit” computers■ Cheaper than similar processors with better programming models<ul style="list-style-type: none">● Motorola 6800● MOS Technologies (MOSTEK) 6502		

x86 history

Name	Date	Transistors
8086	1978	29K
<ul style="list-style-type: none">■ 16-bit processor. Basis for IBM PC & DOS■ Limited to 1MB address space. DOS only gives you 640K		
80286	1982	134K
<ul style="list-style-type: none">■ Added elaborate, but not very useful, addressing scheme■ Basis for IBM PC-AT and Windows		
386	1985	275K
<ul style="list-style-type: none">■ Extended to 32 bits. Added “flat addressing”■ Capable of running Unix■ By default, Linux/gcc use no instructions introduced in later models		

x86 history

Name	Date	Transistors
486	1989	1.9M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M

- Added special collection of instructions for operating on 64-bit vectors of 1-, 2-, or 4-byte integer data

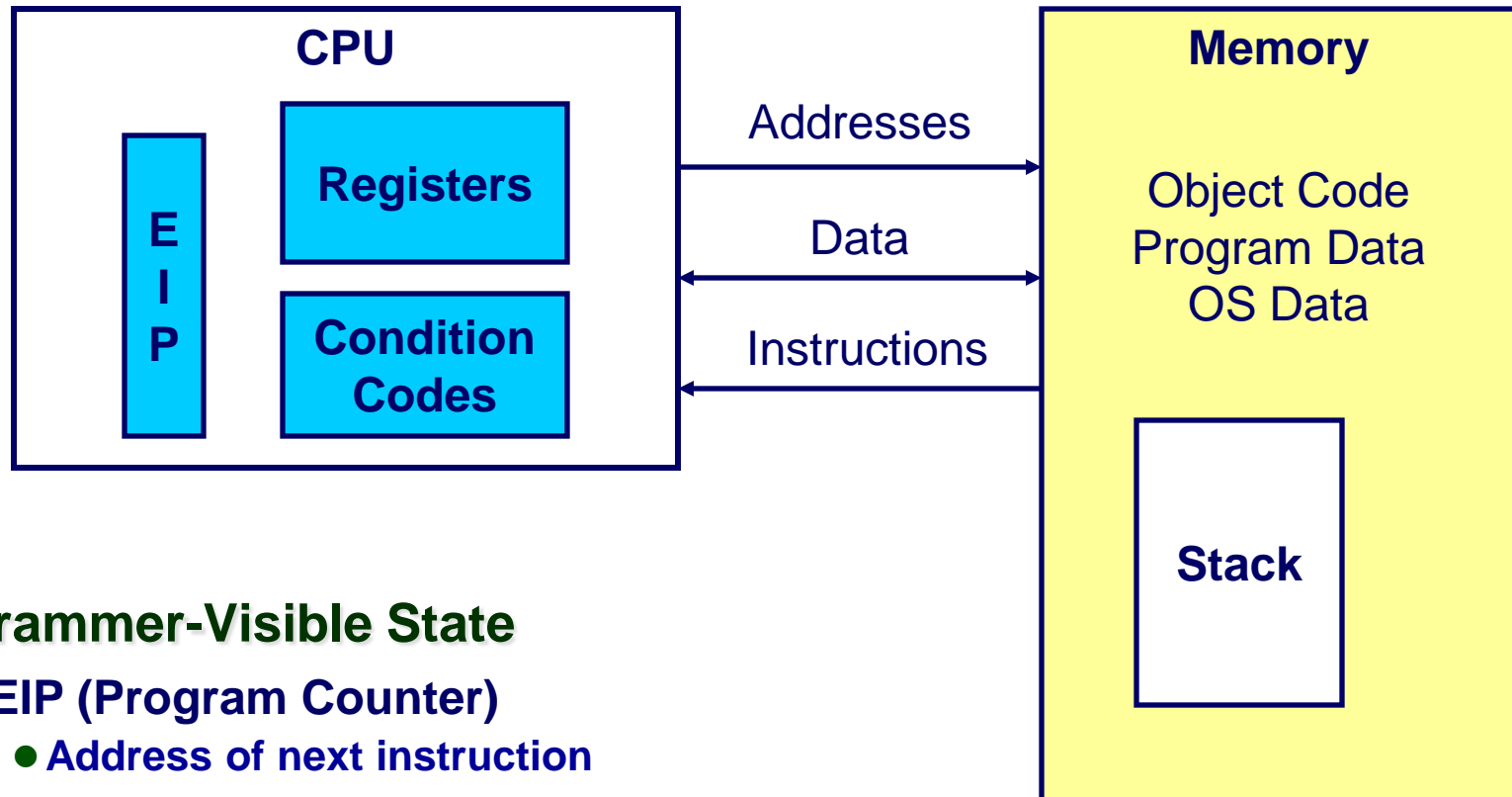
PentiumPro	1995	6.5M
-------------------	-------------	-------------

- Added conditional move instructions
- Big change in underlying microarchitecture
- First “serious” Intel microArchitecture!

x86 history

Name	Date	Transistors
Pentium III	1999	8.2M
■ Added “streaming SIMD” instructions for operating on 128-bit vectors of 1-, 2-, or 4-byte integer or floating point data		
Pentium 4	2001	42M
■ Added 8-byte formats and 144 new instructions for streaming SIMD mode		
...		
8-core Core i7 Haswell-E	2014	2.6B
18-core Xeon Haswell-E5	2014	5.56B

Assembly Programmer's View



Programmer-Visible State

- **EIP (Program Counter)**
 - Address of next instruction
- **Register File**
 - Heavily used program data
- **Condition Codes**
 - Store status information about most recent arithmetic operation
 - Used for conditional branching

- **Memory**
 - Byte-addressable array
 - Code, user data, (most) OS data
 - Includes stack used to support procedures

Moving Data

Moving Data

`movl Source, Dest:`

- Move 4-byte (“long”) word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
 - Like C constant, but prefixed with ‘\$’
 - E.g., \$0x400, \$-533
 - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
 - But %esp and %ebp reserved for special use
 - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
 - Various “address modes”

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

Register Names

Eax Accumulator

Ebx Base register

Ecx Count register

Edx Double-precision register

Esi Source index register

Edi Destination index register

Ebp Base pointer register

Esp Stack pointer

%eax

%edx

%ecx

%ebx

%esi

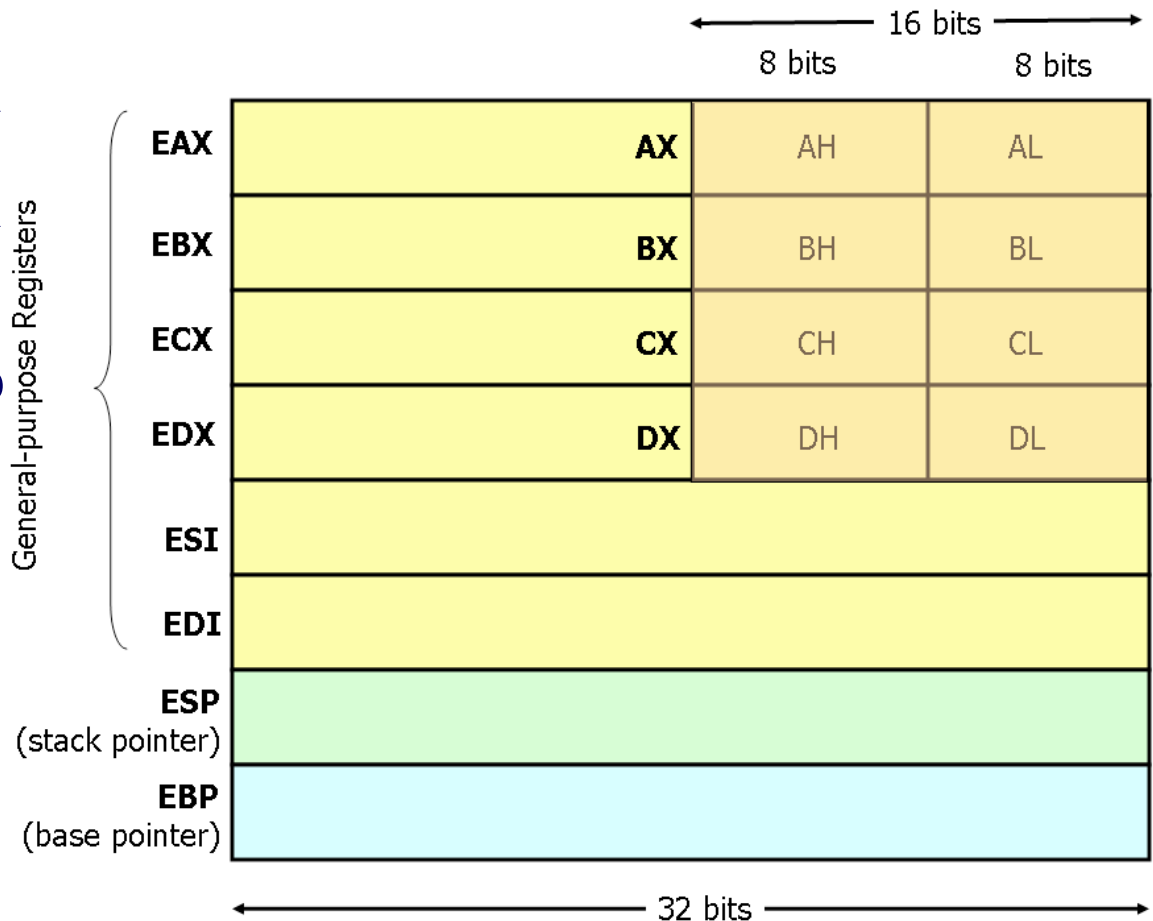
%edi

%esp

%ebp

More register names

ax	Low 16 bits of eax
bx	Low 16 bits of ebx
cx	Low 16 bits of ecx
dx	Low 16 bits of edx
si	Low 16 bits of esi
di	Low 16 bits of edi
bp	Low 16 bits of ebp
sp	Low 16 bits of esp
al	Low 8 bits of eax
ah	High 8 bits of ax
bl	Low 8 bits of ebx
bh	High 8 bits of bx
cl	Low 8 bits of ecx
ch	High 8 bits of cx
dl	Low 8 bits of edx
dh	High 8 bits of dx



x64 Register Names

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

movl Operand Combinations

	Source	Destination		C Analog
movl	Imm	Reg	movl \$0x4,%eax	temp = 0x4;
		Mem	movl \$-147,(%eax)	*p = -147;
	Reg	Reg	movl %eax,%edx	temp2 = temp1;
		Mem	movl %eax,(%edx)	*p = temp;
	Mem	Reg	movl (%eax),%edx	temp = *p;

- Cannot do memory-memory transfers with single instruction

Simple Addressing Modes

Normal **(R)** **Mem[Reg[R]]**

- Register R specifies memory address

```
movl (%ecx), %eax
```

Displacement **D(R)** **Mem[Reg[R]+D]**

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movl 8(%ebp), %edx
```

Using Simple Addressing Modes

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

} Set Up

```
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
```

} Body

```
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

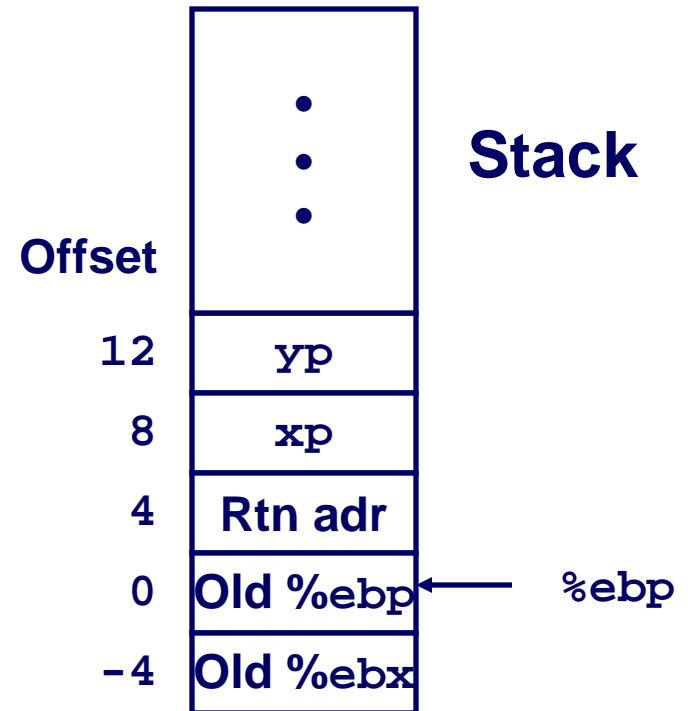
} Finish

Understanding Swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax   # eax = *yp (t1)
movl (%edx),%ebx   # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx
```



Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Offset		Address
			123	0x124
			456	0x120
				0x11c
				0x118
				0x114
yp	12		0x120	0x110
xp	8		0x124	0x10c
	4		Rtn adr	0x108
%ebp	→ 0			0x104
	-4			0x100

```

movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)     # *yp = ebx
    
```

Understanding Swap

%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp →	0	
	-4	
		0x104
		0x100

```

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax   # eax = *yp (t1)
movl (%edx),%ebx   # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx
    
```

Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
YP	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	0x108
		0x104
	-4	0x100

```

movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)     # *yp = ebx
    
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Offset		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	0x108
		0x104
	-4	0x100

```

movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)     # *yp = ebx
    
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Offset		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	0x108
		0x104
	-4	0x100

```

movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)     # *yp = ebx
    
```


Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Offset		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	0x108
		0x104
	-4	0x100

```

movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)    # *xp = eax
movl %ebx, (%ecx)     # *yp = ebx
    
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Offset		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	0x108
	-4	0x104
		0x100

```

movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)    # *yp = ebx
    
```

Indexed Addressing Modes

Most General Form

D(Rb,Ri,S)

Mem[Reg[Rb]+S*Reg[Ri]+ D]

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **Rb:** Base register: Any of 8 integer registers
- **Ri:** Index register: Any, except for %esp
 - Unlikely you’d use %ebp, either
- **S:** Scale: 1, 2, 4, or 8

Special Cases

(Rb,Ri)

Mem[Reg[Rb]+Reg[Ri]]

D(Rb,Ri)

Mem[Reg[Rb]+Reg[Ri]+D]

(Rb,Ri,S)

Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

Expression	Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

Address Computation Instruction

`leal Src, Dest`

- *Load Effective Address Long*
- *Src* is address mode expression
- Set *Dest* to address denoted by expression

Uses

- Computing address without doing memory reference
 - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form $x + k \cdot y$
 - $k = 1, 2, 4, \text{ or } 8$.
- Used heavily by compiler

Some Arithmetic Operations

Format

Computation

Two-Operand Instructions

addl	<i>Src, Dest</i>	<i>Dest = Dest + Src</i>	
subl	<i>Src, Dest</i>	<i>Dest = Dest - Src</i>	
imull	<i>Src, Dest</i>	<i>Dest = Dest * Src</i>	
sall	<i>k, Dest</i>	<i>Dest = Dest << k</i>	Also called shll
sarl	<i>k, Dest</i>	<i>Dest = Dest >> k</i>	Arithmetic
shrl	<i>k, Dest</i>	<i>Dest = Dest >> k</i>	Logical
k is an immediate value or contents of %cl			
xorl	<i>Src, Dest</i>	<i>Dest = Dest ^ Src</i>	
andl	<i>Src, Dest</i>	<i>Dest = Dest & Src</i>	
orl	<i>Src, Dest</i>	<i>Dest = Dest Src</i>	

Some Arithmetic Operations

Format

Computation

One-Operand Instructions

incl Dest

Dest = Dest + 1

decl Dest

Dest = Dest - 1

negl Dest

Dest = -Dest

notl Dest

Dest = ~Dest

Using `leal` for Arithmetic Expressions

```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

`arith:`

```
    pushl %ebp
    movl %esp,%ebp
```

} Set Up

```
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
```

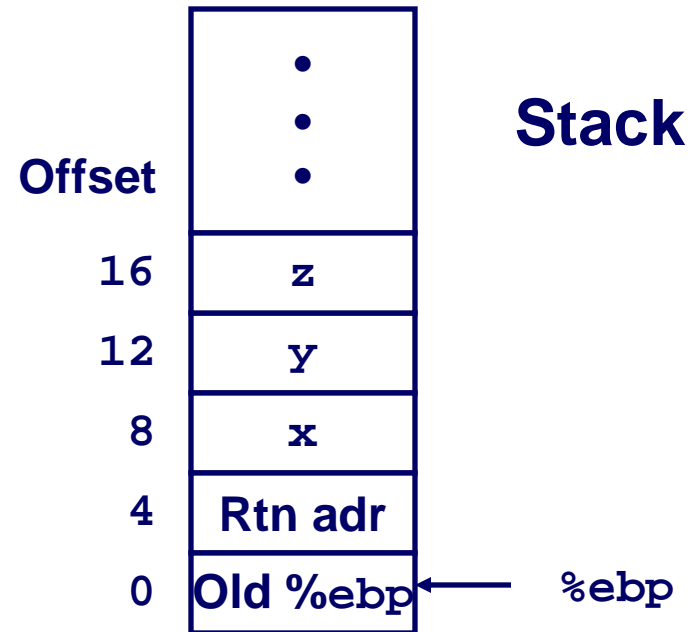
} Body

```
    movl %ebp,%esp
    popl %ebp
    ret
```

} Finish

Understanding arith

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```



```
movl 8(%ebp),%eax      # eax = x
movl 12(%ebp),%edx     # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = 3*y
sall $4,%edx          # edx = 48*y (t4)
addl 16(%ebp),%ecx     # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax        # eax = t5*t2 (rval)
```

Understanding arith

```
int arith
(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
# eax = x
movl 8(%ebp),%eax
# edx = y
movl 12(%ebp),%edx
# ecx = x+y (t1)
leal (%edx,%eax),%ecx
# edx = 3*y
leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
sall $4,%edx
# ecx = z+t1 (t2)
addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
imull %ecx,%eax
```

Another Example

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$$2^{13} = 8192, 2^{13} - 7 = 8185$$

logical:

```
    pushl %ebp
    movl %esp,%ebp          } Set Up

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax        } Body

    movl %ebp,%esp
    popl %ebp
    ret                    } Finish
```

```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
eax = x
eax = x^y      (t1)
eax = t1>>17  (t2)
eax = t2 & 8185
```

CISC Architecture Properties

Instruction can reference different operand types

- Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation

- $R_b + S * R_i + D$
- Useful for arithmetic expressions, too

Instructions can have varying lengths

- IA32 instruction size ranges from 1 to 15 bytes

Flags (condition codes)

	VM	RF		NT	IO PL	IO PL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
--	----	----	--	----	----------	----------	----	----	----	----	----	----	--	----	--	----	--	----

CF	Carry Flag	IOPL	I/O Privilege Level (286+ only)
PF	Parity Flag	IOPL	I/O Privilege Level (286+ only)
AF	Auxiliary Flag	NT	Nested Task Flag (286+ only)
ZF	Zero Flag	RF	Resume Flag (386+ only)
SF	Sign Flag	VM	Virtual Mode Flag (386+ only)
TF	Trap Flag (Single Step)		
IF	Interrupt		
DF	Direction Flag		
OF	Overflow flag		

Flag meaning

CF – Carry Flag

- Set by arithmetic instructions. Can be set, inverted and cleared with STC, CLC or CMC instructions respectively

PF – Parity Flag

- Set by most instructions if the least significant eight bits of the destination operand contain an even number of 1 bits.

AF – Auxiliary Flag

- If a carry or borrow from the most significant nibble of the least significant byte – Aids BCD arithmetic

ZF – Zero Flag

- Set by most instructions if the result of the arithmetic operation is zero

SF – Sign Flag

- On signed operands, this tells whether the result is positive or negative

TF – Trace Flag

- Allows single-step through programs. Executes exactly one instruction and generates an internal exception 1 (debug fault)

Flag meaning

IF – Interrupt Flag

- When set, the processor recognizes the external hardware interrupts on INTR pin. On clearing, anyway has not effect on NMI (external non maskable interrupt) pin or internally generated faults, exceptions, traps etc. This flag can be set and cleared using the STI and CLI instructions respectively

DF – Direction Flag

- Specifically for string instructions. DF = 1 increments ESI and EDI, while DF = 0 decrements the same. Set and cleared by STD and CLD instructions

OF – Overflow Flag

- Most arithmetic instructions set this flag to indicate that the result was at least 1 bit too large to fit in the destination

IOPL – Input Output Privilege Level flags (2 bits)

- For protected mode operations – indicates the privilege level, 0 to 3, at which your code must be running to execute any I/O-related instructions

NT – Nested Task Flag

- When set, it indicates that one system task has invoked another through a CALL instruction as opposed to a JMP. For multitasking ...

RF – Resume Flag

- It is related to Debug registers DR6 and DR7. By setting this, you can selectively mask some exceptions while you are debugging code

Branching/Calling

Jmp <label> **Jump**

J<cond> <label> **Conditional jump**

- *Je (equal), jne (not equal), jz, jg, jge, jl, jle*

Cmp src, dst ***Compare***

Call/ret instructions (push/pop PC in stack)

Calling Conventions

Caller-saved registers are EAX, ECX, EDX

Callee-saved registers are EBX, EDI, and ESI

- **What about ESP and EBP?**

Parameters in stack

Then return address (by call)

Then save EBP

`push ebp`

`mov ebp, esp`

Then make space for local variables

`sub esp, 12 ; For 3 integers`

myFunc PROC

; Subroutine Prologue

```
push ebp          ; Save the old base pointer value.  
mov ebp, esp      ; Set the new base pointer value.  
sub esp, 4        ; Make room for one 4-byte local variable.  
push edi          ; Save the values of registers that the function  
push esi          ; will modify. This function uses EDI and ESI.  
; (no need to save EBX, EBP, or ESP)
```

; Subroutine Body

```
mov eax, [ebp+8]   ; Move value of parameter 1 into EAX  
mov esi, [ebp+12]  ; Move value of parameter 2 into ESI  
mov edi, [ebp+16]  ; Move value of parameter 3 into EDI  
mov [ebp-4], edi   ; Move EDI into the local variable  
add [ebp-4], esi   ; Add ESI into the local variable  
add eax, [ebp-4]   ; Add the contents of the local variable  
; into EAX (final result)
```

; Subroutine Epilogue

```
pop esi           ; Recover register values  
pop edi  
mov esp, ebp      ; Deallocate local variables  
pop ebp          ; Restore the caller's base pointer value  
ret
```

Register names and usage (32-bit naming convention)

eax: Accumulator register, used in arithmetic operations, stores function return value

ebx: Base register, used as a pointer to data

ecx: Counter register, used in shift/rotate operations and loops

edx: Data register, used in arithmetic operations and I/O operations

Register names and usage (32-bit naming convention)

esi: Source index register, used as a pointer to a source in stream/string operations

edi: Destination index register, used as a pointer to a destination in stream/string operations

ebp: Stack base pointer register, used to point to the base of the stack

esp: Stack pointer register, used to point to the top of the stack

cdecl calling convention

1. Function parameters are passed onto the stack from right to left
2. Each function first saves the old stack pointer and sets up a new stack frame
3. Register `eax` is used for the return value
4. The caller cleans up the stack
5. Registers `eax`, `ecx`, and `edx` are caller-saved, the rest are callee-saved.

Intel and AT&T syntax differences

Prefixes

Order of operands

Memory operands and complex operations

Suffix

Prefixes

AT&T

`movl $1, %eax`

Intel

`mov eax, 1`

Prefixes

AT&T

movl \$1, %eax

Immediate
prefix

Register
prefix

Intel

mov eax, 1

No immediate
or register
prefix

Prefixes

AT&T

`movl $1, %eax`

`movl $0xff, %ebx`

Intel

`mov eax, 1`

`mov ebx, 0ffh`

←
'0' prefix if hex
value starts
with letter

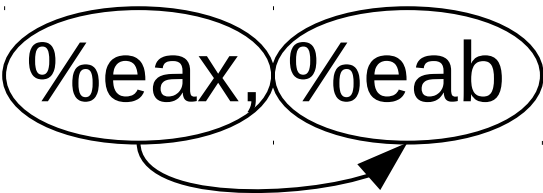
↘
'b' and 'h'
suffixes for
binary and hex
immediates,
respectively

Order of operands

AT&T

instr source, dest

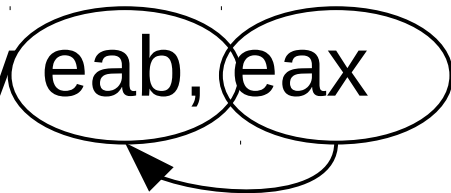
movl %eax, %eax



Intel

instr dest, source

mov eab, eax



Memory operands and complex operations

AT&T

mov (%ebx), eax

disp(base, index, scale)

mov 0x20(%ebx), %eax

Intel

mov eax, [ebx]

[base+index*scale+disp]

mov eax, [ebx+20h]

Suffix

AT&T

Suffixes used for operand sizes.

b: byte (8 bits)

s: short (16 bits)

w: word (16 bits)

l: long (32 bits)

q: quad (64 bits)

`movb %al, %bl`

`movw %ax, %bx`

`movl (%eax), %ebx`

Intel

Similar directives but not necessary since it can be inferred by the size of the destination register.

byte ptr

word ptr

dword ptr

`mov bl, al`

`mov bx, ax`

`mov ebx, dword ptr [eax]`