Sistemi Operativi

Laboratorio – linea 2

1

Matteo Re

Dip. di Informatica
Università degli studi di Milano

matteo.re@unimi.it

http://homes.di.unimi.it/re/solabL2.html

Introduzione al laboratorio

Sistemi Operativi

Laboratorio – linea 2

1

6 (Bruschi) + 4 (Re) ore di lezione settimanali
(12 crediti)
Lezioni di teoria e in laboratorio

Esame:
Scritto con domande a risposta multipla + orale
Prova pratica per la parte di laboratorio

Libro di testo: Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau "Operating Systems: Three Easy Pieces", 2015 (Testo di riferimento) http://ostep.org/

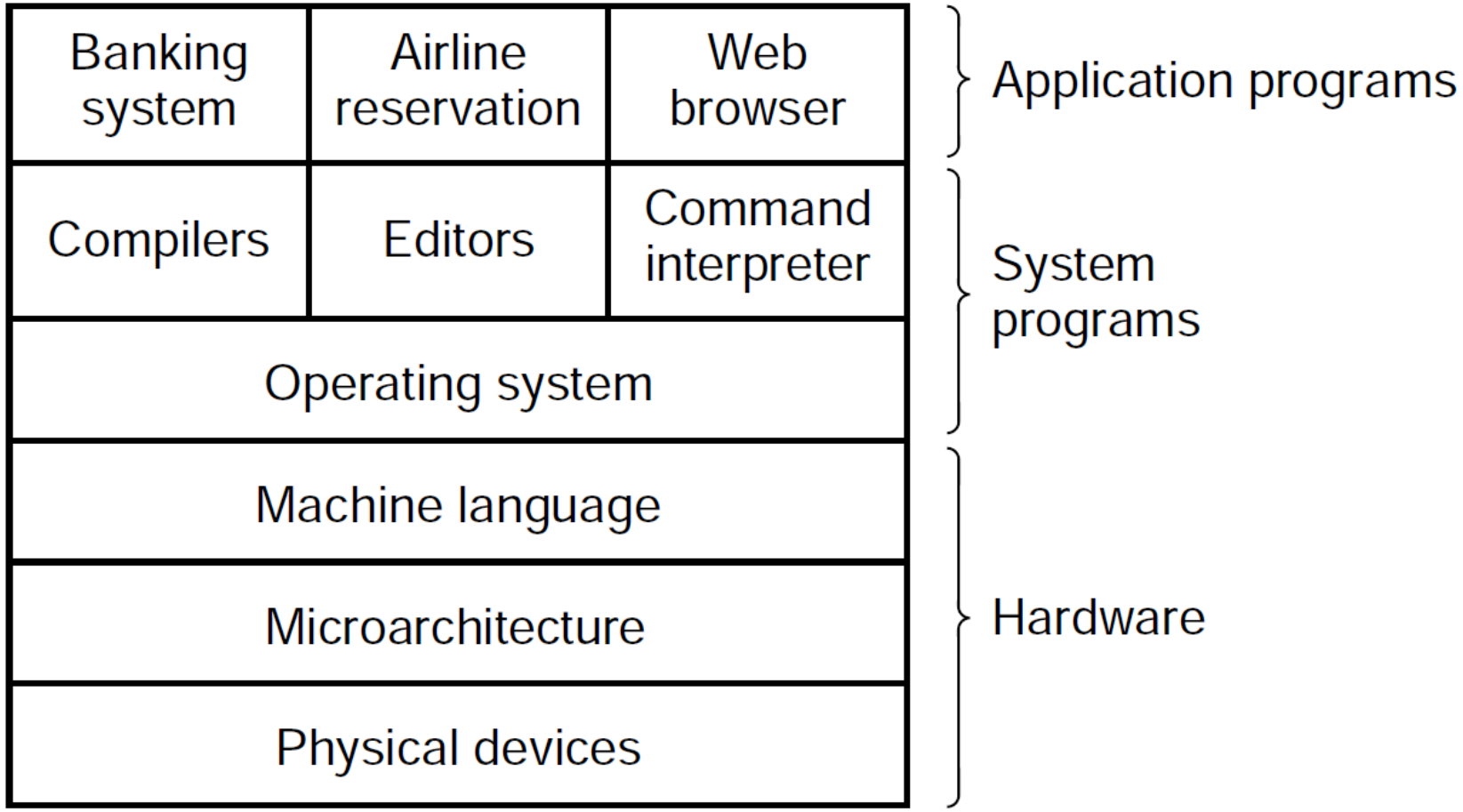
http://homes.di.unimi.it/sisop/
http://homes.di.unimi.it/re/solabL2.html

... COS'E' un sistema operativo?

È un insieme di programmi che:

- Gestisce in modo ottimale le risorse (finite) della macchina.
- Facilita a utenti/programmatori l'utilizzo della sottostante macchina hardware.

VISIONE ASTRATTA delle componenti di un sistema di calcolo.



(modello a "cipolla")

Kernel mode / User mode

- Il s.o. è l'unico programma che esegue con il **totale controllo** delle risorse hardware (**kernel mode**).
- Gli altri programmi si appoggiano unicamente sui servizi del s.o. e la loro esecuzione è **gestita e controllata** dal s.o. (**user mode**)
- In molti processori questa separazione è imposta via hardware

Esecuzione di un programma

Algoritmo : descrizione priva di ambiguità di una attività di elaborazione dell'informazione. Necessità la specifica di un interprete che

- dato un set di istruzioni
- dato un set di dati di input

esegua l'algoritmo

Interprete : può essere un concetto astratto o un dispositivo reale (ad es. un processore IA-32) . E' in grado di riconoscere un set finito di istruzioni, è quindi necessario osservare delle convenzioni sintattiche durante la scrittura dell'algoritmo se Vogliamo che esso possa essere eseguito dall'interprete.

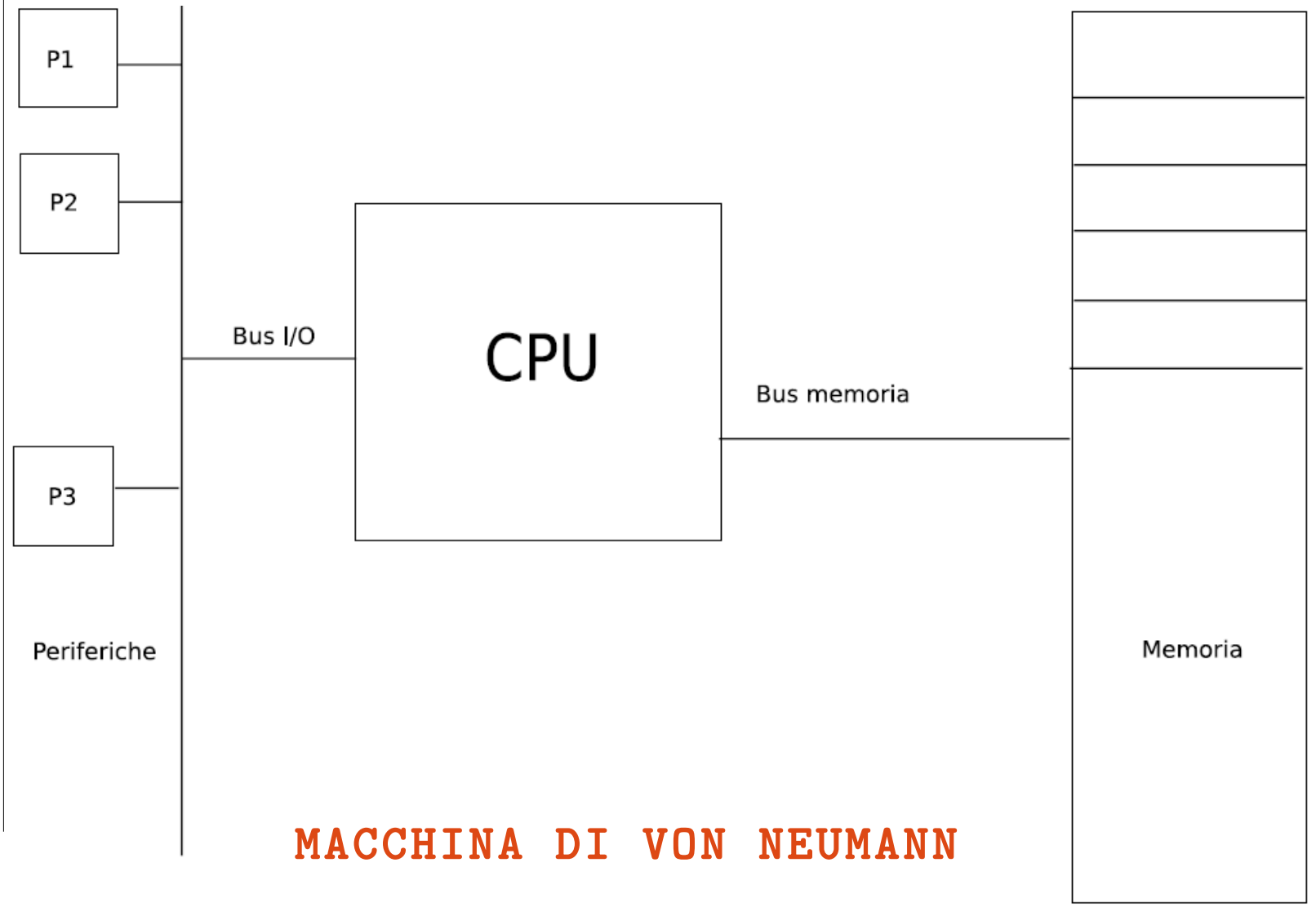
Esecuzione di un programma

Linguaggio di programmazione ad alto livello (C, C++)

→ Espansione in una serie di “operazioni elementari”

Linguaggio macchina **(eseguibile da una CPU)**

Per cercare di capire come un generico algoritmo possa essere eseguito da un calcolatore consideriamo un modello “semplificato” del processore IA-32, la **Macchina di Von Neumann**



MACCHINA DI VON NEUMANN

MACCHINA DI VON NEUMANN

Concetti fondamentali:

- La **memoria** viene utilizzata per conservare sia il programma che i dati sui quali esso opera
- Lo hardware opera secondo il ciclo **fetch**, **decode**, **execute**

FETCH: recupera dalla memoria la prossima istruzione da eseguire

DECODE: decodifica il significato dei bit (codice istruzione)

EXECUTE: esecuzione istruzione ricavata dalla codifica

MACCHINA i386

Evoluzione

- 1978 progenitore 8086 (architettura x86)
- **3**86 indica la terza generazione di CPU della famiglia x86. La i in i386 indica il produttore (intel)
- Prima CPU a **32 bit** (anno inizio produzione: 1985)

Complex Instruction Set Computer (CISC)

- Molte istruzioni differenti ognuna con diverse modalità di utilizzo (es. numero diverso di parametri)

MACCHINA i386

- Diversi **registri**
- **Real** and **Protected** mode

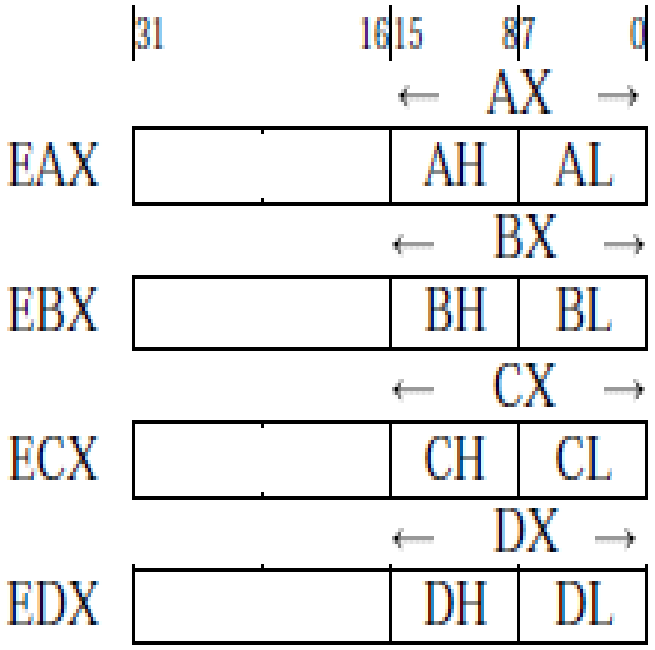
Real vs Protected mode

	Real mode	32-bit Protected mode
Protezioni hw	no	sí
Spazio di indirizzamento	2^{20}	2^{32}

- Real mode: memoria max 2^{20} byte, indirizzo ottenuto con due registri a 16 (`SS:OFFSET`)
  $indirizzo = 16 * selettore + offset$
  - ci sono piú modi per riferirsi allo stesso indirizzo: `07C0:0000` e `0000:7C00` sono la stessa locazione fisica.
  - A20 gate
- Protected mode: il segmento è stabilito da un descrittore (che può essere cambiato solo in kernel mode)

MACCHINA i386 : register set

General Purpose Registers



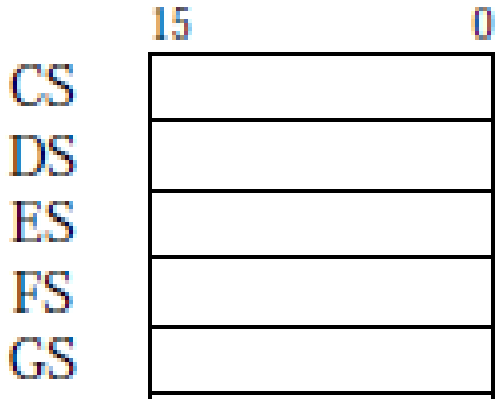
Index Registers

Pointer Registers

MACCHINA i386 : register set

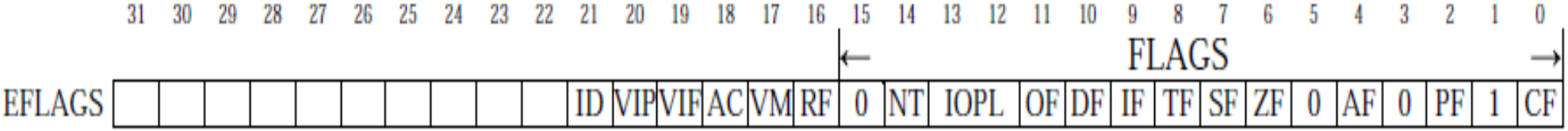
Instruction Pointer

Segment Registers



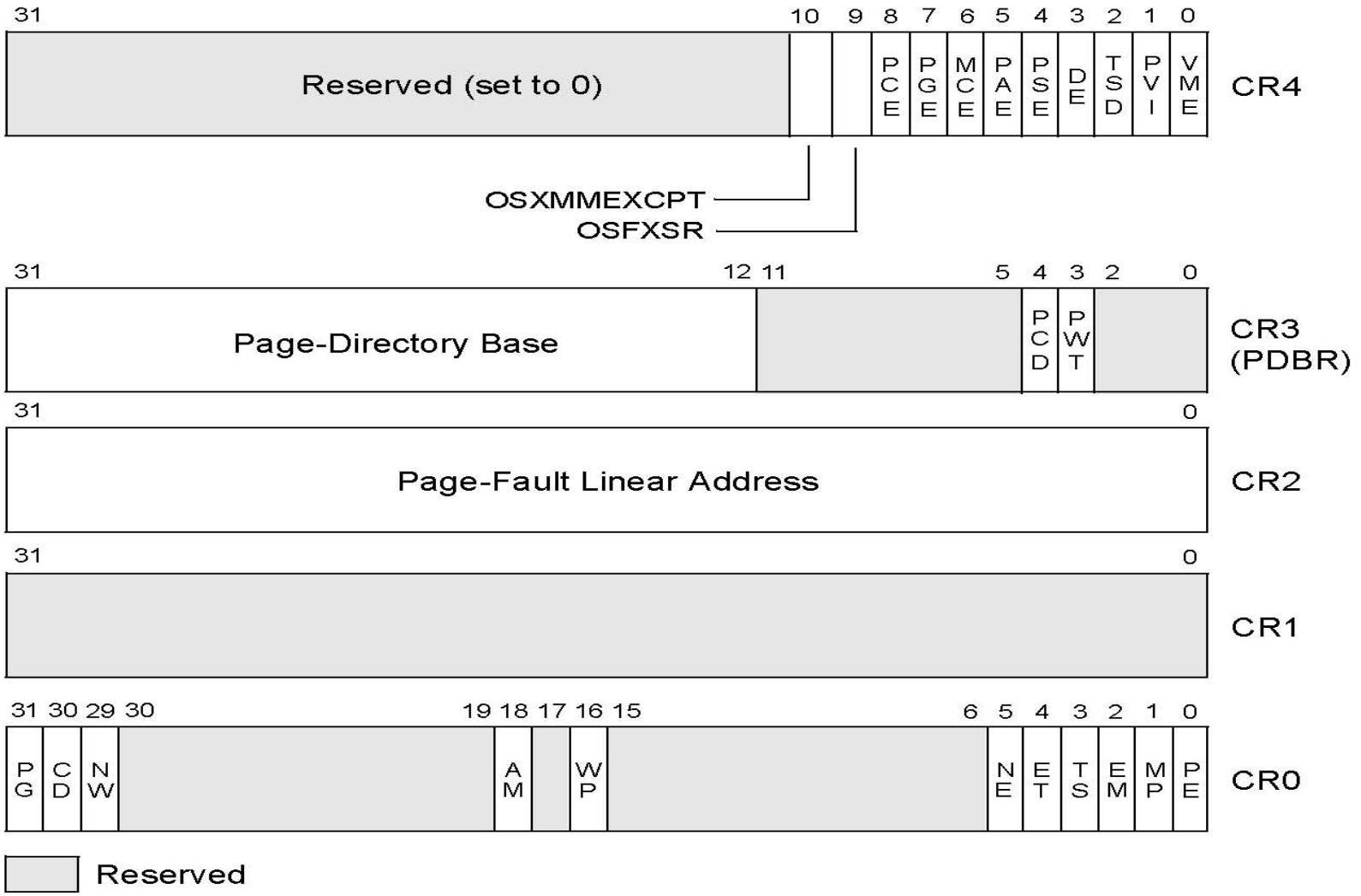
MACCHINA i386 : register set

Flags

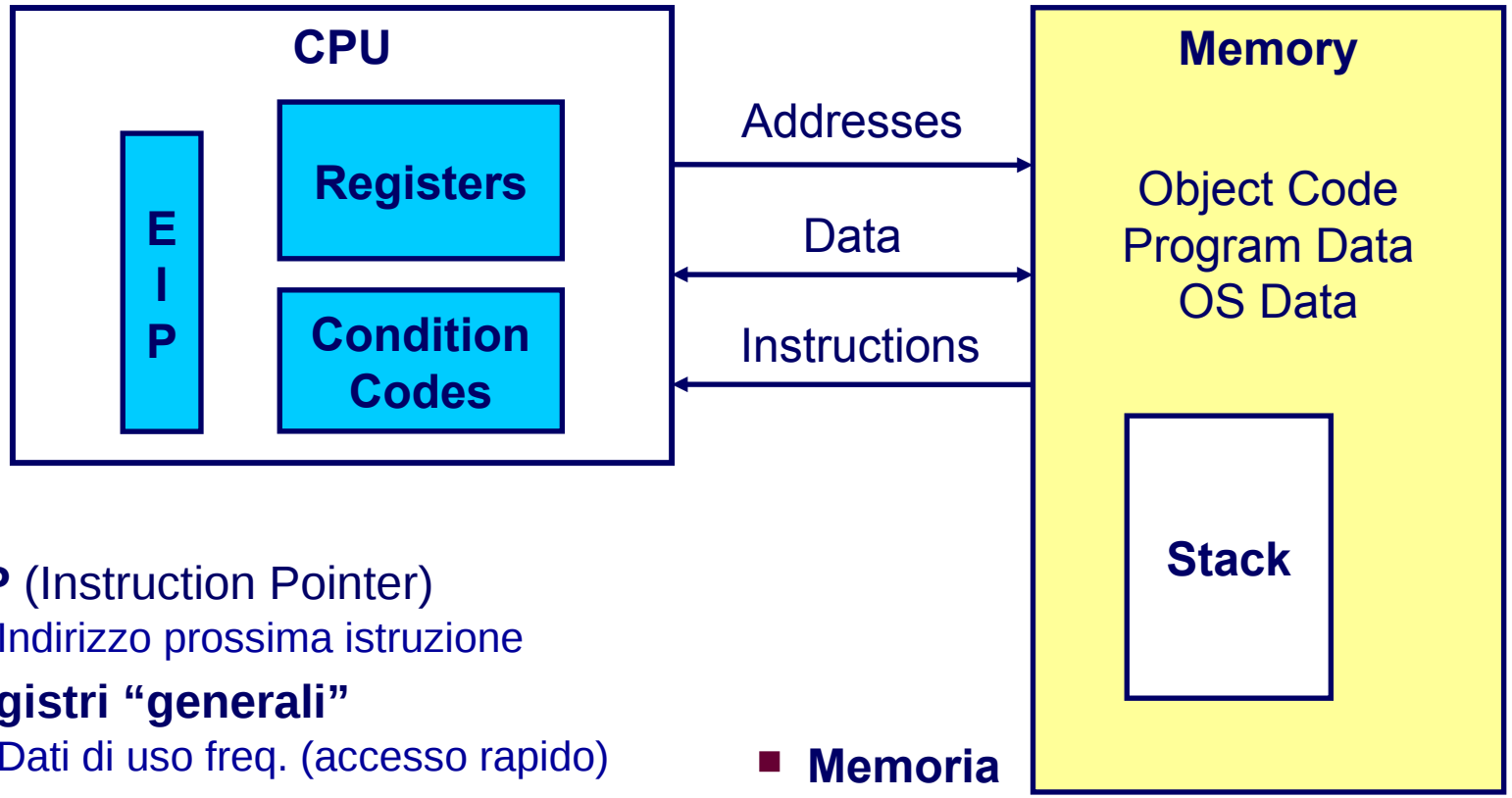


Bit		Flag	Description
0	CF	Carry Flag	Carry from most significant bit, also borrow for most significant bit; can be considered as overflow in unsigned instructions.
2	PF	Parity Flag	Set to 1 if 8 less significant bits of result have even number of 1's, else set to 0.
4	AF	Auxiliary carry Flag	Used as carry flag in BCD instructions.
6	ZF	Zero Flag	Set to 1 if result is zero, else set to 0.
7	SF	Sign Flag	Set to 1 if result is negative (below zero), else set to 0.
8	TF	Trap Flag	Used by debuggers.
9	IF	Interrupt Flag	If set to 1, then interrupts are enabled, else are disabled.
10	DF	Direction Flag	When set to 0, string instructions increment the index registers, else – decrement the index registers.
11	OF	Overflow Flag	Used in signed instructions.
12,13	IOPL	I/O Privilege Level	Indicates the I/O privilege level of the currently running program or task.
14	NT	Nested Task	Controls the chaining of interrupt and called tasks.
16	RF	Resume Flag	Controls the processor's response to instruction-breakpoint conditions.
17	VM	Virtual 8086 Mode	Set to enable virtual-8086 mode; clear to return to protected mode.
18	AC	Alignment check	Set this flag and the AM flag in control register CR0 to enable alignment checking of memory references.
19	VIF	Virtual Interrupt Flag	Contains a virtual image of the IF flag.
20	VIP	Virtual Interrupt Pending	Set by software to indicate that an interrupt is pending.
21	ID	Identification	The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

MACCHINA i386 : register set (CONTROL registers)



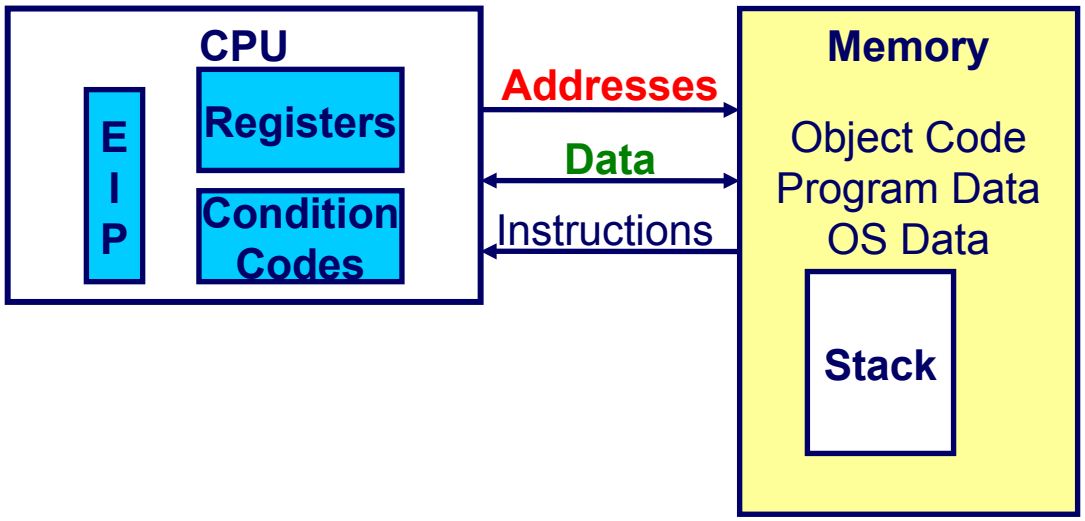
MACCHINA i386 : il punto di vista del programmatore

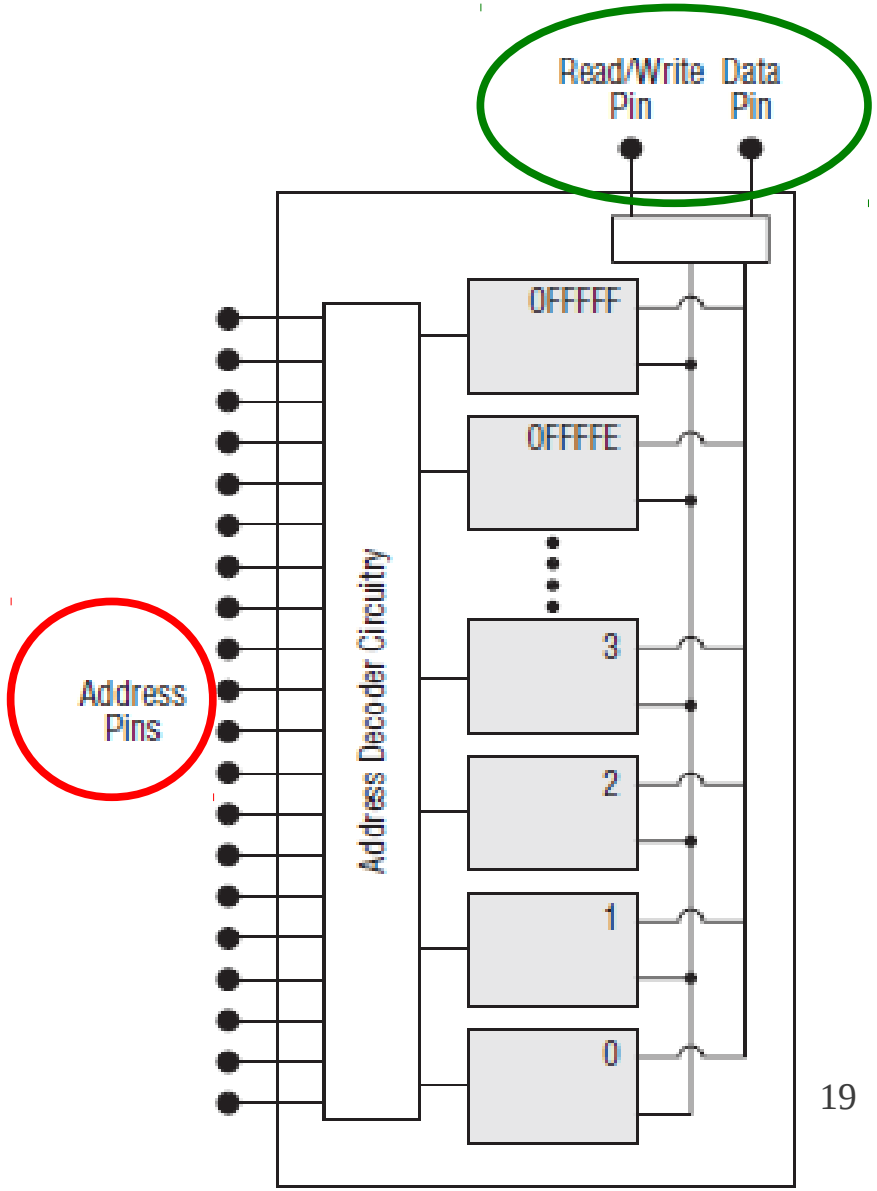


- **EIP** (Instruction Pointer)
  - Indirizzo prossima istruzione
- **Registri "generali"**
  - Dati di uso freq. (accesso rapido)
- **Flag di stato**
  - Informationi riguardo alle più recenti operazioni aritmetiche/logiche
  - Controllo flusso

- **Memoria**
  - **Array ordinato di byte**
  - **Codice, dati (user), dati s.o.**
  - **Stack (supporto procedure)**

"dialogo" tra CPU e memoria



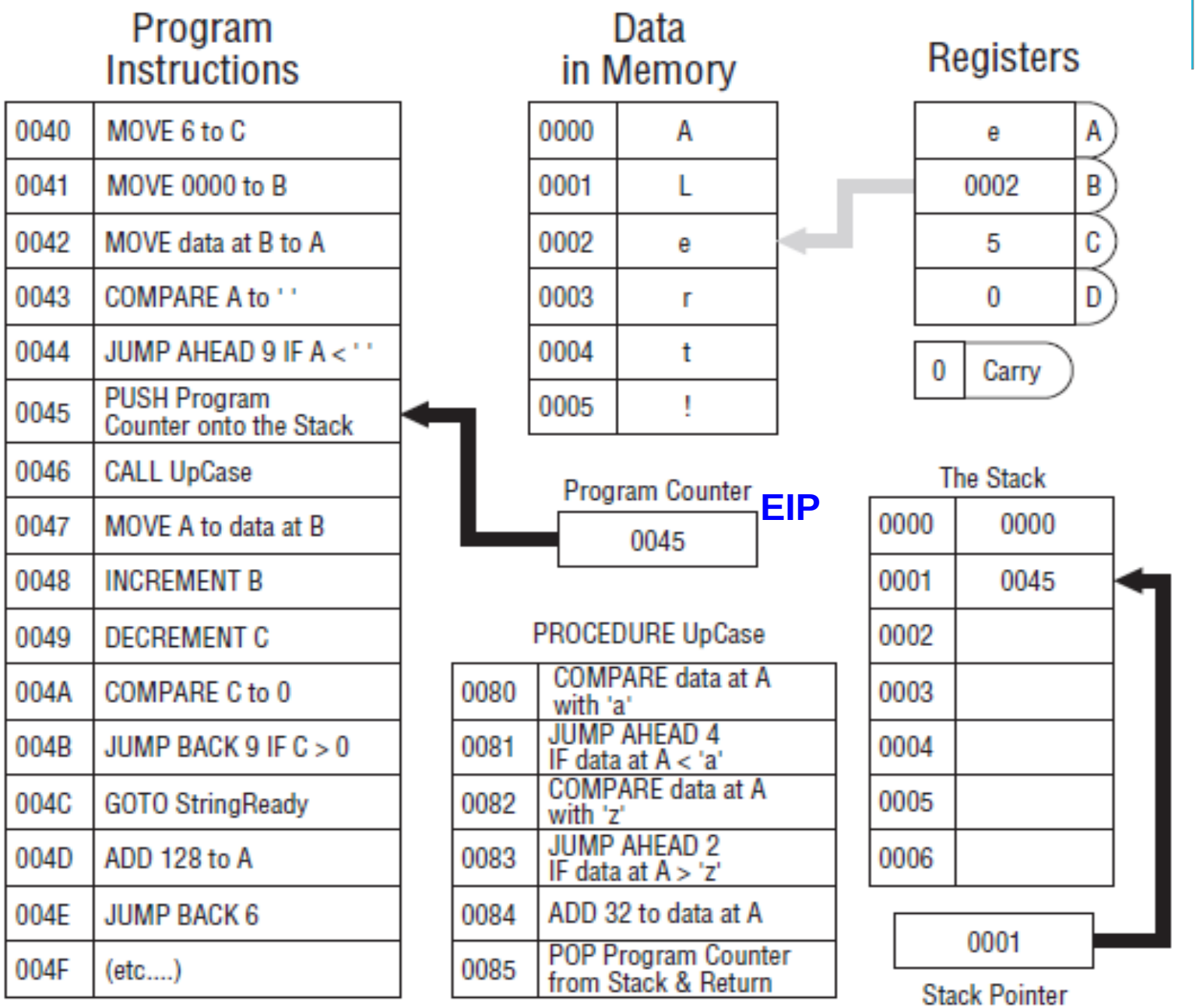


SCHEMA ad alto livello di esecuzione di un programma, in Assembly

WARNING:

- questo **NON** è codice assembly

- questo schema **NON** rappresenta una particolare architettura



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

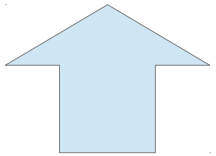
Assembly: diverse famiglie sintattiche (intel , AT&T)

- NASM, http://nasm.sourceforge.org
- PC Assembly Language, by Paul A. Carter http://www.drpaulcarter.com/pcasm/
- Un altro assemblatore molto diffuso è gas (http://www.ibm.com/developerworks/linux/library/l-gas-nasm/index.html)

```
1      mov eax, 3 ; eax = 3
2      mov bx, ax ; bx = ax
3      add eax, 4 ; eax = eax + 4
4      add al, ah ; al = al + ah
5  L8:db "A" ; *L8 = 'A'
6      mov al, [L8] ; al = *L8
```

Assembly: diverse famiglie sintattiche (intel , AT&T)

- **OPCODE** : singola istruzione che può essere eseguita dalla CPU. In linguaggio macchina è un valore rappresentabile come un numero binario o esadecimale, ad es. **B6**
- In linguaggio Assembly è una parola chiave più facile da memorizzare per il programmatore. Ad es. **MOV, ADD, SUB, JMP...**
- **mov eax, 34**



COMMAND DST, SRC

(intel)

Il comando è sempre presente

n. operandi: da 0 a 3 (nei casi + comuni)

Assembly: diverse famiglie sintattiche (intel , AT&T)

Intel (nasm)	AT&T (as86, gas)
mov ebx, eax	movl %**eax**, %**ebx**
mov eax, 42	movl $42, %**eax**
mov [ebx], eax	movl %**eax**, 0(%**ebx**)
mov [ebx+4**], eax**	movl %**eax**, 4(%**ebx**)
mov byte [ebx], al	movb %**eax**, 0(%**ebx**)
call eax	**call** ∗%**eax**

Caratteristiche linguaggio Assembly

Tipi di dati minimali :

- Integer
- Floating-point
- **NON ESISTONO** tipi aggregati (array, struct)
  - Solo byte allocati in modo contiguo in memoria

Operazioni primitive :

- Operazioni aritmetiche su registri e dati in memoria
- Trasferimento dati tra memoria e registri
  - Copia da memoria a registro, da registro a memoria, da registro a registro, ma **NON** da memoria a memoria!
- Controllo di flusso :
  - Salti incondizionati da e verso procedure
  - Salti condizionali

versione ridotta … solo alcuni esempi

IA-32 instruction set (convenzioni operandi)

r8	8-bit general purpose register
r16	16-bit general purpose register
r32	16-bit general purpose register
EDX:EAX	64-bit integer number, EDX – more significant part, EAX – less significant part

imm8	immediate 8-bit value from –128 to 127
imm16	immediate 16-bit value from –32768 to +32767
imm32	immediate 32-bit value from –2147483648 to +2147483647

r/m8	8-bit general purpose register or memory location
r/m16	16-bit general purpose register or memory location
r/m32	32-bit general purpose register or memory location

m	16-bit or 32-bit memory location
m8	8-bit memory location
m16	16-bit memory location
m32	32-bit memory location
m64	64-bit memory location
m128	128-bit memory location
m*N*byte	*N*-byte memory location

m16:16	a memory location containing a far pointer composed of two 16-bit numbers: segment & offset
m16:32	a memory location containing a far pointer composed of numbers: 16-bit segment & 32-bit offset
m16&16	a memory location containing a data pair: 16&16-bit
m16&32	a memory location containing a data pair: 16&32-bit
m32&32	a memory location containing a data pair: 32&32-bit

moffs8	simple 8-bit memory location, which actual address is given by a simple offset relative to segment base
moffs16	simple 16-bit memory location, which actual address is given by a simple offset relative to segment base
moffs32	simple 32-bit memory location, which actual address is given by a simple offset relative to segment base

Sreg	segment register: CS, DS, SS, ES, FS or GS

IA-32 instruction set (Trasferimento dati I)

Instruction	Mnemonic	Operands	Description
Move	MOV	r/m8, r8	Move *r8* to *r/m8*.
		r/m16, r16	Move *r16* to *r/m16*.
		r/m32, r32	Move *r32* to *r/m32*.
		r8, r/m8	Move *r/m8* to *r8*.
		r16, r/m16	Move *r/m16* to *r16*.
		r32, r/m32	Move *r/m32* to *r32*.
		r/m16, Sreg	Move segment register to *r/m16*.
		Sreg, r/m16	Move *r/m16* to segment register.
		AL, moffs8	Move byte at (segment: offset) to AL.
		AX, moffs16	Move word at (segment: offset) to AX.
		EAX, moffs32	Move dword at (segment: offset) to EAX.
		moffs8, AL	Move AL to byte at (segment: offset).
		moffs16, AX	Move AX to word at (segment: offset).
		moffs32, EAX	Move EAX to dword at (segment: offset).
		r8, imm8	Move *imm8* to *r8*.
		r16, imm16	Move *imm16* to *r16*.
		r32, imm32	Move *imm32* to *r32*.
		r/m8, imm8	Move *imm8* to *r/m8*.
		r/m16, imm16	Move *imm16* to *r/m16*.
		r/m32, imm32	Move *imm32* to *r/m32*.

DST ← SRC

IA-32 instruction set (Trasferimento dati II)

Conditional Move	CMOVA	r16, r/m16	Move if above (CF=0 and ZF=0)
	CMOVAE	r32, r/m32	Move if above or equal (CF=0)
	CMOVB		Move if below (CF=1)
	CMOVBE		Move if below or equal (CF=1 or ZF=1)
	CMOVC		Move if carry (CF=1)
	CMOVE		Move if equal (ZF=1)
	CMOVG		Move if greater (ZF=0 and SF=OF)
	CMOVGE		Move if greater or equal (SF=OF)
	CMOVL		Move if less (SF<>OF)
	CMOVLE		Move if less or equal (ZF=1 or SF<>OF)
	CMOVNA		Move if not above (CF=1 or ZF=1)
	CMOVNAE		Move if not above or equal (CF=1)
	CMOVNB		Move if not below (CF=0)
	CMOVNBE		Move if not below or equal (CF=0 and ZF=0)
	CMOVNC		Move if not carry (CF=0)
	CMOVNE		Move if not equal (ZF=0)
	CMOVNG		Move if not greater (ZF=1 or SF<>OF)
	CMOVNGE		Move if not greater or equal (SF<>OF)
	CMOVNL		Move if not less (SF=OF)
	CMOVNLE		Move if not less or equal (ZF=0 and SF=OF)
	CMOVNO		Move if not overflow (OF=0)

```
TMP ← SRC
IF(condition) THEN
  DST ← TMP
END
```

IA-32 instruction set (manipolazione stack)

(push sullo stack)

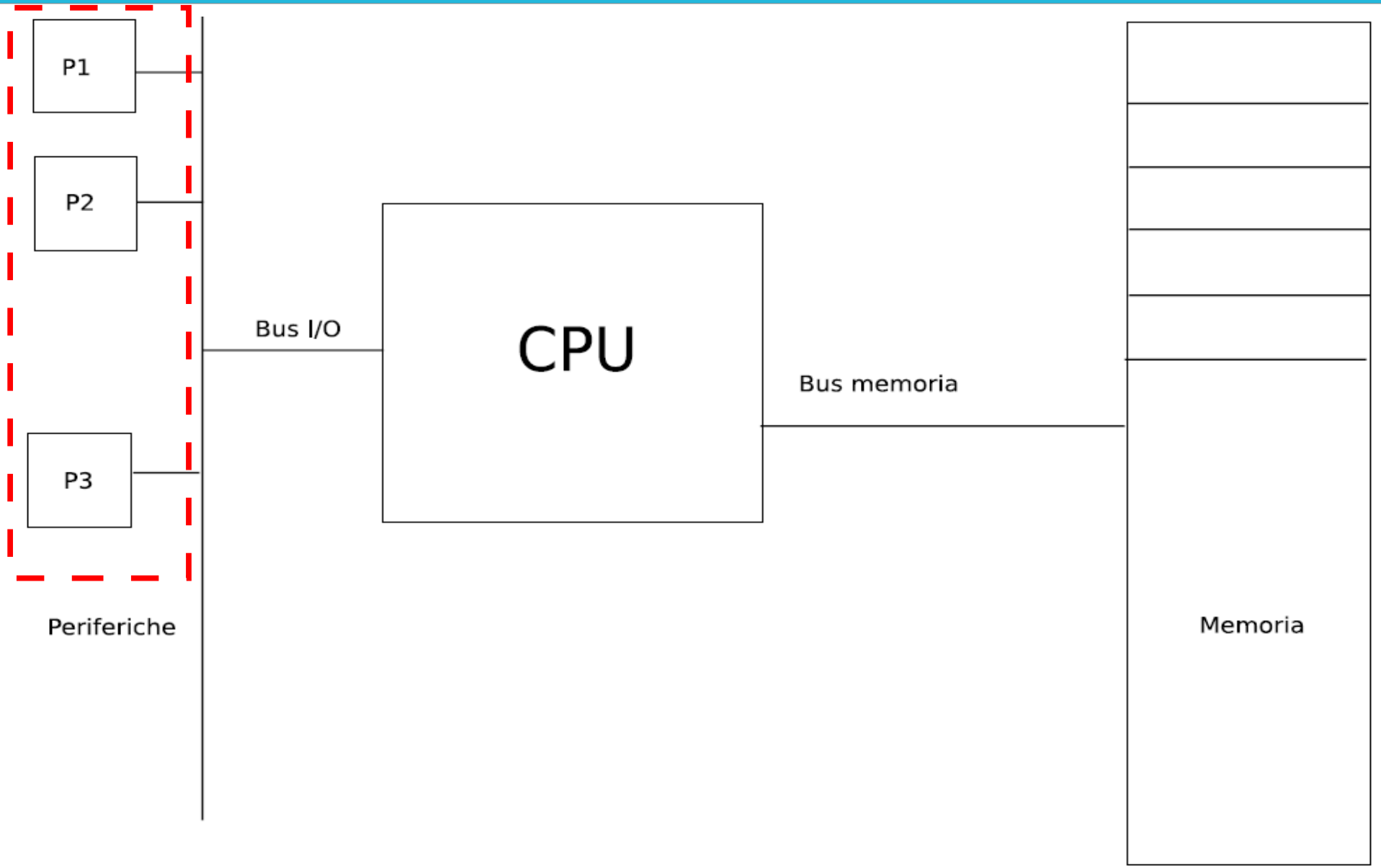
Push onto Stack	PUSH	r/m16 r/m32 imm8 imm16 imm32 DS ES SS FS GS	Decrements stack pointer. Pushes register, memory or immediate value to the top of stack into register or memory, increment stack pointer. ESP ← ESP – OPERANDSIZE/8 SS[ESP] ← SRC

(pop dallo stack)

Pop from stack	POP	r/m16 r/m32 DS ES SS FS GS	Pops top of stack into register or memory, increments stack pointer. DST ← SS[ESP] ESP ← ESP + OPERANDSIZE/8

IA-32 instruction set (I/O porte)

Fermi tutti … **COSA E'** una porta ?


P1
P2
P3
Periferiche
Bus I/O
CPU
Bus memoria
Memoria

Cosa si intende per PORTA ?

Ogni periferica è dotata di un controller. Il controller avrà registri che conservano lo stato della periferica. Come accedere (leggere o scrivere) al contenuto dei registri?

1. Spazi di indirizzamento separati chiamati port. Vi si accede con istruzioni particolari:
   - **out** port, **eax**
   - **in eax**, port
2. Memory-mapped I/O, lo spazio di indirizzamento è unico
   - **mov** [address], **eax**
   - **mov eax**, [address]

IA-32 instruction set (I/O porte)

DST ← Port(SRC)

Input from port	IN	AL, imm8	Inputs byte from given I/O port address into AL.
		AX, inn8	Inputs word from given I/O port address into AX.
		EAX, imm8	Inputs dword from given I/O port address into EAX.
		AL, DX	Inputs byte from I/O port specified in DX into AL.
		AX, DX	Inputs word from I/O port specified in DX into AX.
		EAX, DX	Inputs dword from I/O port specified in DX into EAX.
Output from port	OUT	imm8, AL	Outputs byte from AL to given I/O port address.
		inn8, AX	Outputs word from AX to given I/O port address.
		imm8, EAX	Outputs dword from EAX to given I/O port address.
		DX, AL	Outputs byte from AL to I/O port specified in DX.
		DX, AX	Outputs word from AX to I/O port specified in DX.
		DX, EAX	Outputs dword from EAX to I/O port specified in DX.

Port(DST) ← SRC

IA-32 instruction set

(Aritmetica binaria … solo alcuni esempi)

Increment	INC	r/m8 r/m16 r/m32	Adds 1 to the destination operand, while preserving the state of the CF flag.
Decrement	DEC	r/m8 r/m16 r/m32	Subtracts 1 from the destination operand, while preserving the state of the CF flag.
Add	ADD	AL, imm8 AX, imm16 EAX, imm32 r/m8, imm8 r/m16, imm16 r/m32, imm32 r/m16, imm8 r/m32, imm8 r/m8, r8 r/m16, r16 r/m32, r32 r8, r/m8 r16, r/m16 r32, r/m32	Adds source (second) operand to the destination (first) operand.
Add with Carry	ADC		Adds source (second) operand with CF to the destination (first) operand.
Subtract	SUB		Subtracts source (second) operand from the destination (first) operand.
Subtract with Borrow	SBB		Subtracts source (second) operand with CF from the destination (first) operand.
Compare	CMP		Compares two operands by subtracting the second operand from the first operand and then setting the status flag in the same manner as the SUB instruction.

IA-32 instruction set (istruzioni logiche)

Logical Negation	NOT	r/m8 r/m16 r/m32	Reverses each bit of the operand
Logical AND	AND	AL, imm8 AX, imm16 EAX, imm32 r/m8, imm8 r/m16, imm16 r/m32, imm32 r/m16, imm8 r/m32, imm8 r/m8, r8 r/m16, r16 r/m32, r32 r8, r/m8 r16, r/m16 r32, r/m32	Performs a bitwise AND operation on the destination (first) and source (second) operands and stored the result in the destination operand location.
Logical Inclusive OR	OR		Performs a bitwise OR operation on the destination (first) and source (second) operands and stored the result in the destination operand location.
Logical Exclusive OR	XOR		Performs a bitwise XOR operation on the destination (first) and source (second) operands and stored the result in the destination operand location.

IA-32 instruction set
(trasferimento controllo I)

Instruction	Mnemonic	Operands	Description
Jump	JMP	rel8 rel16 rel32	Jumps near, relative, displacement relative to the next instruction (E)IP ← (E)IP + DST
		r/m16 r/m32	Jumps near, absolute indirect, address given in the register or memory location. (E)IP ← DST
		ptr16:16 ptr16:32	Jumps far, absolute, address given in operand.
		m16:16 m16:32	Jumps far, absolute indirect, address given in memory location.

IA-32 instruction set
(trasferimento controllo II)

Jump if condition	JA	rel8	Jumps near, relative, if above (CF=0 and ZF=0)
	JAE		Jumps near, relative, if above or equal (CF=0)
	JB		Jumps near, relative, if below (CF=1)
	JBE		Jumps near, relative, if below or equal (CF=1 or ZF=1)
	JC		Jumps near, relative, if carry (CF=1)
	JE		Jumps near, relative, if equal (ZF=1)
	JG		Jumps near, relative, if greater (ZF=0 and SF=OF)
	JGE		Jumps near, relative, if greater or equal (SF=OF)
	JL		Jumps near, relative, if less (SF<>OF)
	JLE		Jumps near, relative, if less or equal (ZF=1 or SF<>OF)
	JNA		Jumps near, relative, if not above (CF=1 or ZF=1)
	JNAE		Jumps near, relative, if not above or equal (CF=1)
	JNB		Jumps near, relative, if not below (CF=0)
	JNBE		Jumps near, relative, if not below or equal (CF=0 and ZF=0)
	JNC		Jumps near, relative, if not carry (CF=0)
	JNE		Jumps near, relative, if not equal (ZF=0)
	JNG		Jumps near, relative, if not greater (ZF=1 or SF<>OF)
	JNGE		Jumps near, relative, if not greater or equal (SF<>OF)
	JNL		Jumps near, relative, if not less (SF=OF)
	JNLE		Jumps near, relative, if not less or equal (ZF=0 and SF=OF)
	JNO		Jumps near, relative, if not overflow (OF=0)
	JNP		Jumps near, relative, if not parity (PF=0)
	JNS		Jumps near, relative, if not sign (SF=0)
	JNZ		Jumps near, relative, if not zero (ZF=0)
	JO		Jumps near, relative, if overflow (OF=1)
	JP		Jumps near, relative, if parity (PF=1)
	JPE		Jumps near, relative, if parity even (PF=1)
	JPO		Jumps near, relative, if parity off (PF=0)
	JS		Jumps near, relative, if sign (SF=1)
	JZ		Jumps near, relative, if zero (ZF=1)

IF condition THEN
(E)IP ← (E)IP + DST

Dal codice C al codice macchina (strumenti utilizzati) :

- Compilatore
- Assembler
- Linker

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Simile alla espressione
y += x

```
0x401046:    03 45 08
```

Codice C

- Addizione di due interi con segno

Assembly

- Addizione di 2 interi (da 4-byte)
  - "Long" words
  - Stesse istruzioni per interi con e senza segno
- Operandi:
  - `y`: Registro `%eax`
  - `x`: Memoria M[`%ebp+8`]
  - `t`: Registro `%eax`
    - » Ritorna valore in `%eax`

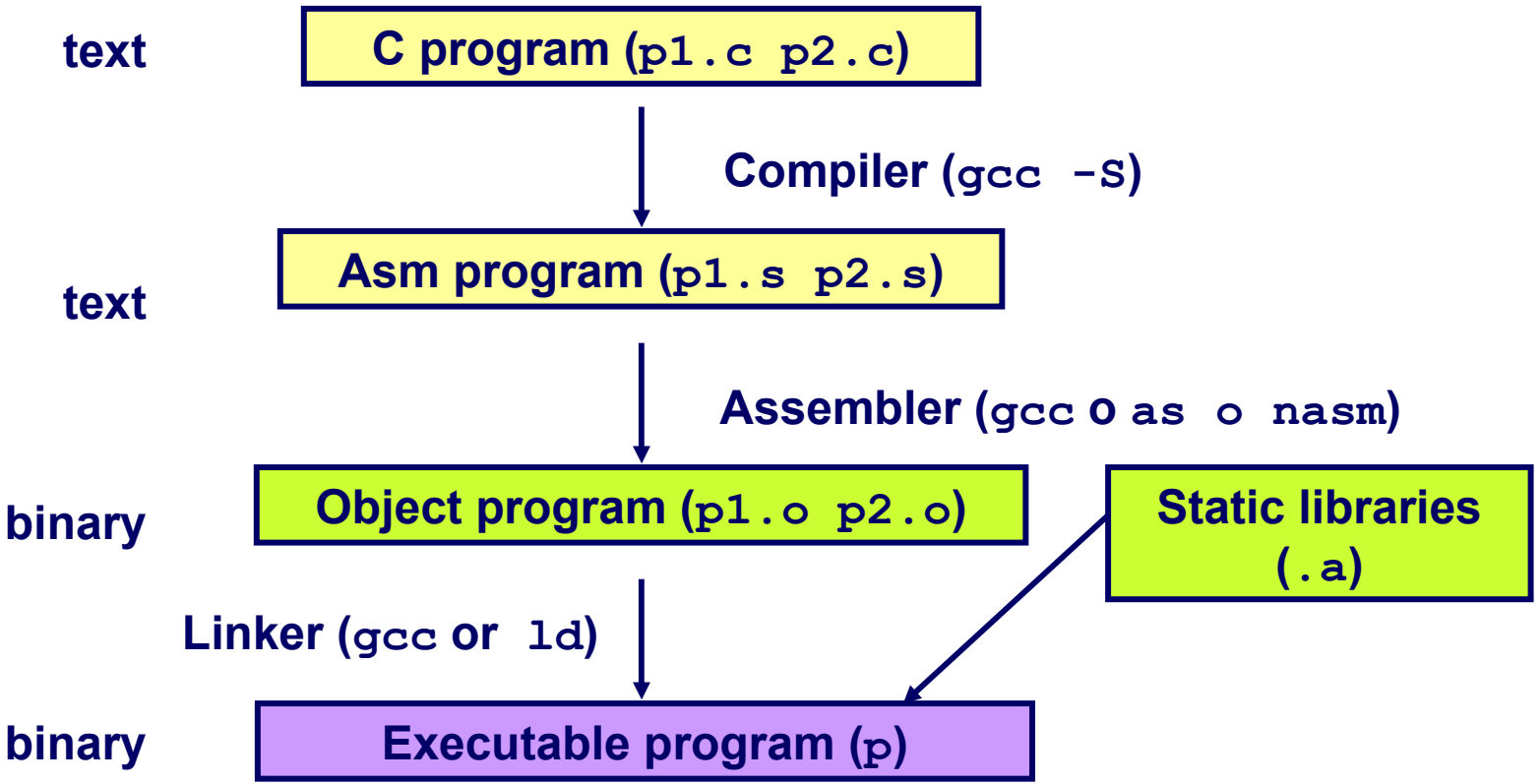
Codice oggetto

- Istruzione in 3-byte
- Salvata all'indirizzo `0x401046`

Object code

Dal codice ad alto livello al codice macchina:

- Sorgenti in file p1.c p2.c
- Compilazione: gcc -O p1.c p2.c -o p
  - Usa ottimizzazione (-O)
  - Mette il risultante codice binario in eseguibile **p**



Generare codice assembly a partire da codice C

C

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Assembly

```
_sum:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%eax
        addl 8(%ebp),%eax
        movl %ebp,%esp
        popl %ebp
        ret
```

(gas AT&T)

Comando :

```
gcc -O -S code.c
```

Produce un file di nome `code.s`

Codice funzione `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- **13 bytes**
- **Ogni istruzione 1, 2, o 3 bytes**
- **Indirizzo iniziale `0x401040`**

Object code

Assembler

- Traduce `.s` in `.o`
- Traduce ogni istruzione nella sua controparte in codifica binaria
- Risultato estremamente “vicino” al codice eseguito dall amacchina
- Mancano I collegamenti tra il codice sorgente contenuto in file diversi.

Linker

- Risolve i riferimenti tra il codice contenuto in diversi sorgenti
- Combina il risultato ottenuto con codice di librerie standard
  - es. codice `malloc`, `printf`
- Alcune librerie sono linkate in modo dinamico
  - Il link è effettuato al momento dell'inizio dell'esecuzione del programma

Disassemblare codice oggetto

```
00401040 <_sum>:
   0:  55                          push   %ebp
   1:  89 e5                       mov    %esp,%ebp
   3:  8b 45 0c                    mov    0xc(%ebp),%eax
   6:  03 45 08                    add    0x8(%ebp),%eax
   9:  89 ec                       mov    %ebp,%esp
   b:  5d                          pop    %ebp
   c:  c3                          ret
   d:  8d 76 00                    lea    0x0(%esi),%esi
```

objdump -d p

- Strumento utile per analisi codice oggetto
- Analisi pattern di bit corrispondenti a serie istruzioni
- Corrispondenza **approssimata** con codice assembly
- Applicabile `a a.out` (eseguibile completo) o file .o

Disassemblare codice oggetto

```
00401040 <_sum>:
   0:  55                      push   %ebp
   1:  89 e5                   mov    %esp,%ebp
   3:  8b 45 0c                mov    0xc(%ebp),%eax
   6:  03 45 08                add    0x8(%ebp),%eax
   9:  89 ec                   mov    %ebp,%esp
   b:  5d                      pop    %ebp
   c:  c3                      ret
   d:  8d 76 00                lea    0x0(%esi),%esi
```

objdump -d p

- Strumento utile per analisi codice oggetto
- Analisi pattern di bit corrispondenti a serie istruzioni
- Corrispondenza approssimata con codice assembly
- Applicabile `a a.out` (eseguibile completo) o file .o

Object

```
0x401040:
0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x89
0xec
0x5d
0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

Mediante debugger gdb

`gdb p  (o file .o)`

`disassemble sum`

- Disassembla funzione

`x/13b sum`

- Esamina 13 byte dall'inizio di `sum`

Cosa possiamo disassemblare?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:     55                      push   %ebp
30001001:     8b ec                   mov    %esp,%ebp
30001003:     6a ff                   push   $0xffffffff
30001005:     68 90 10 00 30          push   $0x30001090
3000100a:     68 91 dc 4c 30          push   $0x304cdc91
```

- Tutto ciò che può essere interpretato come codice eseguibile
- Il disassemblatore interpreta i byte e ricostruisce il sorgente assembly

`muovere dati`

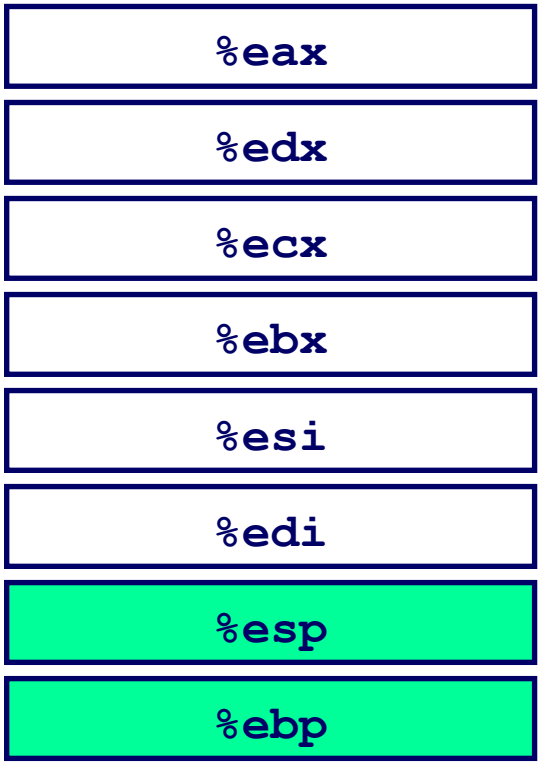
Muovere (copiare) dati

`movl` *Source*,*Dest*:

- Muovi 4-byte (“long”) word
- Utilizzo frequente

Tipi di operando

- **Immediato**: costante intera
  - Come costante C, ma ha prefisso ‘`$`’
  - es. `$0x400`, `$-533`
  - 1, 2, o 4 byte
- **Registri**: eax,edx,ecx,ebx,esi,edi,esp,ebp (o parte di essi)
  - `%esp` e `%ebp` riservati per usi speciali (poi vediamo)
  - Altri hanno usi particolari solo per certe istruzioni
- **Memoria**: 4 byte consecutivi
  - Varie modalità di indirizzamento



`movl`: combinazioni operandi ammesse

	Source	Destination		corrispondente C
`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;`

- **Non** ammette trasferimento da memoria a memoria in una singola istruzione

Esistono diverse modalità di indirizzamento

(R) Mem[Reg[R]]

- Il registro R contiene un indirizzo di memoria

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

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

- Registro R punta a una posizione in memoria
- Costante D specifica offset

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

funzione swap

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

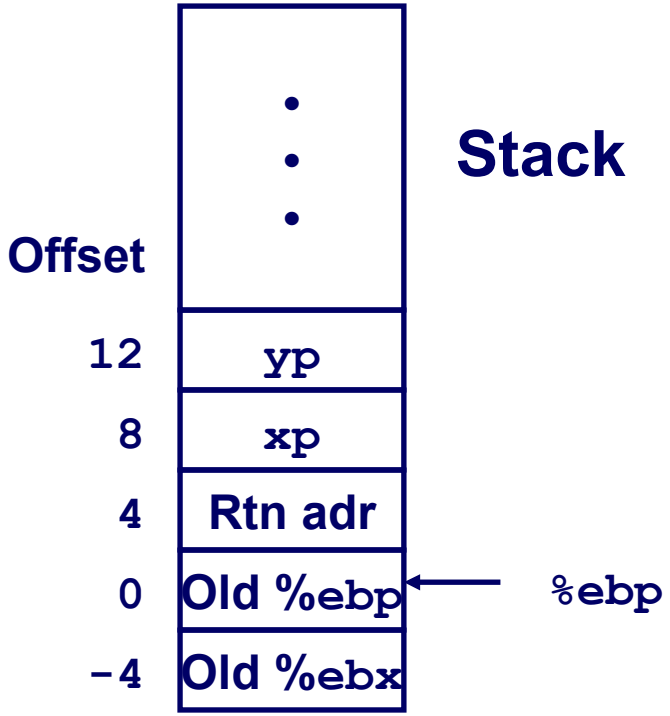
```
swap:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx

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

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

Set Up

Body

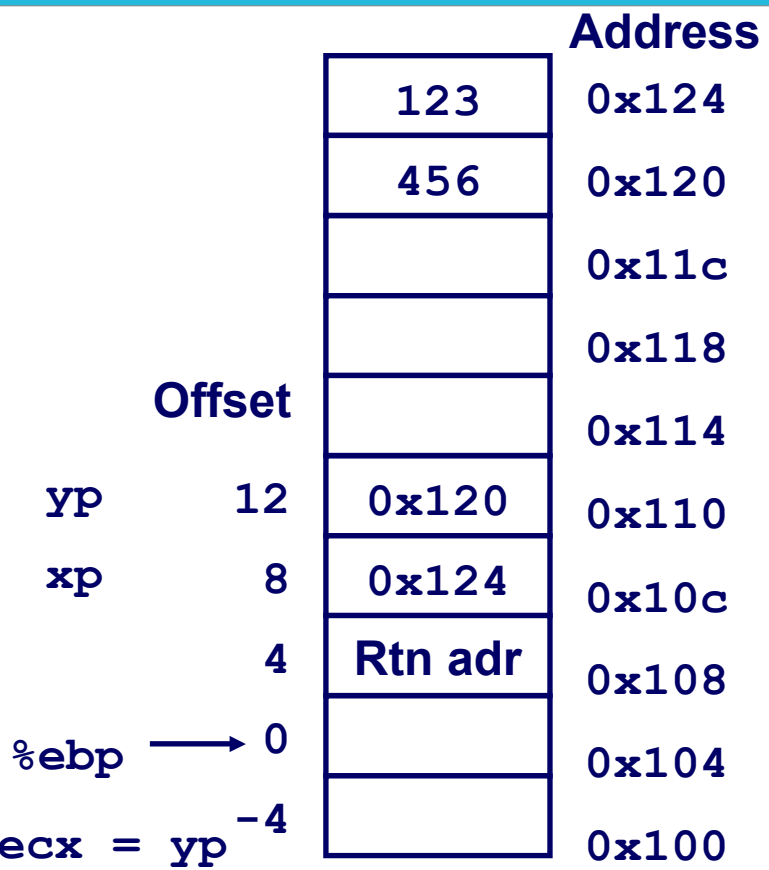
Finish

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



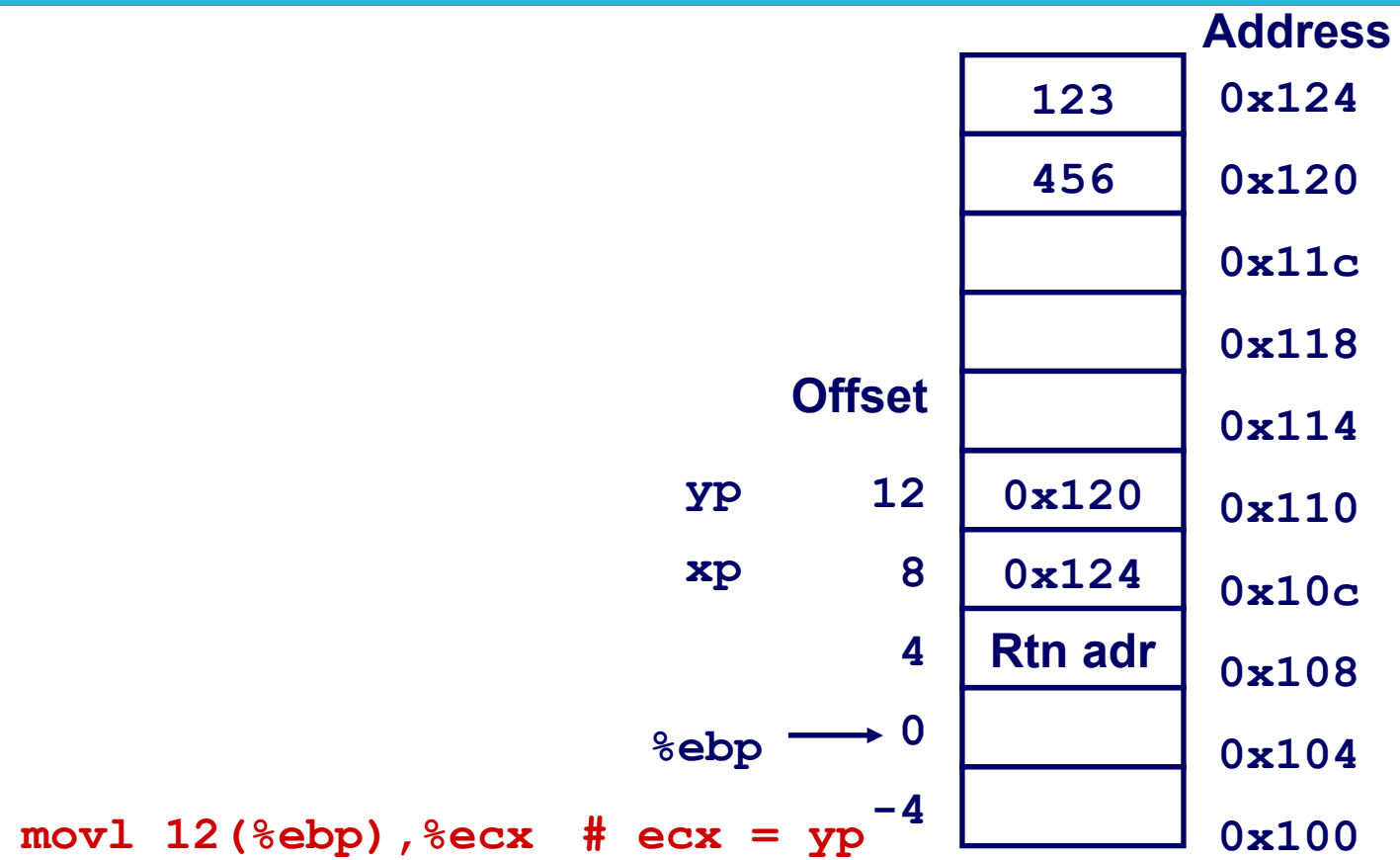
Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
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
```



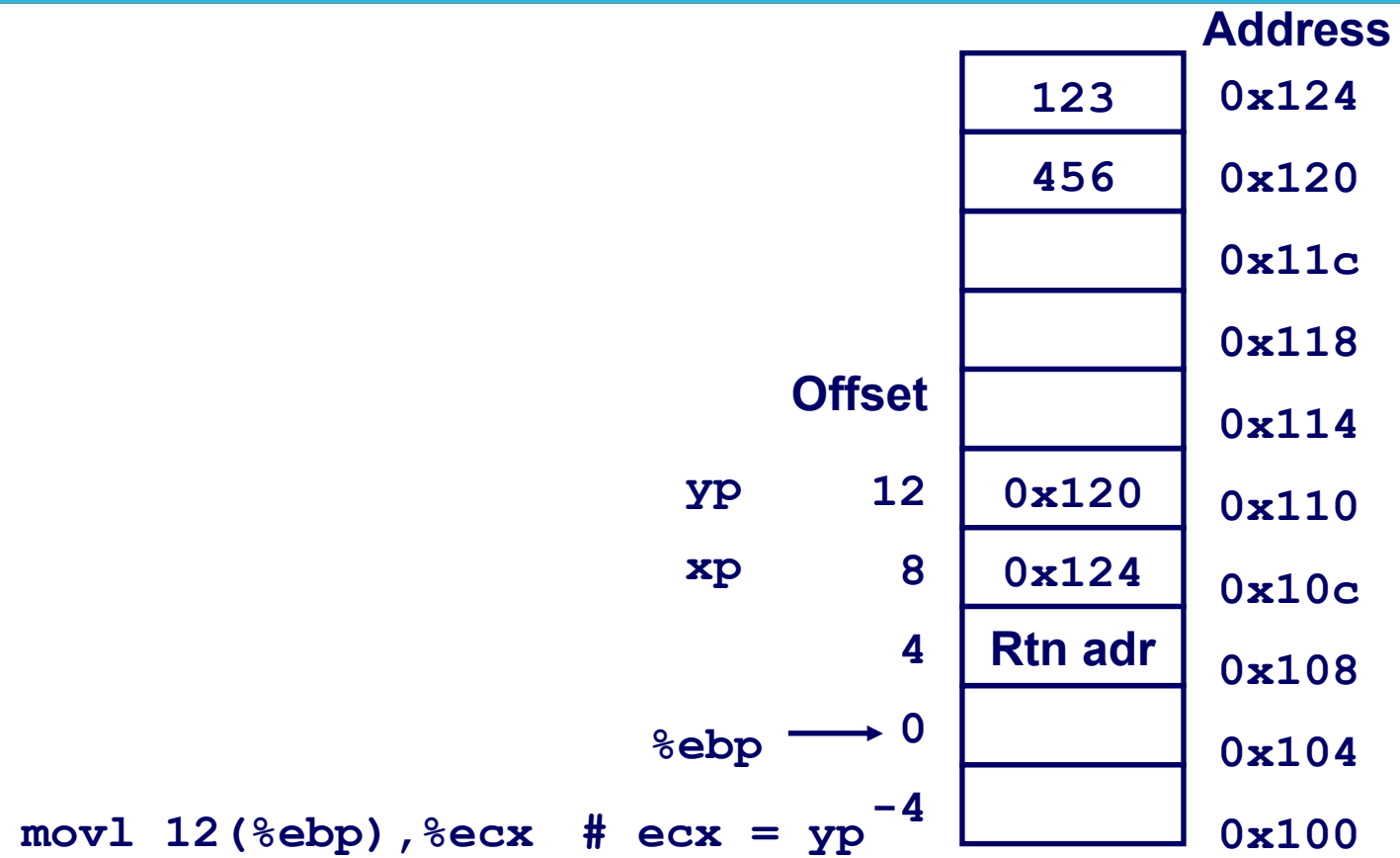

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

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



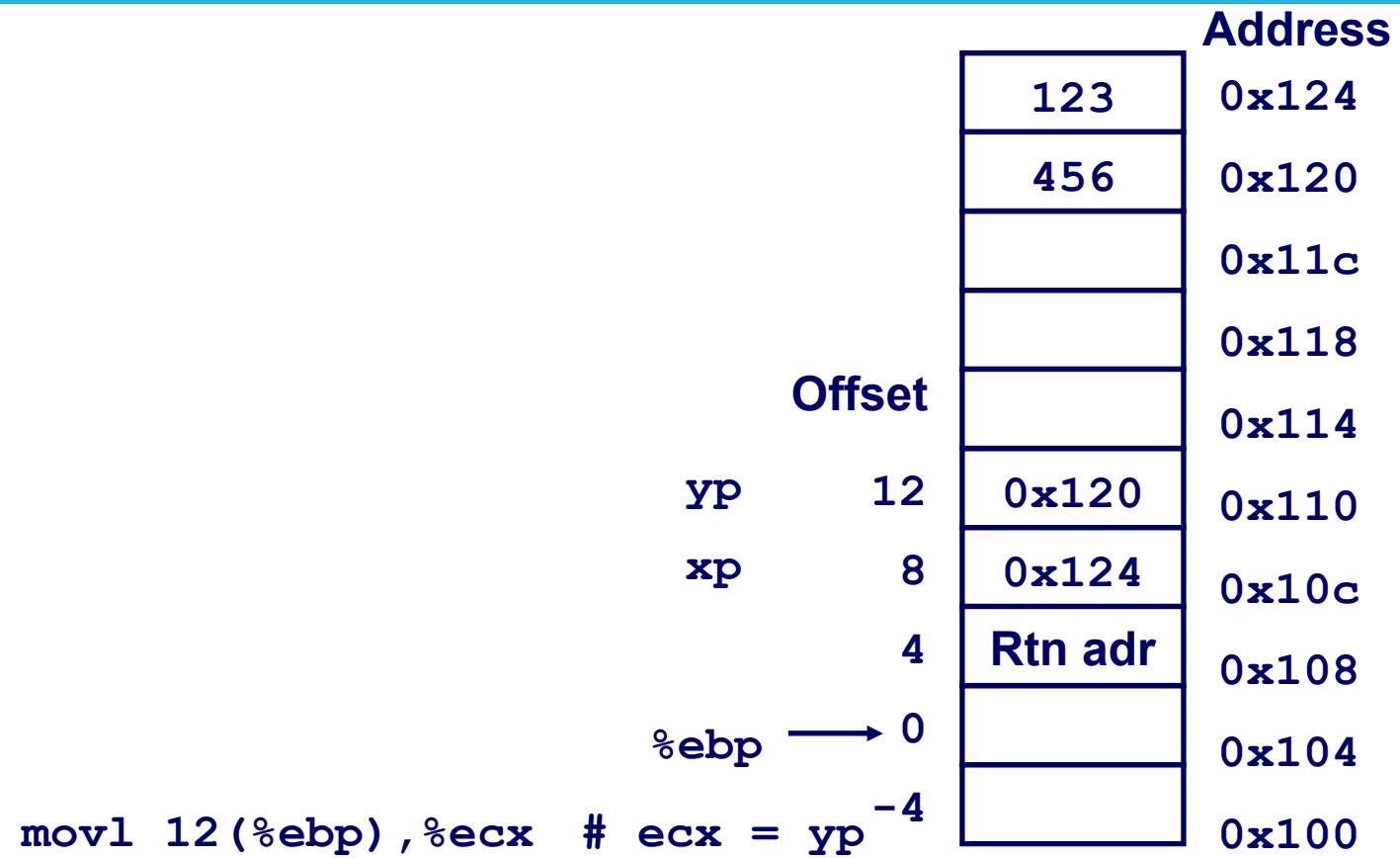

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

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



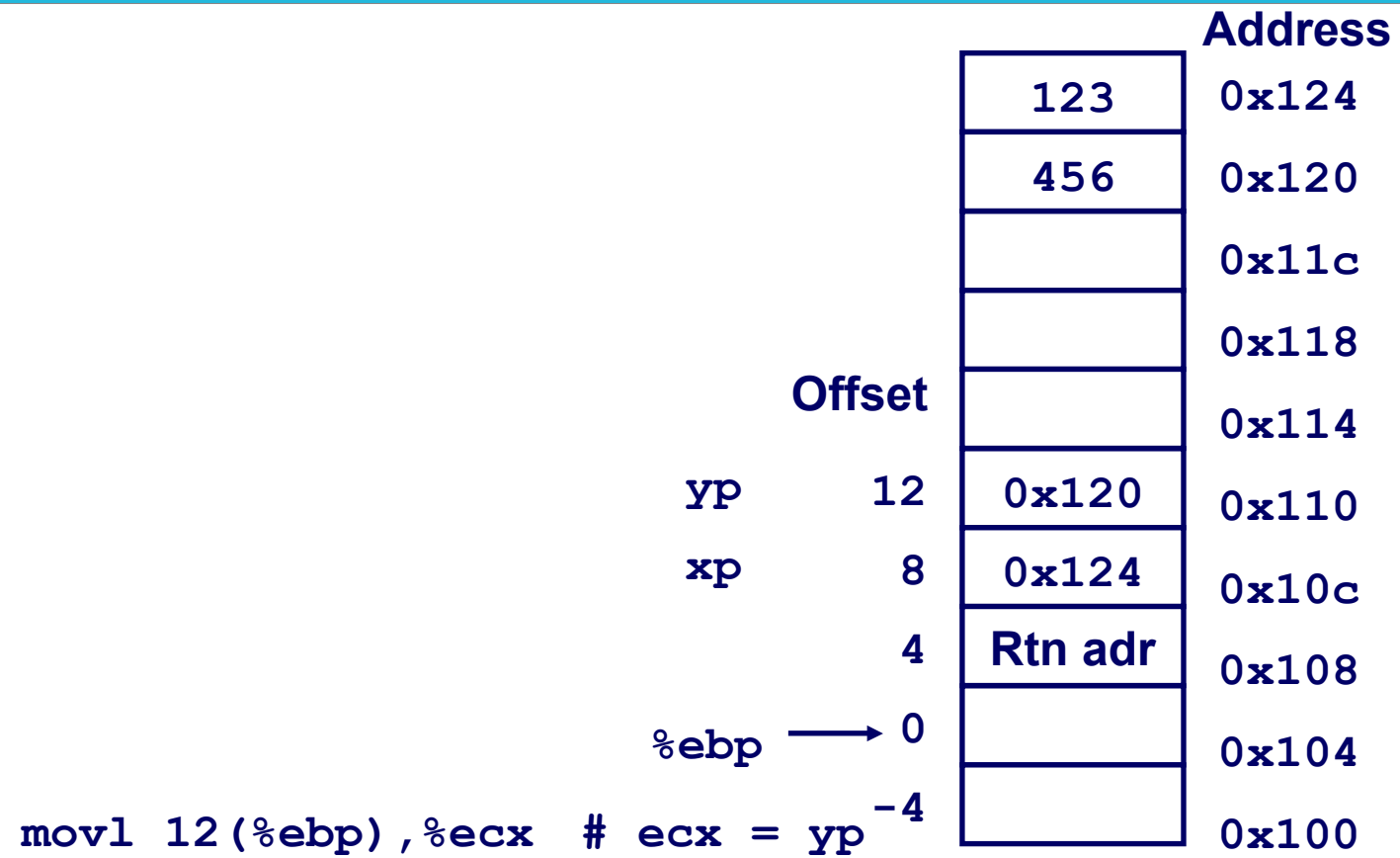

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

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



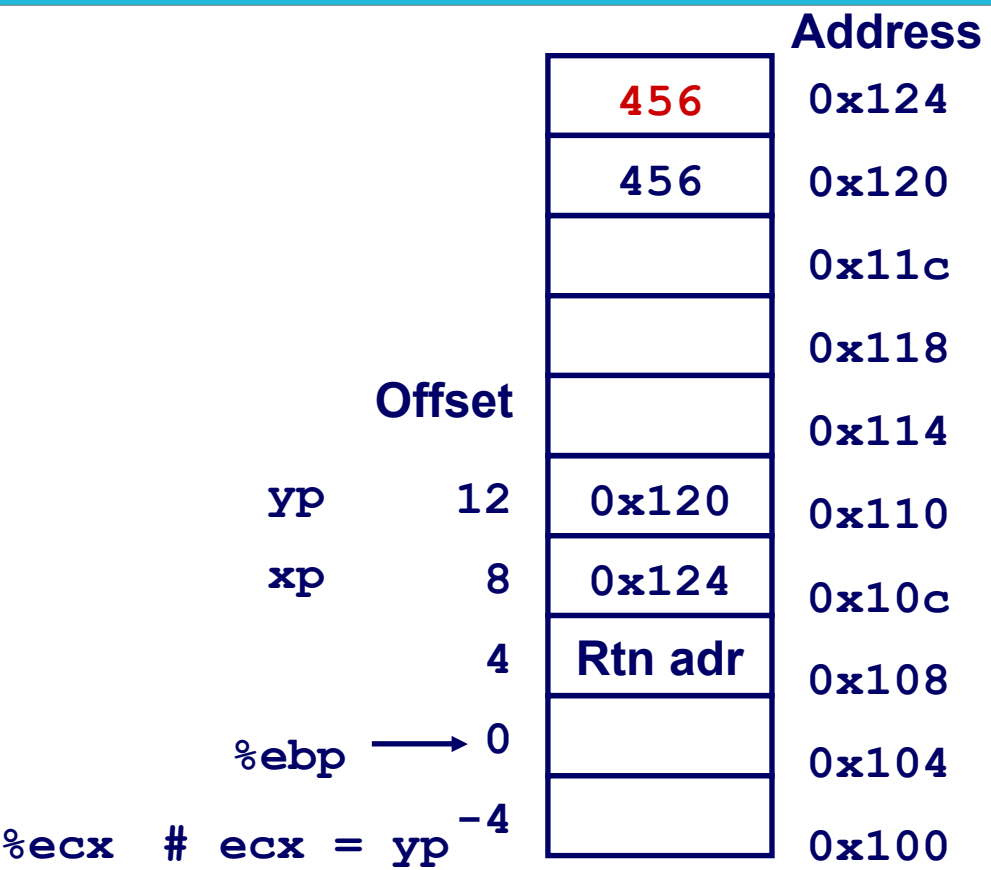

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

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



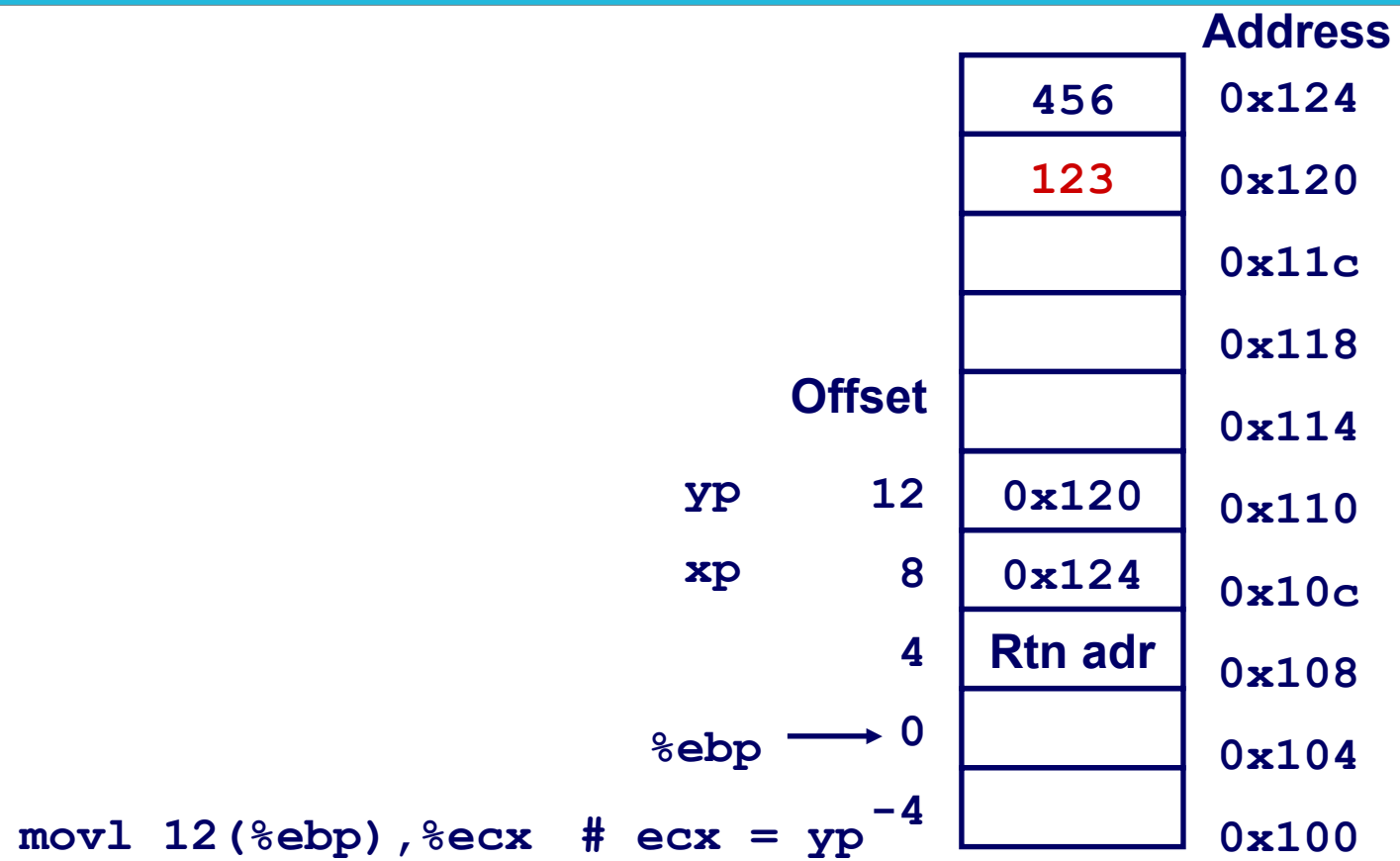

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

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




```
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
```

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




```
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
```

Un ambiente per operare efficacemente :

l'emulatore **Qemu**

Qemu http://fabrice.bellard.free.fr/qemu PC (x86 or x86_64 processor)

- i440FX host PCI bridge and PIIX3 PCI to ISA bridge
- Cirrus CLGD 5446 PCI VGA card
- PS/2 mouse and keyboard
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- NE2000 PCI network adapters
- Serial ports
- PCI UHCI USB controller and a virtual USB hub.

Qemu è in grado di simulare l'hardware IA-32 (e altri) sin nei minimi dettagli

Rende semplice fare esperimenti in cui cerchiamo di eseguire dei programmi *senza l'intervento* di un sistema operativo.

E' possibile utilizzare QEMU in combinazione con un *debugger* ossia un programma che permette di fermare l'esecuzione del programma in punti critici mediante l'impostazione di breakpoint ed esaminare con calma lo stato della memoria.

Qemu è predisposto per interagire con il debugger **GDB** che occorre mettere in comunicazione con Qemu tramite una opportuna connessione di rete.

Sequenza di boot

Cosa succede quando si accende un PC?

1. Inizia l'esecuzione del programma contenuto nel firmware (BIOS)
2. Il BIOS carica il programma contenuto nel boot sector
3. Il programma di boot carica il sistema operativo
4. A questo punto il controllo della macchina è affidato al s.o., a cui dovranno essere richiesti i caricamenti di altri programmi

Sequenza di boot i386 : analisi via Qemu/GDB (Linux)

qemu-system-i386 -S -gdb tcp::42000

eseguibile emulatore:
Il nome può cambiare
(ne esistono molte
versioni)

Stop:
Lancia Qemu e lo
blocca in attesa di
comandi (es. esegui la
prossima istruzione)

Si aggancia al
debugger GDB
rimanendo in attesa
di istruzioni da esso
sulla porta tcp 42000

Sequenza di boot i386 : analisi via Qemu/GDB (Linux)

```
matteo@matteo-UX32VDA:~$ gdb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb) target remote localhost:42000
Remote debugging using localhost:42000
0x0000fff0 in ?? ()
(gdb)
```

Comando per eseguire GDB

Connettiamo GDB all'emulatore Qemu (conferma connessione)

prompt GDB

Esaminare il contenuto della memoria

```
(gdb) x/x 0x0
0x0:    0x00000000
(gdb) x/2x 0x0
0x0:    0x00000000    0x00000000
(gdb) x/i 0x0
0x0:    add    %al,(%eax)
```

x/nfu addr

n f u sono parametri opzionali.

n : repeat count. Quanta memoria mostrare (in unità u)

f : display format. s (stringa terminata da null), i (istruzione codice macchina), x (esadecimale)

u : unit size , uno dei seguenti valori

b → bytes
h → half word (2 bytes)
w → word (4 bytes)
g → giant word (8 bytes)

Il comando examine (x) mostra che anche 0x00000000 è un'istruzione valida in codice macchina ...

Esaminare il contenuto dei registri

```
(gdb) info registers
eax            0x0    0
ecx            0x0    0
edx            0x633  1587
ebx            0x0    0
esp            0x0    0x0
ebp            0x0    0x0
esi            0x0    0
edi            0x0    0
eip            0xfff0 0xfff0
eflags         0x2    [ ]
cs             0xf000 61440
ss             0x0    0
ds             0x0    0
es             0x0    0
fs             0x0    0
gs             0x0    0
(gdb)
```

Comando per esaminare il contenuto dei registri

```
(gdb) info registers
eax            0x0    0
ecx            0x0    0
edx            0x633  1587
ebx            0x0    0
esp            0x0    0x0
ebp            0x0    0x0
esi            0x0    0
edi            0x0    0
eip            0xfff0 0xfff0
eflags         0x2    [ ]
cs             0xf000 61440
ss             0x0    0
ds             0x0    0
es             0x0    0
fs             0x0    0
gs             0x0    0
(gdb)
```

Quale comando sta per essere eseguito dalla macchina?

Per motivi di compatibilità il processore, al momento dell'accensione, è in **modalità REALE** (simile a quella con cui funzionavano I processori Intel 8086).

Gli indirizzi a 20 bit in modalità reale sono ottenuti sempre mediante somma di un indirizzo di base ***b*** ed uno spiazzamento ***o*** :

$i = \mathbf{b} \cdot 16 + \mathbf{o}$

Base in registro **cs**

Offset in registro **eip**

```
(gdb) info registers
eax            0x0    0
ecx            0x0    0
edx            0x633  1587
ebx            0x0    0
esp            0x0    0x0
ebp            0x0    0x0
esi            0x0    0
edi            0x0    0
eip            0xfff0 0xfff0
eflags         0x2    [ ]
cs             0xf000 61440
ss             0x0    0
ds             0x0    0
es             0x0    0
fs             0x0    0
gs             0x0    0
(gdb)
```

Quale comando sta per essere eseguito dalla macchina?

i = **b** · 16 + **o**

In esadecimale moltiplicare per 16 equivale ad aggiungere uno 0 …

i = 0xf000 · 16 + 0xfff0
i = 0xf000**0** + 0xfff0
i = 0xffff0
(si può scrivere anche 0xf0000:0xfff0)

Il costrutture ha scelto questo indirizzo come indirizzo della prima istruzione che verrà eseguita. Questa scelta è cablata nello hardware una volta per tutte.

Quale comando sta per essere eseguito dalla macchina?

La prima istruzione che verrà eseguita dalla macchina è all'inidirizzo **0xffff0** .
Verifichiamo che ci sia effettivamente qualcosa ...

```
(gdb) set architecture i8086
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
(gdb) x/i 0xffff0
   0xffff0:     ljmp   $0xf000,$0xe05b
```

Salta ad un'altra posizione (più bassa) in memoria. Perchè?

$2^{20} = 1{,}048{,}576$

$(ffff0)_h = (1{,}048{,}560)_{10}$

Quale comando sta per essere eseguito dalla macchina?

```
(gdb) x/i 0xffff0
   0xffff0:     ljmp   $0xf000,$0xe05b
```

In effetti c'è qualcosa: un programma precaricato dal costruttore che è in grado di caricare **un altro programma** dalla memoria di massa.

E' un sistema operativo minimale detto **BIOS** (Basic Input/Output System). Serve a **caricare** un sistema operativo più completo (e scelto dall'utente).

Obiettivo del prossimo esperimento :

Sfruttare il BIOS per caricare un programma nostro invece del sistema operativo.

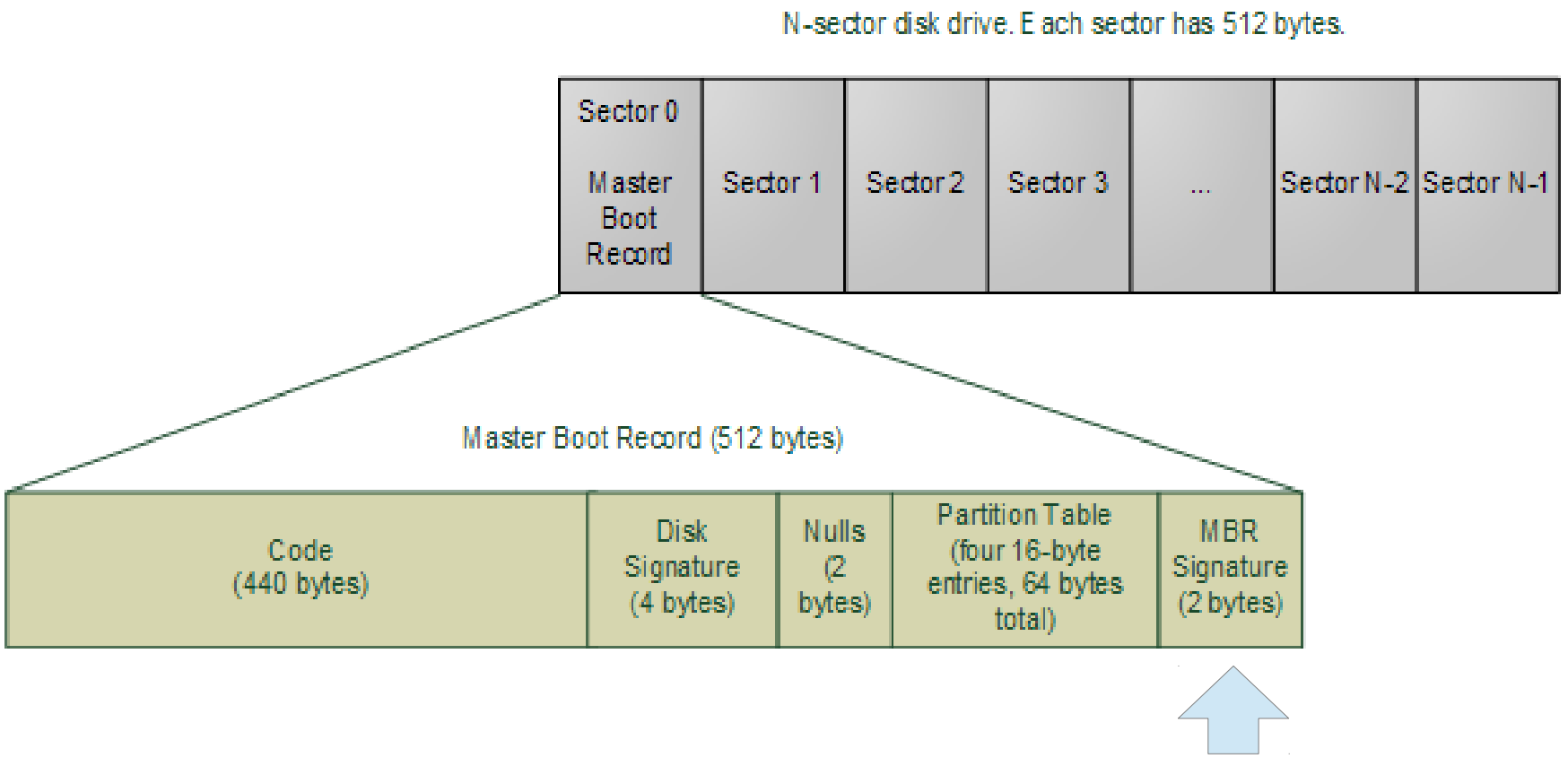
All'interno di GDB utilizzare il comando **c** (continua l'esecuzione). Otterremo vari messaggi d'errore … il che ha senso dato che il BIOS non è riuscito a caricare nulla dalla memoria di massa.

Sequenza di boot

Cosa succede quando si accende un PC?

1. Inizia l'esecuzione del programma contenuto nel firmware (BIOS)
2. Il BIOS carica il programma contenuto nel boot sector
3. Il programma di boot carica il sistema operativo
4. A questo punto il controllo della macchina è affidato al s.o., a cui dovranno essere richiesti i caricamenti di altri programmi

… a proposito del boot sector



Attività nota in forma semi algoritmica sin dal V secolo a.C. è l'algoritmo di Euclide per il calcolo del massimo comun divisore MCD tra due interi positivi. Vgliamo calcolare MCD fra i numeri 420 e 240. Useremo la notazione NASM (intel)

```
def mcd(x,y)
    assert( x>0 and y>0 )
    if x==y: return x
    elif x > y: return mcd(x-y, y)
    else: return mcd(x,y-x)
```

descrizione ad alto livello

```
main:   mov dx, 420      ; dx = 420
        mov bx, 240      ; bx = 240
max:    cmp dx, bx       ;ZF = (dx==bx);CF = (dx < bx)
        je fine          ; if(ZF) goto fine
        jg diff          ; if!(ZF || CF) goto diff
        mov ax, dx       ; ax = dx
        mov dx, bx       ; dx = bx
        mov bx, ax       ; bx = ax
diff:   sub dx, bx       ; dx -= bx
        jmp max          ; goto max
fine:   hlt              ; halt/*dx == mcd(420,240)*/
```

MCD(420,240) in assembly

Perchè il programma possa essere caricato dal BIOS **dobbiamo seguire alcune regole**. Qualora lo si carichi da un disco magnetico (per CD valgono altre regole) il programma :

- **deve** essere conservato nel primo settore del disco
- il primo settore **deve** contenere al massimo **512** byte
- il primo settore **deve** terminare (byte 511 e 512) con i byte **AA55**

NB: L'intera sequenza verrà caricata a partire dall'indirizzo **0x0000:0x7c00**

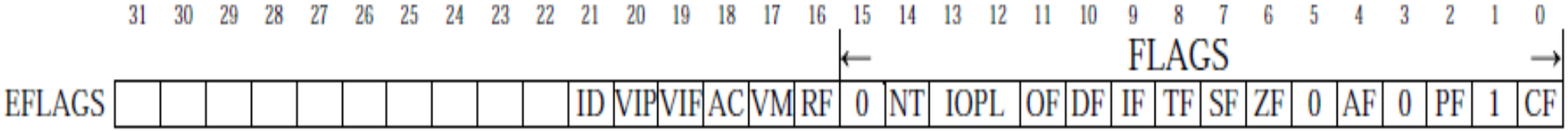
… dobbiamo apportare alcune modifiche al programma mostrato nella slide precedente per poter completare l'esperimento.


```
segment .text
global main

main:   mov dx, 420      ; dx = 420
        mov bx, 240      ; bx = 240
max:    cmp dx, bx       ;ZF = (dx==bx);CF = (dx < bx)
        je fine          ; if(ZF) goto fine
        jg diff          ; if!(ZF || CF) goto diff
        mov ax, dx       ; ax = dx
        mov dx, bx       ; dx = bx
        mov bx, ax       ; bx = ax
diff:   sub dx, bx       ; dx -= bx
        jmp max          ; goto max
fine:   hlt              ; halt/*dx == mcd(420,240)*/

times 510-($-$$)db 0
dw 0xAA55
```

mcd1.asm (versione modificata)

MACCHINA i386 : EFLAGS

Flags



Bit		Flag	Description
0	CF	Carry Flag	Carry from most significant bit, also borrow for most significant bit; can be considered as overflow in unsigned instructions.
2	PF	Parity Flag	Set to 1 if 8 less significant bits of result have even number of 1's, else set to 0.
4	AF	Auxiliary carry Flag	Used as carry flag in BCD instructions.
6	ZF	Zero Flag	Set to 1 if result is zero, else set to 0.
7	SF	Sign Flag	Set to 1 if result is negative (below zero), else set to 0.
8	TF	Trap Flag	Used by debuggers.
9	IF	Interrupt Flag	If set to 1, then interrupts are enabled, else are disabled.
10	DF	Direction Flag	When set to 0, string instructions increment the index registers, else – decrement the index registers.
11	OF	Overflow Flag	Used in signed instructions.
12,13	IOPL	I/O Privilege Level	Indicates the I/O privilege level of the currently running program or task.
14	NT	Nested Task	Controls the chaining of interrupt and called tasks.
16	RF	Resume Flag	Controls the processor's response to instruction-breakpoint conditions.
17	VM	Virtual 8086 Mode	Set to enable virtual-8086 mode; clear to return to protected mode.
18	AC	Alignment check	Set this flag and the AM flag in control register CR0 to enable alignment checking of memory references.
19	VIF	Virtual Interrupt Flag	Contains a virtual image of the IF flag.
20	VIP	Virtual Interrupt Pending	Set by software to indicate that an interrupt is pending.
21	ID	Identification	The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

IA-32 instruction set
(trasferimento controllo II)

Jump if condition	JA	rel8	Jumps near, relative, if above (CF=0 and ZF=0)
	JAE		Jumps near, relative, if above or equal (CF=0)
	JB		Jumps near, relative, if below (CF=1)
	JBE		Jumps near, relative, if below or equal (CF=1 or ZF=1)
	JC		Jumps near, relative, if carry (CF=1)
	JE		Jumps near, relative, if equal (ZF=1)
	JG		Jumps near, relative, if greater (ZF=0 and SF=OF)
	JGE		Jumps near, relative, if greater or equal (SF=OF)
	JL		Jumps near, relative, if less (SF<>OF)
	JLE		Jumps near, relative, if less or equal (ZF=1 or SF<>OF)
	JNA		Jumps near, relative, if not above (CF=1 or ZF=1)
	JNAE		Jumps near, relative, if not above or equal (CF=1)
	JNB		Jumps near, relative, if not below (CF=0)
	JNBE		Jumps near, relative, if not below or equal (CF=0 and ZF=0)
	JNC		Jumps near, relative, if not carry (CF=0)
	JNE		Jumps near, relative, if not equal (ZF=0)
	JNG		Jumps near, relative, if not greater (ZF=1 or SF<>OF)
	JNGE		Jumps near, relative, if not greater or equal (SF<>OF)
	JNL		Jumps near, relative, if not less (SF=OF)
	JNLE		Jumps near, relative, if not less or equal (ZF=0 and SF=OF)
	JNO		Jumps near, relative, if not overflow (OF=0)
	JNP		Jumps near, relative, if not parity (PF=0)
	JNS		Jumps near, relative, if not sign (SF=0)
	JNZ		Jumps near, relative, if not zero (ZF=0)
	JO		Jumps near, relative, if overflow (OF=1)
	JP		Jumps near, relative, if parity (PF=1)
	JPE		Jumps near, relative, if parity even (PF=1)
	JPO		Jumps near, relative, if parity off (PF=0)
	JS		Jumps near, relative, if sign (SF=1)
	JZ		Jumps near, relative, if zero (ZF=1)

IF condition THEN
(E)IP ← (E)IP + DST

Assembliamo il tutto:

linuxprompt$ nasm **-f** bin **-o** mcdboot.bin mcd1.asm

Il programma va assemblato in un formato contenente puro codice macchina (-f bin)

Ed eseguiamo qemu obbligandolo ad utilizzare mcdboot.bin per emulare il primo hard disk (opzione -hda)

linuxprompt$ qemu-system-i386 -S **-hda** mcdboot.bin -gdb tcp::42000

Connettiamo il debugger e indaghiamo ...

```
(gdb) target remote localhost:42000
Remote debugging using localhost:42000
0x0000fff0 in ?? ()
(gdb) x/x 0x7c00
0x7c00:  0x00000000
(gdb) b* 0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) set architecture i8086
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
(gdb)
```

La memoria all'indirizzo 0x7c00 è vuota ...

Impostiamo un breakpoint

Continuiamo l'esecuzione

Riesaminiamo la memoria in 0x7c00

```
(gdb) x/11i 0x7c00
=> 0x7c00:    mov    $0x1a4,%dx
   0x7c03:    mov    $0xf0,%bx
   0x7c06:    cmp    %bx,%dx
   0x7c08:    je     0x7c16
   0x7c0a:    jg     0x7c12
   0x7c0c:    mov    %dx,%ax
   0x7c0e:    mov    %bx,%dx
   0x7c10:    mov    %ax,%bx
   0x7c12:    sub    %bx,%dx
   0x7c14:    jmp    0x7c06
   0x7c16:    hlt
(gdb) si
0x00007c03 in ?? ()
(gdb) p $dx
$1 = 420
(gdb)
```

Ora a questo indirizzo è presente il nostro programma

Eseguiamo una singola istruzione ...

E' stata eseguita l'istruzione
mov $0x1a4, %dx
Quindi in dx dovrebbe esserci il valore 420

in effetti è così

La scheda video

Problema:

L'unico modo di conoscere il risultato calcolato è quello di esaminare il contenuto del registro **dx** quando l'esecuzione raggiunge l'ultima istruzione. Non molto comodo...

Sarebbe meglio rendere il dato disponibile mediante l'utilizzo di una periferica adatta … ad esempio lo schermo.

Per ottenere questo risultato occorre utilizzare la scheda video che controlla lo schermo, **scrivendo i dati da visualizzare nella memoria della scheda**.

La scheda video

Se ci interessa usare la scheda video in modalità testuale essa può generalmente essere pilotata secondo le convenzioni introdotte con l' IBM *monochrome display adapter* (**MDA**) montata sul PC originale nel 1981.

Lo schermo viene utilizzato come una griglia rettangolare di caratteri

MDA era dotata di **4Kb** di memoria accessibili tramite indirizzi della memoria fisica del PC. Questa tecnica si chiama ***memory mapped I/O*** e permette di accedere alle risorse dello hardware periferico tramite le solite operazioni di manipolazione di memoria del sistema.

Indirizzi di memoria → **mapping** →parole di memoria gestite dalla periferica

La scheda video

General **x86 Real Mode Memory Map**:

0x00000000 - 0x000003FF - Real Mode Interrupt Vector Table
0x00000400 - 0x000004FF - BIOS Data Area
0x00000500 - 0x00007BFF - Unused
0x00007C00 - 0x00007DFF - Bootloader
0x00007E00 - 0x0009FFFF - Unused
0x000A0000 - 0x000BFFFF - Video RAM (VRAM) Memory – graphic modes
0x000B0000 - 0x000B7777 - Monochrome Video Memory – txt mode
0x000B8000 - 0x000BFFFF - Color Video Memory – txt mode (color)
0x000C0000 - 0x000C7FFF - Video ROM BIOS
0x000C8000 - 0x000EFFFF - BIOS Shadow Area
0x000F0000 - 0x000FFFFF - System BIOS

25 righe di 80 caratteri

Nel caso di MDA la convenzione è quella di considerare gli indirizzi a partire da **0xb8000** come corrispondenti a un array di 80 * 25 = 2000 **coppie** di byte

La scheda video

0x000A0000 - 0x000BFFFF - Video RAM (VRAM) Memory
0x000B0000 - 0x000B7777 - Monochrome Video Memory
0x000B8000 - 0x000BFFFF - Color Video Memory

Scrivere in 0x000B8000 stampa un carattere sullo schermo. Ad esempio :

```
%define   VIDMEM 0xB8000       ; video memory

mov edi, VIDMEM      ; get pointer to video memory
mov [edi], 'A'       ; print character 'A'
mov [edi+1], 0x7     ; character attribute
```

DMA e memory mapped I/O

Come possiamo scrivere un carattere in una posizione a nostra scelta nella griglia 25 x 80 ?

Una porprietà utile della memoria che stiamo manipolando in questo esempio è che è lineare. Se raggiungiamo la fine di una riga di testo il prossimo carattere verrà stampato sulla riga successiva. Sfruttando la linearità della memoria la formula per scrivere ad una data posizione x(riga)/y(colonna) sullo schermo è :

x + y * colonne_schermo

Ricordiamo, inoltre che, per scrivere 1 carattere servono 2 byte, il primo specifica il carattere (standard ASCII) il secondo il suo formato sullo schermo.

Accesso alla scheda video mappata in memoria

```
segment .text
global main

QUANTI equ 100              ; #define QUANTI 100
N equ QUANTI*2-2            ; #define N (QUANTI*2-2)
main:   mov ax, 0xb800      ; /* mov ds, x è vietato */
        mov ds, ax          ; ds = ax
        mov cx, QUANTI      ; /* cx è indice per loop */
        mov bx, N           ; bx = N
ciclo:                      ; do {
        mov byte[ds:bx], 'm' ; mem[ds:bx]='m'
        sub bx, 2           ; bx -= 2
        loop ciclo          ; } while(cx!=0)
fine:   hlt                 ; halt/*dx == mcd(420,240)*/

times 510-($-$$)db 0
dw 0xAA55
```

mdamio.asm

Accesso alla scheda video mappata in memoria

linuxprompt$ nasm **-f** bin mdamioboot.bin mdamio.asm

linuxprompt$ qemu-system-i386 **-hda** mdamioboot.bin -gdb tcp::42000

… poi connettiamo GDB e procediamo come prima. Il programma stamperà dei caratteri sullo schermo sfruttando il memory mapped I/O.

NB: notate che l'indice del ciclo **cx** è automaticamente decrementato dall'istruzione **loop**.

L'indirizzo effettivo a cui si fa riferimento nella prima **mov** del ciclo è calcolato secondo le regole della modalità reale:
$0xb800 \cdot 16 + (100\text{-}2)_{10} = 0xb8000 + 62_h =$ **0xb8062** . Dopo si procede a ritroso saltando il byte dei parametri di visualizzazione.

Supporto BIOS per la gestione del video

Il sofware di base precaricato potrebbe fornire una libreria di funzioni utili per semplificare la scrittura di programmi che interagiscano con le periferiche.

In effetti è così:
Nei PC IBM compatibili la zona di memoria da **0xA0000** a **0x100000** è in sola lettura (*Read-Only Memory,* ROM) e in essa sono presenti una serie di routine per la gestione dello hardware. Si tratta del *firmware*, software fornito dal produttore e cablato nella memoria della macchina, come il BIOS.

Come possiamo accedere a queste routine?

Ipotesi 1: potremmo usare una chiamata a procedura (**call**). Problema: in varie versioni le medesime routine potrebbero trovarsi ad indirizzi diversi.

Ipotesi 2: meccanismo di ***chiamata implicita***. Sfrutta meccanismo hardware di protezione. E' uno dei meccanismi findamentali di dialogo tra applicazioni e s.o.

Supporto BIOS per la gestione del video

Chiamata implicita:

Sfrutta il meccanismo delle interruzioni hardware (in questo caso è lanciata via software e quindi ri parla di *software interrupt*, invece che di *interrupt request* (IRQ) come nel caso di richieste di interruzioni derivanti da periferiche).

L'effetto è il seguente: il processore salva parte del proprio stato e salta all'indirizzo dell'apposito gestore di interruzione che si può impostare programmando opportunaente PIC e memoria. Al termine dell'esecuzione della procedura che gestisce l'interruzione il processore ripristina il proprio stato al momento antecedente l'arrivo dell'interruzione e procede.

Il produttore del firmware dovrà garantire l'inizializzazione di una tabella di gestori delle interruzioni contenente i dati necessari a raggiungere una routine e pubblicare l'associazione tra una data interruzione e una data periferica.

Supporto BIOS per la gestione del video

Nei BIOS dei PC IBM compatibili l'interruzione 16 (0x10) viene utilizzata per controllare la scheda video e stampare. Il protocollo da seguire è questo:

I) Mettere nel byte basso di AX (AL) il carattere da stampare

II) Mettere nel byte alto di AX (AH) un valore che indica la modalità di visualizzazione (es. 0X0e → modalità testuale standard)

III) Utilizzare il registro BX per impostare gli attrubuti estetici (luminosità, colore ecc.)

IV) lanciare l'interruzione 0x10 (BIOS, scrivi su schermo e sposta cursore)

Sistemi Operativi

Laboratorio – linea 2

1

```
segment .text
global main

main:
; set it only once.
MOV    AH, 0Eh     ; select sub-function.

MOV    AL, 'H'     ; ASCII code: 72
INT    10h         ; print it!
MOV    AL, 'e'     ; ASCII code: 101
INT    10h         ; print it!
MOV    AL, 'l'     ; ASCII code: 108
INT    10h         ; print it!
MOV    AL, 'l'     ; ASCII code: 108
INT    10h         ; print it!
MOV    AL, 'o'     ; ASCII code: 111
INT    10h         ; print it!
MOV    AL, '!'     ; ASCII code: 33
INT    10h         ; print it!
fine:   hlt        ; halt

times 510-($-$$)db 0
dw 0xAA55
```

biosintimp.asm

Supporto BIOS per la gestione del video

linuxprompt$ nasm **-f** bin biosintimpboot.bin biosintimp.asm

linuxprompt$ qemu-system-i386 **-hda** biosintimpboot.bin -gdb tcp::42000

… poi connettiamo GDB e procediamo come prima. Il programma stamperà dei caratteri sullo schermo sfruttando le routine del firmware.

Cosa succede quando si accende un PC?