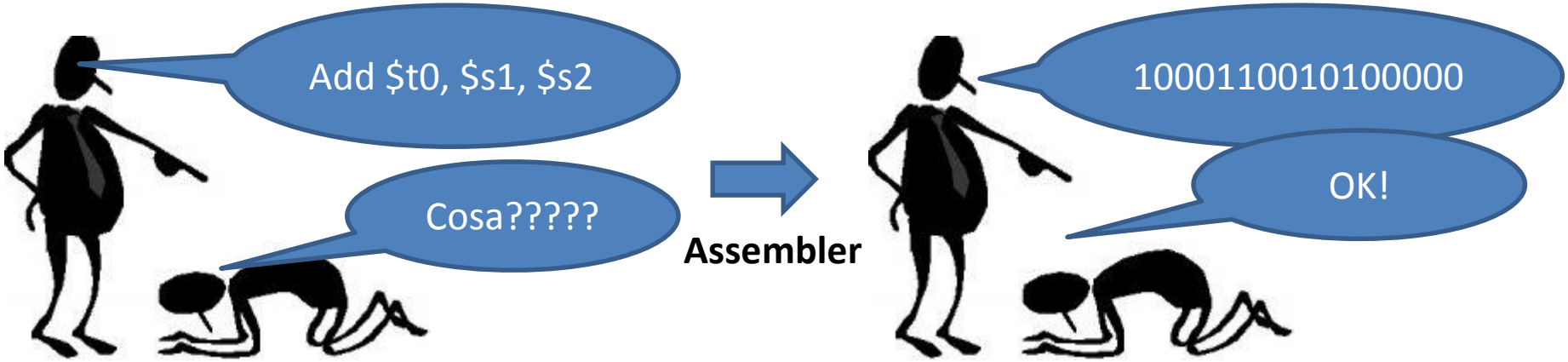
ARCHITETTURA
DEL SET DI ISTRUZIONI

Processore MIPS

Linguaggio Macchina

Michele Favalli

Dal linguaggio Assember al Linguaggio Macchina



Come passare dal linguaggio assembler al linguaggio macchina?

1. Ci deve essere un criterio per **mappare registri «logici» in registri fisici**
   - MIPS: i registri $s0-…$s7 vengono mappati sui registri 16-…23
     i registri $t0-…$t7 vengono mappai sui registri 8-…15
2. Occorre definire il **formato delle istruzioni**

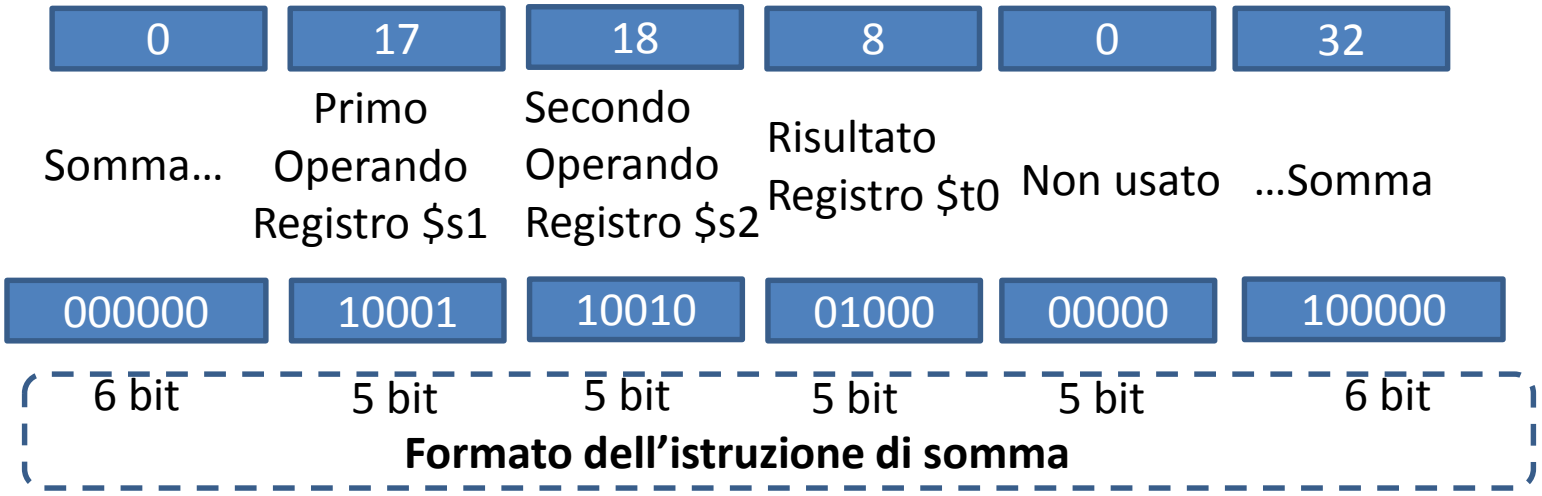
Lunghezza totale possibilmente fissa

I «fields» (gruppi di bit contigui) assumono significato diverso a seconda del tipo di istruzione

Esempio: Architettura MIPS

```
add   $t0, $s1, $s2     # somma $s1 ed $s2, salvando il risultato in $t0
```

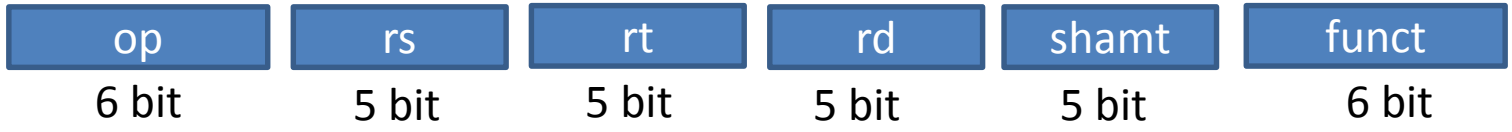
Notazione simbolica per il formato istruzioni mediante numeri decimali



Tutte le istruzioni MIPS sono lunghe 32 bit;
proprio come la dimensione delle parole dati, e dei registri

Formato Istruzioni MIPS

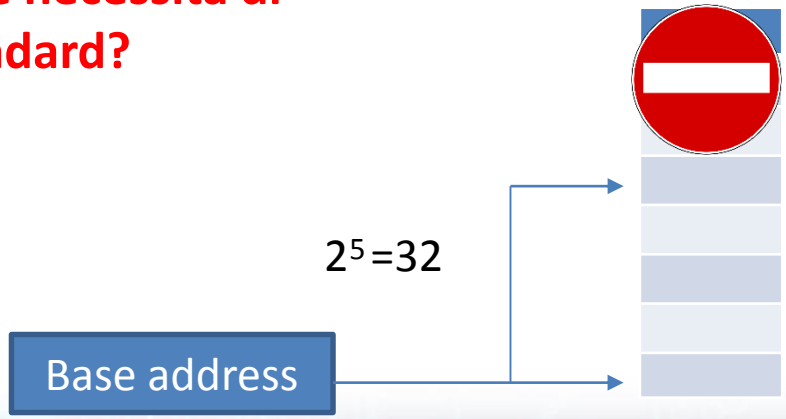
Principio di progettazione:
In generale sarebbe preferibile avere un unico formato per tutte le istruzioni.



OP: OPCODE, è la operazione di base
RS: primo registro operando
RT: secondo registro operando
RD: registro destinazione
SHAMT: ammontare dello shift
FUNCT: FUNCTION CODE, specifica la variante di OP

Cosa succede quando una istruzione necessita di field più lunghi rispetto a quelli standard?

Se uso rs/rt/rd come offset per una load, diverse locazioni di un array potrebbero risultare irraggiungibili!

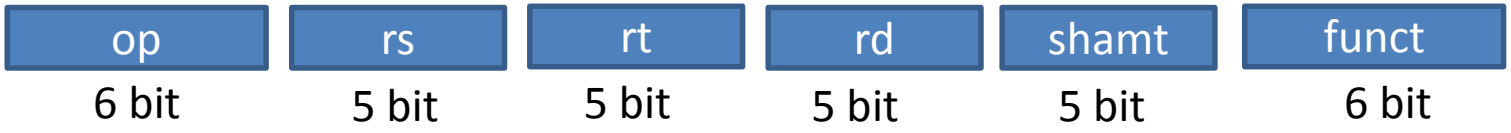


Formati istruzione multipli

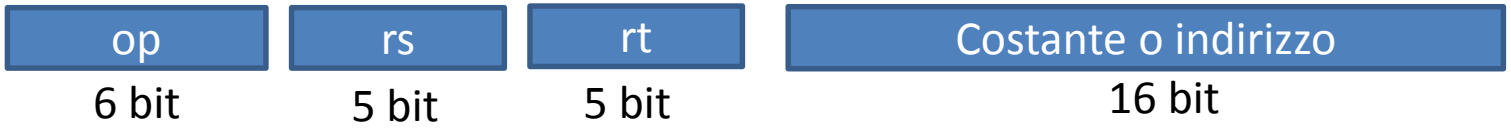
Principio di progettazione:
Soluzioni progettuali efficienti richiedono buoni compromessi

Soluzione trovata dall'architettura MIPS:
Stessa lunghezza per le istruzioni, ma diverso formato!

R-type format

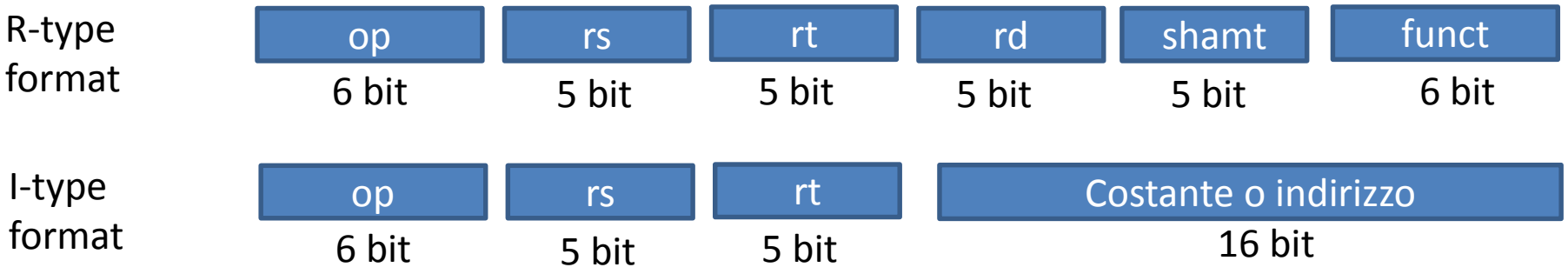


I-type format



- **I-Type**: per istruzioni **«immediate»** o per **istruzioni per il trasferimento dati (load, store)**
- Con istruzioni I-type, una **load** può accedere a qualunque parola entro una regione di +/- (2^15-1) (32767) byte rispetto all'indirizzo del registro base **rs**.
- Analogamente, l'istruzione **addi** può sommare una costante di +/- (2^15-1).
- Notare la problematicità di avere più di 32 registri!

Formati istruzione multipli



- **E' possibile aggiungere altri registri all'architettura?**
  No, altrimenti 5 bit non sono più sufficienti ad indicare i registri, e una parola da 32 bit non è più sufficiente per codificare le istruzioni.
- **Anche in presenza di formati istruzione multipli, si può ridurre la complessità architetturale per il loro supporto rendendo i formati tra loro simili:**
  - *Nel tipo I, un field unico rimpiazza i tre field più a destra del formato R.*
  - *I primi tre field dei formati I ed R hanno la stessa lunghezza.*
- **Ciascun opcode è univocamente associato ad un formato istruzione (I oppure R), così che l'hardware sa come trattare gli ultimi 16 bit**: *come 3 field oppure come uno unico!*

Overview sul formato di alcune istruzioni

Instruction	Type	Op	rs	rt	rd	shamt	funct
add	R	0	src reg	src reg	dst reg	0	32
sub	R	0	src reg	src reg	dst reg	0	34
addi	I	8	src reg	dst reg	costante		
lw	I	35	base reg	dst reg	offset		
sw	I	43	base reg	src reg	offset		

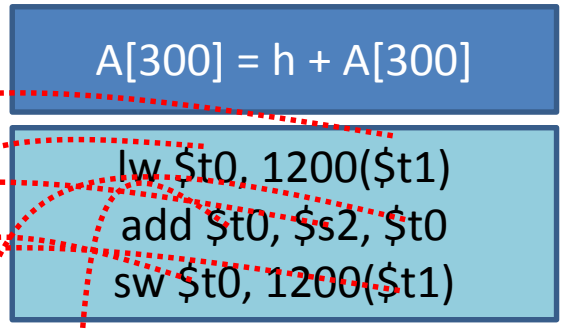
Per semplicità, sono stati usati numeri decimali

- Notare che **add** e **sub** si differenziano solo per il campo *funct*, avendo l'opcode in comune
- Notare che **add** è di tipo R mentre **addi** è di tipo I
- Notare che *rt* è a volte un registro sorgente, a volte destinazione, mentre *rs* è sempre sorgente.
  => Nella microarchitettura ci sarà sicuramente un multiplexer

Esempio

Ipotesi:

- *Il registro $t1 contiene l'indirizzo base dell'array A*
- *Il registro $s2 viene associato dal compilatore alla variabile di programma h*



Linguaggio C

Linguaggio assembler

Op	Rs	Rt	Rd	Shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Linguaggio macchina in notazione decimale

Op	Rs	Rt	Rd	Shamt	funct
100011	01001	01000	0000010010110000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000010010110000		

Codice macchina

Operazioni Logiche

- Con l'esperienza, divenne chiaro che non bastava agire sulle «parole», ma anche sui singoli bit che le componevano!

Da questa esigenza nacquero le istruzioni logiche

Esempio:

- **Istruzione «Shift Left» (sll** in MIPS**)**
  - ✓ Effettua la traslazione di una parola binaria a sinistra di *n* posizioni, riempiendo le posizioni vuote con uno zero. Con n=4 si ha:

9 << 4

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two}$$

Equivale a moltiplicazione per 2^4

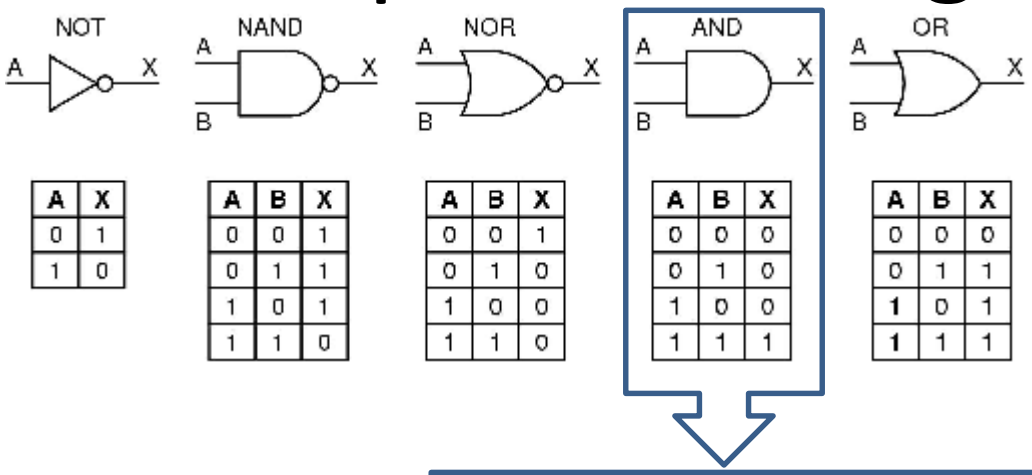
In assembler:

```
sll  $t2, $s0, 4
```

 # Trasla a sinistra di 4 posizioni il contenuto di $s0; risultato in $t2.

- **Analogamente per l'istruzione «Shift Right n» (srl** in MIPS**) ⇔ Divisione per 2^n**

Operazioni logiche bit a bit



A	X
0	1
1	0

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

«0» e «1» corrispondono alle nozioni intuitive del «vero» e «falso»

Registro $t2 **(dato)** 0000 0000 0000 0000 0000 1101 0000 0000$_{two}$

Registro $t1 **(maschera)** 0000 0000 0000 0000 0011 1100 0000 0000$_{two}$

Istruzione `and $t0,$t1,$t2`

Registro risultato $t0 0000 0000 0000 0000 0000 1100 0000 0000$_{two}$

Uno «0» nella maschera ordina di «nascondere» i corrispondenti bit del dato

Operazioni logiche

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Corrispondenza tra operatori logici in C, Java e MIPS ISA

Riassunto

Category	Instruction	Example	Meaning	Comments	
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands; overflow detected	R-type
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 - $s3	Three operands; overflow detected	R-type
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	+ constant; overflow detected	I-type
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND	R-type
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR	R-type
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR	R-type
	and immediate	`andi $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND reg with constant	I-type
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR reg with constant	I-type
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant	R-type
	shift right logical	`srl $$s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant	R-type
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register	I-type
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory	I-type

Il set di istruzioni MIPS si va allargando…..

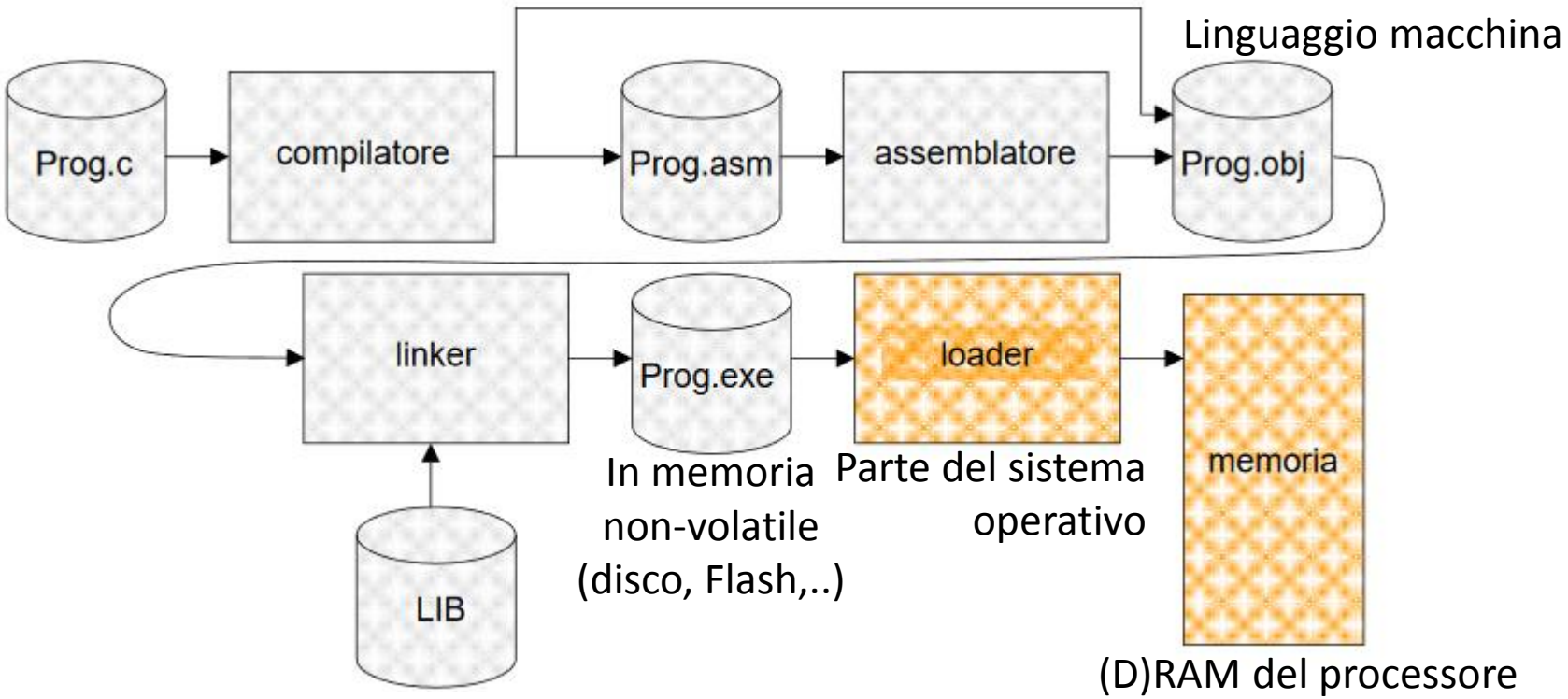
- ***Le istruzioni sono rappresentate come numeri binari, esattamente come i dati.***

  10010101101010101011001100001110

  Istruzione o dato?

La interpretazione dipende da chi ne fa la lettura dalla memoria: control path oppure data path

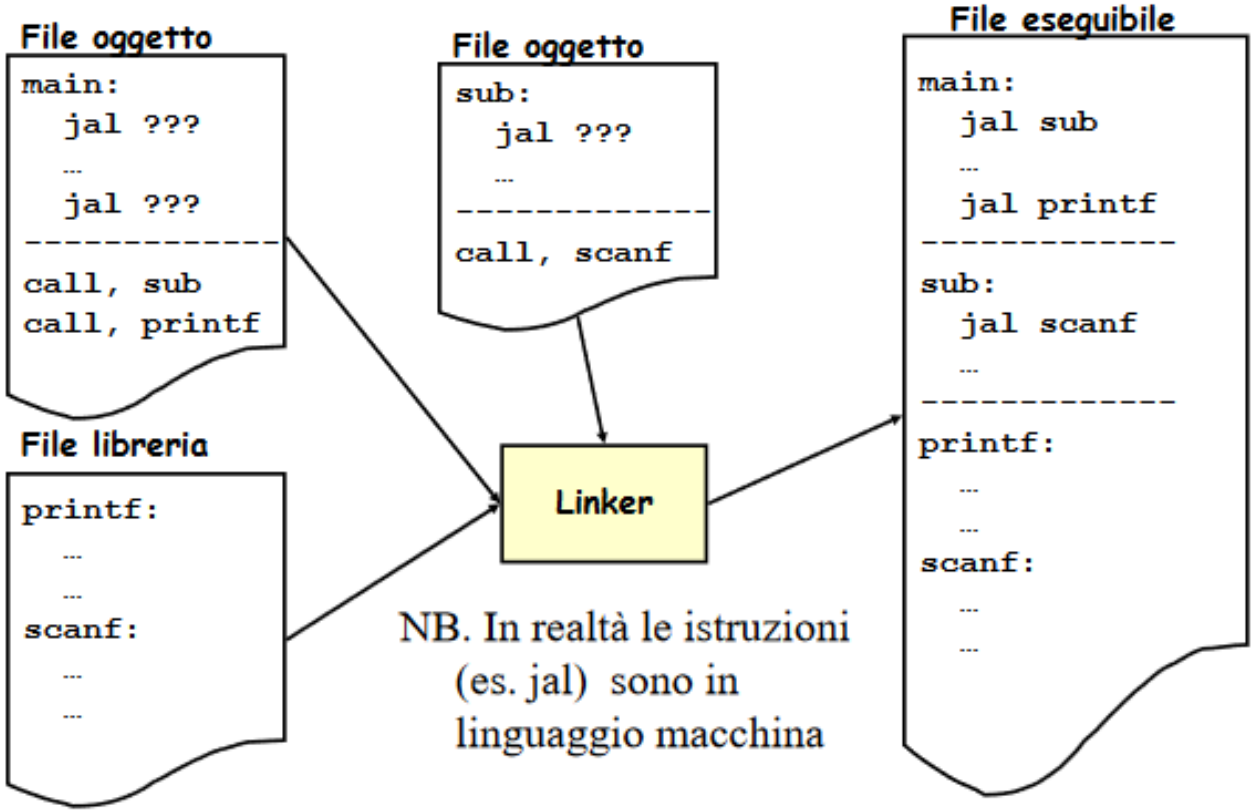
Visione complessiva



Le librerie vengono incluse nella loro interezza, in modo che l'eseguibile sia auto-consistente.
In alternativa, le librerie possono essere caricate dinamicamente.

Linker: combina i file .obj in un unico .exe

1. **Ricercare nelle librerie le routine invocate dal programma**
2. **Determinare i riferimenti assoluti di memoria di istruzioni e dati**
3. **Risolvere i riferimenti tra i diversi fiel**



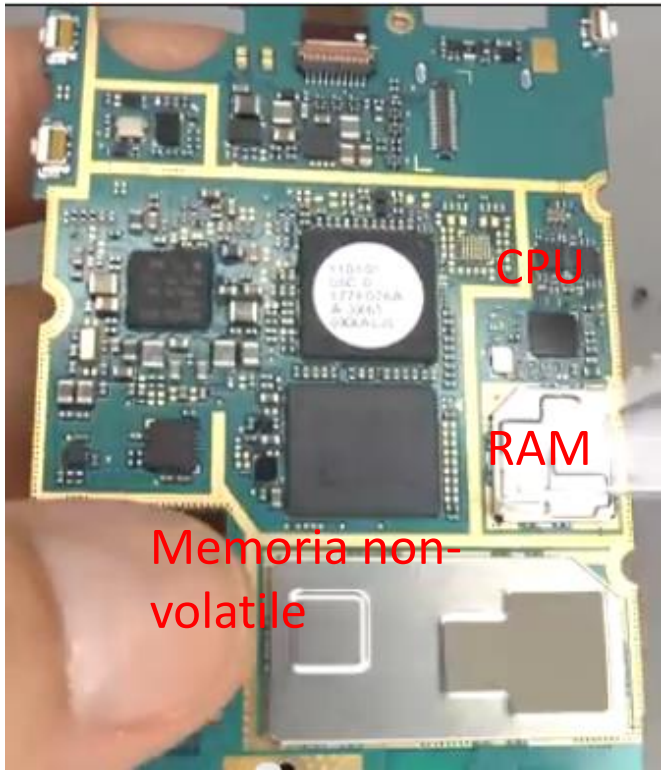
L'eseguibile non deve contenere «unresolved references»

NB L'istruzione «jal» (che vedremo) consente la chiamata di subroutines

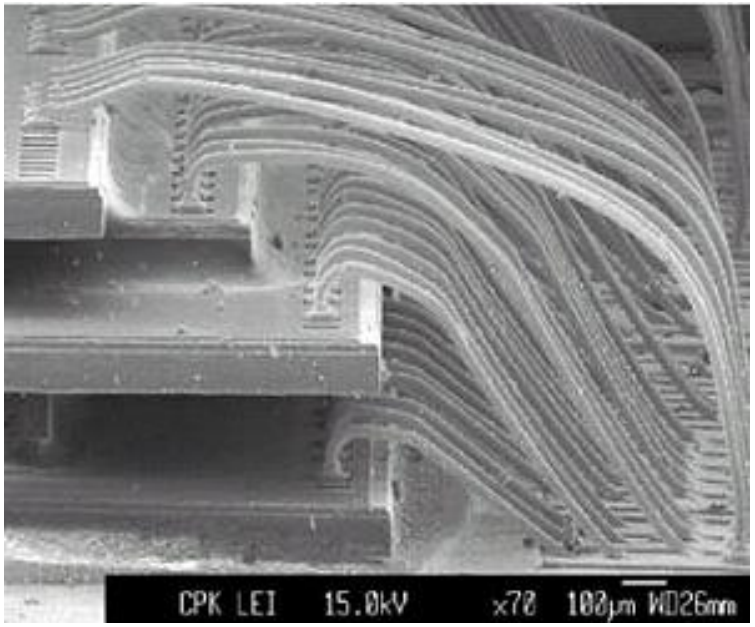
Loader

- Entra in gioco quando si invoca l'esecuzione di un programma sulla macchina target
- Lavora in due fasi
  - Fase di caricamento:
    - legge l'header del file eseguibile e determina le dimensioni del segmento codice e dati.
    - Alloca spazio in memoria sufficiente per codice, dati e stack.
    - Copia istruzioni e dati dal file eseguibile allo spazio di memoria allocato (possono esserci rilocazioni di codice)
  - Fase di attivazione
    - Copia eventuali argomenti passati al programma sullo stack
    - Inizializza i registri del processore
    - Salta alla routine di startup, che copia gli argomenti nei registri e inizializza il PC (program counter)

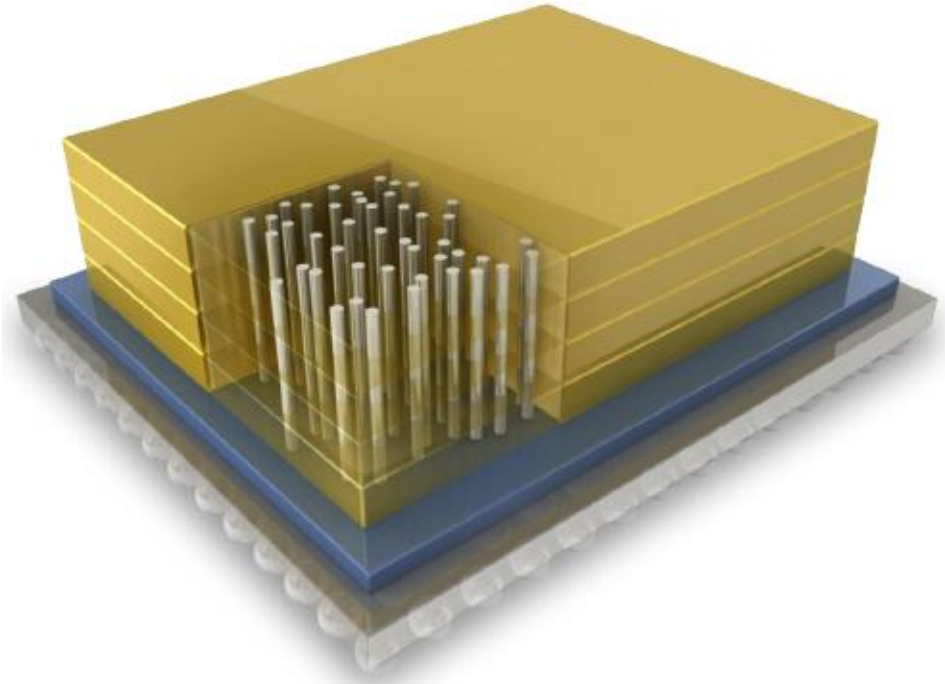
- ***I programmi (istruzioni, dati) sono immagazzinati nella memoria non-volatile del sistema, da cui sono letti quando viene invocato un programma e trasferiti in memoria volatile RAM.***
- ***Dalla RAM vengono lette (fetch) le istruzioni, e vengono effettuate le LOAD/STORE dei dati usando i registri interni del processore come destinazione/sorgente.***
  - **Fetch istruzioni e Dati nei registri.**
  - **Dati: attraverso load/store**
  - **Istruzioni: indirizzamento diretto della memoria tramite il registro PC**



Co-packaging di CPU e memoria

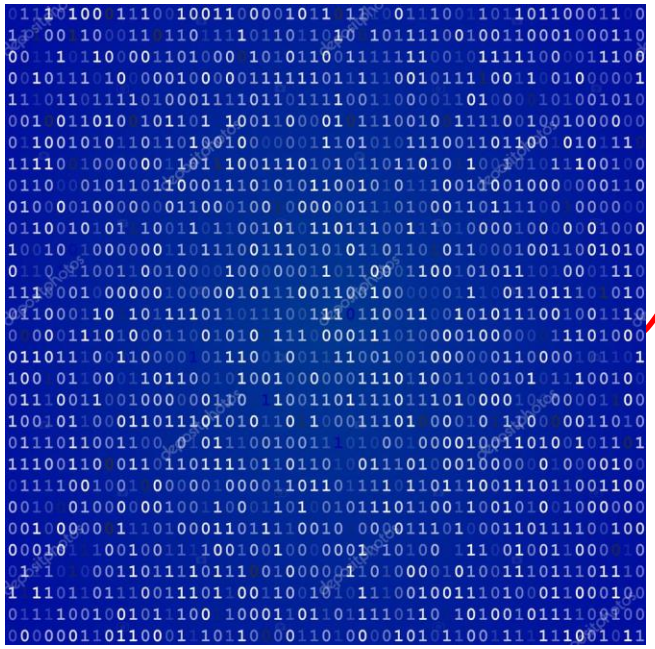


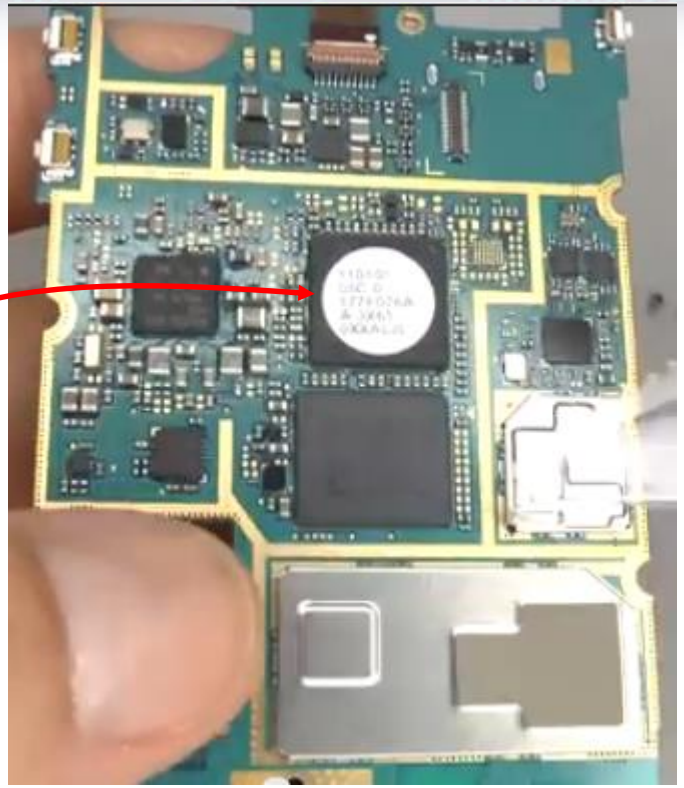
Paradigma System-in-Package (SiP)
Layer di base: CPU
Layer superiori: Memoria, ecc.
Collegamento tramite «Wire bonding»



Micron All-Silicon HMC Cube. Source: Koopmans, Micron, SEMI Strategic Materials Conference 2013

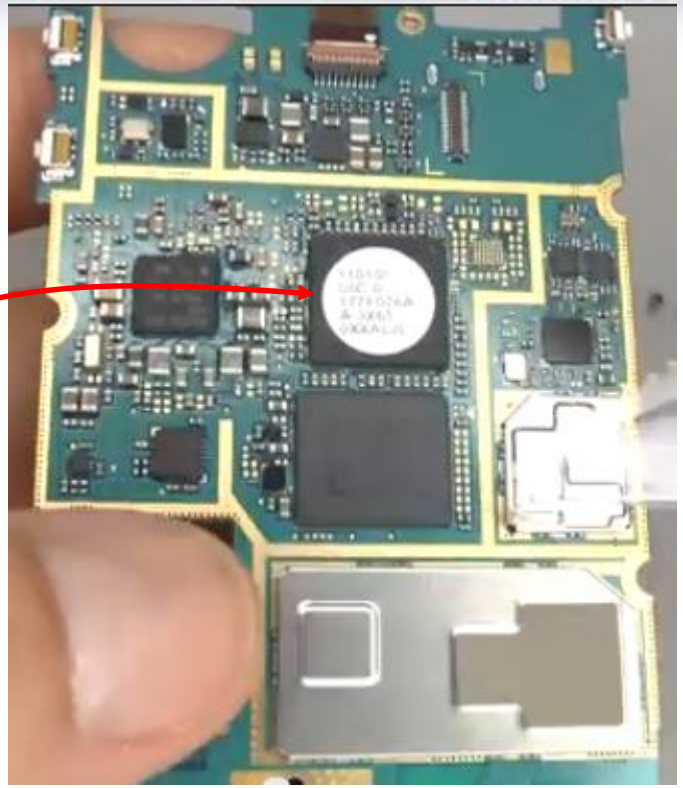
Paradigma 3D stacking
Layer di base: CPU
Layer superiori: memorie, ecc.
Collegamento mediante «through-silicon vias»





I programmi sono forniti come file di numeri binari! Il problema è se questi «binari» siano o meno compatibili con il set di istruzioni del computer su cui vengono fatti «girare» (problema della **«binary compatibility»**).
Ciò porta l'industria ad allinearsi su un numero limitato di set di istruzioni, per favorire la «**portabilità**» del codice.


```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```



Se visualizziamo un file di codice macchina con un editor opportuno, normalmente ciò che vediamo è una sequenza di numeri…..esadecimali!

Si tratta ovviamente solo di un modo semplice per visualizzare numeri binari, i soli che il processore capisce!