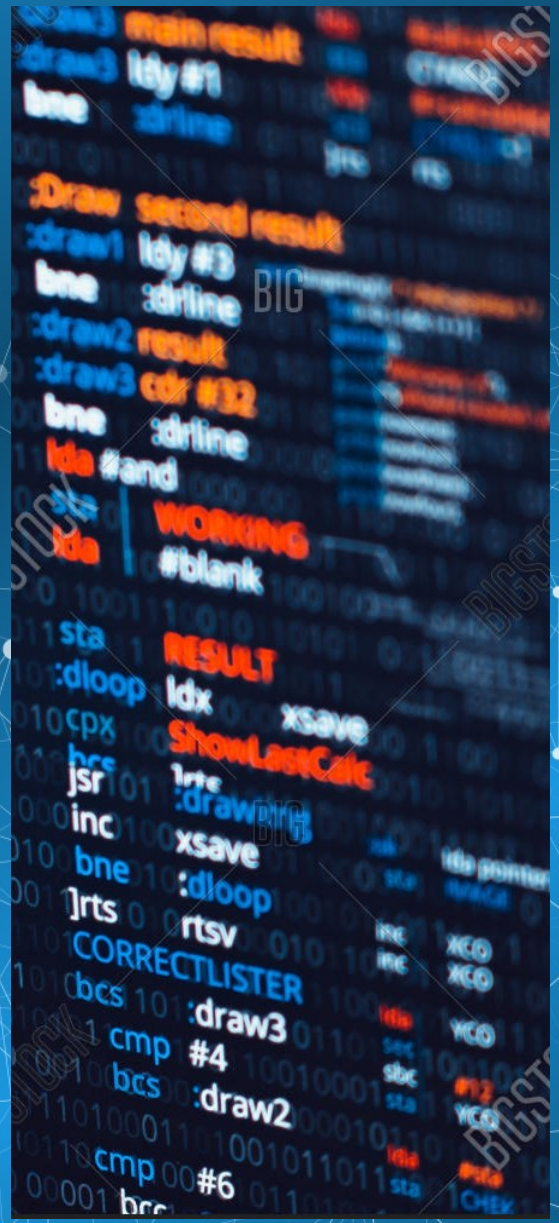
ISLab Intelligent Systems Laboratory

UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Formati di istruzioni e indirizzamento

Prof. Giuseppe Pirlo | Prof. Donato Impedovo | Dott. Stefano Galantucci

Architettura degli Elaboratori e Sistemi Operativi @ Corso di Laurea in Informatica



Architettura CISC e architettura RISC

Architettura CISC

Complex Instruction Set Computer

- Indica un'architettura per microprocessori formata da un set di istruzioni contenente istruzioni in grado di eseguire operazioni complesse (es. lettura di un dato in memoria, modifica e salvataggio direttamente in memoria tramite una singola istruzione)
- ES: I processori dell'architettura x86 di Intel
- Spesso traducono l'istruzione da CISC in un lotto di operazioni elaborate come RISC
- Spesso sono microprogrammate: all'interno della CPU stessa è presente un programma che traduce le istruzioni in fase di decodifica
- Vantaggi: avvicinano il linguaggio macchina ai linguaggi di alto livello

Architettura RISC

Reduced Instruction Set Computer

- Indica un'architettura per microprocessori che sceglie un set di istruzioni più semplice e lineare
- Vantaggi: Eseguono le operazioni in maniera più veloce rispetto alla CISC

Tipi di istruzioni

Istruzioni di trasferimento dati:

Poter copiare dati da una locazione all'altra è fondamentale

Un esempio pratico sono le istruzioni di assegnamento (ad es. A=B copia in A i bit contenuti nella locazione di B)

I dati possono avere due sorgenti: la memoria o i registri, ergo ci sono 4 tipi di trasferimento possibili. Alcune architetture usano un'unica istruzione per tutte le tipologie, altre ad esempio utilizzano le seguenti istruzioni:

- LOAD per copia dalla memoria ai registri
- STORE per copia dai registri alla memoria
- MOVE per copia dai registri ai registri
- In tal caso generalmente non è prevista un'istruzione per copiare dalla memoria alla memoria

Tipi di istruzioni

Operazioni binarie:

Producono un risultato dalla combinazione di due operandi. Tutti gli ISA, ad esempio, posseggono operazioni per somma e sottrazione tra interi.

Tra le operazioni binarie vi sono anche le operazioni booleane. In genere sono disponibili le operazioni AND, OR (e NOT, che è unario) e a volte si trovano anche XOR, NOR e NAND

Queste ultime operazioni sono *bitwise*: effettuano il calcolo *bit a bit*

Piccola digressione: per ottenere tutte le operazioni booleane è sufficiente avere a disposizione il NOT e una tra AND e OR

- Tramite i teoremi di De Morgan posso ricavare l'OR dall'AND e viceversa
- Utilizzo dunque NOT, OR e AND per generare le altre funzioni booleane mediante le equivalenze logiche (ad esempio $A \text{ XOR } B = A\bar{B} + \bar{A}B$), oppure
- Utilizzo le Mappe di Karnaugh o un sistema equivalente per esprimere la funzione booleana

Tipi di istruzioni

Un uso importante dell'AND è l'estrazione della parola. Ciò avviene mediante l'utilizzo di un AND tra il dato originale e una costante, detta **maschera**, che identifica la parola da estrarre:

A	1	1	0	1	1	1	0	1		0	1	1	0	1	0	1	1		0	0	1	0	1	1	0	1		1	0	1	1	1	0	1	1
B (maschera)	0	0	0	0	0	0	0	0		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**		0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
A AND B	0	0	0	0	0	0	0	0		**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**		0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0

La maschera si comporrà di 1 dove vi sono i bit da estrarre e di 0 negli altri bit

Il risultato verrà infine fatto scorrere in modo da isolare a destra il risultato (nell'esempio sopra: di 16 bit)

Tipi di istruzioni

L'uso dell'OR è quello di impacchettare bit in una parola, ovvero l'operazione complementare all'estrazione:

A	1	1	0	1	1	1	0	1		0	1	1	0	1	0	1	1		0	0	1	0	1	1	0	1		0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**
A OR B	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**		**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**		**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**		**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**

Per sostituire una parola (o dei bit) si utilizzano in sequenza estrazione ed impacchettamento:

A	1	1	0	1	1	1	0	1		0	1	1	0	1	0	1	1		0	0	1	0	1	1	0	1		1	1	0	1	1	0	1	0
B (mask)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**		0	0	0	0	0	0	0	0
A AND B	1	1	0	1	1	1	0	1		0	1	1	0	1	0	1	1		0	0	1	0	1	1	0	1		0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0		**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**
A AND B OR C	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**		**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**		**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**		**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**

B viene utilizzata come maschera per cancellare i bit da sostituire (A AND B – estrazione dei bit da preservare), i nuovi bit vengono aggiunti tramite C (impacchettamento mediante OR C)

Tipi di istruzioni

Tra le operazioni unarie troviamo il NOT che calcola il complemento binario (gli 0 diventano 1 e gli 1 diventano 0) e il NEG che calcola l'inverso del valore numerico (es 5 diventa –5)

Le due operazioni, nel caso in cui i numeri siano rappresentati in complemento a uno, coincidono, in tutti gli altri casi il NEG dipende ovviamente dall'implementazione dei numeri

Tipi di istruzioni

Tra le operazioni unarie si trovano anche lo scorrimento (*shift*) e la rotazione:

0	1	0	0	0	0	0	0	0	1	0	1	0	1	1	0
0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	1
1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	1

A

A scorso a destra di 2 bit

A ruotato a destra di 2 bit

Nello shifting, i bit che fuoriescono dalla parola si perdono e gli spazi vuoti che si generano nella direzione opposta allo shifting vengono riempiti con zeri

Sono possibili anche shift e rotazioni (verso destra) con l'*estensione del segno*: in tal caso le posizioni vuote che si generano sulla sinistra vengono riempite con il bit del segno originale

Le moltiplicazioni e le divisioni per 2^k sono un'applicazione importante dello shifting (nei numeri positivi)

Le operazioni di rotazione sono generalmente utilizzate insieme alle già citate operazioni di estrazione e impacchettamento per lo spostamento dei bit

Le operazioni di rotazione non comportano perdita di informazione a differenza delle operazioni di shifting

Tipi di istruzioni

Istruzioni di confronto:

- Uguaglianza tra parole
- Verificare se una certa parola è zero (molto usata)
- Confronto di maggioranza o minoranza tra numeri (dipende naturalmente dall'implementazione del numero)

Istruzioni di input/output:

Interagiscono con i dispositivi di I/O e sono di tre tipologie:

- I/O programmato con attesa attiva
- I/O *interrupt driven*, che viene innescato dagli interrupt
- I/O con DMA – ovvero l'I/O programmato ma viene aggiunto il componente chip *Direct Memory Access* che ha accesso diretto al bus

Tipi di istruzioni

Istruzioni di salto (JUMP):

Il codice è scritto mediante l'ausilio di etichette, che definiscono «sezioni» del programma

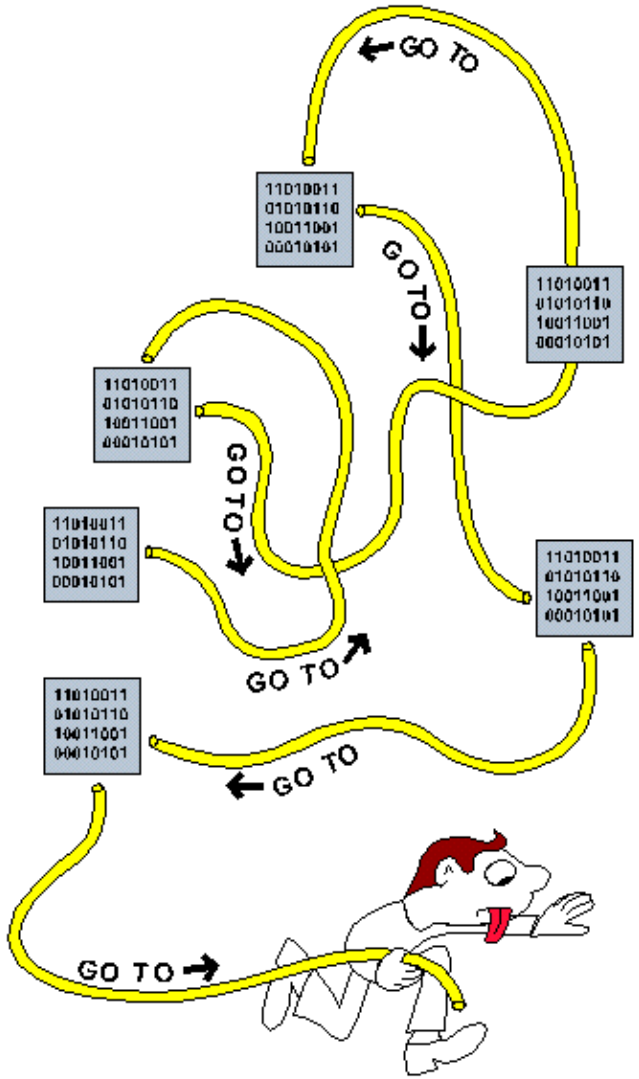
Le istruzioni di salto consentono al programma di passare da una sezione all'altra

Si dividono in:

- Salti incondizionati: (GOTO) si passa direttamente ad un'altra sezione del programma
- Salti condizionati: si verifica una condizione e si passa ad un'altra sezione del programma a seconda del risultato della verifica (un esempio pratico è l'*if*) – viene generalmente implementato mediante il calcolo della condizione e la memorizzazione del risultato in un registro, successivamente viene effettuato il salto condizionato se il valore del registro è 0/1

Istruzioni di ciclo: vengono convertite in salti condizionati

Spaghetti code



Spaghetti Sample

```
1.058 JMPL RBW_SWITCHING
1.059 LABEL FREQUENCY_READOUT
1.060 CALL Sub Frequency Readout
1.061 CALL Sub Frequency Span
1.062 CALL Sub  Noise Sidebands
1.063 JMPL SYSTEM_SIDEBANDS
1.064 LABEL WIDE_OFFSETS
1.065 CALL Sub Noise Wide Offsets
1.066 JMPL SPURIOUS
1.067 LABEL SYSTEM_SIDEBANDS
1.068 CALL Sub System Sidebands
1.069 CALL Sub Residual
1.070 CALL Sub Display Switching
1.071 JMPL SCALE_FIDELITY
1.072 LABEL SWEEP_TIME
1.073 CALL Sub Sweep Time
1.074 JMPL WIDE_OFFSETS
1.075 LABEL SCALE_FIDELITY
1.076 CALL Sub Scale Fidelity
1.077 CALL Sub Input Switching
1.078 CALL Sub Reference Level
1.079 JMPL FREQUENCY_RESPONSE
1.080 LABEL RBW_SWITCHING
1.081 CALL Sub Resolution BW Switching
1.082 JMPL RESOLUTION_BANDWIDTH
1.083 LABEL ABSOLUTE_AMPLITUDE
1.084 CALL Sub Absolute Amplitude
1.085 JMPL FREQUENCY_READOUT
1.086 LABEL RESOLUTION_BANDWIDTH
1.087 CALL Sub Resolution Bandwidth
1.088 JMPL AVERAGE_NOISE
1.089 LABEL FREQUENCY_RESPONSE
1.090 CALL Sub Freq Response
1.091 JMPL SWEEP_TIME
1.092 LABEL AVERAGE_NOISE
1.093 CALL Sub Displayed Average
1.094 JMPL RESIDUAL_RESPONSES
1.095 LABEL SPURIOUS
1.096 CALL Sub TOI
1.097 JMPL GAIN_COMPRESSION
1.098 LABEL 2ND_HARMONIC
1.099 CALL Sub 2nd Harmonic
1.100 CALL Sub Other Spurs
1.101 JMPL END
1.102 LABEL GAIN_COMPRESSION
1.103 CALL Sub GC
1.104 JMPL 2ND_HARMONIC
1.105 LABEL RESIDUAL_RESPONSES
1.106 CALL Sub Residual Responses
1.107 JMPL ABSOLUTE_AMPLITUDE
1.108 LABEL END
```

Structured Sample

```
1.058 JMPL  ResolutionBandwidth
1.059 LABEL ResolutionBandwidth_done
1.060 JMPL  ResolutionSwitching
1.061 LABEL ResolutionSwitching_done
1.062 JMPL  DisplayedAverage
1.063 LABEL DisplayedAverage_done
1.064 JMPL  ResidualResponses
1.065 LABEL ResidualResponses_done
1.066 JMPL  AbsoluteAmplitude
1.067 LABEL AbsoluteAmplitude_done
1.068 JMPL  FrequencyReadout
1.069 LABEL FrequencyReadout_done
1.070 JMPL  FrequencySpan
1.071 LABEL FrequencySpan_done
1.072 JMPL  NoiseSidebands
1.073 LABEL NoiseSidebands_done
1.074 JMPL  SystemSidebands
1.075 LABEL SystemSidebands_done
1.076 JMPL  Residual
1.077 LABEL Residual_done
1.078 JMPL  Display Switching
1.079 LABEL Display Switching_done
1.080 JMPL  ScaleFidelity
1.081 LABEL ScaleFidelity_done
1.082 JMPL  InputSwitching
1.083 LABEL InputSwitching_done
1.084 JMPL  ReferenceLevel
1.085 LABEL ReferenceLevel_done
1.086 JMPL  FreqResponse
1.087 LABEL FreqResponse_done
1.088 JMPL  SweepTime
1.089 LABEL SweepTime_done
1.090 JMPL  NoiseWideOffsets
1.091 LABEL NoiseWideOffsets_done
1.092 JMPL  TOI
1.093 LABEL TOI_done
1.094 JMPL  GC
1.095 LABEL GC_done
1.096 JMPL  2ndHarmonic
1.097 LABEL 2ndHarmonic
1.098 JMPL  OtherSpurs
1.099 LABEL OtherSpurs
```

Tipi di istruzioni

Chiamata a funzione:

Quando una funzione termina la propria esecuzione, il programma deve riprendere dall'istruzione successiva alla chiamata a funzione

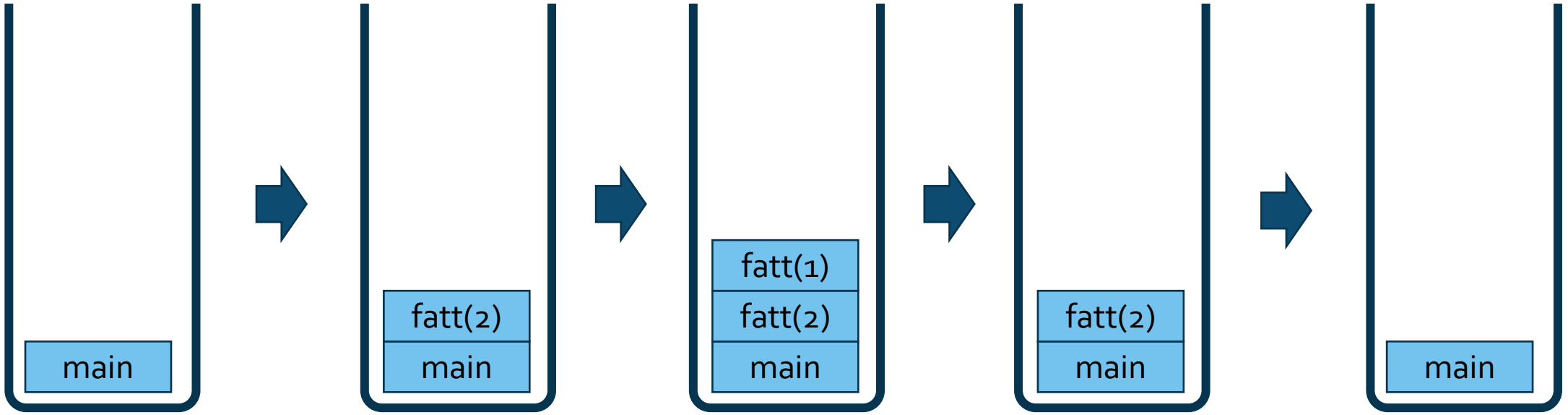
Dunque l'indirizzo di ritorno deve essere memorizzato oppure passato alla funzione chiamata

Il meccanismo così espresso può essere fallace nei casi di funzione che chiama funzione oppure di ricorsione (ovvero una funzione che invoca sé stessa), in quanto potrebbe essere sovrascritto l'indirizzo di ritorno

È pertanto necessario che l'indirizzo di ritorno venga memorizzato ogni volta in una locazione differente

Si utilizza infatti lo **stack dei record di attivazione** (detto anche semplicemente *stack di attivazione* oppure, per sineddoche, *record di attivazione*)

Stack di attivazione: fattoriale ricorsivo



1. Il main invoca la funzione ricorsiva fattoriale (fatt) con argomento n=2

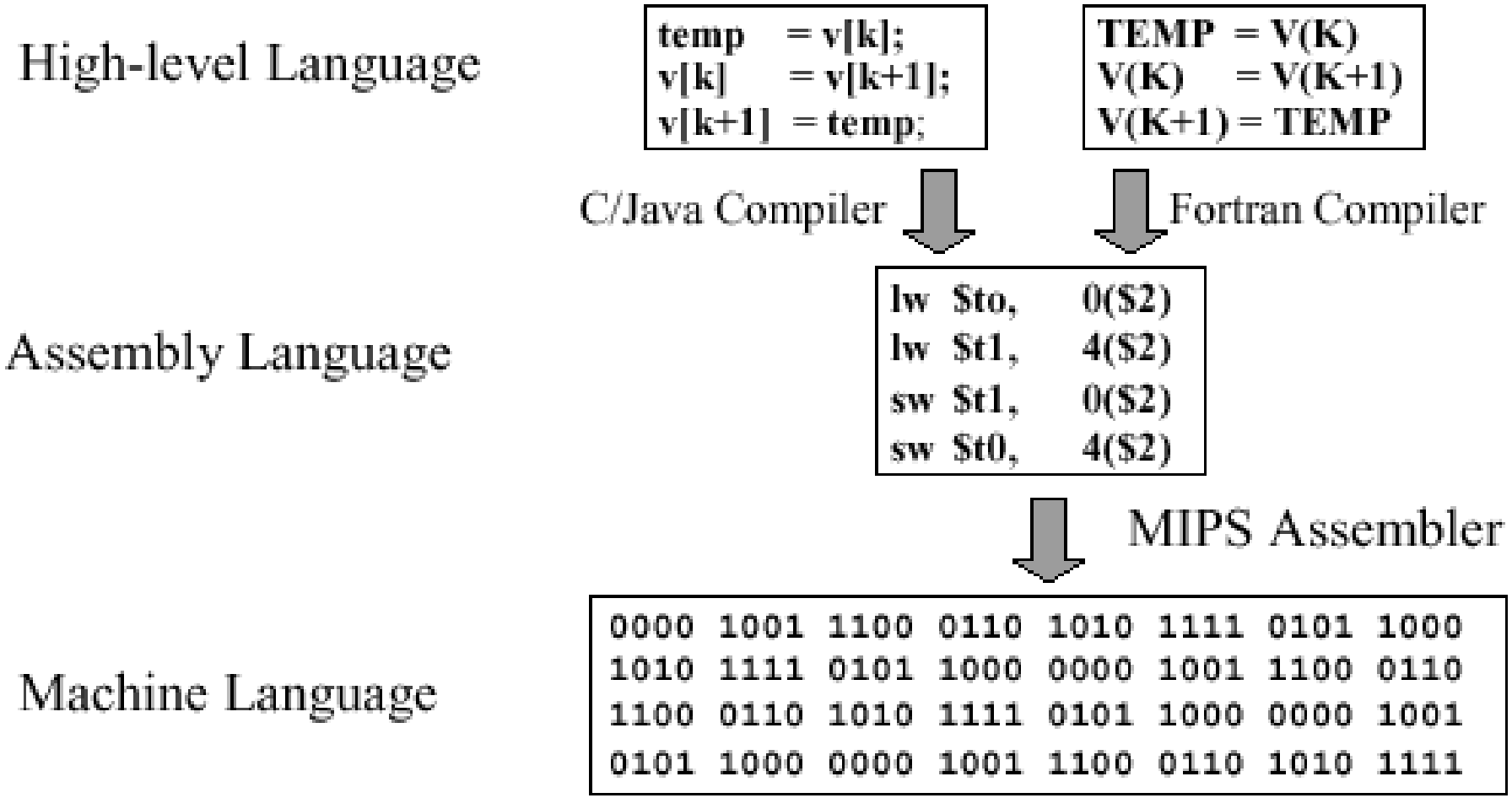
2. La funzione fatt invoca sé stessa con n=1 e viene generato un nuovo record

3. fatt(1) restituisce il valore 1 e il Program Counter si sposta all'indirizzo di ritorno – viene rimosso il record di fatt(1)

4. fatt(2) riceve il valore di output di fatt(1) e produce il proprio output – si torna al main tramite il suo indirizzo di ritorno e viene rimosso il record di fatt(2)

5. Il main riceve il valore di output di fatt(2) e riprende la sua normale esecuzione

Formati di istruzione e indirizzamento



Formati di istruzione e indirizzamento

Un'istruzione si compone di:

- Un **OPCODE**, che identifica l'istruzione da eseguire
- Altre informazioni quali la provenienza degli operandi e la destinazione dei risultati

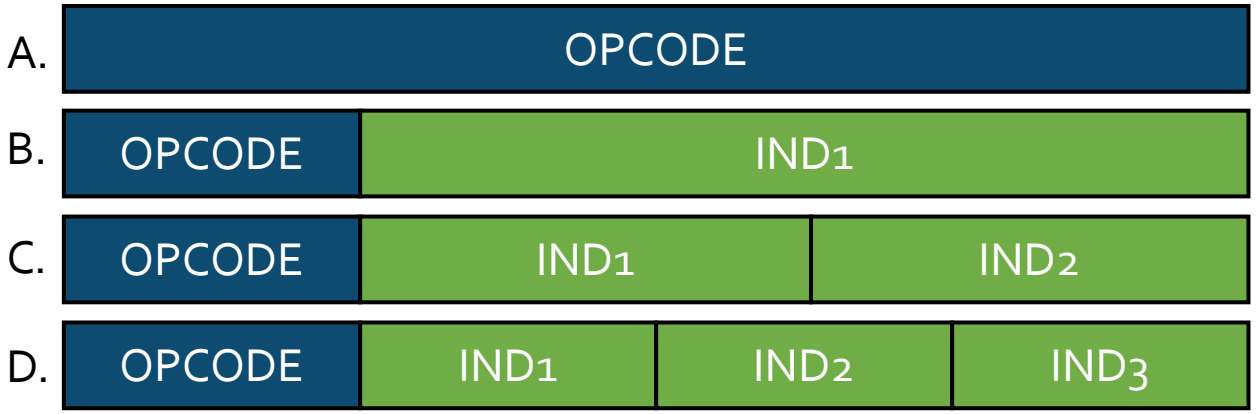
L'argomento che tratta la provenienza degli operandi, ovvero la loro residenza in memoria, prende il nome di **indirizzamento**

Un esempio in MIPS assembler

Label	OPCODE	Destinazione	OP1	OP2	Commento
	move	$a0,	$0		*# $a0=0*
	li	$t0,	99		*# $t0=99*
ciclo:					*# definisco un punto del programma chiamato ciclo*
	add	$a0,	$a0,	$t0	*# $a0 = $a0 + $t0*
	addi	$t0,	$t0,	-1	*# $t0 = $t0 – 1*
	bnez	$t0,	ciclo		*# if ($t0 != zero) vai a ciclo*
	li	$v0,	1		*# stampa il valore in $a0*
	syscall				
	li	$v0,	10		*# termina il programma*
	syscall				

Formati di istruzione

Un esempio di quattro diversi formati di istruzioni (si consideri a parità di dimensione):



- A. Istruzione senza indirizzi
- B. Istruzione con un solo operando
- C. Istruzione con due operandi
- D. Istruzione con tre operandi

Alcuni criteri progettuali

Come gestire le dimensioni delle istruzioni?

1. A parità di progetto, istruzioni più corte sono preferibili:
   - Un programma con istruzioni a 32 bit occupa il doppio dello spazio in memoria dello stesso programma con istruzioni a 16 bit
   - Tuttavia tale fattore potrebbe risultare sempre meno preminente in futuro, in quanto il costo della memoria va sempre più diminuendosi. Dall'altro lato, però, le dimensioni dei software aumentano. Altro fattore da tenere in considerazione, però, è l'ampiezza delle bande di memoria che trasferiscono i dati.
   - Istruzioni più corte sono elaborate più velocemente

Alcuni criteri progettuali

2. E' necessario prevedere spazio sufficiente per rappresentare tutte le istruzioni desiderate:

- Se desideriamo 2^n possibili operazioni differenti, dobbiamo rappresentarle con almeno n bit destinati all'OPCODE.
- È necessario inoltre considerare degli OPCODE in eccedenza per eventuali evoluzioni del progetto

Alcuni criteri progettuali

3. Dimensione degli indirizzi:
   - La dimensione degli indirizzi dipende dalla dimensione delle parole in memoria:
   - $n_{indirizzi} = {dimensione\ memoria}/{dimensione\ parola}$
   - $dimensione\ indirizzo = \lceil \log_2 n_{indirizzi} \rceil$
   - Scegliere la dimensione delle parole tra più o meno bit comporta vantaggi e svantaggi in entrambe le scelte, nei quali però non entriamo nel dettaglio

Alcuni criteri progettuali

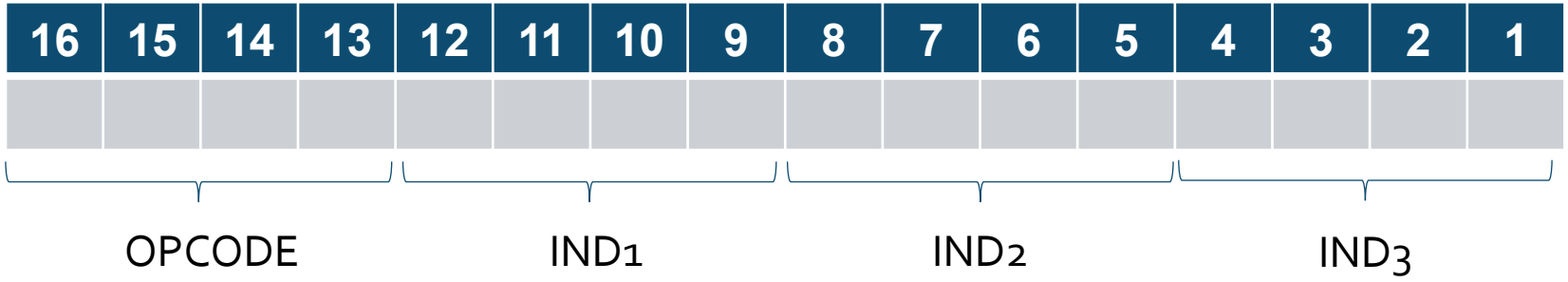
Consideriamo un'istruzione lunga $(n + k)$ bit, dove n è il numero di bit destinati all'OPCODE e k il numero di bit destinati a un operando:

- Abbiamo a disposizione 2^n istruzioni
- Abbiamo a disposizione 2^k parole in memoria da indirizzare

Lo stesso spazio potrebbe essere diviso in $(n - 1)$ bit dedicati all'OPCODE e $(k + 1)$ bit dedicati all'operando, dimezzando il set di istruzioni ma raddoppiando la memoria raggiungibile

Allo stesso modo, potrei dividere in $(n + 1)$ bit dedicati all'OPCODE e $(k - 1)$ bit dedicati all'operando, raddoppiando il set di istruzioni ma dimezzando la memoria raggiungibile

Codice operativo espandibile



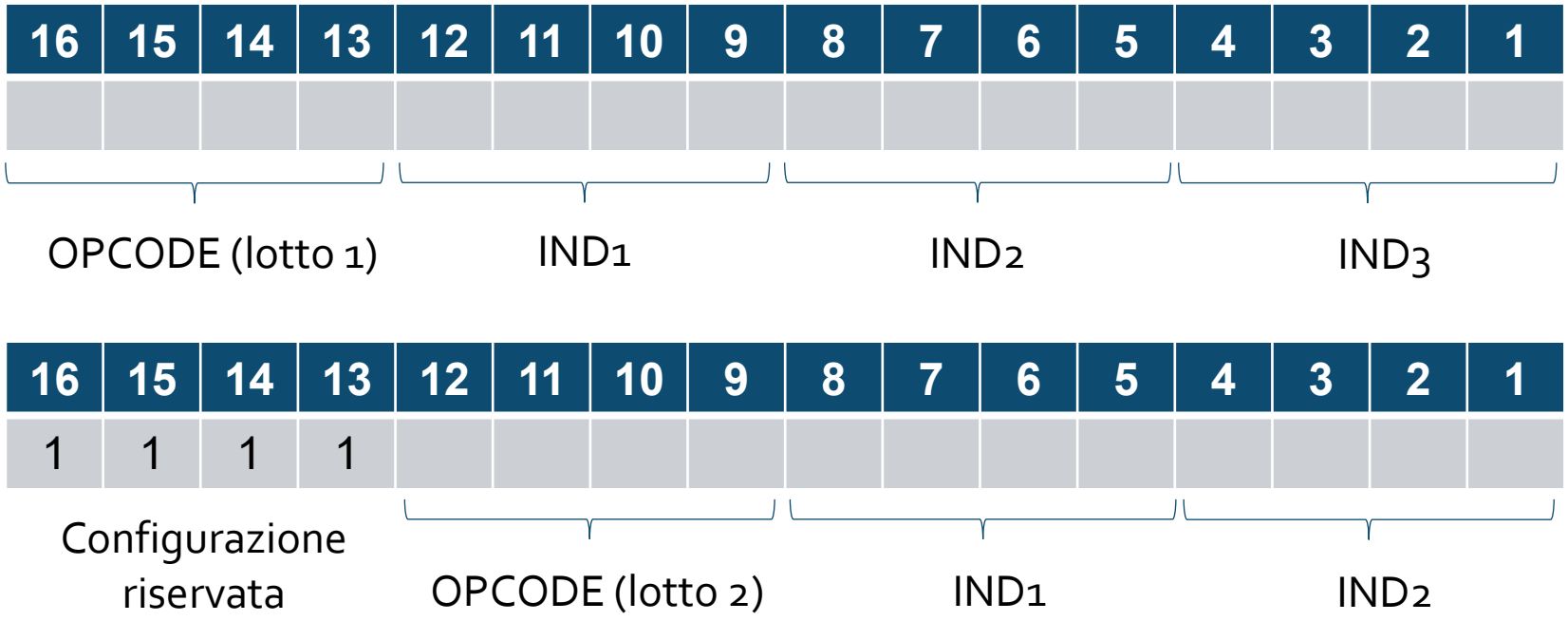
Con questo schema riesco a rappresentare 2^4 istruzioni che operano con fino a tre operandi in memoria

Desidero aumentare il lotto di istruzioni: dovrei aumentare i bit destinati all'OPCODE…

…ma le operazioni che necessitano di meno operandi comportano uno spreco dello spazio destinato agli altri operandi

Soluzione: utilizzo il **codice operativo espandibile** ovvero espando il codice operativo utilizzando i bit degli operandi

Codice operativo espandibile



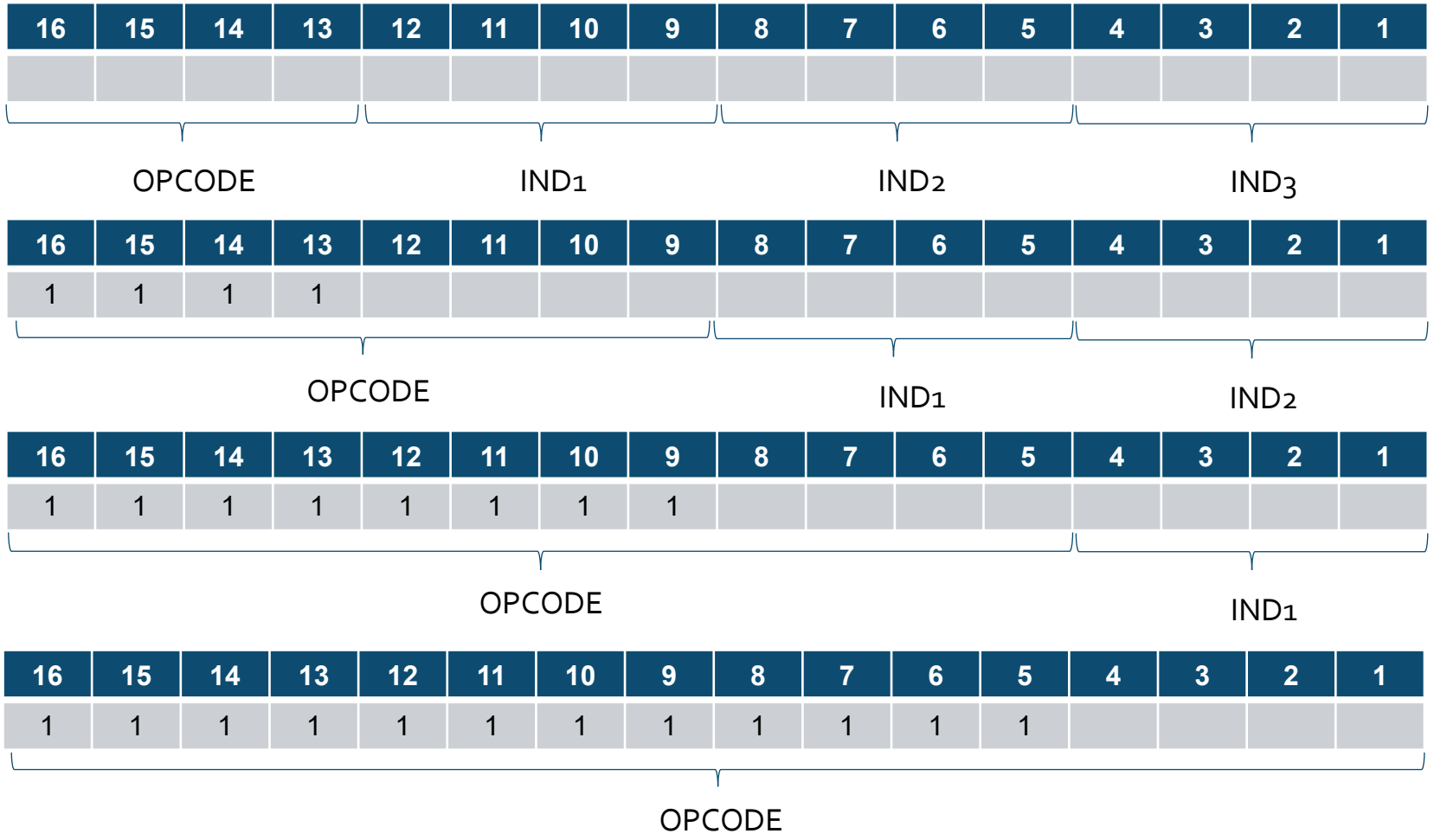
Ho riservato la configurazione 1111 nei bit destinati all'OPCODE, per indicare che l'istruzione è nel formato OPCODE + due operandi.
Di fatto l'OPCODE passa a 8 bit (1111xxxx) nel secondo lotto di istruzioni

Utilizzando questo meccanismo espando il set delle istruzioni a:

- $2^4 - 1$ istruzioni con tre operandi +
- 2^4 istruzioni con due operandi

Codice operativo espandibile

Posso reiterare il meccanismo, ottenendo, ad esempio, con 16 bit di istruzione e indirizzi a 4 bit:



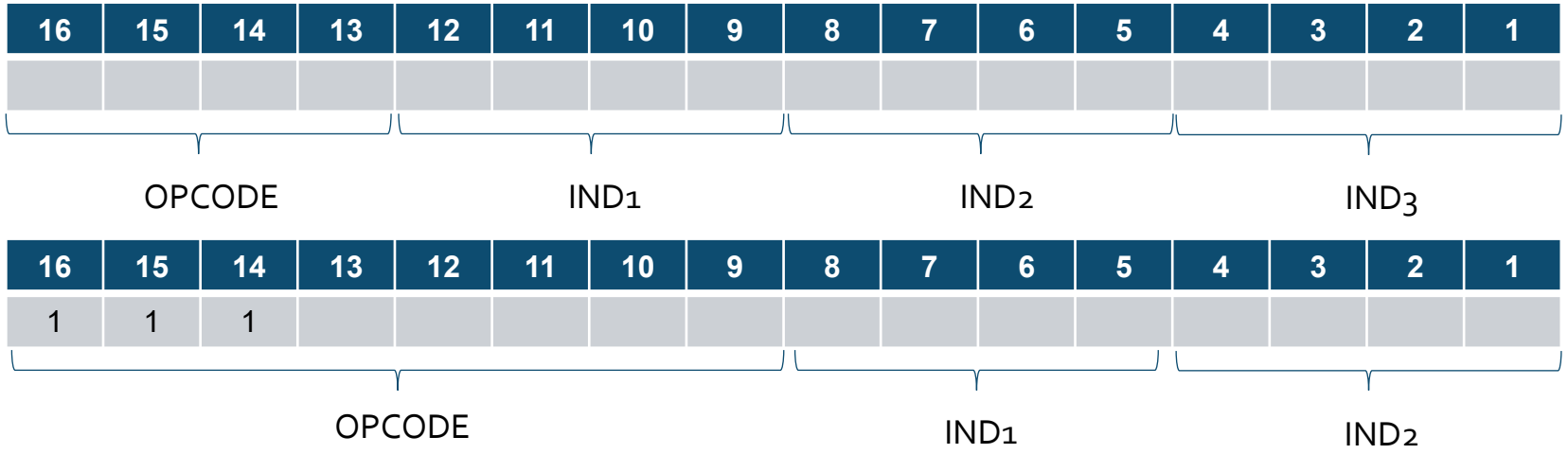
Ottengo un set da:

- $2^4 - 1$ istruzioni a tre operandi
- $2^4 - 1$ istruzioni a due operandi
- $2^4 - 1$ istruzioni a un operando
- 2^4 istruzioni senza operandi

Codice operativo espandibile

La situazione precedentemente descritta è solo un esempio, in quanto si potrebbe scegliere di gestire in qualsiasi maniera differente (in termini numerici) il meccanismo di codice operativo espandibile

Ad esempio, se desidero più istruzioni a due operandi e meno a tre operandi, posso modificare così:



Il set diventa composto da:

- $2^4 - 2$ istruzioni a tre operandi+ (-2 sono le configurazioni 1110 e 1111)
- 2^5 istruzioni con due operandi
- (a meno di ulteriori espansioni)

Indirizzamento

Molte istruzioni contengono operandi, si pone quindi il problema di come specificarne la posizione. Stiamo dunque parlando di **indirizzamento**, che può essere nelle seguenti tipologie:

- Immediato
- Diretto
- A registro
- A registro indiretto
- Indicizzato
- Indicizzato esteso
- A stack

Indirizzamento immediato

L'**indirizzamento immediato** prevede che sia specificato nel campo riservato all'indirizzo, l'operando stesso:

Ad esempio:

MOV	R1	4

Carica la costante 4 nel registo R1

Ha il vantaggio di non richiedere un riferimento supplementare in memoria per effettuare il fetch dell'operando

Naturalmente l'entità del valore è limitata alla dimensione del campo dell'indirizzo

Utilizzato per piccole costanti

Indirizzamento diretto

L'**indirizzamento diretto** utilizza come operando direttamente ciò che è indicato nello spazio riservato all'indirizzo

Ha lo svantaggio di accedere sempre alla medesima locazione, quindi il valore contenuto nella cella di memoria può variare, ma non può cambiare la locazione referenziata

È quindi necessario conoscere in fase di compilazione l'indirizzo che si dovrà referenziare (e non può essere un indirizzo contenuto all'interno di una procedura/funzione, in quanto l'allocazione avverrebbe durante l'invocazione della funzione stessa)

Viene dunque utilizzato unicamente in riferimento a variabili globali

Indirizzamento a registro

L'**indirizzamento a registro** funziona in maniera analoga all'indirizzamento diretto, ma referenzia un registro anziché una locazione nello spazio di indirizzamento

Nota semplicemente come **modalità a registro**

Nelle architetture load/store (ovvero le architetture dove si può operare solo tramite operandi su registri) tutte le istruzioni utilizzano questa modalità, a meno delle istruzioni LOAD e STORE che caricano/scaricano dati in memoria da/verso un registro

Indirizzamento a registro indiretto

Nell'**indirizzamento a registro indiretto** l'operando proviene o è destinato alla memoria, ma l'indirizzo non è contenuto all'interno dell'istruzione

Il campo destinato all'operando contiene un registro che indica la locazione in memoria dell'operando

Quando un indirizzo è utilizzato in questo modo, prende il nome di **puntatore**

Ha il grande vantaggio di non dover indicare in fase di compilazione la locazione della parola in memoria (in quanto potrà essere caricato nel registro a runtime), inoltre la stessa istruzione può essere utilizzata su diverse parole in memoria, semplicemente variando il valore contenuto nel registro

Indirizzamento indicizzato

L'**indirizzamento indicizzato** consente di referenziare una parola in memoria che si trova a un certo spiazzamento rispetto a un registro

L'indirizzamento si ottiene dunque mediante:

- La specifica di un registro (in via esplicita o implicita)
- La specifica di uno spiazzamento costante

Tale meccanismo viene utilizzato in alcune occasioni nelle quali è nota a priori la distanza tra una variabile e l'altra

Può essere utilizzato nel caso opposto: ovvero mantenere un puntatore in memoria nell'istruzione e il piccolo offset in un registro

Indirizzamento indicizzato esteso

Contenuto in alcune macchine, l'**indirizzamento indicizzato esteso** consente di referenziare un indirizzo in memoria ottenuto sommando tra loro il contenuto di due registri, più un eventuale offset aggiuntivo

Disporre di tale possibilità costituisce un grande vantaggio (immaginate scrivere in linguaggio macchina un ciclo che opera su un vettore)

Generalmente le macchine che offrono tale possibilità forniscono anche un offset da 8 o 16 bit

Indirizzamento a stack

Alcune istruzioni possono essere utilizzate in combinazione a una struttura stack

Tale forma di indirizzamento prende il nome di **indirizzamento a stack**

Uno degli esempi pratici di utilizzo è quello legato alla **notazione polacca inversa**

Notazione polacca inversa

Nel mondo reale scriviamo le nostre operazioni algebriche mediante la **notazione infissa**

Si parla di notazione infissa in quanto l'operatore è posto tra gli operandi:

$$A + B$$

Esistono, equivalentemente, la notazione prefissa e la notazione postfissa:

- Notazione prefissa: $+\ A\ B$
- Notazione postfissa: $A\ B\ +$

In tali casi l'operatore è posto prima/dopo gli operandi

Notazione polacca inversa

La notazione postfissa prende anche il nome di **notazione polacca inversa**, dal logico polacco J. Lukasiewicz (1958) che ne ha studiato le proprietà

Presenta alcuni vantaggi rispetto alla notazione infissa:

- Ogni operazione può essere scritta correttamente senza parentesi
- La valutazione delle formule in tale notazione si addice particolarmente ai compilatori con stack
- Gli operatori infissi possiedono un ordine di precedenza, che è arbitrario: ad esempio $a + b \times c$ corrisponde a $a + (b \times c)$ e non a $(a + b) \times c$ perché alla moltiplicazione è assegnato un ordine di priorità più alto

Notazione polacca inversa

Alcuni esempi di notazione polacca inversa:

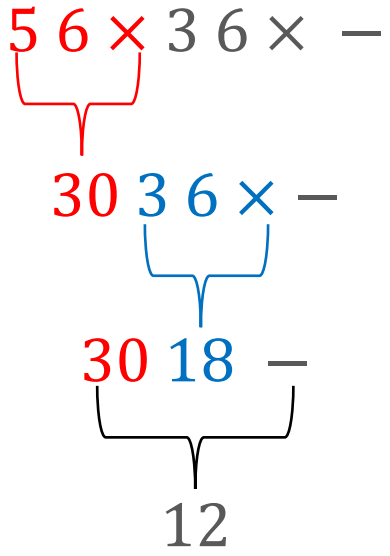
Notazione infissa	Notazione polacca inversa
$A \times B + C$	$A\ B \times C +$
$A + B \times C$	$A\ B\ C \times +$
$A \times B + C \times D$	$A\ B \times C\ D \times +$
$(A + B)\ /\ (C - D)$	$A\ B + C\ D - /$
$\big((A + B) \times C + D\big)\ /\ (E + F + G)$	$A\ B + C \times D + E\ F +\ G\ + /$

(nella parte relativa alla notazione infissa si considera, ovviamente, la moltiplicazione prioritaria rispetto ad addizione e sottrazione)

Notazione polacca inversa

Come si valuta una formula in notazione polacca inversa?

- Quando trovo un valore, lo aggiungo in cima allo stack
- Quando trovo un operando, prendo i due valori in cima allo stack, effettuo l'operazione tra questi due ed inserisco il risultato in cima allo stack



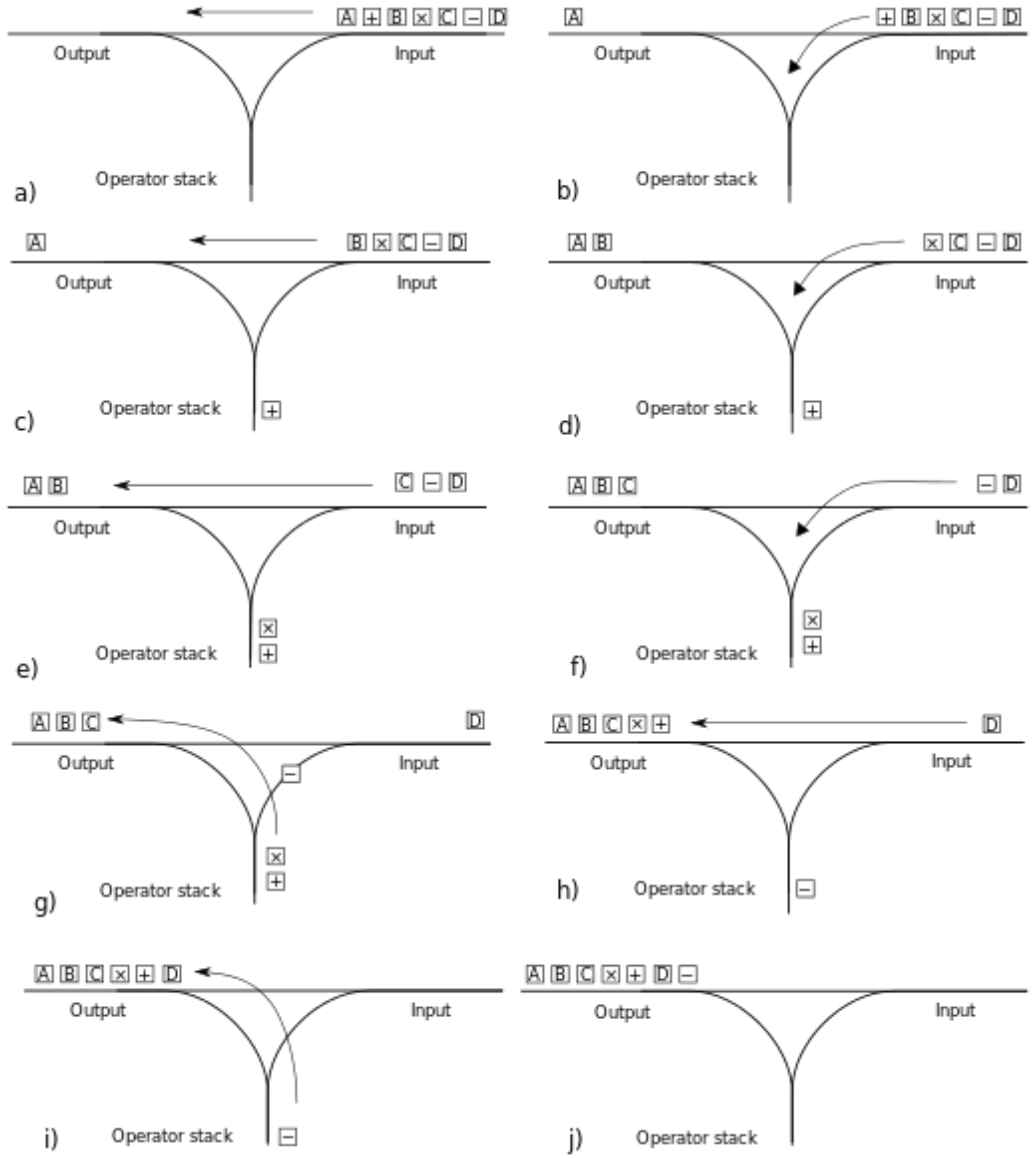
Notazione polacca inversa

Un esempio più complesso che mostra i vantaggi per l'elaboratore:

$(8 + 2 \times 5)/(1 + 3 \times 2 - 4)$

	Stringa rimanente	Istruzione	Stack
1	$8\ 2\ 5\ \times + 1\ 3\ 2\ \times + 4\ - /$	BIPUSH 8	8
2	$2\ 5\ \times + 1\ 3\ 2\ \times + 4\ - /$	BIPUSH 2	8, 2
3	$5\ \times + 1\ 3\ 2\ \times + 4\ - /$	BIPUSH 5	8, 2, 5
4	$\times + 1\ 3\ 2\ \times + 4\ - /$	IMUL	8, 10
5	$+ 1\ 3\ 2\ \times + 4\ - /$	IADD	18
6	$1\ 3\ 2\ \times + 4\ - /$	BIPUSH 1	18, 1
7	$3\ 2\ \times + 4\ - /$	BIPUSH 3	18, 1, 3
8	$2\ \times + 4\ - /$	BIPUSH 2	18, 1, 3, 2
9	$\times + 4\ - /$	IMUL	18, 1, 6
10	$+ 4\ - /$	IADD	18, 7
11	$4\ - /$	BIPUSH 4	18, 7, 4
12	$- /$	ISUB	18, 3
13	$/$	IDIV	**6**

Notazione polacca inversa



Per passare dalla notazione infissa alla notazione polacca inversa si usa l'**algoritmo di scalo di manovra** (*Shunting-yard algorithm*), detto anche *algoritmo di smistamento*, inventato dal noto informatico olandese Edsger Dijkstra.

L'algoritmo consiste nell'analizzare l'espressione e trasferire gli operatori in uno stack apposito, che seleziona se inserire l'operatore o tenerlo in memoria sulla base della priorità dell'operazione