Appunti ISA

Gianluca Mastrolonardo

November 21, 2024

Contents

1 Guida al Documento

Tutto il testo formattato **in questo modo** fa riferimento ad istruzioni ISA

2 Che cos'è l'ISA?

L'*ISA (Istruction Set Architecture),* definisce il linguaggio macchina dell'elaboratore, ovvero tutte le operazioni che un processore può eseguire. Questo è il primo livello che viene sviluppato quando si realizza un architettura. Sebbene in linea di principio il livello ISA sia definito dal modo con cui appare la "macchina" al programmatore in linguaggio macchina, in pratica (quasi) nessuno programma in linguaggio macchina, infatti il codice in linguaggio ISA è per lo più quello prodotto da:

- gli assemblatori;
- compilatori dei linguaggi ad alto livello.

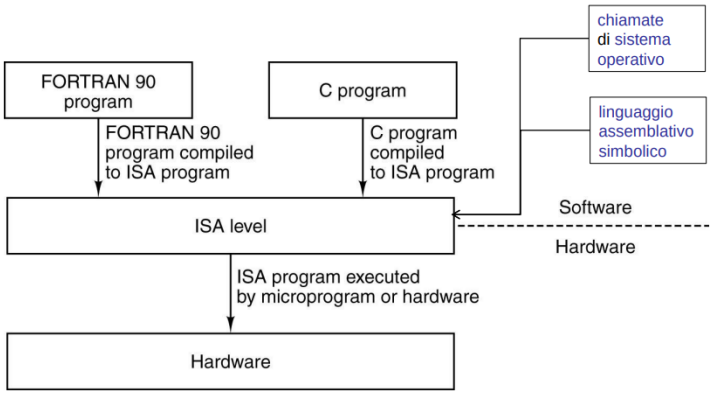
Nella prima parte del corso abbiamo già visto un esempio di linguaggio ISA, ovvero IJVM.

2.1 Differenze tra ISA e Assembler

L'ISA e l'Assembler vengono spesso confusi, ma in realtà sono due concetti molto diversi fra loro. Infatti nell'ISA troviamo tutte le operazioni che il nostro processore è in grado di fornire, invece Assembler è il linguaggio di programmazione che utilizza le operazioni ISA. L'insieme di tutte le operazioni ISA viene anche chiamato **Linguaggio Macchina**. Tutti i programmi scritti in linguaggi ad alto livello devono prima essere compilati, e solo dopo la compilazione l'architettura è pronta per eseguirli. Pertanto possiamo dire che il livello ISA rappresenta l'interfaccia tra Software e Hardware.

Curiosità: Volendo si possono realizzare architetture che eseguono direttamente codice ad alto livello, però questa tecnica è svantaggiosa perché:

- non sfrutta l'incremento di prestazioni garantito dai compilatori;
- penalizzerebbe i programmi scritti in altri linguaggi;
- frammenterebbe l'evoluzione degli elaboratori.



2.2 Backward Compatibility e perché è fondamentale

In un ambiente ideale lo sviluppo dell'ISA dovrebbe sempre essere il giusto compromesso tra:

- compilatori all'avanguardia realizzati dai programmatori;
- hardware all'avanguardia sviluppati dai progettisti.

Sfortunatamente non è così a causa della **Backward Compatibility**, ovvero la possibilità di eseguire programmi realizzati su hardware più vecchi, e quindi con ISA più limitati. Queste operazioni sono difficoltà aggiuntive nello sviluppo di un nuovo ISA.

3 Kernel Mode e User Mode

Una proprietà importante presente nell'ISA è quella di definire più livelli di esecuzione dei programmi. Quelli fondamentali sono due:

- **Kernel Mode**: in questo livello si ha il pieno controllo su tutte le risorse;
- **User Mode**: in questo livello si ha un controllo limitato sulle risorse.

L'utente ha accesso soltanto alla User Mode, dove avrà i suoi programmi, i suoi file, ed in generale le sue risorse. Se per caso l'utente dovesse avviare un programma potenzialmente malevole si attiverebbero le **Trap** (argomento che sarà approfondito dopo qui) , in modo da terminare l'esecuzione del programma e mandare la macchina in stato di allerta. Invece quando l'elaboratore è in Kernel Mode ha la completo accesso a tutte le risorse presenti.

Questa suddivisione è fondamentale per la sicurezza dell'elaboratore, perché così possibili programmi malevoli saranno bloccati, e non hanno accesso a tutte le risorse della macchina perché eseguite in User Mode.

4 Architettura Proprietaria o Open Source?

Come in ogni ambito dell'informatica anche in questo caso bisogna decidere se voler tenere la nostra architettura Closed, ovvero che solo l'azienda produttrice conosce l'architettura della macchina, come per esempio Intel, oppure decidere di mettere a disposizione di tutti lo schema dell'architettura, come per esempio ARM.

5 Caratteristiche di ISA

Le caratteristiche fondamentali dell'ISA sono:

- come è organizzata la memoria (5.1);
- quanti registri ci sono e a che cosa sono dedicati (5.2);
- quali tipi di dati sono implementati sulla macchina (5.3);
- quali istruzioni macchina sono disponibili (5.4);
- qual è il formato delle istruzioni (5.5);
- quali sono le modalità di indirizzamento utilizzabili (5.6).

Ora vediamo tutte queste caratteristiche nel dettaglio.

5.1 Come è organizzata la memoria

La memoria è divisa in celle che hanno indirizzi consecutivi L'organizzazione più comune prevede celle da 8 bit (storicamente motivata dalla codifica ASCII dei caratteri). Ovviamente la memoria è suddivisa in più celle, messe graficamente in colonne. Queste colonne posso essere grandi:

- Bit;
- Byte;
- Word (o parole), ovvero più byte raggruppati assieme.

I formati più comuni sono parole composte da 4 Byte (32 Bit) o da 8 Byte (64 Bit).

I gruppi di parole vengono rinominati blocchi, come fatto anche nelle memorie Cache. Per i blocchi non esistono dimensioni standard, però una *good practice* è quella di formare i blocchi di dimensione 2k Byte. Inoltre, é spesso conveniente che i gruppi di byte che fornamo i blocchi siano "allineati", ovvero che:

- parole di 4 byte siano allineate agli indirizzi multipli di 4 (0,4,8, ...), quindi che terminano con 00;

- parole di 8 byte siano allineate agli indirizzi multipli di 8 (0,8,16,...), quindi che terminano con 000;

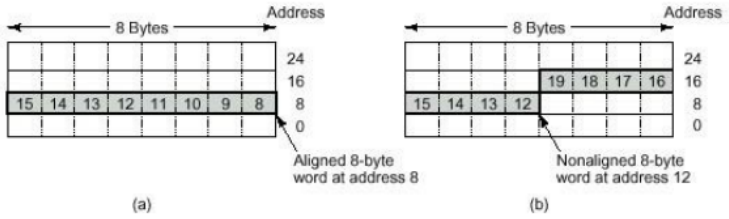


Figure 1: Per esempio le architetture Intel a partire dal Pentium II leggono 8 byte per volta dalla memoria, ad indirizzi multipli di 8, ma il suo ISA non impone allineamento, quindi non è detto che i byte salvati siano effettivamente allineati, portando così uno svantaggio, perché nel caso (a) una sola lettura fornisce l'intera parola, invece nel caso (b) occorrono due letture (e quindi più istruzioni) per comporre le due metà.

5.1.1 Gestione delle dipendenze

Un altra caratteristica fondamentale dell'organizzazione della memoria è quella della semantica delle operazioni di accesso della memoria. Per esempio noi ci aspettiamo un corretto funzionamento di una **LOAD** dopo una **STORE**, per intenderci vogliamo una cosa di questo tipo:

$$x = 4$$
$$x == 4 \rightarrow \text{true}$$

dove in $x = 4$ è avvenuta l'operazione di **LOAD** ed invece in $X == 4$ è avvenuta l'operazione di **STORE**, ed in particolare noi vogliano che ci sia effettivamente salvato il valore 4, per questo motivo è presente il *true* affianco. Nel caso delle architetture dove avviene l'esecuzione ed il ritiro fuori ordine delle istruzioni possono presentarsi comportamenti inattesi. Per risolvere questo problema ci sono tre possibili soluzioni (via via più restrittivi, quindi con sempre meno parallelismo):

- Lasciare al programmatore (al compilatore) la responsabilità di assicurare l'effettiva memorizzazione dei dati tramite una SYNC
- Realizzare in hardware la verifica automatica della presenza di **RAW** (Read After Write) o di **WAR** (Write After Read) nell'esecuzione di **LOAD/STORE**
- Sequenzializzare tutte le **LOAD/STORE** però perdendone completamente il parallelismo

5.1.2 Spazio degli indirizzamenti

L'insieme delle posizioni delle istruzioni e dei dati di un programma definibili tramite le modalità di indirizzamento offerte da ISA è chiamato **Spazio degli indirizzamenti**. La maggior parte delle architetture prevede un solo spazio degli indirizzamento lineare di byte, eventualmente raggruppati in parole di 32 0 64 bit, con indirizzi di k bit possono che indirizzare $2k$ posizioni, da 0 a $2^k - 1$. Oppure ci sono delle architetture che prevedono più spazi di indirizzamenti lineari indipendenti (detti anche segmenti), per le istruzioni e per i dati. Così facendo:

- si può indirizzare spazi più grandi rispetto ai bit a disposizione per definire gli indirizzi: numero di segmento + indirizzo all'interno del segmento
- gli accessi possono essere più controllati specificando permessi di accesso differenti per ciascun segmento
- i segmenti possono crescere/decrescere in modo indipendente

Questo modello di memoria era nato storicamente per poter sfruttare memorie con un numero di locazione più grandi di quelle codificabili con i bit disponibili a livello di ISA (es. dimensione dei registri, campi degli operandi delle istruzioni)

5.1.3 Memorizzazione di dati in più Byte

Nel corso del tempo sono state adottate modalità differenti per memorizzare dati di dimensione superiore al byte, ma le più importanti sono:

- **Little Endian**: byte meno significativo memorizzato nella cella con indirizzo più basso
- **Big Endian**: byte più significativo memorizzato nella cella con indirizzo più basso

Facciamo un esempio con 2 Byte che assieme formano il valore "0000000011111111":

- Little Endian:
  - nell'indirizzo più basso: 11111111;
  - nell'indirizzo più alto: 00000000.
- Big Endian
  - nell'indirizzo più basso: 00000000;
  - nell'indirizzo più alto: 11111111.

Un ulteriore esempio potrebbe essere il numero esadecimale "0x01234567" che compone una parola da 4 Byte:

0x...0011	...
0x...0100	0x67
0x...0101	0x45
0x...0110	0x23
0x...0111	0x01
0x...1000	...

Table 1: Big Endian

0x...0011	...
0x...0100	0x01
0x...0101	0x23
0x...0110	0x45
0x...0111	0x67
0x...1000	...

Table 2: Little Endian

Come possiamo notare nel caso Big Endian presente in Tabella 1 il dato più significativo è posto in cima invece quello meno significativo in fondo; invece nel caso Little Endian della Tabella 2 avviene l'esatto opposto, ovvero che il dato meno significativo è nell'indirizzo più basso, invece il dato più significativo è presente nell'indirizzo più alto.

In IJVM viene adottata una codifica Big Endian.

5.2 Quanti registri ci sono e a che cosa sono dedicati

Tutte le architetture dispongono di registri su cui opera la ALU. La loro funzione è quella di fornire una elevata accessibilità (nel senso di velocità di accesso) ai dati. Normalmente ci sono poche decine di registri (16, 32, ...) alcuni dei quali specializzati (PC, SP, LV, PSW, ...) ed altri di uso generale (es. risultati temporanei, parametri attuali, risultati di funzioni, contatori, ...). Normalmente i registri hanno dimensione multipli di 8 bit (es. 64 bit). Alcuni registri però non sono sempre accessibili, infatti esiste una **distinzione di visibilità** tra i vari registri. Per esempio tutti i registri sono visibili al livello della architettura, ma solo alcuni sono visibili al livello ISA.

5.2.1 Registri fondamentali: i Flags Register

Alcuni registri sono presenti in qualsiasi tipo di architettura, ed uno di questo è il **Flag Register** o **PSW** (Program Status Word), nella quale sono presenti dei Condition Code.
Un Flag Register banale è composta da almeno 4 Condition Code:

- **N**: Posto ad 1 quando il valore restituito dalla ALU **è negativo**;
- **Z**: Posto ad 1 quando il valore restituito dalla ALU **è zero**;
- **V**: Posto ad 1 quando il valore restituito dalla ALU **causa overflow**;
- **C**: Posto ad 1 quando il valore restituito dalla ALU **causa un riporto**;

I condition code sono importanti perché sono asseriti dopo le operazioni aritmetiche o di confronto ed usati dalle istruzioni di salto condizionato. In aggiunta di solito contiene alcuni bit "speciali" come quello per kernel/user mode (visti in Capitolo 3) o per disabilitare gli interrupt (usati dal sistema operativo).

5.3 Quali tipi di dati sono implementati sulla macchina

Una data architettura può gestire in modo nativo diversi tipi di dati, mettendo a disposizione istruzioni specifiche e registri per trattare tali tipi di dati. Tipicamente tutte le ISA gestiscono gli interi (normalmente rappresentati in complemento a 2), ma possono esserci istruzioni specifiche dedicate alle operazioni su:

- naturali (interi senza segno);
- numeri decimali con codifica binaria (BCD - 2 cifre decimali in 1 byte);
- numeri reali, codificati in virgola mobile (Floating Point);
- tipi booleani (cioè singoli bit di un byte o di una parola);
- (stringhe di) caratteri (es. copia di una stringa di caratteri da un punto ad un altro in memoria)
- indirizzi, per cui è previsto l'uso di puntatori

In IJVM i dati sono tutti interi, ed alcuni registri sono dedicati a memorizzare puntatori (SP, PC, LV, CPP)

5.4 Quali istruzioni macchina sono disponibili

Sebbene differiscano tra le diverse architetture, le istruzioni possono essere raggruppate per funzionalità offerte, ad esempio:

- **operazioni di copia di dati**: copia di un dato dalla memoria ad un registro o viceversa
  Esempio: **LOAD, STORE, MOVE**
  Esempio: Per approfondire vai alla Sezione 5.4.1
- **operazioni binarie**: operazioni aritmetico-logiche a due operandi
  Esempio: **ADD, AND, OR**
  Esempio: Per approfondire vai alla Sezione 5.4.2
- **operazioni unarie**:
  Esempio: Per approfondire vai alla Sezione 5.4.3
  - operazioni di traslazione
    Esempio: **SHIFT**

  - semplificazione di operazioni binarie frequenti
    come l'incremento ad 1
    Esempio: **INC**

- **istruzioni di salto ed operazioni di confronto**: utili per controllare il flusso di esecuzione delle istruzioni, infatti il salto condizionato è deciso in base ai bit del Flag Register (vedi Capitolo 5.2.1) impostati in base all'esito delle istruzioni precedenti (es. una istruzione di confronto)
- **istruzioni di I/O**
- **istruzioni di invocazione di procedure**: per gestire la chiamata a procedura (anche ricorsiva) e per garantire la continuazione dell'esecuzione dal punto di chiamata

Vediamo alcune di queste istruzioni più nello specifico.

5.4.1 Operazioni di copia di dati

Come annunciata ad 5.4, le istruzione di trasferimento di dati hanno il compito di copiare il contenuto di una locazione di memoria o di un registro in un'altra locazione o registro, in particolare abbiamo:

- assegnazione del valore di una espressione ad una variabile utilizzando **STORE**
- preparare l'esecuzione delle istruzioni che richiedono gli operandi in registri utilizzando **LOAD**

Solitamente si hanno istruzioni di tipo:

- **LOAD**: memoria → registro
- **STORE**: registro → memoria
- **MOVE**: registro → registro
- non sempre si hanno istruzioni **MOVE** che realizzano trasferimenti memoria → memoria

5.4.2 Operazioni binarie

Come annunciate ad 5.4 sono istruzioni che producono un risultato dalla combinazione di due operandi, in particolare abbiamo:

- operazioni aritmetiche: sono le classiche operazioni aritmetiche come la somma, la sottrazione, la moltiplicazione e la divisione. Non è detto che siano tutte presenti, per esempio in IJVM sono presenti soltanto la somma (**IADD**) e la sottrazione (**ISUB**). Queste operazioni si dividono ancora in:

  - operazioni su interi
  - operazioni in virgola mobile

- operazioni booleane: le macchine supportano due soli tipi di operatori booleani:
  - AND: utile per l'estrazione di una sequenza di bit da una parola o una bit map usando una maschera.
    Esempio: la maschera: 0x00000001 è utile per potere estrarre il bit meno significativo, ad esempio per controllare se il numero è pari o dispari

```
10110111 10111100 11011011 10001011  A
00000000 11111111 00000000 00000000  B (mask)
00000000 10111100 00000000 00000000  A AND B
```

Figure 2: Esempio di maschera sul terzo Byte

  - OR: utile per l'impacchettamento di sequenze di bit in una parola o in una bit map
    *Esempio: vedere l'istruzione **ldc_w3** del microinterprete IJVM, che impacchetta i due byte che compongono l'indice della costante da caricare sullo stack*

```
10110111 10111100 11011011 10001011  A
11111111 11111111 11111111 00000000  B (mask)
10110111 10111100 11011011 00000000  A AND B
00000000 00000000 00000000 01010111  C
10110111 10111100 11011011 01010111  (A AND B) OR C
```

Figure 3: Esempio di maschera e impacchettamento sull'ultimo Byte

5.4.3 Operazioni unarie

Come viste in 5.4 questo gruppo di istruzioni utilizza 1 solo operando, anche se a volte hanno bisogno di specificare una informazione aggiuntiva. Le istruzioni più comune sono

- **SHIFT**: Shift a destra o sinisrta di k byte;
- **CLR A**: $A = 0$;
- **INC1 A**: $A = A + 1$;
- **NEG A**: $A = 0 - A$;
- **NOT A**: $A = A$ XOR 0xFFFFFFFF.

5.4.4 Istruzioni di salto ed operazioni di confronto

Nei programmi i salti condizionati sono necessari per potere alterare il flusso di esecuzione in base al risultato dell'analisi del valore di uno o 2 dati. I confronti vengono fatti

- confrontando una variabile con zero

```
1    x == 0 | x < 0 | etc
2
```

- confrontando una variabile con un altra variabile

```
1    x == y | x < y | etc
2
```

La decisione o meno di eseguire un salto condizionato dipende dal valore di un bit del Flag Register (vedi 5.2.1) della macchina impostato a 0 o a 1 in base all'esito di una istruzione precedente. Si può ipotizzare che i confronti vengano realizzati con una sequenza di 3 bit, dove il bit ad 1 è il risultato del confronto.
N.B: Può esserci un solo bit ad uno.
Esempio: $x = 2 \,|\, y = 4$

$<$	$=$	$>$
1	0	0

In questo caso il primo valore è minore del secondo, quindi avremo ad 1 il bit che ci rappresenta il $<$ e gli altri due bit a zero. Se la nostra operazione fosse stata $x < y$ l'esito sarebbe stato TRUE, quindi sarebbe avvenuto il salto, invece se fosse stata $x = y$ oppure $x > y$ avremmo avuto come esito FALSE, quindi non sarebbe avvenuto nessun salto.

Le istruzioni di salto condizionato possono avere un bit aggiuntivo per indicare se il salto avviene quando il bit da testare è 1 o 0. Se la condizione non è soddisfatta sui processori ARM l'esecuzione equivale ad una **NOP**

5.5 Qual è il formato delle istruzioni

5.5.1 Dimensione delle istruzioni

La codifica binaria delle istruzioni è formata da un codice operativo a cui si possono aggiungere uno o più campi per indicare da dove prelevare i dati e dove memorizzare i risultati (gli operandi di sorgente o destinazione). Il problema generale della indicazione del luogo da cui prelevare ed in cui memorizzare i risultati è chiamato indirizzamento.

- zero indirizzi
- un indirizzo
- due indirizzi
- tre indirizzi

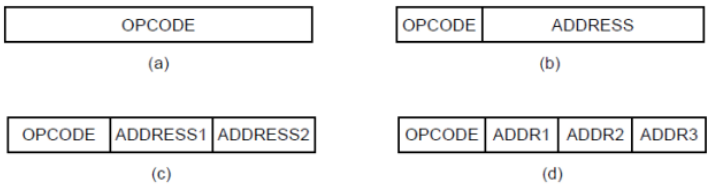


Figure 4: Rappresentazione della lista presente in 5.5.1

Un problema importante nella definizione del formato delle istruzioni è la scelta della loro lunghezza (quanti bit sono necessari per specificarle), infatti occorre fare un compromesso tra espressività e velocità di prelievo, e fare scelte *future proof*, avendo così la possibilità di estendere l'insieme delle istruzioni. La lunghezza può essere fissa oppure può variare in funzione del numero degli operandi.

In IJVM la lunghezza delle istruzioni è variabile.

5.5.2 Dimensione del Codice Operativo

Le istruzioni possono occupare:

- una parola;
- meno di una parola;
- più parole

questo definisce la dimensione del **codice operativo**.

L'importante è mantenere dimensioni che garantiscono l'allineamento delle istruzioni al byte o alla parola.
In IJVM il codice operativo è a lunghezza fissa di 1 Byte

Codice operativo ad espansione: Un esempio di istruzioni a lunghezza fissa (Indirizzi a 4 bit):

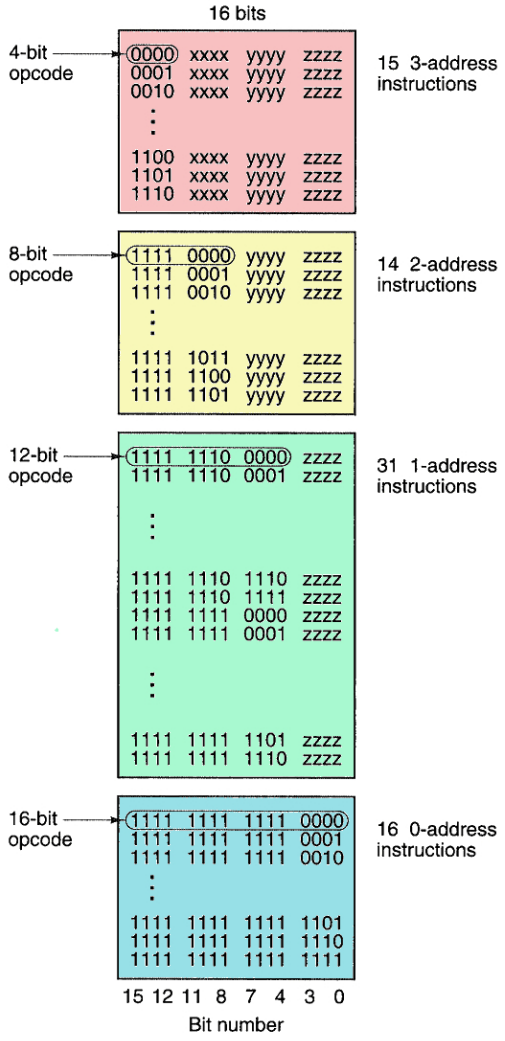


Figure 5: Schema del codice operativo ad espansione

Possiamo notare come i primi 4, 8, 12 o 16 bit rappresentano il valore dell'operation code, invece i restanti (se ci sono) rappresentano gli indirizzi sulla quale deve operare l'istruzione. Se la sequenza di bit dell'operation code è lunga quanto la lunghezza dell'istruzione (in questo caso 4 bit) significa che l'istruzione non accede alla memoria, come per esempio **IADD**, **IAND**, **ISUB**.

5.6 Quali sono le modalità di indirizzamento utilizzabili

In un istruzione possono essere presenti 0,1,2 o 3 operandi, che esprimono la sorgente dei dati e la destinazione del risultato espressa dal codice operativo.

Esempio: **ADD R1, R2**

Istruzione	Semantica	Destinazione	Sorgente
ADD R1, R2	R1 = R1 + R2	R1	R1, R2

Ecco la lista dei vari tipi di indirizzamento:

- indirizzamento immediato (vedi 5.6.1)
- indirizzamento diretto (vedi 5.6.2)
- indirizzamento di registro (vedi 5.6.3)
- indirizzamento indiretto attraverso registro (vedi 5.6.4)
- indirizzamento indicizzato (vedi 5.6.5)
- indirizzamento base-indice (o indicizzato esteso) (vedi 5.6.6)
- indirizzamento relativo allo stack (vedi 5.6.7)

Per le istruzioni di salto si può avere solamente:

- indirizzamento diretto
- indirizzamento indicizzato (relativo ad un registro, per es. PC)

5.6.1 Indirizzamento immediato

Si ha quando **l'operando è contenuto nell'istruzione**.
Esempi:
BIPUSH 10 //codice IJVM per fare il Push di un valore in cima allo stack.
ADDI R1, R2, 20 //semantica: R1 ← R2 + 20

Non avviene nessun accesso aggiuntivo nelle memorie, ed è molto utile per costanti esprimibili da pochi bit

5.6.2 Indirizzamento diretto

Si ha quando l'operando è all'indirizzo di memoria specificato nell'istruzione
Esempio:
LOAD R1 → **LOAD R1 0x102930** //semantica: R1 ← m[0x102930]

Nel esempio presente sopra dobbiamo porre l'attenzione su un paio di cose. Quando noi vogliamo utilizzare un indirizzamento diretto non dobbiamo esplicitamente specificare l'indirizzo di memoria ma scriveremo semplicemente **NOME ISTRUZIONE VARIABILE**, quindi nel caso dell'esempio avremo **LOAD R1**, poi sarà l'ISA che in automatico andrà a prendere dalla memoria il contenuto presente nella variabile ed, infine, verrà posto in cima allo stack.

Quindi in sintesi possiamo dire che l'indirizzamento diretto accede ad una locazione fissa di memoria ed è utile per accedere alle variabili globali di un programma.

5.6.3 Indirizzamento di registro

Si ha quando l'operando è contenuto in un registro,in questo modo il dato è più efficiente da accedere rispetto a dover accedere alle locazioni di memoria.
Esempi:
ADD R1, **R2**, **R3** //Semantica: R1 ← R2 + R3
ADD R1, **R2** //Semantica: R1 ← R1 + R2

Questi tipi di indirizzamenti sono utili per gestire variabili utilizzate frequentemente o per memorizzare risultati intermedi. Questa tecnica è molto utilizzata nelle architetture ARM. Da notare la differenza con **ADDI** vista nell' Indirizzamento immediato (5.6.1) dove veniva sommato da un registro un valore specifico, ma non avveniva la somma di due registri come in **ADD**.

5.6.4 Indirizzamento indiretto attraverso registro

Si ha quando l'operando in esame proviene o è destinato alla memoria, ma il suo indirizzo non è incorporato direttamente nell'istruzione, ma viene utilizzato un puntatore
Esempio:
LOAD R1, **(R2)** //semantica: R1 ← m[R2]

Utilizzando questa tecnica l'istruzione non deve includere un indirizzo "lungo" di memoria, ma solo un indirizzo di registro. Ad ogni esecuzione dell'istruzione R2 può puntare ad una locazione differente (es: l'indirizzo di un elemento di un array all'interno di un ciclo).
Esempio: Prendiamo come esempio questo codice C per la somma degli elementi di un Array:

```c
int r1 = 0;
unsigned char *r2 = (unsigned char*)a;
unsigned char r3 = (unsigned char*)a+dim*4;
do {
    r1 += (int)*r2;
    r2 = r2+size;
} while (r2<r3);
```

```
        MOV R1,#0          // Accumula la somma in R1, inizialmente è 0
        MOV R2,#A          // R2 = indirizzo dell'array A (256 elementi)
        ADDI R3, R2, #1024; // R3 = indirizzo se la prima parola è oltre A
LOOP:   ADD R1,(R2)        // Somma di R1 + valore a cui punta R2
        ADDI R2, R2, #4    // Somma di R2 + 4
        CMP R2,R3          // Confronto R2 ed R3 e setto il condition code
        BLT LOOP           // se R2 <R3 (quindi condition code a zero)
                           // vai a LOOP
```

Come possiamo notare l'istruzione in rosso è un indirizzamento indiretto attraverso registro.

Nota: R3 è utilizzato per far finire il ciclo while

5.6.5 Indirizzamento indicizzato

Si ha quando l'indirizzo di memoria dove si trova l'operando è ottenuto dalla somma del contenuto di un registro (esplicito o implicito) e da uno spiazzamento costante indicato nell'istruzione (o viceversa)
Esempi:
STORE R4, #A(R2) //Semantica: m[#A+R2] ← R4
ILOAD X //Semantica: m[SP++] = m[x+LV] IJVM (cioè: ILOAD (SP), x(LV), con SP e LV impliciti)

Questo tipo di indirizzamento è utile per all'interno di strutture dati complesse (array, record), o per realizzare macchine a stack (es. IJVM)

5.6.6 Indirizzamento base-indice (o Indicizzato esteso)

Si ha quando l'indirizzo di memoria dove si trova l'operando è ottenuto dalla somma di due registri e, opzionalmente, da un offset presente nell'istruzione.
Esempio:
LOAD R4, (R2+R1) //Semantica: R4 ← m[R1+R2]

Utile per gestire gli array: i due registri memorizzano l'indirizzo di base e l'offset (puoi vederlo come un indice).

Differenza tra indiretto tramite registro, indicizzato e base-indice: Sommare gli elementi corrispondenti di due array (A e B) di int (a 4 byte) salvando il risultato nel corrispondente elemento di un terzo array (C). Gli indirizzi contenuti nei registri-puntatore sono indirizzi di byte.

(a) indiretto tramite registro, (b) indicizzato, (c) base-indice

	(a)		(b)		(c)
	MOV R0,#A		MOV R0,#0		MOV R0,#0
	MOV R1,#B				MOV R2,#A
	MOV R2,#C				MOV R3,#B
	ADDI R8,R0,#MAX				MOV R4,#C
LOOP:	LOAD R3,(R0)	*LOOP:*	LOAD R3,#A(R0)	*LOOP:*	LOAD R5, (R2+R0)
	LOAD R4,(R1)		LOAD R4,#B(R0)		LOAD R6, (R3+R0)
	ADD R7, R4, R3		ADD R7, R4, R3		ADD R7, R6, R5
	STORE R7,(R2)		STORE R7,#C(R0)		STORE R7,(R4+R0)
	ADDI R0, R0, #4		ADDI R0, R0, #4		ADDI R0, R0, #4
	ADDI R1, R1,#4				
	ADDI R2, R2,#4				
	CMP R0, R8		CMPI R0,#MAX		CMPI R0,#MAX
	BLT *LOOP*		BLT *LOOP*		BLT *LOOP*

*#MAX=dimTotale array in byte (=4*numElem di tipo int)*
#A,#B,#C Indirizzi di partenza degli array A,B e C

CMP a,b confronta a e b
BLT lab: salta a lab se a < b

5.6.7 Indirizzamento relativo allo stack

Si ha quando gli indirizzi degli operandi sono impliciti, ovvero quando gli operandi sono già sullo stack. Sono tutte istruzioni con 0 operandi. Le espressioni in notazione polacca postfissa sono facilmente e direttamente eseguibili da una architettura a stack.
Esempio: Notazione Polacca inversa:

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

Adesso che abbiamo convertito la nostra espressione in notazione polacca inversa possiamo vedere come funziona l'indirizzamento sullo stack.

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

5.6.8 Come funzionano gli Indirizzamenti nei salti

Si possono applicare gli stessi meccanismi adottati per l'indirizzamento dei dati per modificare il PC, ad esempio:

- indirizzamento diretto: l'indirizzo destinazione è riportato all'interno dell'istruzione.
  GOTO 0x102930 // Semantica: PC = 0x102930

- indirizzamento di registro: l'indirizzo di destinazione è calcolato, scritto in un registro ed usato per il salto (similmente: l'indirizzamento di registro indiretto). É rischioso in quanto può generare bachi di difficile soluzione. Questa soluzione viene principalmente usata per gestire gli indirizzo di ritorno delle procedure.
  GOTO R2 // Semantica: PC = R2 (es. R2 contiene l'indirizzo di ritorno

dalla chiamata di procedura)
GOTO (R2) // Semantica: PC = m[R2]

- indirizzamento relativo: l'indirizzo destinazione è riportato come offset rispetto al valore corrente del PC (con il registro PC implicito).
GOTO offset(PC) // Semantica: PC = PC+offset
Puoi prendere come esempio IJVM, dove il PC è implicito.

6 Architettura Load/Store

6.1 Cos'è un architettura Load/Store

Un architettura Load/Store è un tipo di architettura solo le istruzioni di caricamento (LOAD) e salvataggio (STORE) possono accedere alla memoria principale.

In un'architettura Load/Store, i dati vengono caricati dalla memoria principale nei registri interni del processore utilizzando istruzioni di caricamento. Successivamente, le operazioni vengono eseguite sui dati nei registri e i risultati vengono salvati nella memoria principale utilizzando istruzioni di salvataggio. Questo approccio consente una maggiore flessibilità nella gestione dei dati, consentendo al processore di lavorare sui dati interni in modo più efficiente. L'architettura Load/Store più diffusa è quella ARM.

6.2 Tipi di indirizzamenti utilizzati

Gli indirizzamenti sono composti da:

- un operando che indica un registro;
- un' altro operando che indica l'indirizzo di memoria coinvolto nel trasferimento del dato

I tipi di indirizzamenti utilizzati sono quattro:

- indirizzamento diretto (visto in 5.6.2)
LOAD R1, **0x102930** // Semantica: R1 ← m[0x102930]
STORE R1, **0x102930** // Semantica: m[0x102930] ← R1
- indirizzamento indiretto (visto in 5.6.4)
LOAD R1, **(R2)** // Semantica: R1 ← m[R2]
STORE R1, **(R2)** // Semantica: m[R2] ← R1
- indirizzamento indicizzato, dove A è una costante (visto in 5.6.5)
LOAD R1, **#A(R2)** // Semantica: R1 ← m[R2+A]
STORE R1, **#A(R2)** // Semantica: m[R2+A] ← R1

- indirizzamento base-indice, dove A è una costante (visto in 5.6.6)
  LOAD R1, **(R2, R3)** // Semantica: R1 ← m[R2+R3]
  STORE R1, **(R2, R3)** // Semantica: m[R2+R3] ← R1

Appreso questo concetto è immediato capire che tutte le altre operazioni non usano direttamente la memoria, ma utilizzano solo:

- indirizzamenti a registro
- indirizzamenti immediato

Di solito le istruzioni hanno due o tre operandi, tra cui:

- operando destinazione: sfruttando un indirizzamento a registro
- operando sorgente: sfruttando un indirizzamento a registro o immediato

  Esempi:
  ADD RD, **RN**, **RM** // Semantica: RD ← RN + RM
  ADDI RD, **RN**, **#n** // Semantica: RD ← RN + n; dove n è un numero
  MOVE RD, **RN**, **RM** // Semantica: RD ← RN
  MOVE RD, **RN**, **#n** // Semantica: RD ← n; dove n è un numero