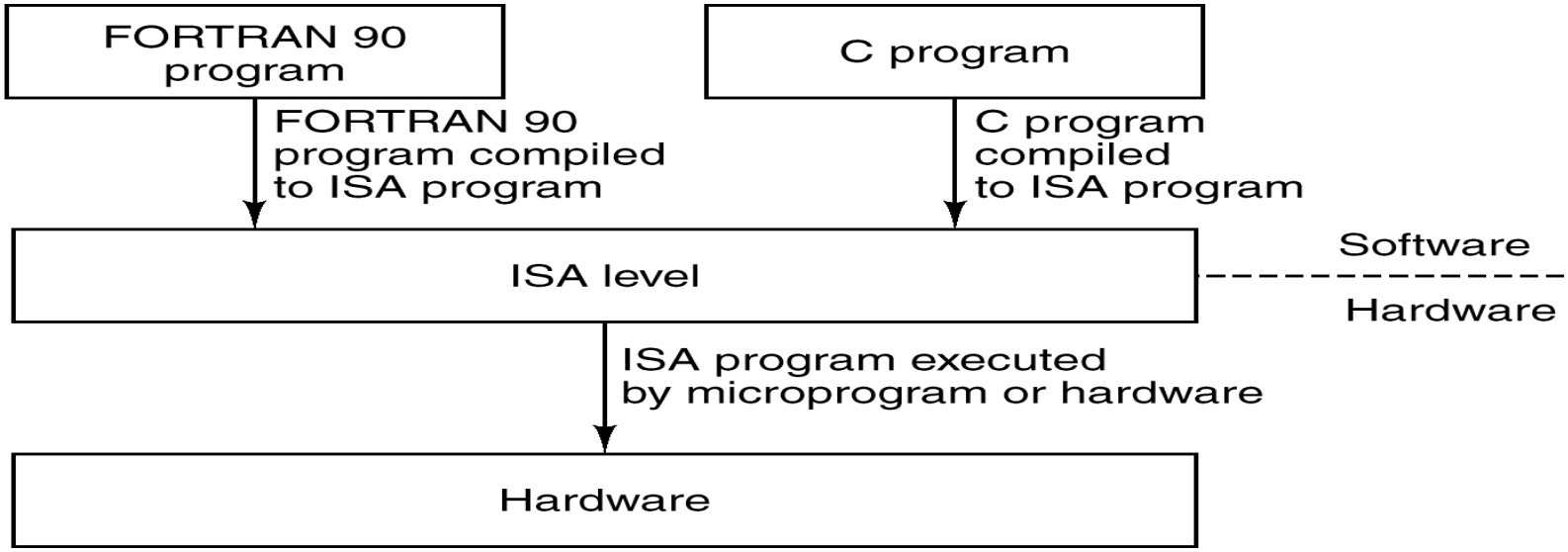
Calcolatori Elettronici

Parte VII: il Livello delle Istruzioni Macchina

Prof. Riccardo Torlone

Universita Roma Tre

Instruction Set Architecture

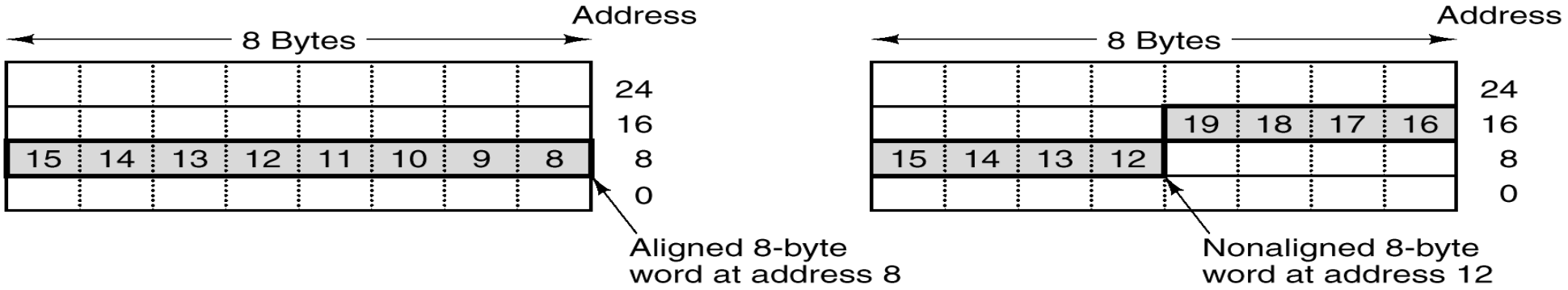


- Il livello ISA è l'interfaccia tra HW e SW
- È il livello più basso a cui il processore è "programmabile"
- Criteri di scelta:
  - Semplicità di implementazione
  - Efficienza della microarchitettura
  - Semplicità di generazione del codice
  - Compatibilità con il passato

Definizione del livello ISA

- Costituisce il riferimento per coloro che scrivono i compilatori (oppure che programmano in assembler)
- In genere è definita in documenti formali:
  - ARM-7 (diverse implementazioni)
  - IA-32 (o x86, implementata solo da Intel)
- Caratteristiche fondamentali:
  - Memoria: organizzazione
  - Registri: quali registri sono visibili al livello ISA e quali funzioni hanno
  - Indirizzamento: come le istruzioni fanno riferimento ai loro operandi
  - Istruzioni: repertorio delle istruzioni interpretate dal livello ISA
  - Modalità di funzionamento:
    - tipicamente *user mode* e *kernel mode*

Modello della Memoria

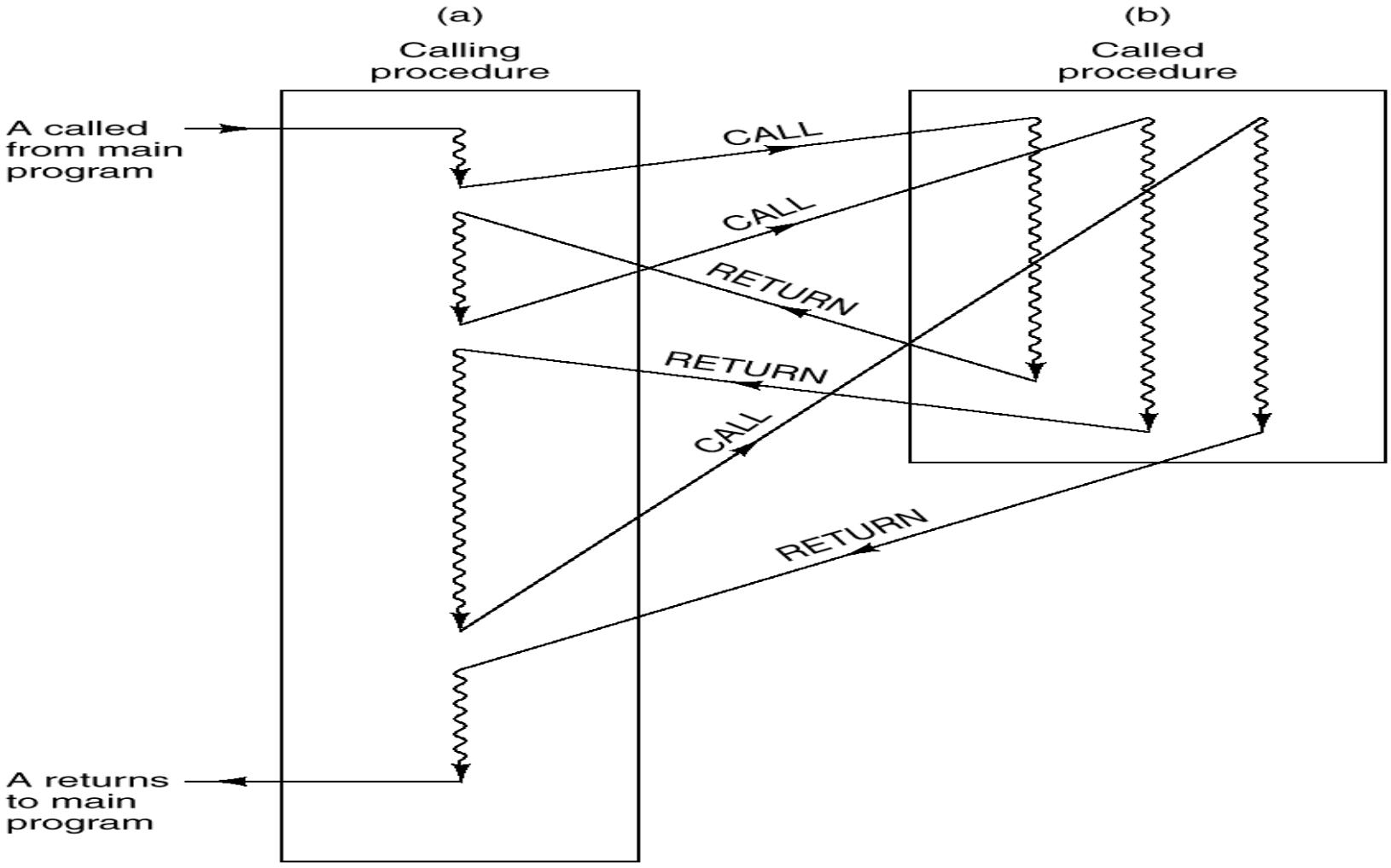


- Celle elementari: byte di 8 bit
- Word: di 4, 8 o + byte
- Allineamento delle word: non cominciano da indirizzi qualsiasi
- Ordinamento dei caratteri: finale grande o finale piccolo
- Spazio degli indirizzi:
  - Tipicamente 2^{32} o 2^{64}
  - A volte separato istruzioni e dati
- Accesso alla memoria: non sempre strettamente sequenziale

Registri

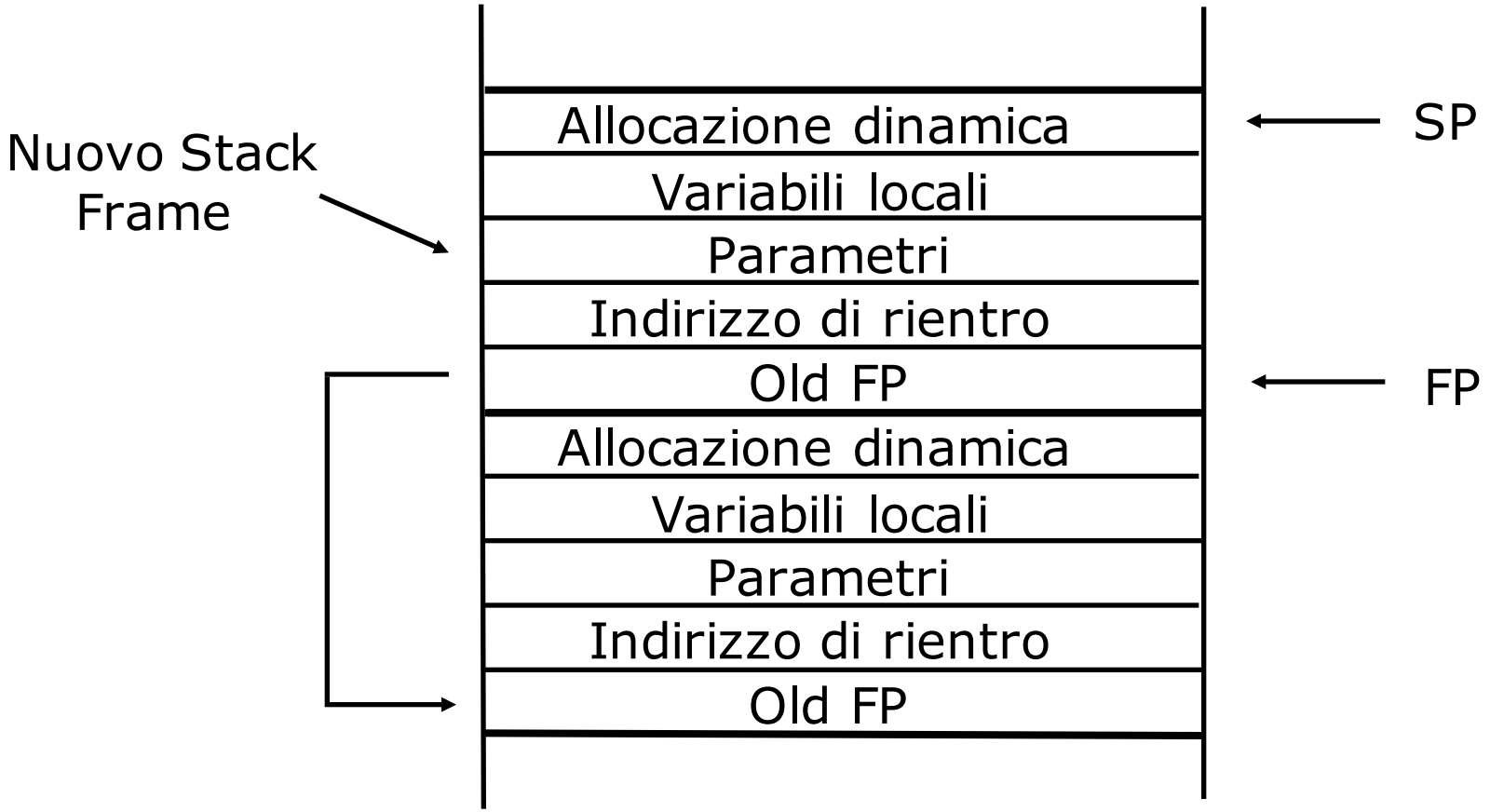
- Tutti i registri visibili al livello ISA sono visibili al livello della micro-architettura
- Non è vero il viceversa
- Registri *general-purpose*:
  - servono per risultati intermedi e per dati di uso molto frequente
- Registri *special-purpose*:
  - hanno ciascuno una funzione specifica (es. Stack pointer)
- Registri visibili solo in kernel mode
- **PSW** (Program Status Word): registro che contiene una serie di flag relativi al risultato della ALU (ed altro)
  - N: risultato Negativo
  - Z: risultato Zero
  - V: oVerflow
  - C: Carry
  - A: Auxiliary carry
  - P: Parità del risultato

Invocazione di funzione

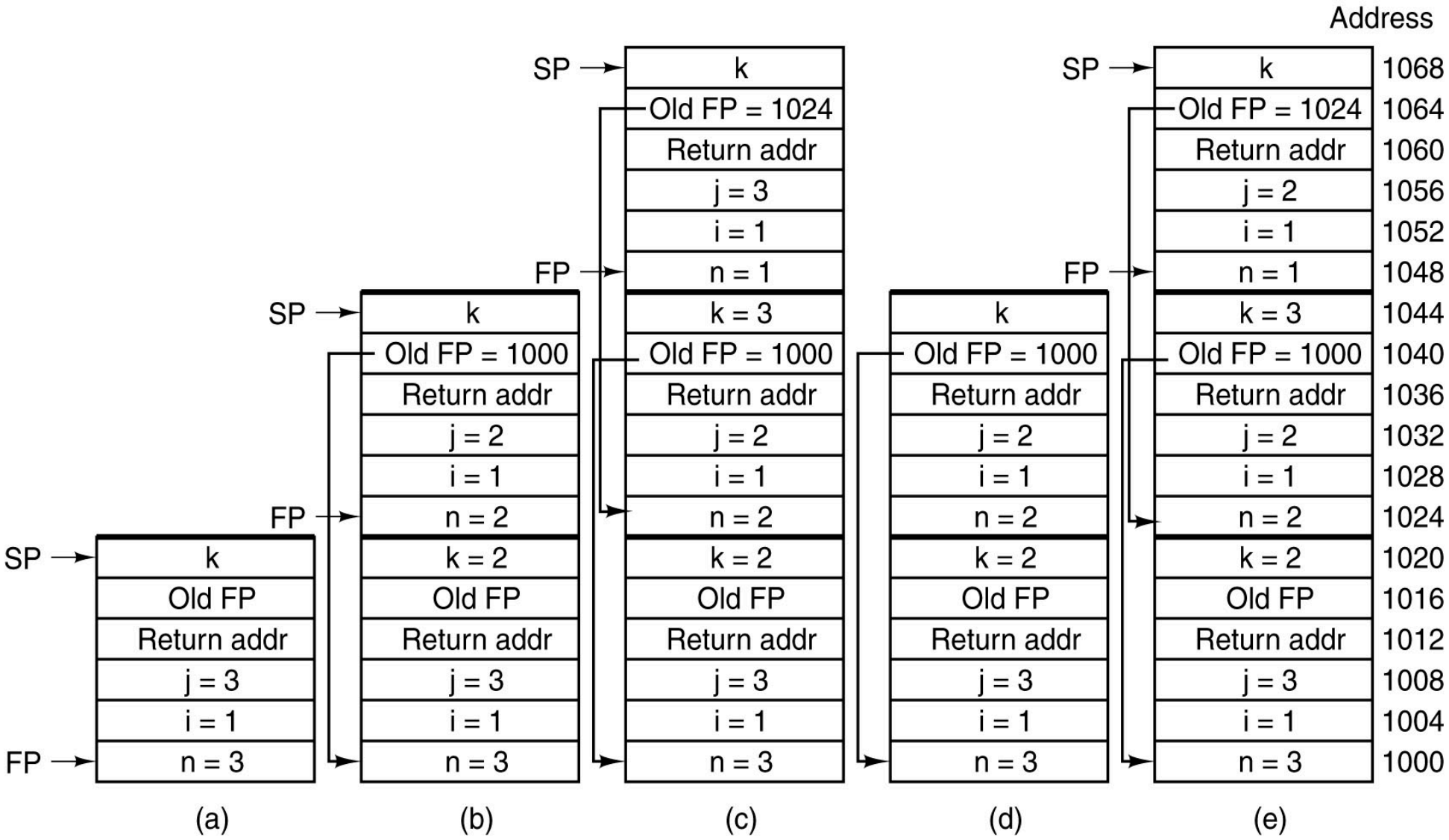


Invocazione di funzione

Per ciascuna invocazione viene allocato in una porzione della memoria organizzata a pila (stack) una nuova area detta *stack frame* (record di attivazione)



Struttura della Stack Frame



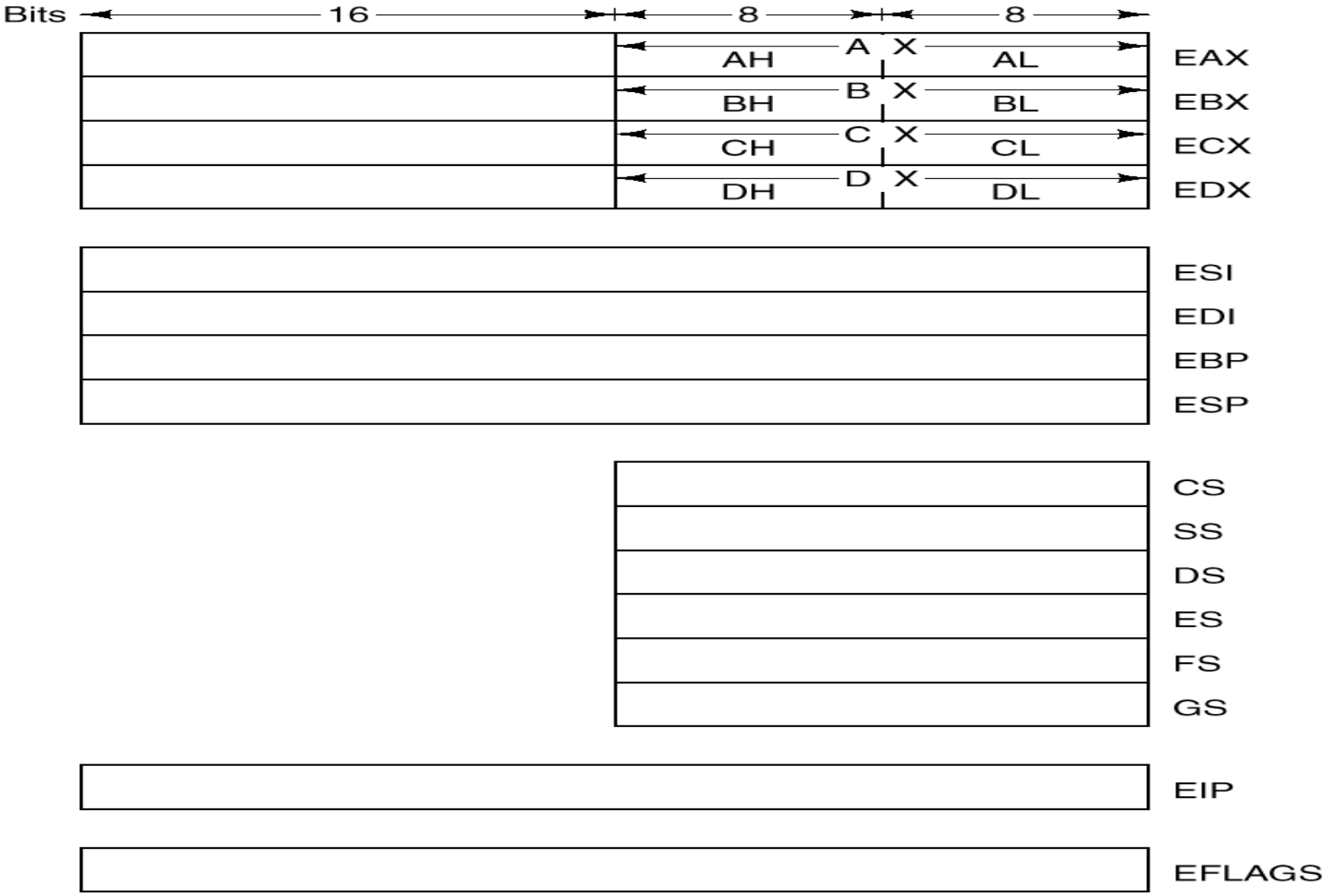
Struttura della Stack Frame

- Lo stack frame contiene:
  - I parametri in entrata e in uscita
  - Le variabili locali
  - L'indirizzo di rientro
  - Un puntatore allo stack frame del chiamante (se esiste)
- Lo stack pointer SP punta alla cima dello stack
- Il frame pointer FP punta alla base del frame in cima alla pila
- L'accesso ai parametri e alle variabili locali e avviene tramite offset da FP
- La posizione rispetto a FP è nota a tempo di compilazione e costante
- La posizione rispetto a SP non è costante (possono essere fatte PUSH e POP durante l'esecuzione)
- All'atto del rientro lo stack frame viene deallocato

Il Livello ISA del Core i7

- Fortemente influenzato dal vincolo di compatibilità all'indietro
- L'architettura è praticamente la IA-32 introdotta con il 80386
- Recente estensione a 64 bit (x86-64)
- Modi di funzionamento:
  - *Real mode*: si comporta come l'8088
  - *Virtual 8086 mode*: come l'8086 ma in ambiente isolato (finestra DOS sotto Windows)
  - *Protected mode*: si comporta al pieno delle funzionalità
- 4 livelli di privilegio: 0=kernel, 3=user
- Spazio di memoria: 16K segmenti di 4GB
- Molti sistemi operativi usano un solo segmento
- È possibile indirizzare il singolo byte
- Word di 4 byte a finale piccolo

Registri ufficiali del Core i7



Registri del Core i7 (2)

- EAX, EBX, ECX, EDX: registri (quasi) general-purpose a 32, 16 o 8 bit:
  - EAX: accumulatore
  - EBX: puntatori a memoria
  - ECX: controllo cicli
  - EDX: estende EAX a 64 bit nelle divisioni e moltiplicazioni
- ESI, EDI: puntatori a memoria (tip. gestione array)
- EBP: *Base Pointer*, punta alla base dello *stack-frame*
- ESP: *Stack Pointer,* punta alla cima dello stack
- EIP: *Instruction Pointer*
- CS..GS: *Registri di Segmento*, ereditati dal passato, puntano a 6 segmenti
- EFLAGS: *Program Status Word*

Il Livello ISA dell'OMAP4430

- L'OMAP4430 è equipaggiato con ARM Cortex A9
- L'ARM Cortex A9 implementa l'architettura ARM v.7
- Architettura a 32 bit, big-endian (verificabile durante l'avvio)
- Spazio indirizzabile 2^{32} bytes (ARM v.8 a 64 bit: maggiore spazio di indirizzamento).
- Architettura load/store
  - solo le LOAD e le STORE fanno riferimento a memoria
- 2 gruppi di registri
  - 16 registri general-purpose a 32 bit (R0-R15)
    - Svolgono quasi sempre lo stesso ruolo
  - 16 registri floating-point a 32 bit
    - Opzionali (deve esistere il supporto VFP)
    - Usati a coppie per precisione doppia

Registri di ARM 7

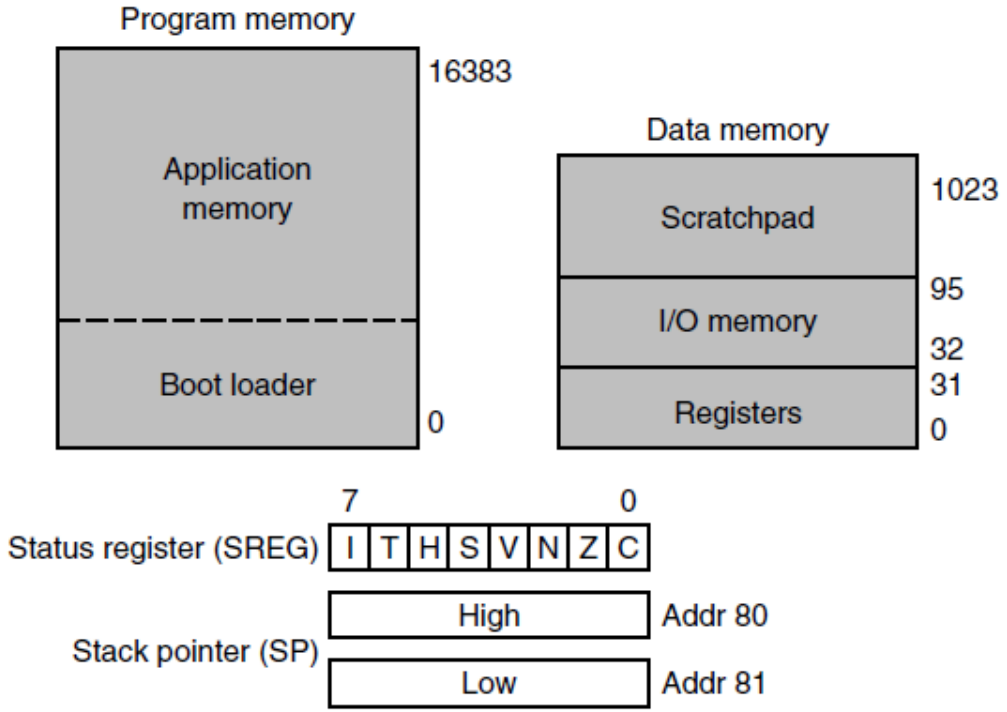
Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

- I registri su interi svolgono (quasi) sempre la stessa funzione
- I registri FP:
  - sono opzionali (deve esistere il supporto VFP)
  - Usati a coppie per precisione doppia
- L'intraprocedure register consente l'indirizzamento a 32 bit per le invocazioni di funzione
- Il program conter (R15) può essere modificato dalla CPU

Il Livello ISA dell'ATmega168

- Target: applicazioni embedded low-end
- Implementa l'architettura AVR
- Una sola modalità e nessuna protezione
- Memoria indirizzabile in spazi separati:
  - dati (SRAM interna)
  - istruzioni (Memoria Flash interna)
  - memorie esterne fino a 64KB+64KB
- Varie versioni:
  - ATmega48: 512B+4KB
  - ATmega88: 1KB+8KB
  - ATmega168:
    - 1KB(dati)+16KB(istruzioni)
    - memoria istruzioni divisa in 2 sezioni
      - Boot loader (le istr. possono aggiornare la Flash)
      - Application (applicazioni di terze parti con firma digitale)

Organizzazione memoria nell'ATmega168



- 32 registri ufficiali da 8 bit (R0-R31)
- Unico spazio di memoria per i dati con:
  - Registri ufficiali
  - Registri per I/O
  - Memoria RAM

Tipi di dati

Core i7

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	× (64-bit)
Unsigned integer	×	×	×	× (64-bit)
Binary coded decimal integer	×			
Floating point			×	×

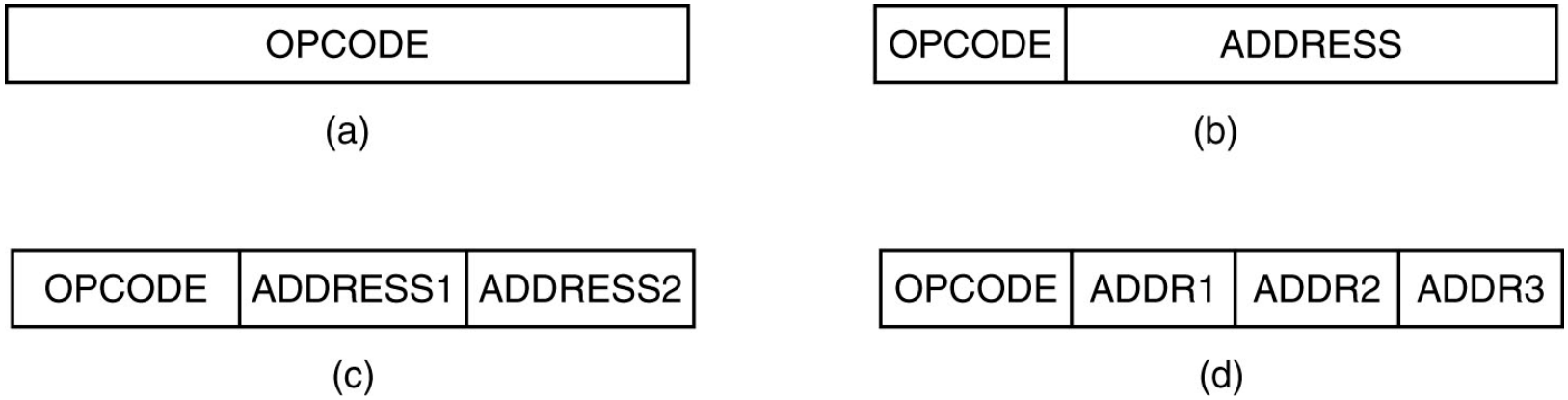
OMAP4430 ARM

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	
Unsigned integer	×	×	×	
Binary coded decimal integer				
Floating point			×	×

ATmega168 AVR

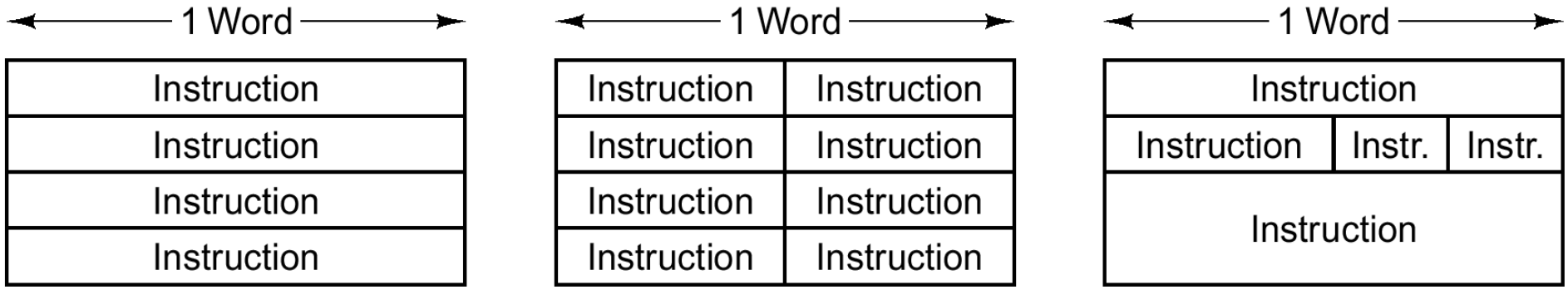
Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×			
Unsigned integer	×	×		
Binary coded decimal integer				
Floating point				

Formato delle Istruzioni



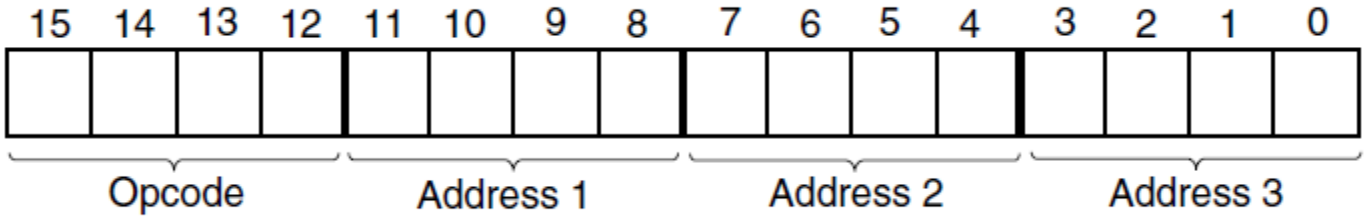
- Lunghezza *fissa* o *variabile*
- Istruzioni corte sono preferibili perché riducono la banda di memoria necessaria al fetch delle istruzioni
- Se l'opcode è di k bit si hanno al massimo 2^k istruzioni diverse
- I campi indirizzi possono fare riferimento sia alla memoria che ai registri
- Complesse *modalità di indirizzamento* permettono di indirizzare gli operandi anche con pochi bit

Lunghezza delle istruzioni e delle word



- La lunghezza fissa semplifica il fetch delle istruzioni
- Una word può contenere una o più istruzioni
- Se le istruzioni hanno formato variabile non c'è relazione tra word e istruzioni

Espansione dei Codici Operativi

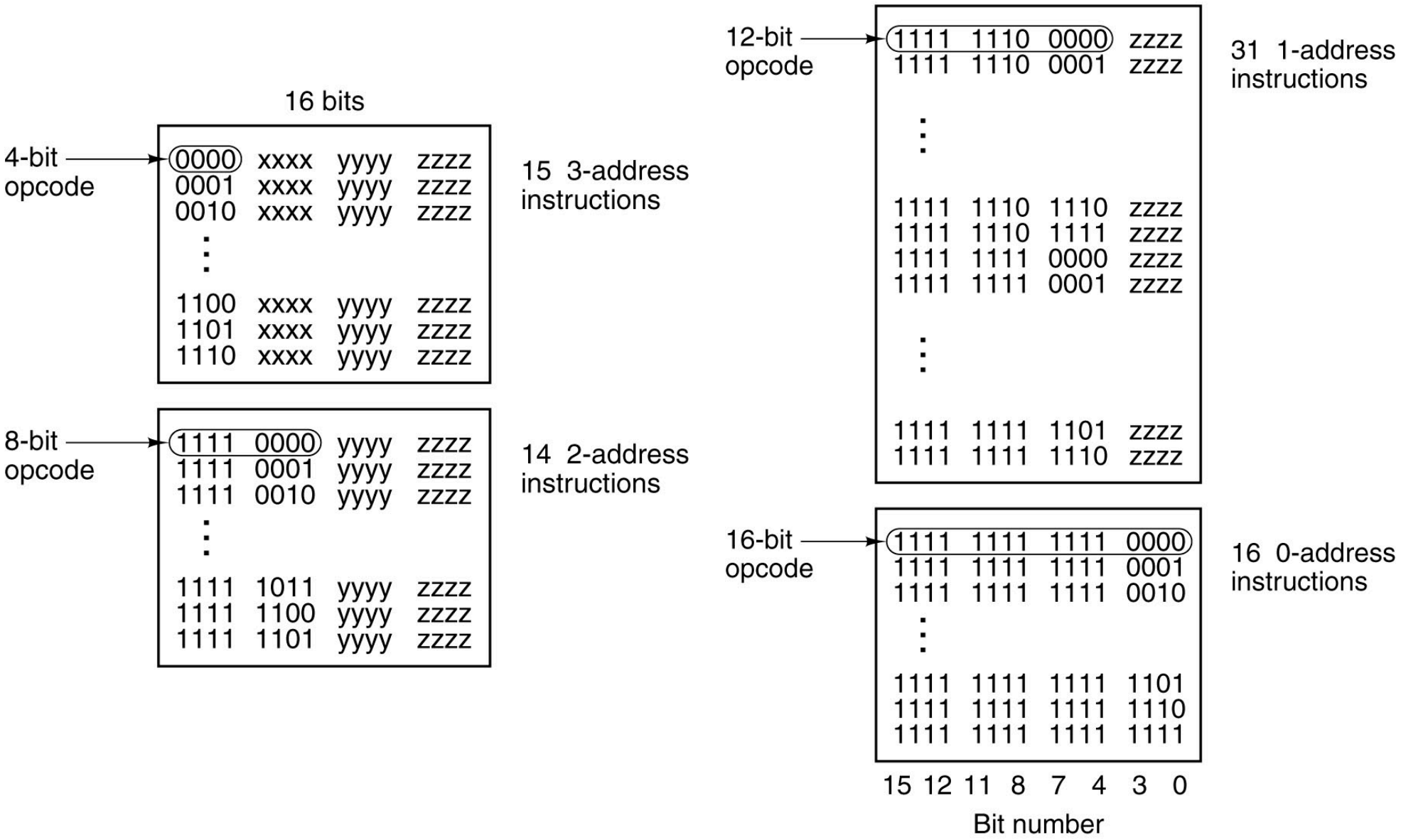


- Il numero di bit dedicati all'opcode non è costante
- Un primo tipo di istruzioni ha un codice operativo corto
- Alcuni valori del codice segnalano che anche i bit successivi ne fanno parte
- Alcuni dei campi indirizzo vengono dedicati all'espansione del codice:

ES

  - Istruzioni a 3 indirizzi: opcode 4 bit
  - Istruzioni a 2 indirizzi: opcode 8 bit
  - Istruzioni a 1 indirizzi: opcode 12 bit
  - Istruzioni a 0 indirizzi: opcode 16 bit

Espansione dei Codici Operativi



Espansione dei codici operativi

Si supponga di voler progettare un linguaggio macchina per una architettura con istruzioni a 16 bit, in cui occorrono i 4 bit per indirizzare ciascun operando, e in cui si vuole utilizzare la tecnica di espansione dei codici operativi

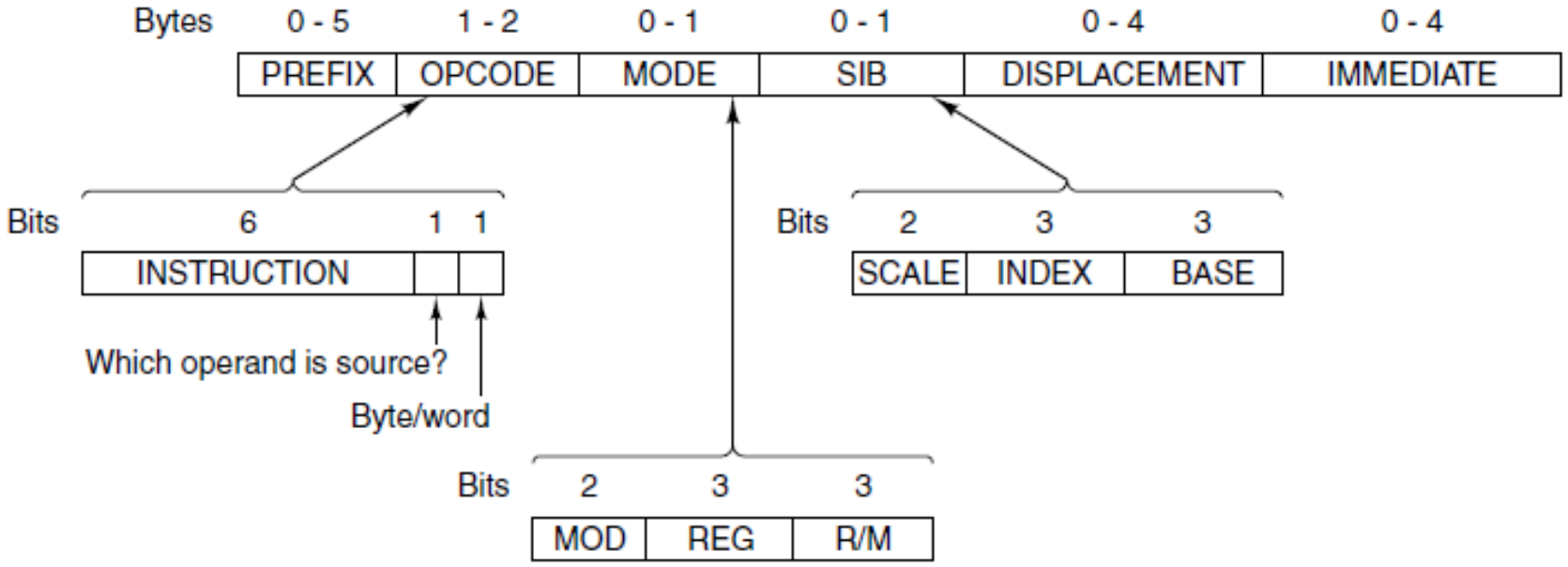
- volendo avere 18 istruzioni a due indirizzi, 8 ad un indirizzo e nessuna a zero indirizzi, quante istruzioni a tre e' possibile avere al massimo?
- mostrare la corrispondente organizzazione dei codici operativi;
- cosa cambia se si vogliono avere 80 istruzioni a un indirizzo invece di 8?

Espansione codici operativi II

Si consideri un'architettura con istruzioni a 16 bit in cui sono necessari 4 bit per indirizzare ciascun operando. Adottando la tecnica dell'espansione dei codici operativi e volendo avere 12 istruzioni a due indirizzi, 120 istruzioni a un indirizzo e 7 a zero indirizzi:

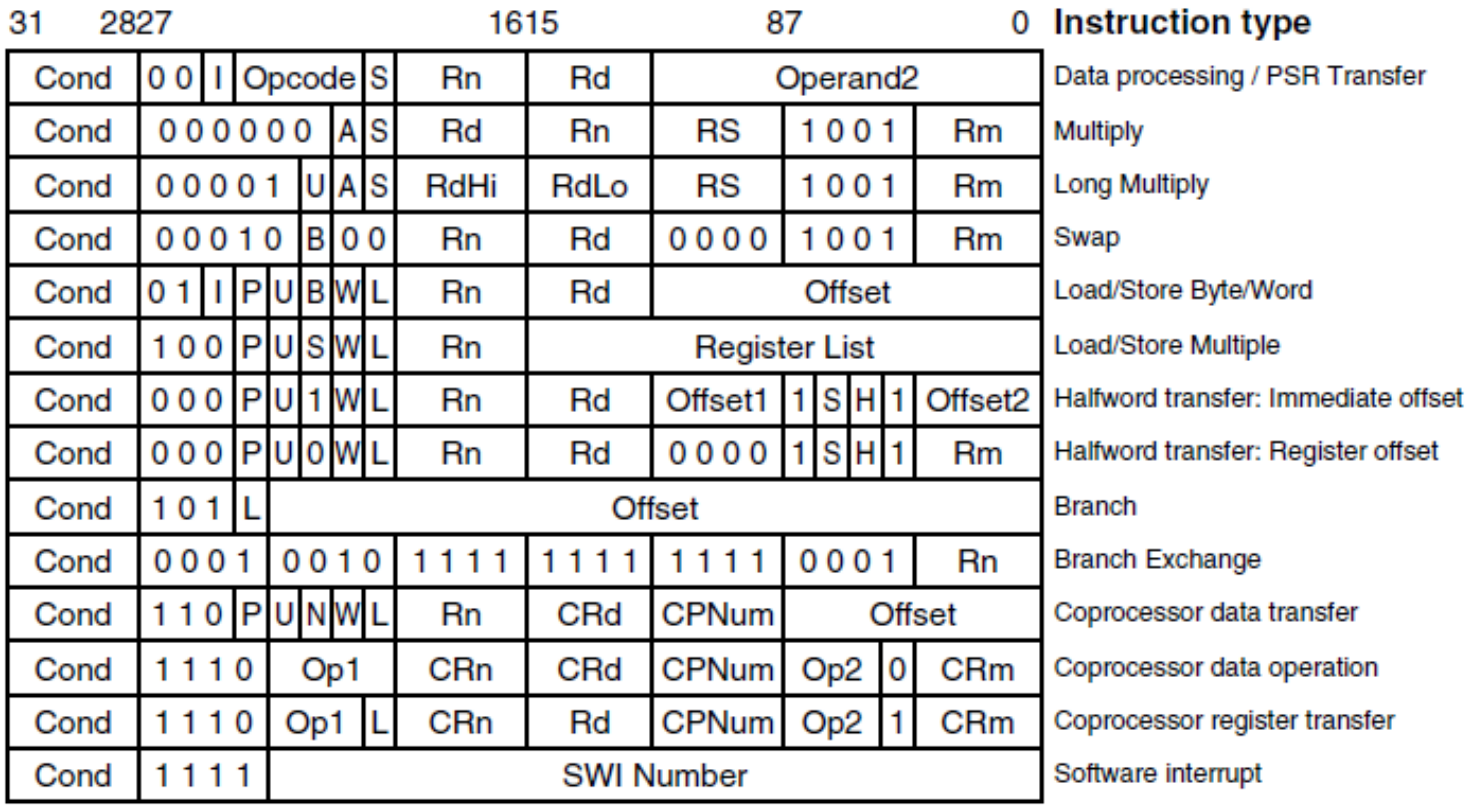
- calcolare quante istruzioni a 3 indirizzi e' possibile avere al massimo;
- mostrare schematicamente una possibile organizzazione delle istruzioni a due indirizzi;
- lasciando invariata l'organizzazione ed il numero di istruzioni ad uno, a due e a tre indirizzi, calcolare quante istruzioni a zero indirizzi e` possibile avere al massimo.

Formato delle Istruzioni del Core i7



- Lunghezza delle istruzioni molto variabile
- Uno dei due operandi è sempre un registro, l'altro può essere sia un registro che in memoria
- MODE stabilisce la modalità di indirizzamento
- L'indirizzo di memoria è lo "spiazzamento" di un segmento (offset), dipende dall'opcode, dai registri usati e dal prefisso
- Possibile l'indirizzamento immediato: operando nell'istruzione

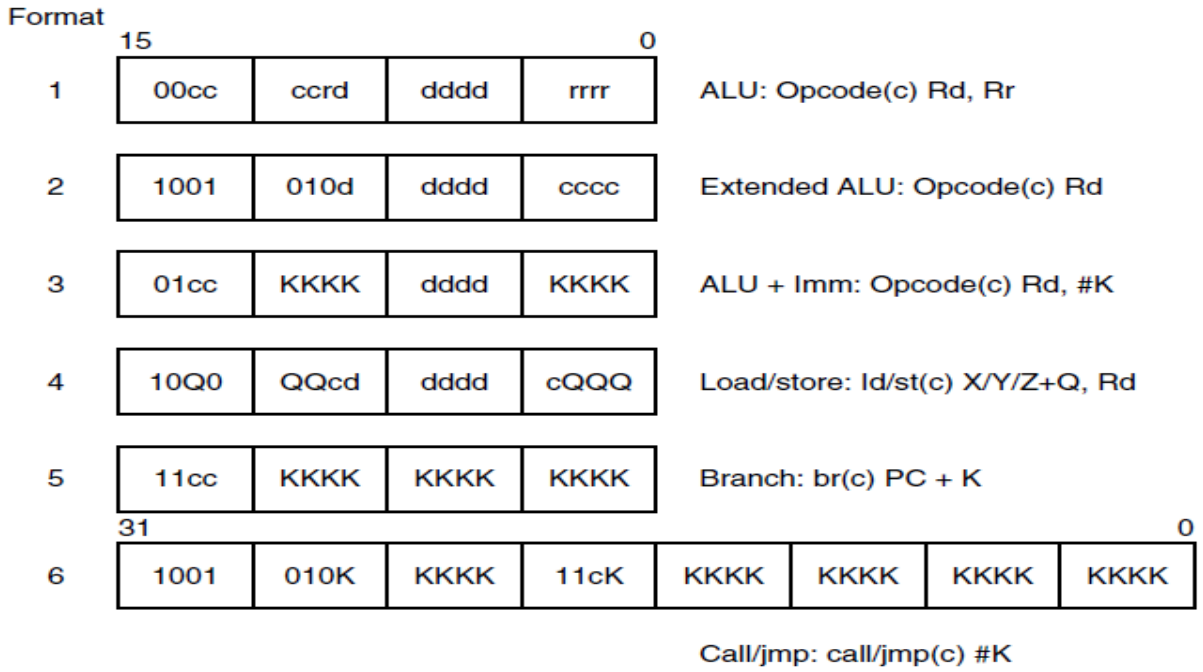
Formato delle istruzioni dell'OMAP4430 ARM



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0	Instruction type
Cond	0	0	I	Opcode				S	Rn	Rd	Operand2						Data processing / PSR Transfer
Cond	0	0	0	0	0	0	A	S	Rd	Rn	RS	1	0	0	1	Rm	Multiply
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	RS	1	0	0	1	Rm	Long Multiply
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	Swap
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset						Load/Store Byte/Word
Cond	1	0	0	P	U	S	W	L	Rn	Register List							Load/Store Multiple
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset1	1	S	H	1	Offset2	Halfword transfer: Immediate offset
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	Halfword transfer: Register offset
Cond	1	0	1	L	Offset												Branch
Cond	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rn	Branch Exchange
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CPNum	Offset					Coprocessor data transfer
Cond	1	1	1	0	Op1				CRn	CRd	CPNum	Op2	0	CRm			Coprocessor data operation
Cond	1	1	1	0	Op1			L	CRn	Rd	CPNum	Op2	1	CRm			Coprocessor register transfer
Cond	1	1	1	1	SWI Number												Software interrupt

- Lunghezza a 16 o 32 bit (due o tre operandi)
- Indirizzamento immediato a 3, 8, 12, 16 e 24 bit
- I bit 25, 26 e 27 definiscono il formato, i bit precedenti l'istr.
- I 4 bit più alti indicano la condizione di esecuz. in base al PSR

Formato delle istruzioni dell'ATmega168 AVR



- Istruzioni a 2 o a 4 byte, 6 formati in tutto:
  - 1, 2 e 3: operazioni su dati nei registri
    - 1: Rd = Rd **op** Rr
    - 2: Rd = **op** Rd
    - 3: Rd = **op** #K
  - 4: load e store
  - 5 e 6: salti e invocazioni di procedure

Modalità di Indirizzamento

- **Immediato**: il valore dell'operando è nell'istruzione
- **Diretto**: l'istruzione contiene l'indirizzo di memoria completo dell'operando
- **Indiretto**: l'indirizzo di memoria fornito contiene *l'indirizzo dell'operando*
- **A registro**: si specifica un registro che contiene l'operando
- **Indiretto a registro**: il registro specificato contiene *l'indirizzo dell'operando*
- **Indicizzato**: l'indirizzo è dato da una costante più il contenuto di un registro
- **A registro base**: viene sommato a tutti gli indirizzi il contenuto di un registro
- **A stack**: l'operando è sulla cima dello stack (o ci deve andare)

Indirizzamenti vari

- Istruzione con indirizzamento a registro e immediato

```
        MOV R1,#0          ; accumulate the sum in R1, initially 0
        MOV R2,#A          ; R2 = address of the array A
        MOV R3,#A+4096     ; R3 = address of the first word beyond A
LOOP:   ADD R1,(R2)        ; register indirect through R2 to get operand
        ADD R2,#4          ; increment R2 by one word (4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

- Programma con indirizzamento indiretto a registro:
  - Calcola la somma degli elementi di un array di 256 interi che inizia all'indirizzo A
  - Ciascun intero occupa 4 byte
  - La somma viene accumulata in R1
  - R2 punta all'elemento corrente

Indirizzamento indicizzato

```
        MOV R1,#0          ; accumulate the OR in R1, initially 0
        MOV R2,#0          ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096       ; R3 = first index value not to use
LOOP:   MOV R4,A(R2)       ; R4 = A[i]
        AND R4,B(R2)       ; R4 = A[i] AND B[i]
        OR R1,R4           ; OR all the Boolean products into R1
        ADD R2,#4          ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

- Calcola l'OR di A[i] AND B[i] dove A e B sono due array
  - (A[0] AND B[0]) OR (A[1] AND B[1]) OR (A[2] AND B[2]) OR …
- R1 accumula l'OR degli AND
- R2 contiene l'indice corrente sugli array
- R3 contiene la costante 4096, per controllare la fine del loop
- R4 è utilizzato per appoggiare i singoli AND

MOV	R4	R2	124300

Indirizzamento indicizzato esteso (con base)

- L'indirizzo di memoria è calcolato sommando il contenuto di due registri:
  - Un registro memorizza la base
  - Un registro memorizza l'indice
- Esempio:
  - Inizializzando R5 con A e R6 con B:

```
…
LOOP: MOV R4,(R2)(R5)
      AND R4,(R2)(R6)
…
```

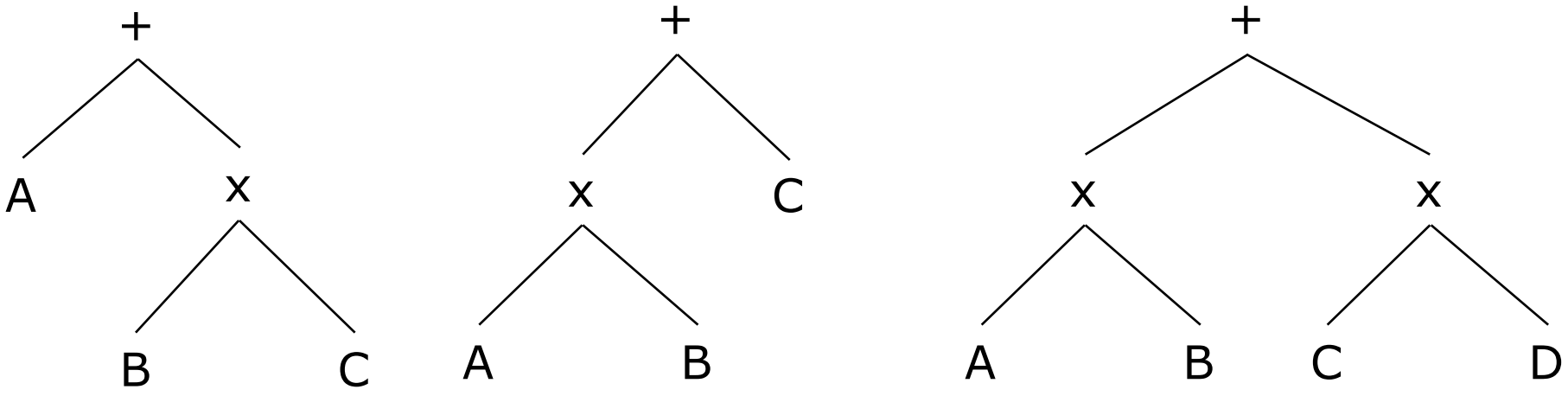
Indirizzamento a Stack

- Lo stack è utilizzato per:
  - Gestire le chiamate di procedura
  - Calcolare espressioni aritmetiche
  - Appoggiare risultati intermedi
- Lo stack pointer SP punta all'elemento affiorante dello stack
- Operazioni fondamentali:
  - PUSH: aggiunge un elemento alla cima dello stack
  - POP: preleva un elemento dalla cima dello stack
  - Operazioni aritmetiche sui due elementi affioranti che mettono al loro posto il risultato

Notazione Polacca Inversa

- Per calcolare un'espressione aritmetica usando lo stack, occorre convertirla dalla forma infissa alla forma postfissa
- Forma infissa: l'operatore è scritto tra gli operandi (necessita di parentesi)
- Forma postfissa: l'operatore segue gli operandi (non richiede parentesi)
- Rappresentando l'espressione con un albero, la forma postfissa si ottiene tramite una visita in postordine

Notazione Polacca Inversa



Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Calcolo di espressioni

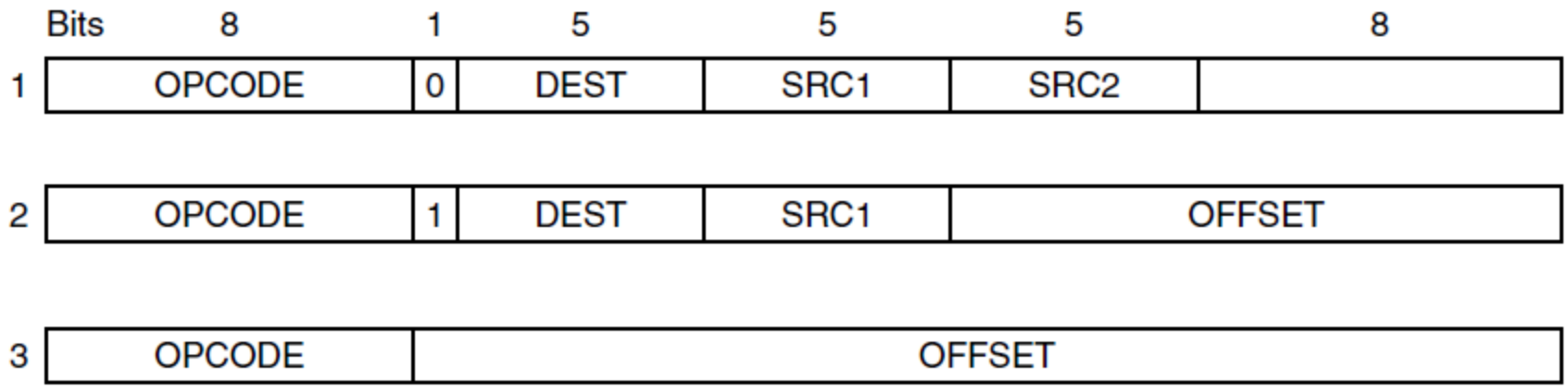
Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	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

- (8 + 2 x 5) / (1 + 3 x 2 – 4)
- Esempio in assembler JVM
- BIPUSH aggiunge l'operando sulla cima dello stack e incrementa SP di 1
- IADD, IMUL ecc. operano sui due elementi affioranti, li rimpiazzano con il risultato e decrementano lo SP
- Scorrendo l'espressione da sinistra a destra:
  - Per gli operandi si fa una BIPUSH
  - Per gli operatori l'operazione corrispondente

Ortogonalità

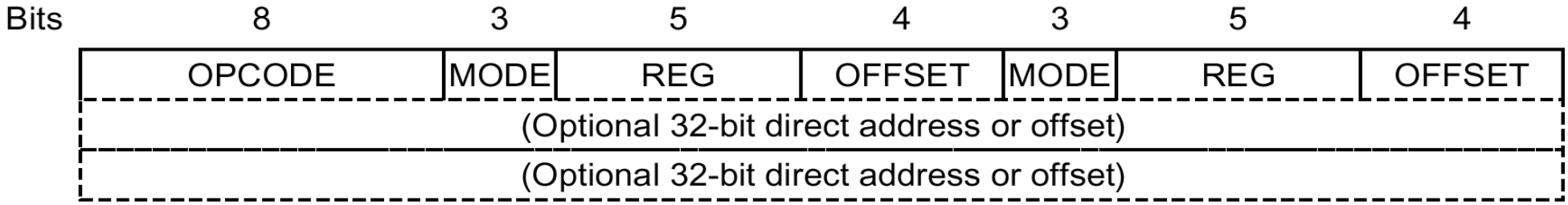
- Un set di istruzioni è caratterizzato da:
  - Codici operativi
  - Modalità di indirizzamento
- In genere non tutte modalità di indirizzamento sono utilizzabili con tutti i codici operativi
- Se questo avviene si dice che i codici e le modalità di indirizzamento sono tra loro ortogonali
- L'ortogonalità è una proprietà molto desiderabile perché semplifica la generazione del codice
- Alcune macchine hanno avuto set di istruzioni ortogonali (es. SPARC)
- Altre (es. Intel) non lo sono per nulla

Linguaggio macchina SPARC



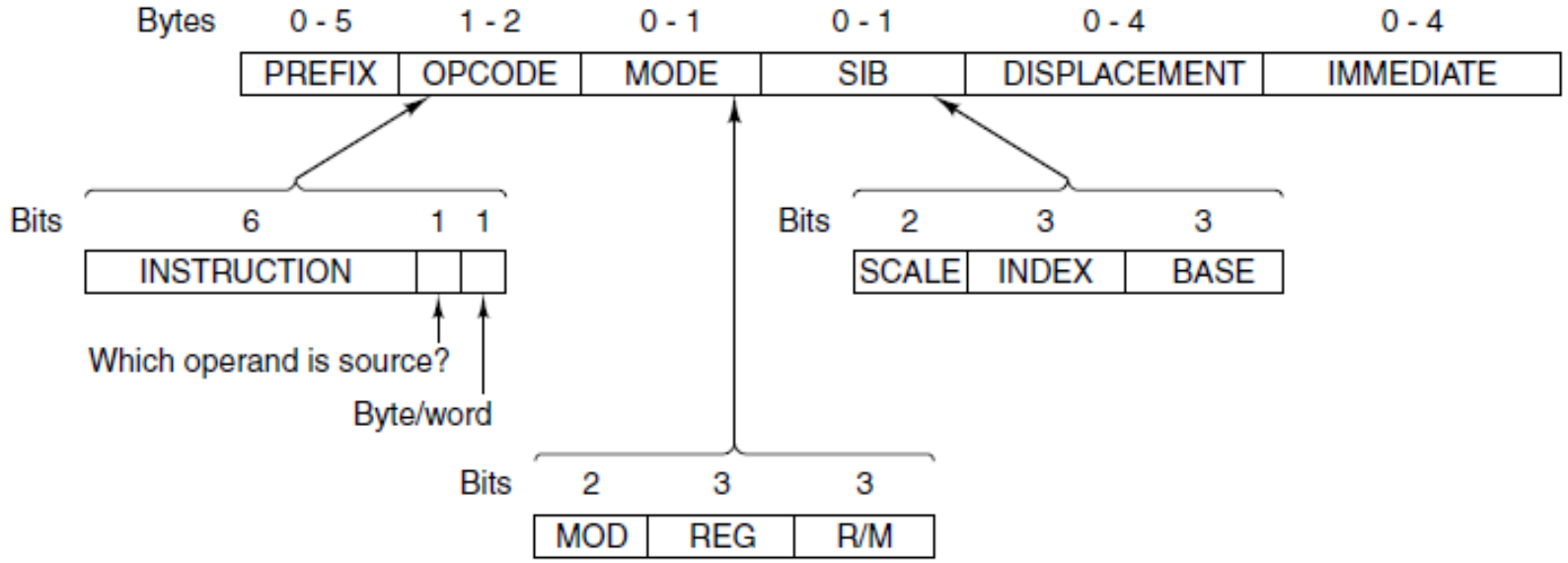
- Indirizzamento immediato o a registri per tutte le istruzioni aritmetiche e logiche
- 5 bit indirizzano i 32 registri della finestra
- Solo le LOAD e le STORE indirizzano la memoria
- Due modi per indirizzare la memoria:
  - tramite registro indice con offset a 13 bit
  - tramite offset a 24 bit

Linguaggio macchina PDP-11



- Istruzioni a 2 operandi con 256 codici operativi
- Indirizzamento simmetrico dei due operandi:
  - REG uno di 32 registri
  - MODE uno di 8 modi di indirizzamento
  - OFFSET di 4 bit
- Una o due word addizionali possono seguire l'istruzione se viene usato il modo diretto o l'offset

Linguaggio macchina x86



- Lunghezza delle istruzioni molto variabile
- Uno dei due operandi è sempre un registro, l'altro può essere sia un registro che una locazione di memoria principale
- MODE ed R/M stabiliscono la modalità di indirizzamento
- L'indirizzo di memoria è l'offset di un segmento, dipende dall'opcode, dai registri usati e dal prefisso
- Possibile l'indirizzamento immediato: operando nell'istruzione

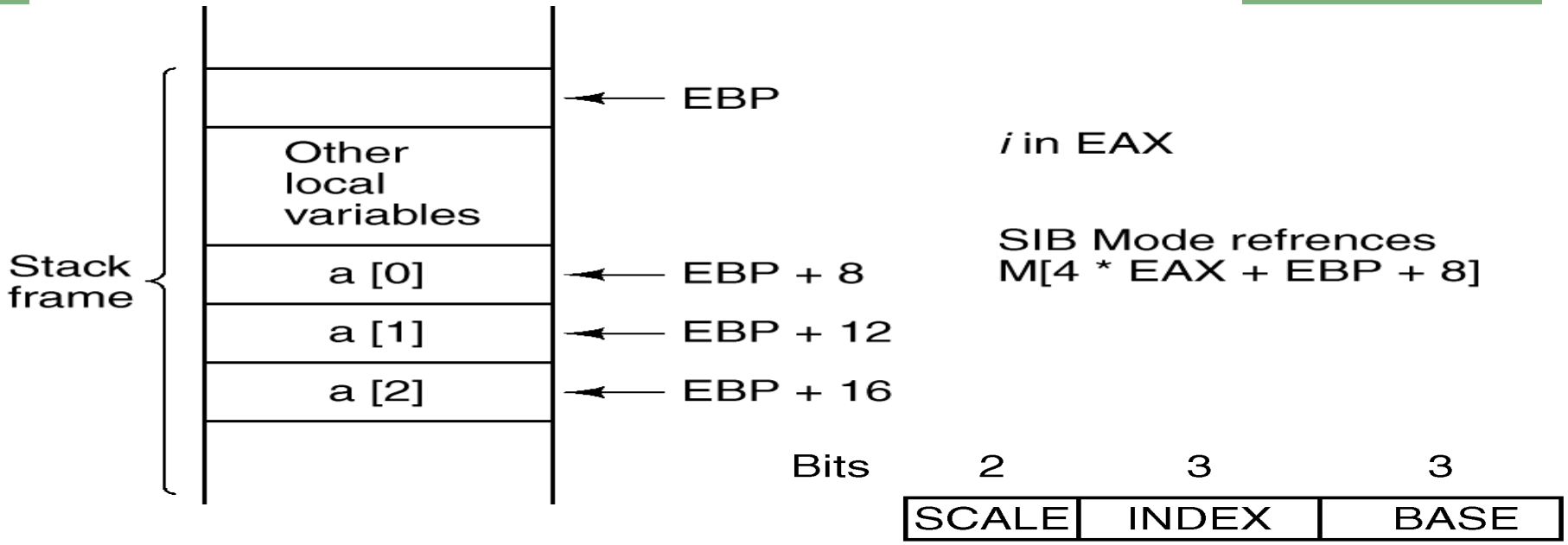
Indirizzamento nel Core i7

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

Indirizzamento Core i7

- Uno degli operandi è sempre un registro specificato dal campo REG dell'istruzione
- L'altro è specificato da MOD e R/M
- 32 modi di indirizzamento possibili
- MOD=00 indirizzamento indiretto a registro, tramite R/M si sceglie quale registro
- MOD=01 ripete le stesse modalità con offset a 8 bit (in coda all'istruzione)
- MOD=10 ripete le stesse modalità con offset a 32 bit (in coda all'istruzione)
- MOD=11 viene utilizzato se il secondo operando è un registro: a 32 bit per le istruzioni a word, a 8 per quelle a byte
- Casi a se sono l'indirizzamento diretto e il SIB

SIB (Scale Index Base)



- Il byte SIB specifica:
  - Fattore di scala: 1, 2, 4, 8 (Es. 4)
  - Registro indice (Es. EAX)
  - Registro base (Es. EBP)
- L'indirizzo viene calcolato come:

  BASE + INDEX·SCALE + OFFSET

  ES EBP + EAX·4 + 8
- Utile nell'accesso ad array (for (i=0; i<n; i++) a[i]=0;)

Indirizzamento OMAP4440 ARM

- Tutte le istruzioni principali usano:
  - Indirizzamento a registro (5 bit), oppure
  - Indirizzamento immediato (12 bit)
- Indirizzamento a memoria solo con LDR e STR tramite
  - Somma di due registri
  - Somma di un registro e di una costante con segno a 13 bit (offset)
  - Somma del PC e di un offset

Indirizzamento ATmega168 AVR

- Quattro modalità di indirizzamento:
  - Istruzioni principali:
    - A registro
    - Immediato (costante con segno a 8 bit)
  - LOAD e STORE
    - Diretto (indirizzo a 7 bit o a 16 bit)
    - Indiretto a registro (a coppie per indirizz. a 16 bit)

Confronto modalità di indirizzamento

Addressing mode	Core i7	OMAP4430 ARM	ATmega168 AVR
Immediate	×	×	×
Direct	×		×
Register	×	×	×
Register indirect	×	×	×
Indexed	×	×	
Based-indexed		×	

Tipi di istruzioni macchina

- Istruzioni di movimento: tra registri o memoria a registri o memoria (in realtà si tratta di copie ovvero assegnazioni);
- Istruzioni binarie: combinano due operandi e producono un risultato (per es. operazioni aritmetiche);
- Istruzioni unarie: prendono un operando e producono un risultato (per es. shift e complementi);
- Salti (condizionali e non): servono a codificare istruzioni condizionali e cicli;
- Chiamate di procedure: alterano il flusso di esecuzione attraverso una gestione della memoria "a pila".

Esempio di istruzioni binarie

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

```
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
```

Esempi di istruzioni unarie

```
00000000 00000000 00000000 01110011  A
00000000 00000000 00000000 00011100  A shifted right 2 bits
11000000 00000000 00000000 00011100  A rotated right 2 bits

11111111 11111111 11111111 11110000  A
00111111 11111111 11111111 11111100  A shifted without sign extension
11111111 11111111 11111111 11111100  A shifted with sign extension

11111111 11111111 11111111 11111110  −1 in ones' complement
11111111 11111111 11111111 11111100  −1 shifted left 1 bit = −3
11111111 11111111 11111111 11111000  −1 shifted left 2 bits = −7

11111111 11111111 11111111 11111111  −1 in two's complement
11111111 11111111 11111111 11111111  −1 shifted right 6 bits = −1
```

Istruzioni di ciclo

- Realizzato con salti condizionati
- Valutazione in coda o valutazione in testa

```
        i = 1;

L1:     first-statement;
        .
        .
        .
        last-statement;
        i = i + 1;
        if (i < n) goto L1;
```

(a)

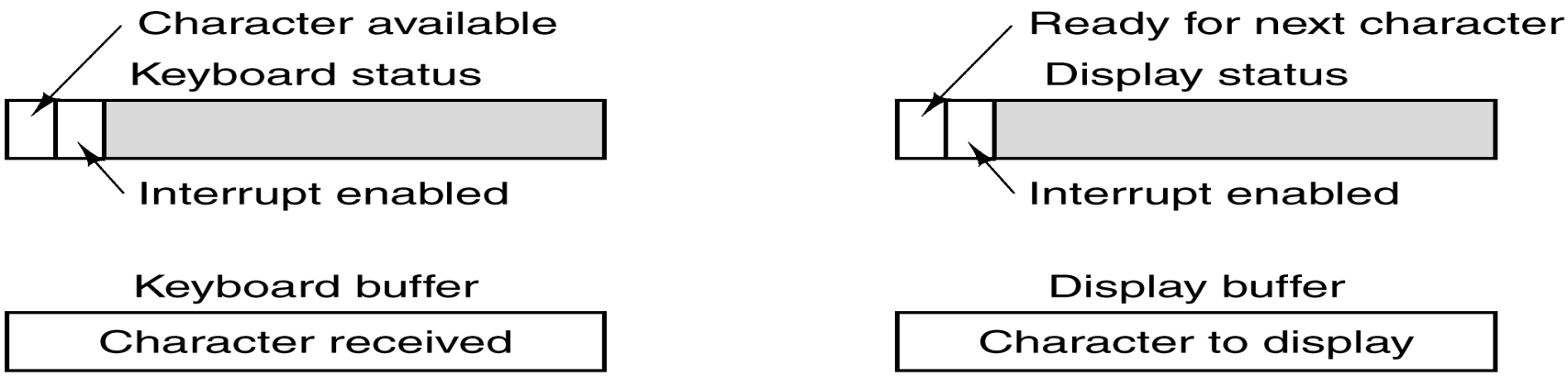
```
        i = 1;
L1:     if (i > n) goto L2;
        first-statement;
        .
        .
        .
        last-statement
        i = i + 1;
        goto L1;
L2:
```

(b)

Istruzioni di I/O

- Un'operazione di I/O consiste nel trasferimento di dati tra un device di I/O e la memoria
- Tre modi fondamentali di gestire l'I/O:
  1) I/O programmato con busy waiting (attesa attiva)
     - La CPU interroga periodicamente i dispositivi (polling) e cicla a vuoto durante le attese (busy waiting)
  2) I/O gestito con interruzioni
     - La CPU effettua i trasferimenti con la memoria e avvia l'operazione di I/O ma si dedica ad altro fino a che il device non manda una interruzione
  3) DMA (Direct Memory Access)
     - La CPU avvia l'operazione poi gestita interamente dal controllore DMA

I/O Programmato



- I *controller* hanno diverse porte che possono essere lette e scritte dalla CPU
- Nei buffer vengono letti o scritti i caratteri scambiati con il controller
- I registri di stato contengono bit che la CPU controlla per sapere se i dati sono disponibili, o possono essere scritti
- Dati i tempi dei dispositivi di I/O il *busy waiting* comporta un notevole spreco della risorsa CPU
- Usato solo in sistemi molto semplici (o d'antiquariato)

I/O Programmato (Esempio)

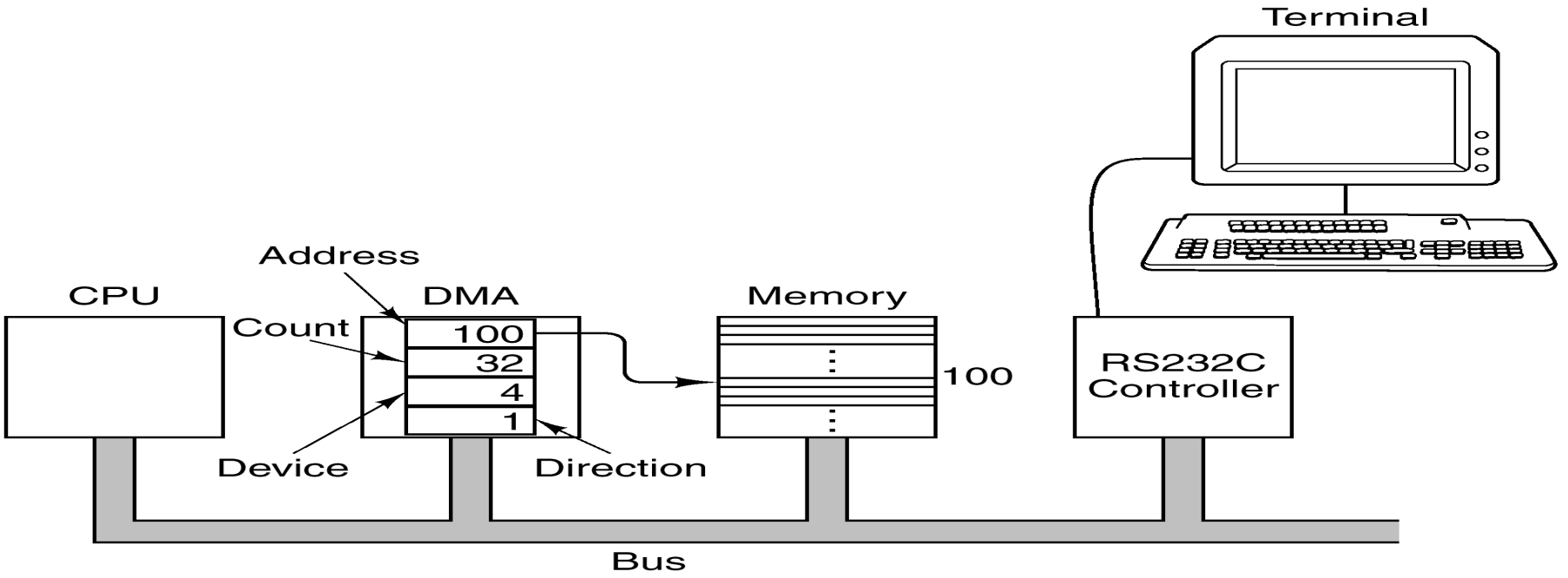
```
// Output a block of data to the device
int status, i, ready;

for (i = 0; i < count; i++) {
    do {
        status = in(display_status_reg);        // get status
        ready = (status >> 7) & 0x01;           // isolate ready bit
    } while (ready != 1);
    out(display_buffer_reg, buf[i]);
}
}
```

I/O con Interrupt

Esempio: lettura da disco

- La CPU avvia l'operazione di I/O scrivendo le informazioni opportune nelle porte del controller
- La CPU passa all'elaborazione di un altro task
- Il controller avvia e sovraintende allo svolgimento dell'operazione di I/O (posizionamento delle testine ecc.)
- Solo quando i dati sono disponibili il controller interrompe la CPU
- La CPU è direttamente coinvolta nel trasferimento dei dati tra controller, essa legge i dati e li copia in memoria

DMA (Direct Memory Access)



- La CPU programma il controller DMA specificando:
  - Quanti byte trasferire
  - Da quale device
  - A che indirizzi
- Il controller gestisce l'intera operazione
- Il controller DMA può gestire più operazioni contemporaneamente

Istruzioni Core i7

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOVcc DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract SRC from DST
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract SRC & carry from DST
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TEST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT n	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Istruzioni OMAP4430 ARM (1)

Loads

LDRSB DST,ADDR	Load signed byte (8 bits)
LDRB DST,ADDR	Load unsigned byte (8 bits)
LDRSH DST,ADDR	Load signed halfwords (16 bits)
LDRH DST,ADDR	Load unsigned halfwords (16 bits)
LDR DST,ADDR	Load word (32 bits)
LDM S1,REGLIST	Load multiple words

Stores

STRB DST,ADDR	Store byte (8 bits)
STRH DST,ADDR	Store halfword (16 bits)
STR DST,ADDR	Store word (32 bits)
STM SRC,REGLIST	Store multiple words

Arithmetic

ADD DST,S1,S2IMM	Add
ADD DST,S1,S2IMM	Add with carry
SUB DST,S1,S2IMM	Subtract
SUB DST,S1,S2IMM	Subtract with carry
RSB DST,S1,S2IMM	Reverse subtract
RSC DST,S1,S2IMM	Reverse subtract with carry
MUL DST,S1,S2	Multiply
MLA DST,S1,S2,S3	Multiple and accumulate
UMULL D1,D2,S1,S2	Unsigned long multiple
SMULL D1,D2,S1,S2	Signed long multiple
UMLAL D1,D2,S1,S2	Unsigned long MLA
SMLAL D1,D2,S1,S2	Signed long MLA
CMP S1,S2IMM	Compare and set PSR

Shifts/rotates

LSL DST,S1,S2IMM	Logical shift left
LSR DST,S1,S2IMM	Logical shift right
ASR DST,S1,S2IMM	Arithmetic shift right
ROR DSR,S1,S2IMM	Rotate right

Boolean

TST DST,S1,S2IMM	Test bits
TEQ DST,S1,S2IMM	Test equivalence
AND DST,S1,S2IMM	Boolean AND
EOR DST,S1,S2IMM	Boolean Exclusive-OR
ORR DST,S1,S2IMM	Boolean OR
BIC DST,S1,S2IMM	Bit clear

Transfer of control

Bcc IMM	Branch to PC+IMM
BLcc IMM	Branch with link to PC+IMM
BLcc S1	Branch with link to reg add

Miscellaneous

MOV DST,S1	Move register
MOVT DST,IMM	Move imm to upper bits
MVN DST,S1	NOT register
MRS DST,PSR	Read PSR
MSR PSR,S1	Write PSR
SWP DST,S1,ADDR	Swap reg/mem word
SWPB DST,S1,ADDR	Swap reg/mem byte
SWI IMM	Software interrupt

S1 = source register
S2IMM = source register or immediate
S3 = source register (when 3 are used)
DST = destination register
D1 = destination register (1 of 2)
D2 = destination register (2 of 2)

ADDR = memory address
IMM = immediate value
REGLIST = list of registers
PSR = processor status register
cc = branch condition

Istruzioni ATmega168 AVR

Instruction	Description	Semantics
ADD DST,SRC	Add	DST ← DST + SRC
ADC DST,SRC	Add with Carry	DST ← DST + SRC + C
ADIW DST,IMM	Add Immediate to Word	DST+1:DST ← DST+1:DST + IMM
SUB DST,SRC	Subtract	DST ← DST - SRC
SUBI DST,IMM	Subtract Immediate	DST ← DST - IMM
SBC DST,SRC	Subtract with Carry	DST ← DST - SRC - C
SBCI DST,IMM	Subtract Immediate with Carry	DST ← DST - IMM – C
SBIW DST,IMM	Subtract Immediate from Word	DST+1:DST ← DST+1:DST - IMM
AND DST,SRC	Logical AND	DST ← DST AND SRC
ANDI DST,IMM	Logical AND with Immediate	DST ← DST AND IMM
OR DST,SRC	Logical OR	DST ← DST OR SRC
ORI DST,IMM	Logical OR with Immediate	DST ← DST OR IMM
EOR DST,SRC	Exclusive OR	DST ← DST XOR SRC
COM DST	One's Complement	DST ← 0xFF - DST
NEG DST	Two's Complement	DST ← 0x00 - DST
SBR DST,IMM	Set Bit(s) in Register	DST ← DST OR IMM
CBR DST,IMM	Clear Bit(s) in Register	DST ← DST AND (0xFF - IMM)
ING DST	Increment	DST ← DST + 1
DEC DST	Decrement	DST ← DST - 1
TST DST	Test for Zero or Minus	DST ← DST AND DST
CLR DST	Clear Register	DST ← DST XOR DST
SER DST	Set Register	DST ← 0xFF
MUL DST,SRC	Multiply Unsigned	R1:R0 ← DST * SRC
MUL DST,SRC	Multiply Unsigned	R1:R0 ← DST * SRC
MULS DST,SRC	Multiply Signed	R1:R0 ← DST * SRC
MULSU DST,SRC	Multiply Signed with Unsigned	R1:R0 ← DST * SRC
RJMP IMM	PC-relative Jump	PC ← PC + IMM + 1
IJMP	Indirect Jump to Z	PC ← Z (R30:R31)
JMP IMM	Jump	PC ← IMM
RCALL IMM	Relative Call	STACK ← PC+2, PC ← PC + IMM + 1
ICALL	Indirect Call to (Z)	STACK ← PC+2, PC ← Z (R30:R31)
CALL	Call	STACK ← PC+2, PC ← IMM
RET	Return	PC ← STACK
CP DST,SRC	Compare	DST – SRC
CPC DST,SRC	Compare with Carry	DST – SRC – C
CPI DST,IMM	Compare with Immediate	DST – IMM
BRcc IMM	Branch on Condition	if cc(true) PC ← PC + IMM + 1
MOV DST,SRC	Copy Register	DST ← SRC
MOVW DST,SRC	Copy Register Pair	DST+1:DST ← SRC+1:SRC
LDI DST,IMM	Load Immediate	DST ← IMM
LDS DST,IMM	Load Direct	DST ← MEM[IMM]
LD DST,XYZ	Load Indirect	DST ← MEM[XYZ]
LDD DST,XYZ+IMM	Load Indirect with Displacement	DST ← MEM[XYZ+IMM]
STS IMM,SRC	Store Direct	MEM[IMM] ← SRC
ST XYZ,SRC	Store Indirect	MEM[XYZ] ← SRC
STD XYZ+IMM,SRC	Store Indirect with Displacement	MEM[XYZ+IMM] ← SRC
PUSH REGLIST	Push Register on Stack	STACK ← REGLIST
POP REGLIST	Pop Register from Stack	REGLIST ← STACK
LSL DST	Logical Shift Left by One	DST ← DST LSL 1
LSR DST	Logical Shift Right by One	DST ← DST LSR 1
ROL DST	Rotate Left by One	DST ← DST ROL 1
ROR DST	Rotate Right by One	DST ← DST ROR 1
ASR DST	Arithmetic Shift Right by One	DST ← DST ASR 1

SRS = source register
DST = destination register
IMM = immediate value

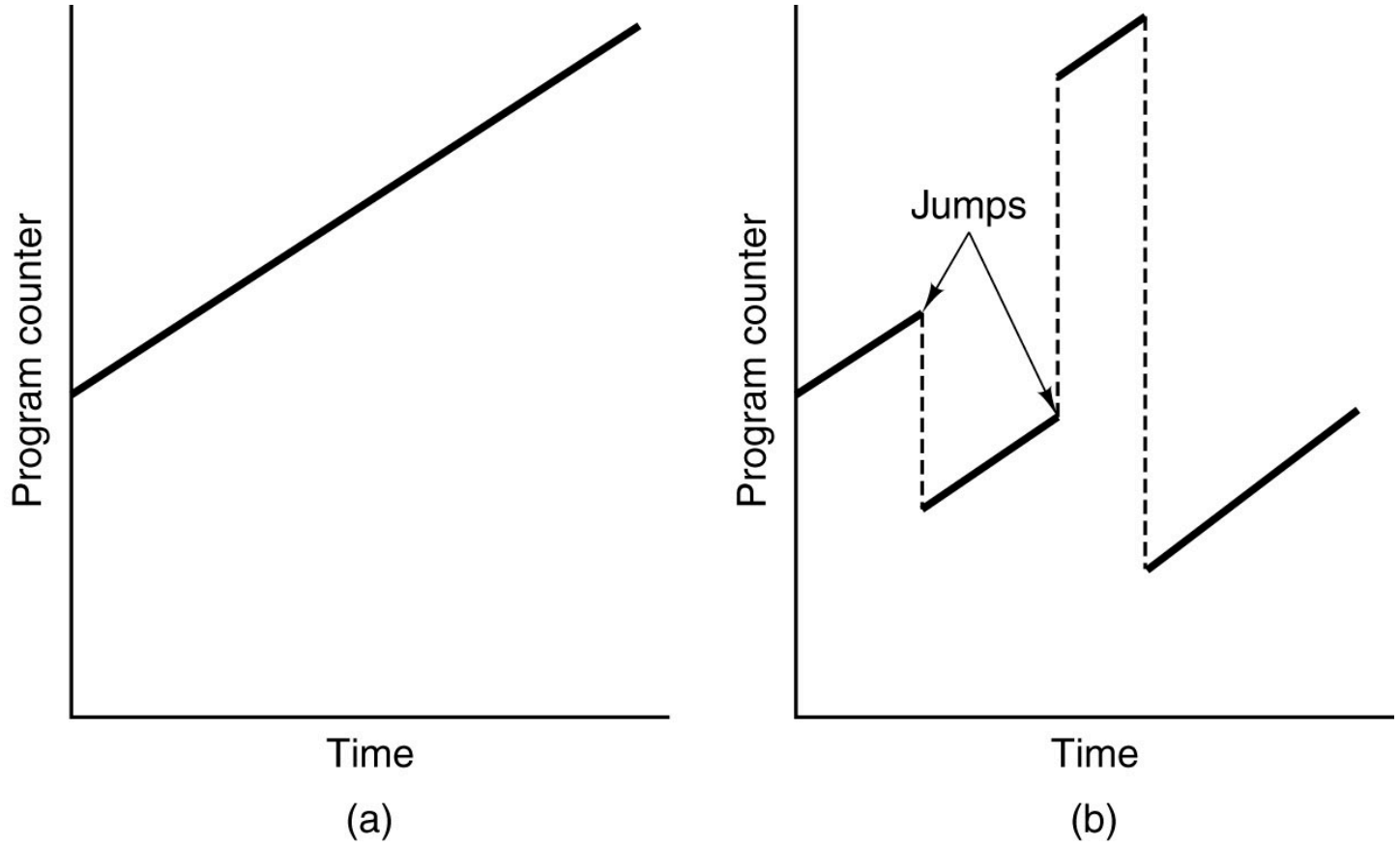
XYZ = X, Y, or Z register pair
MEM[A] = access memory at address A

Controllo del flusso

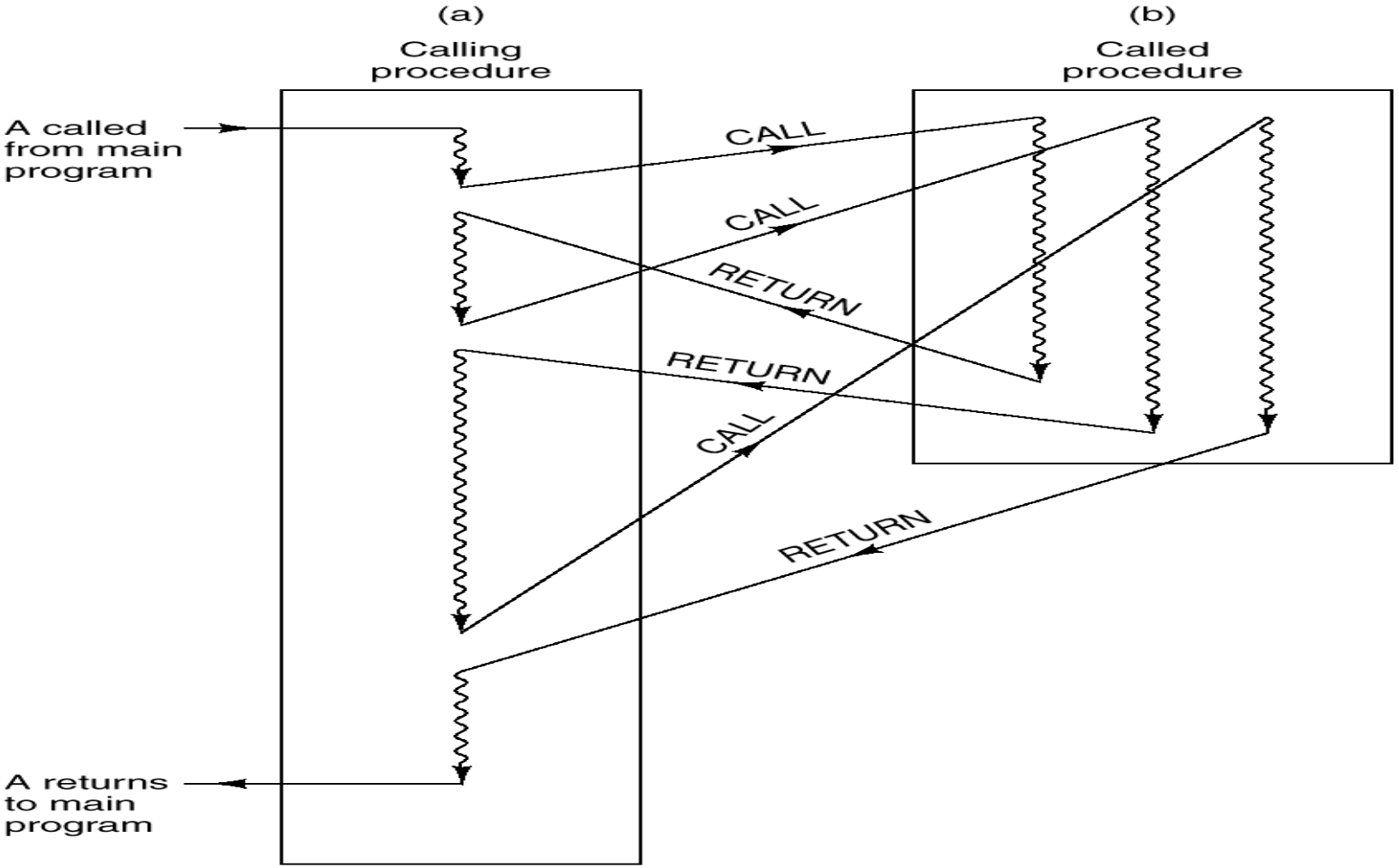
Tecniche che possono alterare l'esecuzione:

- Salti (condizionati e non)
- Chiamata di procedure
- Coroutine
- Trap
- Interrupt

Flusso sequenziale e salti (diramazioni)



Chiamata a procedura

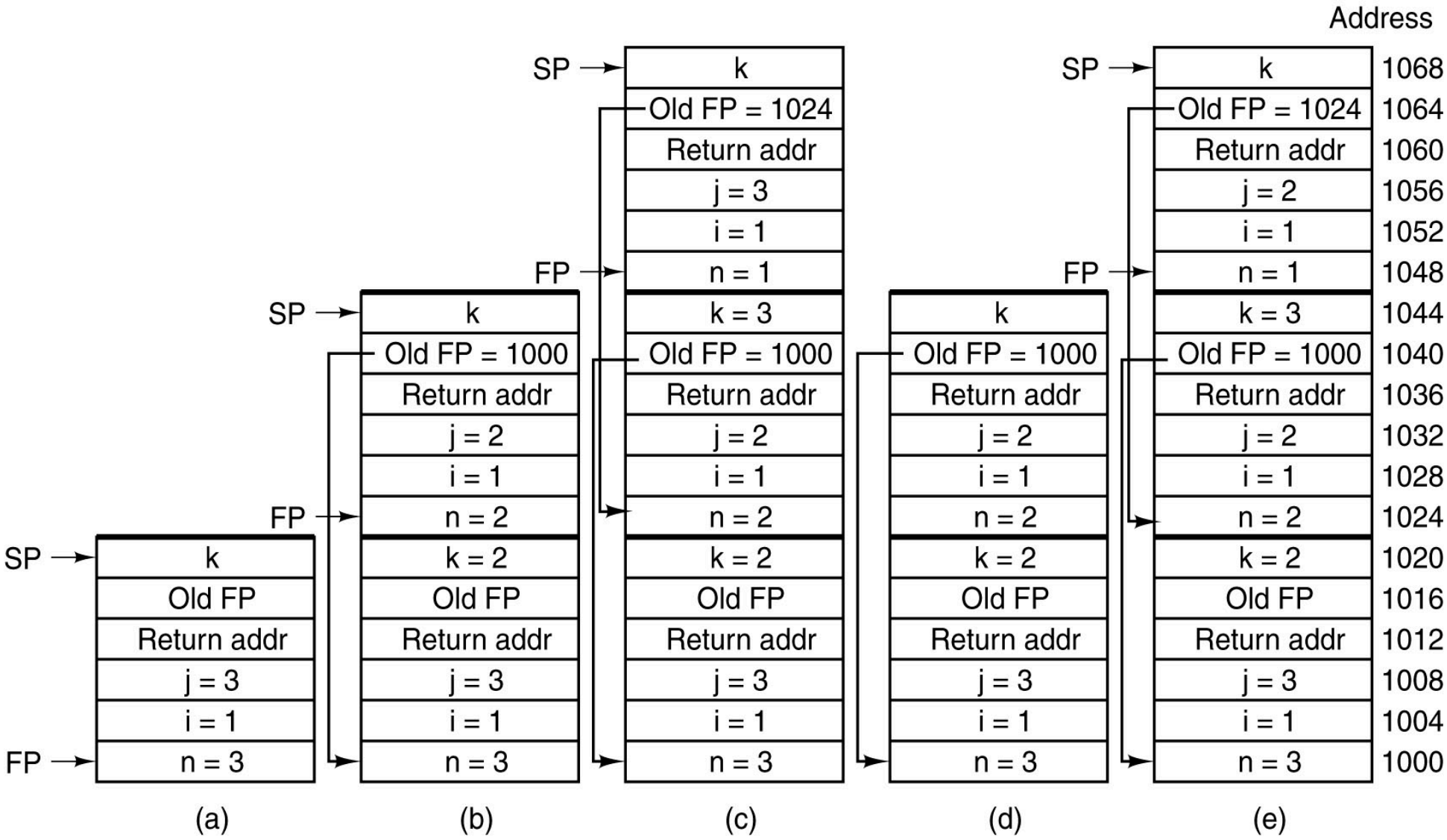


- Situazione asimmetrica tra procedura chiamante e procedura chiamata
- Ciascuna chiamata rientra all'inizio della procedura

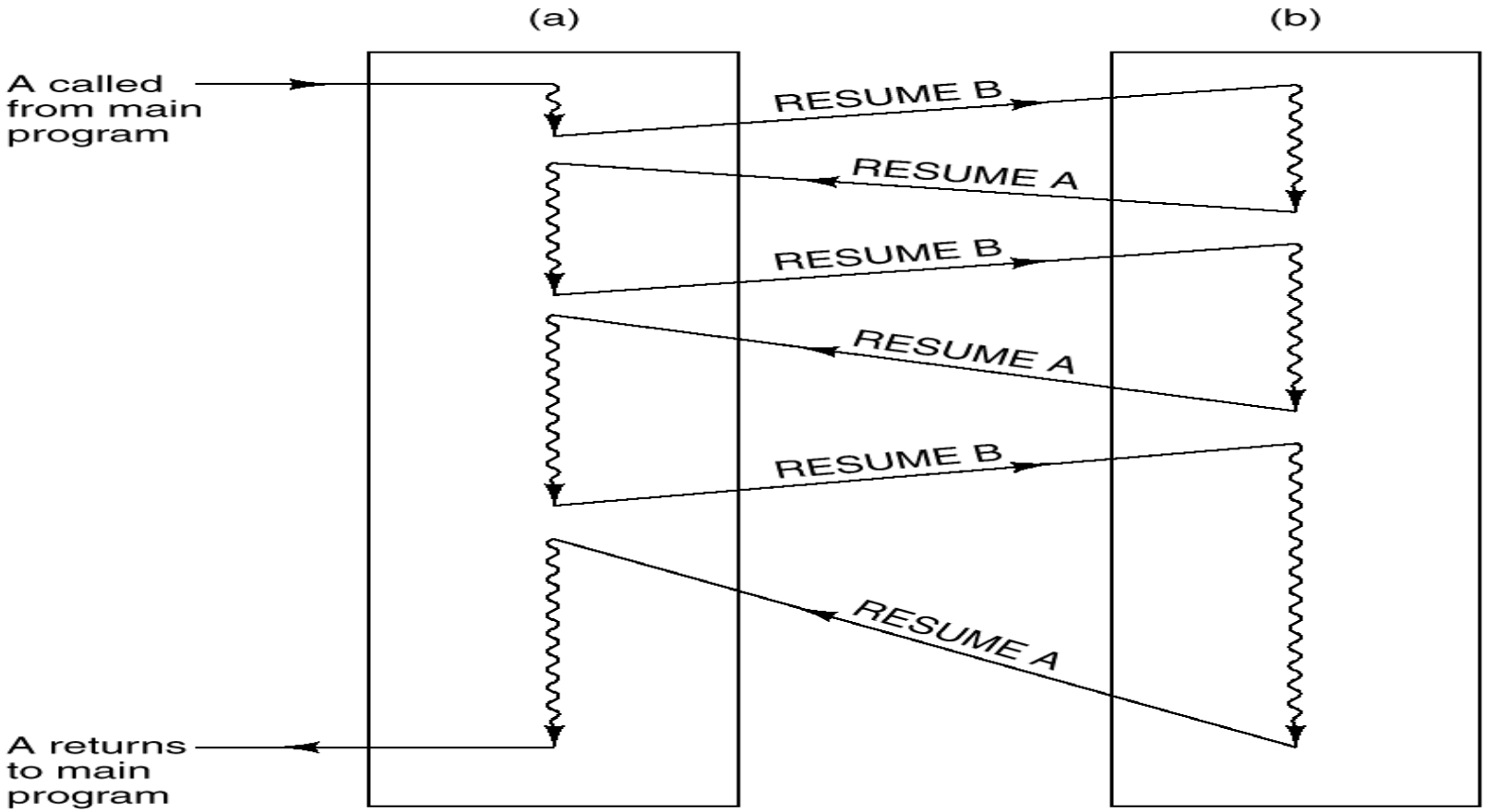
Chiamata di procedura

- Per ciascuna chiamata viene allocato sullo stack un nuovo *stack frame*
- Lo stack frame contiene:
  - I parametri in entrata e in uscita
  - Le variabili locali
  - L'indirizzo di rientro
  - Un puntatore allo stack frame del chiamante
- Lo stack pointer SP punta alla cima dello stack
- Il base pointer BP punta alla base del frame
- L'accesso ai parametri e alle variabili locali e avviene tramite offset da BP
- La posizione rispetto a BP è nota a tempo di compilazione e costante
- La posizione rispetto a SP non è costante (possono essere fatte PUSH e POP durante l'esecuzione)
- All'atto del rientro lo stack frame viene deallocato

Struttura della Stack Frame



Coroutine



- Situazione simmetrica tra le coroutine
- RESUME al posto di CALL e RETURN
- Ciascuna RESUME riparte dall'istruzione successiva alla precedente

Trap

- La trap è una procedura automatica che viene iniziata da una condizione eccezionale che si verifica durante l'esecuzione di un programma
- Le trap sono sincrone e dipendenti da quello che succede sulla CPU, mentre le interruzioni sono asincrone e nascono all'esterno della CPU
- Le trap si originano da test fatti a livello del microprogramma
- La gestione delle trap è affidata al trap handler ed è in tutto simile a quella delle interrupt
- Esempi di trap:
  - overflow e underflow
  - violazione di protezione
  - divisione per zero

Interruzioni: Azioni HW

Quando l'interruzione si origina e viene servita queste azioni preliminari *vengono svolte a livello hardware*:

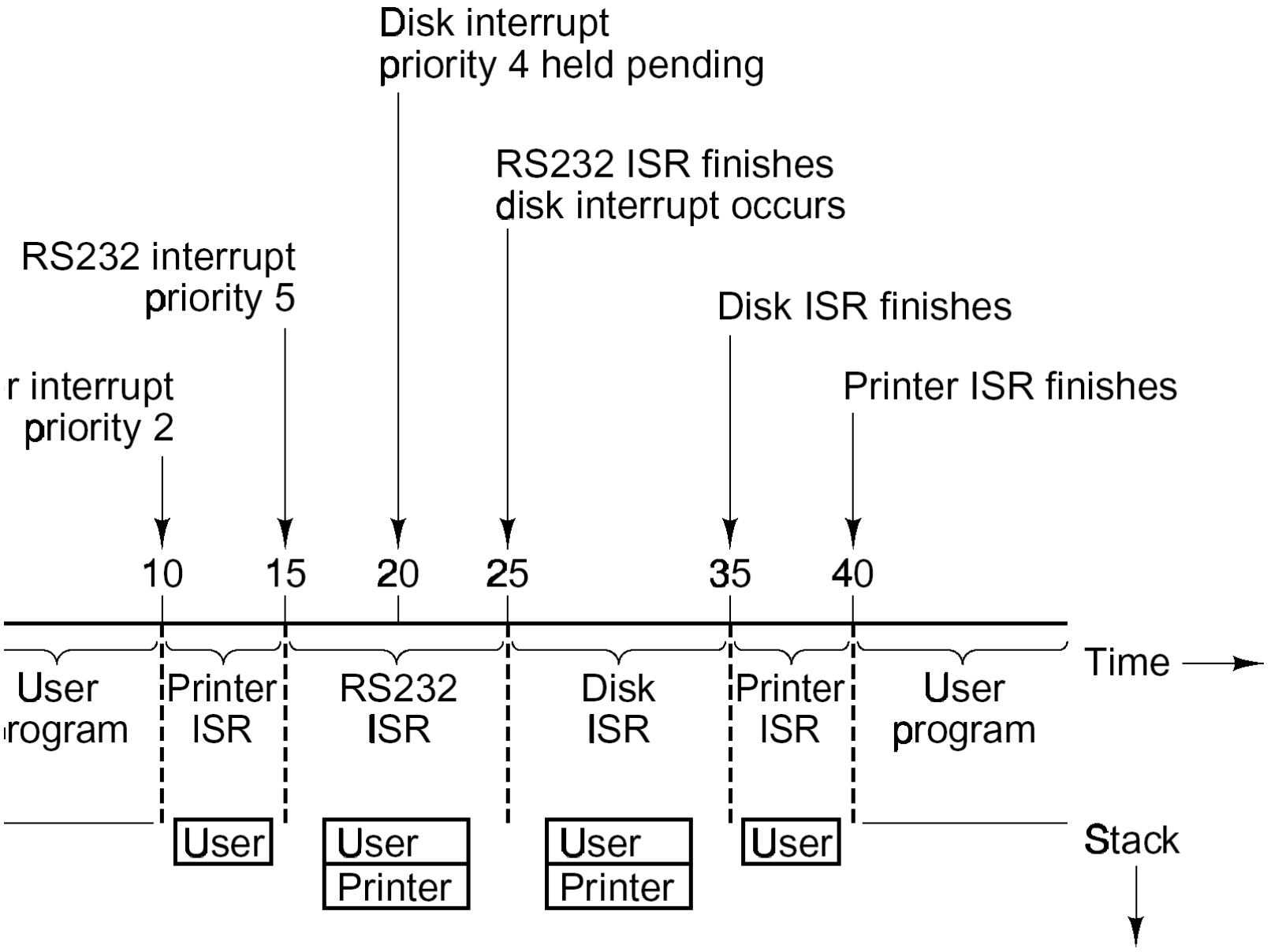
1) Il controller genera l'interruzione
2) La CPU, quando è pronta a servirla, invia il segnale di *acknowledge*
3) Quando il controller vede l'acknowledge risponde mettendo sul bus un identificatore detto *vettore di interruzione*
4) La CPU legge e salva il vettore di interruzione
5) La CPU salva il PC (Program Counter) e la PSW (Program Status word) sullo stack
6) La CPU individua, per il tramite del vettore di interruzione, l'indirizzo iniziale della routine che serve l'interruzione e lo carica nel PC

Interruzioni: Azioni SW

Inizia ora l'esecuzione della routine di servizio che svolge le seguenti azioni:

7) Salva sullo stack i registri della CPU
8) Individua il numero esatto del device tramite lettura di opportuni registri
9) Legge tutti i codici di stato ecc.
10) Gestisce eventuali errori di I/O
11) Legge (o scrive) i dati e incrementa i conteggi
12) Se necessario informa il device che il servizio dell'interruzione è concluso
13) Ricarica tutti i registri salvati sullo stack
14) Esegue un'istruzione di RETURN FROM INTERRUPT ripristinando lo stato della CPU precedente l'interruzione

Interruzioni multiple: esempio di temporizzazione



L'architettura IA-64

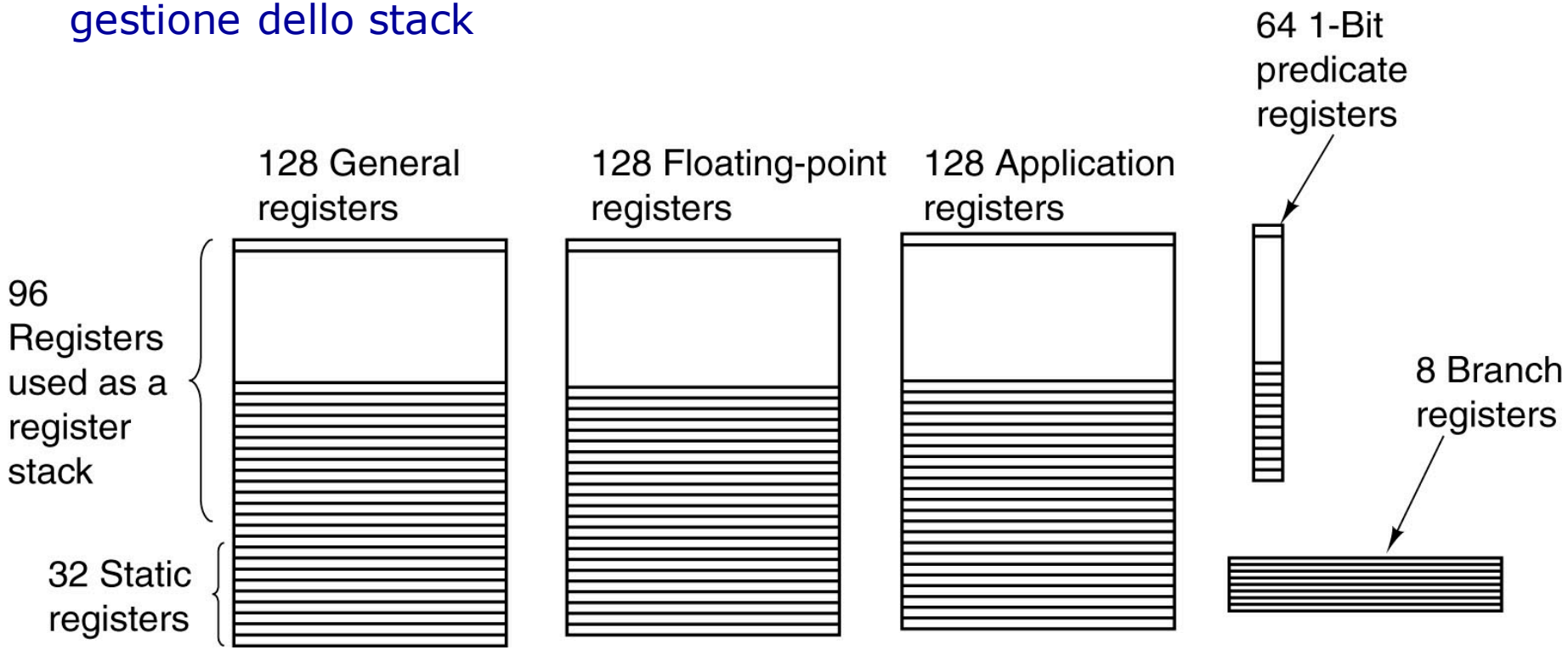
- Dopo aver spremuto fino in fondo la IA-32, Intel è ha proposto nuove ISA che rompono con il passato:
  - EMT-64 (ISA Pentium esteso a 64 bit)
  - IA-64
- La IA-64 è una architettura a 64 bit sviluppata in collaborazione con HP
- Disegno basato in parte sull'architettura PA-RISC di HP
- L'implementazione è una classe di CPU di fascia alta denominata *Itanium*
- Innovativa, ma di poco successo.
- Idea di base:
  - Spostare il carico di lavoro dall'esecuzione alla compilazione

I problemi dell'IA-32

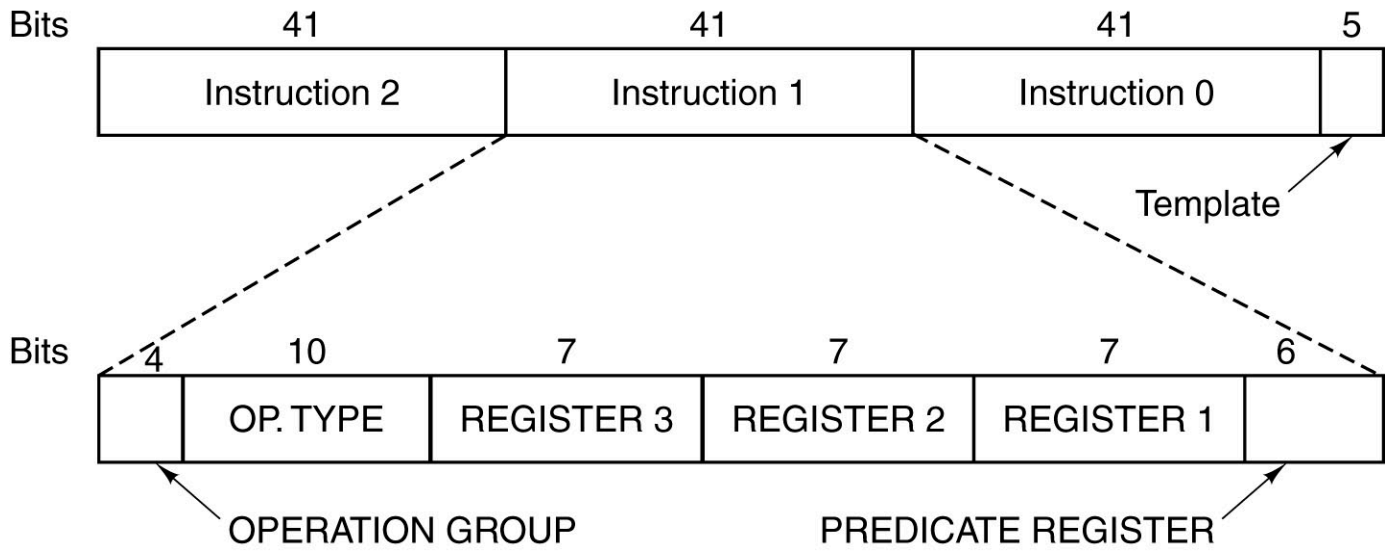
- E' irrimediabilmente CISC: le sue istruzioni possono essere spezzate in istruzioni RISC ma questo richiede tempo e spazio su chip
- Indirizzamento orientato a memoria
- Pochi registri e asimmetrici: molti risultati intermedi devono essere appoggiati in memoria
- Pochi registri = molte dipendenze: rende difficile l'esecuzione parallela di più istruzioni
- Necessita di una pipeline lunga: rende difficile la predizione dei salti
- Rimedia parzialmente con l'esecuzione speculativa
- 4 GB di spazio di indirizzamento: ormai poco per un grosso server. Problema risolto con le architettura a 64b.

Modello memoria IA-64

- 2^{64} byte di memoria lineare
- Parole da 1, 2, 4, 16 e 10 byte
- 128 registri di uso generale e 128 in FP da 64 bit
- 128 registri ad uso speciale
- 64 registri predicativi da 1 bit
- 8 registri di salto
- Ogni procedura vede 32 registri statici e altri (fino a 96) per la gestione dello stack



Scheduling delle istruzioni nell'IA-64



- *EPIC (Explicitly Parallel Instruction Computing)*
- Si cerca di evidenziare le possibilità di esecuzione parallela delle istruzioni raggruppandole
- Le istruzioni vengono in *bundle:* pacchetti di tre senza dipendenze
- I bundle possono essere a loro volta raggruppati
- La parte *template* fornisce informazioni sulle possibilità di esecuzione parallela su unità funzionali indipendenti
- Molto del lavoro è spostato al tempo di compilazione ed ottimizzazione

Esecuzione predicativa

```
if (R1 == 0)                CMP R1,0          CMOVZ R2,R3,R1
    R2 = R3;                BNE L1
                            MOV R2,R3
                        L1:
```

(a) (b) (c)

- Si cerca di diminuire i salti condizionati con la tecnica della *esecuzione predicativa*
- La esecuzione predicativa è un'estensione del concetto di esecuzione condizionale
- Dato il codice sorgente (a) la traduzione classica tramite salto condizionato è data in (b)
- La traduzione in (c) sfrutta un'istruzione ad esecuzione condizionale e non contiene salti condizionati

Esecuzione predicativa (2)

```
if (R1 == 0) {
    R2 = R3;
    R4 = R5;
} else {
    R6 = R7;
    R8 = R9;
}
```

(a)

```
    CMP R1,0
    BNE L1
    MOV R2,R3
    MOV R4.R5
    BR L2
L1: MOV R6,R7
    MOV R8,R9
L2:
```

(b)

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```

(c)

- Nella versione condizionale (c) il codice è costituito da un unico blocco basico senza salti
- L'unico vincolo per l'esecuzione di ciascuna istruzione è la conoscenza della condizione

Esecuzione predicativa (3)

```
if (R1 == R2)
    R3 = R4 + R5;
else
    R6 = R4 – R5
```

(a)

```
    CMP R1,R2
    BNE L1
    MOV R3,R4
    ADD R3,R5
    BR L2
L1: MOV R6,R4
    SUB R6,R5
L2:
```

(b)

```
CMPEQ R1,R2,P4
<P4> ADD R3,R4,R5
<P5> SUB R6,R4,R5
```

(c)

- La prima istruzione stabilisce il valore del registro predicativo P4, e mette P5 al valore negato
- L'esecuzione delle istruzioni successive dipende dai valori di P4 e P5
- Istruzioni predicative possono andare in pipeline senza problemi di stallo