

Appunti di  
Architettura e progettazione dei calcolatori

Corso di laurea magistrale in  
**Ingegneria Informatica**



# Indice

<b>Introduzione</b>	<b>7</b>
<b>1 Richiami ed approfondimenti sui sistemi di elaborazione</b>	<b>9</b>
1.1 Richiami di calcolatori elettronici . . . . .	9
1.1.1 Architettura generale . . . . .	9
1.1.2 Istruzioni . . . . .	11
1.2 Motorola 68k . . . . .	13
1.2.1 Registri General Purpose . . . . .	13
1.2.2 Codici di Spostamento dati o indirizzi . . . . .	13
1.2.3 Codici Aritmetici . . . . .	16
1.2.4 Codici di salto . . . . .	17
1.2.5 Codici Logici . . . . .	19
1.2.6 Codici di Scorrimento . . . . .	20
1.2.7 Codici di Confronto . . . . .	20
1.2.8 Strutture sintattiche fondamentali . . . . .	21
1.2.9 Valutazione degli accessi in memoria . . . . .	23
<b>2 Gestione dei dispositivi di IO</b>	<b>25</b>
2.1 Architettura generale di un dispositivo di I/O . . . . .	25
2.1.1 Modalità di comunicazione . . . . .	26
2.1.2 Interfacciamento CPU e periferica . . . . .	26
2.1.3 BUS . . . . .	27
2.1.4 Driver . . . . .	28
2.1.5 PIA . . . . .	29
2.1.6 Interruzioni . . . . .	39
2.1.7 Estensione del modello IO generale . . . . .	46
2.2 DMA (Direct Memory Access) . . . . .	46
2.2.1 Utilizzo effettivo in ASIM . . . . .	49
2.2.2 Implementazione in Motorola 68k . . . . .	51
2.3 USART . . . . .	56
2.3.1 Comunicazioni Sincrona ed Asincrona . . . . .	56
2.3.2 Intel 8251A . . . . .	58
2.3.3 USART in ASIM . . . . .	61
<b>3 Architettura dei processori</b>	<b>69</b>
3.1 Generalità sul processore . . . . .	69
3.2 Architettura dei Processori moderni . . . . .	71
3.2.1 Multi-Computer e Multi-Processore . . . . .	74

3.2.2	Speed Up . . . . .	74
3.2.3	Coerenza della memoria cache nelle architetture parallele . . . . .	76
3.3	Topologia di interconnessione . . . . .	79
3.4	Sistema Pipeline . . . . .	83
3.4.1	Modelli di sistemi pipeline . . . . .	85
3.4.2	Registri Intermedi . . . . .	86
3.4.3	Gestione dei Salti . . . . .	86
3.4.4	Gestione dei conflitti sui dati . . . . .	89
3.4.5	Gestione delle interruzioni . . . . .	92
3.5	Architetture Superscalari . . . . .	94
3.5.1	Gestione delle Collisioni . . . . .	94
<b>4</b>	<b>Memoria</b>	<b>99</b>
4.1	Memoria cache . . . . .	99
4.1.1	Politiche di Sostituzione . . . . .	100
4.1.2	Gestione dell'allineamento . . . . .	101
4.1.3	Set Associative . . . . .	101
4.1.4	Dimensionamento . . . . .	103
4.2	Memoria Virtuale . . . . .	105
4.2.1	Traduzione degli indirizzi . . . . .	105
4.2.2	Cache e Memoria Virtuale . . . . .	107
4.3	Struttura delle memorie . . . . .	109
4.3.1	Memorie Dinamiche . . . . .	109
4.3.2	Memorie Statiche . . . . .	110
<b>5</b>	<b>Processore MIPS</b>	<b>113</b>
5.1	Architettura MIPS . . . . .	113
5.2	Gestione delle interruzioni . . . . .	114
5.3	Pipeline nel MIPS . . . . .	115
5.3.1	Ipotesi di funzionamento . . . . .	115
5.3.2	Istruction Memory . . . . .	116
5.3.3	Register File (ID) . . . . .	116
5.3.4	ALU . . . . .	117
5.3.5	Data Memory . . . . .	117
5.3.6	Register File (WB) . . . . .	117
5.4	Gestione dei salti . . . . .	117
<b>6</b>	<b>Processori ARM</b>	<b>119</b>
6.1	Architettura ARM . . . . .	119
6.1.1	Modello di programmazione . . . . .	120
6.1.2	Gestione delle Eccezioni . . . . .	122
6.1.3	Differenze principali tra architetture . . . . .	123
6.2	STM32F4 . . . . .	124
6.2.1	CubeIDE . . . . .	124
6.2.2	Gestione mutua esclusione . . . . .	125
6.2.3	DMA . . . . .	128
6.3	BUS e protocolli . . . . .	128
6.3.1	I2C . . . . .	128

<b>7 RISC-V</b>	<b>131</b>
7.1 ABI e Istruzioni . . . . .	131
7.1.1 Jump and Link e Jump and Link Register . . . . .	132
7.1.2 Cheatsheet . . . . .	132
7.2 Livelli di privilegio e Interruzioni . . . . .	134
7.3 Estensioni . . . . .	135
7.4 Pseudo-istruzioni . . . . .	135
7.5 Calling convention . . . . .	136
7.5.1 Stack . . . . .	136
7.6 Confronto con il MIPS . . . . .	137
<b>8 Cloud-Edge</b>	<b>139</b>
8.1 Cloud Computing . . . . .	139
8.1.1 Architettura di un servizio Cloud . . . . .	139
8.1.2 Modelli di servizio . . . . .	140
8.1.3 Modelli di fornitura . . . . .	141
8.1.4 Rischi e Sicurezza . . . . .	141
8.2 IoT . . . . .	141
8.2.1 Architettura di un sistema IoT . . . . .	141
8.2.2 Sicurezza . . . . .	142
8.3 Edge Computing . . . . .	142
8.3.1 Architettura . . . . .	142
8.4 Fog Computing (cenni) . . . . .	143
8.5 Virtualizzazione . . . . .	143
8.5.1 Macchine virtuali . . . . .	144
8.5.2 Hypervisor . . . . .	145
8.5.3 Architetture Intel x86-64 . . . . .	146
<b>9 Esercitazioni</b>	<b>151</b>
9.1 Debunk esercizi PIA . . . . .	151
9.1.1 Esercizio 1 . . . . .	151
9.1.2 Considerazioni finali . . . . .	155
9.2 TAS (Test and Set) . . . . .	156
9.2.1 Esercizio 1 . . . . .	156
9.2.2 Esercizio 2 - Prova intercorso 2023 . . . . .	161
<b>A Appendice</b>	<b>163</b>
A.1 Asim e Asimtool . . . . .	163
A.1.1 Asimtool . . . . .	163
A.1.2 ASIM . . . . .	163
A.1.3 Esecuzione dei programmi . . . . .	164
A.1.4 Configurazione (file .cfg) . . . . .	164



# Introduzione



# Capitolo 1

## Richiami ed approfondimenti sui sistemi di elaborazione

In questo capitolo sarà affrontata principalmente la parte iniziale del corso che si occupa della scrittura di programmi utilizzando il linguaggio Motorola 68k. L'interesse non sarà volto alla tipologia di architettura, anche se a volte ci sarà il bisogno di specificarla, quanto al suo utilizzo effettivo nell'ambito del corso

### 1.1 Richiami di calcolatori elettronici

Il **Motorola 68k** è un microprocessore con architettura di tipo CISC, principalmente costituita da vari registri con diverse tipologie di utilizzo. Tali registri, però, non sono una caratteristica specifica dell'architettura del Motorola 68k, pertanto è buona norma introdurre l'architettura generale di vari tipologie di microprocessori. Tali caratteristiche, quindi, non sono intrinseche del solo Motorola 68k ma sono legate alla natura stessa delle architetture dei vari processori.

#### 1.1.1 Architettura generale

Quando si interagisce con le microarchitetture si lavora con vari tipologie di registri, la cui dimensione è descritta dal costruttore. I registri possono essere divisi principalmente in registri utilizzabili dal programmatore (o registri utilizzabili) e quelli che non possono essere utilizzati dal programmatore (o non utilizzabili). Tale suddivisione vi è poichè alcuni registri all'interno della microarchitettura vengono utilizzati per effettuare delle operazioni pilotate dalla CU. Tali registri sono tutti interni alla CPU (Ricordando che la cpu è formata da CU, ALU e registri interni). I registri interni utilizzabili dal programmatore sono anche chiamati **registri macchina (o registri general-purpose)** e possono essere di vario tipo:

- **Registri Dato:** Registri che sono utilizzati per conservare dati su cui si può operare con varie tipologie di interazioni;
- **Registri indirizzo:** Registri che sono utilizzati per conservare gli indirizzi;
- **Registri Speciali:** Registri utilizzabili dal programmatore ma con funzioni diverse, ovvero:

- **PC (Program Counter)**: memorizza l’indirizzo in memoria della prossima istruzione;
- **IR (Instruction Register)**: Contiene una copia dell’istruzione prelevata dalla memoria;
- **SR (Status Register)**: registro di stato che contiene varie tipologie di informazione come il caso di overflow, di azzeramento del risultato, di privilegio di esecuzione (se in **administrator mode** e quindi con l’accesso ad A’7).

Tra i registri a cui invece il programmatore non ha accesso vi sono:

- **MAR (Memory Access Register)**: Registro utilizzato dal processore per scrivere l’indirizzo di memoria a cui si vuole accedere;
- **MBR (Memory Buffer Register)**: Registro che contiene il dato che si è letto/-scritto in memoria (varia in base ai valori dei segnali di write e di read gestiti dalla CU);

#### 1.1.1.1 Memoria

La memoria, nell’analisi ad alto livello che stiamo conducendo, funziona come un blocco in cui sono memorizzati dati indicizzati. Per cui in base all’operazione della CU, questa modifica i valori di: MAR, MBR, segnale di read e di write. I dati possono essere acceduti in memoria in lettura/scrittura in modi diversi, e in generale la loro disposizione in memoria può variare secondo i paradigmi:

- **little-endian**: nella memorizzazione di un dato binario, riorganizzato in celle lunghe dei byte, i valori più significativi vengono memorizzati nelle celle con indirizzi più alti, mentre le meno significative in quelli con indirizzi più bassi;
- **big-endian**: nella memorizzazione di un dato binario, riorganizzato in celle lunghe 1 Byte, esso sarà memorizzato inserendo i bit più significativi nelle celle con indirizzo più basso mentre i bit meno significativi in quelle con indirizzo più alto;

Ai fini del funzionamento del processore è indifferente quale sia la tipologia di organizzazione dei dati in memoria. Ma bisogna averne una conoscenza perchè è importante far capire all’unità di controllo come deve trattare i dati che sta andando a prelevare. Nel caso del Motorola 68k ci troviamo a contatto con un processore con organizzazione big-endian. Oltre all’organizzazione dei dati un altro problema della gestione della memoria è l’**allineamento**, ovvero l’organizzazione delle celle di memoria in maniera indipendente. Nel caso del motorola 68k, sono consentiti gli accessi in memoria anche a porzioni diverse di byte, ma tali porzioni hanno il vincolo di poter essere obbligatoriamente o da 2 o da 4 Byte che iniziano in posizioni di memoria pari. Quindi si possono avere degli errori se si accede a posizioni di memoria dispari A volte quando si lavora con gli indirizzi di memoria si può andare anche in contro al problema dell’**aliasing**, ovvero un problema che riguarda l’accesso a locazioni di memoria sbagliate rispetto a quelle effettivamente desiderate (nel caso del 68k questo problema è dovuto al fatto che possiede 24 fili di bus ma i registri sono a 32 bit, per cui se prelevo un indirizzo dalla memoria, perdendo gli 8 bit più significativi, se faccio accesso all’indirizzo caricato, posso confonderlo con uno più piccolo).

La memoria quindi memorizza varie tipologie di dati e di istruzioni. Pertanto è corretto dividere la memoria in due parti principali:

- **Area codice (o area istruzioni):** Area dove sono contenuti i programmi e le istruzioni da eseguire, in generale più piccola;
- **Area dati:** Area dove sono memorizzati i dati.

### 1.1.2 Istruzioni

In generale le istruzioni offerte dalle architetture, sono formate da tre principali parti:

- **Codici operativi:** sono la porzione di un'istruzione in linguaggio macchina che indica quale operazione la CPU deve eseguire. Insieme ai suoi operandi, è ciò che compone un'istruzione macchina. Ogni architettura ha un proprio set di codici operativi e un formato di istruzione prestabilito;
- **Operandi sorgente:** Valori che possono essere memorizzati sia in dei registri interni che esterni (in base alla tipologia di operazione), su cui poi lavorano i codici operativi;
- **Operandi destinazione:** Registri o locazioni di memoria in cui si va ad inserire il dato prodotto dai codici operativi in base agli operandi sorgente ricevuti. Solitamente tale operando è indiretto poiché potrebbe, in una successiva operazione, diventare uno dei due operandi sorgente.

In generale un'istruzione è una composizione di bit conservata in memoria, in cui una parte identifica il **codice operativo** da effettuare, mentre l'altra specifica gli operandi, che nel caso di registri interni vengono identificati con il corrispettivo indirizzo, mentre nel caso di indirizzi esterni viene specificata la locazione di memoria da cui prelevare il dato. Le tipologie di indirizzamento che posso essere utilizzate per gli operandi sono:

- **Diretto:** Gli operandi presenti in memoria vengono acceduti specificando l'indirizzo di memoria in maniera plane;
- **Indiretto:** Gli operandi in memoria vengono acceduti in base al valore puntato da un registro indirizzo;
- **Implicito:** alcuni operandi sono dichiarati in maniera implicita all'interno dell'operando (Es. PUSH D3, pusha il valore in cima allo stack);
- **Immediato:** il dato da inserire in una determinata destinazione è direttamente inserito all'interno dell'istruzione (es. MOV #7,D0)
- **Spiazzamento:** la locazione di memoria a cui si vuole fare riferimento viene acceduta tramite uno spiazzamento rispetto ad un indirizzo di memoria;
- **Indice + spiazzamento:** la locazione di memoria a cui si vuole fare riferimento viene acceduta tramite un indice, che identifica una determinata zona della memoria rispetto ad un indirizzo (quindi tipo spiazzamento fisso) + un possibile ulteriore spiazzamento (per capire bene immaginarsi la memorizzazione e l'accesso a valori di una matrice);

Le istruzioni che vengono implementate per una particolare architettura vengono chiamate **ISA(Instruction Set Architecture)**, e che quindi possono essere o di tipo RISC o di tipo CISC in base alle scelte di progettazione:

- **CISC (Complex-Instruction-Set-Computer)**: Le istruzioni a disposizione del programmatore possono essere complesse, ovvero composte da più istruzioni elementari;
- **RISC (Reduced-Instruction-Set-Computer)**: L'architettura del microprocessore permette l'utilizzo di un set più ridotto di istruzioni, semplici e lineari, tali istruzioni a differenza del paradigma CISC, possono essere più veloci, ma non ripagano in termini di complessità per l'effettuazione di determinate operazioni.

Nel nostro caso noi utilizzeremo il Motorola 68k a 16/32 bit, dove tali bit indicano la grandezza dei registri, e di conseguenza, dei bus di collegamento tra essi. L'architettura di tale microprocessore è CISC, ma noi utilizzeremo un set ridotto di tutte le funzioni messe a disposizione dall'M68k, in modo da poter avere anche la confidenza giusta per affrontare, in futuro, anche tipologie di architetture RISC.

Per l'esecuzione di una particolare istruzione, il microprocessore deve, prima prelevarla dalla memoria. L'indirizzo dell' istruzione da prelevare è conservata dal Program Counter. Una volta prelevata l'istruzione dalla memoria tramite i registri MAR ed MBR, questa viene caricata nell'IR, che conserverà tale istruzione durante tutto il processo di decodifica ed esecuzione delle operazioni specificate.

Le istruzioni possono classificate in base al codice operativo nel seguente modo:

- **Trasferimento dati**: Codici che permettono di copiare un determinato valore in un registro (*MOVE*);
- **Aritmetiche**: effettua delle operazioni aritmetiche sugli operandi in ingresso e le memorizza in un operando destinazione. Solitamente le funzioni appartenenti a tale classe lavorano su numeri interi;
- **Logiche**: Operazioni che vengono effettuate sulle stringhe di bit degli operandi con una logica bit a bit. Effettuata l'operazione, il risultato viene inserito all'interno di un registro destinazione;
- **Scorrimento**: operazioni effettuate sugli operandi in ingresso che restituiscono lo scorrimento verso (destra o sinistra) dell'operando e lo memorizzano in una data locazione;
- **Confronto**: gli operandi vengono confrontati ed in base alla tipologia di controllo che voglio effettuare vado a controllare i valori dell'SR che mi interessano;
- **Salto**: Le istruzioni di salto permettono di cambiare il PC e quindi di eseguire (o rieseguire) delle porzioni di codice a cui puntano. Le istruzioni di salto possono essere di tipo condizionato (Bcc) o non condizionato (JMP). Nel primo caso l'istruzione di salto viene effettuata solo se è vera una data condizione, mentre nel secondo caso il salto viene effettuato senza il controllo di alcuna condizione;
- **Input/Output**: Alcune CPU sono dotate di apposite istruzioni per trasferire i dati da e verso le periferiche.

## 1.2 Motorola 68k

Una volta introdotti i concetti teorici e tecnologici propedeutici, si possono iniziare ad osservare i principali costrutti per la programmazione con il Motorola 68000. Conviene, quindi, non solo capire quali sono i codici per le varie tipologie di istruzioni specificate nel paragrafo [1.1.2], ma anche come costruire i principali componenti di un linguaggio di più alto livello (di cui si presuppone una minima conoscenza). I costrutti in questione possono essere: cicli, blocchi di decisione, ecc.

### 1.2.1 Registri General Purpose

Con *registri General-Purpose* (o registri macchina) intendiamo l'insieme dei registri messi a disposizione del programmatore per scrivere codice sfruttando i codici operativi dell'ISA. I registri General-Purpose a disposizione sono un insieme di registri a 32 bit:

- **Registri Dato:** Registri D0,D1,…,D7;
- **Registri Indirizzo:** Registri A0,A1,…,A7 ed A7' (questi ultimi puntano alla cima dello STACK: A7' è accessibile solo in modalità *supervisore*. In generale, ogni modalità di esecuzione dispone di un'area stack separata);
- **Status Register (SR):** Registro a 16 bit che contiene informazioni sia sui risultati delle operazioni dell'ALU che sullo stato dell'esecuzione (se in super-user o meno). Questo registro è sempre accessibile in lettura, mentre è accessibile in scrittura solo in modalità supervisore.

Bit	Sigla	Descrizione
15	T	Trace Enable
14	S	Supervisor State (1=Supervisor)
13-8	-	Non usati (riservati)
7	X	Extend (Estensione del Carry)
6	N	Negative
5	Z	Zero
4	V	Overflow
3	C	Carry
2-0	-	Non usati (riservati)

Tabella 1.1: Bit dello Status Register (SR) del Motorola 68k

### 1.2.2 Codici di Spostamento dati o indirizzi

Il principale codice operativo che nel motorola 68k permette lo spostamento dei dati è la **MOVE**, che può essere differentemente utilizzata in base ai seguenti parametri:

- **Lunghezza degli operandi:** solitamente specificata con la *dot notation* alla fine del comando;

- **Tipologia di indirizzamento:** La **MOVE** è una tra le poche operazioni che ammette tutte le tipologie di indirizzamento possibili (proprietà detta di **ortogonalità**); L'indirizzamento immediato per l'operando di destinazione è l'unico che può causare errore, ma è un'operazione che comunque non ha alcun senso.

La caratterizzazione del comando **MOVE** è la seguente:

```

1      *Indirizzamento diretto D1 = D0 o D1<-D0
2      MOVE D0,D1
3
4      *Indirizzamento indiretto (sorgente), diretto (destinazione)
5      *D0 = (AO), D0 = contenuto del registro in posizione AO
6      MOVE.W (AO),D0
7
8      *Indirizzamento Indiretto completo
9      MOVE.L (AO),(A1) *(istruzione non valida)
10     *o
11     MOVEA.L (AO),A1 *(istruzione valida)
12     *Indirizzamento immediato + indirizzamento Diretto
13     MOVE.L #14,D0
14
15     * Indirizzamento con spiazzamento su registro di indirizzo
16     MOVE.W 4(AO), D0      * D0 = valore all'indirizzo AO + 4
17
18     * Indirizzamento con spiazzamento e registro indice
19     MOVE.L 8(A0, D1.L), D2      * D2 = valore all'indirizzo AO + 8 +
20           D1
21
22     * Indirizzamento PC relativo con spiazzamento
23     MOVE.B 6(PC), D0      * D0 = byte situato 6 byte dopo il Program
24           Counter
25
26     * Push di un registro nello Stack
27     MOVE.L D0, -(A7)      * Salva D0 nello stack (decremento SP)
28
29     * Pop dallo Stack in un registro
30     MOVE.L (A7)+, D0      * Carica D0 con il valore in cima allo
31           stack (incremento SP)
32
33     * Push di un registro di indirizzo
34     MOVEA.L AO, -(A7)      * Salva AO nello stack
35
36     * Pop di un registro di indirizzo
37     MOVEA.L (A7)+, AO      * Carica AO con il valore in cima allo
38           stack
39
40     * Salvataggio multiplo nello stack
41     MOVEM.L D0-D3/A0-A2, -(A7)      * Salva piu' registri nello stack
42
43     * Ripristino multiplo dallo stack
44     MOVEM.L (A7)+, D0-D3/A0-A2      * Ripristina piu' registri dallo
45           stack

```

41                   *\* Ritorno da subroutine (equivalente a POP del PC)*  
42        RTS     *\* Ritorna dall'ultima subroutine chiamata*  
43

**Nota sull'indirizzamento indiretto completo:** Non è possibile utilizzare un'operazione del tipo **MOVE** (A0), (A1) perchè il bus dati dell'architettura del MC68k non supporta simultaneamente operazioni di lettura e scrittura, dunque servono due cicli di bus distinti. Occorre utilizzare dei registri dato interni di appoggio.

**Nota su MOVE.L D0, -(A7):** il pre-decremento nell'operazione di push nello stack è dovuto al fatto che, per costruzione, lo stack "cresce" verso il basso, quindi deve decrescere di 4 byte per ospitare quanto sto inserendo (quanto detto è scollegato dal fatto che il 68k sia big endian).

**Nota su RTS:** come concetto, RTS equivale a MOVE.L (A7)+, PC perché il valore puntato da A7 è quello \* in cima allo stack, in questo caso estraiamo quindi + (lo stack "sale"), in caso contrario avremmo avuto - (quando inseriamo nello stack).

Nel codice precedente sono da notare le seguenti notazioni:

- **.W:** tale parte del comando permette di capire che sto lavorando con operandi lunghi 16 bit (o 2 Byte) esse sono denotate Word. Nel caso in cui non sia specificato alcun markup, allora la move è da intendersi per soli 8 bit
- **.L:** tale parte del comando permette di capire che sto lavorando con operandi lunghi 32 bit (o 4 Byte) esse sono denotate Long Word
- **#14:** Vado ad identificare un valore immediato tramite il termine #<valore>, che sarà convertito in binario dal compilatore e poi inserito all'interno del programma, quindi integrato all'interno della zona istruzioni della mia memoria

### 1.2.3 Codici Aritmetici

Per il Motorola vi sono vari codici aritmetici, che però possono lavorare solo su valori interi e quindi non valori "reali" (o codificati in IEEE 754). I codici aritmetici più importanti, ed in generale, più presenti all'interno delle varie architetture sono i seguenti:

#### 1.2.3.1 Somma

```
1 *Operatore di somma
2 ADD #3, D0 *immediato + diretto, D0 = 3+D0
3 ADD.W #3,D0 *somma con specifica grandezza valore D0 = 3+D0
4 ADD.W D0,D1 *indirizzamenti diretti con D1 = D0+D1
5
6 ADDA.L #1,A0 *somma su registri di tipo indirizzo, somma 1
    direttamente al contenuto di A0, non alla locazione di
    memoria puntata da A0 (NON ci sonon le parentesi)
7
8 ADDQ.W #1,D0 *somma di un valore immediato tra 1 e 8
```

#### 1.2.3.2 Sottrazione

```
1 *Operatore di Sottrazione
2 SUB #3, D0 *immediato + diretto, D0=D0-3
3 SUB.W #3,D0 *sottrazione con specifica grandezza valore D0=
    D0-3
4 SUB.W D0,D1 *indirizzamenti diretti con D1=D1-D0
5
6 SUBA.L #1,A0 *Sottrazione su registri di tipo indirizzo
7
8 SUBQ.W #1,D0 *Sottrazione di un valore immediato tra 1 e 8
```

#### 1.2.3.3 Moltiplicazione

```
1 MULU #3, D0      * Moltiplicazione senza segno immediato +
    diretto, D0 = D0 * 3
2 MULU.W #3,D0     * Moltiplicazione senza segno con specifica
    grandezza valore, D0 = D0 * 3
3 MULU.W D0,D1     * Moltiplicazione senza segno con
    indirizzamento diretto, D1 = D1 * D0
4
5 MULS.W #3,D0     * Moltiplicazione con segno immediato +
    diretto, D0 = D0 * 3
6 MULS.W D0,D1     * Moltiplicazione con segno tra registri, D1
    = D1 * D0
```

#### 1.2.3.4 Divisione

```

1   DIVU #3, D0      * Divisione senza segno immediato + diretto,
2     DO = DO / 3 (quoziente in D0, resto in D1)
3   DIVU.W #3,D0      * Divisione senza segno con specifica
4     grandezza valore, DO = DO / 3
5   DIVU.W D0,D1      * Divisione senza segno con indirizzamento
6     diretto, D1 = D1 / DO

4
5   DIVS.W #3,D0      * Divisione con segno immediato + diretto,
6     DO = DO / 3
7   DIVS.W D0,D1      * Divisione con segno tra registri, D1 = D1
8     / DO

```

Come si nota dalle varie implementazioni dei codici aritmetici, questi non possono utilizzare tipologie di indirizzamento indiretto. Pertanto, prima di effettuare le operazioni aritmetiche, gli operandi di input devono essere caricati nei registri interni ed il risultato sarà poi memorizzato in uno dei due registri impiegati (Come visibile nei commenti dei vari comandi). Inoltre, sia MULU e MULS che DIVU e DIVS lavorano implicitamente su 16 bit nel 68k, e non esistono cose del tipo MULU.B oppure MULU.L (idem per gli altri 3 codici operativi).

#### 1.2.4 Codici di salto

I codici di salto possono essere di 3 tipologie principali nel motorola 68k:

- **Salti incondizionati:** Quando viene incontrata l'istruzione di salto, questa effettua il salto (cambiamento del PC) in maniera immediata e senza la verifica di alcuna condizione;
- **Salti condizionati:** I salti condizionati sono effettuati in base al verificarsi di una determinata condizione. Nel caso del motorola 68k la condizione è associata ai valori dei singoli bit dello Status Register (SR);
- **Salti a subroutine:** I salti a subroutine sono delle tipologie particolari di salto incondizionato, con l'unica differenza che l'indirizzo di memoria da cui si è saltati viene prima memorizzato nello stack e poi viene effettuato il salto. Tale operazione fa in modo che una volta eseguita la subroutine, il sistema possa ritornare al punto a cui si era fermato nel programma principale, sollevando il programmatore dall'onere di gestire manualmente il salvataggio dell'indirizzo di ritorno.

In generale, quando si definisce un codice di salto, bisogna prevedere anche il suo operando, che dal lato del programmatore può essere di principalmente 2 tipi:

- **Label:** dopo l'istruzione si fa riferimento ad una label all'interno del programma scritto che permette di evitare di andare a lavorare con indirizzi di memoria diretti (sarà il compilatore a configurarli ad-hoc);
- **Indirizzamento indiretto:** Tramite dei registri indirizzo si indica la locazione di memoria specifica a cui si vuole saltare;

#### 1.2.4.1 Salti non condizionati

I salti non condizionati in Motorola possono essere i seguenti:

```
1 BRA label      * Salto incondizionato alla label specificata  
2   (branch always)  
2 JMP address    * Salto incondizionato all'indirizzo  
3   specificato  
3 JMP (A0)       * Salto all'indirizzo contenuto in A0
```

Solitamente nelle varie applicazioni si preferisce utilizzare il comando BRA, poiché più semplice da ricordare in riferimento ai comandi di salto condizionato.

#### 1.2.4.2 Salti condizionati

I salti condizionati hanno una forma più o meno uguale in base a quello che si vuole fare. La loro forma nel caso del 68k è del tipo: Bcc. Dove la B sta per BRANCH mentre "cc" sono le componenti che permettono di distinguere la condizione da considerare rispetto ai valori dello SR. In particolare, i salti condizionati operano in base ai valori dei 5 bit del **CCR (Condition Code Register)**, che sono i 5 bit meno significativi dello SR riportati in tabella 1.1.

I principali comandi sono i seguenti:

```
1 BCS label      * Salto se Carry e' settato (C = 1)  
2 BCC label      * Salto se Carry e' azzerato (C = 0)  
3 BVS label      * Salto se Overflow e' settato (V = 1)  
4 BVC label      * Salto se Overflow e' azzerato (V = 0)  
5 BEQ label      * Salto se Zero e' settato (Z = 1)  
6 BNE label      * Salto se Zero e' azzerato (Z = 0)  
7 BMI label      * Salto se Negativo e' settato (N = 1)  
8 BPL label      * Salto se Positivo e' settato (N = 0)  
9  
10 BLT label     * Salto se Minore di (N XOR V = 1)  
11 BLE label     * Salto se Minore o uguale ((N XOR V) + Z = 1)  
12 BGT label     * Salto se Maggiore ((N XOR V) + Z = 0)  
13 BGE label     * Salto se Maggiore o uguale (N XOR V = 0)  
14  
15 BLS label     * Salto se Minore o uguale (C + Z = 1) (senza  
16   segno)  
   BHI label     * Salto se Maggiore (C + Z = 0) (senza segno)
```

#### 1.2.4.3 Salti a subroutine

I salti a subroutine sono una tipologia di salto incondizionato. Tali salti fanno parte di architetture CISC principalmente, poiché alcune architetture RISC non prevedono tali funzioni. La presenza di tali funzioni permette di non dover gestire l'indirizzo di ritorno dalla subroutine, o almeno non direttamente dal programmatore. Le istruzioni che in motorola 68k sono principalmente utilizzate per la chiamata a subroutine sono le seguenti:

```
1 * Salta a una subroutine e salva il ritorno nello stack
2 * (push nello stack)
3 BSR label
4 *Salta a una subroutine con indirizzo specifico o label
5 JSR address/label
6 * Ritorna dalla subroutine (estrae l'indirizzo di
7 * ritorno dallo stack, fa il pop dallo stack)
8 RTS
```

#### 1.2.5 Codici Logici

I codici logici sono operazioni che possono essere effettuate su degli operandi lavorando bit a bit. Da notare che per i codici logici si può avere l'indirizzamento indiretto solo per la sorgente e non per il destinatario. Esempi di codici logici sono i seguenti:

```
1 AND #3, D0      * AND bit a bit con valore immediato, DO=D0&3
2 AND.W D0, D1    * AND tra registri, D1=D1&D0
3 AND.L (A0), D0  * AND tra valore puntato da A0 e DO
4
5 OR #3, D0       * OR bit a bit con valore immediato, DO=D0/3
6 OR.W D0, D1     * OR tra registri, D1=D1/D0
7 OR.L (A0), D0   * OR tra valore in memoria puntato da A0 e DO
8
9 EOR #3, D0      * XOR bit a bit con valore immediato, DO=
10              DOXOR3
11 EOR.W D0, D1    * XOR tra registri, D1 = D1 XOR DO
12 EOR.L (A0), D0  * XOR tra valore puntato da A0 e DO
13 NOT D0         * Complemento bit a bit (negazione), DO = ~DO
```

## 1.2.6 Codici di Scorrimento

I codici di scorrimento permettono di effettuare delle operazioni di shift, che possono essere comode in alcune tipologie di operazioni

```

1   ASL #1, D0      * Shift aritmetico a sinistra di 1 bit
2   *(mantiene il segno)
3   ASR #1, D0      * Shift aritmetico a destra di 1 bit
4
5   LSL #1, D0      * Shift logico a sinistra di 1 bit (0 fill)
6   LSR #1, D0      * Shift logico a destra di 1 bit
7
8   ROL #1, D0      * Rotazione a sinistra di 1 bit
9   *(il bit piu' alto rientra da destra)
10  ROR #1, D0      * Rotazione a destra di 1 bit
11
12 ROXL #1, D0      * Rotazione a sinistra con Carry
13 ROXR #1, D0      * Rotazione a destra con Carry

```

Osserviamo che lo shift aritmetico a sinistra ha l'effetto di moltiplicare per 2 (se convertiamo da decimale a binario il valore del registro prima e dopo lo shift), tranne se perdi bit significativi o alteri il segno, e dopo lo shift aritmetico possono essere modificati i bit del CCR.

## 1.2.7 Codici di Confronto

I codici di confronto sono molto importanti, poiché in accoppiata con i salti condizionati permettono di costruire tutti i costrutti fondamentali che possiamo trovare anche nei linguaggi di alto livello

```

1   CMP #5,D0      * Confronta D0 con 5 (D0 - 5, senza
2   * modificare D0, aggiorna i flag)
3   CMP.W D0,D1     * Confronta D1 con D0 (D1 - D0,
4   * aggiorna solo i flag)
5   CMP.L (A0),D0    * Confronta D0 con il valore in
6   * memoria puntato da A0
7   CMPI #10,D0     * Confronto immediato con D0
8   * (D0 - 10, aggiorna solo i flag)
9   CMPL.L #1000,A0  * Confronta registro indirizzo A0 con 1000

```

Oltre a semplici comparazioni, solitamente, vi sono anche dei comandi che operano sui singoli registri. Non solo per il controllo di tali registri ma anche per l'effettuazione di eventuali operazioni che possono essere comode per una tipologia di interpretazione ad alto livello del codice

```

1   TST D0          * Testa D0 (controlla se e' zero o negativo,
2   * senza modificarlo)
3   TST.W (A0)       * Testa il valore in memoria puntato da A0
4   BTST #3, D0      * Testa il bit 3 di D0
5   * (imposta Z se il bit e' 0)
6   BTST #5, (A0)    * Testa il bit 5 della memoria puntata da A0
7   BSET #3, D0      * Imposta il bit 3 di D0 a 1

```

```

8   BSET #5, (AO)      * Imposta il bit 5 della memoria puntata da
    AO
9   BCLR #3, D0        * Azzerà il bit 3 di D0
10  BCLR #5, (AO)      * Azzerà il bit 5 del registro puntato da AO
11  BCHG #3, D0        * Inverte il bit 3 di D0 (0 -> 1, 1 -> 0)
12  BCHG #5, (AO)      * Inverte il bit 5 del registro puntato da AO
13  TAS D0             * Testa e imposta il bit piu' alto (7) di D0
14  TAS (AO)           * Testa e imposta il bit 7 del valore in
    memoria puntato da AO
15

```

## 1.2.8 Strutture sintattiche fondamentali

Dati i codici di **Salto** [1.2.4] e quelli di **Confronto** [1.2.7], si possono costruire quelle che sono le strutture sintattiche fondamentali

### 1.2.8.1 if-then-else

Per costruire il ciclo if-then-else bisogna per prima cosa comprendere quale sia la condizione, poichè bisognerà identificare:

- **Registro target:** In base a quale registro/operazione devo decretare la condizione?
- **Condizione:** Qual'è la condizione da rispettare?

Scelti questi due parametri allora sarò capace di capire quale codice cmp utilizzare ed in che modo, e quale tipologia di salto condizionato effettuare. Esempio di un classico If-Then:

```

1   MOVE.L D0, D1      * Carica valori nei registri
2                               * (supponiamo che D0 e D1 abbiano già
3                               * valori)
4   CMP.L D1, D0        * Confronta D0 con D1 (D0 - D1)
5   BGT    END_IF       * Se D0 > D1, salta al blocco then
6                               * (condizione = (D1 <= D0))
7   THEN                         * Codice interno all'IF
8   END_IF                      * Codice successivo...

```

Esempio di un If-Then-Else

```

1   MOVE.L D0, D1      * Carica valori nei registri (supponiamo
2                               che D0
3                               * e D1 abbiano già valori)
4   CMP.L D1, D0        * Confronta D0 con D1 (D0 - D1)
5   BGT    THEN         * Se D0 > D1, salta al blocco THEN
6   ELSE   MOVE.L #0, D2 * D2 = 0
7   BRA    END_IF       * Salta oltre il blocco THEN
8   THEN   MOVE.L #1, D2 * D2 = 1
9   END                         * Codice successivo...

```

### 1.2.8.2 Ciclo FOR

Come per l'if il ciclo for è composto principalmente da codici di **salto** e da codici di **confronto**. La struttura è molto simile a quella dell'If-Then-Else con l'eccezione della posizione dei vari salti. Precisamente la struttura di un ciclo for è la seguente:

```

1   MOVE.L #0, D0    * Inizializza il contatore D0 = 0
2
3 FOR CMP.L #10, D0 * Confronta D0 con 10
4     BGE      END    * Se D0 >= 10, esce dal ciclo
5
6             * Corpo del ciclo ...
7     ADDQ.L #1, D0    * Incrementa D0 di 1
8     BRA      FOR     * Ripete il ciclo
9 END

```

### 1.2.8.3 Ciclo While

Il ciclo while segue le regole del ciclo for solo con una condizione differente:

```

1 WHILE CMP.L #0, D1    * Confronta D1 con 0
2     BLE      END    * Se D1 <= 0, esce dal ciclo
3             * Corpo del ciclo
4     SUBQ.L #1, D1    * Decrementa D1 di 1
5     BRA      WHILE   * Ripete il ciclo
6 END

```

### 1.2.8.4 Chiamata a subroutine

Le chiamate a subroutine possono essere viste come una sorta di chiamate a funzione. Esse, quindi, possono avere sia degli operandi di ingresso che degli operandi di uscita. La "comunicazione" degli operandi con la subroutine può avvenire in due principali modi:

- **Con registri interni:** Gli operandi vengono caricati nei registri interni prima di chiamare la subroutine, che poi ci lavorerà sopra. Quindi i registri interni vengono utilizzati come una sorta di canale di comunicazione.
- **Con Stack:** Gli operandi sono allocati sullo stack, ciò richiede quindi una gestione anche del puntatore dello stack SP.

Un esempio di chiamata a subroutine con memorizzazione degli operandi nello stack è il seguente:

```

1   MOVE.L #5, D0        * Carica il primo operando in D0
2   MOVE.L #10, D1       * Carica il secondo operando in D1
3   MOVE.L D0, -(A7)    * Push del primo operando nello stack
4   MOVE.L D1, -(A7)    * Push del secondo operando
5   JSR     SUM_SUB     * Chiamata alla subroutine
6   MOVE.L (A7)+, D2    * prelievo del risultato dallo stack
7   ADDQ.L #8, A7       * Pulizia dello stack
8                   * (2 valori da 4 byte)
9                   * D2 ora contiene il risultato

```

```

10          * Codice successivo eventuale
11 SUM_SUB MOVE.L (A7)+, D0      * Pop del primo operando dallo stack
12           MOVE.L (A7)+, D1      * Pop del secondo operando
13           ADD.L D1, D0        * Somma D0 + D1, risultato in D0
14           MOVE.L D0, -(A7)    * Push del risultato nello stack
15           RTS                 * Ritorna al chiamante

```

Esempio di utilizzo dei registri interni, sia per passaggio operandi di ingresso che di uscita:

```

1      MOVE.L #5, D0          * Primo operando in D0
2      MOVE.L #10, D1         * Secondo operando in D1
3
4      JSR      SUM_REGS     * Chiamata alla subroutine
5      * Dopo il ritorno, il risultato e' in D0
6      * Codice successivo...
7 SUM_REGS:   ADD.L D1, D0    * Somma D0 + D1, risultato in D0
8      RTS                 * Ritorna al chiamante

```

### 1.2.9 Valutazione degli accessi in memoria

Utilizzando i comandi precedentemente presentati, il processore può accedere in memoria principale. Gli accessi in memoria dipendono fortemente dalla tipologia di architettura che ho adottato. In generale gli accessi in memoria possono avvenire per 2 tipologie di operazioni: Accesso in memoria per le Istruzioni (PI) o accesso in memoria Per le Operazioni (PO). Vediamo degli esempi per capire meglio di cosa si sta parlando:

Istruzione	PI	PO
MOVE.L D0,D1	1	0
MOVE.W D0,D1	1	0
MOVE.L #7,D1	3	0
MOVE.W (AO),(A1)	1	2
MOVE.W (AO),VAR	3	2

Tabella 1.2: Conteggio accessi per Architettura a 16 bit e VAR a 32 bit

Per contare gli accessi in memoria bisogna effettuare delle osservazioni in base alla parte che si sta analizzando. Per l'accesso **Per le Istruzioni (PI)** si ragiona sui seguenti accessi:

- **Prelievo dell'istruzione:** Un'operazione che non mancherà mai sarà sempre il prelievo dell'istruzione, che impone che il mio PI non potrà mai essere nullo;
- **Operandi immediati:** se nel mio comando ho degli operandi immediati, allora dovrò accedere anche altre volte alla memoria per il prelievo di tale operando. I miei accessi, per questo caso, sono dettati dalla lunghezza dell'operando rispetto ai miei bus disponibili. Per esempio, se sono in un'architettura a 16 bit devo prelevare un immediato considerato una WORD, allora il numero di accessi aggiuntivi per prelevare l'immediato è uguale a 1. Mentre se stessi lavorango con le Long Word (32 bit), allora il numero di accessi, a parità di architettura, sarà 2;

- **Variabili:** se sto utilizzando delle variabili, che indicano delle locazioni di memoria dirette, allora dovrò fare un numero di accessi alla memoria che mi permette di prelevare gli indirizzi (tali indirizzi sono lunghi tutti 32 bit). Pertanto con un architettura a 16 bit dovrò considerare sempre 2 accessi per ogni variabile per prelevare tali indirizzi;

Per l'accesso **Per gli Operandi (PO)**, i parametri risultano più o meno gli stessi, ad esclusione del prelievo istruzione e della variabile. Le operazioni che si contano per tale processo sono:

- **Indirizzamento Indiretto:** quando vado ad effettuare dei riferimenti a dei registri della memoria con indirizzamento indiretto allora devo prevedere un numero di accessi per il prelievo dell'operando. Quindi bisogna conteggiare il numero di accessi per il prelievo dell'operando in base alla sua lunghezza. Nel caso di architettura a 16 bit si avranno: 1 accesso per prelevare delle word (16 bit) e 2 accessi per prelevare le Long Word (32 bit)
- **Variabili:** Quando si utilizzano le variabili, oltre a prelevare gli indirizzi dalla "zona istruzioni" bisogna prelevare gli operandi. La conta del numero di accessi per il prelievo degli operandi è uguale al caso di indirizzamento indiretto

Nel caso degli accessi PO, si è prevista la considerazione per singoli operandi. Quindi in base al numero di operandi presenti, la loro lunghezza prefissata, il loro modo di indirizzamento, si riesce a comprendere (cumulando le specifiche), il numero di accessi che bisogna effettuare in memoria ed il motivo di tali accessi.

# Capitolo 2

## Gestione dei dispositivi di IO

I dispositivi di input/output, sono tutti quei dispositivi che si connettono alla classica architettura composta solo da processore e memoria centrale. Tra tali dispositivi rientrano: Memorie di massa (HDD e SSD), mouse, tastiera, sensori ecc.

### 2.1 Architettura generale di un dispositivo di I/O

In generale un dispositivo di I/O può essere visto come l'insieme di tre parti fondamentali:

- **Registri Dato, Stato e Controllo:** Tali registri sono quelli che interagiscono in maniera diretta con la CPU, e vengono utilizzati da quest'ultima per controllare e gestire le informazioni del dispositivo. Tali registri sono presenti internamente all'architettura del Calcolatore (ad esempio sulla scheda madre);
- **Sistema di adattamento:** Il sistema di adattamento adatta i segnali provenienti dal mondo esterno per essere letti o scritti nei registri di Dato, Stato e Controllo, e quindi permette di adattare l'attacco esterno (tipo l'USB che utilizza comunicazioni sequenziali), con la comunicazione parallela che il processore ha con i registri;
- **Mondo esterno:** Per mondo esterno si intende tutta la parte che interagisce con il dispositivo in maniera fisica, ed il dispositivo fisico stesso. (immagina una tastiera con il suo connettore USB).

Un esempio di dispositivo esterno è la memoria HDD. La memoria HDD ha difatti i tre registri di Dato, Stato e Controllo: nel processo di scrittura in memoria, la CPU modifica i suddetti registri. Mentre la CPU modifica tali dati, il sistema di adattamento converte i dati presenti in quei tre registri in movimenti della testina + scrittura, rispettando sempre i controlli dati dalla CPU. La scrittura/lettura dei dati tramite la testina e la testina stessa rappresentano, invece, il mondo esterno. Un altro esempio di periferica è la classica porta **UART**, che trasmette i suoi dati in serie, ma il suo controllo avviene in parallelo. Pertanto al suo interno avrà sia un timer per scandire il clock in base alla tipologia di comunicazione, e poi avrà un buffer parallelo-serie, che converte l'informazione da trasmettere in tanti bit seriali. Oltre alla parte parallelo-serie sarà anche dotato di una parte serie-parallelo, nel caso della ricezione.

### 2.1.1 Modalità di comunicazione

Le tipologie di collegamento che si possono avere tra un processore e le sue periferiche sono le seguenti:

- **Collegamento passivo:** la periferica e la CPU non condividono alcun tipo di comunicazione. Quindi la CPU presuppone che la periferica sia sempre pronta ed è quindi solo lei a decidere quando e come utilizzare i dati, anche se questi magari non sono pronti o ben processati;
- **Collegamento Sincrono:** La periferica e la CPU comunicano tramite un clock di riferimento per entrambe;
- **Collegamento con Handshacking:** L'handshacking è una modalità di comunicazione asincrona, poiché si sfruttano dei segnali di comunicazione tra la CPU e la periferica che permettono di sincronizzare le operazioni. Una classica implementazione è quella del segnale di *req* che viene alzato dal processore per far capire che vuole leggere e dall'*ACK* alzato dalla periferica che fa comprendere che il dato è pronto o che è stata presa in carico una determinata operazione richiesta. La differenza con la comunicazione sincrona è l'assenza di base dei tempi comune (clock);
- **Collegamento semisincrono:** La trasmissione dei dati è controllata da una base dei tempi comune, ma vengono usati anche segnali di sincronizzazione aggiuntivi (segnali di start e allineamento periodico) per mantenere la sincronizzazione tra trasmettitore e ricevitore;

### 2.1.2 Interfacciamento CPU e periferica

Per utilizzare le periferiche la CPU deve poter accedere ai registri di Dato, Stato e Controllo di tali periferiche. Le tipologie di interfacciamento che ci possono essere tra CPU e Periferica sono:

- **Memory Mapped I/O:** La CPU fa riferimento ai registri di Dato, Stato e Controllo di una periferica come se fossero dei registri in memoria;
- **I/O Mapped:** La CPU dispone di specifici codice operativi per interagire con le periferiche di I/O;

Nel nostro caso, il Motorola 68k è una tipologia di architettura **memory mapped**, e quindi la trattazione dei registri avviene mediante i codici operativi di accesso ai registri della memoria.

#### 2.1.2.1 Memory Mapped I/O

Nel caso di interfacciamento con una struttura Memory Mapped, l'accesso ai registri di una determinata periferica avvengono tramite bus di collegamento. Ciò rappresenta un limite nell'uso degli indirizzi, poiché quando faccio riferimento ad un registro di una periferica, tale indirizzo non deve appartenere al set di indirizzi della memoria centrale.

### 2.1.2.2 I/O Mapped

Nel caso di interfacciamento con una struttura I/O Mapped, l'accesso ai registri di una determinata periferica avviene mediante degli specifici comandi. Questo perchè le periferiche sono collegate a bus dedicati o hanno una gestione dedicata, che quindi differisce dalle comunicazioni che avvengono in generale all'interno dell'architettura al costo di avere meno modi di indirizzamento, dato che non si userà più la MOVE che è un codice operativo *ortogonale*.

### 2.1.2.3 Logiche di selezione

Nel processo di selezione di una periferica collegata alla cpu tramite un bus, vengono utilizzati degli indirizzi per realizzare una tra queste logiche:

- **Logica tristate:** è una logica che disconnette la periferica dal bus dati quando non riconosce il proprio indirizzo. In questo stato, la periferica non interferisce con le comunicazioni e ignora qualsiasi variazione presente sul bus. La logica tristate si basa sull'indirizzo interno assegnato alla periferica stessa. Si definisce *tristate* perchè un'uscita può trovarsi in tre stati logici:
  - **0:** livello basso;
  - **1:** livello alto;
  - **Alta impedenza:** praticamente disconnesso;
- **Logica Plug-and-play:** in questo caso, l'indirizzo della periferica non è fisso, ma viene selezionato dinamicamente tra un insieme di indirizzi disponibili. Questo consente di configurare automaticamente in fase di inizializzazione le periferiche senza dover intervenire manualmente per impostare l'indirizzo.

## 2.1.3 BUS

I bus sono responsabili del collegamento tra le varie componenti di un calcolatore (CPU, memoria e periferiche di I/O). In generale, le tipologie di bus variano in base all'applicazione e alla tecnologia utilizzata. I bus si contraddistinguono principalmente per la divisione che attuano sui loro collegamenti, ma in generale, le informazioni che vengono trasportate sono le stesse. Queste sono dipartite tra i vari collegamenti presenti in un BUS. I collegamenti generici che si possono identificare in un bus sono:

- **Alimentazione:** Collegamenti che principalmente comprendono la VCC (o più VCC), che sarebbero le tensioni di alimentazione delle componenti, ed il cavo di terra (o GND);
- **Dati:** Collegamenti che trasportano i dati che vengono scambiati tra i vari dispositivi;
- **Indirizzo:** Collegamenti che trasportano gli indirizzi che permettono la selezione dei dispositivi interessati o dei registri a cui si vuole accedere in lettura o scrittura;
- **Controllo:** Collegamenti che trasportano le informazioni inerenti alla tipologia di operazione che si vuole effettuare;

- **Stato:** Collegamenti che permettono il controllo di flusso e la segnalazione di eventuali conflitti o errori;

Data una tipologia di bus, può succedere che la periferica considerata non è compatibile con quella determinata tipologia di bus. Pertanto, si può considerare l'uso di un **adapter**, che mi permette di adattare il bus classico con la tipologia di attacco specifica per la mia periferica. Oltre tutto in alcuni casi, quando il dispositivo non permette la configurazione degli indirizzi, per evitare conflitti, l'adapter gestisce anche la gestione di tale indirizzo rispetto al sistema.

### 2.1.4 Driver

I driver sono dei software di basso livello che consentono di tradurre le richieste delle applicazioni per le periferiche in comandi specifici comprensibili dall'hardware. Ogni tipo di periferica necessita del proprio driver, e spesso è fornito dal costruttore dell'hardware stesso. Nella programmazione di un driver si possono seguire vari approcci in base al modo in cui vengono gestiti i tempi e le modalità di comunicazione con la CPU. L'approccio più semplice e primitivo è il **Polling**. Questo approccio consiste nel dare un primo segnale di controllo alla periferica e si *aspetta* uno specifico valore di stato per poter accedere al dato. Questo sistema è tuttavia inefficiente, poiché mentre la CPU aspetta la riposta della periferica si sprecano dei periodi di clock nei quali la CPU non esegue operazioni utili. Lo stato in cui la CPU permane senza eseguire delle operazioni utili è detto **Busy-waiting**. Un possibile codice di implementazione di questo tipo di approccio è [2.1]:

```

1      ORG      $8000  * Inizializzazione registri
2      MOVE.B #$00,C
3      MOVE.B #$00,S
4      MOVEA.L #VAR1,A0
5      MOVE.W #0,DO
6 FOR     CMP.W  #N,DO  * N byte da ricevere
7 BGE    FUORI
8 * Codice Driver:
9      MOVE.B #$01,C  * Set del controllo
10 L1     MOVE.B S,D1   * Inizio ciclo polling
11     AND.B #$80,D1
12 * Se il bit si e' alzato ho finito.
13 * Altrimenti continuo ad aspettare
14     BEQ    L1
15     MOVE.B D,(A0)+ * Inserisco il dato in memoria
16     MOVE.B #$00,C  * Reset segnale di Controllo
17     MOVE.B #$00,S  * Reset Segnale di stato
18     ADD.B #1,DO
19     BRA    FOR
20 FUORI
21     ORG      $8100
22 D      DS.B   1    * Registro dato
23 S      DS.B   1    * Registro Stato
24 C      DS.B   1    * Registro Controllo
25 N      EQU     5    * Dimensione array da trasferire
26 VAR1  DS.B   5    * Array effettivo di raccolta dati

```

Listing 2.1: Codice polling

Il codice [2.1] presenta però le seguenti criticità:

- **Mancata Generalizzazione:** Si vanno a considerare in maniera diretta i registri in memoria D,S,C che per l'implmentazione di un driver riutilizzabile non è una scelta corretta. Diverse istanze della stessa periferica (collegate quindi ad indirizzi diversi) avrebbero bisogno di un driver diverso;
- **Polling:** L'attesa che viene svolta all'interno di tale codice non permette al processore di eseguire altre operazioni prima di aver ricevuto tutti i caratteri;
- **Gestione dei malfunzionamenti:** Se la periferica ha un qualunque tipo di malfunzionamento e quindi non aggiorna mai il registro di stato, tale ciclo eseguirà all'infinito senza mai fermarsi.

Le due criticità possono essere risolte in diversi modi. La soluzione alla prima problematica è molto semplice: al posto di considerare i registri di Dato, Stato e Controllo in maniera diretta, possono essere considerati come registri indirizzo (Ai), a cui vado ad associare gli indirizzi di tali registri. Tali indirizzi poi vengono settati secondo un determinato criterio prima della chiamata al driver. Per ovviare al secondo problema c'è il bisogno di considerare le **interruzioni**. Per l'ultimo problema la soluzione è l'introduzione di **timer**, che permettono di capire quando un sistema sta impiegando un tempo più grande del dovuto per eseguire un'operazione, e ciò permette di poter gestire situazioni indesiderate di attesa indefinita.

### 2.1.5 PIA

La **PIA (Periferal Interface adapter)** è un dispositivo di comunicazione parallela ad 8 bit. Tale architettura è un dispositivo hardware che si posiziona tra la periferica e il processore stesso. Essa è costituita architetturalmente da due tipologie diverse di porto, il porto A ed il porto B.[2.1] Tali porti hanno dei registri che sono direzionali, quindi possono assumere una sola funzione (tra entrata ed uscita), in base alla loro configurazione. Facendo riferimento alla figura 2.1, osserviamo che il dispositivo si interfaccia al MC68K con un bus dati a 8 bit bidirezionale, tre linee di *chip select* CS, due linee di *register select* RS, una linea di controllo per indicare un'operazione di lettura e scrittura RW, due linee di *interrupt request* IRQ, una linea di *enable* e una linea di *reset*. Il bus dati a 8 bit bidirezionale permette il trasferimento di dati tra il processore e la PIA, e i collegamenti seguono la logica *three state*, ovvero permangono in uno stato di alta impedenza (off) tranne quando il processore performa una lettura su questi 8 bit. La linea enable costituisce l'unico segnale di timing fornito al dispositivo. La linea RW trasmette un segnale generato dal processore per controllare la direzione del trasferimento dati sul bus. Per selezionare il dispositivo, le linee CS0 e CS1 devono essere alte mentre la linea  $\overline{CS2}$  deve essere bassa. Le linee RS sono utilizzate come vedremo tra poco per selezionare registri interni alla PIA in combinazione con i control registers interni alla PIA. Le linee  $\overline{IRQA}$  e  $\overline{IRQB}$  sono utilizzate in accordo a quanto condfigurato nei registri di controllo per abbassarsi e segnalare alla cpu un'interruzione. Come si osserva in figura 2.1, lato periferica invece la PIA si interfaccia mediante due bus dati da 8 bit e quattro linee di interrupt/controllo. I due bus dati fanno riferimento al porto A e al porto B della medesima PIA. Per quanto riguarda il porto A, ognuna delle linee può essere programmata per essere input o output. Questo si può fare settando a 1 (output) o 0 (input) il corrispondente bit del *Data direction register* DDRA. Un discorso analogo può essere fatto

per il porto B. Per quanto riguarda le linee CA1 e CB1, sono linee di input utilizzate per settare i bit inerenti ai flag di interrupt dei control registers di A e B. CA2 è una linea che utilizzeremo con direzionalità output, e la funzionalità sarà chiarita dal control register del porto A. Un discorso analogo può essere fatto per CB2.

#### 2.1.5.1 Configurazione PIA

Dopo un segnale di  $\overline{RESET}$ , tutti i registri della PIA vengono azzerati e segue una fase di configurazione da parte della CPU. In totale, le locazioni accessibili internamente alla PIA sono 6 e illustrate nella tabella 2.1

Indirizzamento interno				
RS1	RS0	CRA2	CRB2	Registro selezionato
0	0	1	X	PRA
0	0	0	X	DRA
0	1	X	X	CRA
1	0	X	1	PRB
1	0	X	0	DRB
1	1	X	X	CRB

Tabella 2.1: Indirizzamento interno

Questi registri sono selezionabili dal processore mediante le linee RS1 ed RS0. I *Control Register* CRA e CRB permettono al processore di controllare l'operazione delle quattro linee di controllo periferica CA1, CA2, CB1, CB2. Permettono alla CPU inoltre di abilitare le linee di interruzione e controllare lo stato dei flag di interruzione. Gli 8 bit del CRA sono commentati in tabella 2.2:

CRA	7	6	5	4	3	2	1	0
	<i>IRQA1</i>	<i>IRQA2</i>	<i>controllo CA2</i>			<i>Accesso DRA</i>	<i>controllo CA1</i>	
CRB	7	6	5	4	3	2	1	0
	<i>IRQB1</i>	<i>IRQB2</i>	<i>controllo CB2</i>			<i>Accesso DRB</i>	<i>controllo CB1</i>	

Tabella 2.2: Control Registers

Vediamo nel dettaglio il significato di questi bit nella tabella 2.3 e nella tabella 2.4:

CRA		
Bit	Valore	Significato
0	0	Disabilita la transizione di IRQA allo stato low in seguito ad una transizione attiva di CA1
	1	Abilita la transizione di IRQA allo stato low in seguito ad una transizione attiva di CA1
1	0	IRQA1 (bit 7) settato alto su transizioni alto-basso di CA1
	1	IRQA1 (bit 7) settato alto su transizioni basso-alto di CA1
2	0	Selezionato DDRA
	1	Selezionato PRA
5,4,3	100	Operiamo secondo il procollo handshaking*
6	X	IRQA2 non è affetto dalle transizioni di CA2 quando b5=0
7	1	Bit di sola lettura, si alza su transizioni attive di CA1. Viene resettato da una lettura dell'output register A da parte della CPU

Tabella 2.3: dettaglio bit CRA

CRB		
Bit	Valore	Significato
0	0	Disabilita la transizione di IRQB allo stato low in seguito ad una transizione attiva di CB1
	1	Abilita la transizione di IRQB allo stato low in seguito ad una transizione attiva di CB1
1	0	IRQB1 (bit 7) settato alto su transizioni alto-basso di CB1
	1	IRQB1 (bit 7) settato alto su transizioni basso-alto di CB1
2	0	Selezionato DDRB
	1	Selezionato PRB
5,4,3	100	Operiamo secondo il procollo handshaking*
6	X	IRQB2 non è affetto dalle transizioni di CB2 quando b5=0
7	1	Bit di sola lettura, si alza su transizioni attive di CB1. Viene resettato da una lettura dell'output register B da parte della CPU

Tabella 2.4: dettaglio bit CRB

Un'importante osservazione va fatta sui bit 5,4,3. Durante il corso è stata affrontata il solo caso in cui  $b_5 = 1$ , ovvero la linea CA2 (CB2) è una linea di **output**, e il caso in cui  $b_4, b_3 = 00$ , ovvero:

- CA2 si abbassa dopo una lettura da parte della CPU dell'output register A, mentre si alza sulla successiva transizione attiva di CA1 come specificato nel bit 1.

- CB2 si abbassa dopo una scrittura da parte della CPU sull'output register B, mentre si alza sulla successiva transizione attiva di CB1 come specificato nel bit 1. b7 va innanzitutto resettato con una *lettura fittizia* sull'output register B.

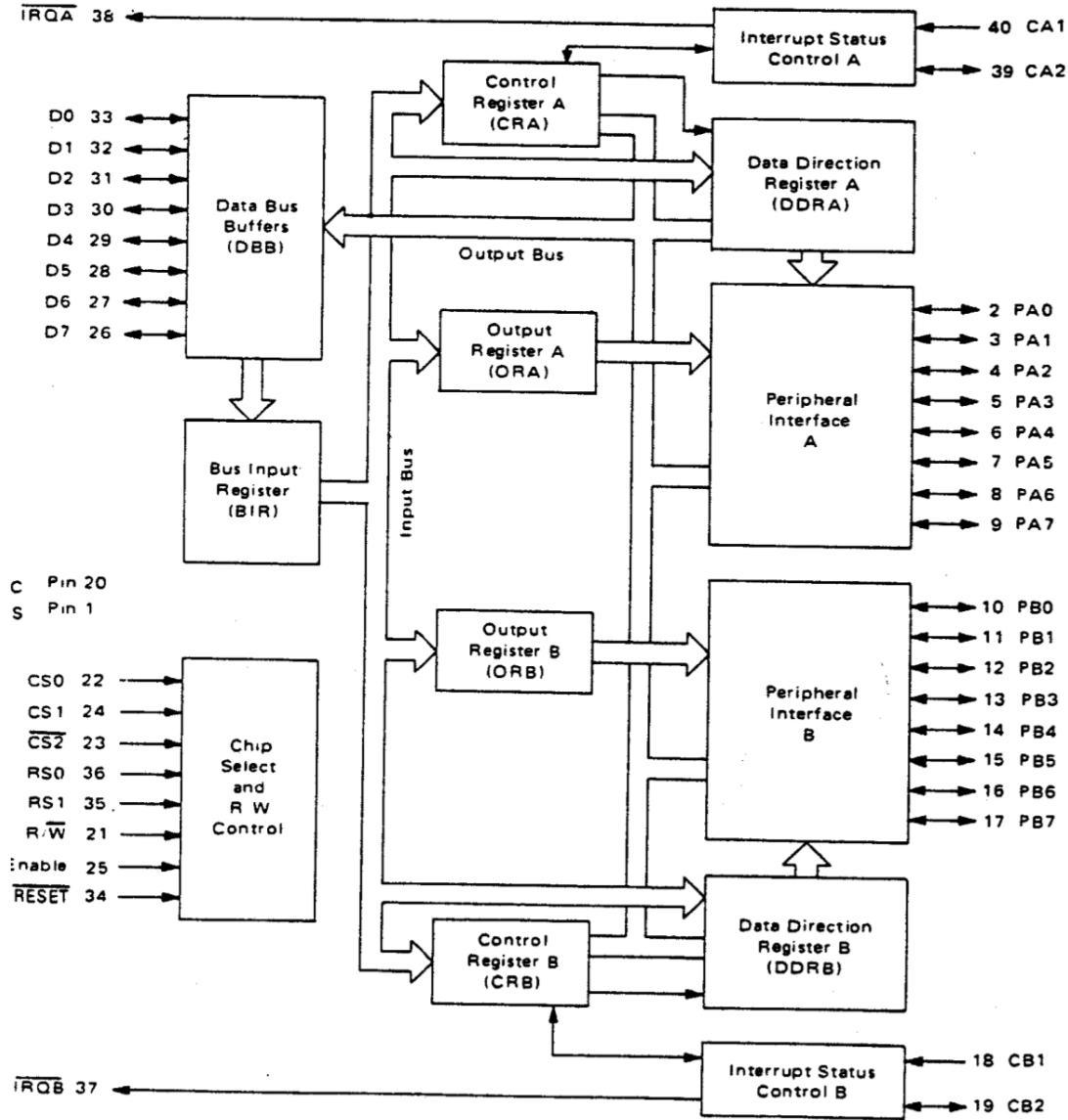


Figura 2.1: Architettura MC68A21CP

### 2.1.5.2 Comunicazione tramite PIA

Per far sì che le due architetture possano comunicare è quindi importante definire il collegamento e la direzionalità. Una possibile architettura di collegamento è quella visibile nella figura [2.2].

Durante il corso, gli esercizi sulla PIA sono stati effettuati sfruttando il simulatore ASIM. La PIA simulata in ASIM ha un comportamento quasi del tutto analogo a quello fin qui presentato per il componente reale. Prima di iniziare la configurazione, bisogna caricare su asim un file .cfg che definisce tutti i dispositivi della simulazione e il loro collegamento, compresa una memoria già inizializzata dove sono già mappati i registri appartenenti

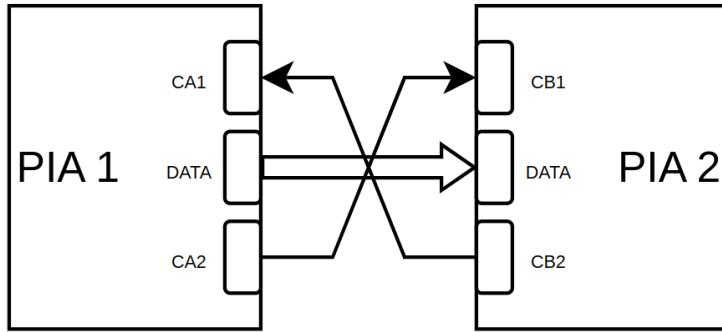


Figura 2.2: Collegamento tra due dispositivi tramite PIA

al modello di programmazione della PIA. A questo punto, è possibile eseguire il codice del *driver*, che accede al registro di controllo e al registro dati per attuare la logica di comunicazione. Riportiamo di seguito i casi in cui si attua una logica di funzionamento tramite polling (e busy waiting della CPU) e tramite interruzione.

```

1      *** LOGICA POLLING INVIO ***
2      ORG      $8200
3      MAIN     JSR      DVROUT
4      MOVEA.L #PIACB,A1
5      MOVEA.L #PIADB,A2
6      MOVEA.L #MSG, A0
7      MOVE     DIM, D0
8      CLR      D1
9      CLR      D2
10     INVIO    MOVE     (A2),D1
11     MOVE     (A0)+,(A2)
12     ADDQ    #1,D2
13     CICLO    MOVE     (A1), D1
14     ANDI    #%"10000000,D1
15     BEQ     CICLO
16     CMP     D2,D0
17     BNE     INVIO
18     LOOP    JMP     LOOP
19
20     DVROUT  MOVE     #0, PIACB
21     MOVE     #$FF, PIADB
22     MOVE     #%"00100100,PIACB
23     RTS
24
25     ORG      $8000
26     MSG      DC.B    1,2,3,4,5,6
27     DIM      DC.B    6
28     PIADB   EQU     $2006  * data register B
29     PIACB   EQU     $2007  * data register A

```

Questo è il codice relativo al dispositivo addetto alla trasmissione di un messaggio composto da 6 caratteri. DVROUT è il codice del DriVer relativo al porto B della pia con

bus dati in uscita (OUT). Questo codice viene subito invocato dal main: innanzitutto, azzera il control register del porto B, e così facendo azzera il bit 2 in modo che il prossimo accesso ad indirizzo pari a PIADB sarà il registro direzione in accordo alla tabella 2.1; dopodichè, il driver scrive la stringa 11111111 all'interno del direction register settando tutte le linee dati in uscita dal porto B; infine, il driver scrive la stringa 00100100 nel registro di controllo: b0,b1 = 00 -> propagazione delle interruzioni disabilitata (infatti stiamo analizzando il caso in cui il trasmettitore agisce in polling e non tramite interruzioni), b2=1 significa che il prossimo accesso ad indirizzo pari a PIADB sarà un accesso al registro dati, b5,b4,b3 = 100 significa che stiamo utilizzando il protocollo handshaking come specificato in tabella 2.4 (per questo è necessaria la lettura fittizia all'inizio). Dopo di che il main inizia la procedura di invio, che consiste: lettura su sul registro dati della PIA, in modo da abbassare il bit 7 di CRB; scrive sul registro dati del porto B, e quest'operazione abbassa CB2 e abbassa di conseguenza CA1, che nella transizione high-low scatena un'interruzione sul porto A dell'altra PIA che dovrà essere gestito, come vedremo in seguito, da un'operazione di lettura; A questo punto, il main inizia un ciclo improduttivo in cui aspetta che il b7 del control register diventi 1 in seguito ad un'operazione di lettura lato ricezione. Usciti dal ciclo, si controlla se restano ancora caratteri da trasmettere ed eventualmente si termina entrando nel ciclo caldo.

```

1 *** LOGICA POLLING RICEZIONE ***
2           ORG      $8200
3 MAIN      JSR      DVAIN
4           MOVE.W   SR,DO
5           ANDI.W   #$D8FF,DO
6           MOVE.W   DO,SR
7 LOOP     JMP      LOOP
8
9 DVAIN    MOVE     #0,PIACA
10          MOVE     #0,PIADA
11          MOVE     #00100101,PIACA
12          RTS
13
14          ORG      $8000
15 MSG      DS       6
16 DIM      DC       6
17 COUNT   DC       0
18 PIADA   EQU      $2004
19 PIACA   EQU      $2005
20
21          ORG      $8700
22 INT3    MOVE.L   A1,-(A7)
23          MOVE.L   A0,-(A7)
24          MOVE.L   DO,-(A7)
25
26          MOVEA.L  #PIADA,A1
27          MOVEA.L  #MSG,A0
28          MOVE     COUNT,DO
29          MOVE     (A1),(A0,DO)
30          ADD      #1,DO
31          MOVE     DO,COUNT
32

```

```

33      MOVE.L  (A7)+,D0
34      MOVE.L  (A7)+,A0
35      MOVE.L  (A7)+,A1
36      RTE

```

La ricezione di un carattere sulla PIA è gestita mediante interruzione di livello 3 (la PIA non supporta le int.vettorizzate in ASIM) che corrisponde al vettore 27 mappato in area ROM alla locazione \$6C: in tale locazione è contenuto l'indirizzo della ISR in RAM (\$8700). All'arrivo dell'interrupt la ISR acquisisce il carattere e lo salva in un'opportuna posizione in memoria. Il main chiama immediatamente il Driver DVAIN, che accede al direction register del porto A, imposta le linee del data register A come input (stringa di zero), imposta il registro di controllo con la stringa 00100101: b0,b1 = 01 -> propagazione delle interruzioni abilitata, b2=1 significa che il prossimo accesso ad indirizzo pari a PIADA sarà un accesso al registro dati, b5,b4,b3 = 100 significa che stiamo utilizzando il protocollo handshaking come specificato in tabella 2.3. Dopodichè, il main imposta lo status register in modo tale da permettere ogni tipo di interruzione, ed entra in un ciclo caldo dove attende un'interruzione. La ISR associata a quest'interruzione salva i registri sullo stack, dopodichè acquisisce il carattere con una lettura dal registro dati e questo fa abbassare il b7 di CRA e CA2. CA2 basso comporta una transizione alto-basso di CB1 che corrisponde all'alzarsi del flag CRB7 (data ack). Dopodichè incrementa i contatori, ripristina i registri e termina. Il problema di questo driver è la **busy wait** in fase di trasmissione, motivo per cui introduciamo la comunicazione parallela tramite PIA sfruttando il meccanismo di interruzione anche in fase di trasmissione.

```

1 *** LOGICA INTERRUPT INVIO ***
2      ORG      $8200
3 MAIN    JSR      DVROUT
4      MOVE.W   SR,D0
5      ANDI.W   #$D8FF,D0
6      MOVE.W   D0,SR
7 *** INVIO PRIMO CARATTERE ***
8      MOVE.L   PIADB,A0
9      MOVE.L   PIACB,A1
10     MOVE.L   MSG,A2
11     MOVE.L   (A0),D0
12     MOVE.L   (A2),(A0)
13     MOVE     #1,COUNT
14 LOOP    JMP     LOOP
15
16 DVROUT  MOVE     #0,PIACB
17      MOVE     #$FF,PIADB
18      MOVE     #%"00100101,PIACB
19      RTS
20
21      ORG      $8000
22 MSG     DC.B    1,2,3,4,5,6
23 DIM     DC.B    6
24 COUNT   DC.B    0
25 PIADB   EQU     $2006  * data register B
26 PIACB   EQU     $2007  * data register A

```

```

27
28      ORG      $8800
29 INT4    MOVE.L   A1,-(A7)
30          MOVE.L   A0,-(A7)
31          MOVE.L   D0,-(A7)
32
33          MOVEA.L #PIADB,A1
34          MOVEA.L #MSG,A0
35          MOVE     DIM,D0
36          MOVE     COUNT,D1
37
38          CMP      D1,D0
39          BEQ      FINE
40 INVIO   MOVE     (A0,D1),(A1)
41          ADDQ     #1,D1
42          MOVE     D1,COUNT
43 FINE    MOVE.L   (A7)+,D0
44          MOVE.L   (A7)+,A0
45          MOVE.L   (A7)+,A1
46          RTE

```

La configurazione effettuata in DVOUT differisce da quella presentata in precedenza solo dalla stringa scritta nel CRB, in quanto le transizioni attive di CB1 scatenano un'interruzione sul CRB7. Il main attiva dunque le interruzioni agendo sullo status register, e in seguito procede con un primo invio (altrimenti il sistema resterebbe in stallo). Osserviamo che è necessaria sempre la lettura fittizia sul registro dati per abbassare CRB7, in quanto non sappiamo a priori in che stato si trovi. Il main scrive dunque un carattere sul registro dati, e questo provoca la transizione alto-basso di CB2 che a sua volta provoca la transizione alto-basso di CA1, che quindi setta a 1 il flag CRA7 e scatena un'interruzione che gestisce il dispositivo ricevitore: questo farà una lettura sul registro dati del porto A, e questo comporterà l'abbassamento di CRA7 e di CA2, che comporterà l'abbassamento di CB1, che scatenerà un'interruzione su CRB7. Questa interruzione sarà gestita dalla ISR4.

```

1 *** LOGICA INTERRUPT RICEZIONE ***
2
3      ORG      $8200
4 MAIN    JSR      DVAIN
5          MOVE.W   SR,D0
6          ANDI.W   #$D8FF,D0
7          MOVE.W   D0,SR
8 LOOP    JMP      LOOP
9
10 DVAIN   MOVE     #0,PIACA
11          MOVE     #$00,PIADA
12          MOVE     #%-00100101,PIACB
13          RTS
14
15 MSG     DS.B    6
16 DIM     DC.B    6
17 COUNT   DC.B    0

```

```

18 PIADB EQU $2006 * data register B
19 PIACB EQU $2007 * data register A
20
21 ORG $8700
22 INT3 MOVE.W A1,-(A7)
23 MOVE.W A0,-(A7)
24 MOVE.W D0,-(A7)
25 MOVE.W D1,-(A7)
26 MOVE DIM,D0
27 MOVE COUNT,D1
28 MOVEA.L #PIADA,A1
29 MOVEA.L #MSG,A0
30 MOVE (A1),(A0,D0)
31 ADDQ #1,D1
32 MOVE D1,COUNT
33
34 MOVE.L (A7)+,D1
35 MOVE.L (A7)+,D0
36 MOVE.L (A7)+,A0
37 MOVE.L (A7)+,A1
38 RTE

```

In questo esempio non cambia nulla dal caso precedente. Per ulteriori esercizi ed esempi, consultare il capitolo 9.

### 2.1.5.3 Tempificazione comunicazione tramite PIA

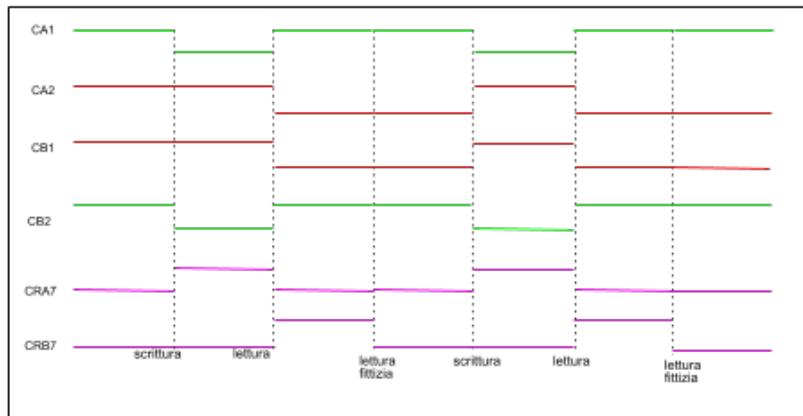


Figura 2.3: tempificazione PIA

Come si può osservare dalla figura 2.3, è possibile schematizzare l'andamento temporale dello stato logico dei segnali coinvolti nella comunicazione:

- **fase 1:** Il primo sistema scrive un carattere sul registro dati del porto B, e, per quanto specificato nei bit[3:5] del registro di controllo del porto B, si abbassa CB2 che è cortocircuitato a CA1 come specificato in figura 2.2. Una transizione alto-basso di CA1 setta a 1 il bit CRA7 e si scatena un'interruzione per il secondo sistema, che attiva la ISR relativa alla lettura del dato;

- **fase 2:** La ISR del secondo sistema legge il dato nel registro dati della PIA porto A e immediatamente si abbassa il flag CRA7 e CA2, cortocircuitato a CB1, e la transizione alto-basso di CB1 setta alto il bit CRB7 del primo sistema e questo scatena un'interruzione che verrà gestita dalla ISR del primo sistema per l'invio di un nuovo dato. La transizione alto basso di CB1 fa tornare alto CB2 e di conseguenza anche CA1 in accordo a quanto specificato nel registro di controllo;
- **fase 3:** Lettura fittizia per assicurarsi che CRB7 sia nello stato low;
- **fase 4:** Inizia una nuova scrittura a carico della ISR del primo sistema, che scrive un carattere nel registro dati del porto B della PIA, e questo fa di nuovo abbassare CB2 e di conseguenza anche CA1. La transizione di CA1 solleva un'interruzione come visto in precedenza che sarà gestita dalla ISR del secondo sistema che leggerà il carattere. La transizione di CA1 alza anche CA2 e CB1 come specificato nel registro di controllo CRA[3:5].
- **fase 5-6:** Si procede in maniera analoga a quanto visto nelle fasi 2 e 3.

## 2.1.6 Interruzioni

L'interruzione è un evento che cambia la normale esecuzione di un programma per fargli eseguire prima del codice specifico per la gestione di quella determinata condizione [2.4]. In generale non è corretto parlare solo di interruzioni, poichè tale termine non comprende anche il caso in cui le interruzioni vengano scatenate dall'interno per casistiche particolari. Difatti è più corretto fare la seguente suddivisione:

- **Interruzioni:** Segnali che sono a contatto con le periferiche e che permettono alla CPU di interrompersi e di eseguire il codice per la gestione della comunicazione con quella data interfaccia. Le interruzioni sono scatenate, quindi, dal dispositivo che vuole interagire con la CPU;
- **Eccezioni:** Funzionano come le interruzioni, con la differenza che vengono scatenate internamente rispetto al processore, quindi non vengono generate dai dispositivi ma dal programma in esecuzione stesso: tale condizione fa eseguire comunque una ISR, con l'obiettivo di dover gestire particolari casistiche (es. divisione per 0).

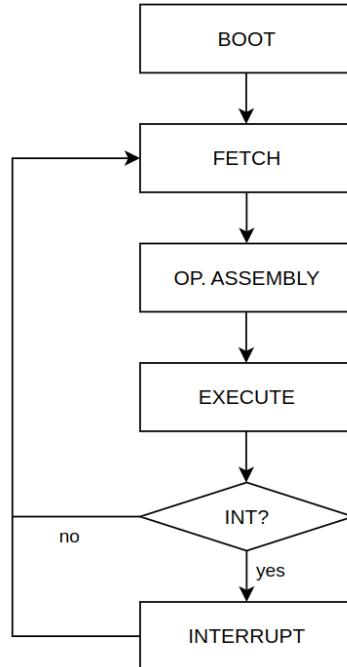


Figura 2.4: Ciclo di esecuzione con interrupt

Quando il processore rileva un'interruzione, interrompe la normale esecuzione e salta ad una funzione speciale denominata *Interrupt Service Routine* ISR. Tale situazione, quindi, ferma il sistema dalla sua normale esecuzione del programma per dare priorità alla gestione dell'interruzione. Questo apre molti dubbi su come gestire lo stato in cui si trova la macchina, poichè se quando torno dalla ISR, ho cambiato qualche registro significativo si potrebbe compromettere il normale funzionamento del programma. In generale i due registri che richiedono l'obbligo di essere salvati sono i registri: **SR(Status register)** e il **PC(Program Counter)**. In generale, i registri che vado a salvare in questo passaggio sono anche detti: **Descrittori di processo**, e descrivono lo stato di funzionamento del

processore quando è stato prelazionato dalla ISR. Ciò permette di proseguire ancora con la normale esecuzione del programma precedentemente in esecuzione prima del salto a subroutine. Analizziamo nel dettaglio il processo di gestione di un'interruzione:

- Esecuzione normale;
- Servizio dell'interruzione:
  - Salvataggio contesto Hardware;
  - Identificazione dispositivo richiedente;
  - Salto all'entry point della ISR;
  - salvataggio del contesto Software;
  - Servizio dell'interruzione;
  - Ripristino contesto software;
  - Ripristino contesto Hadware;
- Ripresa esecuzione normale.

I registri PC e SR devono essere salvati tramite un circuito apposito presente sul processore in ogni caso. La restante parte dello stato di esecuzione viene salvata mediante push sullo stack da parte della stessa ISR. Il salvataggio di questi due registri fondamentali non può avvenire tramite software perché deve essere un'operazione atomica, non interrompibile da interruzioni di priorità più elevata. Gli aspetti da gestire sono molteplici: Occorre definire delle **priorità** nel caso di interrupt molteplici concomitanti, occorre gestire il caso degli interrupt innestati, e infine occorre gestire la presenza di più periferiche che condividono la stessa linea fisica di interruzione.

#### 2.1.6.1 Gestione delle Interruzioni

Nel caso del MC68K, sono presenti 3 linee fisiche di interruzione che codificano sia la presenza di interruzione che il livello di priorità (da 0 a 7 livelli) come mostrato in figura 2.5.



Figura 2.5: Interruzioni nel MC68K

Se il processore sta eseguendo una ISR, lo status register riporterà i 3 bit che codificano la priorità dell'ISR correntemente eseguita. Se si presenta una interrupt con livello di priorità minore o uguale a quella correntemente eseguita, la richiesta di interruzione non viene accettata. Tuttavia, più periferiche possono condividere lo stesso livello di priorità. All'accettazione dell'interruzione, la CPU aspetta sul BUS di sistema una stringa di 8 bit denominata **vettore**, inviato dalla periferica (diverso per ogni periferica), e questo è usato per identificare la periferica. Identificare la periferica significa individuare la corretta ISR che può gestire l'interruzione. Il vettore letto sul bus altro non è che un

indice della **tabella delle interruzioni** gestita dalla CPU. Questa tabella è posta in memoria a partire dall'indirizzo 0 e contiene 256 locazioni di 4 byte. Ogni locazione contiene l'indirizzo della prima istruzione di una differente ISR.

Vector Number(s)	Range	Description
0	0	Reset (SSP)
1	1	Reset (PC)
16–23	16–23	Unassigned, Reserved
25	25	Level 1 Autovector
31	31	Level 7 Autovector
32–47	32–47	Trap #0–15 Instructions
64–255	64–255	User Device Interrupts

Per avere interruzioni particolarmente veloci, è possibile evitare il passaggio di scrittura del vettore sul bus di sistema. Un apposito segnale avvisa il processore che la periferica non presenterà il vettore sul bus, e in questo caso si parla di interrupt **autovettorizzate**. La particolare ISR è derivata direttamente dal livello di priorità, e quindi si può sfruttare l'area della tabella riservata agli autovettori (25–31).

#### 2.1.6.2 PIC

In generale, nel caso di sistema **vettorizzato**, viene in aiuto il componente **PIC (Programmable Interrupt Controller)** [2.6]. Il PIC è un dispositivo che permette di arricchire le modalità di gestione delle interruzioni. Grazie alla programmazione di questo oggetto, possiamo assegnare alle varie periferiche non una sola linea di interruzione con una specifica ISR, ma possiamo esplorare tutto il vettore delle interruzioni, che in teoria è costituito da 256 locazioni. In sostanza, il PIC permette di usare **interrupt vettorizzate**, ovvero il dispositivo fornisce sul data bus un vettore di 8 bit che rappresenta l'indice all'interno della tabella delle interruzioni corrispondente all'indirizzo della corretta ISR. Nel M68k questo protocollo è simulato con il PIC: Il dispositivo non scrive sul data bus il vettore di 8 bit, ma comunica l'interruzione al PIC che si occuperà di capire qual è il vettore corrispondente al dispositivo interrotto. Il PIC estende la gestione delle interruzioni del processore M68K introducendo nuove funzionalità, come la gestione prioritaria, la mascheratura delle interruzioni e le linee di interrupt. Il dispositivo ha in uscita verso il processore una linea di interruzione INT e una linea di INTA (acknowledgement), mentre ha in ingresso 8 linee di interruzioni differenti, a priorità decrescente (0 massima, 7 minima).

Nella trattazione del PIC faremo riferimento ad un particolare chip, lo **82C59A** 2.7, per la sua particolare compatibilità. Un singolo PIC può gestire fino a 8 livelli di priorità, ma è possibile creare un sistema di PIC in cascata per arrivare a gestire fino a 64 livelli di priorità.

Il PIC prevede due modalità di gestione delle priorità:

- **Fully nested mode:** le richieste di interruzione sono ordinate secondo uno schema a priorità fissa che va da IR0 a IR7, con IR0 la linea più prioritaria;
- **Rotate mode:** Schema prioritario a rotazione (Round robin), ovvero la linea di interruzione più prioritaria appena servita diventa la meno prioritaria dopo il servizio. Da programma è possibile configurare il livello di priorità più basso.

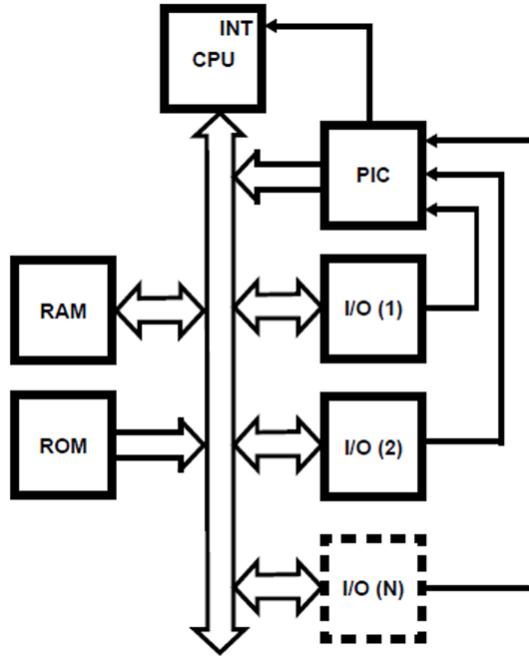


Figura 2.6: PIC

Osservando i componenti interni in figura 2.7, notiamo i seguenti componenti:

- **IRR:** Interrupt request register, ovvero un registro di 8 bit per ciascuna linea di interruzione in ingresso, finalizzato a memorizzare le richieste di interruzione;
- **ISR:** In service register, registro che memorizza i livelli di interruzione correntemente serviti;
- **Priority Resolver:** componente che determina le priorità delle richieste memorizzate dall'IRR e decide *quale* istruzione deve essere servita, ovvero quale bit dell' ISR settare;
- **IMR:** Interrupt Mask Register, registro che contiene una maschera che serve a disabilitare selettivamente una o più linee di interruzione;
- **Blocco R/W control:** Il blocco riceve i comandi dalla CPU per la configurazione del dispositivo e permette di “leggerne” lo stato interno;
- **Cascade buffer comparator:** viene utilizzato quando il PIC è collegato ad altri PIC in cascata, e le linee connesse a questo blocco servono a regolare la logica PIC master/slave.

Procediamo illustrando la modalità di funzionamento (figura 2.8): Inizialmente, uno o più dispositivi connessi al PIC inviano un'interruzione sulla linea alla quale sono connessi. La richiesta che arriva sulla linea  $i$ -esima, pone a 1 l' $i$ -esimo bit del registro **IRR**. A questo punto, il **priority resolver** decide quale dispositivo deve essere servito sulla base delle priorità associate ad ogni dispositivo e in base al contenuto dell'**IMR**, e inoltre una richiesta di interruzione alla CPU (o ad un altro PIC cui è connesso in cascata) tramite la linea INT. La CPU a questo punto invia un ACK sulla linea INTA per segnalare

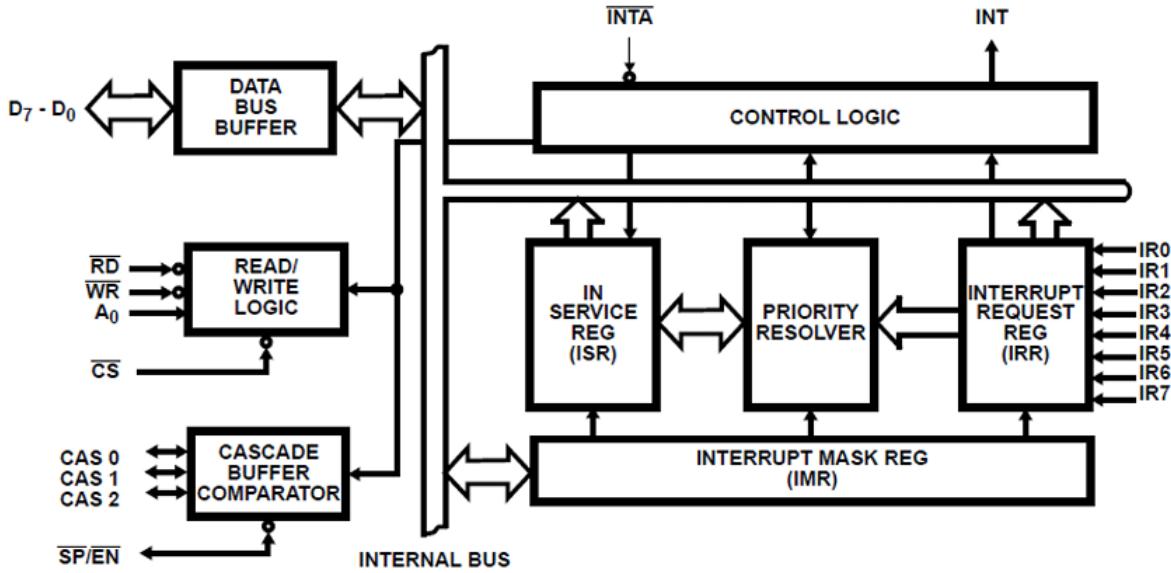


Figura 2.7: 82C59A

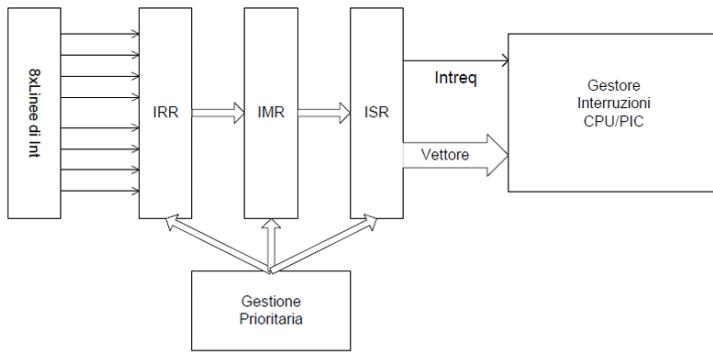


Figura 2.8: Schema PIC

eventualmente la *disponibilità* a servire l'interruzione. Una volta ricevuto l'ACK, il PIC setta il bit del registro **ISR** corrispondente alla linea interrompente a maggiore priorità, e resetta il corrispondente bit del registro **IRR**. Il PIC a questo punto trasmette sul *bus dati* il vettore relativo al dispositivo interrompente e il bit del registro ISR corrispondente al dispositivo viene resettato. Questo ultimo reset può avvenire in due modalità:

- *automaticamente*, se è stata selezionata la modalità *automatic end of interrupt*;
- *manualmente*, settando un bit opportuno EOI (End of interrupt) tramite una parola di controllo appena prima di uscire dalla ISR. È importante osservare che nel componente simulato in ASIM funziona solo questa modalità.

Attraverso i blocchi **Data Bus Buffer** (linee D0-D7) e **R/W logic** (linee  **$\overline{RD}$** ,  **$\overline{WR}$** , **A<sub>0</sub>**,  **$\overline{CS}$** ) è possibile accettare due tipi di command word proveniente dalla CPU:

- **ICWs**: Initialization Command Words, che servono ad *inizializzare* il dispositivo con una sequenza di minimo due e massimo quattro comandi di inizializzazione.

- **ICW1**: identificato da (A0=0,D4=1), inizia la sequenza di inizializzazione resettando il dispositivo e impostando alcuni parametri di configurazione, attraverso gli altri bit Dx.
- **ICW2**: segue la ICW1, identificato da A0=1, specifica i bit di indirizzo per localizzare la tabella dei vettori delle interruzioni in base al tipo di processore selezionato con la ICW1 ( 8080/85 o 80C86/88/286);
- **ICW3**: Identificata da A0=1, questa parola viene letta solo quando c’è più di un 82C59A in cascata, informazione acquisita attraverso la ICW1.
- **ICW4**: Identificata da A0=1, viene usato se in ICW1 è stato specificato IC4=1. Consente di specificare se il dispositivo deve funzionare in special fully nested mode o in buffered mode, il tipo di processore con cui deve interfacciarsi, e se bisogna abilitare l’automatic end of interrupt (**AEOI**).
- **OCWs**: Operation Command Words, servono a configurare la modalità di funzionamento. Dopo aver inizializzato il dispositivo con le ICW, esso è pronto ad accettare richieste di interruzione e può essere configurato con ulteriori parametri operativi:
  - **OCW1**: Setta e resetta i bit della maschera nell’IMR.
  - **OCW2**: consente di configurare il funzionamento del dispositivo per quanto riguarda la modalità *end of interrupt* (permette di resettare in maniera programmatica il bit nel registro ISR) e *rotate* (permette di specificare il meccanismo di rotazione delle priorità man mano che le interrupt su una linea vengono servite).
  - **OCW3**: consente di attivare/disattivare la special mask mode, in cui è possibile mascherare “temporaneamente” uno specifico livello di richiesta senza avere effetto sui livelli più bassi o più alti. Il livello da mascherare è quello specificato in precedenza nella OCW1.

Dal punto di vista esercitativo, al corso è stato presentato un componente da utilizzare nel framework ASIM che rappresenta una versione estremamente semplificata del PIC, il cui schema logico è raffigurato in figura 2.9. Il device simulato presenta 8 linee di richiesta, a ciascuna delle quali è associata una priorità fissa (modalità *fully nested*), a cui possono essere connessi fino a 8 dispositivi con 8 livelli di priorità, stabiliti mediante lo schema *daisy chain*. Il modello di programmazione del dispositivo occupa due locazioni consecutive di memoria e prevede 5 registri accessibili al programmatore.

<b>IMR</b>	(Interrupt Mask Register) permette di mascherare singolarmente le interruzioni
<b>IRR</b>	(Interrupt Request Register): memorizza i segnali di interruzione
<b>ISR</b>	(In Service Register): presenta al gestore delle interruzioni i segnali interrompenti in modo ordinato rispetto alla maschera e alla priorità corrente
<b>CTRL</b>	(Control Register): permettere di controllare il comportamento del dispositivo
<b>TR</b>	(Type Register): memorizza la parte base dell’indirizzo del vettore

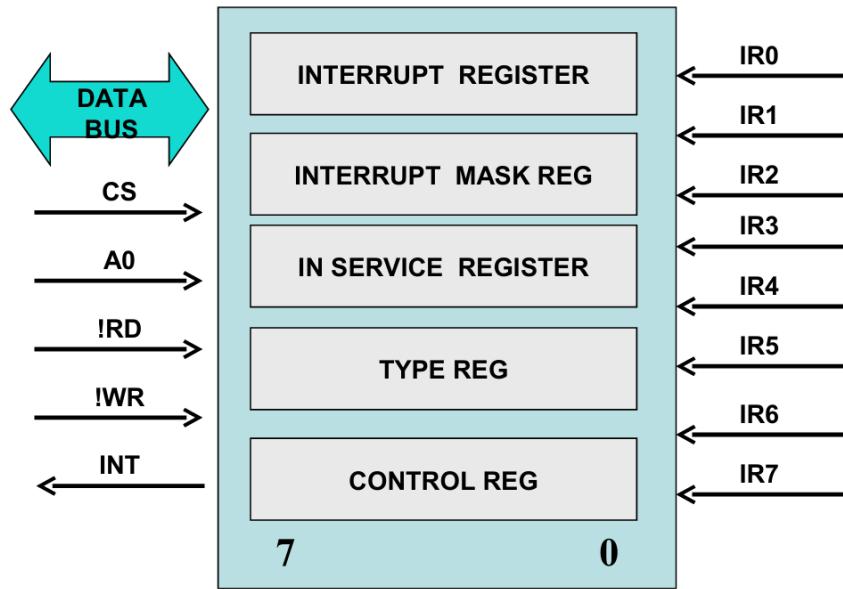


Figura 2.9: PIC semplificato

Vediamo nel dettaglio il funzionamento di questi registri:

- **TYPE REGISTER:** contiene informazioni sui vettori di interruzione associati al PIC. I vettori si ricavano a partire da un *vector number* di base sommando uno spiazzamento su 3 bit, ovvero i 3 bit meno significativi del registro. Lo spiazzamento viene settato automaticamente in base alla linea interrompente. Praticamente al programmatore è permesso di inserire il numero di base, al quale viene automaticamente sommato uno spiazzamento in base alle linee interrompenti. L'accesso a TR viene fatto selezionando in scrittura l'indirizzo dispari subito dopo un *RESET*;
- **INTERRUPT MASK REGISTER:** memorizza le linee di interruzione che devono essere mascherate. Per mascherare una linea basta inserire un 1 nel bit corrispondente. IMR è accessibile sia in lettura che in scrittura ad indirizzo dispari, ma in scrittura il dispositivo non deve essere nello stato *reset*, ovvero deve essere già stato scritto almeno TR;
- **CONTROL REGISTER:** registro da 8 bit accessibile sempre in scrittura ad indirizzo pari:
  - bit 0,1,2: determinano il bit da cancellare in *ISR*.
  - bit 3: bit End of Interrupt, se posto a 1 fa cancellare il bit puntato dai bit 0-2.
  - bit 4: bit Automatic End of Interrupt, se posto a 1 fa cancellare automaticamente il bit in ISR dopo la trasmissione dell'interruzione;
  - bit 5: *non utilizzato*;
  - bit 6: bit Register Interrupt Selector, se il bit 7 è alto seleziona l'accesso a ISR in lettura, se il bit 7 è basso seleziona l'accesso a IRR in lettura;
  - bit 7: permette se posto a 1 la lettura dei registri ISR e IRR (vedi punto precedente).
- **INTERRUPT REQUEST REGISTER:** memorizza le richieste di interruzione relative alle singole linee di interruzione. Questo registro è accessibile solo in lettura

ad indirizzo pari in accordo a quanto scritto nel registro CTRL. Quando la richiesta d'interruzione, arrivata sulla linea n, viene spedita al gestore delle interruzioni (PIC o CPU) collegato al PIC, il bit n-esimo di IRR è cancellato;

- **IN SERVICE REGISTER:** sono memorizzate le interruzioni trasmesse. Questo registro è accessibile solo in lettura all'indirizzo pari in accordo con quanto settato nel registro di controllo.

### 2.1.7 Estensione del modello IO generale

Un modello di architettura dotato solo di PIA (2.1.5) è limitato: può gestire solo caratteri, ha a disposizione solo 7 interruzioni e può generare attese infinite con il protocollo di handshaking. Per risolvere questi problemi, vengono introdotti nuovi elementi nell'architettura: **DMA** per gestire il trasferimento di messaggi invece di caratteri, **PIC** (2.1.6.2) per superare la limitazione sul numero di ISR indirizzabili e **TIMER** per gestire la temporizzazione e il risolvere il problema delle attese infinite (2.10).

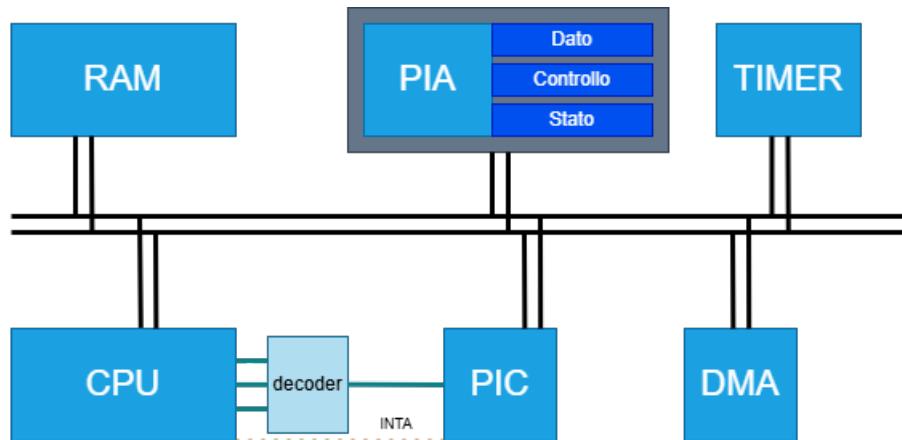


Figura 2.10: Modello IO esteso - schema logico

Il timer espone il modello di programmazione Registro di stato, Registro di valore e Registro di Modo. Nel registro di valore di solito è scritto un istante in cui il timer si "sveglia" e genera un'interruzione: infatti il timer possiede una linea con la quale può comunicare un'interruzione.

## 2.2 DMA (Direct Memory Access)

Il **DMA** è un dispositivo che permette di sollevare il processore dall'onere di trasferire i dati tra varie periferiche. Questo significa che i dispositivi possono comunicare direttamente con la memoria senza passare per la CPU. In particolare, il DMA permette di gestire il trasferimento dati tra:

- **Memoria ↔ Periferica;**
- **Periferica ↔ Memoria;**
- **Memoria ↔ Memoria.**

Il suo principio di funzionamento è semplice, ed è schematizzabile tramite tre registri principali:

- **Registro Indirizzo:** indica l'indirizzo da cui prelevare il dato;
- **Registro Conteggio:** indica il conteggio del numero di dati trasferiti e permette di capire quando interrompere il trasferimento;
- **Registro Identificativo:** tramite tale registro si identifica:
  - o il dispositivo da considerare per il trasferimento;
  - o l'area di memoria da considerare per il trasferimento.

La reale architettura del DMA è però più complessa. La maggior complessità dell'architettura proviene da varie problematiche che si possono riscontrare durante il funzionamento, come, ad esempio, l'accesso al BUS dati in maniera concorrente al processore. È pertanto necessario che il DMA non sia collegato al processore solo tramite il bus dati, ma anche tramite vari segnali di controllo, che permettono al processore e al DMA di potersi coordinare.

Il dispositivo di riferimento nelle esercitazioni del corso è l'**Intel 8237**, che per la sua architettura dispone di 4 canali per il collegamento con 4 dispositivi diversi. Nella versione simulata in ASIM, tale componente è composto di soli 2 canali.

Scendendo più nei dettagli, il dispositivo reale è in grado di sostenere 4 modalità di funzionamento differenti:

- **Single:** si trasferisce una *word* alla volta; dopo aver trasferito una word, il DMA restituisce il BUS al processore almeno per un ciclo;
- **Block:** si trasferisce un intero *blocco* non appena il DMA acquisisce il BUS. Alla fine del trasferimento viene inoltrata un'interruzione al processore che segnala la disponibilità del BUS;
- **On Demand:** simile alla modalità Block, con la differenza che il trasferimento può essere interrotto dal processore e poi ripreso, grazie ai registri contatore, dall'esatto punto in cui era stato interrotto;
- **Cascade:** modalità di funzionamento che permette di collegare più DMA in cascata per gestire più di 4 canali.

Il DMA acquisisce il controllo del bus attraverso il seguente processo (fare riferimento alla figura 2.12):

- IL DMA richiede il possesso del bus al processore tramite il segnale BR Bus Request;
- Il processore fornisce il consenso alla richiesta di utilizzo del bus tramite il segnale BG Bus Grant;
- Il dispositivo che ha richiesto il bus fornisce l'ack al processore mediante il segnale BGACK, e il processore viene scollegato elettricamente dal BUS indirizzi.

Il DMA si interfaccia con le periferiche mediante i segnali DREQ e DACK: DREQ è una richiesta da parte della periferica per il DMA di un trasferimento dati, mentre DACK è la comunicazione da parte del DMA che un dispositivo è stato selezionato per un trasferimento, ed è propagato una volta che il DAC ha acquisito l'accesso al bus indirizzi. Oltre al minor numero di canali, il componente simulato in ASIM non supporta tutte le modalità sopra citate: le modalità utilizzabili in ASIM sono infatti **Single** e **Block**. Guardando la figura 2.11, abbiamo il modello architettonale del componente realizzato in ASIM, in cui i registri posti sulla sinistra sono di comunicazione con il processore, mentre i segnali sulla destra servono per l'interfacciamento con le periferiche collegate ai canali. Per la comunicazione con il processore, i segnali rappresentati hanno il seguente significato:

- **D0-D7**: collegamento al BUS dati da e verso il componente;
- **CS**: segnale binario di selezione del dispositivo;
- **A0-A3**: attenzione! Non tutti i segnali  $A_i$ , ma solo i 4 meno significativi, vengono utilizzati per la selezione dello specifico registro interno;
- **IOR** e **IOW**: segnali di gestione della lettura e della scrittura sul componente e sulle periferiche;
- **MEMR** e **MEMW**: segnali di gestione della lettura e della scrittura sui dispositivi di memoria;
- **CLK** e **Reset**: classici segnali di clock (tempificazione) e reset dei registri del dispositivo;
- **HRQ**: segnale di richiesta del controllo del sistema BUS, solitamente collegato all'ingresso HOLD della CPU;
- **HLDA**: segnale proveniente dalla CPU che segnala l'acquisizione del BUS da parte del processore;
- **EOP**: linea di interruzione *bidirezionale* che si collega al processore per segnalare il completamento del trasferimento.

Dati i due differenti canali, si avranno due periferiche collegate allo stesso dispositivo DMA. La comunicazione può essere effettuata da una sola periferica per volta. Tale decisione è presa secondo un ordine di priorità, per cui il dispositivo collegato ai terminali 0 ha priorità maggiore rispetto a quello collegato ai terminali 1. I segnali che gestiscono le periferiche sono:

- **DREQ0** e **DREQ1**: segnali usati dalle periferiche per richiedere l'accesso al BUS tramite cicli DMA;
- **DACK0** e **DACK1**: segnali con cui il DMA comunica alla periferica la disponibilità a soddisfare la richiesta.

Consideriamo il canale 0 del DMA, ed analizziamo il modello di programmazione. Il registro CADDR0 (Current Adress) contiene l'indirizzo della locazione di memoria interessata dal trasferimento, ed è accessibile in lettura e scrittura all'indirizzo relativo \$0. Il registro

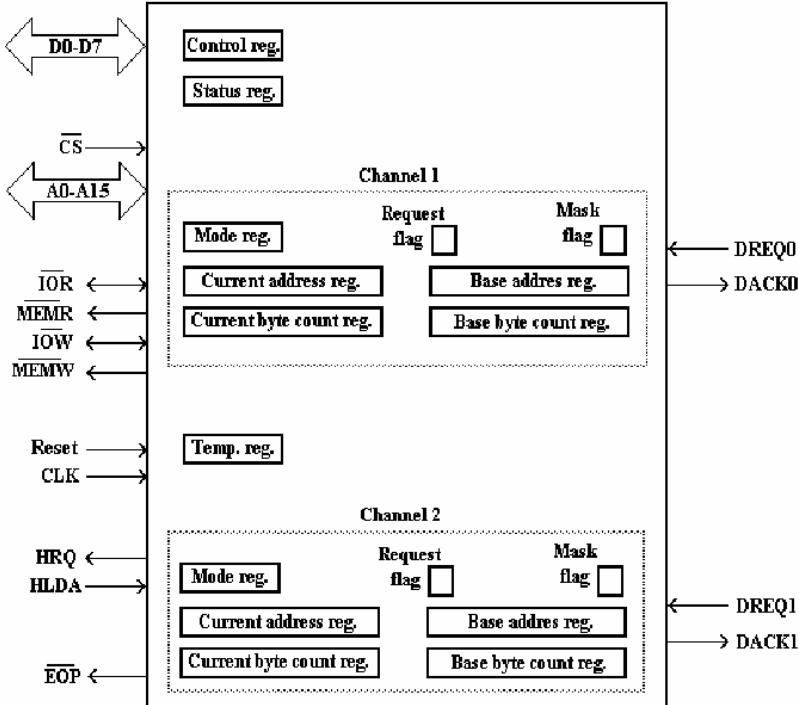


Figura 2.11: Modello di programmazione DMA a 2 canali

BADDR0 (Base Address) contiene l'indirizzo iniziale di CADDR0, ed è accessibile in scrittura automaticamente insieme a CADDR0. Il registro CCOUNT0 (Current Count) contiene il numero attuale di byte da trasferire, ed è accessibile in lettura e scrittura all'indirizzo relativo \$1. Il registro BCOUNT0 (Base Count) contiene il numero di byte iniziale da trasferire, ed è accessibile in scrittura automaticamente insieme a CCOUNT0. Il registro MODE0 contiene informazioni sul modo di funzionamento del canale 0, ed è accessibile in scrittura all'indirizzo relativo \$B. Il Control Register è unico, e i suoi 8 bit sono suddivisi nei 4 meno significativi che indicano lo stato del componente, mentre i 4 più significativi sono bit di controllo. È accessibile sia in scrittura che in lettura all'indirizzo relativo \$8. All'indirizzo relativo \$9 è accessibile in sola scrittura il flag RF (REQUEST FLAG), che è il flag dove segnalare una richiesta da software al DMA per un trasferimento, ed è del tutto analogo di una richiesta da periferica tramite segnali appositi; la selezione del canale avviene sul bit meno significativo del dato scritto nel registro (\*\*\*\*\*0 per il canale 0, \*\*\*\*\*1 per il canale 1) mentre il valore che si vuole assegnare al flag deve essere scritto sul quarto bit del dato (%00001000 → \$08 → RF alto sul canale 0). MF Mask Flag, accessibile in sola scrittura all'indirizzo relativo \$A, serve a mascherare con il valore alto le richieste dei rispettivi canali.

### 2.2.1 Utilizzo effettivo in ASIM

Per capire come utilizzare il DMA, occorre prima stabilire cosa il DMA dovrà fare e in che modo lavorerà. Per definire queste cose si usano i registri di controllo (unico per tutti i canali) e i registri di Modo (uno per ogni canale). Quando si dichiara il componente nel file di configurazione, si definiscono due indirizzi (address 1 e address 2) che individuano sedici locazioni accessibili dal processore come registri di memoria (*memory mapped*). Il

decodificatore collegato sul pin  $\overline{CS}$  attiva il pin quando rileva sul bus indirizzi un valore compreso nell'insieme [address1, address2].

All'interno del file di configurazione vi sono anche le altre voci che identificano come il componente si collega ai vari BUS e agli altri oggetti del sistema. Una buona rappresentazione logica del sistema DMA si trova nella figura 2.12.

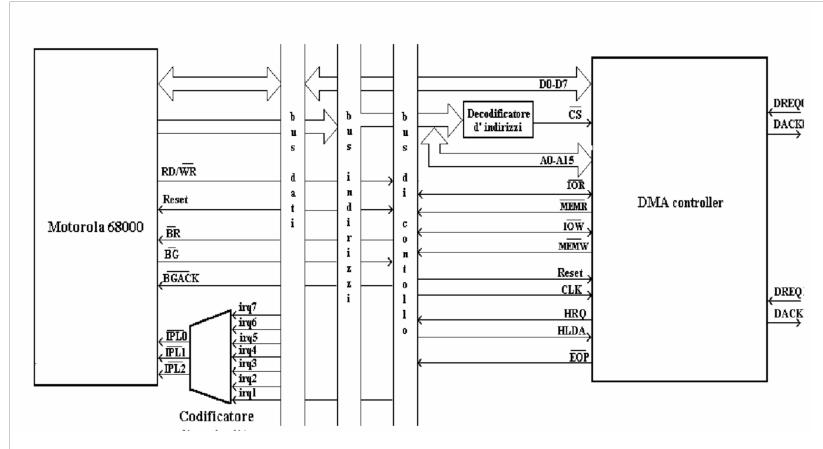


Figura 2.12: Montaggio del componente DMA rispetto al sistema generale

Una volta compresa l'architettura del sistema, possiamo analizzare il significato dei bit dei vari registri interni.

**Registro Mode** Il registro Mode (uno per canale) è descritto nella Tabella 2.5.

Bit	Significato
0	Instrada il dato su un canale: 0 per il canale 0, 1 per il canale 1
1-2	Non utilizzati
3	Direzione di trasferimento: 0 = da memoria a interfaccia, 1 = da interfaccia a memoria
4	Abilita l' <b>Autoinizializzazione</b> : se impostato a 1, alla fine del conteggio resetta i registri con i valori di base
5	Incremento di CADDR (0) o decremento (1)
6	Non utilizzato
7	Modalità di trasferimento: 0 = Single, 1 = Block

Tabella 2.5: Significato dei bit del registro MODE

**Registro di Controllo** Il registro di controllo del DMA controller è illustrato nella Tabella 2.6.

Bit	Significato
0	Termina conteggio per il canale 0
1	Termina conteggio per il canale 1
2	È stata inoltrata una richiesta al canale 0
3	È stata inoltrata una richiesta al canale 1
4	Non utilizzato
5	Abilita il trasferimento da memoria a memoria
6	Impone che in un trasferimento memoria-memoria l'indirizzo sorgente rimanga costante
7	Abilita il DMA controller

Tabella 2.6: Significato dei bit del registro CTRL

## 2.2.2 Implementazione in Motorola 68k

Le implementazioni in ASIM con Motorola 68k sfruttano varie architetture definite nei file di configurazione. Una volta definita l'architettura, si stabilisce come gestire i registri e le periferiche (come il DMA).

### 2.2.2.1 Caso Memoria-Memoria

Nel caso memoria-memoria, il DMA richiede una particolare definizione. Si devono definire:

- l'area dati (origine e destinazione);
- gli intervalli per l'accesso ai registri interni del DMA.

```

1           ORG      $9500
2 origine      DC.B     0,1,2,3,4,5,6,7,8,9 * msg
3 destinazione DS.B     34
4
5 * Definizione di indirizzo e offset per accesso ai registri DMA
6 dma          EQU      $2010
7 caddr0       EQU      0
8 caddr1       EQU      2
9 ccount1      EQU      3
10 cntrl        EQU      8
11 mode         EQU      11
12 reset        EQU      13
13 clearmf     EQU      14
14 writeamf    EQU      15
15 nbytes       EQU      34

```

Una volta definita la parte "statica", si passa alla parte operativa:

```

1   ORG      $8200
2
3   MOVE.W    #dma,A0           * Carico l'indirizzo del DMA
4
5   * Carico l'indirizzo di origine dei dati nel primo registro
     indirizzo

```

```

6      MOVE.W      #origine,caddr0(A0)
7
8 * Carico l'indirizzo di destinazione nel secondo registro
9   indirizzo
10    MOVE.W      #destinazione,caddr1(A0)
11
12 * Numero di caratteri da trasferire
13   MOVE.B      #nbyte,ccount1(A0)
14
15 * Configuro il primo canale: Block, incremento,
16   autoinizializzazione
17   MOVE.B      #$90,mode(A0)
18
19 * Configuro il secondo canale in modo analogo
20   MOVE.B      #$91,mode(A0)
21
22 * Set del control register
23   MOVE.B      #$A0,cntrl(A0)
24 LOOP  JMP      LOOP

```

Alla fine del trasferimento, occorre gestire l'interruzione scatenata. L'ISR ha come scopo il reset del DMA:

```

1      ORG      $8700
2
3 int7   MOVE.L  A0,-(A7)          * Salva il contesto
4       MOVE.W  #dma,A0
5       MOVE.B  #0,reset(A0)        * Resetta il DMA
6       MOVE.L  (A7)+,A0           * Ripristina il registro
7       RTE

```

### 2.2.2.2 Caso Dispositivo → Memoria

La configurazione che studiamo in questo paragrafo rappresenta due sistemi in comunicazione, uno in trasmissione e uno in ricezione tramite due PIA. La PIA in trasmissione manda continuamente caratteri alla PIA in ricezione, il DMA collegato al sistema in ricezione sposta i caratteri ricevuti dalla PIA verso la Memoria, senza passare per il processore. Presentiamo innanzitutto il sistema in trasmissione che denominiamo S1, che effettua il trasferimento con un semplice ciclo.

```

1 ***AREA DATI***
2      ORG      $8000
3 MSG    DC.B    1,2,3,4,5,6
4 DIM    DC.B    6
5
6 ***MAIN***
7      ORG      $8200
8 PIADB EQU     $2006
9 PIACB EQU     $2007
10

```

```

11
12 MAIN      JSR      DVOUT
13          MOVEA.L #PIACB,A1
14          MOVEA.L #PIADB,A2
15          MOVEA.L #MSG,A0
16          MOVE.B  DIM,D0
17          CLR     D1
18          MOVE    D0,D2
19 INVIO    MOVE.B  (AO)+,D1
20          MOVE.B  D1,(A2)
21          ADDQ   #-1,D2
22          BNE    INVIO
23
24 LOOP     JMP     LOOP
25
26 DVOUT    MOVE.B  #0,PIACB
27          MOVE.B  #$FF,PIADB
28          MOVE.B  #%"00100100
29          RTS

```

Osserviamo che in questo caso non effettuiamo l'handshaking, e non effettuiamo la lettura fittizia per assicurarci che CRB7 sia basso, semplicemente scriviamo sul PRB. Questo è dovuto al fatto che il DMA non possiede segnali per parlare direttamente con la PIA, quindi *forziamo* una scrittura senza handshaking. Procediamo con la presentazione del driver di un sistema che riceve un messaggio di 30 caratteri su una PIA e lo copia in memoria direttamente tramite un DMA. Assumiamo che la PIA sia collegata al canale 0 del DMA e che la linea di interruzione del DMA sia collegata sulla linea 7 del processore (Autovettore 31).

```

1 ***AREA DATI ***
2 PIADA   EQU      $2004
3 PIACA   EQU      $2005
4
5 dma     EQU      $2010
6 caddr0  EQU      0
7 ccount0 EQU      1
8 cntrl   EQU      8
9 request EQU      9
10 mode0  EQU      11
11 nbyte   EQU      30
12
13
14          ORG      $8000
15 msg     DS.B     nbyte
16
17 ***AREA CODICE ***
18          ORG      $8200
19 MAIN    JSR      DVAIN
20          MOVE.W  SR,D0
21          ANDI.W  #$D8FF,D0 *stato utente, int abilitate
22          MOVE.W  D0,SR
23

```

```

24      MOVE.W  dma,A1
25      MOVE.B   #nbyte,ccount0(A1)
26      MOVE.W   msg,caddr0(A1)
27      MOVE     #$08,mode0(A1)
28      MOVE     #$80,cntrl(A1)
29      MOVE     #$08,request(A1)
30
31 LOOP    JUMP    LOOP
32
33 DVAIN   MOVE     #0,PIACA
34      MOVE.W  #00,PIADA
35      MOVE     #$00100101,PIACA
36      RTS
37
38 **ISR FINE TRASFERIMENTO**
39      ORG     $8700
40      NOP
41      RTE

```

### 2.2.2.3 Caso Memoria → Dispositivo

In questo caso, un dispositivo deve trasmettere un messaggio da un'area di memoria verso una PIA usando il DMA. La PIA deve essere programmata per ricevere dati, quindi il porto va configurato in INPUT. Nell'esempio presentato a lezione, non si affronta la questione del trasferimento da PIA del sistema S1 a PIA del sistema S2, quindi il programma non funziona ed è presentato al solo scopo didattico di illustrare il passaggio da Memoria e Interfaccia dispositivo tramite DMA.

```

1 ***AREA COSTANTI***
2 PIADA   EQU      $2004
3 PIACA   EQU      $2005
4
5 dma      EQU      $2010
6 caddr0  EQU      0
7 ccount0 EQU      1
8 control EQU      8
9 request EQU      9
10 mode0   EQU      11
11 nbyte   EQU      6
12
13 ***AREA DATI***
14      ORG     $8000
15 MSG     DC.B    1,2,3,4,5,6
16 DIM     DC.B    6
17
18 ***AREA CODICE***
19 MAIN    JSR      PIAINIT
20      MOVE.W  SR,DO
21      ANDI.W  #$D8FF,DO
22      MOVE.W  DO,SR
23

```

```
24      MOVE.W #dma,A1
25      MOVE.W #nbyte,ccount0(A1)
26      MOVE.W #msg,caddr0(A1)
27      MOVE    #$10000000,mode0(A1)
28      MOVE    #$08,request(A1)
29
30 LOOP    JMP     LOOP
31
32 PIAINIT MOVE    #0,PIACA
33      MOVE    #$00,PIADA
34      MOVE    #%"00100100,PIACA
35      RTS
```

## 2.3 USART

L'USART (Universal Synchronous-Asynchronous Receiver-Transmitter) è un interfaccia di comunicazione seriale. In generale i sistemi di comunicazione seriale utilizzano solo un filo per comunicare ( $\text{Tx} \rightarrow \text{Rx}$ ), che permette il collegamento tra il trasmittente ed il ricevente; a questo singolo collegamento, solitamente, si aggiungono altri fili, che aggiungono: controlli di flusso, segnali di tempificazione ecc. Non si scende troppo nei dettagli di tali architetture, poiché poi si vanno a specializzare nei vari dispositivi che vengono prodotti, un esempio sarà l'Intel 8251.

### 2.3.1 Comunicazioni Sincrona ed Asincrona

Come detto nell'introduzione, l'USART (a differenza del suo predecessore, l'UART), prevede sia una modalità di funzionamento **sincrona** che **asincrona**. Tali modalità sono profondamente diverse, sia per quanto riguarda l'architettura sia per quanto riguarda il loro modo di comunicazione dei dati.

#### 2.3.1.1 Comunicazione Sincrona

Nel caso della comunicazione sincrona, i due dispositivi (per una comunicazione unidirezionale) condividono 2 collegamenti:

- $\text{Tx} \rightarrow \text{Rx}$
- $\text{clk\_t} \rightarrow \text{clk\_r}$

Questo perchè una comunicazione sincrona sfrutta un segnale di tempificazione comune. La velocità di trasmissione dei dati è quindi dettata dalla frequenza del clock comune. A differenza della comunicazione **asincrona**, la comunicazione sincrona è molto più veloce e robusta (data la sincronizzazione in hardware tramite il clock condiviso), ma è molto difficile da implementare a livello hardware, poichè bisogna sincronizzare in maniera molto precisa dati e trasmissione di essi.

#### 2.3.1.2 Comunicazione Asincrona

Nel caso della comunicazione di tipo Asincrona, i due dispositivi richiedono un singolo collegamento (sempre ragionando sul singolo canale trasmissivo  $\text{Tx} \rightarrow \text{Rx}$ ). L'invio dei dati è gestito da due bit di start e stop che vanno a scandire, precisamente, l'inizio dell'invio del messaggio e la fine del messaggio. La tipologia di comunicazione, rispetto alla **sincrona** risulta meno efficiente rispetto al trasporto di informazioni, anche se conserva il vantaggio per il rispetto dell'asincronicità intrinseca dei dati, e quindi ottimizza meglio i tempi di invio dei dati (non deve aspettare un fronte del clock condiviso come nel caso della comunicazione sincrona). Nel caso **asincrono**, negli anni 60, è stato definito uno standard di comunicazione, ovvero l'**RS-232**, che definisce in che modo le varie interfacce possano comunicare tra di loro: tale standard predispone varie altre tipologie di collegamenti che sono utilizzate per il controllo di flusso dei vari dati. Tale standard fu creato per permettere la comunicazione tra un dispositivo DCE (Data Communication Equipment) ed un dispositivo DTE (Data Transmission Equipment), tali dispositivi potevano essere di vario tipo, per esempio originariamente erano un calcolatore ed il suo modem. Precisamente, i pin che sono presentati ed utilizzati nello standard sono:

- **RD**: Pin di ricezione di dati seriali;
- **TD**: Pin di trasmissione di dati seriali;
- **DCD**: Data Carrier Detect, è un pin che viene utilizzato per controllare il corretto collegamento tra i due dispositivi (controllo del funzionamento corretto del dispositivo e delle portanti utilizzate);
- **GND**: Ground, è un pin che collega la massa dei due dispositivi;
- **DTR**: Data Terminal Ready, indica il segnale di uscita per poter iniziare o terminare una fase di handshacking (ACK o SYN);
- **DSR**: Data Set Ready, indica il segnale di ingresso che viene inviato dal ricevente (o per indicare un ACK o un SYN);
- **RTS**: Request to send, segnale di uscita per il controllo del flusso;
- **CTS**: Clear To Send, segnale di ingresso per il controllo di flusso;
- **RI**: Ring Indicator, Indica che il trasmittente sta "chiamando", e può scatenare una interruzione per poter gestire la comunicazione.

I segnali **DTR** e **DSR**, sono pin che vanno ad implementare l'handshacking ad un livello molto alto. Mentre **RTS** e **CTS**, vengono utilizzati per comprendere quando il trasmittente può effettivamente trasmettere. Per quando sia un pò confusionario il loro funzionamento, immaginando una comunicazione dove dev'essere effettuato l'invio di messaggi, il primo handshacking serve ad identificare il messaggio, mentre il secondo per trasmettere i vari byte e quindi richiedere al ricevitore se è pronto a ricevere o sta ancora processando il dato precedente.

### 2.3.1.3 Errori principali presenti nella comiunicazione seriale

Quando si fanno interagire due sistemi mediante una comunicazione di tipo seriale asincrona (UART), in fase di ricezione, si può andare incontro ai seguenti tipi di errori:

- **Errore di parità**: Il valore del bit di parità inviato non corrisponde con quello calcolato al ricevitore;
- **Errore di framing**: Non è previsto il bit di stop;
- **Errore di Overrunning**: Errore dovuto alla differenza dei clock di invio. Se il sistema ricevente è più lento del sistema trasmittente, allora si può avere perdita di informazioni (non si coglie bene il bit di start);

Tra questi errori l'unico su cui si può attuare un miglioramento è quello di overrun, questo perchè si può prevedere che il dato venga trasmesso secondo uno specifico protocollo di handshacking. Difatti nelle architetture sono presenti appositi collegamenti per l'implementazione di tale soluzione, anche se quest'ultima rallenta un po la ricezione rendendola meno efficiente.

### 2.3.2 Intel 8251A

L'Intel 8251A è un dispositivo programmabile che permette la comunicazione tra la CPU e una qualunque periferica seriale che utilizza lo standard RS-232. Tale dispositivo riceve i dati da trasmettere in maniera parallela dalla CPU, per poi inviarli in maniera seriale attraverso l'utilizzo del protocollo selezionato in una prima fase di configurazione (per questo programmabile). La sua struttura principale è quella visualizzabile nella figura [2.13].

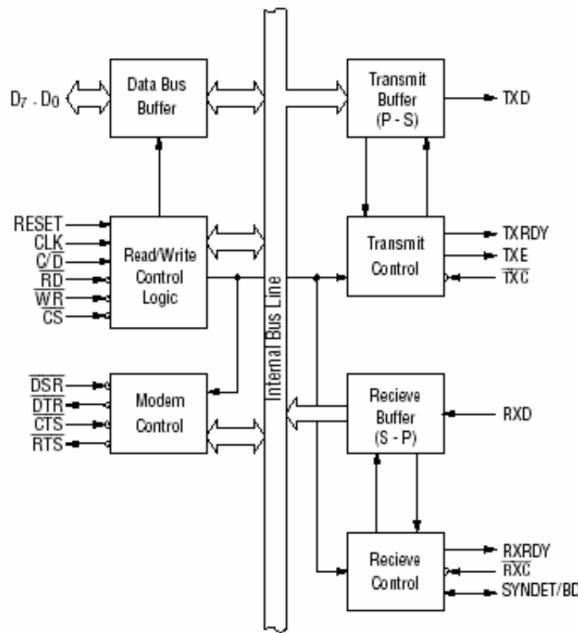


Figura 2.13: Struttura dell'Intel 8251A

In tale struttura logica notiamo vari blocchi differenti, ovvero:

- **Data Bus Buffer:** Buffer in cui vengono conservati i dati (o in ricezione o in trasmissione), in generale la comunicazione con tale blocco avviene in maniera parallela;
- **Read/Write Control Logic:** Blocco di controllo dell'intero sistema, è un' interfaccia di utilizzo degli elementi interni del dispositivo per la CPU. Approfondiamo il significato dei vari pin:
  - **Reset:** Pin che permette alla CPU di resettare tutti i registri interni al dispositivo;
  - **Clk:** Segnale di temporizzazione per il dispositivo;
  - **C/D:** Selezione per il registro di controllo o di dato;
  - **RD e WR:** Segnali che identificano se si vuole effettuare una lettura o una scrittura;
  - **CS:** Chip select, utilizzato per abilitare il dispositivo;
- **Modem Control:** Blocco di collegamento con l'altro dispositivo esterno, contiene i segnali che vengono utilizzati per lo standard RS232;

- **DSR (Data Set Ready)**
  - **DTR (Data Terminal Ready)**
  - **CTS (Clear To Send)**
  - **RTS (Ready To Send)**
- **Transmit Buffer:** Contiene lo shift register impiegato nella fase di trasmissione. Gli vengono passati i dati da trasmettere in parallelo dal BUS dati e poi li invia sotto forma seriale tramite il segnale **TXD**. Il controllo sulla trasmissione viene effettuato dall'apposito blocco con cui si condividono sia un segnale di ingresso che di uscita
  - **Transmit Control:** Il blocco di transmit control viene gestito da vari segnali processati, in base alle richieste della CPU. I suoi segnali di interazione con l'esterno, sono:
    - **TXRDY (Transmitter ReaDY):** Segnale di utilizzato per segnalare alla CPU che il trasmettitore è pronto a ricevere dalla CPU un nuovo dato da trasmettere;
    - **TXE (Transmitter Empty):** Segnale di uscita per identificare che il buffer di trasmissione può essere sovrascritto (Attesa del nuovo dato da trasmettere);
    - **TXC (Transmitter Clock):** Utilizzato nel caso sincrono per gestire il segnale di clock condiviso;
  - **Receive Buffer:** Il blocco di receive buffer, svolge il compito contrario del transmit buffer, ovvero, converte i dati seriali in registro parallelo e li invia sul bus dati. Questo riceve i dati tramite il segnale esterno RXD. Oltre a tale segnale la ricezione è controllata mediante il blocco Receive Control:
  - **Receive Control:** Il blocco di receive control, come il blocco di transmit control, gestisce il sistema di ricezione con l'esterno. Tale gestione viene effettuata attraverso i seguenti segnali:
    - **RXRDY (Receive Ready):** Specifica alla CPU che il sistema ricevente ha ricevuto un dato che è pronto per essere letto;
    - **RXC (Receive Clock):** Segnale utilizzato in ingresso per il clock delle comunicazioni sincrone;
    - **SYNDER/BD (Synchronous Detect/Break Detect):** Segnale utilizzato per ricevere i caratteri di sincronismo, nel caso di comunicazione asincrona, mentre nella modalità sincrona segnala la rilevazione di una condizione di break.

### 2.3.2.1 Intel-8251A in Asim

Come detto in precedenza, l'Intel 8251A è un dispositivo programmabile, quindi richiede una prima fase di configurazione (tramite il suo registro di modo) e poi il suo utilizzo tramite gli altri registri dell'interfaccia. Nel caso di ASIM il modello di programmazione utilizzato è quello visualizzabile alla figura [2.14]

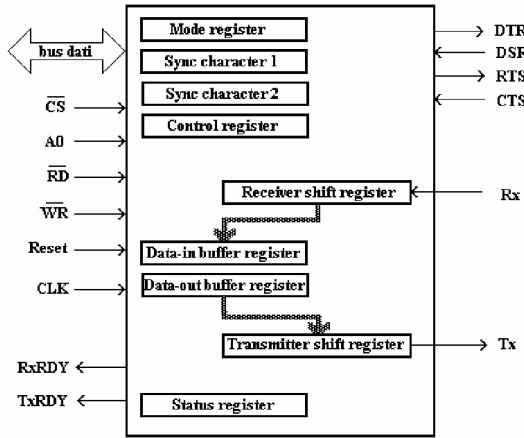


Figura 2.14: Modello di programmazione dell'8251A simulato

Da tale figura dobbiamo discriminare due lati, quello sinistro, che è quello che viene connesso direttamente ai bus e quindi al processore, mentre quello a destra è di interfacciamento verso l'altro dispositivo USART. Notiamo che i pin sono identici a quelli spiegati nell'architettura generale dell'8251 [2.3.2]. Una volta visto il modello di programmazione inserisco l'oggetto all'interno del file di configurazione (.cfg). In generale, come anche per prove precedenti, questi file di configurazione vengono forniti, ma nel caso si volessero creare *from scratch* si rimanda all'appendice A.1.4. I registri con i quali l'USART simulata si interfaccia con la CPU sono:

- **Mode**: Registro di modo con cui posso configurare il funzionamento della USART, quindi selezionare se la comunicazione sia sincrona o asincrona ed il formato dell'informazione da trasmettere (eventuali bit di parità ecc.);
- **CTRL**: Registro di controllo, che permette di poter pilotare i segnali di utilizzo della USART, utile per quando si devono implementare sistemi che utilizzano handshacking ecc.;
- **SYNCH-1**: Registro di Sincronizzazione, utilizzato nella comunicazione sincrona per prevedere l'arrivo del primo simbolo di sincronizzazione;
- **SYNCH-2**: Registro di sincronizzazione, utilizzato nella comunicazione sincrona per prevedere l'arrivo dell'eventuale secondo simbolo di informazione (dipende dalla modalità che si è settata).

Per accedere a tali registri di gestione del dispositivo si fa riferimento all'indirizzo dispari messo a disposizione all'interno del file di configurazione. In generale, tali registri, non hanno ognuno il suo puntatore, ma condividono lo stesso indirizzo, che, in base alla tipologia di funzionamento e di impostazione ne permette l'accesso. Precisamente, ogni volta che si fa un accesso in scrittura all'indirizzo dispari si segue l'ordine che è mostrato all'interno dell'immagine [2.15].

Tali però non sono gli unici registri presenti, poiché sono presenti anche i registri di dato (uno per l'ingresso ed uno per l'uscita) ed il registro di stato, per accedere a tali registri si accede, per il registro di stato, in lettura all'indirizzo dispari, mentre per gli indirizzi di

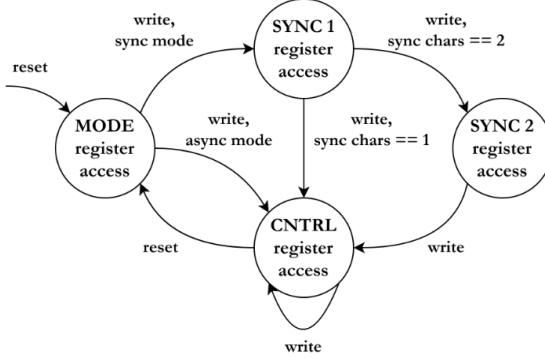


Figura 2.15: Ordine di accesso all'indirizzo dispari

dato si accede, in scrittura all'indirizzo pari per il registro d'uscita (DATAOUT), mentre si accede in lettura al registro pari per l'accesso per il registro di ingresso (DATAIN). Di seguito vi sono le tabelle che presentano la struttura dei registri di gestione (Modo, controllo e stato) [2.8,2.9,2.10] e le modalità di accesso ai differenti registri [2.7]

Indirizzo	Tipo di accesso	Registro selezionato
Pari	Lettura (R)	DATIN
Pari	Scrittura (W)	DATOUT
Dispari	Lettura (R)	STATUS
Dispari	Scrittura (W)	MODE, CNTRL, SYNC1, SYNC2

Tabella 2.7: Accessi ai registri in base all'indirizzo e tipo

Bit	Significato
0	Determina il tipo di trasmissione: 0 per sincrona, 1 per asincrona.
1	Non utilizzato.
2-3	Numero di bit d'informazione per carattere: 00 = 5 bit, 01 = 6 bit, 10 = 7 bit, 11 = 8 bit.
4	Abilita la presenza del bit di parità (1 = presente).
5	Tipo di parità: pari se 1, dispari se 0.
6	In modalità asincrona: 0 = 1 bit di stop, 1 = 2 bit di stop.
7	In modalità sincrona: 0 = 1 carattere di sincronismo, 1 = 2 caratteri.

Tabella 2.8: Significato dei bit del registro MODE

### 2.3.3 USART in ASIM

In questo paragrafo esporremo e commenteremo alcuni esempi di utilizzo del componente ASIM che simula una USART.

Bit	Significato
0	Abilita il trasmettitore.
1	Attiva il segnale di handshaking DTR.
2	Abilita il ricevitore.
3	Non utilizzato.
4	Cancella i 3 bit d'errore nel registro STATUS.
5	Attiva il segnale di handshaking RTS.
6	Resetta l'interfaccia seriale.
7	Pone il ricevitore nello stato “hunt” per cercare i caratteri di sincronismo.

Tabella 2.9: Significato dei bit del registro CNTRL

Bit	Informazione indicata se posto ad 1
0	È stato copiato il carattere da DATOUT in TSHIFT (inizio trasmissione). Si azzera alla scrittura successiva.
1	Un carattere è stato ricevuto in RSHIFT e copiato in DATIN. Si azzera alla lettura.
2	In trasmissione sincrona, il trasmettitore è privo di dati.
3	È stato rilevato un errore di parità.
4	È stato rilevato un errore di overrun.
5	È stato rilevato un errore di framing.
6	È stato rilevato il/i caratteri di sincronismo previsti.
7	È stato attivato il segnale di handshaking DSR.

Tabella 2.10: Significato dei bit del registro STATUS

### 2.3.3.1 USART trasmissione ↔ ricezione asincrona

Iniziamo vedendo un esempio di programma che serve a provare una semplice configurazione costituita da due sistemi S1 e S2 dotati di processore MC68K e in comunicazione tra loro mediante USART mappata a \$2004. I due USART sono interconnessi tra loro e consentono ai due sistemi di scambiare un messaggio. La configurazione che prendiamo in considerazione durante gli esercizi è di tipo *NULL MODEM* (figura 2.16), e prevede che i segnali di handshaking vengano direttamente scambiati tra i terminali senza la presenza di un modem. In questa configurazione RTS serve a comunicare la capacità di ricevere messaggi più che una richiesta di invio. I sistemi verranno inizializzati ponendo DTR,RTS = 1,1 in modo da stabilire l'handshacking in partenza.

Il sistema S1 trasferisce un vettore di N caratteri verso il sistema S2 sul canale seriale utilizzando un protocollo asincrono. Il seguente codice è un driver per la programmazione del sistema S1 che effettua il trasferimento con un semplice ciclo, quindi **senza interruzioni nel sistema di trasmissione**:

```

1 ***AREA DATI ***
2      ORG      $8000
3 MSG      DC.B    170,204,153,129,195,6
4 DIM      DC.B    6
5 COUNT   DC.B    0          * contatore caratteri inviati
6

```

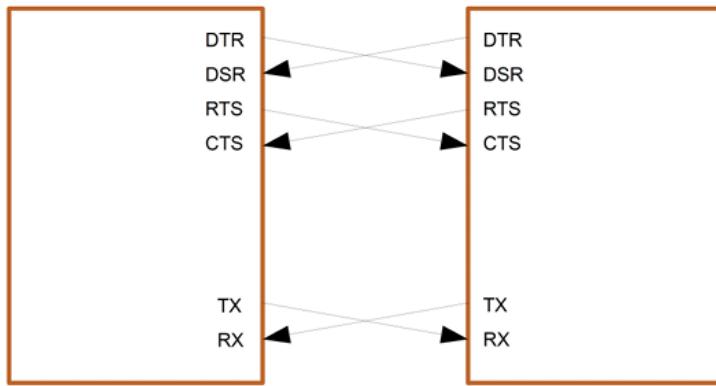


Figura 2.16: Connessione USART in simulazione

```

7 ***AREA CODICE***
8      ORG      $8200
9 USARTD  EQU      $2004    * registro DATO
10 USARTC EQU      $2005    * registro CTRL
11
12 MAIN     JSR      INIT
13      MOVE.W   SR,DO
14      ANDI.W   #$D8FF,DO
15      MOVE.W   DO,SR
16
17      MOVEA.L  #USARTD,A1
18      MOVEA.L  #USARTC,A2
19      MOVEA.L  #MSG,A0
20      MOVE.B   DIM,D3
21      CLR      D1
22      CLR      D2
23 DSR      MOVE      (A2),DO
24      ANDI      #$80,DO
25      BEQ      DSR
26 TxRDY    MOVE      (A2),DO
27      ANDI      #$01,DO
28      BEQ      TxRDY
29 INVIO    MOVE      (A0)+,D1
30      MOVE      D1,(A1)
31      ADD      #1,D2
32      CMP      D2,D3
33      BNE      TxRDY
34 LOOP     JMP      LOOP
35
36 INIT     MOVE.B   $%01011101,USARTC *MODE
37      MOVE.BD  $%00100011,USARTC *CTRL
38
39      ORG      $8800
40      RTE      * interruzione

```

Nella porzione di codice etichettata DSR, il programma effettua un test sul registro di stato della USART, e controlla che il bit relativo a Data Set Ready (ovvero altro sistema

pronto a ricevere dati); questo sarà sempre alto, anche nel prossimo codice, perchè DTR è inizializzato alto nella procedura INIT in entrambi i sistemi, quindi l'handshaking andrà *artificialmente* sempre a buon fine. Il seguente codice invece serve a programmare il sistema S2, che riceve il messaggio sulla seriale utilizzando le interruzioni. Il segnale di interruzione, associato al segnale RxRDY della seriale, è collegato alla linea a priorità 3 del processore come impostato nel file di configurazione. La routine è dunque mappata sull'autovettore 27.

```

1 ***AREA DATI***
2      ORG      $8200
3 MSG      DS .B    6
4 DIM      DC .B    6
5 COUNT   DC .B    0
6
7 ***AREA CODICE***
8 USARTD  EQU      $2004
9 USARTC  EQU      $2005
10
11 MAIN    JSR      INIT
12      MOVE .W   SR ,DO
13      ANDI .W   #$8dff ,DO
14      MOVE .W   DO ,SR
15      MOVEA .L  #USARTC ,A2
16 DSR     MOVE     (A2) ,DO
17      ANDI     #$80 ,DO
18      BEQ      DSR
19 LOOP    JMP      LOOP
20
21
22 INIT    MOVE     #$5d ,USARTC * MODE
23      MOVE     #$36 ,USARTC * CTRL
24      RTS
25
26      ORG      $8700
27 INT3    MOVEA .L A1 ,-(A7)
28      MOVEA .L A0 ,-(A7)
29      MOVE .L   DO ,-(A7)
30
31      MOVEA .L #USARTD ,A1
32      MOVEA .L #MSG ,AO
33      MOVE     COUNT ,DO
34
35      MOVE     (A1),(AO ,DO)
36      ADD      #1 ,DO
37      MOVE     DO ,COUNT
38
39      MOVE .L   (A7)+ ,DO
40      MOVEA .L (A7)+ ,AO
41      MOVEA .L (A7)+ ,A1
42      RTE

```

Come si evince dal codice, il sistema S2 al fine di ricevere il dato viene interrotto da un'interruzione scatenata dal segnale RxRDY, che si alza ogni qualvolta viene ricevuto un carattere. Analogamente, è possibile introdurre il meccanismo delle interruzioni anche per quanto riguarda il sistema S1, e far leva sul corrispettivo segnale TxRDY, che si alza ogni volta che un carattere viene inviato correttamente dalla seriale. La routine di servizio è mappata sull' autovettore 28, indirizzo \$70 in ROM. L'indirizzo posto in tale locazione, che costituisce l'entry point della ISR, è \$8800. La differenza è che nella programmazione del sistema S1 deve essere effettuato un *primo invio*:

```

1 PRIMO    MOVE      (A0) ,D1
2          MOVE      D1 ,A1
3          ADD       #1 ,D2
4          MOVE      D2 ,COUNT
5 LOOP     JMP       LOOP

```

Inoltre, deve essere gestita l'interruzione associata a TxRDY:

```

1          ORG      $8800
2 INT4     MOVEA .L A1 ,-(A7)
3          MOVEA .L A0 ,-(A7)
4          MOVE .L  D0 ,-(A7)
5          MOVE .L  D1 ,-(A7)

6
7          MOVEA .L #USARTD ,A1
8          MOVEA .L #MSG ,AO
9          MOVE      COUNT ,DO
10         MOVE     DIM ,D1

11
12         CMP      DO ,D1
13         BEQ      FINE
14 INVIO    ADDA .L  DO ,AO
15         MOVE .B  (AO) ,D1
16         MOVE .B  D1 ,(A1)

17
18         ADD      #1 ,DO
19         MOVE     DO ,COUNT
20 FINE     MOVE .L  (A7)+ ,D1
21         MOVE .L  (A7)+ ,DO
22         MOVEA .L (A7)+ ,AO
23         MOVEA .L (A7)+ ,A1
24         RTE

```

### 2.3.3.2 USART trasmissione ↔ ricezione sincrona

La comunicazione sincrona avviene tramite clock, che può essere trasmesso dal Master oppure una base dei tempi comune. In ricezione, è fondamentale che il ricevitore si aspetti di byte si sincronizzazione (SYNC1 ed eventualmente SYNC2). Quando il ricevitore legge questa sequenza, sa che la comunicazione è *agganciata*. come per gli altri casi, presentiamo il codice relativo al sistema S1 trasmettitore: La trasmissione avviene tramite un semplice ciclo.

```

1 ***AREA DATI ***

```

```

2      ORG      $8000
3 MSG      DC.B     170,204,153,129,195,6
4 DIM      DC.B     6
5 COUNT   DC.B     0
6
7 ***AREA CODICE***
8      ORG      $8200
9 USARTD  EQU      $2004
10 USARTC  EQU      $2005
11
12 MAIN    JSR      INIT
13      MOVE.W  SR,DO
14      ANDI.W  #$D8FF,DO
15      MOVE.W  DO,SR
16
17      MOVEA.L #USARTD,A1
18      MOVEA.L #USARTC,A2
19      MOVEA.L #MSG,A0
20      MOVE.B  DIM,D3
21      CLR     D1
22      CLR     D2
23 DSR     MOVE.B  (A2),DO
24      ANDI    #$80,DO
25      BEQ    DSR
26 TxRDY   MOVE.B  (A2),DO
27      ANDI    #$01,DO
28      BEQ    TxRDY
29 INVIO   MOVE    (A0)+,D1
30      MOVE    D1,A1
31      ADDQ    #1,D2
32      CMP     D2,D3
33      BNE    TxRDY
34
35 LOOP    JMP     LOOP
36
37 INIT    MOVE.B  $%10001100,USARTC *MODE
38      MOVE.B  #%FF,USARTC      *SYNC1
39      MOVE.B  #$00100011,USARTC *CTRL
40      RTS
41
42      ORG      $8800
43      RTE      * ISR...

```

In questo caso il byte di SYNC è 0xFF. Vediamo il codice per quanto riguarda la ricezione:

```

1 ***AREA DATI***
2      ORG      $8200
3 MSG      DS.B     6
4 DIM      DC.B     6
5 COUNT   DC.B     0
6
7 ***AREA CODICE***

```

```

8  USARTD  EQU      $2004
9  USARTC  EQU      $2005
10
11 MAIN     JSR      INIT
12         MOVE.W   SR,DO
13         ANDI.W   #$8dff,DO
14         MOVE.W   DO,SR
15         MOVEA.L  #USARTC,A2
16 DSR      MOVE     (A2),DO
17         ANDI    #$80,DO
18         BEQ     DSR
19 LOOP    JMP     LOOP
20
21
22 INIT    MOVE     #%"10001100,USARTC *MODE
23         MOVE     #%"ff,USARTC      *SYNC1
24         MOVE     #%"10100110,USARTC *CTRL
25         RTS
26
27         ORG     $8700
28 INT3    MOVEA.L A1,-(A7)
29         MOVEA.L A0,-(A7)
30         MOVE.L  DO,-(A7)
31
32         MOVEA.L #USARTD,A1
33         MOVEA.L #MSG,A0
34         MOVE    COUNT,DO
35
36         MOVE    (A1),(A0,DO)
37         ADD     #1,DO
38         MOVE    DO,COUNT
39
40         MOVE.L  (A7)+,DO
41         MOVEA.L (A7)+,AO
42         MOVEA.L (A7)+,A1
43         RTE

```

L'unica differenza rispetto al caso sincrono è che in questo caso i sistemi devono impostare che la periferica lavora in modalità sincrona, il numero di caratteri di sincronizzazione, e il carattere di sincronizzazione (0xFF).



# Capitolo 3

## Architettura dei processori

I processori che sono utilizzati al giorno d'oggi sono vari e possono essere contraddistinti in base al loro sistema di funzionamento, in base alla tipologia di codici operativi che possono essere utilizzati (ISA - Instruction Set Architecture) e dalla tipologia di architettura adottata (CISC o RISC). In generale, però, l'architettura di un processore, internamente, non cambia, ciò che può cambiare sono le modalità con cui tale processore va ad eseguire le istruzioni in un certo modo. Tale capitolo, quindi, non avrà solo lo scopo di introdurre le architetture dei processori più utilizzate, ma avrà anche lo scopo di definire quali tra le scelte disponibili, sono più efficienti o veloci ed il perché di tali considerazioni.

### 3.1 Generalità sul processore

Il processore, di per se, è un componente atto ad eseguire dei codici operativi predefiniti all'interno della sua ISA. Oltre al codice operativo, in un processore, vi è anche la sua **architettura**, che può essere di vario tipo. In generale un processore, a livello architettonico, è formata dai seguenti componenti:

- **Unità di controllo:** Determina i passi elementari che deve fare un processore al fine di eseguire un'istruzione;
- **Registro Program Counter:** Registro contenente il puntatore alla prossima istruzione;
- **Registro Instruction Register:** Registro contenente l'istruzione che sta venendo eseguita;
- **Registro Memory Address:** Registro di interfacciamento con la memoria, per gli indirizzi;
- **Registro Memory Buffer:** Registro di interfacciamento della memoria, per i dati;
- **Registri dato ad uso generico:** Registri dato e indirizzo presenti nell'architettura;
- **Unità Logico-Aritmetica (ALU):** Unità che permette le esecuzioni, dati gli operandi, di operazioni logico-aritmetiche.

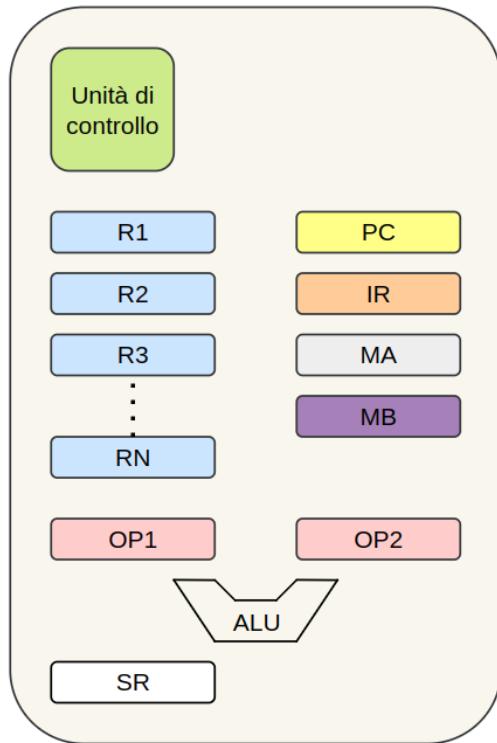


Figura 3.1: Architettura di un processore generico

In particolare, per l'**Unità Logico Aritmetica**, vi è un'interazione anche con un altro registro all'interno della stessa architettura, ovvero il registro di **stato** o **Status Register (SR)**, tale interazione è legata alle caratteristiche principali che può avere un risultato di un'operazione aritmetica (1.2.4).

È possibile visualizzare una pseudoarchitettura del processore all'interno dell'immagine [3.1]

Oltre che la sua architettura interna, il processore è caratterizzato anche dal flusso di esecuzione di una specifica istruzione. Tale flusso può cambiare di architettura in architettura e sarà oggetto di approfondimento per i prossimi paragrafi.

## 3.2 Architettura dei Processori moderni

La classica architettura di un processore, riguardo l'esecuzione delle istruzioni, è poco efficiente, poichè bisogna aspettare sempre il termine di un'istruzione per eseguire quella successiva. Per velocizzare tale tipologia di sistema abbiamo 2 principali strade:

- **Elettronicamente:** Si aumenta la frequenza di clock all'interno del nostro sistema (in gergo si utilizza il termine Overclock). Tale soluzione, per quanto semplice, è molto pericolosa, poichè dopo una certa soglia, non posso più aumentare la frequenza di clock. Aumentare troppo il clock potrebbe danneggiare i componenti per la troppa energia da dissipare e quindi bisognerebbe prevedere anche delle architetture costruite ad-hoc;
- **Architetturalmente:** Vado a modificare l'architettura per gestire un nuovo modo di funzionamento del classico flusso di funzionamento di un processore. Tale modifica permetterebbe di poter eseguire più istruzioni contemporaneamente. Le tipologie di approcci che si possono avere in base a questa soluzione sono 2:
  - **Parallelismo a livello di processo:** Ho a disposizione più processori (parallelismo esplicito) che vanno ad eseguire in maniera concorrente tali processi;
  - **parallelismo a livello di istruzione:** Un singolo processore riesce ad eseguire più istruzioni in maniera parallela;

Le due macrosoluzioni non sono mutuamente esclusive, quindi si potrebbe anche pensare di effettuare una combinazione di esse. Guardando nello specifico alla soluzione di tipo **architetturale** si possono incontrare varie strade per poter implementare il parallelismo delle istruzioni. Per poter meglio comprendere come effettuare la suddivisione del lavoro tramite le varie architetture, bisogna comprendere bene come strutturare un **processo**. Tale entità la possiamo vedere come:

- Formata da più task disgiunti ed indipendenti;
- Formata da un solo programma che richiede un'esecuzione ad elevate prestazioni.

Le architetture che negli anni sono state progettate per la distribuzione del carico di lavoro, dato da un singolo processo sono varie (quindi ci troviamo nel secondo caso). Le tipologie principali sono:

- **SISD (Single Instruction Single Data):** Architettura in grado di eseguire una istruzione alla volta lavorando su dati singoli. Tale tipologia di architettura rispetta per filo e per segno la classica architettura di Von Neumann, la tipologia di parallelismo che si può implementare su tali sistemi è solo tramite un cambio di contesto, quindi definendo uno scheduling.
- **SIMD (Single Instruction Multiple Data):** Architettura progettata per l'esecuzione di una singola istruzione su più dati. Tale tipologia di architettura è molto buona per l'esecuzione di prodotti vettoriali. Il parallelismo di tale macchina è intrinseco rispetto ai dati, poichè si agisce effettuando una singola istruzione sui vari dati.

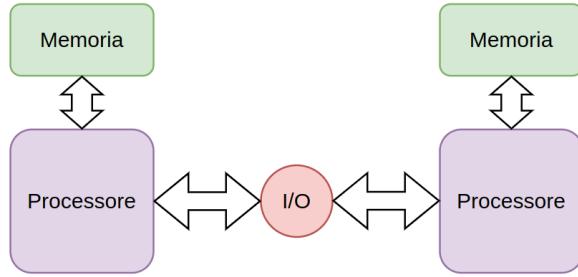


Figura 3.2: Sistemi "gemelli" o sistema multicompiler

- **MISD (Multiple Instruction Single Data)**: (Tali architetture sono state aggiunte solo per conoscenza personale, ma non sono state spiegate dal professore) Architettura in grado di eseguire una moltitudine di istruzioni su di un singolo dato. Tale tipologia di architettura è la meno utilizzata, poiché si cerca di eseguire sempre delle operazioni contemporanee rispetto ai dati.
- **MIMD (Multiple Instruction Multiple Data)**: Architettura che consente di eseguire più istruzioni su più dati. Essi eseguono quindi in parallelo, più istruzioni diverse su più dati diversi. Esse sono le più complicate per via della condivisione della memoria, difatti, ve ne sono varie, in base a come si vadano ad accedere i vari dati in memoria.

Nel nostro caso, il Motorola 68k è una tipologia di sistema **SISD**. Ad oggi i sistemi maggiormente utilizzati sono i sistemi **MIMD**, che prevedono l'utilizzo di più unità di elaborazione per poter determinare uno specifico risultato. Tali sistemi, come detto in precedenza, possono essere di vario tipo e possono essere strutturati in vario modo. Un primo approccio è quello di costruire due sistemi "gemelli", ovvero, costruire due calcolatori differenti che tramite la comunicazione I/O gestiscono le varie operazioni da effettuare. Tale sistema, però, non è molto efficiente, poiché le comunicazioni tramite dispositivi di I/O non è veloce come i processori, per cui si avrebbe un rallentamento delle operazioni. Per sopperire a tale problema, allora si potrebbe pensare di utilizzare un sistema di memoria condivisa tra i due processori, in modo da evitare i dispositivi di I/O. La problematica che si andrebbe a presentare in quest'altro caso sarebbe l'accesso al BUS, che nonostante sia velocissimo, ha bisogno di una buona comunicazione tra i due processori. Per migliorare ancora tale tipologia di architettura, allora, si potrebbe pensare di utilizzare una gerarchia di memorie, che permetta di ridurre gli accessi in memoria pilotati dai vari processori, che possono lavorare sulle loro memoria "private" in maniera del tutto autonoma e concorrente senza dover schedulare l'accesso al BUS. I sistemi che sfruttano le gerarchie di memoria prendono il nome di **sistemi multicore**, che non hanno solo 2 livelli di gerarchia di memoria, ma ne hanno vari, in base ai casi.

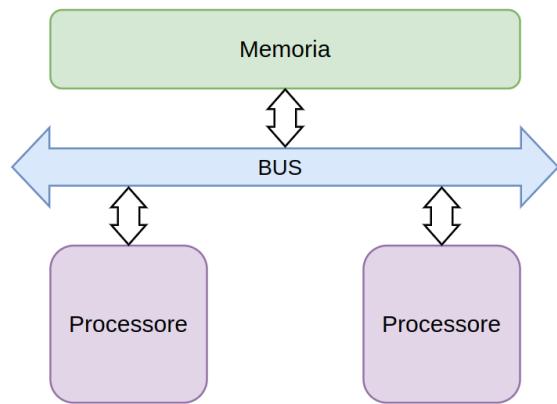


Figura 3.3: Sistema a memoria condivisa

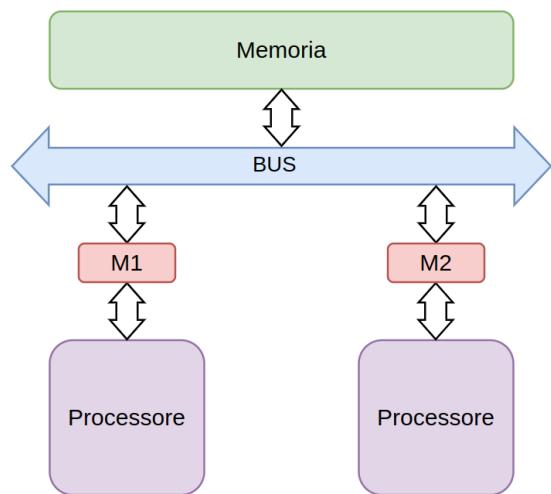


Figura 3.4: Sistema multicore

### 3.2.1 Multi-Computer e Multi-Processore

Concentriamo la nostra attenzione sui sistemi MIMD. Escludendo la possibilità di avere un parallelismo interno, è possibile individuare due categorie di sistemi che permettono a più processori di lavorare su dati diversi, questi sono detti Multi-Computer e Multi-Processore.

Un primo metodo utile a far interagire due (o più) sistemi differenti è introdurre un intermediario, ovvero un particolare tipo di sistema di I/O che sia efficace e veloce. In tal caso, quanto più rapidi saranno i processori, tanto più dovrà esserlo il sistema (oltre che la rete che li interconnette). La limitazione principale di tale modello è la sensibilità ai limiti tecnologici dovuti all'interconnessione. Il sistema globale è detto **Multi-Computer** ed è particolarmente utilizzato per applicazioni che richiedono calcoli dedicati [3.5].

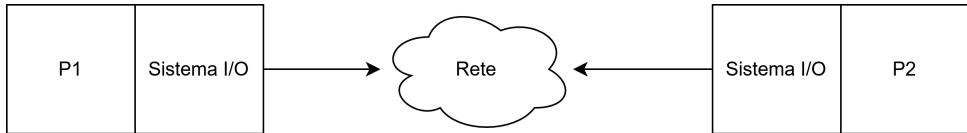


Figura 3.5: Architettura di un sistema Multi-Computer.

Un'altra possibilità per far interagire i processori è direttamente tramite la memoria, ottenendo un sistema complessivo detto **Multi-Processore** [3.6]. In tal caso, la comunicazione tra i processori avviene direttamente tramite bus, superando il problema legato alla fisica realizzabilità di connessioni su larga scala. Il vantaggio di questi sistemi è che i dati possono essere trasferiti molto rapidamente in memoria grazie al bus, mentre lo svantaggio è relativo alla competizione tra i processori negli accessi in memoria. Una possibile miglioria all'architettura proposta consiste nell'integrare a ciascun processore una memoria interna più piccola che gli permetta di gestire le istruzioni. In altre parole, è necessario gestire una gerarchia delle memorie. Potremmo (erroneamente) pensare che aggiungere più core permetta di velocizzare il sistema, in realtà questo è sbagliato perché complicherebbe notevolmente le connessioni del bus, il quale diventerebbe il collo di bottiglia del modello. L'esistenza di un modello non esclude la possibilità di inserirne un

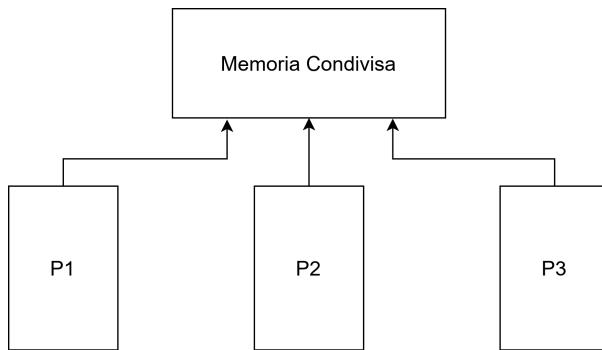


Figura 3.6: Architettura di un sistema Multi-Processore.

altro, cosa che accade tipicamente nei sistemi moderni.

### 3.2.2 Speed Up

La legge di **Amdhal** afferma che il tempo di calcolo di un programma è dato da:

$$t_s = t_{ss} + t_{sp}$$

Ovvero, la somma tra il tempo di esecuzione di una parte del programma che è strettamente seriale e il tempo di esecuzione della parte del programma che è parallelizzabile. In altre parole, la legge dice che una macchina parallela può essere veloce quanto si vuole, ma ci sarà sempre una parte del codice che non è parallelizzabile e che quindi rallenterà l'esecuzione. Un parametro che ci permette di valutare se risulta conveniente ricorrere a un'architettura parallela è lo **speed-up**, che rappresenta il guadagno che si quantifica con una macchina parallela rispetto all'utilizzo di una macchina puramente sequenziale:

$$S = \frac{T_{SEQ}}{T_{PAR}} \rightarrow N(\text{teorico})$$

Dove  $N$  è il numero di unità di calcolo. Il limite riportato è puramente teorico poiché non tiene conto dell'interazione necessaria fra i nodi, che richiede sicuramente ulteriore tempo. Volendo quindi essere più precisi, dunque, la formula precedente va corretta nel seguente modo:

$$S = \frac{T_{SEQ}}{T_{SEQ}/N + T_{INT}} = \frac{1}{1 + \frac{T_{INT}}{T_{SEQ}}N} N(*)$$

Dove  $T_{INT}$  è il tempo di interazione citato. Dalla seconda espressione, è evidente che per convergere allo speed-up teorico è necessario che il termine  $\frac{T_{INT}}{T_{SEQ}}N \ll 1$ . Possiamo ulteriormente modificare il termine:

$$\frac{T_{INT}}{T_{SEQ}}N = \frac{T_{INT}}{\frac{T_{SEQ}}{N}} = \frac{T_{INT}}{T_{CalcP}}$$

Dove  $T_{CalcP}$  è un termine che ingloba il solo calcolo parallelo senza interferenze. Da questo, è evidente (come potevamo immaginare) che se si calcola poco e si spreca molto tempo nelle interazioni risulta impossibile convergere allo speed-up teorico. L'equazione (\*) è fondamentale, in quanto lega il problema (tempo di calcolo parallelo, quanto calcolo bisogna fare) con la tecnologia (tempo di interazione necessario per scambiare i dati, dipendente dalla velocità della rete). Una facile conseguenza di questo è che se abbiamo dei processori molto lenti e bisogna fare molto calcolo, la rete può essere anche poco veloce perché in questa situazione non si trae alcun vantaggio dalla velocità della rete, se invece bisogna fare poco calcolo e i processori sono veloci, allora la rete deve essere veloce. Questo concetto viene in genere chiamato *grain size*.

Altro parametro spesso usato per valutare la convenienza di un'architettura parallela è l'efficienza, definita come il rapporto fra lo speed-up ed  $N$ :

$$\epsilon = \frac{S}{N} = \frac{1}{1 + \frac{T_{INT}}{T_{CalcP}}} = \frac{1}{1 + a} \approx 1 - a \rightarrow 1$$

In tutti i rami dell'ingegneria l'efficienza (o rendimento) rappresenta sempre "*quanto si guadagna rispetto a quanto si spende*". Se  $a$  è piccolo l'efficienza è approssimabile secondo Taylor ad  $1 - a$ , da questo si capisce che l'efficienza è legata al rapporto tra il tempo di interazione ed il tempo di calcolo parallelo. In più, se il tempo di interazione fosse (idealmente) 0, si raggiungerebbe il limite teorico dell'efficienza (1). Tuttavia, come abbiamo visto questo risulta impossibile secondo Amdhal.

Ricordiamo che per poter eseguire un programma con la programmazione parallela, e quindi risolvere un problema che di solito ha una soluzione sequenziale in modo parallelo,

devono essere valide la *proprietà commutativa* e la *proprietà associativa*. Queste due proprietà assicurano la possibilità di poter scomporre e calcolare separatamente le soluzioni a sottoproblemi, per poi ricomporle ottenere la soluzione finale.

Per mostrare un'esempio applicativo, consideriamo le due matrici  $A$  e  $B$  con dimensione  $4 \times 4$ , e ipotizziamo di volerne fare il prodotto riga per riga. Quindi, l'elemento  $C_{ij}$  della matrice risultato  $C$ , è il prodotto scalare fra due righe delle matrici:  $C_{ij} = R_{iA} \cdot R_{jB}$ . Le due proprietà associativa e commutativa valgono senza dubbio, per cui il prodotto matri- ciale è sicuramente adatto per il calcolo parallelo. Inoltre, non c'è bisogno di una fase di comunicazione fra i diversi nodi. Potremmo suddividere il lavoro tra due macchine dando ad ognuna la matrice  $B$  e una coppia di vettori riga di  $A$  (ad esempio  $A_1, A_2$  e  $A_3, A_4$ ). In questo caso, il costo di interferenza è 0 e proprio per questo motivo l'efficienza è 1. Se aumentiamo il numero di processori da 2 a 4, affidando ad ogni processore il compito di fare il prodotto tra  $B$  ed un unico vettore di  $A$ , l'efficienza resta 1, quello che cambia è lo speed-up. Per il caso a due processori è 2, mentre diventa 4 nel caso a quattro processori. Da questi due semplici casi si potrebbe pensare di aumentare il numero di processori così da ottenere uno speed-up sempre crescente. La considerazione è certamente vera, però c'è lo svantaggio dell'occupazione della memoria, in quanto ad ogni processore bisogna affidare una copia della matrice  $B$  ed un vettore di  $A$ . In generale bisogna sempre fare i conti con le risorse disponibili nel sistema, mentre la soluzione proposta sembra ignorare del tutto questa regola.

Una seconda possibile soluzione al problema proposto consiste nel suddividere entrambe le matrice sulle due macchine a disposizione. Nonostante questo permetta di risparmiare in termini di memoria, ci sarà sicuramente un'influenza sull'efficienza. In tal caso, le due macchine devono scambiarsi informazioni sulle righe delle matrici per completare il calcolo, per cui sarà necessario un tempo di interazione. In questa soluzione, il DMA si rende sicuramente utile.

Una terza ed ultima soluzione consiste nell'utilizzo di un'architettura su più nodi (multi-computer), su ciascuno dei quali possiamo risolvere i prodotti  $C_{ii} = A_{ii} \cdot B_{ii}$ . Il problema è che in tal modo stiamo popolando la sola diagonale principale della matrice  $C$ , per poter ottenere il risultato completo è necessario ruotare in ognuno dei nodi la riga di  $B$ .

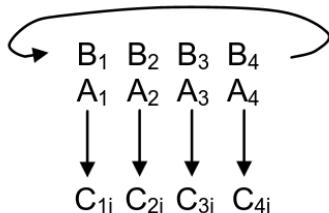


Figura 3.7: Calcolo parallelo della matrice  $C$  su 4 nodi.

### 3.2.3 Coerenza della memoria cache nelle architetture parallele

In generale, l'obiettivo che si pone nella realizzazione di un'architettura è quello di diminuire i costi di interazione senza agire sul software, ma solo sull'hardware. Ricordiamo che il costo di interazione è il carico introdotto dalla comunicazione per lo scambio di

informazioni che avviene tra due o più processi.

Per un sistema multiprocessore, tale costo è determinato dal carico a cui è sottoposto il bus comune con cui i processi interagiscono per accedere alla memoria condivisa. Al fine di diminuire l'iterazione tra i processi, è necessario introdurre, come sarà discusso nel capitolo sulle memorie [??], una gerarchia di memorie. Ciò si traduce nel fornire ai diversi processori una propria memoria più veloce di quella comune, quindi una *memoria cache* [3.8]. Con la soluzione illustrata, però, nasce un problema di gestione della **coerenza**

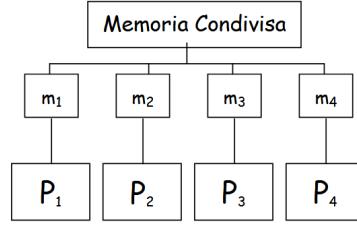


Figura 3.8: Sistema multiprocessore con gerarchia delle memorie.

**della memoria** condivisa. Non tanto per le istruzioni eseguite dai diversi processori, le quali devono essere solamente lette, ma per i dati condivisi che possono essere modificati. Il problema può essere affrontato tramite due strategie:

- **Write Through:** Ogni qual volta un dato viene modificato, il suo valore viene immediatamente aggiornato in memoria condivisa.
- **Write Back:** Le modifiche non sono riportate immediatamente, ma in un secondo momento.

Per entrambe le tecniche, si necessita quindi di una politica di *invalidazione dei dati modificati* da un processo, i quali possono essere presenti nelle cache di altri processori o della memoria comune. Ovviamente, nel caso della write through, la memoria condivisa viene aggiornata immediatamente, per cui non possiede mai valori da invalidare. Tutto ciò porta a un'ulteriore sovraccarico del bus comune.

Concentriamoci prima sulla soluzione adottata da write through. Per capirne il funzionamento, supponiamo che ci siano i due processori  $P_i$  (spia) e  $P_j$  (spiato) che operano sugli stessi dati. In particolare, osserviamo il processo dal punto di vista del processore spia a fronte delle azioni compiute sui dati sia da se stesso che dall'altro processore. Ogni dato presente in cache, può assumere solo due stati: *valido* e *non valido* [3.9]. Passiamo

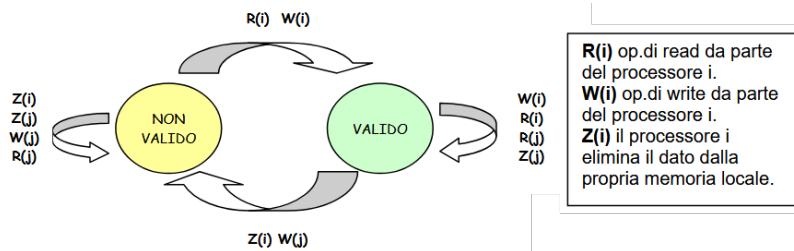


Figura 3.9: Automa per la gestione della coerenza in write through.

ora a considerare il caso write back, in cui bisogna tenere conto che l'aggiornamento del dato in memoria comune non avviene subito ma in un secondo momento.

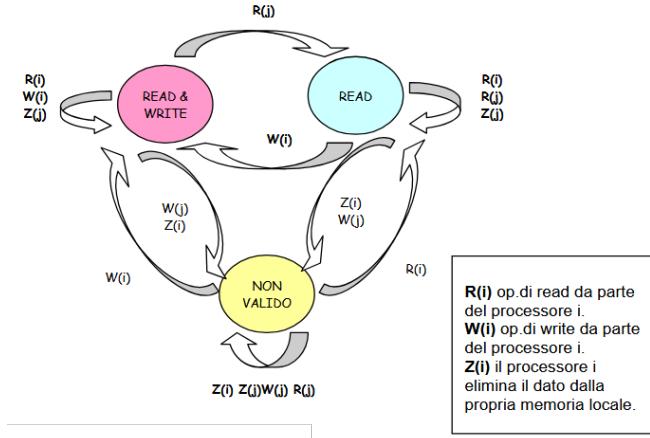


Figura 3.10: Automa per la gestione della coerenza in write back.

Lo stato  $RW$  è quello in cui  $P_i$  può effettuare qualunque tipo di operazione sul dato. Se  $P_j$  legge il dato, si va in uno stato  $RO$  (read only), in cui si deve tener conto del fatto che adesso anche altri processori stanno leggendo il dato in questione e affinché questo sia possibile bisogna effettuare un flush in memoria comune. Da questo stato non si esce mai fintantoché i vari processori si limitano a leggere il dato. Una eventuale riscrittura da parte di  $P_i$  riporta allo stato  $RW$ . Questa operazione corrisponde a invalidare il dato dal punto di vista di  $P_j$ . Possiamo rendercene conto osservando il grafo: sia dallo stato  $RW$  che dallo stato  $RO$ , l'operazione  $W(j)$  porta sempre nello stato  $NV$  dal punto di vista di  $P_i$ . Da qui si capisce che entrambi gli stati ( $RO$  e  $RW$ ) rappresentano la validità del dato in memoria. Quindi, in regime write back una qualunque operazione di scrittura dal parte di un processore rende il dato invalido per tutti gli altri, come d'altra parte è ovvio. Da uno stato invalido si può tornare se  $P_i$  legge o scrive il dato. In particolare,  $R(i)$  porta in  $RO$ , e  $W(i)$  in  $RW$ . Ci si potrebbe domandare perché sono stati distinti i due stati, entrambi validi,  $RW$  e  $RO$ . Il motivo è che nel caso in cui da  $NV$  si passa a  $RW$ , il processore  $P_i$  si limita ad aggiornare nella propria memoria locale. Se invece si passa ad  $RO$ , si forza il sistema di gestione a prelevare il dato dalla memoria centrale e a scaricarlo nella cache di  $P_i$ . Le due situazioni sono quindi sostanzialmente diverse. Da quanto detto, lo stato che risulta più critico per il sistema è quello di  $RW$ , dove i dati modificati dal processore  $P_i$  sono validi solo per se stesso, e quindi non allineati alla memoria condivisa.

Oltre alle tecniche standard di write through e write back, è stata ideata un'ulteriore tecnica detta di *write once*, nella quale la prima volta viene eseguito il write through e se in seguito è necessario effettuare altre scritture sullo stesso dato, si passa alla modalità write back, in modo da evitare di appesantire il traffico sul bus. Osserviamo che l'automa a stati finiti per la write once richiede 4 stati [3.11]. Un'ultima tecnica per la coerenza fa uso di *puntatori*. È presente una shared memory che indica in ogni istante in quale cache è presente un certo dato, e i puntatori consentono di accedervi. Quando una cache invalida il dato, invece di spostare fisicamente i dati, si possono usare i puntatori per creare delle catene logiche che consentono di andare a prendere sempre la versione corretta del dato. Osserviamo che la macchina non sa e non deve preoccuparsi di dove si trovano i dati validi, tutto è gestito dall'hardware.

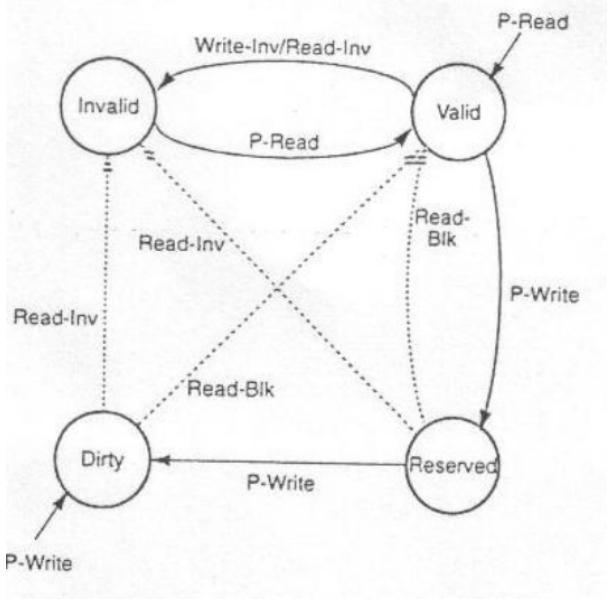


Figura 3.11: Automa per la gestione della coerenza in write once.

### 3.3 Topologia di interconnessione

Poniamoci ora nel caso di architetture multicomputer, in cui a differenza di quelle multiprocessore, non esiste memoria condivisa né un bus comune, ma la comunicazione tra i processori dipende dalla **topologia di rete**, in particolare dal diametro (massima distanza tra due nodi). Questo rende più complesso ridurre i costi di comunicazione, poiché spesso non è possibile ottenere una totale connettività. Un’ulteriore difficoltà riguarda l’allocazione dei processi: quando i processi sono più dei processori, è necessario bilanciare la comunicazione e l’uso delle risorse. Idealmente, si dovrebbero:

- Assegnare allo stesso nodo i processi che comunicano spesso.
- Distribuire su nodi diversi quelli che richiedono molte risorse computazionali.

Per ottimizzare l’allocazione servono funzioni obiettivo specifiche, ma la loro definizione è complessa perché dipende dai dati trattati dai processi.

Nell’ottica in cui ci poniamo (punto di vista puramente hardware), la configurazione descritta dall’architettura non può risolvere il problema dall’allocazione dei processi, ma può far fronte a quello della connettività, cercando di rendere quanto più semplice possibile la comparazione tra grafo del problema e quello dell’architettura. Esistono due soluzioni, le *reti dirette* e *reti indirette*.

Le **reti dirette** definiscono una topologia fissa, per cui la compatibilità tra i grafi è realizzata in maniera analitica attraverso l’utilizzo di formule provenienti dal campo della ricerca operativa. Le **reti indirette**, invece, sono realizzate attraverso particolari dispositivi chiamati *switch*, che consentono di ottenere una totale connettività attraverso un sistema dinamico di comunicazione. Per capire meglio come questo sia possibile, è necessario approfondire alcuni dettagli di realizzazione sugli switch. Innanzitutto, l’obiettivo principale di questi dispositivi è gestire dei conflitti, ovvero di situazioni in cui due o più

nodi cercano di comunicare contemporaneamente con un altro. A questo proposito, è possibile classificare gli switch in due categorie:

- **A singolo stadio** (Diretti).
- **A N stadi** (Indiretti).

Per creare uno **switch a connessione diretta** tra due nodi diversi, è necessario avere due informazioni: l'indirizzo del nodo sorgente e quello del nodo destinazione. Il dispositivo che si vuole realizzare dovrà avere  $N$  nodi in ingresso ed  $N$  nodi in uscita. Per poter implementare tale architettura si deve scomporre il sistema in due sottosistemi diversi, quello di ingresso e quello di uscita. Tale suddivisione viene realizzata mediante multiplexer e demultiplexer, entrambi generalmente realizzati in logica three-state [3.12]. L'inconveniente隐含的 del meccanismo implementato, è che automaticamente si rea-

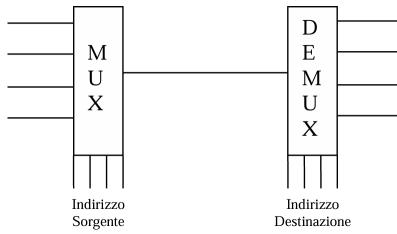


Figura 3.12: Switch a singolo stadio.

lizza una mutua esclusione tra i nodi dovuta al fatto che, potrà essere attivo un solo collegamento per volta. Quindi, per realizzare un collegamento di  $N/2$  comunicazioni simultanee, si dovrà replicare l'hardware  $N/2$  volte. Le caratteristiche che rendono favorevole questa soluzione sono: assenza di stadi intermedi, buona efficienza, basso parallelismo e parallelismo fisico ottenuto riproducendo lo stesso elemento più volte.

La soluzione trovata non è esente dal problema dei conflitti, poiché aumentando il parallelismo in hardware si va semplicemente a rendere possibile la comunicazione parallela, ma se i nodi di ingresso vogliono comunicare con lo stesso nodo di uscita è inevitabile che si creano collisioni. Si rende necessario, quindi, predisporre un gestore dei conflitti. La soluzione con **switch indiretti**, si basa su una serie di stadi intermedi per stabilire la comunicazione tra più nodi. Sono state proposte nel tempo varie topologie per implementare tale logica, ad esempio è facile rendersi conto che, utilizzando dispositivi a due ingressi e due uscite per realizzare gli stadi, con  $\log_2 n$  stadi è possibile risolvere il problema della connettività, avendo tra l'altro il vantaggio di usare blocchi molto semplici che devono solo decidere se procedere in una direzione o in un'altra. Ovviamente esiste un problema non banale, come si fa a collegare i blocchi tra di loro (indirizzamento)? Per switch a connessione diretta l'indirizzamento è semplice perché gli indirizzi vanno direttamente nel multiplexer e nel demultiplexer per creare il path. Nel caso di architetture multistadio invece, l'indirizzo disposto in  $k$  bit, deve essere diviso in tante parti quanti sono gli stadi, in modo da comandare opportunamente i blocchi. Questo significa che non esiste una logica di controllo centralizzata come nel caso precedente, ma la logica di controllo è distribuita negli stadi. Ovviamente è una condizione abbastanza forte dal punto di vista dell'architettura, che è possibile realizzare soltanto utilizzando particolari proprietà topologiche della rete. Un esempio che implementa una soluzione molto interessante è rappresentata dalla **Omega Network**, la quale si basa sul concetto del *perfect*

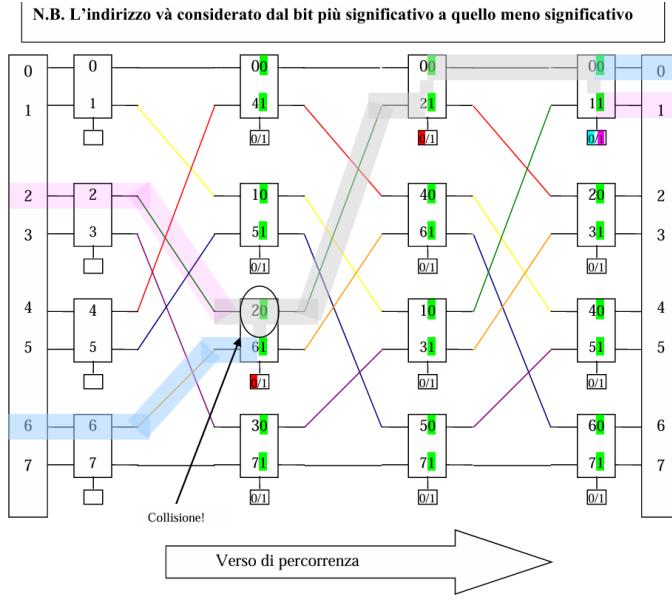


Figura 3.13: Switch a  $N$  stadi con omega network.

*shuffling* [3.13]. Il vantaggio evidente, come detto prima, risiede nella semplicità dei componenti usati per realizzare il dispositivo che implementa il routing, ovvero blocchi con due ingressi e due uscite. L'unica nota negativa, invece, risulta proprio l'utilizzo di più stadi. Va notato che, la flessibilità indotta dall'architettura, grazie alla forte modularità dei componenti, rende lo switch molto scalabile, infatti è facile realizzarne uno di grandi dimensioni, cosa che non è semplicemente possibile per gli switch diretti. Il problema che si verifica è, però sempre lo stesso, se due nodi di ingresso vogliono parlare con lo stesso nodo di uscita simultaneamente generano un conflitto, inoltre la contesa può anche avvenire nei nodi intermedi. Per comprendere come gestire tali conflitti, ricordiamo che la comunicazione può avvenire secondo due modalità:

- **Wormhole:** Il pacchetto viene suddiviso in piccole unità, appena la prima (di solito l'header) arriva a uno switch, può essere subito inoltrata anche se il resto del pacchetto non è ancora arrivato. Lo svantaggio è che può provocare deadlock se molti pacchetti competono per le stesse risorse. I vantaggi sono una latenza più bassa (opo che l'header ha conquistato il canale i successivi frammenti viaggiano senza ritardi) e buffer di dimensioni minori. Cosa accade se il flusso di frammenti si blocca, ad esempio a causa di un conflitto? Esistono quattro soluzioni:
  1. **Blocco conservativo:** Il messaggio si ferma finché il problema non si risolve. È semplice ma può bloccare molti nodi.
  2. **Distruzione del messaggio:** Il messaggio viene eliminato. Rete con possibile perdita di dati.
  3. **Re-instradamento:** Si cerca un percorso alternativo, ma è possibile solo in reti non statiche e con topologie flessibili (quindi non per omega network). Inoltre, in questo caso i pacchetti possono arrivare fuori ordine, quindi serve una strategia software che si occupi di riordinarli.
  4. **Cut-through:** Il nodo blocca memorizza temporaneamente i dati, mischiando wormhole e store and forward. Serve però un minimo di buffer (es. shift-register e contatore).

- **Store & Forward:** Ogni nodo intermedio deve ricevere interamente il pacchetto, e solo dopo lo memorizza (store) e lo inoltra (forward) al nodo successivo. Gli svantaggi sono i tempi di ritardo e la necessità di buffer per memorizzare le informazioni.

La differenza chiave tra i due è quindi che nel wormhole c'è una negoziazione del percorso, il quale è dinamico e dipende dal primo frammento, mentre nello store and forward la negoziazione di tratta, ovvero avviene a ogni stadio.

## 3.4 Sistema Pipeline

Una architettura pipeline è una tipologia di soluzione che viene implementata internamente nei processori per permettere l'esecuzione parallela di più istruzioni. Per comprendere meglio cosa si intende per esecuzione parallela di più istruzioni, andiamo a considerare la strutturazione interna del flusso di esecuzione di una normale istruzione. L'esecuzione si divide nelle seguenti fasi (visualizzabili anche alla figura [3.14]):

- **Istruction Fetch (IF)**: Ovvero il prelievo dell'istruzione dalla memoria;
- **Decode (DEC)**: Decodifica ed interpretazione dell'istruzione;
- **Execute (EX)**: Esecuzione effettiva dell'istruzione;
- **Memory write (MW)**: Effettuo il prelievo dei dati all'interno della memoria;
- **Register write (RW)**: Inserisco i dati che mi interessano dai registri, in memoria.

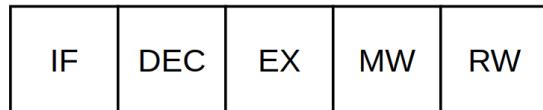


Figura 3.14: Architettura classica pipeline (ordine di esecuzione fasi)

Definito il flusso di esecuzione di un'istruzione, l'architettura pipeline cerca di eseguire le fasi di un'istruzione in maniera del tutto separata, in modo da poter eseguire più fasi di istruzioni differenti, aumentando il throughput di 5 volte rispetto alla classica architettura sequenziale di Von Neumann. Per capire meglio questo concetto facciamo un esempio: Devo eseguire un programma che ha 5 istruzioni, allora parto con l'esecuzione della prima fase, che preleverà l'istruzione  $i_1$ , una volta prelevata, l'istruzione  $i_1$  passa alla fase di decode, mentre, allo stesso istante, viene caricata l'istruzione  $i_2$  (quindi viene effettuata la Istruction Fetch dell'istruzione successiva mentre viene decodificata la precedente). Se eseguiamo tale procedura per tutte le fasi noteremo che ad ogni impulso di clock (a regime), il processore darà un risultato, quindi non ci sarà bisogno di eseguire tutte le istruzioni una per volta, poiché la suddivisione delle fasi ne permette un'esecuzione "parallela" (o meglio dire pipeline).

Per fare in modo di isolare l'esecuzione delle varie fasi, tra i vari blocchetti saranno presenti dei registri, che permettono di conservare lo stato su cui una determinata fase sta lavorando (come le architetture pipeline in elettronica [3.15]).



Figura 3.15: Architettura classica pipeline con registri

Esistono tuttavia dei limiti dell'architettura pipeline la cui risoluzione verrà discussa (almeno per quelli che ci riguardano) nei prossimi paragrafi:

- **Vincolo di tempo:** Le fasi presentate operano in pipe, quindi funzionano bene se tutte le unità funzionali impiegano un tempo più o meno costante. Se così non fosse, l'intero processore sarà rallentato dall'unità funzionale più lenta. La condizione di tempo di risposta costante è garantita solo se le istruzioni sono particolarmente semplici (condizione tipica dei sistemi RISC);
- **Vincolo tecnologico:** Tutte le unità funzionali sono pilotate contemporaneamente dallo stesso clock. Il problema tecnologico è che il clock può essere schematizzato elettronicamente come un circuito induttivo/capacitivo, e di conseguenza le ultime unità a percepire il segnale di clock, se la frequenza del treno di impulsi è troppo elevata, troppo *filtrato* (a causa del filtro capacitivo) o troppo rumoroso (a causa dell'effetto induttivo). Questo fenomeno prende il nome di Skew del segnale di clock, e dunque è opportuno adottare delle misure di ridondanza e controllo correttezza a monte di ogni collegamento unità funzionale-clock;
- **Problema delle istruzioni di salto:** Quando si incontra un'istruzione di salto, bisogna eseguirla completamente (fine fase EX), prima di sapere se il salto deve essere effettuato o meno e prima di conoscere l'indirizzo della prossima istruzione da prelevare. La filosofia generale è quella di non fermare la pipe, e quindi continuare a caricare istruzioni in modo sequenziale, e all'occorrenza eliminarle dalla pipe (*branch penalty*) (3.4.3);
- **Problema del conflitto sui dati:** Spesso accade che due istruzioni consecutive cercano di leggere e scrivere contemporaneamente lo stesso registro e ciò potrebbe causare gravi errori. Nei processori CISC i conflitti sui dati sono più probabili, siccome quest'ultimi tendono a garantire tutti i modi di indirizzamento per tutte le istruzioni. Il conflitto sui dati è confinato alle ultime tre fasi: EX, MEM e WB;
- **Problema della gestione delle interruzioni:** In un sistema pipeline è più complesso gestire sia le interruzioni esterne che garantire il corretto ordine di esecuzione delle interruzioni interne;
- **Problema dei conflitti per l'accesso in memoria:** Più fasi possono tentare di accedere contemporaneamente alla memoria, per eseguire operazioni di lettura delle istruzioni o scrittura sui dati.

Un'architettura di questo tipo, però, richiede una serie di ipotesi, ovvero:

- Divisione di memoria dati e memoria istruzioni (altrimenti dovrei gestire anche dei conflitti tra l'istruzione fetch e la register/memory write);
- Le memorie devono garantire degli accessi molto veloci: ad ogni colpo di clock vengono effettuate 5 fasi e potenzialmente scritto un risultato.
- Le operazioni aritmetico-logiche devono essere effettuate prevalentemente tra i registri interni del processore, condizione che caratterizza fortemente i processori RISC;
- Le istruzioni devono avere tutte lunghezza fissa;

Guardando le ipotesi possiamo capire che alcune tipologie di operazioni, che solitamente effettuevamo sul processore motorola, ora dovranno essere scompattate in varie operazioni. Un esempio classico è il comando **ADD VAR,D1**, che utilizzava l'indirizzamento diretto per il prelievo dell'operando dalla memoria. In questo caso, però, l'indirizzamento diretto non è possibile, poichè si andrebbe ad invalidare un'ipotesi, ovvero, la lunghezza fissa dell'istruzione (che dovrebbe poi contenere l'indirizzo di memoria). Pertanto non sono previsti tutti i modi di indirizzamento.

Per comprendere meglio la problematica, consideriamo di avere un prelievo dalla memoria con un'architettura a 16-bit, ma con il memory address ed il memory buffer a 32-bit. Pertanto il seguente comando non sarebbe possibile: **MOVE VAR,D0**; poichè richiederebbe il prelievo dell'indirizzo di memoria da 32-bit dalla memoria, ma per effettuare tale operazione, avendo solo 16 bit, avrei bisogno di due istruzioni che caricano, una i primi 16-bit e l'altra i restanti 16. Tale suddivisione, però non viene fatta dal programmatore, ma dal compilatore. Ci sono varie istruzioni che sono come la MOVE, tali istruzioni sono dette pseudo-istruzioni, poichè il compilatore andrà a suddividerle in più operazioni differenti al fine di raggiungere il risultato desiderato.

Questa cosa ci permette di capire, a questo punto, la suddivisione tra architettura di tipo CISC e architetture di tipo RISC. Le architetture di tipo CISC permettono l'esecuzione di istruzioni che sono più articolate, ma a costo di una complessità architetturale maggiore, mentre nelle architetture RISC, data la semplicità dell'architettura, le tipologie di operazioni che si possono effettuare sono ridotte ma più veloci.

### 3.4.1 Modelli di sistemi pipeline

Il sistema pipeline, dato il suo sistema di funzionamento, può introdurre varie tipologie di problematiche. Negli anni si sono sviluppate varie tipologie di soluzioni differenti. Le principali architetture con cui si va a contatto al giorno d'oggi sono:

- **MIPS**: Tipologia di ISA sviluppata da Patterson che poi ha venduto, per cui ora la sua implemetazione è proprietaria;
- **RISC-V**: Tipologia di ISA molto simile al MIPS, ma open-source;
- **ARM**: Tipologia di ISA proprietaria, utilizzatissima in svariati amiti (particolarmente in quello industriale), la cui implementazione è proprietaria;

Nel caso particolare di questo corso, andremo a vedere il funzionamento del RISC-V facendo riferimento sempre al MIPS, per cui saranno queste le due tipologie di architetture che si andranno ad approfondire.

Le principali problematiche che bisogna affrontare all'interno di un architettura pipeline sono le seguenti:

- **Interruzioni**: Quando bisogna gestire un' interruzione la gestione dell'architettura pipe si complica, poichè bisogna capire chi ha interrotto e bisogna salvare lo stato di tutte le istruzioni che stanno eseguendo, che risulta una cosa molto onerosa e complicata;
- **Concorrenza sui registri**: Se due istruzioni, devono utilizzare un'informazione presente nello stesso registro ad esempio:  $R1 = R2 + R3$ ;  $R0 = R1 + R4$ ; Notiamo che per eseguire la seconda istruzione vi è bisogno del completamento della prima, ma

il risultato effettivo viene scritto solo alla fine del ciclo, per cui si potrebbe incorrere in vari errori;

- **Salti:** Quando devo effettuare un salto, se considero il caso condizionato, non so a quale ramo andrò a saltare, è quindi più complicato capire quale sarà l'istruzione successiva da eseguire;
- **Gestione delle pipe multiple:** ho molteplici pipe di esecuzione, che quindi richiede una loro gestione per prevenire eventuali conflitti;

Una delle problematiche che maggiormente incide è quella riguardante il salto, poichè, dato che non conosco quale sia l'istruzione successiva, vado a bloccare la pipe appena noto che ho un'istruzione di salto, ed appena è verificata la condizione, vado a prelevare tale istruzione dalla memoria. Però questa soluzione risulta molto inefficiente, poichè introduce dei periodi in cui la pipe rimane in stallo. Difatti, una soluzione che è stata trovata è quella della branch prediction, per cui vado a processare le istruzioni successive, cercando di prevedere quale sarà il branch da eseguire, solo nel caso in cui mi accorgo che sto sbagliando vado ad effettuare il blocco della pipe, altrimenti continuo con la normale esecuzione del programma.

### 3.4.2 Registri Intermedi

I registri che sono posti tra una fase e l'altra della pipeline sono molto più complessi di quel che si crede. Essi non contengono solo dati informativi (operandi e risultati), ma contengono anche: traccia dell'operazione da effettuire, destinazione del risultato ecc. L'architettura e le componenti di cui si è parlato sopra, quindi, sono solo una parte di quello che è effettivamente stato realizzato sull'hardware del dispositivo. La schematizzazione che abbiamo fatto ci permette di capire bene il funzionamento di un sistema pipeline senza entrare troppo nei dettagli della sua implementazione hardware.

### 3.4.3 Gestione dei Salti

L'architettura pipeline ha una criticità di cui abbiamo già attentamente discusso in precedenza, i salti. Quando sopraggiunge un'istruzione di salto, riusciamo a captare che è così solo nella fase di esecuzione dell'istruzione, mentre la pipe ha continuato a caricare le istruzioni in modo sequenziale, che si troveranno rispettivamente nella fase di IF e ID e non avranno quindi ancora modificato lo stato del processore e della memoria. Nel caso in cui il salto non deve essere eseguito, allora la pipe continuerà a funzionare normalmente, mentre se il salto deve essere eseguito, le istruzioni caricate dovranno essere eliminate dalla pipe (*pipe flush*) e dovrà essere caricata l'istruzione a cui punta il salto e le successive. Il ritardo che segue quest'ultimo caso è detto *branch penalty*. L'obiettivo è quello di confinare la gestione dei salti nelle fasi IF e ID, perché in quelle fasi le istruzioni non hanno ancora modificato lo stato del processore e della memoria, e quindi la rete di controllo hardware deve riguardare solo queste due fasi. In generale, il problema è risolvibile fondamentalmente mediante due approcci: l'approccio conservativo e l'approccio ottimistico (*branch prediction*). Nel caso dell'approccio conservativo, quando il processore interpreta durante la fase ID un'istruzione come istruzione di salto, ferma la pipe e disabilita la propagazione dell'istruzione che si trova erroneamente nella fase IF, determina l'istruzione a cui saltare in fase EX e la preleva. Si torna insomma al modello sequenziale di Von

Neumann. Il conto è salato se consideriamo che le istruzioni di salto costituiscono il 25% di un programma, e infatti ne consegue un notevole spreco delle risorse di parallelismo fornite dalla pipe. Questo approccio è *leggermente* migliorabile attraverso l'hardware, anticipando la decisione inerente al salto alla fase di ID, in base alla condizione di salto. Ad esempio l'istruzione **JNZ <LABEL>** controlla se il flag Z del SR è alto, e in tal caso non occorre saltare e quindi l'istruzione che si è prelevata nel frattempo è giusta. Molti ritardi possono essere evitati dal programmatore o dal compilatore: il programmatore può contribuire al buon funzionamento del sistema, scrivendo le istruzioni in un ordine tale da minimizzare le probabilità di stallo della pipe. Nel caso di un costrutto if-then-else, ad esempio, conviene inserire nel ramo then l'alternativa più probabile. Il compilatore, da parte sua, può operare vari accorgimenti; ad esempio, siccome un ciclo for è sempre tradotto in un if-then-else, esso deve inserire l'alternativa più probabile nel ramo then. In fase di compilazione è possibile evitare l'approccio conservativo. Se immediatamente prima del salto c'è un'istruzione con operandi da cui non dipende l'esito della scelta di saltare o meno, è possibile in fase di compilazione inserire l'istruzione indipendente dopo il branch, in modo da sfruttare lo slot di tempo in cui l'istruzione viene caricata ed entra nella pipe, e quindi non c'è bisogno di pipe flush. Qualora non sia possibile invertire una istruzione if con quella che la precede, il compilatore potrebbe decidere di mettere subito dopo la if una **nop**, istruzione che non ha alcun effetto e quindi non è mai errato inserirla nella pipe. In questo modo si può operare senza considerare alcun tipo di approccio. Possiamo aggiungere che se stiamo considerando un'istruzione di salto in un processore CISC con una condizione di salto elaborata, allora il numero di nop che bisogna inserire a seguito dell'istruzione di salto risulta essere superiore a 1, in quanto ricordiamo che la pipe per i CISC è più lunga e la valutazione della condizione coinvolge più fasi oltre le prime due, di caricamento e decodifica. Una soluzione meno conservativa a quella appena presentata è di utilizzare la **branch-prediction** (approccio ottimistico).

**Branch prediction** La branch prediction è una tecnica che cerca di prevedere a quale ramo un'istruzione di salto condizionato possa saltare. Tale tecnica valuta le prestazioni in base a quanto si possa "perdere" in termini di efficienza (ricaricare il branch corretto rispetto a quello predetto). Per capire bene questa cosa andiamo a considerare il seguente pseudocodice:

```

1 for (int i = 0; i < N; i++) {
2     for (int j = 0; j < N; j++) {
3         operazioni
4     }
5 }
```

Tali for prevedono un controllo iniziale sulla variabile. Vedendo come sono strutturati tale controllo prevede l'esecuzione del ramo else una sola volta (guardando il for interno) ogni N passi. I modi per prevedere il branch possono essere vari, e possono essere descritti mediante degli appositi automi a stati finiti. Un primo approccio molto basilare è quello di andare a cambiare il branch da caricare successivamente ad ogni errore di decisione e quindi eseguire le operazioni descritte dall'automa [3.16].

Tale soluzione, però, guardando al nostro caso, non è proprio ottimale, poiché per quel singolo fail che avviene ad ogni N interazioni dovrò assorbirmi 2 fault. Per evitare tale condizione, e quindi rendere la persistenza più forte, vado a costruire un automa a 4 stati che mi permette di rendere la condizione di "cambio del branch" più solida, poiché solo in

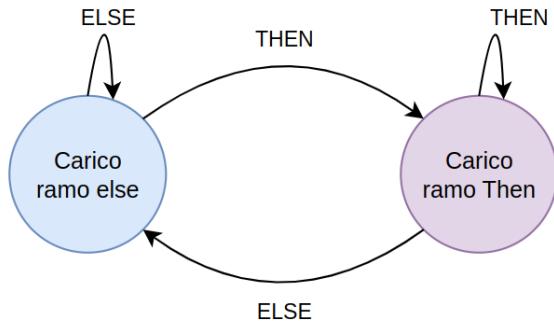


Figura 3.16: Automa della branch prediction base

caso di due fault successivi (fault = errore nel riconoscere il branch giusto), allora cambio il mio branch effettivo. L'automa che meglio spiega tale principio è quello visualizzabile all'immagine [3.17]

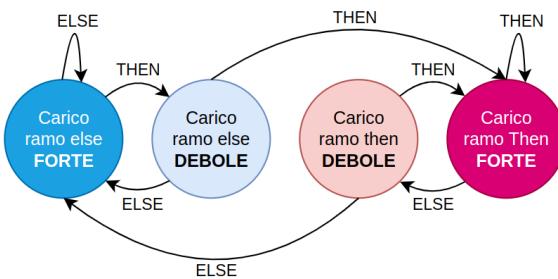


Figura 3.17: Automa della branch prediction avanzato

Per implementare la predizione a livello hardware, il processore utilizza una **tabella di predizione** dei salti. Una possibile struttura è la seguente.

- Indirizzo dell'istruzione di salto.
- Indirizzo di destinazione (dove saltare).
- 2 Bit per codificare lo stato dell'automa.

La tabella viene aggiornata in parallelo durante la fase di Execute (EX), ovvero nel momento in cui il processore verifica se la predizione è corretta. Se c'è stato un errore, si aggiorna lo stato dell'automa (e la corrispondente voce nella tabella). Tale struttura è memorizzata in un apposita area di memoria, detta **Branch History Table** (BHT).

Cerchiamo di capire come sia possibile realizzare una memoria dalle caratteristiche descritte in precedenza. Osserviamo innanzitutto che un tale funzionamento non può essere ottenuto da un classico schema di MUX e DEMUX, in quanto l'ingresso della memoria sarà una chiave (indirizzo 100) e l'uscita sarà un valore (indirizzo 104 o 104). Per cui, la memoria si compone di una serie di chiavi, a ognuna delle quali viene associata un'informazione. È inoltre presente un comparatore legato a ciascuna chiave, che restituisce un valore alto se questa coincide con il valore della chiave fornito in ingresso. Tutte le informazioni memorizzate sono collegate a un MUX, che viene pilotato dalle uscite dei

comparatori. Una tale tipologia di memoria è detta **associativa** [3.18], e differisce dalle classiche memorie indirizzabili quali, RAM o hard disk.

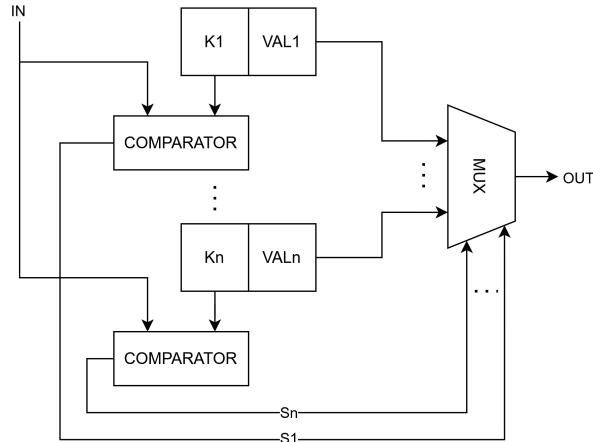


Figura 3.18: Architettura di una memoria associativa.

### 3.4.4 Gestione dei conflitti sui dati

Se due istruzioni consecutive tentano di accedere contemporaneamente ad uno stesso registro/locazione di memoria, come già accennato in precedenza, si generano **conflitti sui dati**, classificabili in *Read after Write* (R/W) e *Write after Read* (W/R). Le operazioni R/R e W/W, invece, non causano problemi: la prima perché la lettura non modifica lo stato dei registri/memoria, la seconda perché conta solo la scrittura operata dalla seconda istruzione. Consideriamo un esempio di conflitto R/W.

```

1 i : R2 = R1 + R3
2 i+1: R4 = R2 + R5

```

La seconda istruzione (i+1) vuole leggere R2 prima che la prima (i) abbia terminato di scriverlo. Questo genererebbe uno stall, perché la pipeline dovrebbe aspettare. Il problema viene risolto utilizzando la tecnica dell'anticipo degli operandi, anche nota come **Operand Forwarding**: il dato appena calcolato dall'ALU (fase EX della i) viene “girato” direttamente alla EX della i+1 senza aspettare che venga scritto in R2 (fase WB). Dunque, la soluzione consiste nel creare un canale hardware che renda i risultati disponibili appena l'ALU li calcola. Notiamo come questa sia realizzabile solo se si opera su registri del processore ed ogni fase impiega un solo ciclo di clock (tipico dei processori RISC).

Cerchiamo di complicare leggermente il conflitto.

```

1 i : R1 = MEMA * carico da memoria in R1.
2 i+1: R2 = R1 + R3 * uso R1 subito dopo.

```

L'accesso alla memoria è molto più lento di quello ai registri, dunque, anche con la tecnica di forwarding vista in precedenza non possiamo prendere un dato dalla memoria prima che sia pronto. Se i sta ancora caricando da memoria (fase MEM), la i+1 che vuole usare il contenuto di R1 non può partire se R1 non è ancora stato aggiornato. Soprattutto se il dato non è in cache, la fase di MEM può impiegare anche decine di cicli.

La linea guida generale seguita dai processori con Pipeline è che la pipe non debba mai

essere interrotta: lo scopo è mantenere un flusso continuo di istruzioni attraverso le varie fasi. Utilizzare la proprietà associativa e commutativa, quando possibile, permette di modificare l'ordine delle istruzioni non bloccando la pipe. La proprietà associativa consente di associare in un solo gruppo le istruzioni indipendenti tra loro; la commutativa, invece, permette di individuare i gruppi di istruzioni invertibili, ovvero la cui inversione non causa errori di esecuzione. Tutto ciò viene implementato in hardware prima della fase di Execute (EX). Se tali proprietà non dovessero essere soddisfatte, la pipe andrebbe in stall.

In tal caso, è possibile adottare un'altra tecnica nota come **Internal Forwarding**, che consiste nell'ibernare temporaneamente le istruzioni bloccate (ad esempio in attesa di dati dalla memoria), sospendendo la loro esecuzione. Nel frattempo, la pipeline può continuare con le istruzioni successive. Quando l'operazione bloccata è pronta per essere completata, viene ripresa nel corretto ordine. In questa soluzione quindi, le istruzioni vengono prelevate in sequenza ma non eseguite necessariamente in sequenza. Notiamo come questo comportamento viola il modello sequenziale di Von Neumann, in cui le istruzioni dovrebbero essere eseguite nell'ordine in cui sono scritte. Un problema importante che può emergere in questo contesto è la condivisione di registri tra istruzioni diverse. Cosa succede se più istruzioni accedono allo stesso registro? Se una lo modifica mentre un'altra lo sta ancora usando, si crea un conflitto sui dati, chiamato anche **Hazard**. Una soluzione possibile è quella di creare più copie del registro coinvolto, in modo da tenere traccia dei diversi stati dello stesso dato nel tempo. Questo può essere implementato tramite una forma di indirizzamento indiretto: ogni registro ha uno o più puntatori che indicano sezioni di memoria dedicate a contenere le vecchie versioni del registro stesso [3.19]. Per funzionare correttamente, è necessario che il numero di registri fisici (cioè le copie disponibili) sia maggiore rispetto ai registri logici visibili al programmatore, in modo da evitare conflitti durante l'esecuzione in parallelo delle istruzioni. Per implementare

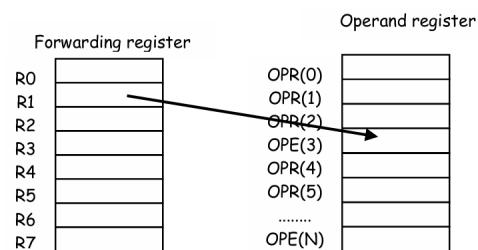


Figura 3.19: Struttura dei registri copia con indirizzamento indiretto.

questa tecnica si utilizza un'area di memoria interna al processore detta **tabella di ibernazione**, che tiene nota di tutte quelle istruzioni già decodificate, ovvero di cui si ha già la chiara conoscenza degli operandi e delle operazioni che queste richiedono [3.20]. Se per un determinato tipo, è necessario ibernare più istruzioni di quanto la tabella permette, il blocco della pipe diventa inevitabile. Notiamo anche come sia possibile ibernare soltanto un'operazione di LOAD e STORE alla volta, questo perché finché un'operazione in memoria non è conclusa non se ne possono attivare altre dello stesso tipo. Gli altri registri della tabella potrebbero essere aumentati per una maggiore efficienza, ma di conseguenza crescerebbe anche il costo implementativo della soluzione. I registri del processore, che in questo contesto prendono il nome di **Forwarding Register**, non contengono i valori effettivi, ma dei puntatori ai registri che contengono i veri valori, detti **Operand Regi-**

	operando1		operando2		
	TAG	VALORE	TAG	VALORE	RISULTATO
ADD					
ADD					
MUL					
MUL					
DIV					
BOO					
LOAD					
STORE					

Figura 3.20: Tabella di ibernazione.

ster e situati in una memoria esterna. Ad esempio, se il registro R1 contiene il valore 3, ciò significa che il suo valore effettivo è memorizzato nel registro OPR(3). I registri operando contengono, oltre al valore attuale dei registri del processore, anche i valori che essi hanno assunto nel corso dell’elaborazione, ovvero la loro ‘storia passata’ [3.21]. Inoltre, vengono riportate due informazioni aggiuntive:

1. Il numero di operazioni sospese che dipendono da esso.
2. Un bit di validità, per indicare se il valore è valido (0) o meno (1).

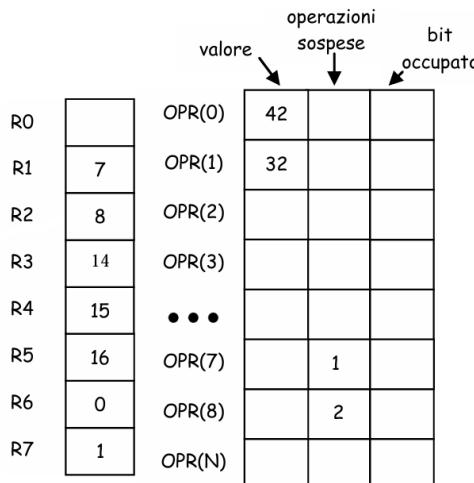


Figura 3.21: Struttura dei registri.

Non ci resta che cercare di capire come una tale struttura possa essere utilizzata per evitare lo stalllo della pipe in caso di conflitti sui dati. Per farlo, prendiamo come esempio il seguente codice.

```

1 i:    R1 = MEMA
2 i+1:  R2 = R1 + R3
3 i+2:  R4 = R2 + R5
4 i+3:  R2 = R6 + R7
5 i+4:  R4 = R2 + R4

```

Per prima cosa, l’istruzione *i* tenta di caricare un dato dalla memoria al registro R1. Supponendo che il dato non sia presente in cache (cache miss), questo non è subito disponibile e l’istruzione non può essere completata. Di conseguenza, viene ibernata: viene memorizzato che il valore destinato a R1 sarà disponibile in futuro e si aggiorna il registro OPR(1), che funge da puntatore al valore effettivo, indicando dove il dato arriverà.

Subito dopo, l'istruzione  $i+1$  ha bisogno del valore contenuto in R1, che però ancora non è stato caricato. Anche questa istruzione viene quindi ibernata e si aggiorna il contatore di operazioni sospese su R2. L'istruzione  $i+2$ , a sua volta, dipende dal valore di R2, che è stato aggiornato da una istruzione ancora in attesa. Per questo motivo anche  $i+2$  viene ibernata, e si aggiorna nuovamente il numero di operazioni sospese su R2, che diventa 2. Arrivati a  $i+3$ , questa istruzione può essere eseguita subito perché i registri R6 e R7, da cui dipende, non sono coinvolti in operazioni ibernate. Viene così calcolato un nuovo valore per R2 e si aggiorna il puntatore OPR(2) con un nuovo riferimento. Nel frattempo, anche se si aggiorna R2, le informazioni precedenti relative alle istruzioni ancora ibernate non vengono perse, perché il sistema gestisce separatamente i valori e i riferimenti alle istruzioni in sospeso. Quando il dato atteso da MEMA per l'istruzione  $i$  finalmente arriva, questa può essere completata. Di conseguenza si sbloccano le istruzioni  $i+1$  e  $i+2$  che dipendevano da quel dato, e possono essere eseguite una dopo l'altra. A ogni esecuzione, si aggiornano i puntatori e i contatori associati: quando il numero di operazioni sospese su un registro scende a zero, il valore è considerato valido e il bit di “occupato” si azzera, rendendo il registro di nuovo disponibile per istruzioni future.

Le due fasi della pipeline cruciali per gestire le tabelle di ibernazione e i registri sono ID e EX. Durante la fase di decodifica il processore capisce cosa deve fare l'istruzione, se non può essere eseguita viene ibernata e aggiunta alla tabella di ibernazione. È in questa fase che dunque si aggiornano i registri del processore, cambiando i loro puntatori. Durante la fase di esecuzione le istruzioni vengono effettivamente eseguite, i valori reali negli Operand Register (OPR) vengono calcolati, scritti, o rimossi se non più necessari. Per cui, è durante questa fase che si aggiornano i registri operandi. Un'altra cosa interessante da notare è il comportamento del sistema una volta che i dati in attesa dalla memoria diventano disponibili. In tal caso, è possibile iniziare a sbloccare le istruzioni precedentemente congelate, rendendole pronte ad essere eseguite. Questo però, genera uno stallo temporaneo della pipe. Fortunatamente, il blocco riguarda solo le prime fasi della pipeline (tipicamente IF e ID) e non le fasi critiche come l'esecuzione (EX) o la scrittura (WB). Questo è importante, poiché così non vengono alterati i registri e lo stato interno del processore.

### 3.4.5 Gestione delle interruzioni

Con l'introduzione del sistema a pipeline e del meccanismo dell'internal forwarding, che altera la sequenzialità delle istruzioni eseguite dal processore, il problema della **gestione delle interruzioni** diventa sempre più complesso. Infatti, può accadere che le istruzioni si completino in ordine diverso da quello di esecuzione. Se un'istruzione causa un'eccezione prima che le precedenti siano terminate, si parla di **interruzione non precisa**. Per evitarlo, si mira a una **gestione precisa**, dove il sistema blocca la pipeline, completa le istruzioni precedenti e impedisce l'avvio di quelle successive, mantenendo il comportamento sequenziale previsto dal modello di Von Neumann.

Ricordiamo che le interruzioni sono dovute a diversi fenomeni, ma che in generale si classificano in interne ed esterne. Le **interruzioni esterne** sono le più facili da gestire. Supponiamo di avere una pipe nella quale, in un certo momento, sono state attivate un certo numero di istruzioni. In base al modello classico di Von Neumann, occorre terminare l'operazione in corso nella pipe (e le eventuali operazioni ibernate) e salvare lo

stato del processore prima di servire una interruzione. Seguendo tale modello, quindi, quando giunge un'interruzione da parte di una periferica di I/O, la ISR sarà inserita nella pipe come una qualsiasi altra istruzione. Se il sistema delle interruzioni è vettorizzato, è la stessa periferica a identificarsi mediante un numero, che il processore somma ad un indirizzo base, per calcolare l'indirizzo iniziale della ISR. Nel caso non vettorizzato, occorre interrogare i registri di stato di tutte le periferiche per scoprire quale ha causato l'interruzione (polling). Tale concetto si estende al caso Pipelined, salvo che le operazioni in corso sono più di una.

Più critico è il caso delle **interruzioni interne**. Poiché si tratta di interruzioni che hanno effetto all'interno della pipe e non all'esterno, tali eventi possono dare luogo ad un comportamento del programma diverso da quello di Von Neumann. Ad esempio, potrebbe accadere che in una sequenza  $(i, i+1)$ , l'istruzione  $i+1$  generi un'eccezione prima che l'istruzione  $i$  abbia il tempo di essere eseguita. Per garantire che il comportamento del sistema, in queste condizioni, sia quello di Von Neumann, occorre complicare in maniera notevole l'hardware. Esempi di soluzioni che possono essere adottate sono:

- **Rinuncia alle interruzioni precise:** Si accetta l'esecuzione non sequenziale, dando priorità all'interruzione che arriva per prima. Così facendo, si demanda al software la decisione sul capire se ci sono eventuali altre istruzioni in pipe che possono interrompere. È poco efficiente, ma accettabile dato che le eccezioni sono rare.
- **Ricostruzione delle interruzioni precise:** Le istruzioni possono essere *out-of-order*, ma si impiegano due possibili approcci:
  1. Nell'approccio *conservativo*, ogni istruzione prosegue solo se è sicuro che non causerà eccezioni. Questo riduce il parallelismo, ma garantisce precisione.
  2. In quello *ottimistico*, il processore prosegue comunque e, in caso di errore, effettua un **roll-back** (ripristino dello stato precedente). Questo richiede però di salvare lo stato del sistema.
- 1. La tecnica del **Check Point** consiste nel salvare periodicamente lo stato del processore (inclusi registri e Program Counter) in una memoria dedicata. In caso di eccezione o interruzione, si può ripristinare l'ultimo stato salvato e riprendere l'esecuzione in modo sequenziale da lì. Un aspetto critico è la gestione della memoria, poiché alcune istruzioni eseguite dopo il checkpoint potrebbero aver alterato dati che non avrebbero dovuto essere modificati. Questo richiede eventualmente meccanismi aggiuntivi per annullare anche le scritture in memoria. Inoltre, è importante scegliere con attenzione quando effettuare i salvataggi: se troppo frequenti, possono introdurre un sovraccarico in termini di tempo e spazio; se troppo distanti, in caso di eccezione si deve ripetere un ampio numero di istruzioni.
- 2. La tecnica dell'**History Buffer** prevede l'uso di un buffer hardware che conserva, per ogni istruzione eseguita, i valori originali dei registri o delle locazioni di memoria che essa modifica. Questo permette di annullare gli effetti di istruzioni che non dovrebbero essere state eseguite, tramite un'operazione di *undo*. Le informazioni salvate nel buffer restano disponibili fino a quando non è certo che tutte le istruzioni precedenti sono state eseguite correttamente (cioè non hanno generato eccezioni). In caso di eccezione, il sistema può ripristinare lo stato precedente annullando, in ordine inverso, tutte le modifiche fatte dalle istruzioni speculative o

successive a quella che ha causato l'errore. Il buffer deve essere sufficientemente grande da contenere le modifiche delle istruzioni speculative che si trovano in esecuzione simultanea nella pipeline. Questa tecnica ha il vantaggio di non introdurre overhead in assenza di errori, poiché la memorizzazione dello stato precedente avviene in parallelo all'esecuzione normale del processore. È per questo motivo che la tecnica dell'History Buffer viene implementata completamente in hardware.

Notiamo come in entrambe le tecniche, a seguito di eventuali eccezioni, l'esecuzione si riprende *sequenzialmente* dal punto di interruzione. Con ciò si intende che l'esecuzione deve proseguire in ordine corretto delle istruzioni del programma, come se l'errore non fosse mai avvenuto.

In un approccio basato sul roll-back è fondamentale essere conservativi sulla scrittura in memoria, ovvero bisogna fare attenzione a non scrivere (fase WB) troppo presto. Se lo si fa e poi si scopre che un'istruzione successiva ha generato un'eccezione, la scrittura sarebbe già avvenuta e non potremmo più tornare indietro in modo pulito. In altre parole, dopo aver ripristinato lo stato corretto (tramite checkpoint o undo), non si può ripartire da uno stato speculativo o da metà pipeline, ma bisogna ricominciare l'esecuzione delle istruzioni in ordine, partendo dal Program Counter corretto.

## 3.5 Architetture Superscalari

L'utilizzo delle pipe porta dei vantaggi non in termini di attraversamento di una istruzione, che rimane invariato, ma in termini di produttività. Per ottenere prestazioni ancora migliori, è possibile realizzare un'architettura con più pipe che eseguono diverse istruzioni in parallelo, così da aumentare la produttività del sistema. Una tale architettura viene chiamata **Superscalare**. Tuttavia, questa introduce delle problematiche che vanno necessariamente risolte, queste sono:

1. Le pipe devono condividere un unico accesso in memoria comune per il prelievo delle istruzioni. In altre parole, anche in presenza di più pipe tutte devono leggere dalla stessa memoria. Questo può creare un collo di bottiglia, perché solo una pipeline alla volta può leggere da lì.
2. Se le pipe non sono del tutto indipendenti ma condividono delle stazioni, nascono dei problemi di conflitto nell'utilizzo delle unità funzionali condivise. Se due pipeline vogliono usare la stessa unità (ad esempio, la ALU) allo stesso momento, nasce un collisione e una delle due deve aspettare. Questo riduce l'efficienza.

### 3.5.1 Gestione delle Collisioni

La **gestione delle collisioni** nella architetture superscalari avviene completamente in hardware. Per comprendere al meglio come un processore con una tale architettura gestisca il problema, introduciamo un esempio. Supponiamo che la CPU sia in grado di realizzare addizione e moltiplicazione in floating point. Come possiamo immaginare, alcune delle operazioni presenti nell'addizione potrebbero essere richieste anche dalla moltiplicazione (e viceversa), mappiamo dunque all'interno di una tabella le diverse operazioni necessarie a completare le due [3.1 e 3.2].

L'introduzione di tali tabelle semplifica notevolmente la scrittura del **vettore delle collisioni**, ovvero un particolare vettore binario utilizzato dal compilatore per gestire il

	1	2	3	4	5	6	7
Ex Add	X						
Mult		X	X				
Man Add			X	X			
Renorm					X		X
Round						X	
Shift A							
Lead 1							
Shift B							

Tabella 3.1: 7 stadi richiesti dalla moltiplicazione.

	1	2	3	4	5	6	7	8	9
Ex Add	X								
Mult									
Man Add				X					
Renorm									X
Round							X		
Shift A		X	X						
Lead 1					X				
Shift B						X	X		

Tabella 3.2: 9 stadi richiesti dall'addizione.

problema dei conflitti tra pipe. Poniamoci nell'ipotesi semplificativa di due sole pipe, in modo da visualizzare più facilmente il processo. Per ricavare il vettore, è sufficiente sovrapporre le tabelle delle operazioni che vogliamo eseguire, opportunamente shiftate a seconda di quale delle due sia eseguita prima. Ad esempio, se vogliamo eseguire una moltiplicazione X e al ciclo di clock successivo un'altra moltiplicazione Y, la sovrapposizione delle tabelle fornisce il seguente risultato [3.3].

	1	2	3	4	5	6	7
Ex Add	X	Y					
Mult		X	XY	Y			
Man Add			X	XY	Y		
Renorm					X	Y	X
Round						X	Y
Shift A							
Lead 1							
Shift B							

Tabella 3.3: Collisioni tra due moltiplicazioni sfasate di 1 ciclo di clock.

La tabella evidenzia come le due operazioni, eseguite secondo la tempificazione descritta, presentano delle collisioni al tempo 3 e 4 (presenza contemporanea di X e Y). Ripetendo la sovrapposizione appena mostrata per diverse tempificazioni (dopo 2 cicli, dopo 3 cicli, eccetera), è possibile giungere al corrispondente vettore delle collisioni: 110000. È fondamentale ribadire che il vettore si limita solamente a segnalare la presenza di una collisione

(1 nella stringa) nel caso in cui le operazioni venissero eseguite con quella tempificazione, ma ciò non ha niente a che vedere con le fasi della pipeline. In altre parole, se nella stringa c'è un 1 nella seconda posizione, questo NON significa che la collisione riguarda la fase di decodifica.

È da notare come in tal caso non valga la proprietà commutativa, realizzando infatti prima una moltiplicazione e poi un'addizione non ci sarebbe alcuna collisione. Nel nostro esempio ci siamo fermati a realizzare il collision vector per 1 solo caso, ovvero quello in cui una moltiplicazione segue una moltiplicazione. In generale, è necessario costruirlo per tutte le possibili combinazioni di operazioni (*mul* segue *add*, *add* segue *add*, eccetera). Se per ipotesi avessimo  $n$  istruzioni da realizzare, sarebbe necessario costruire  $2^n$  vettori, complicando notevolmente il lavoro del compilatore.

Ritorniamo all'esempio e cerchiamo di capire come il compilatore gestisca le collisioni per una generica sequenza di operazioni, che supponiamo essere: *add*, *mul*, *mul*. È innanzitutto necessario costruire il vettore di collisione per le operazioni *add-mul* e *mul-mul*. A questo punto, bisogna inserire la seconda moltiplicazione tenendo conto sia della moltiplicazione precedente che della prima addizione. In generale potrebbero presentarsi diverse combinazioni delle operazioni da eseguire, per cui la verifica dei conflitti tra pipe avviene tramite un **sistema di decisione** completamente realizzato in hardware, con una struttura fatta nel seguente modo [3.22]. In particolare, il registro a scorrimento fa

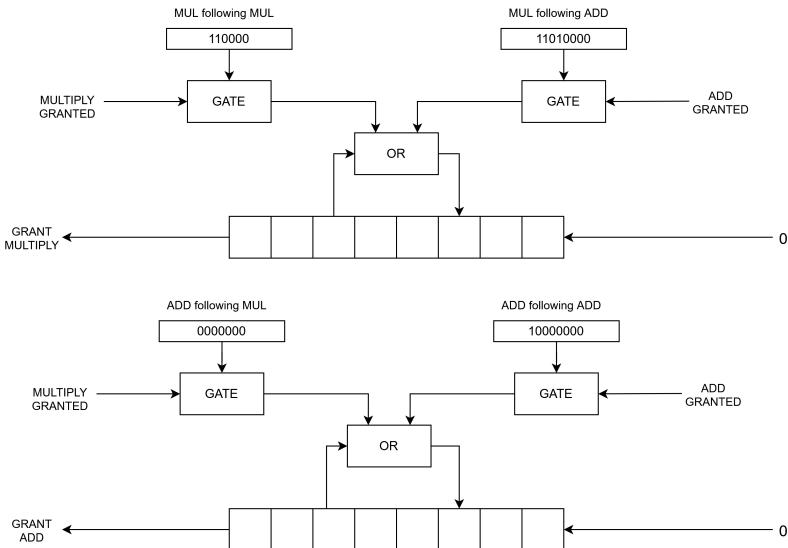


Figura 3.22: Hardware per la gestione delle collisioni.

proseguire la "storia" delle istruzioni, mentre la OR "unisce" la storia precedente con la nuova. Quindi, la OR garantisce che non ci sia collisione né con l'istruzione che stiamo aggiungendo, né con la storia di istruzioni precedenti. I segnali di *multiply granted* e *add granted*, indicano rispettivamente la possibilità di eseguire la moltiplicazione e l'addizione. I segnali *grant multiply* e *grant add*, invece, abilitano le rispettive operazioni quando assumono valore 0. Il motivo per cui tali segnali sono attivi sullo 0 è semplice, la OR mette in relazione la storia passata (presente nel registro a scorrimento) e l'operazione che vogliamo eseguire (descritta dal vettore delle collisioni), se la OR bit a bit tra i due non genera conflitti, allora il bit che esce dallo shift register è uno 0. È fondamentale notare come la struttura sia suddivisa in due parti, quella superiore che riguarda soltanto

la moltiplicazione, e quella inferiore che invece riguarda l'addizione.

C'è da osservare che i problemi relativi a una tale soluzione sopraggiungono in corrispondenza di istruzioni di salto. In tal caso, a seconda o meno della verifica della condizione di salto, bisogna arrestare non solo una pipe, ma molteplici e gestirne di conseguenza il riavvio. Considerando che i salti costituiscono circa il 25% delle istruzioni, una semplice soluzione consiste nel dedicare una pipe specifica alla predizione dei salti. Questa predizione andrebbe elaborata il prima possibile, in modo da ridurre al minimo gli inconvenienti all'interno delle restanti pipe ed aumentare quindi l'efficienza complessiva del sistema.

Un tale tipo di soluzione hardware per la gestione delle collisioni è tipica dei sistemi **DSP** (Digital Signal Processor), come quelli usati per audio, video, telecomunicazioni. In particolare, il suo utilizzo permette di schedulare le istruzioni nel modo più efficiente possibile evitando stalli o rallentamenti.



# Capitolo 4

## Memoria

La realizzazione di memorie centrali con elevate prestazioni in termini di tempo di accesso e capacità di archiviazione richiede l'integrazione di moduli di memoria aventi caratteristiche diverse, integrando memorie veloci di ridotte dimensioni con memorie di maggiore dimensione ma di minore velocità. Tale soluzione architetturale prende in letteratura il nome di **gerarchia delle memorie**.

### 4.1 Memoria cache

In un'architettura gerarchica su due livelli, come quella mostrata in figura [4.1], la memoria M1 deve necessariamente essere veloce ed è quindi di tipo statico, mentre la memoria M2, di dimensioni maggiori e destinata tipicamente a contenere tutti i dati e tutte le istruzioni, è di tipo dinamico. Senza scendere troppo in dettagli tecnici, le memorie statiche sono generalmente caratterizzate da un elevato numero di transistor e dimensioni molto ridotte, mentre le memorie dinamiche sono basate su effetti capacitivi e tendono a perdere la carica, dunque devono ricevere processi di refresh che consistono nel riscrivere nuovamente il dato. La prima tipologia di memoria prende convenzionalmente il nome di **memoria cache**.

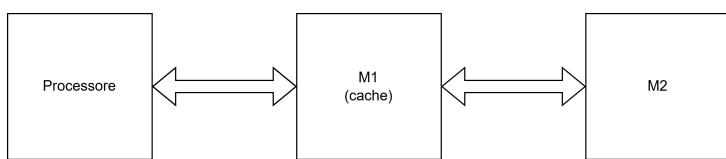


Figura 4.1: Gerarchia delle memorie su due livelli.

Come ben sappiamo, la memoria centrale lavora con il concetto di indirizzo: viene fornito un valore al decoder e questo fa accedere a una corrispondente locazione di memoria. Questo principio di funzionamento vuole implicitamente afferire che ogni indirizzo è valido, ossia esiste sempre qualche informazione (anche se nulla) contenuta in quest'ultimo. Tale comportamento differisce da quello delle memorie associative, nelle quali una volta fornita la chiave, l'oggetto associato potrebbe non esistere. Le memorie cache funzionano proprio con questa filosofia, sono infatti organizzate in blocchi di dati contigui indicizzati da un'opportuna chiave, detta anche tag. Il motivo della contiguità dei dati nei blocchi va ricercato nel **Principio di Località** (spaziale), il quale stabilisce che se si accede a un indirizzo di memoria, è probabile che presto saranno necessari dati contenuti in indirizzi vicini. Quando il processore fornisce l'indirizzo dell'informazione a cui accedere questo

viene scomposto in due parti: la prima identifica la chiave, mentre la seconda specifica la posizione del dato da prelevare all'interno nel blocco. Se la porzione di chiave individuata non combacia con nessuna delle chiavi presenti, e quindi il corrispondente blocco non è in cache, si verifica un cache miss. Questa prima tipologia di architettura per le memorie cache presentate prende il nome di **Full Associative** [4.2], ed è una soluzione leggermente più semplice di quella **Set Associative** presentata di seguito. È molto importante notare che il processore per accedere a una parola in memoria ne fornisce sempre l'indirizzo, indipendentemente dal fatto che questa si trovi in cache oppure no, in quanto il processore è del tutto trasparente rispetto all'organizzazione delle memorie.

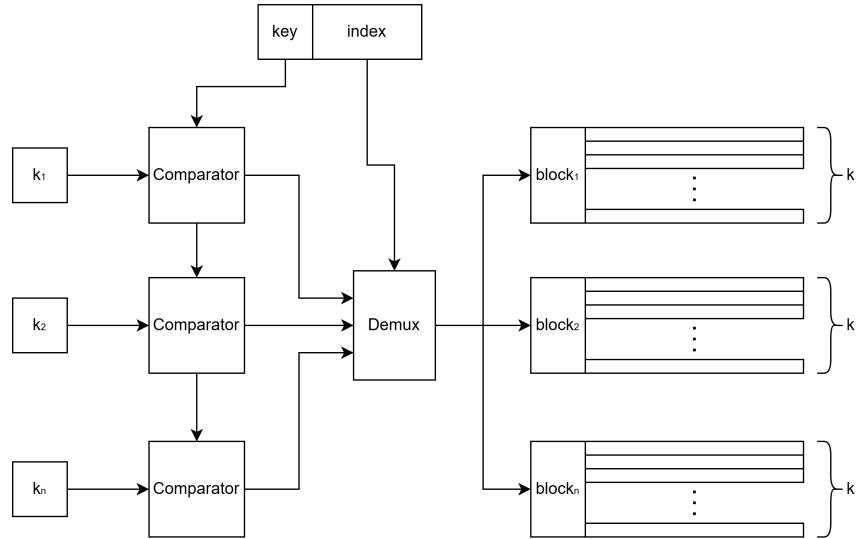


Figura 4.2: Architettura di una memoria cache Full Associative.

Cerchiamo di capire il motivo per cui una tale suddivisione degli indirizzi si rende necessaria. Come anticipato, le memorie cache sono caratterizzate da dimensioni molto ridotte, per cui se l'intero indirizzo venisse utilizzato per indicizzare un dato, si avrebbe uno spreco eccessivo (inefficienza). Ad esempio, se utilizzassimo indirizzi a 32 bit per indicizzare dati su 32 bit, il 50 dello spazio verrebbe sprecato per l'indirizzo.

#### 4.1.1 Politiche di Sostituzione

Focalizziamo ora la nostra attenzione su cosa accade in caso di cache miss. In tale evenienza, la memoria cache deve interagire con la RAM per recuperare il dato mancante e sostituire uno dei suoi blocchi con uno contenente il dato appena prelevato. Questo processo può essere molto problematico nel caso in cui il programma esegue molti salti, a causa della continua sostituzione dei blocchi, ed è per questo motivo che un programma andrebbe scritto tenendo sempre presente il principio di località. Per quanto detto, la cache si comporta come un vero e proprio processore, in quanto interagisce con la RAM in caso di miss, ma come avviene la scelta della **politica di sostituzione**? Ovvero, come viene scelto il blocco da sostituire? Una prima semplice scelta potrebbe essere selezionare casualmente il blocco, una scelta leggermente più intelligente prevede di aggiungere un meccanismo che tenga conto di quanto ciascun blocco viene richiesto. Per poter implementare la seconda modalità, ciascun blocco memorizza, oltre ai dati, un bit di flag per la validità e due bit di conteggio. Ogni volta che un blocco viene utilizzato, i bit di conteggio sono azzerati, mentre ad ogni accesso in cache per cui il blocco non è richiesto,

il valore di conteggio si incrementa. Quando è necessario sostituire un blocco, si sceglie quello con valore di conteggio più elevato (quello più vecchio) e in caso di pareggi si può utilizzare la scelta randomica. Il bit di validità è fondamentale all'avvio del sistema, in cui i dati presenti nei blocchi sono spazzatura e saranno quindi necessari diversi cache miss per poter portare la cache in uno stato "corretto". Ulteriori politiche di sostituzione prevedono l'impiego di algoritmi di tipo Round Robin e FIFO.

#### 4.1.2 Gestione dell'allineamento

I dati presenti in cache potrebbero non corrispondere, e quindi non essere allineati, con quelli presenti nella RAM. Notiamo come il problema non si verifichi quando sono effettuate letture, ma esclusivamente in presenza di operazioni di scrittura. Sono evidentemente necessarie delle tecniche di **gestione dell'allineamento**. Le due principali soluzioni adottate sono:

- **Write Through:** Ogni volta che viene scritta una parola in cache, in parallelo si effettua anche la scrittura in RAM. Dunque, le memorie risulteranno sempre allineate.
- **Write Back:** L'allineamento con la versione della parola in RAM avviene soltanto se il blocco in cui questa è contenuta è stato modificato, e la scrittura avviene prima che il blocco venga sostituito. Per realizzare questa tecnica è necessario aggiungere un ulteriore bit di modifica, oltre a quelli necessari per la politica di sostituzione. Quando il bit è alto, oltre a sostituire il blocco è necessario anche scrivere in RAM la versione aggiornata, per cui le due memoria non sono sempre allineate.

Notiamo come l'aggiunta di un DMA potrebbe complicare il progetto del sistema a seconda della soluzione di allineamento adottata. Nel caso del WT, il DMA può attivarsi immediatamente in quanto i dati sono sempre allineati, ma il bus viene utilizzato moltissimo, poiché ogni scrittura in cache comporta una corrispondente scrittura in RAM. Nel caso di WB, invece, prima di avviare il DMA è necessario forzare l'allineamento dei dati.

Come possiamo immaginare, più la cache è vicina al processore e minore saranno i tempi di propagazione dei chip (maggiore velocità). Tuttavia, questo ha una limitazione in termini di dimensione della cache, in quanto più vicina questa sarà alla CPU, minore dovrà essere lo spazio occupato. Per questo motivo, nei sistemi moderni si utilizzano cache di diversi livelli e che lavorano nella maggior parte dei casi in Write Back. Inoltre, è molto probabile che le cache a bordo (vicine al processore) si trovino su processori di tipo RISC.

#### 4.1.3 Set Associative

Una delle scelte più difficili riguardo l'architettura delle memorie cache riguarda la dimensione dei blocchi. Ad esempio, se il programma in esecuzione è molto locale, allora avere pochi blocchi molto grandi è sicuramente conveniente, mentre se il programma è poco locale la cosa migliore è avere molti blocchi piccoli. Osserviamo anche che maggiore è la dimensione dei blocchi, maggiore è la quantità di dati da trasferire in caso di cache miss, dunque idealmente vorremmo tanti blocchi piccoli per velocizzare gli spostamenti. Tuttavia, avere un numero elevato di blocchi porta fondamentalmente a due complicazioni:

- Il numero di comparatori necessari per la verifica della chiave cresce.
- Avere un elevato numero di chiavi porta al problema di inefficienza introdotto prima (spreco troppo spazio per memorizzare le chiavi).

È fondamentale trovare dunque un giusto compromesso, cosa possibile complicando leggermente l'architettura delle memorie cache vista in precedenza e introducendone una di tipo **Set Associative**. Tale architettura può essere immaginata come una via di mezzo tra cache diretta (semplice ma rigida) e completamente associativa (flessibile ma costosa). Questa volta, la cache viene suddivisa in più set, ognuno dei quali contiene molteplici blocchi. Un dato può essere memorizzato in qualsiasi blocco del set, ma la ricerca avviene soltanto nel set giusto e non in tutta la memoria come avviene in quelle completamente associative. Per poter realizzare questo funzionamento, l'indirizzo fornito dal processore deve ora essere scomposto in tre parti: la prima identifica la chiave del blocco, la seconda il set in cui cercare e la terza l'offset rispetto al blocco [4.3]. In una tale architettura si definisce *grado di associatività* il numero di blocchi per set, una cache 4-way usa 4 blocchi per ciascun set.

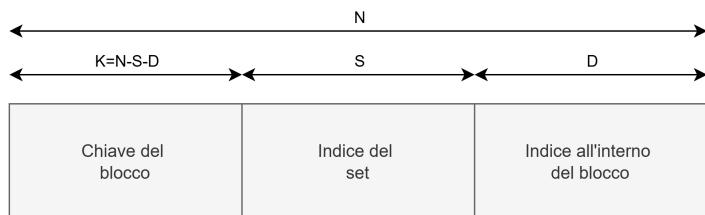


Figura 4.3: Struttura dell'indirizzo utilizzato per memorie cache set associative.

La ricerca di un dato in cache avviene in due fasi:

- Selezione del set**: Utilizzando i bit dell'indirizzo destinati all'indice del set, un decoder seleziona l'insieme (set) corretto di blocchi della cache. Solo i blocchi di questo set saranno coinvolti nella ricerca.
- Confronto delle chiavi**: All'interno del set selezionato, i tag (le chiavi) di tutti i blocchi vengono confrontati in parallelo con il tag calcolato dall'indirizzo. Questo confronto determina se il blocco cercato è presente (cache hit) o meno (cache miss).

Per ragioni di efficienza, questa architettura prevede che tutti i confronti sulle chiavi del set vengano effettuati in parallelo e che, mediante i D bit, i dati di tutti i blocchi contenuti nel set individuato vengano comunque selezionati, ma al processore sarà trasmesso soltanto il dato appartenente al blocco per il quale si è verificata la corrispondenza tra le chiavi. In questo modo, la selezione del dato nel blocco e la verifica della corrispondenza tra chiavi può essere fatta in parallelo. Ovviamente, nel caso in cui non si riscontri alcuna corrispondenza tra le chiavi, si avrà un miss e il dato dovrà essere prelevato dalla RAM (o dalla memoria superiore nel sistema gerarchico).

All'inizio del capitolo è stato affermato che un'architettura Set Associative consente un risparmio in termini di numero di comparatori e di lunghezza delle chiavi, rispetto a un'organizzazione completamente associativa. Vediamo ora in che modo ciò avviene. Consideriamo, a parità di lunghezza dell'indirizzo N, una cache completamente associativa e una set-associativa.

1. Nel caso completamente associativo, non esistendo un campo indice per selezionare un set, la lunghezza del campo chiave sarà pari a  $K=N-D$ , dove  $D$  rappresenta il numero di bit usati per l'offset all'interno del blocco.
2. Nel caso set-associativo, parte dell'indirizzo viene utilizzata per selezionare il set: supponendo di avere  $2^s$  set, la lunghezza del campo chiave si riduce a  $K=N-S-D$ .

Ne consegue che l'organizzazione Set Associative richiede meno bit per ogni tag, e, poiché la ricerca viene effettuata solo sui blocchi appartenenti al set selezionato, sono necessari meno comparatori rispetto all'organizzazione Full Associative, dove il confronto deve avvenire con tutti i blocchi della cache.

Consideriamo un esempio numerico su come si calcolano i bit necessari per ciascun campo di un indirizzo a 32 bit in una cache set-associativa 4-way. Innanzitutto definiamo una dimensione della cache, un valore comune è 16 KB = 16384 B con blocchi da 64 B, a partire dal quale:

$$\begin{aligned} numBlocchi &= \frac{dimCache}{dimBlocco} = \frac{16384}{64} = 256 \\ numSet &= \frac{numBlocchi}{gradAssociativita} = \frac{256}{4} = 64 \end{aligned}$$

Per ottenere il numero di bit necessari a codificare l'offset del blocco basta fare il logaritmo in base 2 della dimensione di un blocco ( $\log_2(64) = 6$ ), mentre per il set basta farlo per il numero di set trovato. Il campo chiave si ottiene come differenza tra i 32 bit disponibili e la somma degli altri due campi.

*Nota importante:* Il professore potrebbe chiedere di calcolare la dimensione di ciascun campo stabilendo delle dimensioni iniziali diverse, ad esempio assegnando un grado di associatività e un fissato numero di blocchi. In tal caso, il procedimento di calcolo potrebbe variare leggermente.

Oltre al full e set associative, citiamo anche il *mapping diretto*, il quale richiede che un blocco di dati presenti in memoria principali si trovi alla stessa posizione nella cache. La traduzione si realizza attraverso una funzione modulo.

#### 4.1.4 Dimensionamento

Dimensionare un sistema significa determinare le sue caratteristiche fisiche sulla base dell'architettura studiata. Consideriamo, in particolare, il **dimensionamento di una memoria cache**. In questo contesto, è necessario definire tre parametri fondamentali: il numero di blocchi ( $k$ ), il numero di settori ( $s$ ) e la lunghezza di ciascun blocco ( $L$ ). La dimensione complessiva della cache si ottiene come prodotto di questi fattori,  $dim = ksL$ . È utile osservare che il dimensionamento della memoria virtuale, tema di maggiore interesse nei nostri studi, può essere interpretato come un caso particolare del dimensionamento della cache, data la somiglianza strutturale tra le due tipologie di memoria.

L'obiettivo del dimensionamento è minimizzare il numero di cache miss. Tuttavia, non esiste una configurazione ottimale valida in assoluto, poiché le prestazioni della cache dipendono fortemente da fattori quali la località del programma e i dati di input utilizzati. Per affrontare questa variabilità, i tool di dimensionamento automatico (come SPIM)

testano il sistema con diverse tipologie di applicazioni, per valutare il comportamento della cache in una vasta gamma di situazioni. In generale, il rapporto tra la dimensione della cache e il numero di cache miss può essere rappresentato graficamente, mostrando come l'aumento della dimensione influenzi (non sempre linearmente) il tasso di miss [4.4].

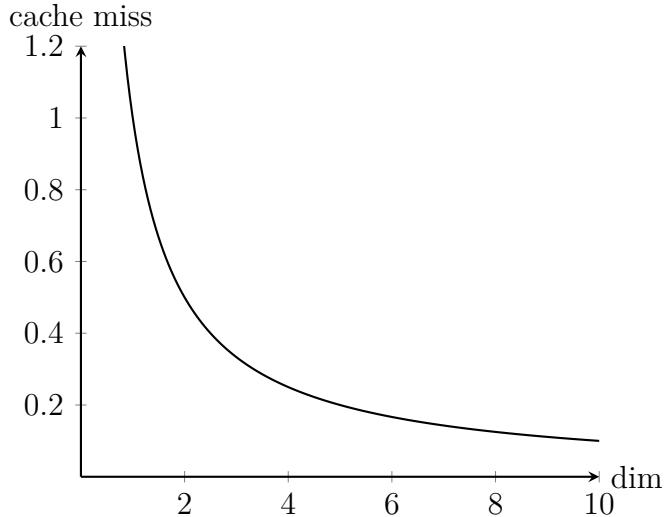


Figura 4.4: Grafico di dimensionamento della memoria cache.

Per dimensionare correttamente la cache, è importante individuare un punto nel grafico che consenta di mantenere costante il numero di cache miss, evitando però una dimensione eccessiva della memoria. In pratica, i tool di dimensionamento ripetono questa analisi per ciascuna delle applicazioni in ingresso, valutando una configurazione che rappresenti una media ottimizzata. Naturalmente, il dimensionamento non può prescindere dai vincoli di realizzabilità fisica, trasformando il problema in un problema di ottimo vincolato.

Il problema, come definito, si presenta con una sola equazione e tre incognite, il che implica un numero infinito di soluzioni possibili. Tuttavia, molti di questi casi possono essere esclusi in base a vincoli fisici o pratici. Possiamo ulteriormente ridurre il numero di combinazioni imponendo vincoli sui singoli parametri. Ad esempio:

1. Il numero di blocchi  $k$  può assumere valori discreti e non troppo grandi, come 4, 8 o 16, a causa dei problemi di gestione di un numero eccessivo di blocchi.
2. La dimensione dei blocchi  $L$  non può essere arbitrariamente grande, spesso viene scelta proporzionalmente al numero di blocchi, ad esempio  $L = pk$ .
3. Una volta fissati  $k$  e  $L$ , il numero di set  $s$  risulta vincolato dagli altri due parametri.

Nonostante questi vincoli, il numero di combinazioni possibili rimane elevato, rendendo impraticabile una ricerca esaustiva. Per questo motivo, si ricorre spesso a euristiche, basate su osservazioni statistiche del comportamento tipico della cache: si cerca di stimare la probabilità di cache miss dato un certo applicativo e una configurazione specifica della cache. Infine, notiamo che è possibile ridurre ulteriormente il numero di simulazioni necessarie. Se, per esempio, è già disponibile una simulazione per  $k = 4$ , possiamo inferire anche quella per  $k = 2$ . In effetti, ciò presuppone un'ipotesi forte: che il comportamento della cache non cambi in modo significativo al variare della sua dimensione. Questa semplificazione è spesso accettata per rendere il problema trattabile.

## 4.2 Memoria Virtuale

La **memoria virtuale** è una tecnica che consente a un sistema operativo di far sembrare che ogni processo abbia a disposizione un ampio spazio di memoria continuo e privato, anche se la memoria fisica (RAM) è limitata. In pratica, La memoria virtuale separa lo spazio di memoria logico (visto dal programma) da quello fisico. Inoltre, quando non c'è abbastanza memoria disponibile, parte dei dati può essere temporaneamente spostata su disco, in un'apposita area detta di swap. I componenti principali di una memoria virtuale sono:

- **Spazio di indirizzamento virtuale:** È l'insieme degli indirizzi visti da un processo e viene suddiviso in pagine (solitamente di 4 KB ciascuna).
- **Memoria fisica:** Divisa in frame, della stessa dimensione delle pagine, ospita le pagine virtuali attualmente in uso.
- **Tabella delle pagine:** Mappa ogni pagina virtuale al corrispondente frame fisico (se presente). Contiene anche bit di validità, bit di accesso/scrittura, e altri flag.
- **Memory Management Unit (MMU):** Hardware che effettua la traduzione automatica da indirizzo virtuale a fisico.
- **Translation Lookaside Buffer (TLB):** Piccola cache specializzata per la tabella delle pagine. Serve a velocizzare la traduzione virtuale-fisica evitando accessi ripetuti alla page table.
- **Spazio di swap (su disco):** Contiene le pagine che non stanno nella RAM. Quando serve una pagina assente in RAM (page fault), viene caricata da qui.

### 4.2.1 Traduzione degli indirizzi

Per poter accedere a una qualsiasi locazione, indipendentemente da dove essa sia situata (cache o RAM), il processore genera un indirizzo logico, il quale verrà poi opportunamente tradotto in un indirizzo fisico. L'indirizzo logico, è concettualmente costituito da due parti: la prima è l'identificativo di pagina, la seconda è lo spiazzamento (offset) nella pagina. A seconda della struttura della memoria virtuale (dimensione e numero delle pagine), il numero di bit richiesti per codificare i due campi può variare. Se un processo cerca di fare riferimento a un dato che fa parte di un blocco residente (cioè già caricato in memoria centrale), una struttura hardware effettua la traduzione dell'indirizzo logico in fisico, utilizzando delle opportune tabelle caricate dal sistema operativo. Un processo che invece tenta di fare riferimento a un blocco non residente, genera un'eccezione e viene sospeso, sarà poi compito del sistema operativo gestire il recupero del dato dalla memoria di massa. Il processo di traduzione avviene grazie a una tabella delle pagine mantenuta nella memoria centrale dal processore, la quale, per ogni pagina virtuale riporta l'identificativo della pagina fisica nella quale il SO l'ha allocata. Questa allocazione è dinamica, ovvero la pagina logica può essere posta in punti diversi della memoria principale in seguito a successive operazioni di allocazione e deallocazione. Il processo di traduzione effettuato dalla MMU avviene tramite le seguenti fasi [4.5].

1. La porzione di identificativo della pagine viene prelevata dall'indirizzo virtuale.

2. L'identificativo viene sommato al puntatore alla tabella delle pagine, in modo da ricavare l'indirizzo della pagina fisica in cui il dato è allocato.
3. L'indirizzo base ottenuto al punto precedente viene sommato con l'offset, prelevato dalla seconda parte dell'indirizzo logico.

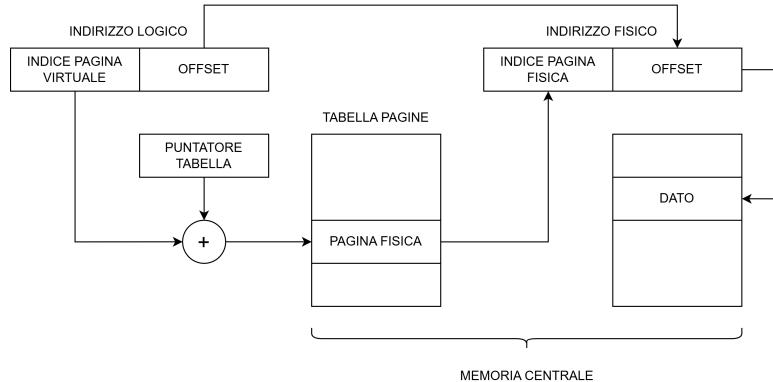


Figura 4.5: Processo di traduzione da indirizzo logico a fisico.

Se durante il secondo passaggio la tabella delle pagine non trova alcuna corrispondenza, viene generata un'eccezione di tipo *page fault* in seguito gestita dal sistema operativo, che ha il compito di attivare il driver che permette di leggere i dati dalla memoria di massa e scriverli in quella principale, aggiornando infine la tabella. In realtà, come anche anticipato durante la discussione degli elementi fondamentali delle memorie virtuali, esiste un importante componente noto come TLB che agisce da memoria cache per il processo di traduzione. Prima di procedere con la traduzione dell'indirizzo logico, la MMU consulta la TLB: se trova una corrispondenza, restituisce subito l'indirizzo base della pagina fisica; altrimenti, procede normalmente con il processo descritto in precedenza.

Ricapitolando, il processore genera indirizzi virtuali che vengono tradotti dalla MMU in indirizzi fisici, utilizzati per accedere alla cache. In caso di *cache miss* (blocco assente nella memoria cache), si accede alla memoria principale, e se si verifica un *page fault* (pagina assente anche in memoria principale), il sistema operativo deve occuparsi del recupero della pagina dal disco. Le politiche di sostituzione della memoria virtuale sono identiche a quelle viste per la cache. Tuttavia, essendo le dimensioni delle pagine molto maggiori di quelle dei blocchi, è assolutamente da evitare la tecnica del write through.

Un meccanismo di traduzione leggermente più avanzato consente di verificare se un certo dato è presente nella cache direttamente a partire dall'indirizzo logico, rimandando il calcolo dell'indirizzo fisico e addirittura evitando (quando possibile) il passaggio per il TLB. Per poter fare ciò, è necessario utilizzare come chiave dei blocchi in cache direttamente la parte di indice della pagina dell'indirizzo fisico, che in generale ha una dimensione maggiore essendo il numero di pagine maggiore del numero di blocchi. Con un tale approccio, si verificherebbe una complicazione in corrispondenza dei cache miss, poiché solo a seguito di questi si procederebbe alla traduzione dell'indirizzo. Per risolvere il problema, il componente di traduzione calcola preventivamente in parallelo l'indirizzo fisico, in modo tale che se il dato non è in cache non ci sono ulteriori sprechi di tempo per la traduzione. Tale sistema è noto come **virtual cache**. Notiamo come questo differisca

dalla semplice traduzione spiegata in precedenza, che è un modello seriale, poiché passa sequenzialmente da indirizzo logico a fisico. Le cache virtuali non sempre sono utilizzate, in quanto complicano notevolmente il progetto delle memorie di primo e secondo livello.

È importante notare che con l'uso di una cache virtuale, l'indirizzo virtuale generato dalla CPU è quello usato per accedere ai dati in cache, questo significa che la suddivisione della cache (settori e blocchi) deve essere fatta in relazione alla memoria virtuale. L'idea è che, durante il processo di traduzione con la MMU, ci accorgiamo che la porzione di offset non cambia tra l'indirizzo fisico e quello logico, per questo motivo conviene usare questa porzione per l'indice del set e l'offset nel blocco, mentre il restante spazio viene usato per la chiave. Questo significa che stiamo utilizzando lo stesso numero di bit per codificare numero di pagine e numero di blocchi, ma essendo la cache di dimensioni minori potrebbero avanzare alcuni bit.

In generale la dimensione delle pagine è fissata a 4 KB ( $2^{12}$  B), quindi servono 12 bit per codificare la posizione di un dato nella pagina (offset), mentre i restanti saranno usati per l'indice della pagina.

#### 4.2.2 Cache e Memoria Virtuale

Ricapitoliamo il processo di accesso a un dato nel caso di cache (fisica) e memoria virtuale.

1. La CPU genera un indirizzo virtuale (VA).
2. La MMU (Memory Management Unit) cerca nella TLB (Translation Lookaside Buffer) se c'è già la traduzione da indirizzo virtuale a indirizzo fisico. Se la TLB ha un hit, si ottiene subito l'indirizzo fisico, altrimenti si consulta la tabella delle pagine in memoria per fare la traduzione.
3. Una volta che si ha l'indirizzo fisico, si fa il lookup nella cache. Se c'è un cache hit, si legge direttamente da lì, altrimenti si va in memoria principale (RAM) a prendere il dato.

Osserviamo come memoria principale e cache utilizzino entrambe lo stesso indirizzo fisico per accedere ai propri dati, ma quello che cambia è l'interpretazione. Per la cache, avremo ad esempio una suddivisione per set, offset nel blocco e chiave; mentre per la RAM la suddivisione sarà per pagina e offset nella pagina. Nel caso di cache virtuale, l'indirizzo virtuale generato dal processore può essere direttamente utilizzato per accedere alla cache, e in caso di miss si usa la MMU e la TLB per la traduzione e l'accesso in RAM.

Per concludere, cerchiamo di puntualizzare alcuni concetti generali che riguardano i sistemi che dispongono sia di memorie cache sia di meccanismi hardware e software per la gestione della memoria virtuale. Il rapporto tra memoria virtuale e cache può in generale essere descritto tramite la seguente tabella [4.1].

Questa vuole stabilire che un dato presente in cache deve necessariamente essere anche presente nella memoria virtuale, in quanto una pagina è in grado di contenere più blocchi. Per lo stesso motivo, il viceversa non vale, ovvero un dato presente nella memoria virtuale non per forza sarà anche nella cache.

<b>Cache</b>	<b>Memoria Virtuale</b>	<b>Possibile?</b>
Miss	Page Fault	Sì
Miss	Page Hit	Sì
Hit	Page Fault	No
Hit	Page Hit	Sì

Tabella 4.1: Compatibilità tra tipi di cache e memoria virtuale

La prima osservazione è che i meccanismi di gestione della memoria virtuale consentono di determinare se e in quale posizione della memoria centrale è contenuto un dato o un'istruzione, mentre la cache è unicamente un sistema per velocizzare l'accesso a dati e istruzioni. Banalmente, ne deriva che se nel caso in cui il dato contenuto in una pagina non sa presente in memoria centrale, non potrà essere presente neanche nella cache. È bene notare, che dal punto di vista funzionale per il processore sia la presenza della cache che quella della memoria virtuale è del tutto trasparente, in quanto tali soluzioni architetturali sono in grado di garantire che ogni richiesta di accesso a un dato da parte del processore sia sistematicamente sempre eseguita, indipendentemente da dove questo è situato.

Vi sono notevoli similitudini tra il principio di funzionamento delle cache della memoria virtuale, in quanto entrambe sono basati sui principi di località e gerarchia. Tuttavia, le differenze tra le due sono molto profonde, e vanno oltre le sole differenze nei tempi di accesso. Innanzitutto, le dimensioni dei blocchi delle cache sono minori di quelle delle pagine, stanti le differenti dimensioni delle memorie in gioco. Ne consegue che la possibilità di non trovare un dato nella cache è molto più alta di quella di non trovare un dato in una pagina residente in memoria. Altra sostanziale differenza sta nel fatto che la gestione della memoria cache è sempre fatta totalmente in hardware per esigenze prestazionali. La gestione della memoria virtuali, invece, può avvenire secondo due modalità: operando su indirizzi fisici come fa la memoria centrale, oppure operando su indirizzi logici (virtual cache). Una cache che utilizza indirizzi fisici è in generale più lenta, perché prima deve essere eseguita un'operazione di traduzione per ottenere un corrispondente indirizzo logico, cosa che richiede del tempo, da aggiungere a quello richiesto dall'operazione di accesso della cache. Qualora invece l'indirizzo in ingresso alla cache sia virtuale, non è necessaria alcuna traduzione preventiva, rendendo il processo più veloce poiché l'operazione di traduzione avviene soltanto in caso di cache miss. Gli svantaggi di tale approccio sono una maggiore complessità nella gestione della allocazione dei dati nella cache. Ad esempio, nel caso di sistemi multitasking, due processi distinti potrebbero voler accedere a uno stesso indirizzo virtuale che potrebbe entrambi a indirizzare lo stesso dato in cache invece di dati distinti. Le soluzioni in questo caso sono due:

1. L'uso di un unico spazio di memoria virtuale comune a tutti i processi del sistema. Questa soluzione che elimina alla base il problema è stata resa possibile dalla disponibilità nei processori più recenti di indirizzi a 64 bit.
2. La possibilità di invalidare la cache e rinnovarla completamente (cache flushing) a seguito di un task switching. In questo modo, i dati nella cache sarebbero riferibili a un solo processo, ma ciò influisce negativamente sulle prestazioni.

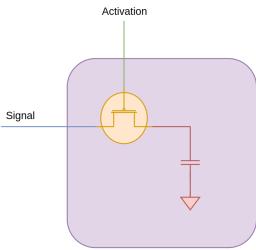


Figura 4.6: Cella di una memoria DRAM

## 4.3 Struttura delle memorie

Le memorie hanno in generale diverse tipologie si strutture di controllo in base alla loro costruzione. La costruzione di una memoria non è universale, ciò è dovuto alla differente implementazione che tali memorie hanno a livello pratico. In una maniera più generale, le memorie possono essere suddivise in due categorie:

- **Memorie statiche:** Le memorie statiche sono memorie che sono costruite tramite un insieme di transistor opportunamente collegati, non richiedono alcun tipo di sistema di refresh dei dati
- **Memorie dinamiche:** Le memorie dinamiche sono memorie che funzionano mediante effetti capacitivi. Pertanto richiedono che vi sia implementato un opportuno sistema di refresh

Tali memorie vengono utilizzate in base all'ambito di applicazione e alla velocità richiesta. Tali argomentazioni saranno affrontate nei capitoli appositi

### 4.3.1 Memorie Dinamiche

Le memorie dinamiche sono memorie che utilizzano gli effetti capacitivi per memorizzare l'informazione. Un esempio semplice di cella di memorizzazione di una memoria dinamica è osservabile alla figura [4.6]. Come possiamo notare, la memorizzazione dipende fortemente dalla carica di un condensatore, che quindi richiede di dover implementare un meccanismo di refresh. Tale meccanismo di refresh può essere fatto in vari modi, in base alle modalità di accesso ai dati della specifica memoria.

Le memorie, oltre alla cella, sono composte da altri componenti di "gestione". Tali elementi sono:

- **Decoder:** Passa dall'indirizzo inserito alla linea (riga) da andare ad attivare
- **Multiplexer:** Permette di selezionare a quale elemento si sta facendo riferimento in fase di lettura (Selezione della colonna)
- **Demultiplexer:** Permette di selezionare un elemento da andare a modificare, spendibile particolarmente nella fase di scrittura dell'elemento (Selezione della colonna su cui andare a scrivere)
- **Sistema di refresh:** Sistema che permette di "rinfrescare" i dati che sono presenti nella memoria, che ad intervalli di tempo esegue il refresh. Tale condizione,

quindi, presuppone un disattivamento della memoria per qualche periodo di tempo, sinonimo di lentezza aggiunta

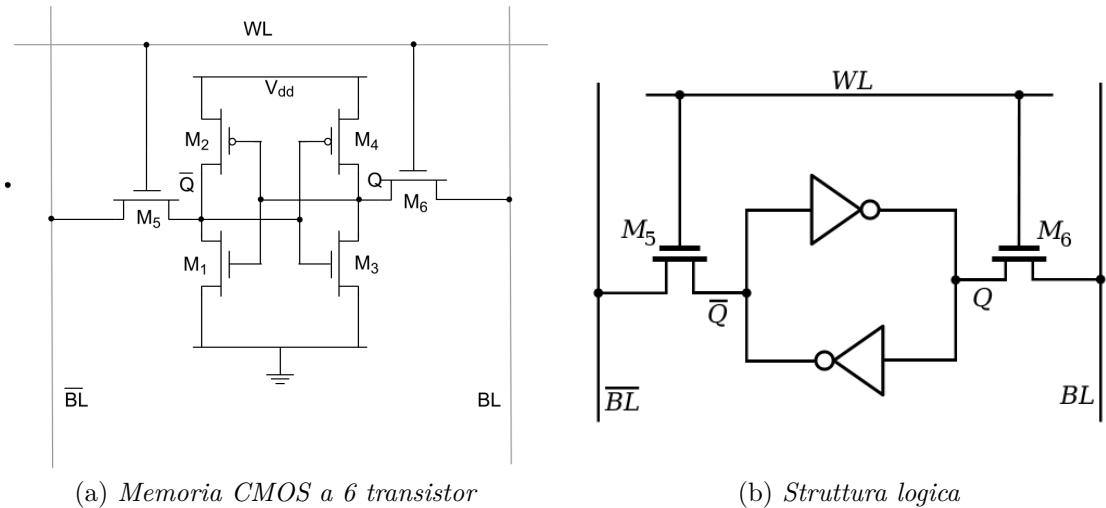
- **Contatori:** Contatori per ricordare, in fase di refresh, l'indice della riga da rinfrescare e il tempo da attendere prima di effettuare un nuovo rinfresco

Il funzionamento principale di una memoria DRAM (che possiamo vedere come una sorta di modello di programmazione) è pilotata da 2 principali indirizzi (provenienti dal singolo indirizzo fisico), che ne permettono di selezionare la riga e la colonna. Tali registri sono chiamati **RAS**(Row Address Strobe o Row Address Select) e **CAS**(Column Address Strobe o Column Address Select), il loro preciso valore viene estratto dalla divisione dell'indirizzo FISICO posto nell'apposito ingresso indirizzi (dell'interfaccia di memoria dall'esterno). La memoria parte col selezionare un intera riga, che, passa in prima battuta tra una serie di amplificatori e poi, tramite il CAS, si va a selezionare il dato desiderato. Tale meccanismo di funzionamento, permette alla memoria, ad ogni lettura di una particolare riga, di poterne effettuare anche il redresh. Tale refresh, però, può essere effettuato anche per determinati tempi di delay (immaginiamo se una riga non viene mai acceduta, c'è bisogno di refresharla prima che i dati all'interno dei condensatori vadano persi), la fase di refresh avviene regolarmente, la pausa tra una fase di refresh e l'altra viene effettuata tramite l'utilizzo di un apposito contatore, che ogni N-impulsi di clock, dove N è il numero di impulsi di clock per scandire un determinato tempo, tale tempo viene deciso in base al tempo di decadenza che caratterizza i condensatori utilizzati per la costruzione della memoria. Pertanto, tali memorie, per effettuare un'operazione, nel peggior dei casi, devono far aspettare almeno il tempo di refresh e poi accedere allo specifico dato. Pertanto tale tecnologia non è tanto utilizzata nelle memorie cache, quando nelle RAM. Difatti ad oggi le RAM sono realizzate principalmente con strutture di tipo dinamico. Oltre ad un tipo di vantaggio fisico, vi è anche un grande vantaggio logico, utilizzando una tecnologia di tipo dinamico per la realizzazione delle RAM, il passaggio di dati tra la cache e la memoria avviene più velocemente, poiché tramite un singolo RAS, ottengo tutta una riga di informazioni che possono essere trasmesse direttamente in cache (tramite le sole modifiche del CAS). In questo modo i trasferimenti tra la memoria cache e quella RAM risultano più ottimizzati e veloci. Per ottimizzare al meglio questo processo, si potrebbe pensare di ottimizzarlo tramite azioni che possono avvenire su fronti di clock differenti, tutte queste operazioni, però, richiedono un sistema puramente sincrono, mentre nel caso delle memorie il funzionamento dev'essere per forza semi-sincrono. Per la memoria cache la semi-sincronicità è verificata per via della presenza dei cache miss (richiedono di dover prelevare dati da altre memorie esterne), mentre nel caso delle memorie dinamiche è dovuta alla presenza della fase di refresh, poiché, se arriva una richiesta durante la fase di refresh, questa viene fatta attendere (poiché la fase di refresh ha maggiore priorità), per poi essere adeguatamente servita.

### 4.3.2 Memorie Statiche

Le memorie statiche sono memorie realizzate principalmente con strutture di transistor Mosfet collegati in un certo modo. Una volta registrato il dato all'interno di tali celle, esso rimane "intrappolato" nella rete logica associata. La struttura di una cella semplice di memoria statica è quella visualizzabile alla figura [4.7]

Tale tipologia di memoria è impiegata principalmente nella realizzazione di memorie cache, poiché risulta altamente veloce ed efficiente. Anche se richiede una complessa gestio-



(a) Memoria CMOS a 6 transistor

(b) Struttura logica

Figura 4.7: Struttura CMOS di una cella di memoria statica semplice

ne dell'hardware (conservazione dello stato di ossido di silicio), la velocità ne compensa, questo poichè la memoria sarà sempre disponibile per il prelievo dei dati senza dover affrontare alcun tipo di refresh, come nel caso delle specifiche memorie dinamiche



# Capitolo 5

## Processore MIPS

Il **processore MIPS** (Microprocessor without Interlocked Pipeline Stages) è un'architettura RISC progettata per essere semplice ed efficiente. Usa un set ridotto di istruzioni, tutte della stessa lunghezza (32 bit), e adotta una pipeline a 5 stadi per eseguire le istruzioni in modo parallelo.

### 5.1 Architettura MIPS

L'**architettura del MIPS** è un'architettura di tipo RISC che consiste di una *integer processing unit* (la CPU) e un insieme di *coprocessori* (fino a 4) [5.1]. Lo standard richiede obbligatoriamente la presenza del coprocessore 0, mentre per gli altri viene lasciata libertà di decisione a chi lo implementa. Nello specifico, la CPU dispone di 32 registri

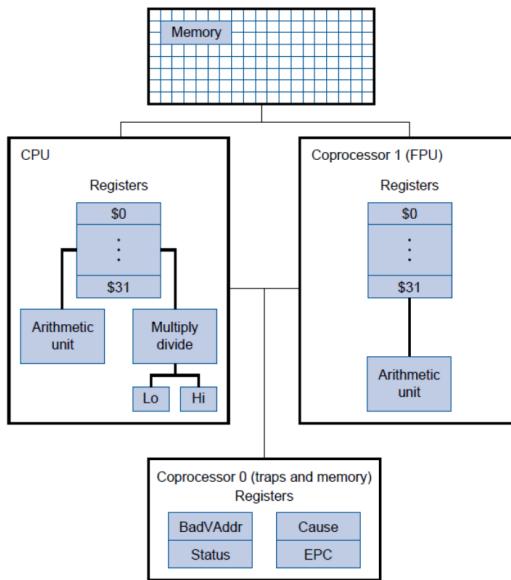


Figura 5.1: Architettura del MIPS.

general purpose a 32 bit per operazioni su interi. Il coprocessore 0 contiene le informazioni necessarie alla gestione delle eccezioni e interruzioni, oltre che lo Status Register (SR). Mentre il coprocessore 1 è la floating point unit, e consiste di 32 registri general purpose a 32 bit per operazioni in virgola mobile. Il processore ha due modalità di

funzionamento, dette *user* e *kernel mode*. I task utente non possono in alcun modo accedere alle risorse di memoria dedicate ai task kernel, incluse le strutture dati del sistema operativo e i dispositivi di I/O. Inoltre, i task utente non possono in alcun modo modificare il funzionamento della macchina, in quanto non possono accedere al coprocessore 0.

Notiamo dall'architettura come le operazioni di moltiplicazione e divisione siano gestite da una componente hardware separata e che lavora su due registri speciali (HI e LO). Il motivo è che queste operazioni richiedono molti cicli di clock, e interromperle potrebbe essere troppo oneroso.

Il MIPS ha un'architettura della memoria totalmente diversa rispetto a quella presentata nel classico modello di Von Neumann, ispirandosi al modello Harvard che prevede due memorie separate per dati e istruzioni (anche se queste condividono lo stesso spazio fisico) [5.2]. Inoltre, all'interno della CPU viene posta una piccola memoria veloce che contiene una serie di registri generali utilizzabili durante le istruzioni, nota come *Register File*. Tale memoria ha due porte di lettura e una di scrittura, per cui può leggere da due registri contemporaneamente (ad esempio, dove sono gli operandi) e scriverne uno solo alla volta (ad esempio, per salvare il risultato).

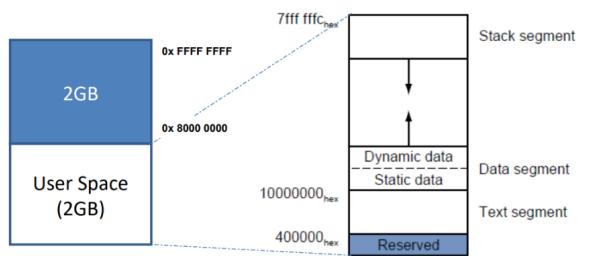


Figura 5.2: Organizzazione della memoria nel MIPS.

## 5.2 Gestione delle interruzioni

Per quanto riguarda la **gestione delle interruzioni**, questa avviene completamente in software, grazie anche all'uso del coprocessore 0 che ha il compito di memorizzare tutte le informazioni necessarie. I due registri utilizzati sono Status, che contiene l'interrupt mask, e Cause, che invece contiene il tipo di interruzione e le interruzioni pendenti. Infine, il registro EPC contiene l'indirizzo che ha causato l'eccezione, in modo da poter permettere di ritornare dopo aver eseguito la ISR. Diversamente da quanto accade nel Motorola 68000, è presente un unico Interrupt Handler richiamato da qualsiasi tipo di interruzione. Tale routine risulterà particolarmente complessa, in quanto deve decidere cosa fare in base al tipo di interrupt generata e ai bit presenti nel registro di stato (un vero e proprio switch-case). Più precisamente:

1. In seguito a un'eccezione o un'interruzione, il processore MIPS salta all'indirizzo che contiene l'exception handler.
2. Il codice contenuto a tale indirizzo esamina la causa dell'eccezione e invoca il sistema operativo, il quale effettua le dovute operazioni (sospensione di un processo, caricamento di una pagina, lettura o scrittura su I/O).

- Dopo aver gestito l'eccezione/interruzione, l'handler ritorna all'istruzione contenuta in EPC oppure a quella immediatamente seguente.

Una tale gestione semplifica notevolmente l'hardware, lasciando una gestione flessibile al sistema operativo.

## 5.3 Pipeline nel MIPS

Le fasi fondamentali che caratterizzano la pipe del MIPS differiscono leggermente da quelle nel caso generale descritto nell'apposito capitolo [??]. Queste sono:

- Instruction Memory (IM):** Si preleva l'istruzione dalla memoria delle istruzioni e si incrementa il Program Counter.
- Register File (Reg):** L'istruzione viene decodificata, si prelevano degli operandi dai registri generali presenti nel register file e si preparano i segnali di controllo per la fase di esecuzione.
- Execute (EX):** Vengono eseguite le operazioni logico-aritmetiche pilotate dai segnali della fase precedente, oppure calcolato l'indirizzo di memoria per eventuali operazioni di *load* e *store*. In caso di istruzione di salto, si calcola l'indirizzo a cui saltare.
- Data Memory (DM):** Avviene la lettura o scrittura dalla memoria dati. Le istruzioni ALU semplici saltano questa fase.
- Register File (Reg):** Il risultato dell'ALU o il dato letto dalla memoria viene scritto nel registro di destinazione.

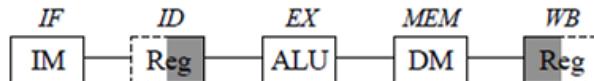


Figura 5.3: Divisione temporale delle fasi della pipeline nel MIPS.

### 5.3.1 Ipotesi di funzionamento

Analizziamo come l'architettura del MIPS garantisca il soddisfacimento delle ipotesi fondamentali della pipeline.

Come per tutte le architetture pipelined, anche il MIPS richiede che tra le varie fasi vi siano dei registri che conservino lo stato dell'operazione. Il MIPS, pertanto, presenta vari casi di conflitto che richiedono delle **ipotesi** sul sistema stesso. Tali ipotesi sono:

- Velocità della memoria:** La memoria che sarà utilizzata dal MIPS sarà acceduta molto frequentemente (precisamente 5 volte in più rispetto al caso senza pipe).
- Concorrenze sulla memoria:** Presenza di due memorie, una per le istruzioni ed una per i dati. Tali memorie vengono previste per evitare la concorrenza tra la fase di fetch (prelievo dell'istruzione dalla memoria) e la fase di MEM (lettura o scrittura dalla memoria).

- **Concorrenza sui registri:** Potenzialmente sul register file possono verificarsi dei conflitti, in particolare quando contemporaneamente un’istruzione vi accede in lettura e una in scrittura. Per risolvere il conflitto, le operazioni di lettura e scrittura vengono effettuate in due intervalli separati dello stesso ciclo di clock, e in particolare la fase di decode (lettura) opera nella seconda parte del ciclo di clock, mentre la fase di write back (scrittura) opera nella prima parte, come riportato in figura 5.3.

### 5.3.2 Istruction Memory

La **fase di Istruction Memory** è caratterizzata da 2 operazioni, quella di prelievo dell’indirizzo e quella di incremento del PC. L’istruzione viene letta dalla Instruction Memory (indirizzata dal PC) e posta nel registro di comunicazione tra le fasi di IM e Reg. Dopodiché, il PC viene incrementato e copiato nel registro di comunicazione, perché potrebbe, nel caso di istruzioni di salto, servire a fasi successive. Pertanto la sua parte architetturale è formata dalle componenti:

- **ADD:** Per favorire l’incremento del PC (che avviene molto frequentemente), l’operazione viene effettuata da un circuito dedicato, in modo da calcolare il prossimo indirizzo del PC senza interferire con altre operazioni dell’ALU.
- **MUX:** Sceglie se considerare l’indirizzo di memoria successivo del PC o un registro di memoria dettato dalla fase di MEM (caso di istruzioni di salto).
- **PC:** Registro Program Counter.
- **Istruction Memory:** Prelievo dell’istruzione da eseguire dalla memoria.

### 5.3.3 Register File (ID)

Nella **fase di Register File**, il MIPS va a decodificare ed interpretare il comando. In particolare, vengono letti gli eventuali registri sorgente (nel caso di indirizzamento register o immediato) e memorizzati nel registro di comunicazione tra Reg e ALU. Nel caso di istruzioni con indirizzamento immediato, i 16 bit del valore vengono prima convertiti in un valore a 32 bit estendendo il segno. Inoltre, il valore del PC precedentemente passato dalla fase di IM viene propagato insieme ad altre informazioni al registro di comunicazione successivo. Osserviamo che il registro destinazione in cui memorizzare il dato nella fase Reg finale deve essere propagato lungo tutta la pipeline. Le parti che compongono l’architettura di questa fase sono:

- **Registri:** Registri interni del processore che possono essere pilotati sia in lettura che in scrittura tramite dei segnali esterni.
- **Estensione del segno:** Blocco di estensione con segno dei possibili valori immediati contenuti nell’istruzione.

Il blocco estensione del segno è fondamentale perché l’architettura MIPS utilizza istruzioni di lunghezza fissa di 32 bit per semplificare il design del processore e migliorare la velocità di decodifica, e dunque la codifica dei valori acceduti con indirizzamento immediato è fissata a 16 bit. L’estensione a 32 bit è necessaria perché garantisce che il valore immediato possa essere direttamente utilizzato per le operazioni aritmetiche o logiche con i registri (che sono anch’essi a 32 bit).

### 5.3.4 ALU

Nella **fase ALU** viene eseguita effettivamente l'istruzione. In particolare, vengono prelevati i dati e i bit di controllo dal registro Reg/ALU e vengono effettuate le operazioni logico-aritmetiche dalla ALU. I risultati vengono memorizzati nel registro di comunicazione tra ALU e DM. In caso di istruzioni di salto, viene presa la decisione di saltare o meno, e viene calcolato il nuovo valore di PC, che viene inserito sempre nel registro di comunicazione ALU e DM. In caso di istruzioni *load* o *store*, viene calcolato l'indirizzo dell'operando in memoria e posto nel solito registro intermedio delle fasi. La parte architettonale è composta da vari componenti:

- **ADD**: Strumento che viene utilizzato per calcolare un eventuale offset rispetto ad un valore, e viene utilizzato per calcolare il nuovo valore del PC in caso di salti condizionati o incondizionati.
- **LEFT SHIFT 2**: Moltiplica per 4 il valore per cui voglio saltare (in modo da saltare a 4 indirizzi più avanti).
- **ALU**: Strumento di calcolo aritmetico-logico.
- **MUX**: Seleziona o il dato immediato o un secondo dato proveniente da un registro, proveniente dalla fase precedente.

### 5.3.5 Data Memory

Nella **fase di Data Memory** avviene l'accesso alla memoria dati, secondo l'indirizzo comunicato dalla fase precedente nel registro ALU/DM. In caso di istruzione di *load*, il dato viene letto e propagato nel registro DM/Reg. La sua architettura è composta principalmente dal singolo elemento di accesso alla memoria dati.

### 5.3.6 Register File (WB)

Nella seconda **fase di Register File** si verifica cosa bisogna scrivere all'interno dei registri interni. Il componente che meglio indica il suo funzionamento è il multiplexer finale, che serve per specificare se il dato da caricare all'interno dei registri sia il risultato dell'ALU o qualche valore proveniente dalla memoria dati.

## 5.4 Gestione dei salti

Il MIPS adotta una strategia di **gestione dei salti** molto diversa da quella discussa nel caso generale. Infatti, subito dopo un'istruzione di salto, c'è una *branch delay slot*: l'istruzione successiva viene sempre eseguita, indipendentemente dal risultato del salto. Lo scopo è mantenere la pipeline piena e attiva mentre si valuta il salto. La decisione se saltare o meno avviene nella fase ID, ma nel frattempo (come sappiamo dal comportamento generale) la pipeline ha già precaricato l'istruzione successiva, detta *delay slot*. A quel punto possono verificarsi due situazioni:

1. Se il salto non viene effettuato, si continua semplicemente con l'esecuzione della successiva istruzione.

2. Se il salto viene effettuato, si esegue sempre prima la delay slot e poi si passa all'istruzione situata all'indirizzo del salto.

Per quanto detto, il MIPS non usa predizione dei salti, ma fa sempre l'assunzione che il salto non sarà preso (predict-not-taken) e in caso contrario esegue comunque le istruzioni che non sarebbero dovute essere eseguite (la delay slot) per evitare di sprecare cicli di clock. Questo presuppone che sia il programmatore a fare attenzione a non piazzare istruzioni "sensibili" dopo salti.

È assolutamente importante, a causa della strategia di gestione dei salti adottata, inserire sempre un'istruzione utile dopo un salto, altrimenti la branch delay slot rimarrebbe indefinita. Per poter realizzare questa accortezza, si inserisce un'istruzione NOP (no operation) dopo il salto nel caso in cui non ci siano istruzioni successive da eseguire.

# Capitolo 6

## Processori ARM

In questo capitolo verrano affrontate le tematiche inerenti al processore ARM e la programmazione del microcontrollore *STM32F4 Discovery/Nucleo*.

### 6.1 Architettura ARM

L'architettura ARM è un'architettura RISC sviluppata da *ARM Holding*. La ARM holding si occupa in realtà solo del design delle **IP-Core**: Un IP Core (Intellectual Property Core) è un blocco funzionale di circuiti elettronici progettato per essere riutilizzabile e concesso in licenza ad altre aziende. Un SoC (System on Chip) è un singolo chip che integra al suo interno tutti (o quasi tutti) i componenti fondamentali di un sistema elettronico. In pratica, è un intero computer miniaturizzato su un solo pezzo di silicio. Un SoC è costituito da alcuni componenti fondamentali (figura 6.1):

- Core principale e logica di *debug* con protocollo JTAG (protocollo di comunicazione seriale standardizzato usato per il debug, la programmazione e il test);
- Interfacce modulari per la generazione di un segnale di clock e un segnale di reset;
- Interrupt controller (ARM consente di avere due tipologie di interruzioni: quelle normali e quelle veloci, ad esempio i trasferimenti del DMA);
- Bus di interconnessione: i bus nelle architetture ARM assumono che non tutte le periferiche hanno bisogno della stessa velocità di trasferimento, e per questo motivo ce ne sono di due tipologie:
  - – APB - advanced peripheral bus, per le periferiche più lente;
  - AXI - advance eXtensible interface, supporta connessioni veloci e complesse (many to many), dunque necessita di una *matrice di interconnessione*;
- Memoria flash;
- Memoria RAM.

L'architettura presentata è generale ed è comunque molto flessibile: ogni azienda produttrice la personalizza in base alle proprie esigenze e alle mansioni per cui il sistema è dedicato. Distinguiamo immediatamente tre famiglie di CPU ARM:

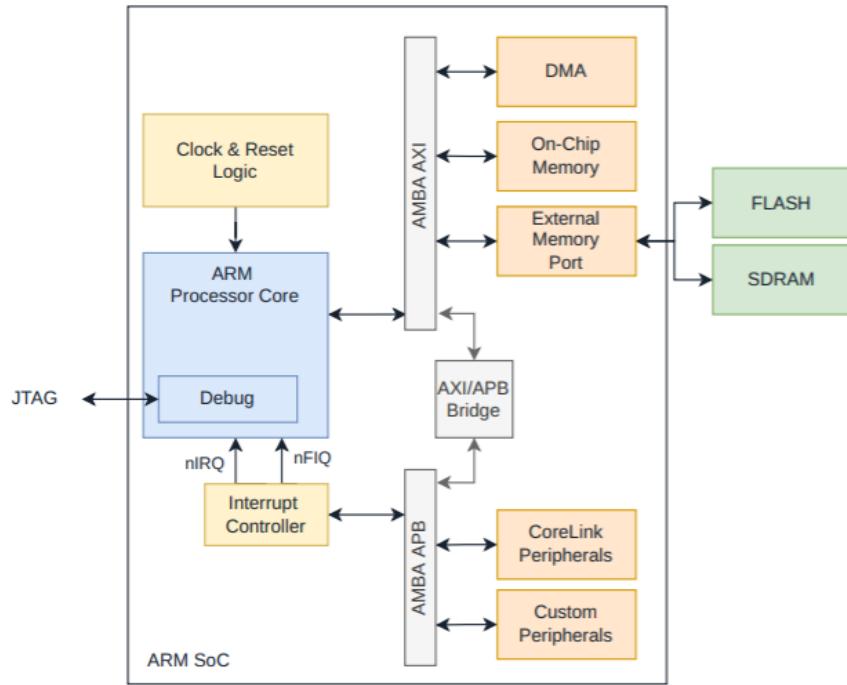


Figura 6.1: Architettura interna SoC

- **ARM CORTEX A** - fascia ad alte prestazioni, orientata ad applicazioni general purpose;
- **ARM CORTEX R** - CPU pensate specificamente per supportare applicazioni time critical;
- **ARM CORTEX M** - CPU pensate per sistemi embedded come i microcontrollori.

È importante insistere sulla differenza tra architettura di un processore e implementazione di un processore che supporti tale architettura: **L'architettura** definisce un modello di programmazione, un insieme di registri, un insieme di istruzioni e un modello per la gestione delle interruzioni/eccezioni; **L'implementazione** invece è la realizzazione particolare di una determinata architettura (ARM-A8 e ARM-A9 sono entrambe implementazioni dell'architettura ARMv7-A, ma presentano dettagli realizzativi della pipeline diversi, pur presentando lo stesso set di istruzioni).

### 6.1.1 Modello di programmazione

ARM v-7 è un'architettura RISC a 32 bit, in cui la maggior parte delle istruzioni esegue in un solo colpo di clock. L'insieme di registri è **Ortogonale** (quando si afferma che l'insieme dei registri di un processore è ortogonale, si intende che tutti i registri possono essere utilizzati in modo intercambiabile all'interno delle istruzioni della CPU. In altre parole, ogni istruzione che opera sui registri può essere applicata a qualsiasi registro, senza restrizioni o privilegi particolari per alcuni di essi). Inoltre è un architettura *load-store*, ovvero si può accedere alla memoria principale solo attraverso le due istruzioni load e store, mentre tutte le altre istruzioni operano su registri interni del processore

(scelta tipica delle architetture RISC). Molte architetture ARM supportano due famiglie di istruzioni:

- ARM Native - set di istruzioni a 32 bit;
- Thumb - set di istruzioni a 16/32 bit, meno efficienti ma in generale più compatte, ideali per sistemi in cui è fondamentale il management della memoria.

ARM ha sette diverse modalità operative, ognuna ha uno stack dedicato ed usa un determinato sottoinsieme di registri. Questa scelta è denominata **Register banking**: Il register banking nei processori ARM è una tecnica che consiste nell'avere più copie fisiche (banchi) di alcuni registri, usate in base alla modalità di esecuzione o al tipo di eccezione. Questo consente di velocizzare i context switch (cambi di contesto), evitando il salvataggio immediato dei registri in memoria. Lo scopo è quello di ottimizzare il cambio di contesto, evitando di salvare e ripristinare lo stato precedente, passando istantaneamente a un set alternativo di registri già pronti (figura 6.2).

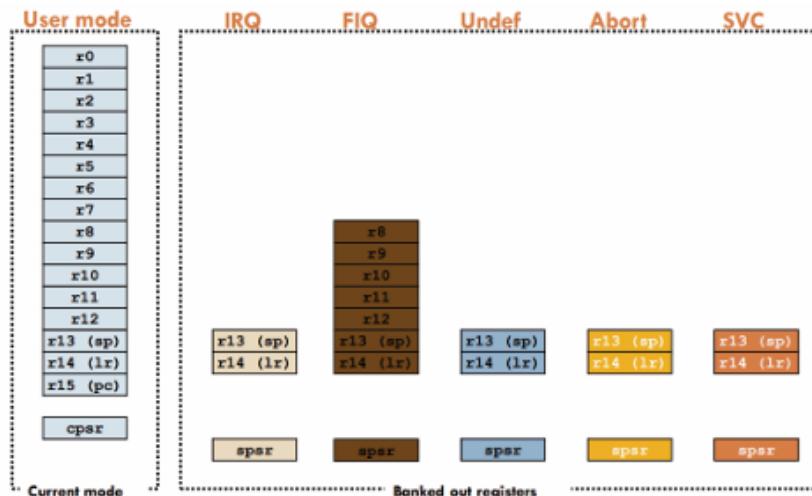


Figura 6.2: Sistema Register Banking

Le modalità operative sono:

- **SVC**: Stato di avvio, il processore può raggiungere questo stato eseguendo una SuperVisor-Call;
- **FIQ**: Stato raggiunto quando la linea Fast Interrupt ReQuest rileva un segnale;
- **IRQ**: Stato raggiunto quando la linea Interrupt ReQuest rileva un segnale;
- **Abort**: Stato raggiunto quando viene effettuato un accesso illegale in memoria;
- **Undef**: Stato raggiunto quando viene tentata un'istruzione non esistente o non definita in memoria;
- **System**: modalità sistema, permette di accedere ai registri *user* in maniera privilegiata;
- **User**: modalità utente, unica modalità non privilegiata, destinata ai programmi utente.

SVC, FIQ, IRQ, Abort e Undef sono anche definite modalità 'eccezione'.

Su ARMv7-M le modalità operative invece sono solo due:

- *Thread Mode*: modalità per programmi utente. È la modalità in cui si avvia il processore;
- *Handle Mode*: usata da tutti i gestori delle eccezioni e delle interruzioni.

In questo processore privilegi, stack e registri possono essere configurati.

### 6.1.2 Gestione delle Eccezioni

Definiamo *eccezione* in questo contesto qualsiasi evento che interrompa il normale flusso di esecuzione di un programma. Queste possono essere interne (memory faults) o esterne (bus error), possono essere inoltre sincrone (esecuzione di una SVC, ovvero una supervisor call) o asincrone (richiesta di una periferica).

Dal punto di vista architetturale, le interrupt sono vettorizzate. Questo presuppone che esista una *interrupt vector table* e che esista un intermediario (qualcosa di simile al PIC 2.1.6.2) tra processore e sorgenti di interruzione, che può essere un **Generic Interrupt Controller** nei sistemi A e R, mentre un **Nester Vector Interrupt Controller** nei sistemi M.

Le funzioni principali del GIC sono:

- Ricevere interruzioni hardware da periferiche;
- Classificare le interruzioni in base a priorità, tipo e target;
- Decidere a quale *core* inoltrare l'interruzione;
- Mascherare, abilitare, cancellare interrupt.

NVIC è un oggetto simile ma progettato per il real time, con latenza minima e gestione semplice delle interruzioni nidificate. È di solito integrato nel core.

Quando viene scatenata un'eccezione, viene gestita nel seguente modo:

- 1 - Il processore salva il PC nel registro **LR\_<modalità>**;
- 2 - Il processore salva il CPSR (stato corrente del processore) in **SPSR\_<modalità>**;
- 3 - Cambia la modalità operativa (IRQ mode o FIQ mode);
- 4 - Disabilita se necessario altre eccezioni;
- 5 - Consulta il vettore delle interruzioni;
- 6 - Esegue il codice dell'handler;
- 7 - Ripristino di CPSR e valore del PC.

Il salvataggio di CPSR non è altro che un salvataggio dei registri correntemente usati.

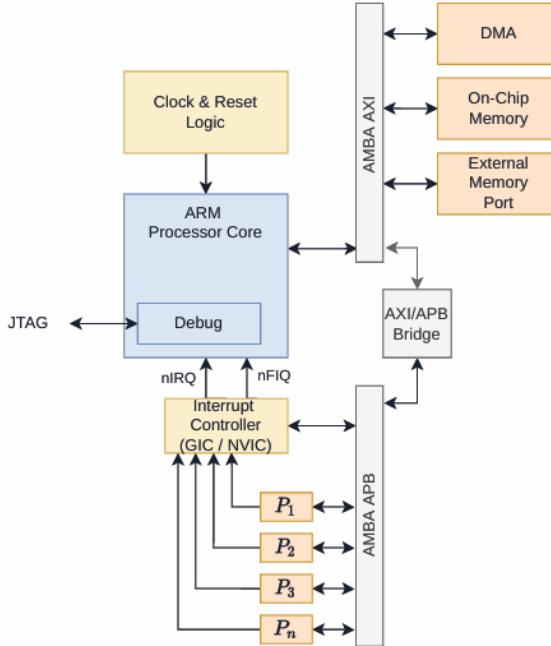


Figura 6.3: Vista architetturale

### 6.1.3 Differenze principali tra architetture

Illustriamo le principali differenze tra ARM, MIPS e M68K:

- Instruction Set Architecture (ISA):
  - M68k dispone di molte istruzioni in grado di accedere alla memoria principale *direttamente*;
  - MIPS e ARM hanno un'ISA di tipo *load-store*: solo queste possono accedere alla memoria principale, le altre operazioni sono eseguite sui registri interni;
- lunghezza istruzioni:
  - M68k dispone di istruzioni a lunghezza variabile;
  - MIPS e ARM dispongono solo di istruzioni a lunghezza fissa;
- Modi di indirizzamento:
  - M68k dispone di 6 modi di indirizzamento, più le varianti per un totale di 14;
  - MIPS ne ha soltanto 3;
  - ARM ha gli stessi di MIPS più **pc-relative** utilizzabile nella load per caricare costanti salvate in area programma, **pre-indexed** e **post-indexed** utilizzabili per incrementare automaticamente il registro prima o dopo l'accesso in memoria;
- Flusso di controllo:
  - in MIPS le istruzioni testano il contenuto di un registro;

- in ARM/68k le istruzioni testano lo stato del processore (le istruzioni aritmetiche e logiche alterano lo stato);
- Subroutines:
  - MIPS/ARM salvano l'indirizzo di ritorno in un registro;
  - M68k salva l'indirizzo di ritorno sullo stack;

## 6.2 STM32F4

La Board si divide in due settori: Uno che contiene l'interfaccia di programmazione (si tratta di una scheda di progettazione), e si tratta di un altro microcontrollore removibile della ST che fa da interfaccia verso il computer (riceve il binario dal pc, e si interfaccia con il core con protocollo SVG o JTAG); L'altro settore invece contiene il SoC, che è proprio l'oggetto etichettato STM32F407VTGT6 (fig 6.4), mentre tutto ciò che c'è intorno consente all'utente di interfacciarsi con il SoC. I piedini presenti tra il SoC e la board sono condivisi tra più periferiche, quindi in fase di configurazione dobbiamo capire quale delle periferiche lo userà. Questi *piedini* poi dovranno interagire con il mondo esterno, e ci viene in aiuto il *jumper cable* che consente di collegare i pin della board con la periferica esterna.

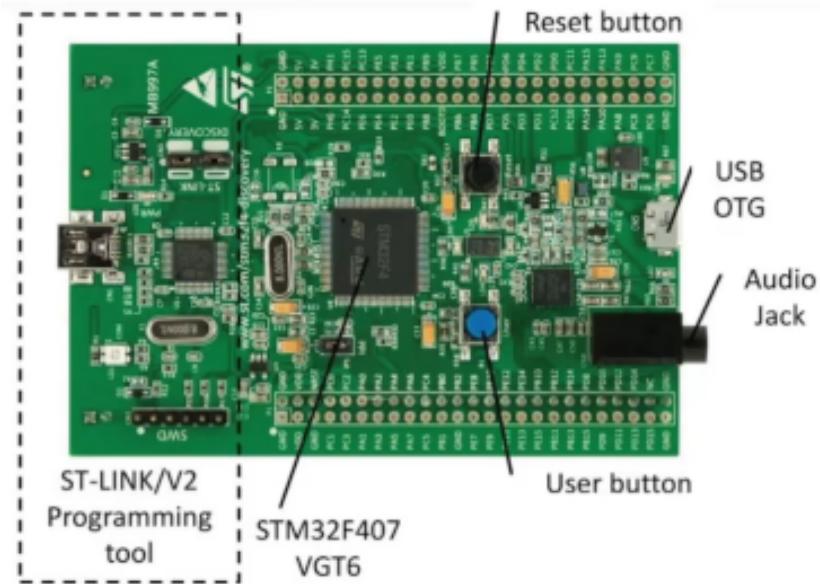


Figura 6.4: STM32F4

Le periferiche illustrate sul *reference manual* sono memory mapped, quindi raggiungibile mediante accessi in memoria. Per ogni periferica, in fase di configurazione si va a scrivere in alcuni registri in memoria.

### 6.2.1 CubeIDE

L'IDE utilizzato al corso per la programmazione della scheda è STM32CubeIDE. Selezionare la giusta board è importante perché non si sta programmando un SoC nudo, ma una board, dove le periferiche sono mappate in maniera fissa su alcune piedinature del

SoC. Una volta creato il progetto, ST fornisce un'interfaccia grafica per configurare i pin del MCU. Poiché i pin sono molto numerosi, sono raggruppati in *porte*, enumerate con lettere. Ogni porta gestisce un determinante sottoinsieme di PIN. Per evitare di configurare ogni periferica attraverso l'accesso in memoria e il modello di programmazione, i produttori di microcontrollori offrono un'astrazione delle periferiche denominate **HAL** (Hardware abstraction layer). Questi offrono un'interfaccia di programmazione astratta della periferica.

### 6.2.2 Gestione mutua esclusione

Per parlare della necessità della mutua esclusione, conviene partire da un esempio. Consideriamo la trasmissione UART in ISR, processo in cui sono coinvolte la periferica UART e due principali ISR. La **UART** è logicamente composta da (figura 6.5):

- **Transmit Data Register:** Registro contenente i byte/word da inviare, e viene acceduto in scrittura dalla CPU;
- **Shift Register:** Registro a scorrimento per la trasmissione seriale. Durante una trasmissione, il contenuto del TDR viene copiato nello Shift Register, e ogni byte viene shiftato verso l'esterno tramite il controllo della CU;
- **Control Unit:** Unità di controllo della periferica con i suoi registri dedicati.

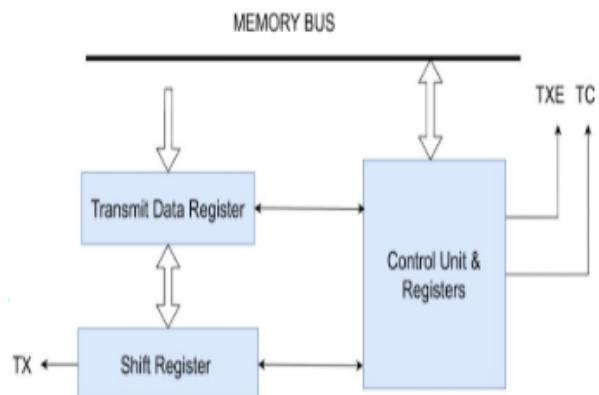


Figura 6.5: UART schema logico

Due **ISR** sono principalmente coinvolte nel processo di trasmissione:

- **Transmit Empty:** indica che il contenuto del registro TDR è stato copiato nello shift register, quindi TDR può essere sovrascritto da un nuovo byte/word;
- **Transmit Complete:** Trasmissione completata.

Queste interruzioni vengono attivate mediante la scrittura dei registri interni della CU *TXEIE* e *TCE*. Presentiamo la funzione che serve a trasmettere dati via UART in modalità interrupt *IT* non bloccante.

```

1 HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart ,
2   const uint8_t *pData , uint16_t Size){
  if (huart->gState == HAL_UART_STATE_READY){
  
```

```

3     huart->pTxBuffPtr   = pData; // Copia buffer e dimensione del
4         buffer.
5     huart->TxXferSize   = Size;
6     huart->TxXferCount = Size;
7     huart->TxISR        = NULL;
8     huart->ErrorCode   = HAL_UART_ERROR_NONE;
9     huart->gState       = HAL_UART_STATE_BUSY_TX; // Aggiorna lo
10        stato della periferica per indicare che ci sia una
11        trasmissione in atto.
12    // Setta la ISR da invocare in dipendenza della dimensione
13    // della parola
14
15    if ((huart->Init.WordLength == UART_WORDLENGTH_9B) && (huart->
16        Init.Parity == UART_PARITY_NONE))
17        huart->TxISR = UART_TxISR_16BIT;
18    else
19        huart->TxISR = UART_TxISR_8BIT;
20    /* Scatena la prima invocazione della ISR, abilitando le
21       trasmissioni di Transmit Buffer Enable*/
22    ATOMIC_SET_BIT(huart->Instance->CR1, USART_CR1_TXEIE);
23    return HAL_OK;
24}
25else{
26    return HAL_BUSY;
27}
28}

```

La firma della funzione ci dice che la funzione accetta come parametri un puntatore alla struttura che contiene informazioni sulla periferica UART virtualizzata, un puntatore al buffer di dati da trasmettere, e il numero di byte da trasmettere; ritorna uno stato che può essere HAL\_OK o HAL\_BUSY. Se lo stato della UART è READY, allora il codice procede ad una configurazione. Presteremo particolare attenzione alla scrittura sulla CU che permette l'abilitazione dell'interruzione TXE, che è atomica. Per poter inviare il primo carattere (oppure i primi 16 bit del messaggio, in base alla modalità) è necessario scatenare una prima interruzione di trasmissione. Per questo motivo, la funzione scrive il bit TXEIE al fine di abilitare le interruzioni nel caso di TDR vuoto. A questo punto, scatterà la prima interruzione e dato che il TDR è vuoto all'inizio, il byte/word verrà copiato in TDR e in seguito trasmesso. Scrivere un solo bit in un registro, dato che l'architettura ARM è load-store, coinvolge almeno le operazioni di LOAD, OR e STORE. Supponiamo che il registro dell'unità di controllo sia a 8 bit con valore iniziale 0x00, e supponiamo che il main voglia scrivere 1 nel bit meno significativo. Se una ISR interrompe il main prima che possa fare la OR e caricare il nuovo byte sul registro, e modifica il registro per esempio scrivendo il bit più significativo, il main avrà uno stato del registro non consistente, e opererà come se la scrittura non fosse mai avvenuta. Quindi il risultato sarà determinato solo dalle istruzioni eseguite dal main (fig 6.6) Questa è una situazione molto diffusa durante la scrittura dei registri di controllo delle periferiche. La soluzione messa a disposizione da ARM è un *TAG* esclusivo ad un determinato indirizzo di memoria, denominato **local-exclusive-monitor**. Il tag indica che l'indirizzo di memoria viene acceduto in maniera esclusiva. In particolare, ARM mette a disposizione due istruzioni per gestire i registri di memoria esclusivi:

- **LDREX**: Load exclusive, carica il valore di un indirizzo di memoria ed inserisci il

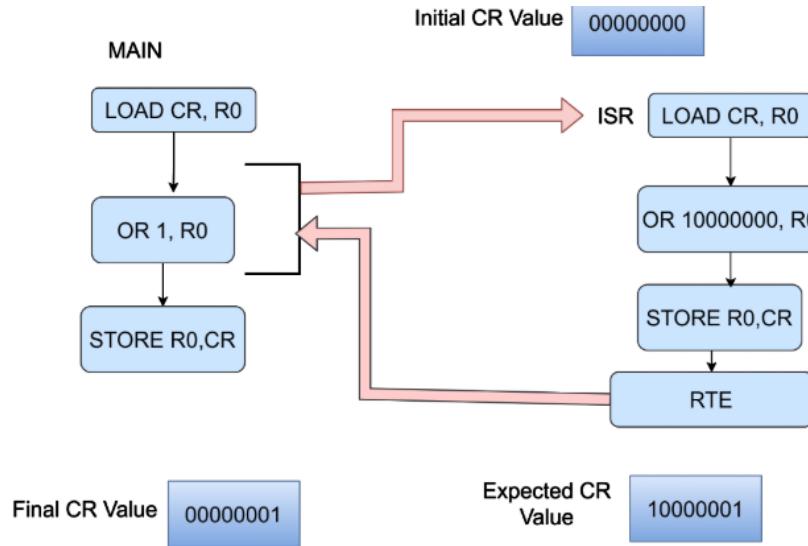


Figura 6.6: Esempio di inconsistenza

TAG di esclusivo;

- **STREX:** Scrivi il valore di un indirizzo di memoria solo se ha il tag esclusivo; Dopo la scrittura, il tag viene rimosso. Se l'indirizzo di memoria non ha il tag esclusivo, la scrittura fallisce.

Il funzionamento di questo meccanismo è illustrato dall'automa semplificato in figura 6.7.



Figura 6.7: Automa semplificato

Usando LDREX e STREX al posto delle normali istruzioni di LOAD e STORE, risolviamo il problema presentato precedentemente come illustrato in figura 6.8:

Questo meccanismo non causa deadlock, infatti il main riuscirà a scrivere al secondo tentativo, mentre la ISR riuscirà al primo. Questa funzionalità viene realizzata dalla primitiva ATOMIC\_SET\_BIT(REG,BIT):

```

1 #define ATOMIC_SET_BIT(REG, BIT)
2 do{
3     uint32_t val;
4     do{
5         val = __LDREXW((__IO uint32_t *)&REG | (BIT));
6     }while(__STREXW(val, (__IO uint32_t *)&(REG)) != 0U)
7 }
```

Osserviamo che do...while(0) è una tecnica tipica della definizione di funzione tramite MACRO, non ha nessun significato logico a livello di codice ma solo sintattico.

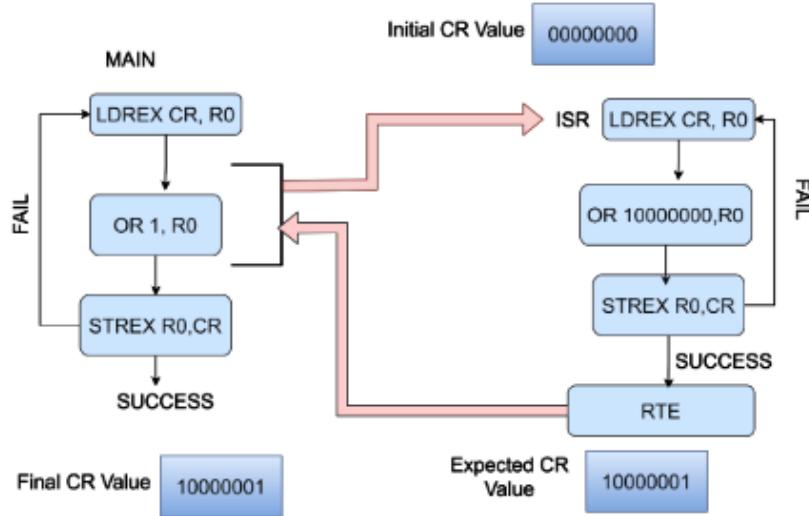


Figura 6.8: Risoluzione inconsistenza

### 6.2.3 DMA

Nell’interfaccia grafica di CUBE *Pinout & configuration*, è possibile selezionare nell’interfaccia Connectivity le impostazioni delle varie periferiche. Nelle impostazioni della UART, è possibile configurare le impostazioni del DMA. Cliccando su *Request*, si può configurare il DMA il ricezione o trasmissione. Il SoC ha diversi tipi di DMA, e ogni periferica può accedere ad un determinato DMA di uno specifico tipo, non tutto è connesso a tutto. In fase di configurazione si imposta a quale DMA la periferica può accedere e in che modalità. L’HAL della UART offre delle funzioni per la ricezione e la trasmissione mediante DMA. Consultare l’HAL descriptor nella documentazione per ulteriori informazioni.

## 6.3 BUS e protocolli

In questa sezione presentiamo i bus e i protocolli visti a lezione inerenti all’architettura ARM utilizzata sul SoC.

### 6.3.1 I<sup>2</sup>C

*I<sup>2</sup>C* (Inter-Integrated Circuit) è un sistema di comunicazione seriale nato negli anni ’80 sincrono utilizzato tra circuiti integrati. Risponde alla problematica di ridurre il più possibile il numero di cavi in una comunicazione seriale, ed è utilizzata per interconnettere sensori ad un’architettura principale. Come tutti i sistemi seriali, è sincrono. Questo protocollo/bus distingue i device in *Master*, che iniziano la comunicazione, e *Slave*. Questo sistema prevede più device master connessi allo stesso bus. Tecnicamente prevede due segnali, uno per la trasmissione dei dati e uno per il segnale di sincronizzazione. Il limite principale di questo protocollo è la velocità di trasmissione.

I cavi necessari sono:

- SDA - Serial Data: cavo per la trasmissione dei dati;
- SCL - Serial Clock: cavo per la trasmissione del clock;

- GND - Ground comune tra dispositivi;

Ai segnali di SDA e SCL viene aggiunta una terza linea Vcc, a cui sono connessi i SDA e SCL tramite resistori di pull-up: infatti lo stato di riposo delle linee è il valore logico HIGH (figura 6.9).

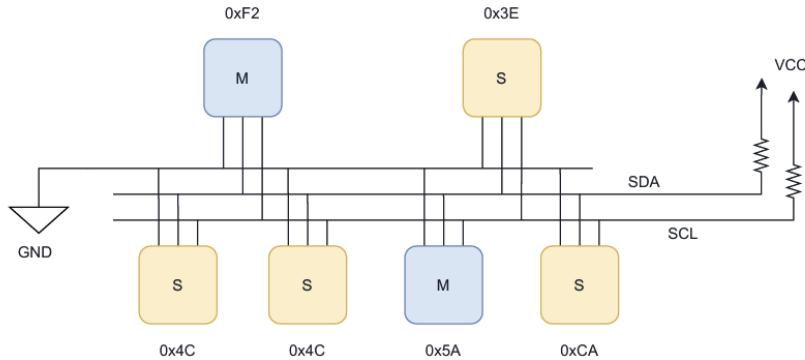


Figura 6.9: i2c bus - schema logico

Ogni device connesso sul bus è identificato da un indirizzo a 7 bit. L'indirizzo è assegnato al device in fase di produzione, e nella maggior parte dei casi non è modificabile. Possono tuttavia coesistere dispositivi aventi lo stesso indirizzo

La comunicazione I2C viene iniziata da un master, che comunica il bit S di *START*, ovvero una transizione da alto a basso del segnale di dato mentre il clock SCL è a livello logico alto. Dopodiché, il master porta SCL a livello logico basso e impone sul segnale SDA il valore del primo bit. Dopo i transienti, il dato è stabile e può essere letto (tratto verde di figura 6.10), segnalato dalla commutazione di SCL. Si continua in questo modo trasmettendo tutti gli altri bit. La transazione termina con un ottavo bit finale che indica la natura dell'operazione (Lettura o scrittura verso devices slave). Osserviamo infine che SDA viene commutato da basso ad alto quando SCL è alto.

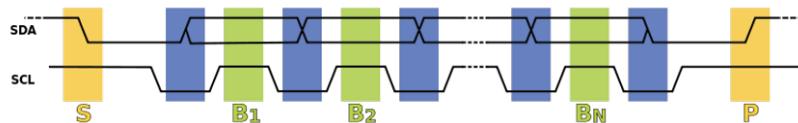


Figura 6.10: i2c bus - tempificazioni

Il protocollo definisce tre tipi fondamentali di transazioni, ognuna delle quali inizia con uno START e finisce con uno STOP.

- messaggio singolo in cui un master scrive dati a uno slave;
- messaggio singolo in cui un master legge dati da uno slave;
- formato combinato, dove un master effettua almeno due lettura o scrittura a uno o più slave.

Dopo lo START, il master impone sul bus l'indirizzo dello slave con cui vuole comunicare. Indirizzi e dati per convenzione sono trasmessi dal bit più significativo. Nel caso di messaggio singolo, il master inizia lo scambio di informazioni inviando lo start bit seguito dall'indirizzo dello slave con cui vuole comunicare. Segue un ottavo bit che indica se vuole

trasferire informazioni allo slave o riceverne. Se lo slave indirizzato esiste, il master prende controllo della linea dati sul successivo impulso alto del SCL imponendo SLC LOW. Se il master desidera scrivere allo slave, allora invia ripetutamente un byte con lo slave che invia un bit ACK. Se il master desidera leggere dallo slave, allora riceve ripetutamente un byte dallo slave, e invia un bit ACK dopo ogni byte tranne l'ultimo. Poiché il bus è multimaster e in generale è condiviso, può capitare che il bus venga conteso. Il protocollo di risoluzione adottato da I2C è deterministico: ogni trasmettitore controlla il livello della linea dati (SDA) e lo confronta con i livelli che si aspetta; se non corrispondono, quel trasmettitore ha perso l'arbitrato e abbandona questa interazione del protocollo.

Una caratteristica fondamentale di I2C è il **Clock Stretching**: un dispositivo slave indirizzato può tenere la linea di cloack SCL bassa dove aver ricevuto o inviato un byte, indicando che non è ancora pronto ad elaborare altri dati. Il master che sta comunicando con lo slave non può finire la trasmissione del bit corrente, ma deve aspettare che la linea di clock vada effettivamente alta. Se lo slave è in clock-stretching, la linea di clock sarà ancora bassa perché le connessioni sono *open drain*: questo significa che nessuno può forzare la linea alta, ma può solo forzarla bassa.

# Capitolo 7

## RISC-V

### 7.1 ABI e Istruzioni

Il RISC-V si compone di 32 registri general purpose (x0-x31) di lunghezza differente a seconda della versione (RV32 o RV64). La **Application to Binary Interface** (ABI) è lo strumento utilizzato per definire lo scopo di ciascun registro, assegnando a ognuno di questi un alias [7.1]. Una tale suddivisione standardizzata aiuta il programmatore a scrivere codice più leggibile e strutturato: sapendo quali registri usare per specifiche funzioni, il codice risulta più prevedibile e manutenibile. Il registro x0 è di sola lettura e

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6-7	t1-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments/return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporaries
f0-7	ft0-7	FP temporaries
f8-9	fs0-1	FP saved registers

Figura 7.1: Nome ABI di alcuni dei registri più importanti.

contiene sempre valore zero, ed è utile nelle istruzioni che necessitano di un valore zero senza utilizzare un immediato. Il **formato delle istruzioni** è basato su lunghezza fissa pari a 32 bit (anche per la versione RV64). In particolare, il RISC-V possiede soltanto 47 istruzioni, tutto ciò che non rientra in queste viene gestito con delle estensioni ISA. Ciascuna istruzione può appartenere a uno dei seguenti tipi:

- **R (Register)**: A questa categoria appartengono le istruzioni aritmetiche su registri. Nell'istruzione vengono inclusi gli operandi (registri) sorgente e destinazione.
- **I (Immediate and Load)**: A questa categoria appartengono le istruzioni aritmetiche con immediati e di load. Nell'istruzione vengono specificati gli operandi (registri) sorgente e destinazione, e l'immediato su 12 bit. Nel caso di *load*, viene utilizzato un immediato come offset per accedere a un area di memoria.

- **S (Store)**: A questa categoria appartengono le istruzioni di *store*, in cui la destinazione non è un registro ma una locazione di memoria (operando implicito).
- **U (Upper immediate)**: A questa categoria appartengono solo due istruzioni, che richiedono uno spazio maggiore per codificare un immediato di grandi dimensioni. Queste sono *load upper immediate* (lui) e *add upper immediate to program counter* (auipc).
- **J (Jump)**: A questa categoria appartengono le istruzioni di salto incondizionato con un offset con segno. Le due istruzioni possibili sono *jump and link* (jal) e *jump and link register* (jalr).
- **B (Branch)**: A questa categoria appartengono le istruzioni di salto condizionato. In particolare, l'offset utilizzabile viene codificato su 12 bit, per cui il salto è limitato a  $\pm 2^{12}$  half-words (2 bytes), ovvero un range di  $\pm 4$  KiB. Il salto avviene se la condizione da verificare assume valore true.

A seconda del tipo, la struttura di ciascuna istruzione cambia, ovvero varia il modo in cui i 32 bit nella stringa vengono interpretati. L'unica cosa che resta invariata sono i bit relativi al campo *opcode*, che vanno da 0 a 6. Notiamo come in alcune istruzioni sia presente un campo *function* oltre a quello di *opcode*, il motivo di una tale scelta è semplificare la decodifica delle istruzioni. L'*opcode* individua una generica categoria di istruzioni (ad esempio, operazioni aritmetiche), mentre *function* specifica l'operazione esatta da eseguire (add, multiply, eccetera). Gli operandi sono inclusi nelle istruzioni. L'architettura RISC-V appartiene chiaramente alla categoria di *load and store* (essendo RISC), per cui non sono ammesse operazioni direttamente in memoria e non servono altre istruzioni per caricare gli operandi.

### 7.1.1 Jump and Link e Jump and Link Register

L'istruzione **jal** permette di effettuare salti incondizionati con un offset con segno. L'offset viene codificato come un immediato su 21 bit, per cui il range del salto è limitato a  $\pm 2^{20}$  half-words (2 bytes), ovvero  $\pm 1$  MiB. Può essere utile per implementare statement di tipo *goto* o chiamate a funzioni.

$$pc = pc + \text{signed}(imm[20 : 0])$$

L'istruzione **jalr** permette anch'essa salti incondizionati con offset con segno, questa volta non incrementando il PC con l'offset, ma sommando l'offset a una locazione di memoria conservata in un registro. Questo permette dei salti di dimensione pari a  $\pm 2^{32}$ , ovvero  $\pm 4$  GiB. Risulta utile per implementare ritorni da chiamate a funzioni.

$$pc = rs1 + \text{signed}(imm[11 : 0])$$

### 7.1.2 Cheatsheet

Riportiamo un **cheatsheet** contenente le principali istruzioni, utili a implementare programmi di base, delle 47 disponibili nell'ISA base del RISC-V (RV32I).

- Aritmetica (tipo R e I)
  - $add\ rd, rs1, rs2 \rightarrow rd = rs1 + rs2$
  - $addi\ rd, rs1, imm \rightarrow rd = rs1 + imm$
  - $sub\ rd, rs1, rs2 \rightarrow rd = rs1 - rs2$
  - $and\ rd, rs1, rs2 \rightarrow rd = rs1 \& rs2$  (anche con immediato)
  - $or\ rd, rs1, rs2 \rightarrow rd = rs1 | rs2$  (anche con immediato)
- Confronti
  - $slt\ rd, rs1, rs2 \rightarrow$  setta  $rd$  a 1 se  $rs1 < rs2$ , altrimenti a 0
  - $slti\ rd, rs1, imm \rightarrow$  setta  $rd$  a 1 se  $rs1 < imm$ , altrimenti a 0
- Accesso alla memoria (load/store)
  - $lw\ rd, offset(rs1) \rightarrow$  load della word dalla locazione  $(rs1 + offset)$  in  $rd$
  - $lh\ rd, offset(rs1) \rightarrow$  load della half-word dalla locazione  $(rs1 + offset)$  in  $rd$
  - $lb\ rd, offset(rs1) \rightarrow$  load del byte dalla locazione  $(rs1 + offset)$  in  $rd$
  - $sw\ rs2, offset(rs1) \rightarrow$  store della word  $rs2$  nella locazione  $(rs1 + offset)$
  - $sh\ rs2, offset(rs1) \rightarrow$  store della half-word  $rs2$  nella locazione  $(rs1 + offset)$
  - $sb\ rs2, offset(rs1) \rightarrow$  store del byte  $rs2$  nella locazione  $(rs1 + offset)$
- Salti condizionati (branch)
  - $beq\ rs1, rs2, label \rightarrow$  salta a  $label$  se  $rs1 = rs2$
  - $bne\ rs1, rs2, label \rightarrow$  salta a  $label$  se  $rs1 \neq rs2$
  - $blt\ rs1, rs2, label \rightarrow$  salta a  $label$  se  $rs1 < rs2$
  - $bge\ rs1, rs2, label \rightarrow$  salta a  $label$  se  $rs1 \geq rs2$
- Salti e chiamate
  - $jal\ rd, offset \rightarrow$  salta all'indirizzo  $PC + offset$  e salva l'indirizzo di ritorno in  $rd$  (solitamente  $ra$  dell'ABI)
  - $jalr\ rd, rs1, offset \rightarrow$  salta all'indirizzo  $(rs1 + offset) \& 1$  e salva l'indirizzo di ritorno in  $rd$  (solitamente  $ra$  dell'ABI)
  - $ecall \rightarrow$  chiamata a sistema (system call)

Riportiamo l'esercizio proposto a lezione, che richiede di copiare un vettore in memoria in un altro. Si nota l'uso della pseudo-istruzione *la* (load address), messa a disposizione dal compilatore per caricare un indirizzo. Si rimanda al paragrafo delle pseudo-istruzioni per ulteriori dettagli [??].

```

1 main:
2     la    a0, src      # carica indirizzo sorgente in a0
3     la    a1, dst      # carica indirizzo destinazione in a1
4     la    t0, len      # carica indirizzo di len in t0
5     lw    a2, 0(t0)    # carica valore di len in a2

```

```

6    jal  ra, copia      # chiama funzione copia, salva ritorno in ra

7
8 copia:                  # funzione per copiare
9    beq  a2, x0, fine   # se lenght == 0, termina
10   ciclo:
11     lw    t0, 0(a0)    # carica elemento da src
12     sw    t0, 0(a1)    # scrive elemento in dst
13     addi a0, a0, 4     # src ++
14     addi a1, a1, 4     # dst ++
15     addi a2, a2, -1    # length --
16     bne  a2, x0, ciclo # se length != 0 ripeti
17   fine:
18     jalr x0, ra, 0      # ritorna al chiamante (main)

```

## 7.2 Livelli di privilegio e Interruzioni

Il RISC-V supporta tre livelli di **privilegio**: *User*, *Supervisor* e *Machine* [7.2]. Un

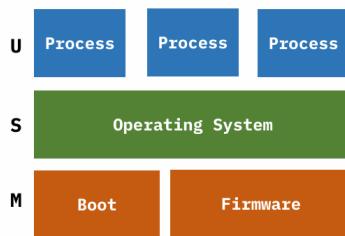


Figura 7.2: Livelli di privilegio del RISC-V.

insieme di registri, noto come **Control and Status Registers** (CSRs), ha il compito di rappresentare lo stato dell’architettura (stato della CPU, interruzioni, eccezioni, eccetera). Questi sono localizzati in uno spazio di indirizzamento privato e possono essere acceduti solo tramite istruzioni speciali:

- Istruzioni di *read* e *write*.
- Istruzioni di *set* e *clear*.
- Varianti delle istruzioni con registri e immediati.

Per quanto riguarda la **gestione delle interruzioni ed eccezioni**, questa avviene tramite i CSRs in due fasi:

- **Fase offline:** Avviene durante il boot del sistema e viene eseguito del codice speciale che:
  1. Abilita un bit speciale per le interruzioni. In particolare, il bit *MIE* nel CSR *mstatus*, che si occupa dello stato della CPU.
  2. Abilita tutti gli specifici interrupt interessati tramite i bit di un secondo registro. In altre parole, si smascherano i bit del CSR *mie* corrispondenti alle linee di interruzione che interessano.

- **Fase online:** Accade quando effettivamente si verifica un'interruzione o un'eccezione.
  1. La CPU salva il PC in un apposito registro, il *mepc*.
  2. La CPU salta all'indirizzo nel registro *mtvec*, il quale contiene l'indirizzo della tabella dei vettori.
  3. La CPU salta al gestore dell'interrupt (handler). Il modo in cui viene prelevato l'indirizzo dipende da se le interruzioni sono vettorizzate, specificato nel registro *mcause*.
  4. Il gestore termina con l'istruzione *mret*, che permette di ritornare dal trap ripristinando il valore del PC da *mepc*.

RISC-V introduce delle istruzioni privilegiate chiamate **system instructions**, usate da sistemi operativi e ambienti virtualizzati per gestire system calls e sincronizzazione. Possono inoltre essere utilizzate per il debugging, ma richiedono un modulo esterno per funzionare.

Il **Supervisor Binary Interface** (SBI) è un'interfaccia standard per permettere al software a livello supervisor (S-mode) di funzionare su hardware diversi, mantenendolo portatile. OpenSBI è la piattaforma open-source più usata per implementare SBI.

## 7.3 Estensioni

Una delle più importanti **estensioni** del RISC-V è *RV64*, nella quale registri e CSRs hanno una dimensione di 64 bit, anche se le istruzioni sono sempre di lunghezza pari a 32 bit. Sono incluse delle specifiche istruzioni per manipolare dati a 32 bit, utili per compatibilità o efficienza.

Il nucleo base di RISC-V (RV32I) ha solo 47 istruzioni, per cui è molto semplice e minimale. Per questo motivo, l'architettura è estendibile e modulare, in modo da adattarla alle esigenze del sistema. Si può estendere aggiungendo nuove funzionalità tramite **estensioni**, ciascuna con un proprio *opcode*. Le implementazioni RISC-V vengono identificate dal nome delle estensioni supportate, ad esempio: RV32IA = RV32I (base) + A (Atomic Instructions).

Un possibile esempio è RV32E, una variante semplificata di RV32I pensata per micro-controllori o sistemi con pochissime risorse. Riduce i registri da 32 a solo 16, occupando meno spazio sull'hardware. L'estensione di tipo C, invece, utilizza istruzioni compresse da 16 bit (anziché 32), permettendo di ridurre la dimensione del codice in memoria.

## 7.4 Pseudo-istruzioni

Le **pseudo-istruzioni** sono delle scorciatoie per il programmatore che permettono di scrivere codice più leggibile e facile da scrivere, ma che non appartengono all'ABI. In altre parole, non esiste una corrispondenza con il codice macchina, è compito dell'assemblatore convertirle in istruzioni comprensibili all'hardware.

Supponiamo ad esempio di voler fare la *load* di un immediato molto grande, più dei 12 bit normalmente disponibili, in un registro. Usando la pseudo-istruzione *li* è possibile scaricare questo compito complesso all’assemblatore, che scomporrà la *load* in tante *load* e *add* diverse, incrementando il valore con diverse operazioni.

## 7.5 Calling convention

La **Calling Convention** definisce le regole che governano come le funzioni si chiamano tra loro e gestiscono parametri e valori di ritorno. Tipicamente, il processo avviene attraverso alcune fasi alternate tra chiamante e funzione chiamata:

- Caller (chi chiama):
  1. Salva il proprio contesto sullo stack, includendo i registri.
  2. Salva l’indirizzo di ritorno in a0-a7 e se necessario pusha eventuali parametri sullo stack.
  3. Utilizza l’istruzione *jal* per saltare alla procedura, salvando l’indirizzo di ritorno in *ra*.
- Callee (funzione chiamata):
  1. Preleva gli eventuali parametri dallo stack.
  2. Effettua le operazioni.
  3. Memorizza il risultato da ritornare in a0-a1.
  4. Utilizza la pseudo-istruzione *ret* per ritornare al chiamante.
- Caller (chi chiama):
  1. Ripristina il contesto salvato precedentemente sullo stack.
  2. Legge il valore ritornato da a0-a1.

### 7.5.1 Stack

La convenzione adottata dal RISC-V è che lo **stack** cresca dall’alto verso il basso in termini di indirizzi. Per gestire lo stack frame sono presenti due CSRs:

- **Stack Pointer (sp)**: Punta alla base del corrente stack frame.
- **Frame Pointer (fp)**: (Normalmente) punta alla fine del precedente stack frame. Normalmente non viene utilizzato, in quanto non è obbligatorio dell’implementazione del RISC-V. Tuttavia esistono diversi vantaggi nel suo impiego, ad esempio l’indirizzo di ritorno ha un offset fissato dal *fp* corrente.

Ricordiamo che lo stack frame è una porzione di memoria nello stack che viene allocata ogni volta che una funzione viene chiamata [7.3]. Serve a gestire le variabili locali, parametri, indirizzo di ritorno, e registri salvati. È fondamentale notare che l’uso esatto del registro *fp* non è fisso e può cambiare in base al compilatore usato e alle opzioni di compilazione. Nel RISC-V, il frame pointer è solitamente s0, ma non obbligatoriamente.

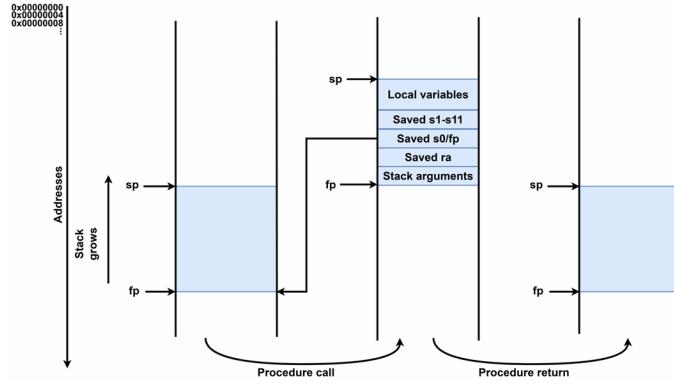


Figura 7.3: Esempio di stack frame nel RISC-V.

## 7.6 Confronto con il MIPS

L'ISA del MIPS presenta molte similitudini con quella del RISC-V, poiché entrambe seguono una stessa filosofia di design.

- **Istruzioni:** Entrambe hanno istruzioni a lunghezza fissa di 32 bit.
- **Registri:** Entrambe hanno 32 registri general purpose.
- **Supporto 64-bit:** Entrambe supportano versioni a 64 bit (RV64 e MIPS-64).
- **Architettura:** Entrambe sono di tipo load and store.
- **Salvi condizionati:** Mentre il RISC-V offre diverse modalità di comparazione (beq, bne, blt, bge, eccetera), il MIS ne possiede solo due (beq e bne).

Inoltre, i mnemonici delle istruzioni sono perlopiù identici, tranne qualche piccola eccezione.

Si consiglia una lettura del documento "Domande RISC-V" presente sul team, in quanto ci sono molte domande (con le rispettive risposte) *particolari* per comprendere meglio alcuni dettagli su questa architettura.



# Capitolo 8

## Cloud-Edge

### 8.1 Cloud Computing

Il **Cloud Computing** è un insieme di tecnologie che permettono di compiere operazioni di elaborazione, networking e archiviazione di dati utilizzando risorse hardware/software distribuite (tipicamente remote) e virtualizzate. Le principali caratteristiche che contraddistinguono questa tecnologia sono:

- **On-demand self-service:** Un customer può unilateralmente rifornirsi di capacità di calcolo senza interazione umana.
- **Broad network access:** Le risorse sono disponibili in rete e sono accessibili tramite meccanismi standard.
- **Resource pooling:** Le risorse fornite dal provider sono distribuite dinamicamente a più utenti, realizzando un modello multi-tenant.
- **Rapid elasticity:** Le risorse possono essere fornite e rilasciate in modo elastico, e in alcuni casi automatico, così da scalare rapidamente in accordo con la richiesta.
- **Measured service:** Il sistema cloud controlla ed ottimizza l'uso delle risorse, monitorandolo.

Nel modello classico di gestione dei servizi IT (on-premise), un'azienda acquisisce e gestisce in-house le soluzioni hardware e software necessarie al proprio business. Questo significa che l'azienda è proprietaria dell'hardware e delle licenze del software che utilizza, e che il personale IT (interno/esterno all'azienda) si occupa della loro completa gestione. Nel modello Cloud invece, le risorse sono ospitate all'interno dell'infrastruttura di proprietà del provider, e le aziende possono accedervi in qualsiasi momento attraverso piattaforme applicative accessibili via web. I servizi Cloud possono essere erogati secondo diversi modelli di servizio e di fornitura, in modo che un'azienda possa scegliere in base alle proprie esigenze.

#### 8.1.1 Architettura di un servizio Cloud

L'**architettura di un generico servizio Cloud** viene definita dal NIST attraverso la figura che segue [8.1], definisce cosa ciascun servizio deve fare e non come debba essere implementato. Si individuano 5 *attori* principali, ossia entità (sistema o persona) che partecipa ad una transazione o processo del sistema Cloud.

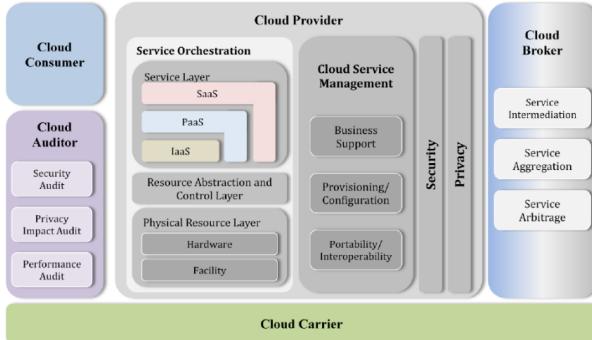


Figura 8.1: Modello NIST per il Cloud Computing.

- **Cloud consumer:** Utilizza servizi offerti da Cloud providers.
- **Cloud provider:** È responsabile di rendere servizi disponibili a parti interessate.
- **Cloud auditor:** Conduce una valutazione indipendente dei servizi cloud, delle operazioni del sistema informativo, delle prestazioni e della sicurezza del sistema Cloud.
- **Cloud broker:** Gestisce l'uso, le performance e la distribuzione dei servizi cloud; negozia le relazioni tra provider e consumer.
- **Cloud carrier:** Intermediario che fornisce connettività e trasporto dei servizi cloud, da provider e consumer.

### 8.1.2 Modelli di servizio

I **modelli di servizio** riguardano che tipo di risorsa IT il provider fornisce. Le tre tipologie principali sono le seguenti.

- **SaaS (Software-as-a-service):** Acquisizione di intere applicazioni software in esecuzione su infrastrutture Cloud e accessibili da remoto on-demand. In questo caso è il provider a gestire l'aggiornamento, la manutenzione, la scalabilità e le responsabilità in termini di sicurezza. Dunque, il customer non deve affrontare spese riguardanti queste gestioni. Si specifica che il software è di possesso del provider, il customer paga quindi per poter mantenere il servizio attivo e raggiungibile da altri utenti.
- **IaaS (Infrastructure-as-a-service):** Acquisizione di una infrastruttura virtuale al di sopra delle quali il customer può installare ed eseguire software arbitrario. L'utente non può gestire l'infrastruttura sottostante, ma ha il controllo su sistema operativo, storage, applicazioni e un controllo limitato su alcuni componenti di rete, come i firewall. Quelle che sono messe a disposizione sono quindi sono features hardware, per tale motivo all'IaaS ci si riferisce spesso come HaaS (Hardware as a Service).
- **PaaS (Platform-as-a-service):** Acquisizione di ambienti e tool web-based per lo sviluppo, il testing, la messa in esercizio, l'hosting e la manutenzione delle proprie applicazioni.

### 8.1.3 Modelli di fornitura

I **modelli di fornitura** riguardano dove e da chi è ospitato il Cloud. Si individuano tre tipologie principali.

- **Cloud pubblico:** Il servizio è fornito da un provider terzo (es. AWS), quindi le risorse sono condivise tra più clienti (multi-tenant). La scalabilità è alta e i costi sono contenuti, ma c'è meno controllo diretto.
- **Cloud privato:** Il servizio è dedicato a un'unica organizzazione, quindi può essere gestito internamente o da un provider esterno. C'è più controllo e sicurezza, ma i costi di gestione sono più alti.
- **Cloud ibrido:** È una combinazione di pubblico e privato. Le risorse possono muoversi tra i due, ad esempio si tengono dati sensibili nel cloud privato e quelli meno critici nel pubblico.

### 8.1.4 Rischi e Sicurezza

In generale, affidandosi al Cloud si perde (totalmente o parzialmente, a seconda del modello di servizio e di fornitura scelto) il controllo dei propri dati e sistemi, e ci si espone a dei **rischi di sicurezza** che dipendono sia dalle misure di sicurezza implementate dall'utente nelle proprie applicazioni che dalle politiche di protezione implementate dal fornitore del servizio Cloud (modello di responsabilità condivisa). Oltre ai classici requisiti di sicurezza, esistono poi dei vincoli minimi in termini di sicurezza ed affidabilità richiesti dalle norme vigenti nazionali ed europee, che non tutti i servizi Cloud disponibili potrebbero essere in grado di rispettare.

ENISA (European Network and Information Security Agency) riporta i principali rischi di sicurezza del Cloud, i quali andrebbero tenuti d'occhio e cercati di mitigare per quanto possibile.

## 8.2 IoT

**Internet of Things** (IoT) viene generalmente definito come:

*"Un'infrastruttura globale per la società dell'informazione, che abilita servizi avanzati interconnettendo cose (fisiche e virtuali) basate su informazioni e comunicazioni interoperabili, esistenti e in evoluzione tecnologie".*

I dispositivi coinvolti in un Sistema IoT possono essere anche molto eterogenei in quanto a capacità di elaborazione e memorizzazione, connettività e alimentazione. Alcuni tra i campi di applicazione sono: smart factory, smart home, smart city, e-Health (monitoraggio dei pazienti a distanza) e smart energy.

### 8.2.1 Architettura di un sistema IoT

Un sistema IoT si sviluppa solitamente tramite un'**architettura a 3 livelli**:

- **Livello dei dispositivi:** Comprende sensori, attuatori, e sistemi embedded in grado di generare tipicamente grosse moli di dati.
- **Livello di connettività:** Include i protocolli che consentono ai dispositivi di comunicare con il livello dei servizi e applicazioni.
- **Livello dei servizi e applicazioni:** Solitamente include servizi Cloud che si occupano di elaborare i dati generati dai dispositivi.

### 8.2.2 Sicurezza

I dispositivi IoT sono **insicuri per natura**: sono connessi e quindi potenzialmente accessibili e spesso non possiedono la capacità computazionale necessaria per proteggersi adeguatamente (ad esempio, attraverso la cifratura dei dati scambiati o l'utilizzo di meccanismi di autenticazione).

Data la grandissima diffusione di dispositivi IoT in svariati contesti, essi sono particolarmente soggetti ad attacchi di tipo ransomware, denial of service, furto di identità e di dati, man-in-the-middle e altri, che possono avere effetti estremamente negativi non solo in termini di security in sè (confidenzialità, integrità e disponibilità), ma anche sulla privacy e sulla safety.

Per garantire adeguata protezione, è necessario prevedere meccanismi di sicurezza che si adattino alle capacità dei singoli dispositivi (ad esempio, meccanismi di cifratura lightweight basati su architetture hardware a basso impatto) e che siano in grado di assicurare, fra gli altri, autenticazione, controllo degli accessi e sicurezza dei canali di comunicazione.

## 8.3 Edge Computing

Il termine **Edge Computing** significa letteralmente "*elaborazione ai margini*", ovvero elaborazione delle informazioni in prossimità della relativa sorgente. L'Edge Computing nasce per rispondere all'esigenza di ridurre la latenza e il consumo di banda che caratterizzano tipicamente i sistemi IoT classici, in cui grossi moli di dati vengono inviati ai servizi Cloud per elaborazioni successive. Il paradigma Edge, inoltre, consente di ridurre i rischi di sicurezza legati alla delocalizzazione dei dati, che potrebbero avere in taluni casi requisiti stringenti di sicurezza/privacy.

### 8.3.1 Architettura

I componenti principali nell'**architettura di un sistema di Edge Computing** sono:

- **Edge devices:** Dispositivi edge e IoT attrezzati per raccogliere dati ed eseguire elaborazioni in locale.
- **Edge server o gateway:** Server perimetrali utilizzati per distribuire le applicazioni sui dispositivi. Sono in costante comunicazione con i dispositivi utilizzando degli agenti installati su ciascuno di questi.
- **Edge network o micro data center:** Tecnologie di rete avanzate. Possono essere immaginate come un cloud locale con cui i dispositivi possono comunicare.

- **Enterprise Hybrid Cloud:** Si occupa dell'archiviazione e la gestione dei dati a livello aziendale (analisi complesse e dashboard).

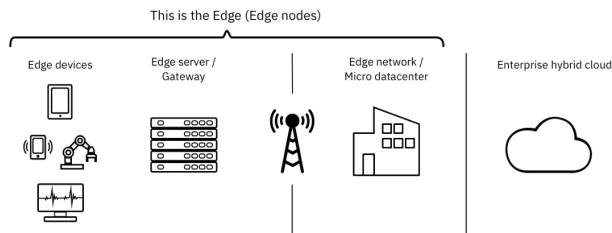


Figura 8.2: Architettura di un sistema di Edge Computing.

## 8.4 Fog Computing (cenni)

Per **Fog Computing** si intende un paradigma per l'elaborazione dati che calca alcuni dei principi del paradigma di Cloud Computing, rispondendo tuttavia ad alcune limitazioni che quest'ultimo presenta. Nonostante non esiste una definizione univoca per questo paradigma, analizzando le diverse definizioni attribuite è possibile osservare una serie di punti in comune che ne definiscono le proprietà principali:

- **Decentralizzazione dei servizi:** Offre dei servizi dislocati su nodi fisicamente disaccoppiati e geograficamente distribuiti.
- **Architettura gerarchica:** L'organizzazione avviene per livelli, ciascuno dei quali ha un ruolo specifico (raccogliere, elaborare e inoltrare i dati).

Per quanto detto le similitudini con il classico Edge Computing sono molte, motivo per cui i due sono spesso confusi, tuttavia sono due le cose che contraddistinguono il Fog Computing:

- In generale, i Fog Node sono tipicamente schierati nelle posizioni intermedie fra i nodi Edge e i servizi Cloud.
- A differenza degli Edge Node, le capacità di calcolo, immagazzinamento dati e networking dei Fog Node sono nettamente superiori.

## 8.5 Virtualizzazione

La **virtualizzazione** è una tecnologia che consente di creare una versione virtuale (cioè simulata via software) di qualcosa che normalmente è fisico o reale, come un sistema operativo o un computer.

Che sia eseguita in hardware, software o incorporata in sottosistemi, la virtualizzazione viene sempre ottenuta utilizzando e combinando tre semplici tecniche [8.3].

- Il **multiplexing** espone una risorsa tra più entità virtuali. Ce ne sono due tipologie, nello spazio e nel tempo. Con il multiplexing *nello spazio*, la risorsa fisica viene

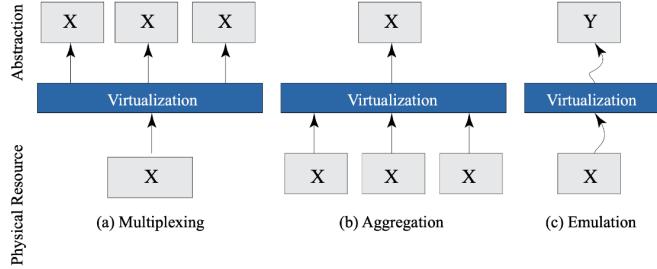


Figura 8.3: Le tre tecniche base di virtualizzazione.

partizionata in entità virtuali. Ad esempio, il sistema operativo multiplexa diverse pagine di memoria fisica su diversi spazi di indirizzi. Con il multiplexing *temporale*, la stessa risorsa fisica viene programmata temporalmente tra entità virtuali. Ad esempio, lo scheduler del sistema operativo suddivide i core della CPU tra l'insieme di processi eseguibili.

- **L'aggregazione** fa il contrario, raggruppa più risorse fisiche e le fa apparire come un'unica entità. Ad esempio, un controller RAID aggrega più dischi in un unico volume. Una volta configurata, l'interfaccia delle risorse fisiche deve apparire come unica, per cui l'utilizzatore è del tutto ignaro del processo di aggregazione.
- Infine, con l'**emulazione**, invece di accedere direttamente all'hardware reale, il programma accede a un "livello software" che finge di essere un certo hardware. L'emulatore crea un finto dispositivo che si comporta come uno vero, anche se quel dispositivo non esiste fisicamente nel computer. Ad esempio, gli emulatori tra architetture eseguono un'architettura del processore su un'altra.

Le tre tecniche possono essere combinate per formare uno stack di esecuzione completo.

### 8.5.1 Macchine virtuali

Una **macchina virtuale** è un ambiente di elaborazione completo con capacità di elaborazione, memoria e canali di comunicazione isolati. È possibile fare la seguente classificazione.

- **Macchine virtuali basate sul linguaggio**, come *Java Virtual Machine*.
- **Macchine virtuali leggere**, che si basano su una combinazione di meccanismi di isolamento hardware e software per garantire che le applicazioni in esecuzione direttamente sul processore siano isolate in modo sicuro da altre e dal sistema operativo sottostante. Un esempio sono i *container*.
- **Macchine virtuali a livello di sistema**, in cui viene emulato l'intero hardware di un calcolatore, permettendo di installare un sistema operativo intero. Le piattaforme che permettono di eseguire questa tipologia di macchine virtuali sono:
  - *Hypervisor*: Si basa sull'esecuzione diretta sulla CPU per la massima efficienza.
  - *Simulatore di macchina*: Implementato come una normale applicazione a livello utente.

## 8.5.2 Hypervisor

Un **hypervisor** (o Virtual Machine Monitor - VMM) è un software di sistema che permette di creare ed eseguire macchine virtuali, con l'obiettivo di ridurre al minimo i costi di esecuzione. Quando più VM coesistono simultaneamente sullo stesso sistema di computer, l'hypervisor multiplexa (cioè alloca e schedula) le risorse fisiche in modo appropriato tra le macchine virtuali. Le architetture dell'hypervisor possono essere classificate come segue [8.4].

- **Tipo 1 (bare-metal)**: Il VMM viene eseguito direttamente sull'hardware macchina host (bare), senza necessità di un sistema operativo.
- **Tipo 2 (hosted)**: Il VMM viene eseguito su un host esteso, sotto il sistema operativo host.

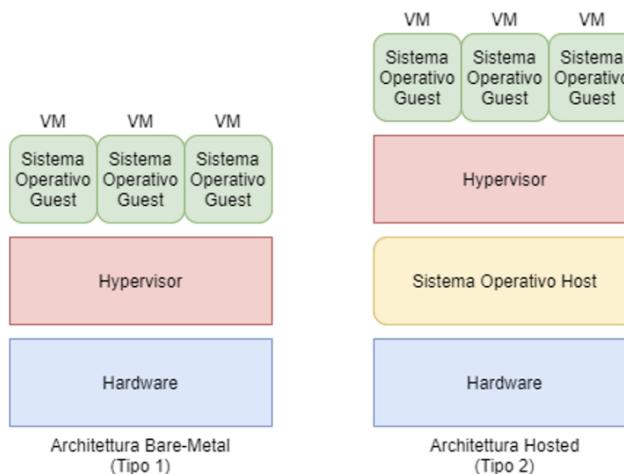


Figura 8.4: Tipologie di hypervisor.

In entrambi i tipi, il VMM crea la macchina virtuale. Tuttavia, in un ambiente di tipo 1, il VMM sulla macchina bare deve eseguire anche lo scheduling del sistema e l'allocazione (reale) delle risorse. Di conseguenza, il VMM di tipo 1 può dover includere tale codice, non specificamente necessario per la virtualizzazione. In un sistema di tipo 2, le funzioni di allocazione delle risorse e creazione dell'ambiente per la macchina virtuale sono eseguiti dal sistema operativo host.

Osserviamo come l'enfasi sulla differenza tra i due sia sull'allocazione delle risorse e non sul fatto che l'hypervisor venga eseguito in modalità privilegiata o meno. In particolare, un hypervisor può essere di tipo 2 anche quando viene eseguito in modalità kernel, ad esempio *VMware Workstation* funziona in questo modo.

Nella sua forma più elementare, un hypervisor utilizza due delle tre tecniche di virtualizzazione: multiplexa (nello spazio e possibilmente nel tempo) il processore fisico ed emula tutto il resto, in particolare i dispositivi I/O. Questa combinazione di tecniche è necessaria e sufficiente nella pratica per raggiungere i criteri di efficienza. Il motivo è che se non si multiplexa bene la CPU e la MMU (Memory Management Unit), l'hypervisor sarebbe costretto a simulare tutto, rendendo le VM molto lente.

In modo molto simile a come la scheduler gestisce l'esecuzione di diversi processi sullo stesso processore, anche l'hypervisor deve decidere quale VM può utilizzare la macchina. Dunque, l'hypervisor carica nel processore i registri della VM attiva, le sue tabelle di memoria e la lascia eseguire. Ovviamente, l'hypervisor è anche responsabile di garantire la proprietà di sicurezza della macchina virtuale. Assicura quindi che la CPU virtuale venga sempre eseguita con privilegi ridotti, ad esempio non facendogli eseguire istruzioni privilegiate.

Per comprendere come avvenga la **gestione delle istruzioni privilegiate**, è necessario inanzitutto distinguere:

- **Dynamic Binary Translation (DBT)**: L'intero codice della macchina virtuale viene eseguito normalmente, ma se viene richiamata qualche istruzione privilegiata, l'hypervisor la intercetta e tenta di tradurla in una equivalente e sicura. L'istruzione tradotta viene eseguita sul vero processore. Inoltre, questa viene salvata in una cache, così che il processore possa eseguirla nuovamente in futuro senza ritradurla.
- **Full-virtualization**: La macchina virtuale crede di essere su un vero computer fisico, e non sa di essere "virtuale". In questo caso l'hypervisor emula tutto l'hardware e il sistema operativo guest non deve essere modificato. Se prova ad eseguire istruzioni privilegiate, scatta una *trap* e l'hypervisor interviene (trap-and-emulate). Il vantaggio è la possibilità di installare qualsiasi sistema operativo senza modificarlo, mentre lo svantaggio è l'overhead dovuto all'intercettazione delle istruzioni privilegiate.
- **Para-virtualization**: Il sistema operativo sa che sta girando in una macchina virtuale e collabora con l'hypervisor. In questo caso, il sistema operativo guest viene modificato per chiamare direttamente l'hypervisor quando deve fare operazioni privilegiate, per cui non si usano trap, ma chiamate specifiche dette *hypercalls*. Il vantaggio è la rapidità, mentre lo svantaggio è l'uso di soli i sistemi operativi modificati per supportare questo tipo di virtualizzazione.

Osserviamo con qualche dettaglio in più il processo della *trap and emulate*.

1. Il sistema operativo guest esegue codice normalmente sul processore reale.
2. Il processore incontra un'istruzione privilegiata, come un cambio di modalità di esecuzione (da utente a root).
3. Il processore blocca l'istruzione e genera una *trap*. Il controllo passa quindi al supervisore (che gira a un livello più alto, o con più privilegi, del guest).
4. L'hypervisor simula via software il comportamento dell'istruzione.
5. Dopo l'emulazione, l'esecuzione torna al sistema operativo guest.

### 8.5.3 Architetture Intel x86-64

Le CPU Intel definiscono 4 **privilegi di esecuzione**, anche detti *ring* [8.6]. Il primo livello (ring 0) è il privilegio più alto, mentre il quarto (ring 3) è il più basso. Ciascuna istruzione deve essere gestita secondo privilegi di esecuzione adeguati. Ad esempio, il

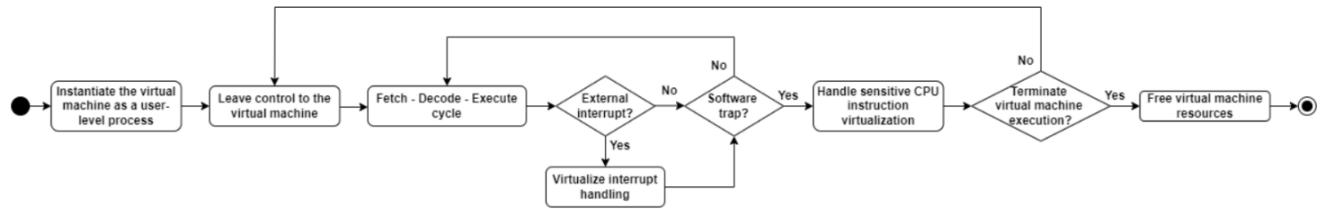


Figura 8.5: Flusso della tecnica trap and emulate.

codice utente viene eseguito sul ring 3, mentre le funzioni del kernel (come ISR o system calls) devono essere eseguite su ring più bassi. In generale, l'esecuzione di istruzioni su ring non adatti porta alla generazione di trap o effetti inattesi. Nelle CPU Intel, gli

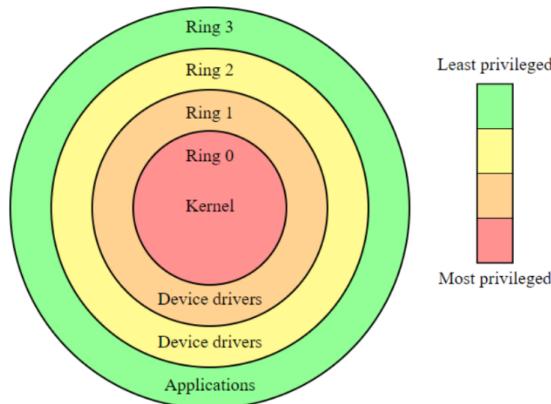


Figura 8.6: Privilegi di esecuzione nelle CPU Intel.

eventuali sistemi operativi guest eseguono nei ring 1-3. In caso di invocazione di istruzioni privilegiate, il controllo viene passato all'hypervisor, che esegue nel ring 0 e le gestisce con una tecnica di emulazione del tipo DBT [8.7]. Spostiamo ora la nostra attenzione

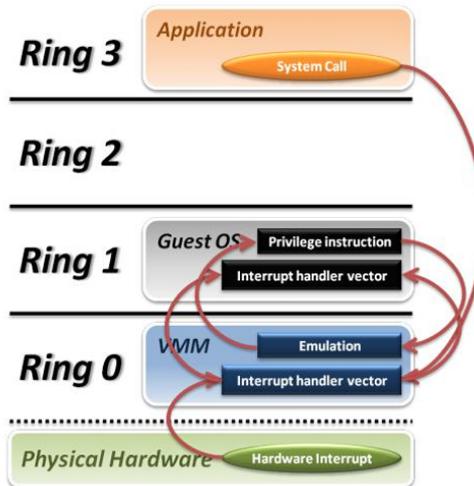


Figura 8.7: Gestione delle istruzioni privilegiate del guest os in Intel.

sulle **architetture Intel x86-64** e cerchiamo di capire se su queste sia possibile adottare

una tecnica del tipo *trap and emulate*. Innanzitutto, le CPU Intel x86-64 posseggono un ISA di istruzioni privilegiate che non sono in grado di generare traps via software quando eseguite nel livello utente (ring 1-3). Per tale motivo, teoricamente queste architetture non si dimostrano adatte per il meccanismo di trap and emulate. Per risolvere questa limitazione, sono stati introdotti vari supporti:

1. Estensione dell'ISA.
2. Registri che implementano strutture di controllo per gestire le macchine virtuali.
3. Meccanismi hardware per aggiornare i registri con le informazioni principali del cambio di contesto tra macchina virtuale e hypervisor.

**Intel VT-x** è l'estensione hardware che permette il supporto della virtualizzazione nelle architetture x86-64. Questa aggiunge al processore due modalità di esecuzione speciali, ciascuna delle quali ha sempre i classici 4 livelli di privilegio (rings).

- **VMX Root Operation:** Modalità in cui gira l'hypervisor (cioè il software che controlla le VM), ha il pieno controllo dell'hardware.
- **VMX Non-Root Operation:** Modalità in cui girano le macchine virtuali guest, sembrano avere il controllo dell'hardware ma in realtà sono sorvegliate dall'hypervisor.

Il cambio di modalità operativa avviene in automatico se la VM in modalità non-root tenta di eseguire un'istruzione privilegiata (il cambio potrebbe non avvenire se il sistema non è configurato per farlo). L'hardware mantiene due tipi di informazioni di stato (registri, flag, eccetera), la *guest-area* (per le macchine virtuali) e la *host-area* (per il sistema host). In particolare, la **Virtual Machine Control Structure** (VMCS) è una struttura speciale gestita dal processore che contiene tutte le informazioni necessarie per gestire il passaggio tra le due modalità e salvare/ripristinare lo stato del processore quando avviene il cambio di contesto. Il cambio di contesto tra macchina virtuale e macchina host (e viceversa) avviene mediante:

- **VM Entry:** Il processore passa dal sistema host alla macchina virtuale guest, per cui viene ripristinato lo stato presente nella guest-area.
- **VM Exit:** La VM cerca di eseguire un'istruzione che richiede l'intervento dell'hypervisor.

Con una tale gestione della virtualizzazione, il flusso di esecuzione delle macchine virtuali può procedere normalmente: non è richiesta alcuna re-implementazione delle primitive del guest-os. In caso di istruzioni privilegiate, viene avviata una VM Exit che permette di procedere emulando l'istruzione [??]. Le istruzioni eseguite in modalità VMX Non-Root che causano una VM Exit sono:

- *Istruzioni che causano VM Exit incondizionatamente*, ovvero istruzioni che fanno sempre scattare un VM Exit, indipendentemente dalla configurazione del sistema. Un esempio è *CPUID* che restituisce informazioni sul processore.
- *Istruzioni che causano VM Exit condizionatamente*, ovvero istruzioni possono causare un VM Exit, ma solo se configurato così nel controllo VMCS. L'hypervisor specifica quindi quali istruzioni rientrano in questa categoria.

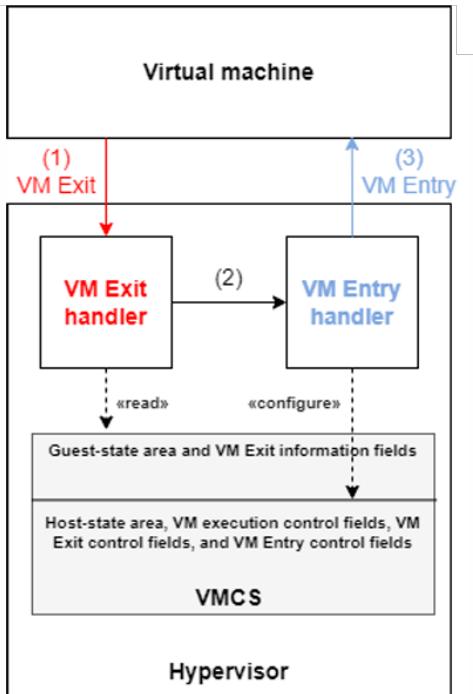


Figura 8.8: Flusso di esecuzione per istruzioni privilegiate della VM.

- *Eccezioni*, quando il guest genera un'eccezione.
- *Interrupt esterni*, dovuti a periferiche.
- *Non-Maskable Interrupts* (NMIs), tipicamente usate per condizioni critiche come errori hardware.

Anche nel caso di interruzioni esterne, il comportamento del sistema è molto simile. Queste infatti generano un VM Exit appositamente gestito dall'hypervisor, il quale:

- Se la macchina virtuale ha le interruzioni disabilitate, l'hypervisor avvia una VM Entry specificando che la VM Exit accadrà solo quando le interruzioni saranno abilitate.
- Se la macchina virtuale ha le interruzioni abilitate, l'hypervisor avvia una VM Entry trasferendo la richiesta di interruzione alla VM.

L'architettura Intel x86-64 supporta anche la **virtualizzazione della memoria** attraverso le due tecniche di indirizzamento lineare e paginazione. Con indirizzamento lineare si intende il processo di mappare gli indirizzi generati dalla CPU in uno spazio degli indirizzi virtuali per la memoria virtuale. Mentre la paginazione è ciò che consente di associare a ciascun indirizzo lineare al corrispondente indirizzo fisico [8.9]. Gli indirizzi



Figura 8.9: Processo di traduzione indirizzi da VM a sistema host.

fisici generati dalle macchine virtuali vengono tradotti in indirizzi fisici per la macchina host tramite un meccanismo noto come *Extended Page Table* (EPT), che fa tutto in hardware.

# Capitolo 9

## Esercitazioni

In questo capitolo saranno affrontate tutte le tematiche riguardanti le esercitazioni di maggiore interesse. Quindi saranno approfondite le sole esercitazioni in cui sono stati affrontati anche degli argomenti teorici importanti.

### 9.1 Debunk esercizi PIA

In questo paragrafo verrà commentato il codice diffuso sul canale Teams e presentato a lezione relativo alle esercitazioni sulla PIA (Marzo 2025).

#### 9.1.1 Esercizio 1

Il programma serve a provare la configurazione **communic asincrona** costituita da due sistemi simmetrici ciascuno con un processore M68000, una ROM di 8K (addr \$0-\$1FFF), una RAM di 10K (addr \$8000-\$A7FF), un device parallelo PIA mappato a \$2004, un device seriale di tipo TERMINAL mappato a \$2000. I due PIA sono interconnessi e mediante un protocollo di handshaking consentono ai due sistemi di scambiarsi i caratteri digitati sul dispositivo TERMINAL. I device interagiscono con i rispettivi processori mediante le linee di interruzione utilizzando un meccanismo di interrupt autovettorizzato (TERMINAL e PIA non supportano le int.vettorizzate). In particolare i dati immessi da tastiera sono acquisiti, alla pressione del tasto ENTER, mediante interruzione di livello 1, che corrisponde al vettore 25 mappato in area ROM alla locazione \$64: in tale locazione è contenuto l'indirizzo della ISR in RAM (\$8500). Nella ISR, il dato viene inviato alla sezione A del dispositivo parallelo PIA per la trasmissione verso il dispositivo PIA connesso all'altro sistema. La ricezione di un carattere sul dispositivo PIA e' gestita mediante interruzione di livello 3, che corrisponde al vettore 27 mappato in area ROM alla locazione \$6C: in tale locazione è contenuto l'indirizzo della ISR in RAM (\$8700). All'arrivo dell'interrupt la ISR acquisisce il dato e lo invia al terminal per la visualizzazione. Un'ulteriore ISR mappata sull'autovettore 26 gestisce le condizioni di buffer full sul TERMINAL. Tale ISR invia sulla PIA i 256 caratteri nel buffer. Nell'esercizio vedremo uno stesso programma caricato per entrambi i sistemi speculari, sfruttando il meccanismo delle interruzioni: infatti le interruzioni generate dai dispositivi (autovettorizzate) saranno diverse e permetteranno comportamenti diversi tra i due sistemi.

```
1      BEGIN ORG      $8200  
2  
3
```

```

4 PIADA EQU      $2004 ;indirizzo di PIA-A dato, usato in input
5 PIACA EQU      $2005 ;indirizzo di PIA-A stato/controllo
6 PIADB EQU      $2006 ;indirizzo di PIA-B dato, usato in output
7 PIACB EQU      $2007 ;indirizzo di PIA-B controllo
8
9 TERD  EQU      $2000 ;indirizzo di TERMINAL registro dato
10 TERC  EQU      $2001 ;indirizzo di TERMINAL registro stato/
    controllo
11
12      JSR      DVAIN ;inizializza PIA porto A
13      JSR      DVBOUT ;inizializza PIA porto B
14      JSR      DVTER  ;inizializza terminal
15      MOVE.W  SR,DO ;legge il registro di stato
16      ANDI.W  #$D8FF,DO ;maschera per reg stato (stato
        utente, int abilitati)
17      MOVE.W  DO,SR ;pone liv int a 000
18
19      LOOP   JMP  LOOP  ;ciclo caldo dove il processore attende
        interrupt

```

Notiamo subito che in questo esercizio è necessario configurare il dispositivo PIA sia sul porto A che sul porto B per entrambi i sistemi, perché vogliamo garantire una comunicazione *FULL DUPLEX* come mostrato in figura 9.1. Come negli altri esercizi, configureremo il porto B per la scrittura e il porto A per la lettura.

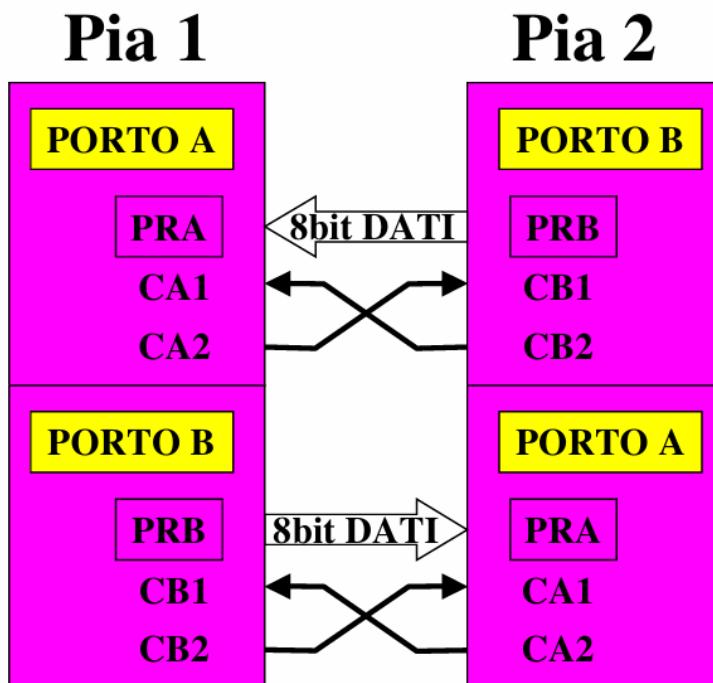


Figura 9.1: Configurazione Full Duplex

Le routine dedicate alla configurazione dei porti A e B sono uguali a quelle viste nell'esercizio 2 di questa sezione (??). La routine dedicata alla configurazione del terminale consta di una sola istruzione e ritorna:

```
1 DVTER MOVE.B #$3f,TERC ;seleziona indirizzo stato/controllo  
2 RTS
```

In pratica viene soltanto scritto il bye 00111111 nel registro di controllo/stato del terminale, il cui significato è chiarito dall'immagine 9.2.

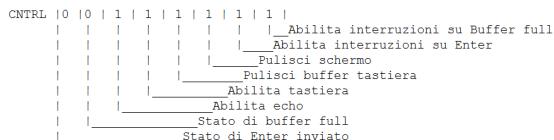


Figura 9.2: Configurazione registro controllo/stato del terminale

Dopodichè, il main entra in un ciclo idle in cui attende le interruzioni (dopo averle abilitate sul registro di stato). Vediamo nel dettaglio la ISR per la gestione dato proveniente dalla tastiera di TERMINAL e spedito, per tramite del PIA porto B, all'altro sistema. ISR associata all'interrupt di liv. 1, #vect 25 mappato a \$64 della ROM con ISR a \$8500.

```

1      ORG $8500      ricevi da tastiera
2      INT1    MOVE.L  A0,-(A7)      ;push di A0,A1,A2,D0,D1 in stack
3          supervisor
4          MOVE.L  A1,-(A7)
5          MOVE.L  A2,-(A7)
6          MOVE.L  D0,-(A7)
7          MOVE.L  D1,-(A7)
8          MOVEA.L #TERD ,AO
9          MOVEA.L #PIADB ,A1
10         MOVEA.L #PIACB ,A2

11     INPUT  MOVE.B  (A0),D0      ;acquisisci dato da terminal

12
13     *trasferisci il carattere letto alla PIA-B con handshaking
14         MOVE.B  (A1),D1      ;lettura fittizia
15         MOVE.B  D0,(A1)      ;Dato su bus di PIA porto B:
16             dopo la scrittura di PRB, CB2 si abbassa
17     *           ;cio fa abbassare CA1 sulla porta gemella
18     dell'altro sistema generando
19     *           ;un'interruzione che coincide con il segnale
20     DATA READY

21
22     ciclo2  MOVE.B  (A2),D1      ;In attesa di DATA ACKNOWLEDGE
23         ANDI.B  #$80,D1      ;aspetta che CRB7 divenga 1
24         BEQ  ciclo2

25
26     *fine trasferimento e handshaking

27
28     CMP.B      #13,D0      ;Se il carattere ricevuto  ENTER
29     BNE       INPUT      ;termina altrimenti prossimo carattere
30     ORI.B  #$1C,TERC      ;riabilita tastiera , pulisce buffer e
31         video
32
33     MOVE.L  (A7)+,D1      ;ripristino di D0,D1,A2,A1,A0

```

```

29      MOVE.L  (A7)+,D0
30      MOVE.L  (A7)+,A2
31      MOVE.L  (A7)+,A1
32      MOVE.L  (A7)+,A0
33      RTE

```

In questo caso è presente per forza di cose un ciclo improduttivo (ciclo2) perchè bisogna procedere sequenzialmente al set di istruzioni successivo che prevede un salto a INPUT se il carattere ricevuto non è quello finale (ENTER). Dopodichè resetta il terminale riabilitando tastiera e pulendo buffer e video scrivendo nel registro di controllo in accordo a quanto esposto nella figura 9.2, e ritorna.

Procediamo vedendo la ISR per il *buffer full*: praticamente è identica a quella per l'acquisizione del messaggio in seguito a ENTER, ma in questo caso vengono inviati tutti e 256 i caratteri conservati nel buffer del terminale. ISR a \$8600 associata all'interrupt di livello 2 #vect (24+2) => mappato a  $4^*26 = 104 = \$68$ .

```

1      ORG  $8600
2      INT2  MOVE.L  A0,-(A7)      ;push di A0,A1,A2,D0,D1 in stack
3          supervisore
4      MOVE.L  A1,-(A7)
5      MOVE.L  A2,-(A7)
6      MOVE.L  D0,-(A7)
7      MOVE.L  D1,-(A7)
8      MOVE.L  D2,-(A7)
9      MOVEA.L #TERD,A0
10     MOVEA.L #PIADB,A1
11     MOVEA.L #PIACB,A2
12     MOVE.B  #255,D2      ;#caratteri da trasferire
13
14
15     *trasferisci il carattere letto alla PIA-B con handshaking
16     MOVE.B  (A1),D1      ;lettura fittizia => serve per
17         azzerare CRB7 dopo il primo carattere, altrimenti
18         resta settato con l ack
19     MOVE.B  D0,(A1)      ;Dato su bus di PIA porto B: dopo
20         la scrittura di PRB, CB2 si abbassa
21     *          ;cio fa abbassare CA1 sulla porta gemella dell'
22         altro sistema generando
23     *          ;un'interruzione che coincide con il segnale
24     DATA READY
25
26
27     ciclo3  MOVE.B  (A2),D1      ;In attesa di DATA ACKNOWLEDGE
28     ANDI.B  #$80,D1      ;aspetta che CRB7 divenga 1
29     BEQ  ciclo3
30
31
32     *fine trasferimento e handshaking
33
34     DBRA      D2,SWAP ;contatore di bit inviati
35     ORI.B  #$1C,TERC ;riabilita tastiera ,pulisce buffer e
36         video

```

```

30      MOVE.L  (A7)+,D2 ;ripristino di D0,D1,A2,A1,A0
31      MOVE.L  (A7)+,D1
32      MOVE.L  (A7)+,D0
33      MOVE.L  (A7)+,A2
34      MOVE.L    (A7)+,A1
35      MOVE.L    (A7)+,AO
36      RTE

```

L'ultima interruzione è quella che scatena il porto A in seguito alla ricezione di un carattere.

```

1      ORG $8700
2
3      INT3      ANDI.B   #%"11101001,TERC ;disabilita tastiera,
4          cancella video,interruzioni su enter
5          MOVE.L  A1,-(A7)    ;salvataggio registri
6          MOVE.L  AO,-(A7)
7          MOVE.L  DO,-(A7)
8
9          MOVEA.L #TERD,A0 ;inizializzazione indirizzi device
10         MOVEA.L #PIADA,A1
11
12         MOVE.B  (A1),(AO) ;acquisisce il carattere e lo
13             trasferisce a Terminal
14         *           ;la lettura da PRA fa abbassare CRA7 e CA2 =>
15             ;nell'altro sistema si abbassa CB1
16         *           ;cio corrisponde all'attivazione di CRB7 che
17             funge da DATA ACKNOWLEDGE
18
19         MOVE.L  (A7)+,DO    ;ripristino registri
20         MOVE.L  (A7)+,AO
21         MOVE.L  (A7)+,A1
22
23         ORI.B  #$12,TERC ;riabilita tastiera e interruzioni su
24             enter
25         RTE
26
27         END BEGIN

```

### 9.1.2 Considerazioni finali

In conclusione, è utile chiarire che le linee di interruzione IRQA e IRQB sono direttamente collegate ai flag CRA7 e CRB7, che si alzano quando c'è una transizione attiva dei segnali CA1/CB1 rispettivamente. I flag di interruzione CRA7/CRB7 vengono automaticamente abbassati dopo un'operazione di READ sul porto corrispondente, e questo è il motivo per cui nella parte software è necessaria la *lettura fittizia*. Per quanto riguarda i bit del registro di stato/controllo del dispositivo PIA, in questa esercitazione abbiamo visto la configurazione per il protocollo di comunicazione handshaking. Le altre modalità, definite in base ai bit 5,4 e 3 dei registri di controllo, disciplinano il momento in cui CA2/CB2 deve essere rialzato, quindi non cambia nulla dal punto di vista delle funzionalità.

## 9.2 TAS (Test and Set)

TAS è un’istruzione che:

- **Controlla** se il bit più significativo dell’operando è 0 (semaforo libero);
- **Set** del bit più significativo a 1 (semaforo occupato) nel caso lo trovi libero.

A seguito dell’operazione i bit N e Z dello SR vengono aggiornati. Questa operazione è atomica (indivisibile), quindi usa un solo ciclo read-modify-write. La sua principale applicazione è nei sistemi multiprocessore: infatti un processore che esegue TAS non può essere interrotto tra la fase di Test e la fase di Set, rendendo consistente l’operazione di acquisizione del semaforo. Il metodo di indirizzamento è indiretto o tramite registro.

### 9.2.1 Esercizio 1

Due nodi B e C inviano K messaggi da N caratteri al nodo A, ognuno tramite una periferica PIA. Senza particolari ipotesi semplificative sulla priorità di un nodo rispetto ad un altro, scrivere il codice assembler eseguito dal nodo A in modo che:

- Se il nodo B trasmette un messaggio ad A, A deve terminare la ricezione dell’intero messaggio prima di ricevere un eventuale messaggio da C.
- Analogamente, se il nodo C trasmette un messaggio ad A, A deve terminare la ricezione dell’intero messaggio prima di ricevere un eventuale messaggio da B.

Il collegamento tra i dispositivi avviene tramite PIA secondo il meccanismo indicato precedentemente. Per la configurazione fare riferimento alla figura 9.3

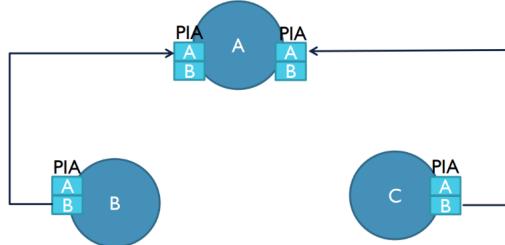


Figura 9.3: Schema logico

Possiamo considerare il nodo A come una risorsa condivisa dai nodi B e C. Siamo nel caso in cui è possibile utilizzare l’istruzione TAS: Prima di effettuare l’invio del messaggio, B e C devono verificare che A non sia impegnato con l’altro nodo. Questo è possibile farlo se A definisce una variabile *possesso*, dove il valore 0 indica il possesso di B, 1 indica il possesso di C, -1 indica risorsa libera. L’accesso alla variabile *possesso* deve essere mutualmente esclusivo, per evitare race conditions. Per questo motivo la variabile è acceduta tramit un lock, controllato e settato tramite TAS. Poichè stiamo utilizzando la PIA in configurazione tale per cui la ricezione di un messaggio dal nodo A causa interruzione, possiamo programmare l’ISR di A relativa alla ricezione di un messaggio da parte di B o da parte di C.

- **ISRB:** Per prima cosa si controlla che A non sia impegnato nella comunicazione con C, controllando la variabile possesso (in mutua esclusione). Se il possesso è di B (0) o è libero (-1) allora si può procedere all'invio, altrimenti B aspetta.
- **ISRC:** Per prima cosa si controlla che A non sia impegnato nella comunicazione con B, controllando la variabile possesso (in mutua esclusione). Se il possesso è di C (1) o è libero (-1) allora si può procedere all'invio, altrimenti C aspetta.

Questo meccanismo è pericoloso in quanto permette il DeadLock. Serve un meccanismo per riprendere la ricezione sospesa. Il motivo per cui una ricezione possa essere sospesa è un'interruzione di priorità maggiore o un malfunzionamento della periferica.  
vediamo lo pseudocodice relativo alla ISRB:

```

1   if (sem == verde){      \\istruzione atomica TAS
2       sem = rosso;
3       if (possesso != 1){ \\possesso non di C
4           possesso = 0;
5           sem = verde;
6           leggo_car_b();
7           car_counter_b++;
8           if (car_counter_b==N){ \\fine trasmissione
9               if(c_sospeso){
10                   possesso = 1; \\possesso di C
11                   leggo_car_c();
12                   car_counter_c++;
13               }
14               else{
15                   possesso=-1;
16               }
17           }
18       }
19       else{
20           sem = verde;
21           leggo_car_c();
22           car_counter_c++;
23       }
24       return 0;
25   }
26   return 0;

```

Nel codice vediamo che tramite l'istruzione TAS controlliamo se il semaforo è verde e lo settiamo subito a rosso. Quindi, se il possesso non è di C, B prende il possesso e reimposta il semaforo a verde, dopodichè procede con la lettura del prossimo carattere. Se il messaggio è finito, se C è sospeso C viene sbloccato attraverso la lettura del carattere di C, altrimenti viene liberato il possesso. Se il possesso è già di C, C potrebbe aver provato ad accedere alla sezione critica (invio del messaggio) mentre B controllava in mutua esclusione il possesso. In questo caso, C viene sbloccato attraverso la lettura di un carattere. Se in primo luogo B trova il semaforo rosso, B viene sospeso. Per verificare se B o C sono sospesi, si controlla nel registro di stato se CRA7B o CRA7C sono alti (interruzione pendente). Lo pseudocodice della ISRC è praticamente speculare a quello presentato sopra. Presentiamo di seguito l'implementazione commentata dell'esercizio.

### 9.2.1.1 Nodo A

```

1          ORG      $8000
2 MSG_CHAR    EQU      3
3 N_ITER      EQU      5
4 N_MEX_B     EQU      2      * Msg per iterazione
5 N_MEX_C     EQU      3      * Msg per iterazione
6
7 ***PIA ***
8 PIADA       EQU      $2004
9 PIACA        EQU      $2005
10 PIADA2      EQU      $2008
11 PIACA2      EQU      $2009
12
13 ***VARIABILI DEL SISTEMA A ***
14 BUFF_B       DS.W     64
15 BUFF_C       DS.W     32
16 CTR_CH_B    DC.W     0
17 TOT_B        DC.W     0
18 CTR_MS_B    DC.W     0
19 CTR_CH_C    DC.W     0
20 TOT_C        DC.W     0
21 CTR_MS_C    DC.W     0
22
23 END_B        DC.W     0
24 END_C        DC.W     0
25 K            DC.W     0
26
27          ORG      $8200
28 INIT         JSR      INIT
29             MOVE.W   SR,D1
30             ANDI.W   #$D8FF,D1
31             MOVE.W   D1,SR
32 LOOP         JMP      LOOP
33
34 INIT         MOVE     #$0,PIACA
35             MOVE     #$0,PIADA
36             MOVE     #00100101,PIACA
37             MOVE     #$0,PIACA2
38             MOVE     #$0,PIADA2
39             MOVE     #00100101,PIACA2
40             RTS
41
42 IB           CMP.W   #1,END_B
43             BEQ     IB_RTE
44             MOVEM.L DO-D2/A0,-(SP)
45             MOVE.W   TOT_B,DO
46             MOVE.W   CTR_CH_B,D1
47             MOVEA.L #BUFF_B,A0
48             MOVE     PIADA,(AO,DO)
49             ADDI.W   #1,DO

```

```

50      MOVE.W DO ,TOT_B
51      ADDI.W #1 ,D1
52      MOVE.W D1 ,CTR_CH_B
53      CMP.W #MSG_CHAR ,D1
54      BNE      EIB
55      MOVE.W #0 ,CTR_CH_B
56      MOVE.W CTR_MEX_B ,D1
57      ADDI.W #1 ,D1
58      MOVE.W D1 ,CTR_MEX_B
59      CMP.W #N_MEX_B_ITER ,D1
60      BNE      EIB
61      MOVE.W #1 ,END_B
62      CMP.W #1 ,END_C
63      BNE      EIB
64      MOVE.W K ,D2
65      ADDI.W #1 ,D2
66      MOVE.W D2 ,K
67      CMP.W #N_ITER ,D2
68      BEQ      EIB
69      MOVE.W #0 ,END_B
70      MOVE.W #0 ,END_C
71      MOVE.W #0 ,CTR_MEX_B
72      * Recupera il messaggio di C se si era bloccato.
73      MOVE.B PIACA2 ,DO
74      ANDI.B #$80 ,DO
75      BEQ      EIB
76      * Aggiorna i contatori e poi leggi il caratteri
77      MOVE.W TOT_C ,DO
78      MOVE.W DO ,D1
79      ADDI.W #1 ,D1
80      MOVE.W D1 ,TOT_C
81      MOVE.W #1 ,CTR_CH_C
82      MOVEA.L #BUFF_C ,AO
83      MOVE.B PIADA2 ,(AO ,DO)
84 EIB      MOVEM.L (SP)+ ,DO-D2/A0
85 IB_RTE    RTE
86
87 IC       CMP.W #1 ,END_C
88 BEQ      IC_RTE
89 MOVE.L DO-D2/A0 ,-(SP)
90 MOVE.W TOT_C ,DO
91 MOVE.W CTR_CH_C ,D1
92 MOVEA.L #BUFF_C ,AO
93 MOVE.B PIADA2 ,(AO ,DO)
94 ADDI.W #1 ,DO
95 MOVE.W DO ,TOT_C
96 ADDI.W #1 ,D1
97 MOVE.W D1 ,CTR_CH_C
98 CMP.W #MEX_CHAR ,D1
99 BNE      EIC
100 MOVE.W #1 ,END_C

```

```

101      CMP.W #1,END_B
102      BNE    EIC
103      MOVE.W K,D2
104      ADDI.W #1,D2
105      MOVE.W D2,K
106      CMP.W #N_ITER,D2
107      BEQ    EIC
108      MOVE.W #0,END_B
109      MOVE.W #0,END_C
110      MOVE.W #0,CTR_MEX_B
111      * Recupera il messaggio di C se si era bloccato.
112      MOVE.B PIACA,DO
113      ANDI.B #$80,DO
114      BEQ    EIC
115      * Aggiorna i contatori e poi leggi il caratteri
116      MOVE.W TOT_B,DO
117      MOVE.W DO,D1
118      ADDI.W #1,D1
119      MOVE.W D1,TOT_B
120      MOVE.W #1,CTR_CH_B
121      MOVEA.L BUFF_B,A0
122      MOVE.B PIADA2,(AO,DO)
123      EIC      MOVEM.L (SP)+,DO-D2/A0
124      IC_RTE   RTE

```

Vediamo il codice inherente al sistema B:

```

1          ORG      $8000
2 DIM      EQU      3
3 MES      DC.B     1,2,3,0
4 CNT      DC.W     0
5 PIADB    EQU      $2006
6 PIACB    EQU      $2007
7
8 ORG $8200
9 INIT     JSR      INIZIALIZZA
10
11 *Invio dimensione
12      MOVE.W #DIM,D3
13      MOVEA.L #PIADB,A0
14      MOVEA.L #MES,A1
15      MOVE.W #1,CNT * Inizia scrivendo 1 nel contatore di
16      bytes
17      MOVE.W CNT,DO  * DO mantiene sempre il conteggio di
18      bytes
19
20      MOVE.W SR,D1
21      ANDI.W #$D8FF,D1 *maschera per le interruzioni.
22      MOVE.W D1,SR
23      MOVE.B (AO),D2  * Effettua la lettura fittizia
24      MOVE.B (A1),(AO) * Scrivi il primo messaggio

```

```

24  LOOP  JMP  LOOP
25
26
27  INIZIALIZZA
28      MOVE.B #0,PIACB
29      MOVE.B #$FF,PIADB
30      MOVE.B #%00100101,PIACB
31      RTS
32
33  *interruzione livello 4 mappato a $70
34      ORG $8800
35  INT4      MOVE.B (A0),D2 * Lettura fittizia
36      MOVE.B (A1,D0),(A0) * Invia il carattere successivo
37      ADDI.W #1,D0
38      CMP.W D3,D0
39      BNE EI4
40      MOVE.W #0,D0
41  EI4      RTE
42  END      INIT

```

Il sistema C è praticamente speculare.

### 9.2.2 Esercizio 2 - Prova intercorso 2023

Un sistema è composto da 3 unità (A,B,C) tra loro collegate mediante due periferiche parallele che interconnettono A con B e A con C rispettivamente. Il sistema opera effettuando k iterazioni ( $k > 2$ ), in ciascuna delle quali A deve ricevere globalmente 2 messaggi di N caratteri da B e 1 messaggio di N caratteri da C ( $N > 2$ ). I messaggi da B e da C possono essere ricevuti in un ordine qualsiasi ma non deve essere mai possibile ricevere caratteri appartenenti a messaggi diversi intervallati tra di loro. Vediamo lo pseudocodice relativo alla ISRB:

```

1   if (sem == verde){          \\istruzione atomica TAS
2       sem = rosso;
3       if (possesso != 1 && end_B == 0){
4           possesso = 0;
5           sem = verde;
6           leggo_car_b();
7           car_counter_b++;
8           if(car_counter_b == N){
9               mex_counter_b++;
10              if(mex_counter_b == N_mex_b){
11                  end_B = 1;
12              }
13              if(c_sospeso){
14                  possesso = 1; \\possesso di C
15                  leggo_car_c();
16                  car_counter_c++;
17              }
18              else{
19                  possesso=-1;
20              }

```

```

21         }
22     }
23     else{
24         sem = verde;
25         leggo_car_c();
26         car_counter_c++;
27     }
28     return 0;
29 }
30 return 0;

```

La differenza rispetto all'esercizio 9.2.1, B in questo caso per procedere deve controllare sia che la variabile possesso non sia di B, sia che A non abbia già ricevuto il numero di messaggi per quell'iterazione. Dopo la ricezione del carattere, non solo viene gestito il contatore relativo ai caratteri nel messaggio, ma anche un contatore che tiene conto del numero di messaggi ricevuti nella corrente iterazione. Se questo numero diventa uguale a N, si pone  $end_B = 1$ , e alla successiva interruzione non si riceverà alcun carattere, ma si sveglierà C se necessario e si uscirà.

# Appendice A

## Appendice

### A.1 Asim e Asimtool

Per la scrittura e la simulazione dei codici, saranno utilizzati i seguenti strumenti:

- **ASIM:** Strumento per la simulazione del motorola 68k
- **ASIM-Tool:** Editor di testo e compilatore dei file .a68

#### A.1.1 Asimtool

Per asimtool, dopo aver scritto il file bisogna generare il file LIS, che poi sarà inserito all'interno del simulaore ASIM Tale file va generato secondo il seguente path:

**Assemble -> Assemble File <Nome\_File>.a68**

Fatta tale operazione, nella cartella dove vi è salvato il file a68 dovrebbe essersi generato il file LIS

Nel caso ci fossero particolari errori, asimtool li mostrerà a video specificando le righe su cui tali errori si presentano. Si invita a tenere ben cura della spaziatura tra i vari comandi e la loro leggittima posizione

#### A.1.2 ASIM

Una volta generato il file LIS con ASIM-tool, aprire ASIM ed impostare l'ambiente. Per impostare l'ambiente è richiesto un file cfg, che riporta i vari componenti che saranno mostrati all'interno del simulatore (tipo la memoria, il processore ecc.). Il file base.cfg può essere trovato sui canali ufficiali degli studenti o può essere richiesto al professore. Tale file non contiene altro che una lista di componenti che verranno poi mostrati all'interno del simulatore. Una volta aperto il file bisogna seguire il seguente path:

**Window -> Tile**

Tale opzione ci permette di poter vedere tutte le schermate aperte in maniera ordinata. Successivamente all'ordinamento delle schermate, bisogna "attivare" la configurazione, per fare ciò bisogna premere su di un tasto nella barra degli strumenti in Alto con illustrata una grossa I. Una volta attivato il nostro ambiente, tenere cura di selezionare la finestra su cui c'è scritto di caricare il LIS. Una volta fatto questo in alto, tra i menu comparirà una nuova voce, ovvero: **Proc\_Unit**. Una volta apparso tale menu basterà seguire il seguente path per poter selezionare il file LIS:

**Proc\_Unit -> Load Assembler**

Tale comando permetterà di poter caricare il file LIS generato da Asimtool, che dovrà essere selezionato appositamente tramite il file explorer. Una volta caricato il file LIS bisognerà solo eseguire il programma. Si consiglia, prima di eseguire, di attivare la visualizzazione dei registri interni. Tale cosa potrà essere fatta, selezionando la finestra in cui è caricato il file LIS e poi seguire il seguente path:

#### **Proc\_Unit -> Show Registers**

Questo permetterà di poter visualizzare i registri interni del processore nella parte bassa della finestra

### **A.1.3 Esecuzione dei programmi**

Per l'esecuzione dei programmi, si può procedere in due modi:

- **Passo Passo:** premendo sull'omino lento in alto
- **Fino alla fine:** premendo sull'omino che sembra correre

Il consiglio è sempre quello di verificare il funzionamento del programma passo passo e poi di utilizzare l'esecuzione veloce.

Per verificare o controllare particolari indirizzi di memoria si può utilizzare un tool interno. Selezionando la memoria (quella che solitamente ha colori blu) e poi seguendo il seguente path:

#### **Memory -> Show\_Loc**

Si aprirà una finestra che ci permetterà di scrivere la locazione di memoria che vogliamo controllare. Una volta inserita e aver premuto "ok", la finestra mostrerà la memoria all'indirizzo richiesto in alto.

### **A.1.4 Configurazione (file .cfg)**

I sistemi hardware da simulare su ASIM sono caratterizzati da *configurazioni*. Una configurazione contiene la descrizione degli oggetti da simulare e delle loro interconnessioni. Gli oggetti simulabili da ASIM sono raggruppati in quattro categorie:

- **Processore;**
- **Nodo;**
- **BUS/Memoria;**
- **Device;**

L'appartenenza di un oggetto ad una classe è determinata dal soddisfacimento di alcune proprietà.

#### **A.1.4.1 Processore**

È un processore un qualsiasi dispositivo che ha lo stato interno rappresentato dal contenuto dei suoi registri più un insieme di informazioni riguardanti i bus a cui è connesso e un set di eventuali richieste di interruzioni. Un processore gode delle seguenti proprietà:

- Può eseguire un programma contenuto in memoria (interna al dispositivo o esterna);

- Può accedere attraverso il BUS ad una memoria esterna o ad altri dispositivi per operazioni di lettura o scrittura;
- Può servire e/o gestire interruzioni provenienti da altri dispositivi, eventualmente assegnando a questi un livello di priorità.

#### A.1.4.2 Nodo

È un nodo un dispositivo che raccoglie informazioni relative alle connessioni con altri nodi ed ai messaggi in transito da/verso questi, e che gode delle seguenti proprietà:

- può consentire la connessione tra dispositivi dello stesso tipo;
- può gestire, come nodo intermedio, la connessione tra dispositivi dello stesso tipo, instradando la comunicazione;
- può avere capacità autonome di elaborazione e trasformare i messaggi dati o ricevuti;
- Può essere connesso ad un BUS/memoria.

#### A.1.4.3 BUS/Memoria

È dispositivo il cui stato interno è caratterizzato dai valori assunti da un insieme molto ampio di registri e/o da informazioni riguardanti i dispositivi che esso connette e che gode delle seguenti proprietà:

- può consentire a dispositivi di tipo processore di leggere o scrivere i suoi registri interni (la selezione avviene in base all'indirizzo);
- può consentire a dispositivi di tipo processore di leggere o scrivere i registri dei dispositivi di tipo device (la selezione avviene in base all'indirizzo);
- regola l'accesso di più dispositivi di tipo processore ai suoi registri ed ai device (in ogni istante al più un accesso è in corso, gli altri processori devono attendere);
- gestisce la memoria fisica effettuando il “mapping” degli indirizzi virtuali negli indirizzi fisici;
- può consentire l'esecuzione di cicli non interrompibili di lettura-modifica;
- può connettersi ad altri dispositivi dello stesso tipo sia per costruire accessi da un altro bus sia verso un altro bus.

#### A.1.4.4 Device

È un device tutto ciò che non rientra nelle categorie precedenti, e che gode delle seguenti proprietà:

- può essere connesso ad un bus/memoria in modo che un processore possa accedere ad esso;
- può essere in grado di generare delle interruzioni da inviare ad un processore;

- può essere connesso e comunicare con un altro device della stessa macchina o di un'altra macchina;
- può connettere dispositivi di tipo processore e bus/memoria a dispositivi del tipo bus/memoria.

#### A.1.4.5 Generazione di configurazioni

Dal menù *edit* di ASIM è possibile generare una configurazione. I file che contengono le informazioni relative ad una configurazione hanno come estensione *.cfg*. La configurazione di un determinato oggetto viene specificata attraverso la compilazione manuale dei seguenti campi:

- **Configuration Name:** Nome della configurazione corrente;
- **Chip Name:** Nome in codice dell'oggetto se esistono più oggetti in una classe;
- **Type:** Classe di appartenenza dell'oggetto;
- **Identif:** Codice numerico che identifica il dispositivo nell'ambito della configurazione;
- **Bus:** Identificatore di un oggetto della classe Bus con cui lo specifico oggetto può interagire in qualche modo;
- **Address:** Indirizzi fisici compresi nello spazio di indirizzamento visto dall'oggetto;
- **com1,com2,...,com4:** Campi che codificano particolari caratteristiche dell'oggetto.

Partiamo dalla configurazione di un oggetto di tipo **Processore**, esposta in figura A.1:

Name	Nome dell'oggetto processore (chiave) {M68000,...}
Identif	Intero (\$01..\$FF) che identifica univocamente l'oggetto nella configurazione ASIM
Type	CPU
Address1	indirizzo da assegnare al reset all'User Stack Pointer (USP)
Address2	indirizzo da assegnare al reset al Supervisor Stack Pointer (SSP)
BUS	Identificatore del bus a cui è connesso il processore
COM1	n.s.
COM2	n.s.
COM3	n.s.
COM4	n.s.

<b>CHIP Name: M68000</b> <b>Type: CPU.      Identif: 01.      BUS: 0002.</b> <b>Adres 1: 00009000.      Address 2: 00009200.</b> <b>Com1: 0000. Com2: 0000. Com3: 0000. Com4: 0000.</b>
--

Figura A.1: Configurazione CPU

È importante osservare che un generico device generante interruzioni può essere connesso ad un dispositivo in grado di gestirle (CPU, PIC). ASIM implementa un meccanismo

di base di gestione delle interruzioni di tipo *vettorizzato*: Tre linee, IPL0, IPL1, IPL2 (Interrupt request Priority Level) codificano l'evento interruzione in ingresso secondo il livello prioritario assegnato al dispositivo interrompente; Il processore, all'arrivo di un'interruzione (sia essa di tipo Hw che Sw) da servire (e non mascherare) avvia un ciclo di bus mediante il quale acquisisce dalla periferica interrompente uno specifico numero di vettore espresso su 8 bit (l'area ROM dei vettori occupa, infatti, il primo K dello spazio indirizzi di memoria); Tale numero, moltiplicato per 4 (mediante l'esecuzione di uno shift a sinistra di 2 posizioni), fornisce il puntatore ad una locazione di memoria ROM contenente il vettore associato all'Interrupt Service Routine (puntatore alla ISR). Tutti i vettori sono di 32 bit (espressi su due word). Fa eccezione il vettore di reset numero 0, di 4 word di cui 2 usate per inizializzare il PC e 2 word per inizializzare il supervisor stack pointer. Ogni segnale di interruzione è descritto da una struttura dati di 2 byte contenenti i seguenti tre campi:

- **vector number**: bit b7-b0 del byte più significativo;
- **priority**: bit b7-b4 del byte meno significativo;
- **linea di interruzione**: bit b3-b0 del byte meno significativo.

Procediamo con la configurazione di un oggetto di tipo **Bus/Memoria**, esposta in figura A.2:

Name	MEMORIA (nome generico opzionale);
Identif	Intero (\$01..\$FF) che identifica univocamente l'oggetto nella configurazione ASIM;
Type	MMU/BUS;
Address1	Indirizzo base RAM;
Address2	Indirizzo base ROM;
BUS	Identificatore di bus esterno verso cui esportare lo spazio indirizzi delle memorie RAM/ROM (dare visibilità del proprio spazio indirizzi verso dispositivi non appartenenti alla medesima configurazione BUS);
COM1	Identificatore di bus esterno da cui importare la visibilità dello spazio di memoria RAM/ROM e anche degli oggetti device, cpu ad esso connessi (possibilità di avere visibilità di altro spazio indirizzi);
COM2	dimensione (in esadecimale) della memoria RAM in blocchi da 1 Kbyte;
COM3	dimensione (in esadecimale) della memoria ROM in blocchi da 1 Kbyte;
COM4	n.s.

Figura A.2: Configurazione BUS/Memoria

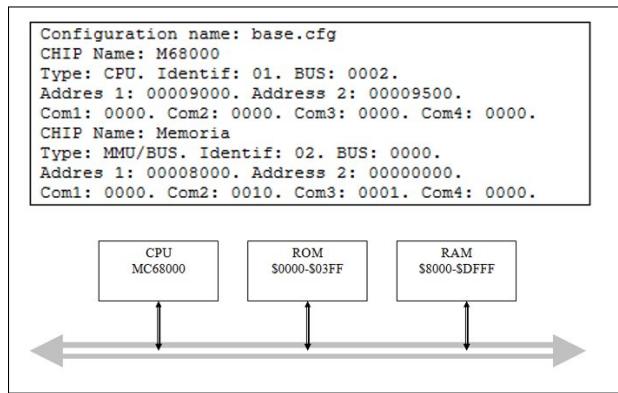


Figura A.3: Esempio configurazione 1

Configuration name: base.cfg

CHIP Name: M68000  
Type: CPU. Identif: 01. BUS: 0002.  
Addres 1: 00009000. Address 2: 00009500.  
Com1: 0000. Com2: 0000. Com3: 0000. Com4: 0000.

CHIP Name: Memoria  
Type: MMU/BUS. Identif: 02. BUS: 0000.  
Addres 1: 00008000. Address 2: 00000000.  
Com1: 0003. Com2: 0010. Com3: 0001. Com4: 0000.

CHIP Name: Memoria2  
Type: MMU/BUS. Identif: 03. BUS: 0000.  
Addres 1: 0000D000. Address 2: 00000000.  
Com1: 0000. Com2: 0001. Com3: 0000. Com4: 0000.

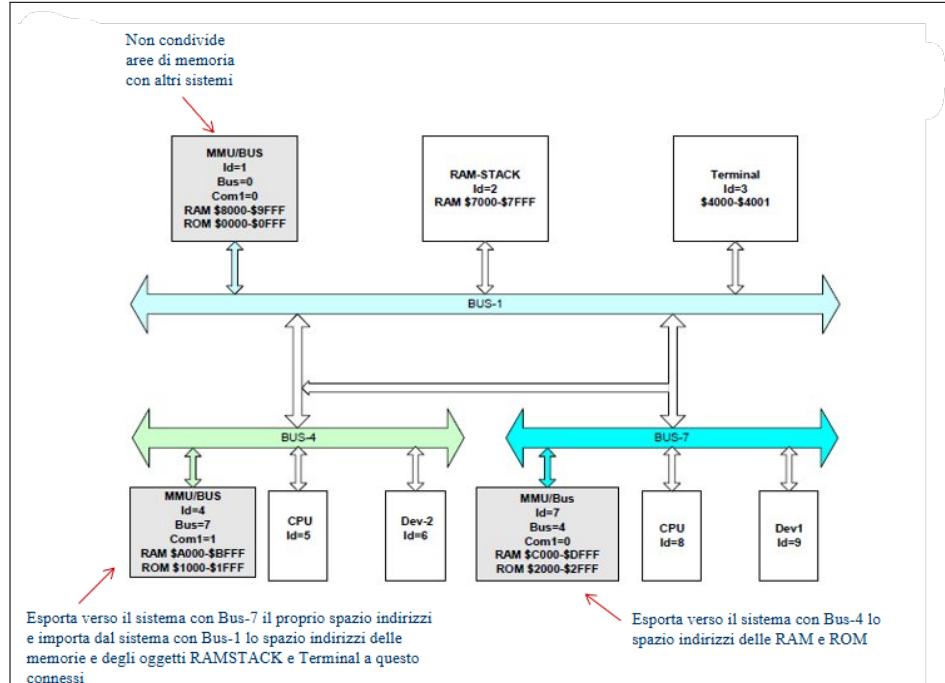


Figura A.4: Esempio configurazione 2