

Appunti di
Architettura e progettazione dei calcolatori

Corso di laurea magistrale in
Ingegneria Informatica

Indice

Introduzione	7
1 Richiami ed approfondimenti sui sistemi di elaborazione	9
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	26
2 Gestione dei dispositivi di IO	29
2.1 Architettura generale di un dispositivo di I/O	29
2.1.1 Modalità di comunicazione	30
2.1.2 Interfacciamento CPU e periferica	30
2.1.3 BUS	31
2.1.4 Driver	32
2.1.5 PIA	33
2.1.6 Interruzioni	40
2.1.7 Estensione del modello IO generale	47
2.2 DMA (Direct Memory Access)	47
2.2.1 Utilizzo effettivo in ASIM	49
2.2.2 Implementazione in Motorola 68k	51
2.3 USART (Universal Synchronous-Asynchronous Receiver-Transmitter) . . .	53
2.3.1 Comunicazioni Sincrona ed Asincrona	53
2.3.2 Intel 8251A	55
3 Architettura dei processori	61
3.1 Generalità sul processore	61
3.2 Architettura dei Processori moderni	63
3.2.1 Multi-Computer e Multi-Processore	66
3.2.2 Speed Up	66

3.2.3	Coerenza della memoria cache nelle architetture parallele	68
3.3	Sistema Pipeline	71
3.3.1	Modelli di sistemi pipeline	73
3.3.2	Architettura del MIPS	74
3.3.3	Registri Intermedi	77
3.3.4	Gestione dei Salti	77
3.3.5	Gestione dei conflitti sui dati	80
3.3.6	Gestione delle interruzioni	83
3.4	Architetture Superscalari	85
3.4.1	Gestione delle Collisioni	85
4	Memoria	89
4.1	Memoria cache	89
4.1.1	Politiche di Sostituzione	90
4.1.2	Gestione dell'allineamento	91
4.1.3	Set Associative	91
4.1.4	Dimensionamento	93
4.2	Memoria Virtuale	94
4.2.1	Traduzione degli indirizzi	95
4.2.2	Cache e Memoria Virtuale	96
4.3	Struttura delle memorie	97
4.3.1	Memorie Dinamiche	98
4.3.2	Memorie Statiche	99
5	Processori ARM	101
5.1	Architettura ARM	101
5.1.1	Modello di programmazione	102
5.1.2	Gestione delle Eccezioni	104
5.1.3	Differenze principali tra architetture	105
5.2	STM32F4	106
5.2.1	CubeIDE	106
5.2.2	Gestione mutua esclusione	107
5.2.3	DMA	110
5.3	BUS e protocolli	110
5.3.1	I2C	110
6	Esercitazioni	113
6.1	Debunk esercizi PIA	113
6.1.1	Esercizio 1	113
6.1.2	Esercizio 2	117
6.1.3	Esercizio 3	119
6.1.4	Considerazioni finali	123
6.2	TAS (Test and Set)	123
6.2.1	Esercizio 1	124
6.2.2	Esercizio 2 - Prova intercorso 2023	125

A Appendice	127
A.1 Asim e Asimtool	127
A.1.1 Asimtool	127
A.1.2 ASIM	127
A.1.3 Esecuzione dei programmi	128
A.1.4 Configurazione (file .cfg)	128

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, essa è 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 varie microarchitetture 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 un determinato dato su cui vado ad operare con varie tipologie di interazioni
- **Registri indirizzo:** Registri che sono utilizzati per conservare gli indirizzi a cui magari si vuole accedere in memoria (tipo puntatori in C/C++)
- **Registri Speciali:** Registri utilizzabili dal programmatore ma con funzioni diverse, ovvero:

- **PC (Program Counter)**: memorizza la posizione del prossimo comando da eseguire del nostro programma
- **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 grado 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:

- **MA (Memory Access)**: Registro utilizzato dal processore per scrivere l’indirizzo di memoria a cui si vuole accedere
- **MB (Memory Buffer)**: 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 per noi funziona come un blocco a cui dati indicizzati. Per cui in base all’operazione che la CU va ad effettuare, modifica i valori di: MA, MB, segnale di read e di write. Posso memorizzare i dati in memoria in vario modo, quindi posizionandoli in vario modo, tali posizioni rispettano le seguenti due tipologie di organizzazione:

- **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

È indifferente ai fini del funzionamento del processore 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 il suo allineamento in memoria, dato dal fatto della gestione delle sue celle di byte 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 rispetto a queste due parti, per cui, la memoria sarà organizzata da due principali parti:

- **Area codice (o area istruzioni):** Area dove sono contenuti i programmi e le istruzioni da eseguire
- **Area dati:** Area dove sono memorizzati i dati

In generale le aree minori sono le aree codice, mentre le restanti sono aree dati

1.1.2 Istruzioni

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

- **Codici operativi:** Operazioni elementari implementate in base all'architettura del processore, quindi istruzioni appartenenti al processore che specificano tutti i registri ed i passaggi da effettuare per determinate operazioni
- **Operandi sorgente:** Valori che possono essere memorizzati sia in dei registri interni che esterni (in base alla tipologia di operazione), su cui poi i codici operativi appartenenti all'istruzione vanno a lavorare
- **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 poichè potrebbe diventare uno dei due operandi sorgente

In generale un'istruzione è una composizione di bit, essa stessa immagazzinata 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.

Le architetture con cui sono progettate i più moderni processori possono essere di 2 tipologie:

- **CISC (Complex-Instruction-Set-Computer)**: dove le istruzioni a disposizione del programmatore possono essere anche più complesse (comprendono l'utilizzo anche di più istruzioni semplici), un classico esempio è la memorizzazione di un dato in una memoria, che nel caso CISC può essere effettuato tramite un singolo comando
- **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 (come nel caso della memorizzazione di una dato in memoria)

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. La specifica su quale istruzione prelevare la conserva il PC (Program Counter) che conserva l'indirizzo di memoria da cui prelevare la prossima istruzione. Una volta prelevata l'istruzione dalla memoria tramite gli indirizzi MA ed MB, questa viene caricata nell'IR, che conserverà tale istruzione durante tutto il processo di decodifica ed esecuzione delle operazioni specificate

Le istruzioni, in generale possono essere classificate nel seguente modo:

- **Trasferimento dati**: Codici che mi permettono di copiare un dato da un determinato operando e spostarlo nell'altro (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 degli operandi con una logica bit a bit, effettuata l'operazione il risultato viene inserito all'interno di una data destinazione
- **Scorrimento**: operazioni effettuate sugli operandi in ingresso che restituiscono lo scorrimento verso (destra o sinistra) dell'operando e lo memorizzano in una data destinazione
- **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 apposite

1.2 Motorola 68k

Una volta introdotti i concetti "teorici" e tecnologici da conoscere, si possono iniziare ad osservare i principali costrutti per la programmazione con il Motorola 68k. Conviene, quindi, non solo capire quali sono i determinati codici per le varie tipologie di istruzioni specificate in [1.1.2], ma anche come costruire i principali componenti di un linguaggio di più alto livello (di cui si presupponen una minima conoscenza) tali costrutti possono essere: cicli, blocchi di decisione, ecc.

1.2.1 Registri General Purpose

I registri General-Purpose (o registri macchina) è l'insieme dei registri che sono messi a disposizione del programmatore per scrivere vari codici in accoppiata con gli specifici codici operativi. I registri a disposizione, nel caso del motorola 68k sono i seguenti:

- **Registri Dato:** Registri D0,D1,...,D7
- **Registri Indirizzo:** Registri A0,A1,...,A7 ed A7' (utilizzato nel caso di privilegi alti)
- **Status Register (SR):** Registro che contiene vari controlli sia sui risultati delle operazioni dell'ALU che sullo stato dell'esecuzione (se in super-user o meno)

Tali registri sono fondamentali per l'esecuzione dei comandi come nel caso del 68k. Con tali registri saranno implementati tutti gli algoritmi utili nel resto del corso

1.2.2 Codici di Spostamento dati o indirizzi

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

- **Lunghezza degli operandi:** solitamente specificata con delle lettere alla fine del comando
- **Tipologia di indirizzamento:** La **MOVE** è una tra le poche operazioni che ammette tutte le tipologie di indirizzamento possibili, l'unico che può portare degli errori è l'indirizzamento immediato per l'operando di destinazione, che di per sé non ha 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 A1 = AO
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(AO, D1.L), D2      * D2 = valore all'indirizzo AO + 8 +
20           D1

21     * Indirizzamento PC relativo con spiazzamento
22     MOVE.B 6(PC), D0      * D0 = byte situato 6 byte dopo il Program
23           Counter

24     * Push di un registro nello Stack
25     MOVE.L D0, -(A7)      * Salva D0 nello stack (decremento SP)

26
27     * Pop dallo Stack in un registro
28     MOVE.L (A7)+, D0      * Carica D0 con il valore in cima allo
29           stack (incremento SP)

30
31     * Push di un registro di indirizzo
32     MOVEA.L A0, -(A7)      * Salva A0 nello stack

33
34     * Pop di un registro di indirizzo
35     MOVEA.L (A7)+, A0      * Carica A0 con il valore in cima allo
36           stack

37     * Salvataggio multiplo nello stack
38     MOVEM.L D0-D3/A0-A2, -(A7)      * Salva piu' registri nello stack

39
40     * Ripristino multiplo dallo stack
41     MOVEM.L (A7)+, D0-D3/A0-A2      * Ripristina piu' registri dallo
42           stack

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

```

Nota su MOVE.L D0, -(A7): il - nel "pushare" nello stack è dovuto al fatto che, per costruzione, lo stack "cresce" verso il basso, quindi quindi deve decrescere di 4 byte per

ospitare quanto vi sto inserendo, ciò è così per costruzione ma è scollegato dal fatto che il 68k sia big endian).

Nota su RTS: concettualmente, 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

Come si può notare il comando **MOVE** permette di poter utilizzare tutti i tipi di indicizzazione di memoria possibili

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 condizionati:** 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 incondizionati:** I salti condizionati sono effettuati in base al verificarsi di una determinata condizione. Nel caso del motorola 68k la condizione è associata ai valori associati ai 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, senza che il programmatore debba gestire direttamente tale condizione

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  
   (branch always)  
2 JMP address    * Salto incondizionato all'indirizzo  
   specificato  
3 JMP (A0)       * Salto all'indirizzo contenuto in A0
```

Solitamente nelle varie applicazioni si preferisce utilizzare il comando BRA, poichè 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 eguale 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, ed, in particolare, sono:

```
1 0 - C - Carry (riporto / overflow logico)  
2 1 - V - Overflow aritmetico  
3 2 - Z - Risultato uguale a 0  
4 3 - N - Risultato negativo  
5 4 - X - Extend (per operazioni multiple)
```

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

1.2.5 Codici Logici

I codici logici sono operazioni che possono essere effettuate su degli operandi operando 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 = DO  
2   & 3  
3 AND.W D0, D1    * AND tra registri, D1 = D1 & DO  
4 AND.L (A0), D0  * AND tra valore in memoria puntato da A0 e  
5   DO  
6  
7 OR #3, D0       * OR bit a bit con valore immediato, DO = DO  
8   / 3  
9 OR.W D0, D1    * OR tra registri, D1 = D1 / DO  
10 OR.L (A0), D0  * OR tra valore in memoria puntato da A0 e DO  
11  
12 EOR #3, D0     * XOR bit a bit con valore immediato, DO = DO  
13   XOR 3  
14 EOR.W D0, D1   * XOR tra registri, D1 = D1 XOR DO  
15 EOR.L (A0), D0 * XOR tra valore in memoria puntato da A0 e DO  
16  
17 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 (mantiene il segno), da notare che lo shift aritmetico 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)
2   ASR #1, D0      * Shift aritmetico a destra di 1 bit
3
4   LSL #1, D0      * Shift logico a sinistra di 1 bit (riempie con 0)
5   LSR #1, D0      * Shift logico a destra di 1 bit
6
7   ROL #1, D0      * Rotazione a sinistra di 1 bit (il bit piu' alto rientra da destra)
8   ROR #1, D0      * Rotazione a destra di 1 bit
9
10  ROXL #1, D0     * Rotazione a sinistra con Carry
11  ROXR #1, D0     * Rotazione a destra con Carry
```

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 modificare D0, aggiorna i flag)
2   CMP.W D0, D1     * Confronta D1 con D0 (D1 - D0, aggiorna solo i flag)
3   CMP.L (A0), D0    * Confronta D0 con il valore in memoria puntato da A0
4
5   CMPI #10, D0      * Confronto immediato con D0 (D0 - 10, aggiorna solo i flag)
6
7   CMPA.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, senza modificarlo)
2   TST.W (A0)        * Testa il valore in memoria puntato da A0
3
```

```

4   BTST #3, D0      * Testa il bit 3 di D0 (imposta Z se il bit e
      , 0)
5   BTST #5, (AO)    * Testa il bit 5 della memoria puntata da AO
6
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
10  BCLR #3, D0      * Azzera il bit 3 di D0
11  BCLR #5, (AO)    * Azzera il bit 5 della memoria puntata da AO
12
13  BCHG #3, D0      * Inverte il bit 3 di D0 (0 -> 1, 1 -> 0)
14  BCHG #5, (AO)    * Inverte il bit 5 della memoria puntata da
      AO
15
16  TAS D0           * Testa e imposta il bit piu' alto (7) di D0
17  TAS (AO)         * Testa e imposta il bit 7 del valore in
      memoria puntato da AO

```

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 andare ad effettuare (cerca un uguaglianza a 0, una maggiorazione, una minorazione, cosa sto cercando? quale operazione?)

Esempio di un classico If-then:

```

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

```

Esempio di un If-Then-else

```

1      MOVE.L  DO, D1          * Carica valori nei registri
2          (supponiamo che DO e D1 abbiano già valori)
3      CMP.L  D1, DO          * Confronta DO con D1 (DO -
4          D1)
5      BGT     THEN_BLOCK    * Se DO > D1, salta al
6          blocco THEN
7
8          * ELSE block
9      MOVE.L  #0, D2          * D2 = 0
10     BRA     END_IF         * Salta oltre il blocco THEN
11          per evitare di eseguirlo
12
13 THEN_BLOCK:   MOVE.L  #1, D2          * D2 = 1
14
15 END_IF:       * Codice successivo...

```

Come possiamo notare dal codice dell'If-then-else, abbiamo un insieme di salti sia condizionati che non condizionati che sono pilotati da una specifica istruzione di compare

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, DO          * Inizializza il contatore
2          DO = 0
3
4 FOR_LOOP:   CMP.L  #10, DO          * Confronta DO con 10
5          BGE     FOR_END        * Se DO >= 10, esce dal
6          ciclo
7
8          * Corpo del ciclo
9          NOP                  * Istruzione di esempio (
10         da sostituire con il codice reale)
11
12 ADDQ.L  #1, DO          * Incrementa DO di 1
13 BRA     FOR_LOOP        * Ripete il ciclo
14
15 FOR_END:   * Codice successivo dopo il ciclo

```

1.2.8.3 Ciclo While

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

```

1 WHILE_LOOP:   CMP.L  #0, D1          * Confronta D1 con 0
2          BLE     WHILE_END        * Se D1 <= 0, esce dal
3          ciclo
4
5          * Corpo del ciclo
6          NOP                  * Istruzione di esempio

```

```

6           SUBQ.L  #1, D1          * Decrementa D1 di 1
7           BRA     WHILE_LOOP    * Ripete il ciclo
8
9
10  WHILE_END:   * Codice successivo dopo il ciclo

```

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 comunicazione
- **Con Stack:** Gli operandi sono locati 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
2          D0
3      MOVE.L  #10, D1         * Carica il secondo operando in
4          D1
5
6      MOVE.L  D0, -(A7)       * Push del primo operando nello
7          stack
8      MOVE.L  D1, -(A7)       * Push del secondo operando
9          nello stack
10
11     JSR     SUM_SUB        * Chiamata alla subroutine
12
13     MOVE.L  (A7)+, D2       * Il chiamante preleva il
14     risultato dallo stack
15
16
17     ADDQ.L  #8, A7         * Pulizia dello stack (2 valori
18     da 4 byte)
19
20     * D2 ora contiene il risultato della somma
21
22
23     * Codice successivo...
24
25
26  SUM_SUB:   MOVE.L  (A7)+, D0      * Pop del primo operando dallo
27      stack
28     MOVE.L  (A7)+, D1      * Pop del secondo operando
29     dallo stack
30
31     ADD.L   D1, D0         * Somma D0 + D1, risultato in
32     D0

```

```
22      MOVE.L  D0, -(A7)          * Push del risultato nello  
           stack  
23      RTS                      * Ritorna al chiamante  
24
```

Esempio di chiamata a subroutine con operandi nello stack e risultato memorizzato nello stack

```

1      MOVE.L #5, D0          * Carica il primo operando in
2          D0
3      MOVE.L #10, D1         * Carica il secondo operando in
4          D1
5
6      MOVE.L D0, -(A7)       * Push del primo operando nello
7          stack
8      MOVE.L D1, -(A7)       * Push del secondo operando
9          nello stack
10
11     JSR      SUM_SUB      * Chiamata alla subroutine
12
13     MOVE.L (A7)+, D2        * Il chiamante preleva il
14         risultato dallo stack
15
16     ADDQ.L #8, A7          * Pulizia dello stack (2 valori
17         da 4 byte)
18
19     * D2 ora contiene il risultato della somma
20
21     * Codice successivo...
22
23     SUM_SUB:   MOVE.L (A7)+, D0      * Pop del primo operando dallo
24         stack
25         stack
26     MOVE.L (A7)+, D1      * Pop del secondo operando
27         dallo stack
28
29     ADD.L   D1, D0          * Somma D0 + D1, risultato in
30         D0
31
32     MOVE.L D0, -(A7)       * Push del risultato nello
33         stack
34
35     RTS                  * Ritorna al chiamante

```

Caso 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
6      * Dopo il ritorno, il risultato e' in D0
7
8      * Codice successivo...
9
10    SUM_REGS:  ADD.L   D1, D0      * Somma D0 + D1, risultato in D0
11
12    RTS                  * Ritorna al chiamante

```

1.2.9 Valutazione degli accessi in memoria

Quando utilizzo i comandi precedentemente presentati avrò una quantità diversa di accessi in memoria in base alla composizione che ho dato al mio codice. 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
<code>MOVE.L D0,D1</code>	1	0
<code>MOVE.W D0,D1</code>	1	0
<code>MOVE.L #7,D1</code>	3	0
<code>MOVE.W (A0),(A1)</code>	1	2
<code>MOVE.W (A0),VAR</code>	3	2

Tabella 1.1: 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 immegiato considerato una WORD, allora il numero di accessi aggiuntivi per prelevare l'immediato è uguale a 1. Mentre se stessi lavorango con le Long World (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 operando. 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 (o I/O), 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. Data l'eterogeneità di tali periferiche è richiesto che queste ultime siano gestite in un certo modo, o almeno, che la loro gestione principale sia di un certo tipo. Ciò, quindi, pone le basi su come dovremmo interfacciarcici all'utilizzo di tali dispositivi.

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 di quel dato 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. Quindi immaginiamoci anche una tastiera con il suo connettore USB.

Un esempio di dispositivo esterno è la memoria HDD. La memoria HDD ha infatti i tre registri di Dato, Stato e Controllo, quando si vuole scrivere su tale memoria, la CPU va a modificare i registri in modo da garantire tale operazione. 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 tra loro, la comunicazione è sincronizzata da un clock comune;
- **Collegamento con Handshacking:** L'handshacking è una modalità di comunicazione asincrona, poichè si sfruttano dei segnali di comunicazione tra la CPU e la periferica che permettono di capire quando il dato è "pronto" o meno. 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:** Si condividono le stesse modalità di una comunicazione con handshacking, con la differenza che la sincronizzazione delle due parti avviene mediante uno stesso clock;

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 i bus di collegamento classici, che collegano anche la memoria ecc. Ciò quindi mi limita nell'utilizzo degli indirizzi, poichè, 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

Quando devo selezionare la mia periferica a cui faccio riferimento, utilizzo una serie di indirizzi. Tali indirizzi possono essere utili al fine di realizzare i seguenti tipi di logica:

- **Logica tristate:** Logica che quando una periferica non vede il suo indirizzo sui bus adeguati smette di interagire con il sistema, quindi ignora la variazione dei dati sul bus. Tale logica, quindi utilizza l'indirizzo interno della nostra periferica.
- **Logica Plug-and-play:** L'indirizzo della periferica viene scelto in base ad una serie di indirizzi disponibili.

2.1.3 BUS

I bus sono i collegamenti che interconnettono le varie componenti di un calcolatore, ovvero, CPU, memoria e periferiche di I/O. In generale non vi è una tipologia unica di bus, ve ne sono varie in base alla tipologia di utilizzi e alla tipologia di tecnologie utilizzate. I bus, si contraddistinguono principalmente per la divisione che attuano sui loro collegamenti, ma in generale, le informazioni che vengono trasportate sono solitamente le stesse. Le informazioni, quindi, 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 capitare che la periferica che vado ad utilizzare non è ad-hoc per quella determinata tipologia di bus. Pertanto, quello che posso fare, è considerare l'utilizzo 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 programmi che permettono di capire come il processore vada ad utilizzare una determinata periferica. Le tipologie di approccio che si possono avere nella scrittura dei driver sono varie, la più primitiva è il polling. Il **Polling** è un modo con cui il processore va ad interagire con la periferica. In generale si va a dare un primo segnale di controllo alla periferica e si aspetta uno specifico valore di stato per poter accedere al dato. Tali sistemi sono altamente inefficienti, poiché mentre la CPU aspetta la risposta della periferica passano dei periodi di clock dove la CPU rimane in attesa. Il tempo che quindi la CPU rimane senza eseguire delle operazioni utili è detto **Busy-waiting**. Un possibile codice di implementazione del polling è [2.1]

```

1      ORG      $8000
2      *Inizializzo lo stato dei miei registri
3      MOVE.B   #$00,C
4      MOVE.B   #$00,S
5      *Vado a considerare la zona di memoria dove voglio salvare i dati
6      MOVEA.L   #VAR1,A0
7      MOVE.W   #0,DO
8      *Devo prelevare N dati quindi ciclo N volte
9      FOR      CMP.W   #N,DO
10     BGE     FUORI
11
12     *Qui devo scrivere il driver sapendo che devo ricevere un byte
13
14     MOVE.B   #$01,C  *Vado a settare un controllo
15     *Qui inizia il ciclo di polling dove attendo uno specifico valore
16     dello stato
16     L1      MOVE.B   S,D1
17     AND.B   #$80,D1  *Se il bit si e' alzato ho finito.
18           Altrimenti continuo ad aspettare
18     BEQ     L1
19
20     *Qui il dato e' stato letto, poiche' ho il flag di stato alzato
21     MOVE.B   D,(A0)+ *Inserisco il dato in memoria
22     MOVE.B   #$00,C  *Vado a resettare il segnale di Controllo
23     MOVE.B   #$00,S  *Vado ad "eludere" il sistema su un
24           segnale di stato
25
25     ADD.B   #1,DO          *Incremento il conteggio
26     BRA    FOR            *Ripeti
27 FUORI
28
29     ORG      $8100
30 D      DS.B    1      *Registro dato
31 S      DS.B    1      *Registro Stato
32 C      DS.B    1      *Registro Controllo
33
34 N      EQU     5      *Quantita' di valori da considerare
35 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 e C. Che per l'implmentazione di un driver riutilizzabile non è proprio la scelta corretta. Uno stesso dispositivo accessibile mediante diversi indirizzi avrebbe 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 problematiche (o criticità), possono essere affrontate in vario modo. Per la prima la soluzione è molto semplice, al posto di andare a 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, invece, al secondo problema c'è il bisogno di considerare le **interruzioni**. Mentre 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, ciò permette di poter gestire ed uscire da situazioni di eventuali guasti.

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 specifica impostazione e configurazione. Prima di parlare di più porti ci concentriamo sulle comunicazioni a singolo porto; in generale le comunicazioni che avvengono tra due interfacce della PIA sono configurabili tramite i bit di configurazione del chipset. Nel nostro caso, la comunicazione che maggiormente utilizzeremo è quella dotata di handshacking, per cui si avrà una configuraizone ed un collegamento simile all'immagine [2.2]. Il dispositivo PIA simulato in ASIM è derivato da quello commerciale MC6821. Questo è dotato di sei registri a 8 bit, tra cui: due registri per il trasferimento dei dati da e verso la periferica (*PRA* e *PRB*); due registri di controllo/stato (*CRA* e *CRB*); infine due registri per il controllo della direzione dei dati (*DRA* e *DRB*). Questi registri sono accessibili mediante indirizzamento interno secondo la tabella 2.1:

Questi registri sono selezionabili dal processore mediante le linee RS1 ed RS0. Nel dettaglio, i registri di controllo sono a 8 bit suddivisi come indicato in tabella 2.2:
Entrando nei dettagli del processo, bisognerà fare particolare attenzione ai seguenti registri:

- **CA1,CB1:** Sono registri che possono assumere solo direzione di ingresso e solitamente vengono usati come "lettori" di segnali SYN o segnali ACK da parte dell'altro dispositivo;

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

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

Tabella 2.2: Control Registers

- **CA2,CB2:** Sono i registri che possono essere configurati(sia di ingresso che di uscita), ed in generale, in base al protocollo che si vuole interpretare, vengono settati in una determinata modalità di funzionamento (dipendente dalla tipologia di protocollo che si vuole implementare);
- **Dato:** Il bus dati trasmette parallelamente i dati tra le due periferiche in base al protocollo di handshacking utilizzato. I bus dati sono unidirezionali ma la direzione è programmabile.

Per far sì che le due architetture possano comunicare è quindi importante definire come si andranno a collegare e quindi la direzione e l'interpretazione che bisogna dare ai registri. Una possibile architettura di collegamento è quella visibile all'immagine [2.2]. Le linee D0-D7 sono di interfacciamento con il processore e sono bidirezionali in base alla natura dell'operazione richiesta (lettura o scrittura). Poiché la PIA ha due porti distinti, può essere collegata al processore con due linee di interruzioni differenti denominate **IRQA** (porto A) e **IRQB** (porto B).

Una volta definita la tipologia di architettura si va a decidere come queste periferiche dovranno interagire tra loro (definizione del protocollo), per cui si va ad impostare uno specifico registro di configurazione che permetterà di impostare le seguenti opzioni:

- **Interrupt o polling:** A che livello di priorità si andrà ad impostare l'interruzione della PIA;
- **Modalità di funzionamento:** se attuo l'handshacking o altre tipologie di protocolli;
- **Lettura o scrittura:**

Una volta definita la struttura del registro di configurazione questo viene settato per impostare la PIA. Una volta impostato il modo di funzionamento si va a gestire il tut-

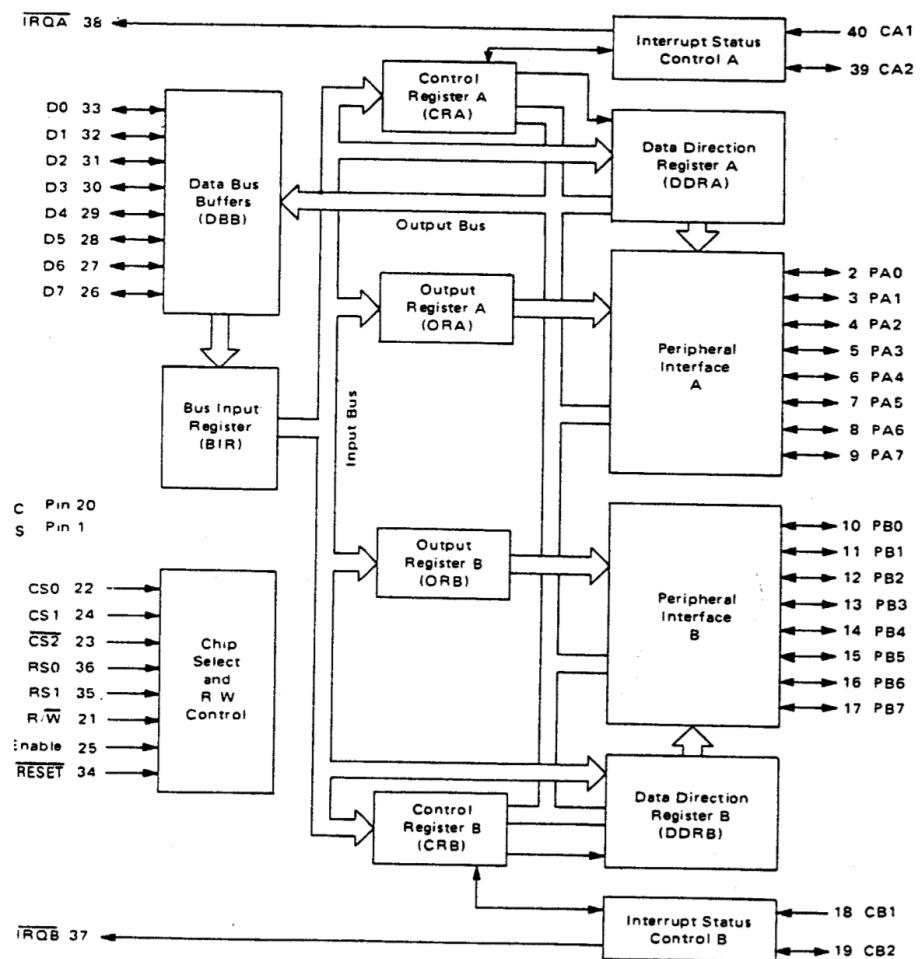


Figura 2.1: Architettura base della PIA

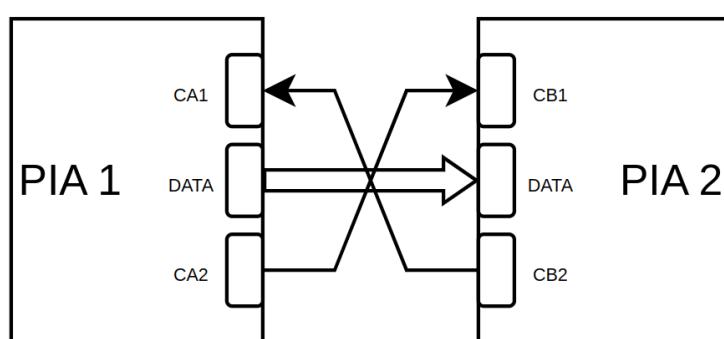


Figura 2.2: Collegamento tra due dispositivi tramite PIA

to. Quindi, se si è scelto un funzionamento tramite interrupt si avrà un certo tipo di comportamento, altrimenti se ne avrà un altro.

Il funzionamento generale della fase di **handshaking** tra due dispositivi PIA è gestita, per i nostri esempi, in un particolare modo. Per fare chiarezza andremo a dividere la fase di ricezione dalla fase di trasmissione. In fase di **trasmissione** le operazioni che si vanno ad effettuare sono le seguenti:

- **Inserimento del dato:** La periferica posiziona il dato sul bus dati collegato in maniera diretta con la PIA. Una volta effettuato l'inserimento, la periferica tramite la linea CA1 invia un segnale di *dato pronto* alla PIA, quindi sulla transizione di CA1, in accordo alla configurazione, si propaga un'interruzione al processore. Internamente, la periferica alzerà CA2 (in accordo al protocollo handshaking 100);
- **Attesa dell'ack:** Il sistema aspetta che l'ack si abbassi per capire che il dato è stato inviato correttamente. Una volta ottenuto l'ACK, la PIA abbassa (dopo una lettura fittizia) CA2, che viene interpretato dalla periferica come *registro dati libero*;
- **Lettura fittizia:** Dato che voglio inviare un nuovo dato, ho bisogno di abbassare il bit CRA7, ma non posso farlo in maniera diretta (configurando un apposito registro di controllo), poiché il bit CRA7 è buona norma che sia in sola lettura. Di conseguenza per abbassare tale valore vado ad effettuare una lettura a vuoto (o lettura fittizia), che mi permette di abbassare il bit CRA7 e la linea CA2 in modo da poter proseguire nelle operazioni;

Nella fase di ricezione, invece, si ha:

- **Attesa del dato:** Si attende che venga inviato un dato sul porto della PIA, tale attesa può essere effettuata o tramite Polling, andando a controllare in continuazione il valore del bit CRB7, o attivando un'interruption diretta all'arrivo dell'evento sul bit CB1.
- **Ricezione del dato:** Una volta compreso che vi è un dato pronto, si effettua una lettura del dato, ed in automatico, la PIA abbasserà il bit CRB7 ed invierà un segnale di ACK, abbassando il valore in CB2
- **Fine:** Nel caso di polling, il sistema continuerà ad aspettare nuovi dati andando a controllare il bit CRB7. Mentre nel caso delle interrupt continuerà il suo normale flusso di esecuzione

Andando ad analizzare il codice delle varie operazioni è importante capire la metodologia di funzionamento del sistema, sia per costruire l'architettura in maniera adeguata che per scrivere i driver in maniera corretta.

2.1.5.1 Configurazione della PIA

La PIA prima di essere utilizzata, richiede una propria configurazione, in parte dipendente dall'architettura e da una parte inherente al driver che si deve implementare. Per la parte hardware dobbiamo scrivere (almeno per le nostre simulazioni in ASIM), il **file di configurazione** che definisce tutti gli specifici collegamenti tra i dispositivi con gli specifici significati. Mentre nel caso di configurazione del driver, una volta definita l'architettura su cui si andrà ad eseguire il codice, si vanno ad impostare i registri di controllo e dato, in modo da poter utilizzare il nostro dispositivo (sia con il polling che con interrupt).

Definizione dei registri di controllo e dato

Il registro di controllo ed il registro di stato vengono inizializzati all'interno della configurazione. Quindi prima di iniziare a scrivere il driver bisogna osservare per bene il file di configurazione. Più precisamente dobbiamo osservare i seguenti due parametri della nostra PIA:

- **Address 1:** Indirizzo su cui è mappata la PIA;
- **Address 2:** Indirizzo del registro di controllo della PIA (dove saranno inserite le configurazioni).

Una volta osservati tali parametri si inizializzano i registri di dato e controllo nel seguente modo:

```
1 PIADB      EQU      $2006 *indirizzo del registro dato  
2 PIACB      EQU      $2007 *indirizzo del registro di controllo
```

Tali registri saranno utilizzati all'interno delle ISR per poter controllare/comunicare con la PIA, sia per configurarla (in una fase iniziale). Più precisamente andiamo ad effettuare le seguenti due operazioni:

- **Configurazione:** dove si imposta il registro di controllo in base a quello che si vuole effettuare, quindi viene impostato nelle fasi iniziali del driver;
- **Gestione:** Vengono definite le modalità di funzionamento in base alla tipologia di filosofia adottata.

Un main che è uguale in entrambe le tipologie di comportamento (interruzioni e polling) è il seguente:

```
1 *MAIN  
2 MAIN JSR      DVROUT *Configurazione della PIA
```

Per la fase di configurazione, la procedura rimane la stessa a meno della maschera, pertanto viene riportato qui il caso con maschera e saranno riaffrontate le due differenti gestioni all'interno degli appositi paragrafi [2.1.5.2] e [2.1.5.3].

```
1 DVROUT      MOVE.B    #0,PIACB      *seleziona il registro  
               direzione di PIA porto B  
2 MOVE.B      #$FF,PIADB      *accede a DRB e pone DRA=1 : le  
               linee di B sono linee di output  
3 MOVE.B      #%00100101,PIACB      *imposta il registro di controllo  
               in base alla sua mappatura  
4 *           ; i bit CRB7 e CRB6 sono a sola lettura  
5 RTS
```

Per capire meglio come impostare il registro di controllo, riportiamo di seguito un'analisi più approfondita dei bit dei registri di controllo esposti nella tabella 2.2: I bit 0,1 -> di CRA (o CRB) servono a controllare il flag di interruzione IRQA1 (IRQB1) in posizione 7 nella parola di stato-controllo. Lo stato del flag IRQA1 si propaga sulla linea di interruzione IRQA verso il processore generando un'interruzione. In particolare, il bit 0 stabilisce se le interruzioni vengono propagate al processore (valore 1) o se vengono *mascherate* (valore 0), mentre il bit 1 serve a stabilire se l'interruzione viene propagata sul fronte di salita del segnale low->high (valore 1) o sul fronte di discesa high->low (valore

0) di CA1. Per i bit 3,4,5 di CRA (o CRB) occorre fare una distinzione: se la linea è stata programmata come linea di ingresso (settando il bit 5 a 0), si comporta come i bit 0,1 con b3 nella funzione di b0 e b4 nella funzione di b1; se la linea è stata programmata come linea di uscita (settando il bit 5 a 1), CA2 (CB2) permette di controllare la periferica tramite la linea CA2. Sono previsti 3 possibili modi di sincronizzazione codificati con i bit b4 e b3, ovvero **100 -> Modo Handshake**, **101 -> Modo impulsivo** (l'abilitazione di CA2 segue il profilo dell'impulso di un clock) e **11x -> Modo dipendente dal bit 3** (ovvero CA2 alto o basso in base a come viene manualmente settato il bit 3).

2.1.5.2 Gestione PIA senza Interrupt

La gestione senza interrupt sfrutta la tecnica del polling per monitorare volta per volta la struttura dei registri di stato, che quindi mi permette di capire quando agire sul dato o meno. Per effettuare il polling ho bisogno di configurare il registro di controllo in modo da non far attivare le interrupt. Seguendo lo schema presente alla fine del paragrafo [2.1.5.1], dovrò impostare come controllo la seguente sequenza: 00100100, dove indichiamo che non vi è bisogno dell'uso delle interrupt. Questa sequenza significa che le interrupt non vengono propagate al processore (b0=0), con le linee AD1 e AD0 si accede al registro dati (b2=1), il protocollo scelto è l'handshaking (b5b4b3 = 100).

Prendendo in considerazione la prima parte del main, quindi, vado a definire come subroutine di configurazione la seguente schematica:

```

1 DVBOUT MOVE.B #0,PIACB      *seleziona il registro direzione di
2                   PIA porto B
3 MOVE.B #$FF,PIADB          *accede a DRB e pone DRA=1 : le linee
4                   di B sono linee di output
5 MOVE.B #%00100100,PIACB    *imposta il registro di controllo
6                   come indicato precedentemente
7                   *i bit CRB7 e CRB6 sono a sola lettura
8 RTS

```

Una volta impostata la periferica, il polling viene effettuato sul registro di controllo. Di seguito vi è l'esempio di un invio di un dato, dove il controllo sull'ack viene effettuato all'interno di un ciclo, senza considerare l'utilizzo delle interrupt.

```

1 ORG $8200
2 MAIN JSR DVBOUT *inizializza PIA
3
4 MOVEA.L #PIACB,A1 *indirizzo registro di controllo CRB
5 MOVEA.L #PIADB,A2 *indirizzo registro PRB
6 MOVEA.L #MSG,A0 *indirizzo area messaggio
7 MOVE.B DIM,DO *dim del messaggio
8
9 CLR D1 *appoggio
10 CLR D2 *contatore elementi trasmessi
11
12
13 INVIO MOVE.B (A2),D1      *lettura fittizia da PRB =>
   serve per azzerare CRB7 dopo il primo carattere, altrimenti
   resta settato con l'ack

```

```

14      MOVE.B (A0)+,(A2) *carattere corrente da trasferire su
           bus di PIA porto B: dopo la scrittura di PRB, CB2 si
           abbassa
15      *           *cio' fa abbassare CA1 sulla porta gemella dell'
           altro sistema generando
16      *           *un'interruzione che coincide con il segnale DATA
           READY
17      ADD.B #1,D2          *incremento contatore elementi
           trasmessi
18
19 ciclo2  MOVE.B (A1),D1    *In attesa di DATA ACKNOWLEDGE
20      ANDI.B #$80,D1      *aspetta che CRB7 divenga 1
21      BEQ ciclo2
22
23      CMP D2,DO *controllo se ho finito di trasmettere
24      BNE INVIO
25
26 LOOP     JMP LOOP   *ciclo caldo dove il processore ha completato
           la trasmissione

```

2.1.5.3 Gestione PIA con Interrupt

La gestione della pia con il funzionamento dell'interrupt va a sfruttare il meccanismo delle interrupt autovettorizzate, per cui oltre a definire il registro di controllo in un certo modo, dobbiamo caricare all'interno dell'area degli indirizzi autovettorizzati, l'indirizzo della nostra ISR (che si traduce in ASIM nel caricare il file di configurazione della memoria fornito dal professore). Il caricamento dei dati non fa altro che inserire l'indirizzo di memoria della nostra ISR all'interno dell'apposita cella identificata. Come prima cosa definiamo il registro di configurazione come: 00100101

Il codice che va a configurare il registro di controllo è il seguente:

```

1 DVAIN MOVE.B #0,PIACA    *mette 0 nel registro controllo cosi'
           al prossimo accesso a PIADA (indirizzo pari)
2      *           *selezionera' il registro direzione del porto A
3      MOVE.B #$00,PIADA    ;accede a DRA e pone DRA=0 : le
           linee di A sono linee di input
4      MOVE.B #%00100101,PIACA  ;imposta il registro di
           controllo come indicato sopra, ponendo IRQA1=1 e IRQA2
           =1
5      *           ;i bit CRA7 e CRA6 sono a sola lettura
6      RTS

```

Oltre alla configurazione, andiamo ad attivare il meccanismo delle interrupt all'interno del processore, ed andiamo ad impostare la modalità utente, in modo da poter visualizzare anche che le ISR vengono eseguite sempre in modalità supervisore. Per capire meglio tale passaggio, di seguito vi è il MAIN:

```

1 MAIN  JSR DVAIN *inizializza PIA porto A
2
3      MOVE.W SR,DO *legge il registro di stato
4      ANDI.W #$D8FF,DO *maschera per reg stato (stato utente,
           int abilitati)

```

```

5      MOVE.W  D0,SR *pone liv int a 000
6
7 LOOP    JMP  LOOP   *ciclo caldo dove il processore attende
           interrupt

```

Una volta scritto il main ed aver fatto tutte le dovute configurazioni, possiamo osservare la scrittura del driver, che avrà il suo indirizzo di inizio caricato nel sistema delle interrupt autovettorizzate, che in questo caso sarà 8700:

```

1          ORG $8700
2
3 INT3      MOVE.L  A1,-(A7)      ;salvataggio registri che saranno
           utilizzati
4          MOVE.L  A0,-(A7)
5          MOVE.L  D0,-(A7)
6
7          MOVEA.L #PIADA,A1
8          MOVEA.L #MSG,A0 *indirizzo area di salvataggio
9          MOVE.B  COUNT,D0 *contatore corrente degli elementi
           ricevuti
10
11
12          MOVE.B  (A1),(A0,D0) *acquisisce il carattere e lo
           trasferisce in memoria
13 *          *la lettura da PRA fa abbassare CRA7 e CA2 => nell'
           altro sistema si abbassa CB1
14 *          *cio' corrisponde all'attivazione di CRB7 che funge
           da DATA ACKNOWLEDGE
15
16          ADD.B #1,D0
17          MOVE.B  D0,COUNT
18
19          MOVE.L  (A7)+,D0      *ripristino registri
20          MOVE.L  (A7)+,A0
21          MOVE.L  (A7)+,A1
22
23          RTE

```

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 [2.3]. In generale non è corretto parlare solo di interruzioni, poiché tale termine non comprende o non può comprendere 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

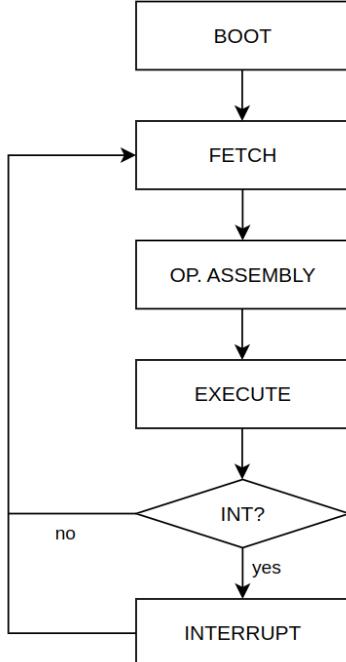


Figura 2.3: Ciclo di esecuzione con interrupt

- **Eccezioni:** Funzionano come le interruzioni, con la differenza che vengono scatenate internamente rispetto al processore, quindi non vengono gestite dai dispositivi ma dal programma stesso, tale condizione fa eseguire comunque una ISR, con l'obiettivo di dover gestire particolari casistiche (es. divisione per 0)

Quindi quando le interruzioni sono scatenate vanno ad effettuare una chiamata a subroutine particolare, tale chiamata è detta ISR (Interrupt-Services-Routine). Tale situazione, quindi, ferma il sistema dalla sua normale esecuzione del programma per dare priorità alla gestione dell'interruzione. Questo, quindi, apre molti dubbi su come gestire lo stato in cui si trova la macchina, poiché 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 salvari 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: **Descrittore di processo**, tali registri, quindi, descrivono lo stato di funzionamento del mio processore quando poi è stato prelazionato dalla mia ISR. Ciò mi permette di proseguire ancora con la normale esecuzione del programma prefissato.

2.1.6.1 Gestione delle Interruzioni

Una volta definito cosa sono le interruzioni è di fondamentale importanza capire come il processore le gestisce. Le principali modalità di gestione delle interruzioni sono due e sono:

- **Vettorizzate:** Ogni livello di priorità di interrupt è collegato al processore. I fili di collegamento per le interrupt sono limitati, quindi più dispositivi possono collegarsi sullo stesso cavo di interrupt. Il processore, quindi, per identificare il dispositivo che

ha scatenato l'Interrupt va a controllare i bus, su cui il dispositivo ha caricato il suo codice identificativo. Identificare il dispositivo, vuol dire identificare la tipologia di ISR da andare ad utilizzare. Gli indirizzi degli entry-point delle varie periferiche sono memorizzati in memoria a partire dall'indirizzo 0 a seguire per 256 locazioni di 4 byte. Tali locazioni si dividono nel seguente modo:

- **Funzioni speciali:** Da 0 a 24, gli entry-point identificano delle funzioni speciali o di gestione aritmentica
- **Interruzioni autovettorizzate:** da 25 a 31 sono indicizzate le locazioni per il funzionamento autovettorizzato
- **Trap:** da 32 a 47 sono indicizzate le funzioni per la gestione dei Trap
- **Utilizzabili:** da 48 a 256 sono locazioni disponibili per l'inserimento degli entry-point per la gestione di diverse periferiche
- **Autovettorizzate:** A differenza del caso vettorizzato, evita la lettura del codice identificativo, poichè ogni livello di interrupt è collegato al vettore delle ISR autovettorizzate e permette di selezionare in maniera "ignorante" l'ISR alla locazione della tipologia di priorità inserita

2.1.6.2 PIC

In generale, nel caso di sistema **vettorizzato**, viene in aiuto il componente **PIC (Programmable Interrupt Controller)** [2.4]. 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** 2.5, 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.

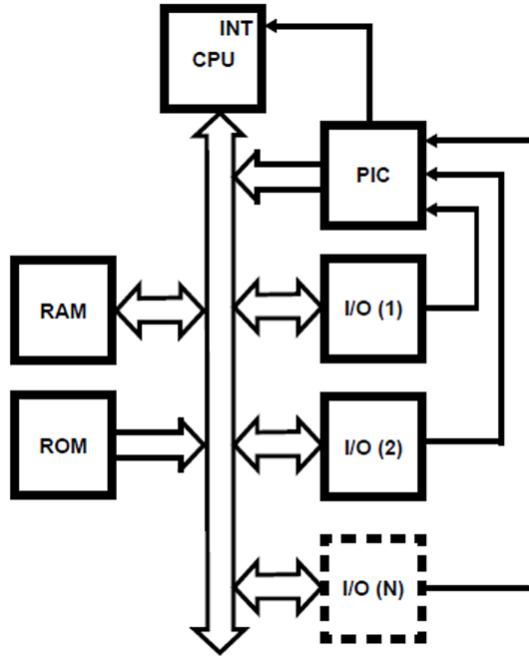


Figura 2.4: PIC

Osservando i componenti interni in figura 2.5, 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: 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 eventualmente la disponibilità a servire l'interruzione. Una volta ricevuto l'ACK, il PIC setta il bit

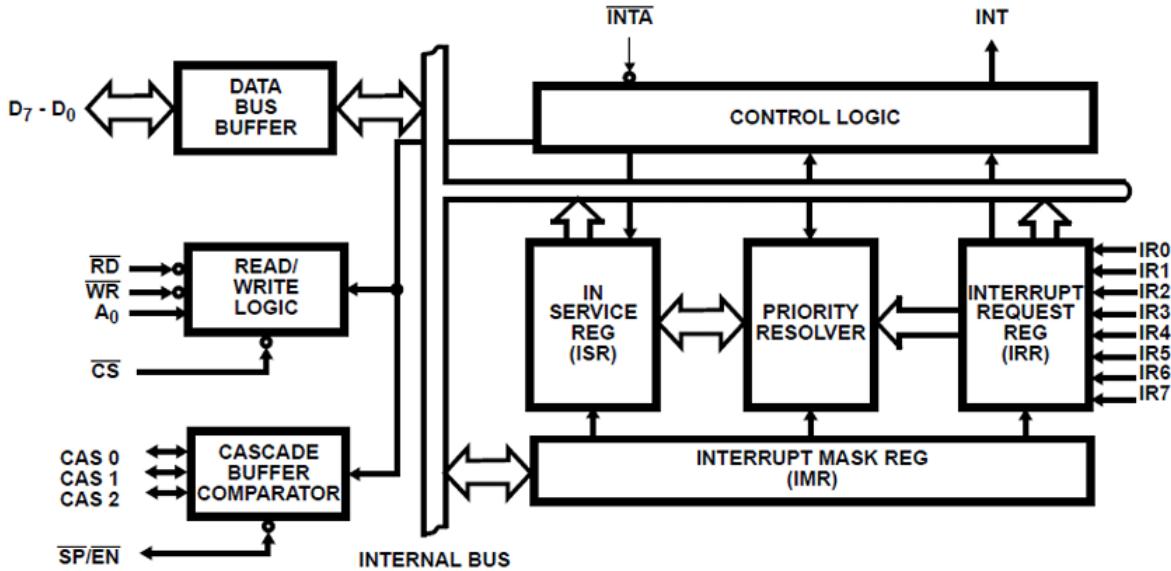


Figura 2.5: 82C59A

del registro **ISR** corrispondente alla linea interrompende 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 interrompende 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 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} , A0, \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 ($A_0=0, D_4=1$), 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 $A_0=1$, 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 $A_0=1$, questa parola viene letta solo quando c'è più di un 82C59A in cascata, informazione acquisita attraverso la ICW1.
 - **ICW4**: Identificata da $A_0=1$, 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 2.6. 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.

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

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 linea interrompente. 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 corri-

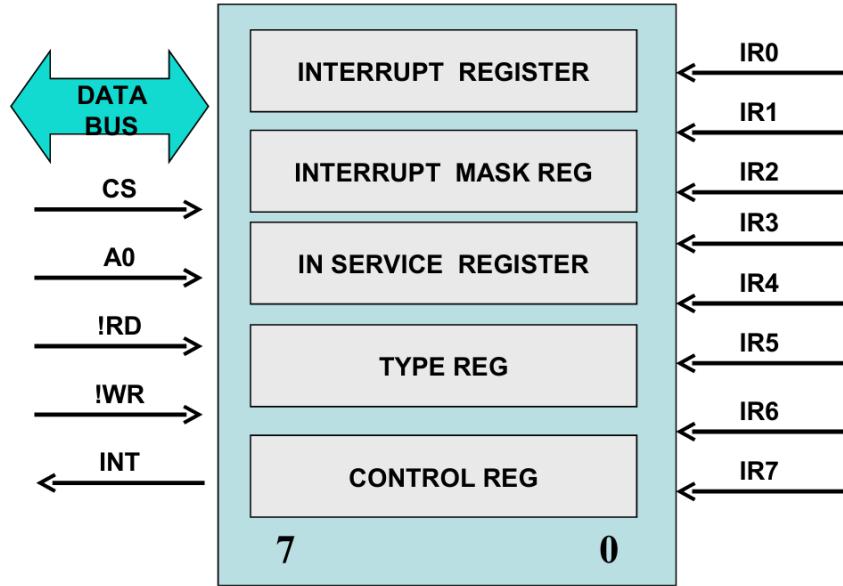


Figura 2.6: PIC semplificato

spondente. 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 (2.7).

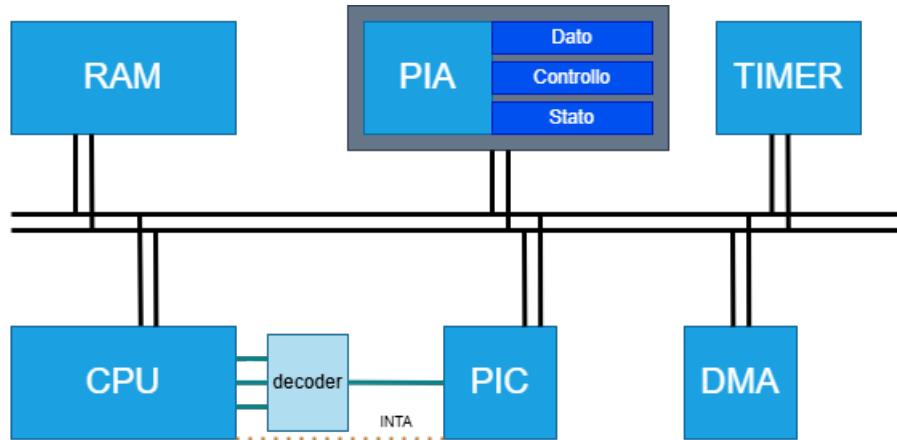


Figura 2.7: Modello IO esteso - schema logico

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. 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 su 3 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 bisogna interrompere il trasferimento;
- **Registro Identificativo:** Tramite tale registro si vanno ad identificare:
 - 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 tipologie di 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 dato, ma anche tramite vari bus di controllo, che permettono al processore ed al DMA di poter comunicare e coordinarsi.

Il dispositivo a cui si fa riferimento nelle esercitazioni del corso è l'**Intel 8237**, che per la sua architettura ha 4 canali distinti (quindi è in grado di gestire 4 trasferimenti alla volta). Nella sua 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, ovvero:

- **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 fine del trasferimento (e quindi la disponibilità del BUS);
- **On Demand**: Simile alla modalità Block, con l'unica differenza che il trasferimento può essere interrotto dal processore, per poi essere 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 in modo da poter gestire più di 4 canali.

Oltre al minor numero di canali, il componente simulato in ASIM non supporta tutte le modalità sopra-citate, difatti le modalità utilizzabili in ASIM sono **Single** e **Block**.

Guardando la figura [2.8], abbiamo il modello architettonale del componente realizzato in ASIM, di cui i registri posti sulla sinistra sono di comunicazione con il processore, mentre i segnali sulla destra sono quelli che vengono utilizzati per l'interfacciamento con le periferiche collegate ai corrispettivi canali.

Per la comunicazione con il processore, il significato che hanno i segnali rappresentati è il seguente:

- **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 da considerare;
- **IOR** e **IOW**: Segnali di gestione della lettura e della scrittura sul componente e per le periferiche;
- **MEMR** e **MEMW**: Segnali di gestione della lettura e della scrittura sui dispositivi di memoria;
- **CLK** e **Reset**: Classici segnali di clock (tempificazione delle operazioni) e reset (dei registri del dispositivo);
- **HRQ**: Segnale di richiesta del controllo del sistema BUS, che solitamente è collegata all'ingresso HOLD della CPU;

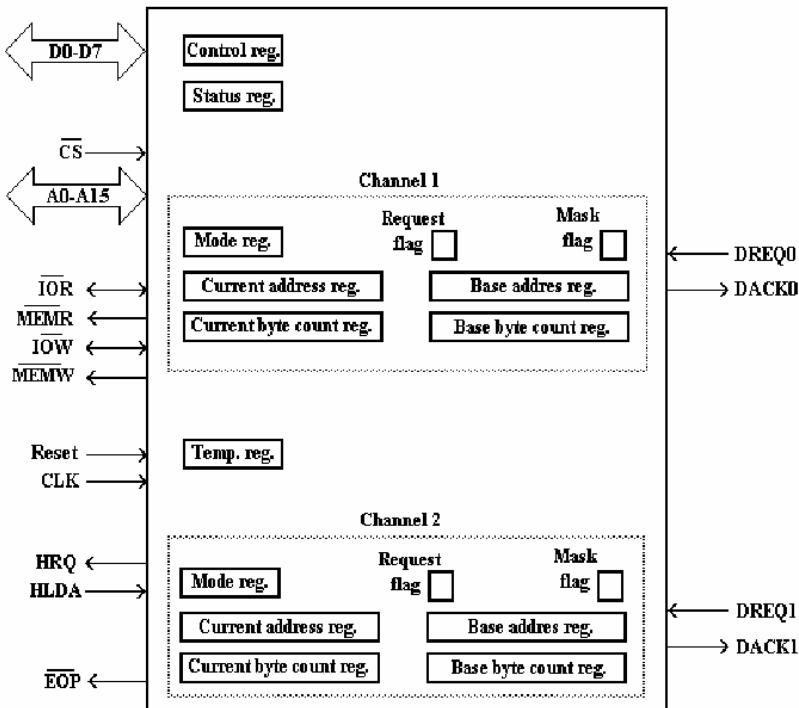


Figura 2.8: Modello di programmazione DMA a 2 canali

- **HLDA:** Segnale proveniente dalla CPU che segnala l'acquisizione del sistema BUS da parte del processore;
- **EOP:** Linea di interruzione *bidirezionale* che bisogna collegare al processore per segnalare il completamento del trasferimento richiesto.

Dati i due differenti canali si avranno 2 periferiche collegate allo stesso dispositivo DMA. Pertanto, la comunicazione, potrà essere effettuata da una sola di queste periferiche per volta. Tale decisione viene effettuata secondo un ordine di priorità, per cui il dispositivo collegato ai terminali 0 avrà una priorità maggiore rispetto a quello collegato ai terminali 1. I segnali che gestiscono le periferiche sono:

- **DREQ0 e DREQ1:** Che sono due segnali che utilizzano le periferiche per richiedere l'accesso al BUS tramite dei cicli DMA;
- **DACK0 e DACK1;** Sono due segnali che permettono di comunicare alla periferica (da parte del DMA) la disponibilità a soddisfare una richiesta di accesso tramite DMA al bus.

2.2.1 Utilizzo effettivo in ASIM

Per capire al meglio come utilizzare il DMA, bisogna, in una prima fase, capire cosa tale DMA dovrà fare ed in che modo esso dovrà lavorare. Per definire queste cose vi sono i registri di controllo (unico per tutti i canali) ed i registri di Modo (uno per ogni canale). Quando si va a dischiarare il componente all'interno del file di configurazione, si vanno a definire due indirizzi (address 1 ed address 2), che permettano di scandire tra di loro

sedici locazioni. Ciò ci serve per permetterci di poter accedere ai vari registri presenti all'interno del DMA, andandoli ad astrarre in registri di memoria (Memory-mapped). Precisamente, il funzionamento è gestito dal decodificatore collegato sul pin \overline{CS} , che va ad attivare il pin quando rileva sul bus indirizzi un indirizzo appartenente all'insieme [address1, address2]. All'interno del file di configurazione vi sono anche le altre voci, che identificano specificatamente come il componente si collega ai vari BUS e ai vari componenti. Una buona architettura di tale sistema è quella che si può osservare nella figura [2.9]

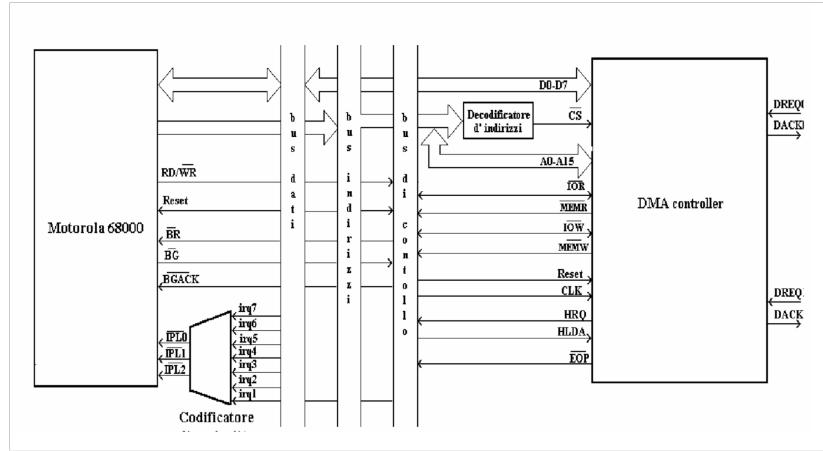


Figura 2.9: Montaggio del componente DMA rispetto al sistema generale

Una volta compresa l'architettura del sistema, possiamo iniziare a comprendere in che modo andare a configurare il funzionamento del DMA. Precisamente andiamo a vedere in primis il significato associato ai bit dei vari registri interni al componente. Il primo registro che si può andare ad approfondire è il registro Mode, di cui ne sono presenti due, uno per ogni canale. Nella tabella 2.3 sono illustrati i significati associati agli 8 bit del registro.

Bit	Significato
0	Instrada il dato su un canale, 0 per il canale 0 ed 1 per il canale 1
1-2	Non utilizzati
3	Indica la direzione di trasferimento: 0 -> Da memoria ad interfaccia, 1 -> Da interfaccia a memoria
4	Abilita la Autoinizializzazione , e quindi se impostato a 1, quando il DMA termina il "conteggio" resetta i suoi registri con i reciproci registri di base
5	Definisce se vogliamo effettuare l'incremento di CADDR (caso di 0) oppure vogliamo impostare il decremento (caso di 1)
6	Non Utilizzato
7	Imposta la modalità di trasferimento: 0 -> Single, 1 -> Block. Ad ogni funzionamento si ha una gestione differente del BUS (nel caso block solo alla fine del trasferimento)

Tabella 2.3: Significato dei bit dei registri MODE

Oltre alla configurazione del singolo canale, devo anche impostare il registro di controllo dell'intero DMA controller.

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 da memoria a memoria, l'indirizzo sorgente rimanga costante
7	Abilita il DMA controller

Tabella 2.4: Significato dei bit dei registri CTRL

2.2.2 Implementazione in Motorola 68k

Le implementazioni in ASIM con motorola 68k sfruttano varie architetture (descritte dai file di configurazione). Una volta definita l'architettura si va a definire in che modo bisogna gestire i registri e l'utilizzo delle periferiche (Come il DMA). Pertanto, bisogna distinguere i differenti utilizzi del DMA, per cui si divideranno le varie casistiche.

2.2.2.1 Caso Memoria-Memoria

Nel caso memoria-memoria, il DMA richiede una particolare definizione. Le prime cose che bisogna fare sono: definizione dell'area dati (dove prendere e dove trasferire i dati); definizione degli intervalli per l'accesso ai vari registri interni del DMA.

```

1 * Area dati
2           ORG      $9500
3 origine      DC.B     '0123456789 messaggio da trasferire'
4 destinazione DS.B     34
5
6 * Definizione di indirizzo ed offset per l'accesso ai registri
   del DMA
7 dma    EQU      $2010
8 caddr0  EQU      0
9 caddr1  EQU      2
10 ccount1 EQU      3
11 cntrl   EQU      8
12 mode    EQU      11
13 reset   EQU      13
14 clearmf EQU      14
15 writeamf EQU      15
16 nbytes  EQU      34

```

Una volta definita tutta la parte "statica" ovvero comprendente solo i dati utili per le configurazione, si passa alla parte operativa, dove si va a configurare effettivamente il dispositivo e si va ad avviare il suo funzionamento:

```

1      ORG    $8200
2      * Carico l'indirizzo del DMA
3      MOVE.W    #dma ,A0
4
5      * Carico l'indirizzo di origine dei dati all'interno del
6      primo registro indirizzo
7      MOVE.W    #origine,caddr0(A0)
8
9      * Carico l'indirizzo di destinazione dei dati all'interno
10     * del secondo registro indirizzo
11     MOVE.W    #destinazione,caddr1(A0)
12
13     * Carico il numero di caratteri da cui e' composta
14     * la stringa all'interno del secondo counter
15     MOVE.B    #nbyte,ccount1(A0)
16
17     * Imposto una comunicazione (in base ai bit) che
18     * dev'essere block, con incremento ed
19     * definendo una comunicazione memoria-interfaccia per il
20     * primo canale
21     * Sto settando il primo canale utilizzando un VALORE pari
22     MOVE.B    #\$90,mode(A0)
23
24     * Faccio una configurazione eguale a quella di prima solo
25     * che vario
26     * il bit meno significativo per settare tramite VALORE
27     * il secondo canale di comunicazione
28     MOVE.B    #\$91,mode(A0)
29
30
31     * Vado a settare l'ultimo registro, quello di controllo
32     * che permette al DMA di poter iniziare a lavorare
      MOVE.B    #\$A0,cntrl(A0)
33
34
35     * Ciclo di attesa che simula lavoro del processore
36 loop   JMP    loop

```

Una volta impostato il DMA e fatto partire, bisogna gestire l'interruzione che sarà scatenata quando esso avrà finito di lavorare. (pertanto, dato che stiamo lavorando con le interrupt, nel caso di asim, dobbiamo impostare la memoria in modo da inserire l'indirizzo della ISR all'interno della tabella delle ISR autovettorizzate). L'ISR che si va a scrivere per il DMA, è una ISR che ha come unico scopo quello di resettare il DMA. Per cui si avrà:

```

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

```

2.3 USART (Universal Synchronous-Asynchronous Receiver-Transmitter)

L'USART (Universal Synchronous-Asynchronous Receiver-Transmitter) è un interfaccia di comunicazione seriale. In generale i sistemi di comunicazione seriale utilizzano solo 1 filo per comunicare (Tx -> 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, poichè 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 nel precedente paragrafo, l'USART (a differenza del suo predecessore, l'UART), prevede 2 modalità di funzionamento:

- **Sincrona**
- **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 di tipo Sincrona, i due dispositivi (per una comunicazione unidirezionale) condividono 2 collegamenti:

- Tx -> Rx
- clk_t -> 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 Tx -> 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**, uno standard che definisce in che

modo le varie interfacce possono 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 tutto, originariamente erano un calcolatore ed il suo modem. Precisamente, i pin che sono presentati ed utilizzati nello standard sono:

- **RD:** Sarrebbe il pin di ricezione di dati seriali
- **TD:** Sarebbe il 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 sue dispositivi in modo da poter trasmettere il segnale senza intoppi di tipo elettrico
- **DTR:** Data Terminal Ready, indica il segnale di uscita per poter finire/iniziare una fase di handshaking (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", solitamente tale pin va a scatenare una interruzione per poter far gestire la comunicazione

I segnali **DTR** e **DSR**, sono pin che vanno ad implementare l'handshaking 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 handshaking 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 [2.10]

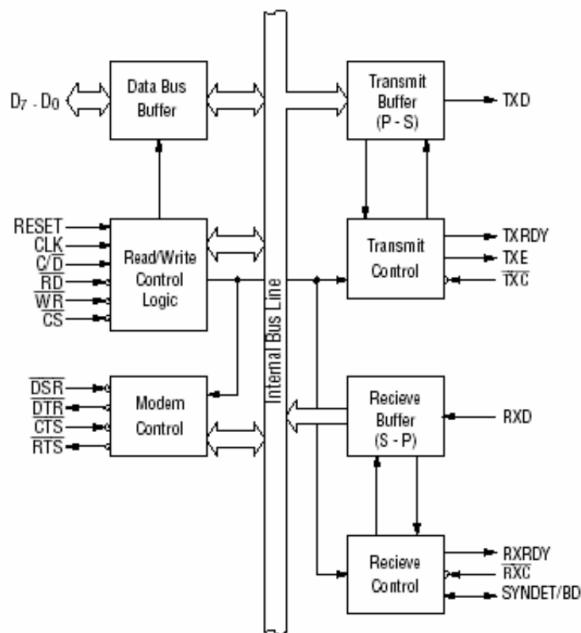


Figura 2.10: Struttura dell'Intel 8251A

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:** Componente di controllo dell'intero sistema, va a creare un interfaccia di utilizzo degli elementi interni del dispositivo per la CPU. Vedendo meglio 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, difatti 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 un nuovo dato
 - **TXE (Transmitter Empty)**: Segnale di uscita per identificare che il buffer di trasmissione può essere sovrascritto (Attesa del nuovo dato)
 - **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. Esso riceve i dati tramite il segnale esterno RXD. Oltre a tale segnale ne ha altri che ne fanno il controllo e la gestione della comunicazione
- **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 secondo 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, nel caso di comunicazioni sincrone, ricevere i caratteri di sincronismo, mentre nella modalità sincrona segnala la rilevazione di una congizione di break

2.3.2.1 Intel-8251A in Asim

Come detto in precedenza, l'Intel 8251A è un dispositivo programmabile, quindi richiede di 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 [2.11]

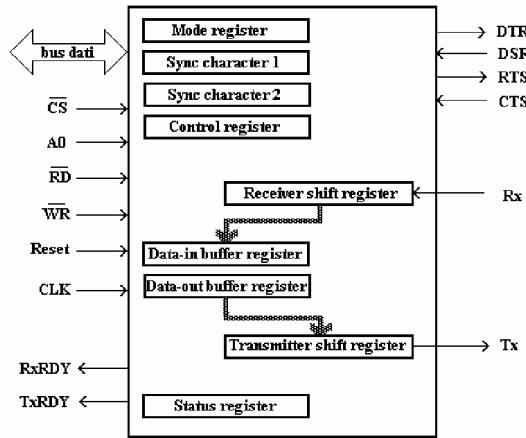


Figura 2.11: Modello di programmazione dell'8251A simulato

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, vado ad inserire il componente nella mia architettura simulata, quindi lo vado ad inserire all'interno del file di configurazione (.cfg). In generale, come anche per prove precedenti, tali file vengono passati da terze parti, nel caso si volessero configurare *from scratch* si rimanda al capitolo A.1.4. I registri con cui ci si va ad interfacciare con il motorola 68K 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 [2.12].

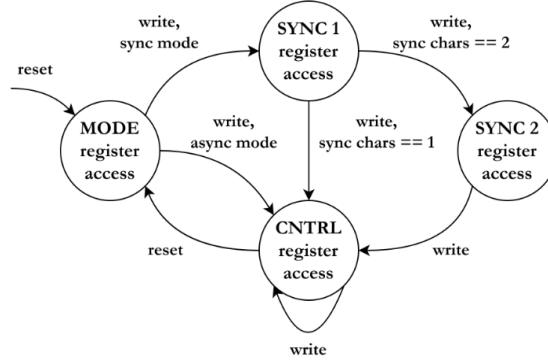


Figura 2.12: Ordine di accesso all'indirizzo dispari

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 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) [2.6,2.7,2.8] e le modalità di accesso ai differenti registri [2.5]

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 2.5: 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 2.6: Significato dei bit del registro MODE

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 2.7: 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 2.8: Significato dei bit del registro STATUS

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.

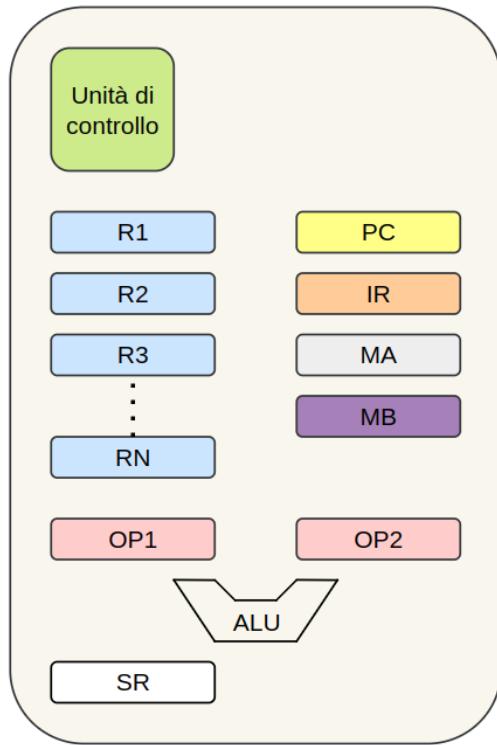


Figura 3.1: Architettura di un processore generico

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 far 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 livello di Processo:** Ho a disposizione più processori (parallelismo esplicito) che vanno ad eseguire in maniera concorrente tali processi;
 - **Parallalismo livello 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.

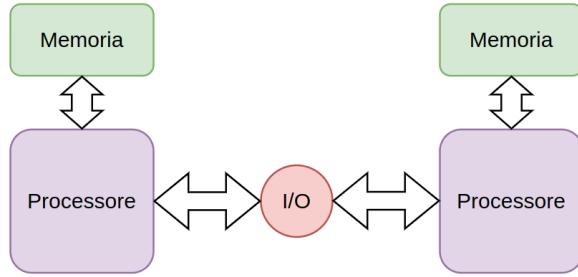


Figura 3.2: Sistemi "gemelli" o sistema multicompiler

- **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.

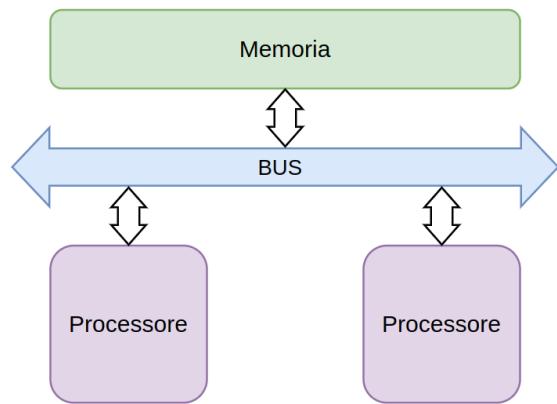


Figura 3.3: Sistema a memoria condivisa

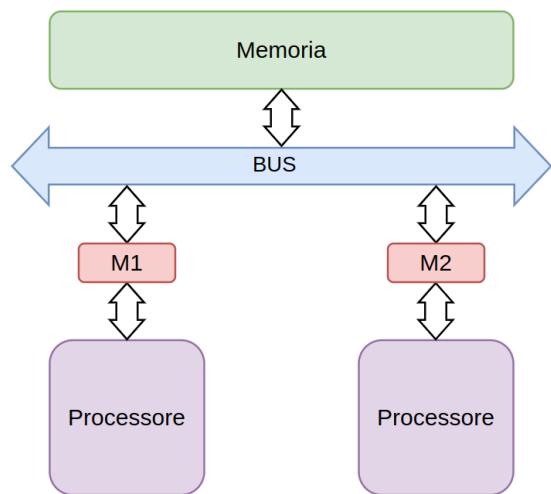


Figura 3.4: Sistema multicore

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].

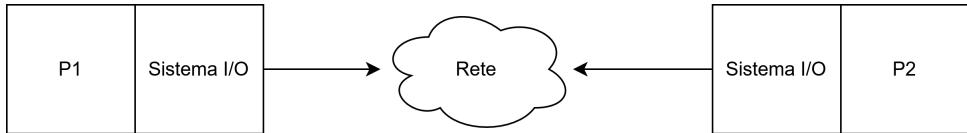


Figura 3.5: Architettura di un sistema Multi-Computer.

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

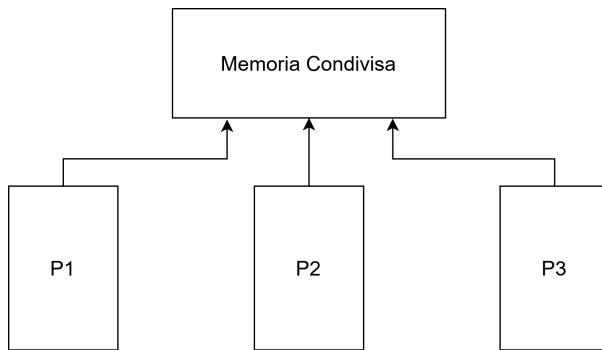


Figura 3.6: Architettura di un sistema Multi-Processore.

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 4×4 , 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 .

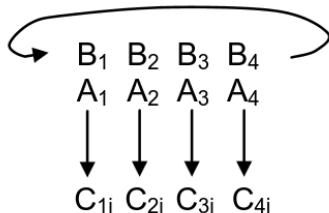


Figura 3.7: Calcolo parallelo della matrice C su 4 nodi.

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**

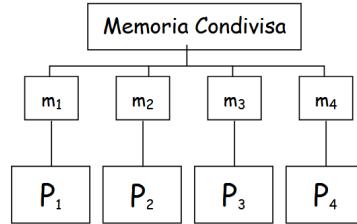


Figura 3.8: Sistema multiprocessore con gerarchia delle memorie.

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 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

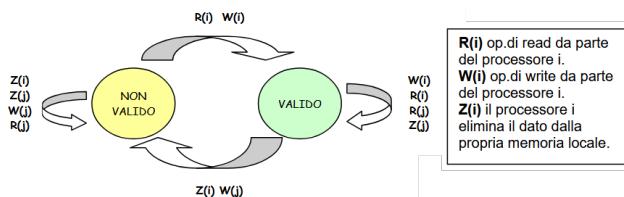


Figura 3.9: Automa per la gestione della coerenza in write through.

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.

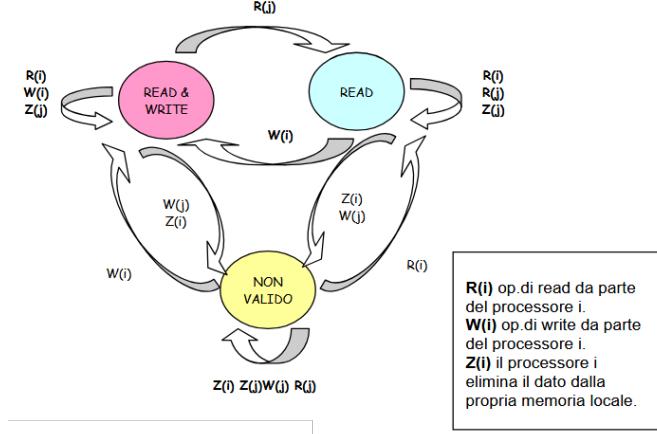


Figura 3.10: Automa per la gestione della coerenza in write back.

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.

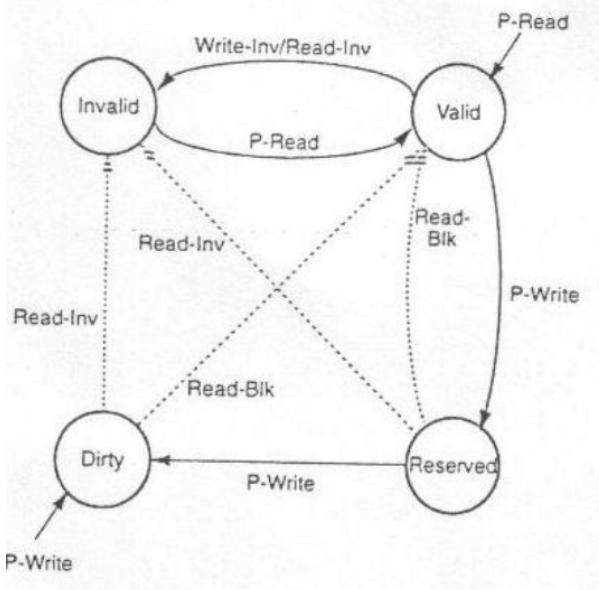


Figura 3.11: Automa per la gestione della coerenza in write once.

3.3 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 [3.12]):

- **Instruction 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.

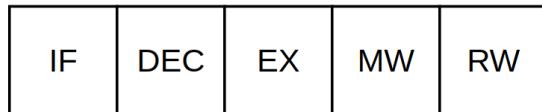


Figura 3.12: Architettura classica pipeline (ordine di esecuzione fasi)

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 i1, una volta prelevata, l'istruzione i1 passa alla fase di decode, mentre, allo stesso istante, viene caricata l'istruzione i2 (quindi viene effettuata la Instruction 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 [3.13]).



Figura 3.13: 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*) (3.3.4);
- **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, poiché 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**; poiché 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, poiché 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.

3.3.1 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 registi**: 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.

3.3.2 Architettura del MIPS

Il MIPS (Microprocessor without Interlocked Pipelined Stages) è una tipologia di architettura Pipelined di tipo RISC. Nel precedente capitolo abbiamo visto le fasi di esecuzione di un'architettura pipelined generale [3.12], però, nel caso del MIPS, tali fasi variano leggermente. Difatti il MIPS è caratterizzato dalle seguenti fasi:

- **Istruction Fetch**: Prelievo dell'istruzione dalla memoria e incremento del program counter;

- **Decode:** Si vanno a prelevare gli operandi dal *register file* e si preparano i segnali di controllo per la fase di esecuzione. Il register file è una componente fondamentale all'interno di un processore. Si tratta di un insieme organizzato di registri che servono per archiviare temporaneamente dati e istruzioni durante l'esecuzione dei programmi. Supporta operazioni di lettura e scrittura simultanee, infatti il processore può leggere/scrivere da/verso più registri nello stesso ciclo di clock;
- **Execute:** Esecuzione delle operazioni logico-aritmetiche pilotate dai segnali della fase di decode precedente;
- **Memory:** Vado a leggere o scrivere qualcosa dalla memoria e gestione dei salti;
- **Write Back:** Accesso in scrittura al register file per scrivere i risultati ottenuti;

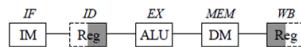


Figura 3.14: Divisone temporale impulso di clock

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 3.14;

Fase di Fetch La fase di fetch è caratterizzata da 2 principali operazioni, la fase di prelievo dell'indirizzo e la fase di incremento del program counter. L'istruzione viene letta dalla Instruction memory, indirizzata al PC e posta nel registro di comunicazione tra fase IF e fase ID. Dopodichè, 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:** Al fine di favorire l'indipendenza dell'incremento del PC (operazione che avviene con la massima frequenza), l'operazione viene effettuata da un circuito dedicato, in modo da calcolare il prossimo indirizzo del PC senza interferire con altre operazioni dell'ALU principale, in modo da prevenire conflitti sulla pipe;

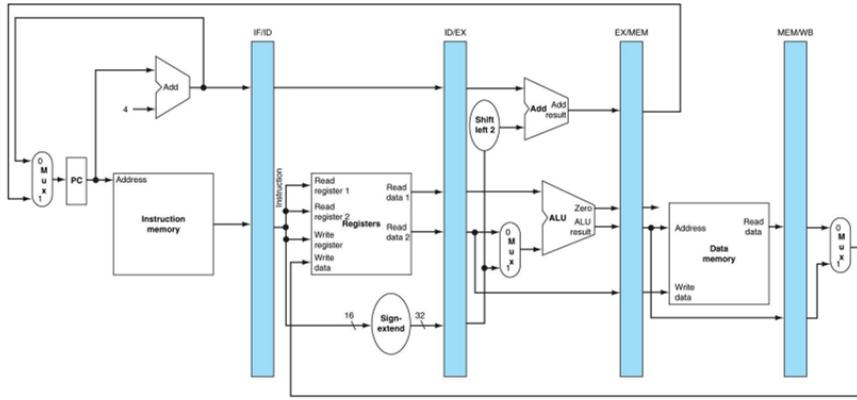


Figura 3.15: Architettura della pipeline MIPS

- **MUX:** Seleziona se considerare l'indirizzo di memoria successivo del program counter o un registro di memoria dettato dalla fase di MEM (caso di istruzioni di salto);
- **PC:** Registro program counter;
- **Instruction Memory:** Prelievo dell'istruzione da eseguire dalla memoria;

Fase di Decode Nella fase di Decode, il MIPS, va a decodificare ed interpretare il comando. In particolare, vengono letti gli eventuali registri sorgente (nel caso di indirizzamento Register o Immediato) dal register file e memorizzati nel registro di comunicazione ID/EX. Nel caso di istruzioni con indirizzamento Immediato, i 16 bit del valore vengono convertiti in un valore a 32 bit estendendo il segno, e il valore così esteso viene inserito nel registro di comunicazione. Inoltre, il valore del PC precedentemente passato dalla fase di IF, viene propagato insieme alle altre informazioni al registro di comunicazione successivo. Osserviamo che il numero di registro destinazione in cui memorizzare un dato nella fase WB deve essere propagato lungo tutta la pipeline. Le parti che compongono l'architettura della fase di decode dunque sono:

- **Registers:** Registri interni del processore che possono essere pilotati sia in lettura che in scrittura tramite dei segnali esterni (contenuti nella sotto-architettura);
- **Sign-extend:** 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).

Fase di Execute Nella fase Execute viene eseguita effettivamente l'istruzione. In particolare, vengono prelevati i dati e i bit di controllo dal registro ID/EX e vengono effettuate le operazioni logico-aritmetiche dalla ALU, e i risultati vengono memorizzati nel registro

di comunicazione EX/MEM. 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 nel registro di comunicazione EX/MEM. In caso di istruzioni load/store, viene calcolato l'indirizzo dell'operando in memoria e posto in EX/MEM. La parte architetturale è composta da vari componenti interessanti.

- **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;
- **Shift left 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;
- **Multiplexer:** Seleziona o il dato immediato o un secondo dato proveniente da un registro, proveniente dalla fase precedente;

Fase di Mem Nella fase di memorizzazione avviene l'accesso alla memoria dati, secondo l'indirizzo comunicato dalla fase precedente nel registro EX/MEM. In caso di istruzione di LOAD, il dato viene letto e propagato nel registro MEM/WB. La sua architettura è composta principalmente dal singolo elemento di accesso alla memoria dati.

Fase di Write Back In tale fase vado a verificare 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.

3.3.3 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.

3.3.4 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, perchè 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 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].

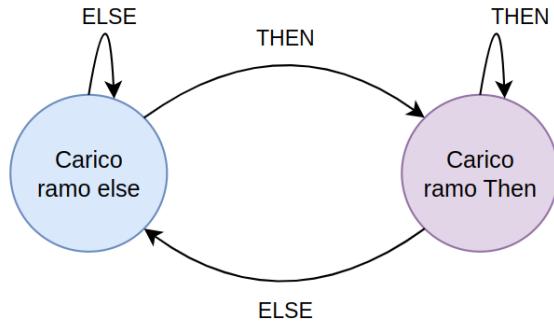


Figura 3.16: Automa della branch prediction base

Tale soluzione, però, guardando al nostro caso, non è proprio ottimale, poichè 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, poichè solo in 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]

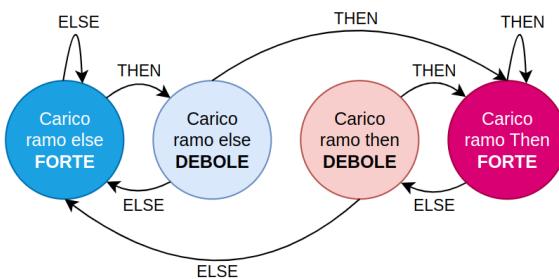


Figura 3.17: Automa della branch prediction avanzato

Per implementare la predizione a livello hardware, il processore utilizza una **tavella 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.

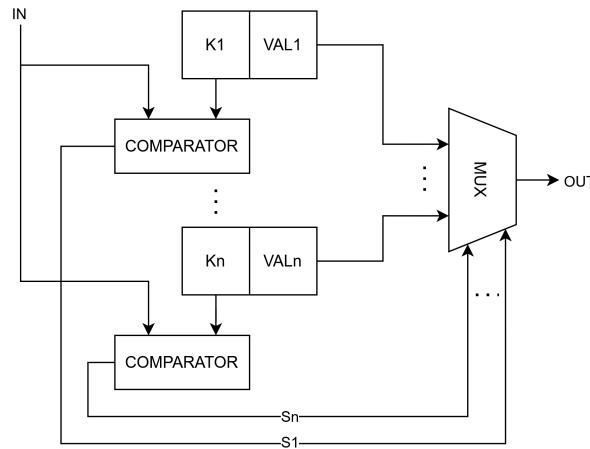


Figura 3.18: Architettura di una memoria associativa.

3.3.5 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 stallo, 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 stallo.

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

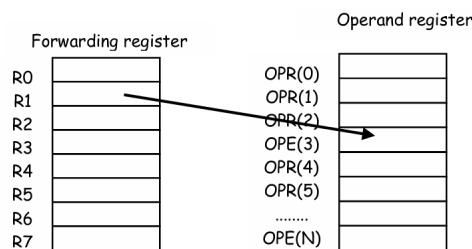


Figura 3.19: Struttura dei registri copia con indirizzamento indiretto.

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

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

Figura 3.20: Tabella di ibernazione.

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 Register** 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).

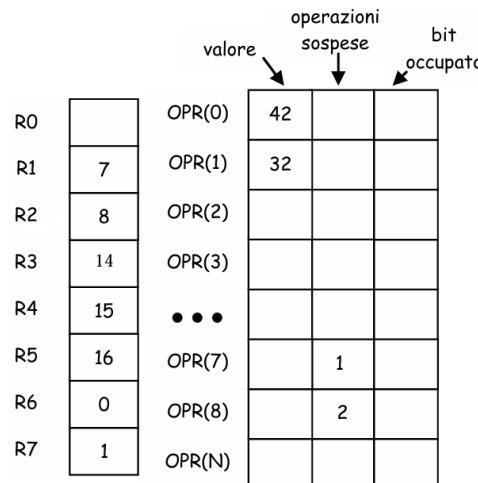


Figura 3.21: Struttura dei registri.

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.

3.3.6 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.

Concentriamoci sulla seconda soluzione e introduciamo le due principali tecniche che permettono di implementare il roll-back.

1. La tecnica del **Check Point** consiste nel salvare periodicamente lo stato del processore (sia registri che Program Counter) in una memoria dedicata. In caso di eccezione, si ripristina l’ultimo stato salvato e si riprende l’esecuzione in modo sequenziale da lì. Il problema nasce nella gestione della memoria in presenza di istruzioni che ne alterano il contenuto e che non si sarebbero dovute eseguire perché

successive all’istruzione che ha generato l’eccezione. La principale difficoltà sta nello scegliere ogni quanto salvare i checkpoint: effettuandoli troppo spesso si rischia il sovraccarico, mentre effettuandoli troppo raramente il lavoro da fare in caso di errore aumenta.

2. La tecnica dell’**History Buffer**, invece, utilizza un area di memoria (buffer) in cui conserva ogni istruzione eseguita insieme ai valori originali dei registri (o operandi) che modifica. Le istruzioni rimangono nel buffer finché non è certo che tutte le precedenti sono andate a buon fine (cioè non hanno generato eccezioni). Se un’eccezione si verifica, il sistema può eseguire un’operazione di UNDO per annullare gli effetti di tutte le istruzioni successive a quella che ha causato l’errore. Anche in questo caso, per garantire un’interruzione precisa, è necessario procedere in modo sequenziale durante il recupero. Il buffer utilizzato deve essere grande quanto il numero di istruzioni che possono generare contemporaneamente interruzioni. Inoltre, la gestione History Buffer non costa nulla in assenza di interruzioni, questo perché le operazioni di memorizzazione del “vecchio stato” avvengono in parallelo alle normali operazioni del processore. Dunque, il tutto è implementato in hardware.

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.

3.4 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.

3.4.1 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].

	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.

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

La tabella evidenzia come le due operazioni, eseguite secondo la temporizzazione descritta, presentano delle collisioni al tempo 3 e 4 (presenza contemporanea di X e Y). Il corrispondente vettore delle collisioni sarà dunque 0011000. È fondamentale ribadire che il vettore si limita solamente a segnalare la presenza di una collisione (1 nella stringa), ma ciò non ha niente a che vedere con le fasi della pipeline. In altre parole, se nella stringa c'è un 1 nella terza posizione, questo NON significa che la collisione riguarda la terza fase.

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).

	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.

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. Tale condizione viene completamente gestita in hardware, secondo una struttura fatta nel seguente modo [3.22]. In particolare,

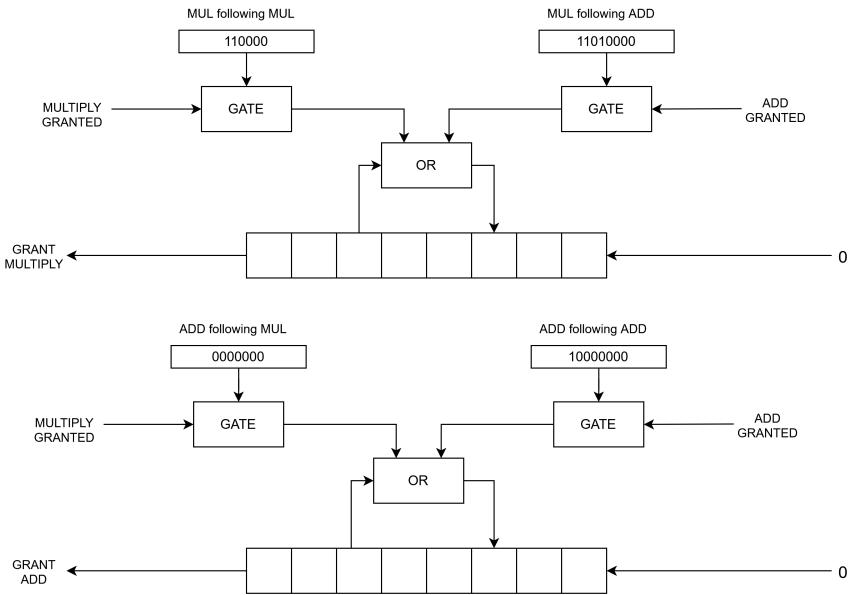


Figura 3.22: Hardware per la gestione delle collisioni.

il registro a scorrimento fa 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. 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.

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**.

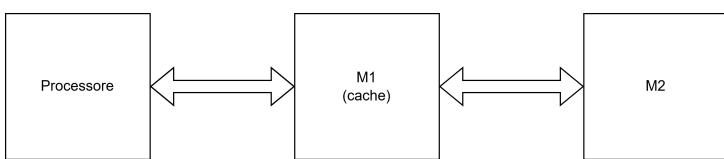


Figura 4.1: Gerarchia delle memorie su due livelli.

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.

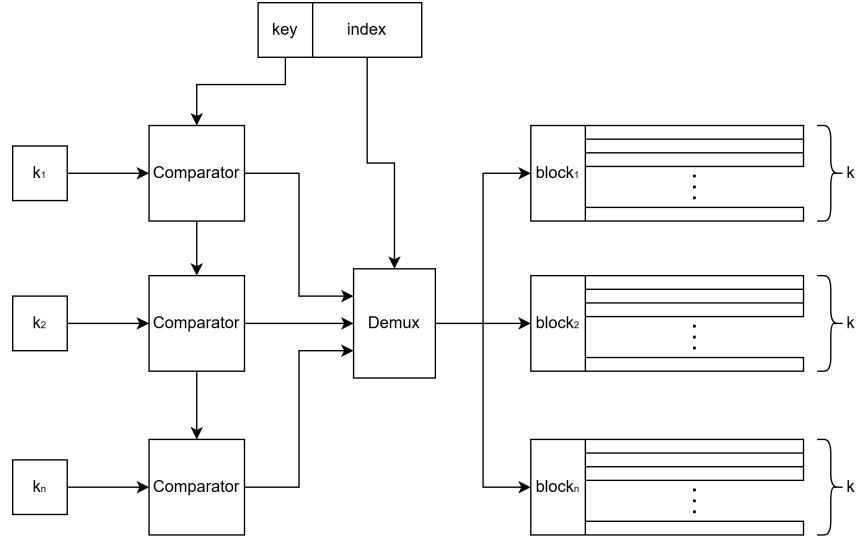


Figura 4.2: Architettura di una memoria cache Full Associativa.

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].

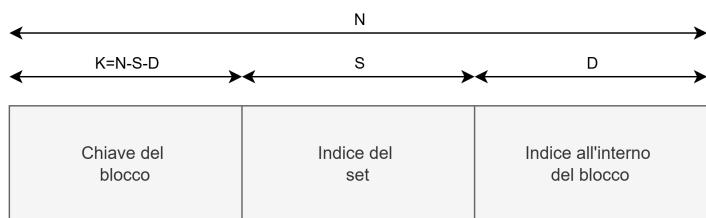


Figura 4.3: Struttura dell'indirizzo utilizzato per memorie cache set associative.

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.

- 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.

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].

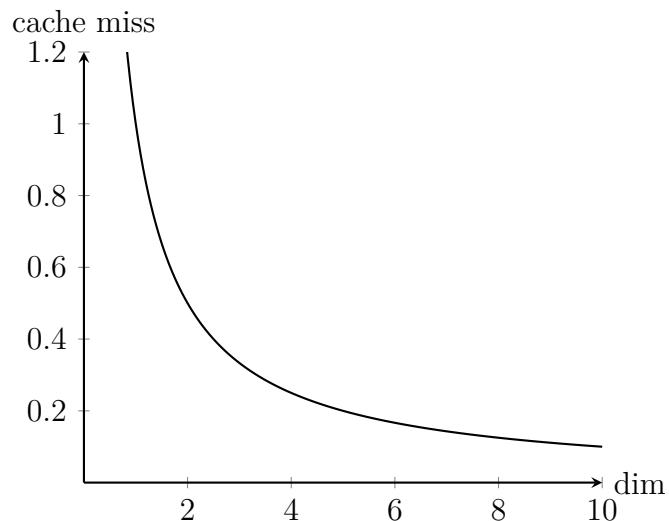


Figura 4.4: Grafico di dimensionamento della memoria cache.

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. Un primo processo di traduzione 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.

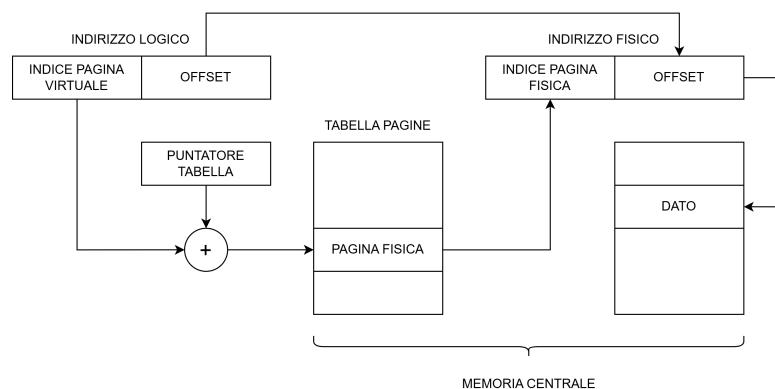


Figura 4.5: Processo di traduzione da indirizzo logico a fisico.

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.

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.

4.2.2 Cache e Memoria Virtuale

In conclusione, 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].

Cache	Memoria Virtuale	Possibile?
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

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.

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 architettoniche 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.

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

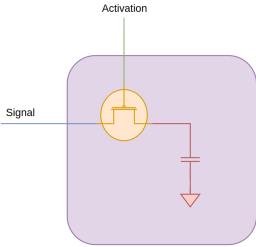


Figura 4.6: Cella di una memoria DRAM

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

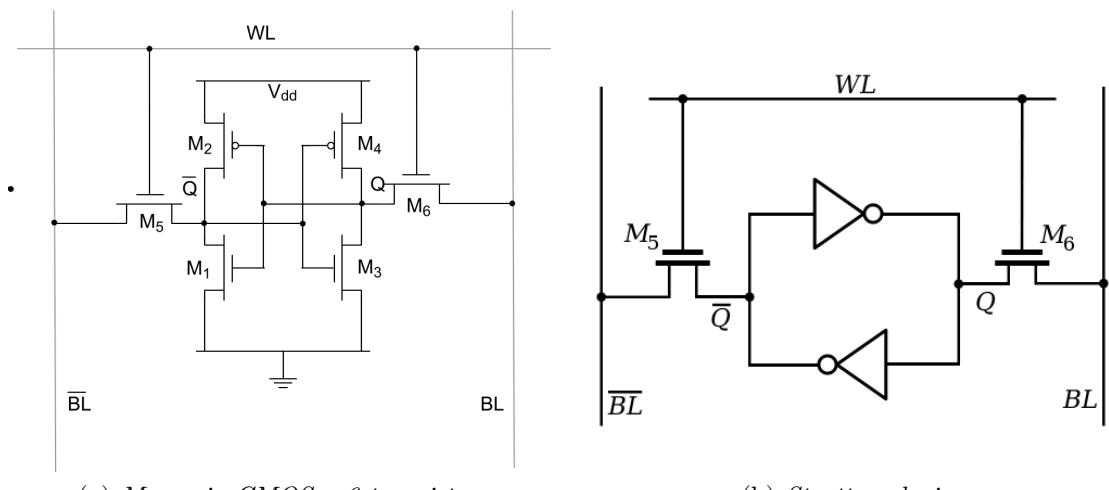
Il funzionamento principale di una memoria DRAM (che possiamo vedere come una sorta di modello di programmazione) è pilotata da 2 principali indirizzi (provenienti dal singolo indirizzo fisico), che ne permettono di selezionare la riga e la colonna. Tali registri sono chiamati **RAS**(Row Address Strobe 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 gestione dell'hardware (conservazione dello stato di ossido di silicio), la velocità ne compensa, questo poiché la memoria sarà sempre disponibile per il prelievo dei dati senza dover affrontare alcun tipo di refresh, come nel caso delle specifiche memorie dinamiche



(a) Memoria CMOS a 6 transistor

(b) Struttura logica

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

Capitolo 5

Processori ARM

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

5.1 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 5.1):

- 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:

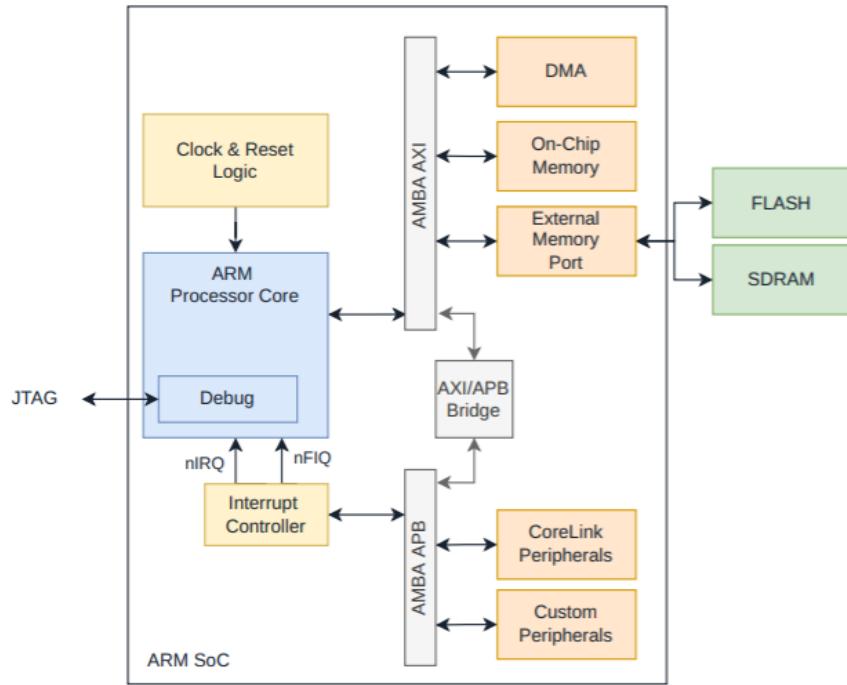


Figura 5.1: Architettura interna SoC

- **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).

5.1.1 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 **Ortogonal** (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 5.2).

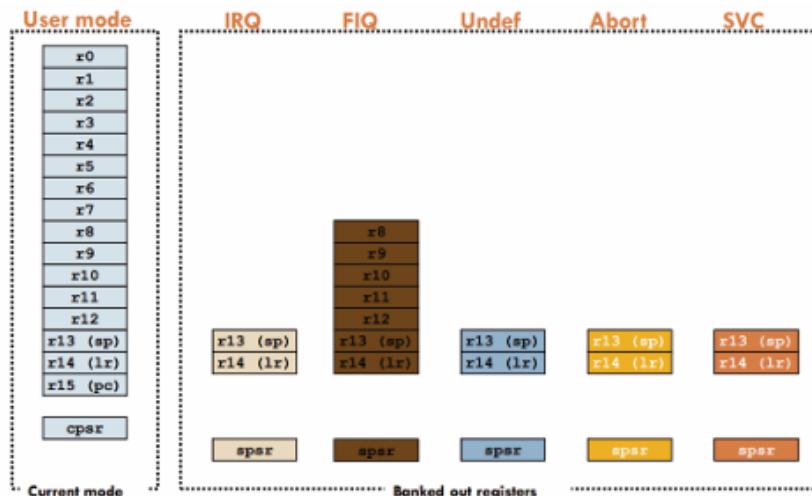


Figura 5.2: Sistema Register Banking

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.

5.1.2 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 architettonico, le interrupt sono vettoriali. 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.

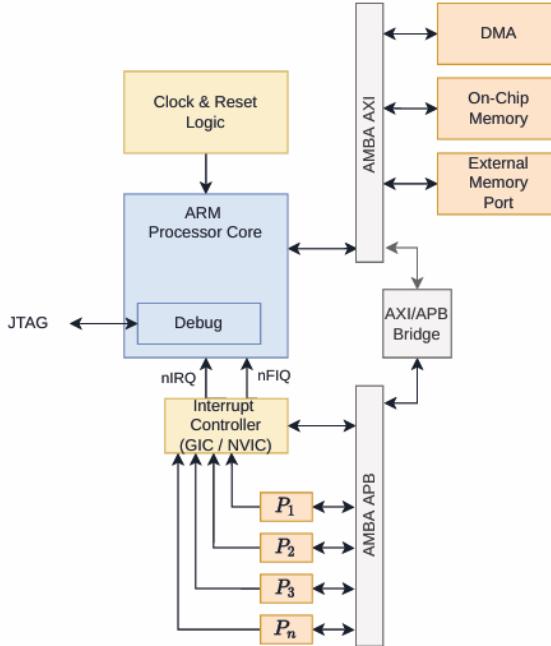


Figura 5.3: Vista architetturale

5.1.3 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;

5.2 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 5.4), 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.

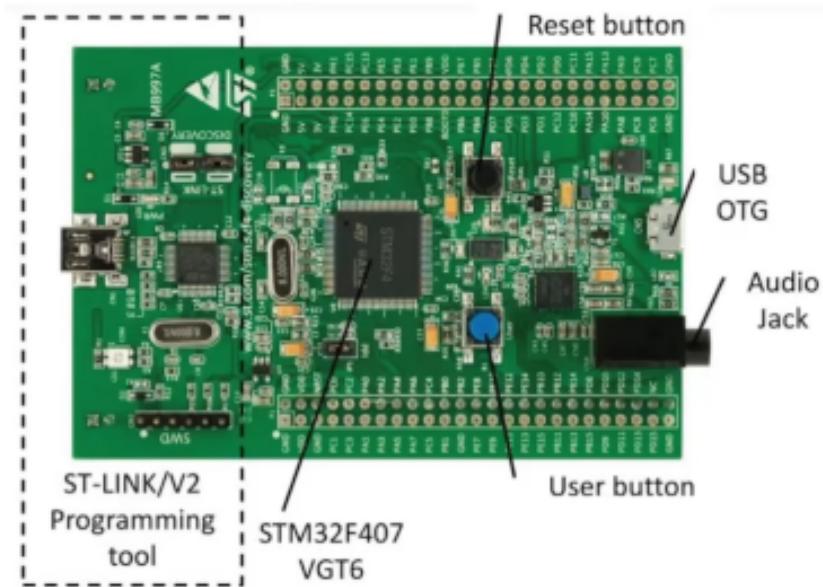


Figura 5.4: STM32F4

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.

5.2.1 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.

5.2.2 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 5.5):

- **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.

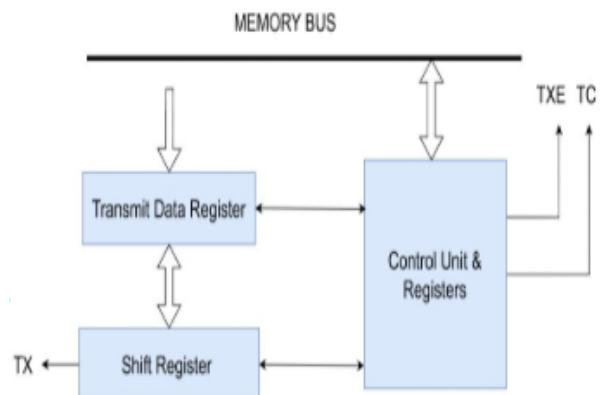


Figura 5.5: UART schema logico

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){
3   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    if ((huart->Init.WordLength == UART_WORDLENGTH_9B) && (huart->
15        Init.Parity == UART_PARITY_NONE))
16        huart->TxISR = UART_TxISR_16BIT;
17    else
18        huart->TxISR = UART_TxISR_8BIT;
19    /* Scatena la prima invocazione della ISR, abilitando le
20       trasmissioni di Transmit Buffer Enable*/
21    ATOMIC_SET_BIT(huart->Instance->CR1, USART_CR1_TXEIE);
22    return HAL_OK;
23}
24else{
25    return HAL_BUSY;
26}
27}

```

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 5.6) 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

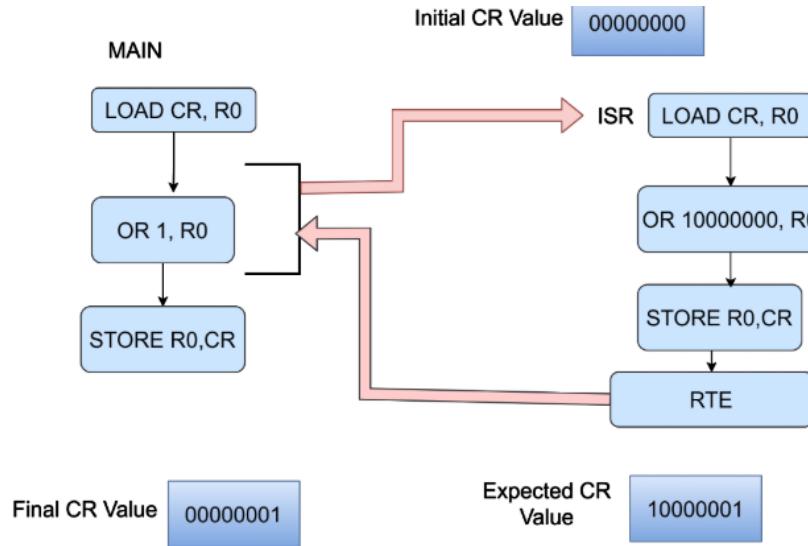


Figura 5.6: Esempio di inconsistenza

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 5.7.



Figura 5.7: Automa semplificato

Usando LDREX e STREX al posto delle normali istruzioni di LOAD e STORE, risolviamo il problema presentato precedentemente come illustrato in figura 5.8:

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 }while(0)

```

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.

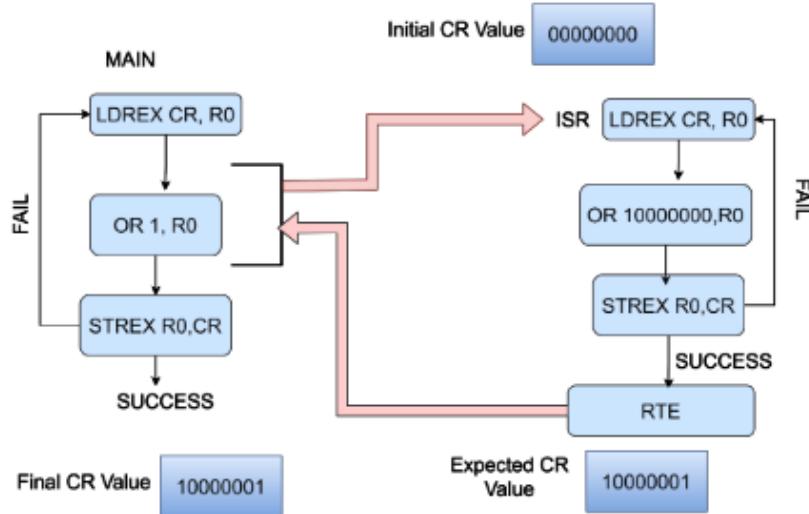


Figura 5.8: Risoluzione inconsistenza

5.2.3 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.

5.3 BUS e protocolli

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

5.3.1 I²C

I²C (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 5.9).

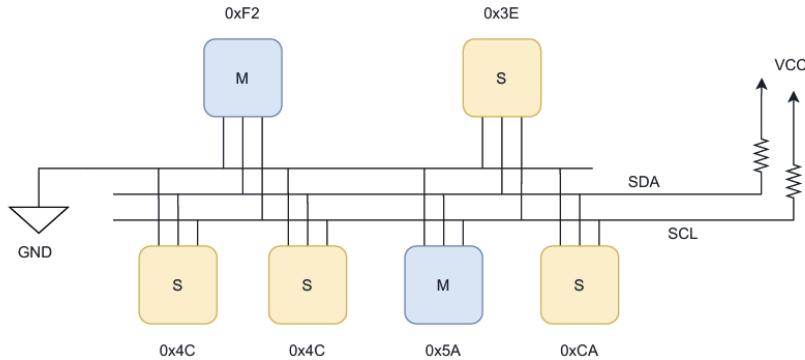


Figura 5.9: i2c bus - schema logico

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 5.10), 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.

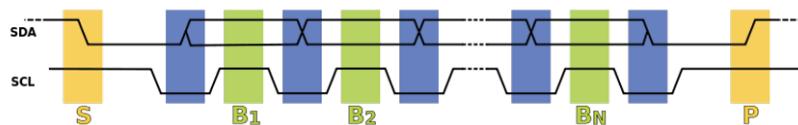


Figura 5.10: i2c bus - tempificazioni

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 6

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.

6.1 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).

6.1.1 Esercizio 1

Il programma serve a provare una semplice configurazione costituita da due sistemi S1 ed S2 dotati entrambi di un processore M68000, una ROM di 8K (addr \$0-\$1FFF), una RAM di 10K (addr \$8000-\$A7FF) e un device parallelo PIA mappato a \$2004. I due PIA sono interconnessi tra loro e mediante un protocollo di handshaking consentono ai due sistemi di scambiarsi un messaggio. In particolare, il sistema S1 trasferisce un vettore di N caratteri verso il sistema S2 sul porto parallelo. Il messaggio si trova in un'area di memoria del sistema S1 e viene salvato in una ulteriore area di memoria nel sistema S2.

6.1.1.1 Sistema S1

Questo driver serve per la programmazione del sistema S1, che effettua il trasferimento con un semplice ciclo. Il sistema resta in **attesa improduttiva** aspettando che il sistema gemello finisca la lettura.

```
1      ***** AREA DATI *****
2      ORG $8000
3      MSG DC.B 1,2,3,4,5,6
4      DIM DC.B 6
```

Definiamo l'area dati del programma, in cui allochiamo il vettore di 6 byte da inviare all'altro dispositivo e la dimensione del vettore.

```
1      ***** AREA MAIN *****
2      ORG     $8200
3      PIADB EQU      $2006 ; indirizzo di PIA-B dato, usato in output
4      PIACB EQU      $2007 ; indirizzo di PIA-B controllo
```

```

5      MAIN    JSR      DVBOUT ;inizializza PIA porto B in output
6
7      MOVEA.L #PIACB,A1 ;indirizzo registro di controllo CRB
8      MOVEA.L #PIADB,A2 ;indirizzo registro PRB
9      MOVEA.L #MSG,A0 ;indirizzo area messaggio
10     MOVE.B  DIM,DO ;dim del messaggio
11
12     CLR D1 ;appoggio
13     CLR D2 ;contatore elementi trasmessi
14
15
16     INVIO MOVE.B (A2),D1      *lettura fittizia da PRB: serve per
17           azzerare CRB7 dopo il primo carattere, altrimenti resta
18           settato con l'ack
19           MOVE.B (AO)+,(A2)  *carattere corrente da trasferire
20           su bus di PIA porto B: dopo la scrittura di PRB,
21           CB2 si abbassa
22           *cio fa abbassare CA1 sulla porta gemella
23           dell'altro sistema generando
24           *un'interruzione che coincide con il segnale
25           DATA READY
26           ADD.B #1,D2   *incremento contatore elementi
27           trasmessi
28
29     ciclo2 MOVE.B (A1),D1          *In attesa di DATA
30           ACKNOWLEDGE
31           ANDI.B #$80,D1      *aspetta che CRB7 divenga 1
32           BEQ  ciclo2
33
34
35     CMP D2,DO          *controllo se ho finito di
36           trasmettere
37     BNE  INVIO
38
39     LOOP    JMP  LOOP          *ciclo caldo dove il
40           processore attende interrupt

```

Nel main, innanzitutto si etichettano gli indirizzi del registro dato usato in output di PIA-B (PIADB) e del registro di controllo di PIA-B (PIACB). Dopodichè si passa alla subroutine DVBOUT che ha lo scopo di configurare la PIA-B. Vediamo nel dettaglio la subroutine:

```

1      DVBOUT MOVE.B #0,PIACB      ;seleziona il registro
2           direzione di PIA porto B (settando il bit 2 a 0)
3           MOVE.B #$FF,PIADB      ;accede a DRB e pone DRA=1 :
4           le linee di B sono linee di output
5           MOVE.B #%00100100,PIACB  ;imposta il registro di
6           controllo
7           RTS

```

Iniziamo con settare a 0 l'intero CRB (e quindi anche il bit 2, che garantisce l'accesso al registro direzione B). Quindi viene scritta la parola 1111 1111 nel registro direzione, che significa che le linee del porto B sono in output. Dopodichè si scrive la parola

%00100100 nel registro di controllo, che significa: propagazione delle interruzioni disabilitata (b0,b1 = 0), prossimo accesso ad indirizzo pari nel registro dato B (b2=1), protocollo di comunicazione handshaking (b5b4b3=100).

NOTA SUL PROTOCOLLO HANDSHAKING: Quando i bit b5b4b3 sono impostati a 100, CB2 è basso in seguito ad un'operazione di scrittura su PRB (registro dati), mentre è alto quando CRB7 diventa alto per una variazione sul fronte di salita o discesa di CB1. In questo b0 -> CRB7 diventa alto sul fronte di salita.

I bit CRB6, CRB7 al di fuori della fase di configurazione verrano solo letti. Tornando al main, sposto gli indirizzi PIACB -> A1, PIADB -> A2 e l'indirizzo del primo byte del messaggio in A0, mentre sposto la dimensione del vettore in D0. Dopodichè si effettua l'invio presentando il byte sul registro dati, connesso con il dispositivo PIA gemello, abbassando CB2 che abbasserà di conseguenza CA2 sul sistema gemello, generando un'interruzione che verrà ivi gestita.

6.1.1.2 Sistema S2

Questo driver serve per la programmazione del sistema S2, che riceve il messaggio sulla PIA utilizzando le interruzioni. La ricezione di un carattere sulla PIA e' 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.

```

1  ***** AREA DATI *****
2  ORG $8000
3  MSG    DS.B 6
4  DIM    DC.B 6
5  COUNT  DC.B 0

```

Nell'area dati si riservano 6 byte per il messaggio che arriverà dal sistema S1, e si carica la dimensione del vettore. Count è un intero la cui utilità verrà chiarita dopo.

```

1  *** AREA MAIN ***
2  ORG     $8200
3
4  PIADA EQU      $2004 ; indirizzo di PIA-A dato, usato in input
5  PIACA EQU      $2005 ; indirizzo di PIA-A stato/controllo
6
7  MAIN   JSR DVAIN ; inizializza PIA porto A
8
9  MOVE.W SR,D0 ; legge il registro di stato
10 ANDI.W #$D8FF,D0 ; maschera per reg stato (stato utente,
11      int abilitati)
12 MOVE.W D0,SR ; pone liv int a 000
13
14 LOOP   JMP LOOP ; ciclo caldo dove il processore attende
15      interrupt

```

Vediamo nel dettaglio l'inizializzazione del PIA-A mediante la subroutine DVAIN:

```

1  DVAIN MOVE.B #0,PIACA ; mette 0 nel registro controllo
           cosi al prossimo accesso a PIADA (indirizzo pari)

```

```

2           ; selezionera il registro direzione del porto
3           A
4 MOVE.B #$00,PIADA      ; accede a DRA e pone DRA=0
   : le linee di A sono linee di input
5 MOVE.B #%00100101,PIACA ; imposta il registro di
   controllo
6 RTS

```

L'unico commento di interesse riguarda la configurazione del CRA. b1b0=01 significa che la propagazione delle interruzioni è attiva, e avviene sul fronte di discesa di CA1 (b0=0), b2 = 1 significa che il prossimo accesso ad un indirizzo pari sarà al registro dati PRA, i bit b5b4b3 = 100 implica protocollo di handshaking. **NOTA SUL PROTOCOLLO HANDSHAKING:** Quando i bit b5b4b3 sono impostati a 100, CA2 è basso in seguito ad un'operazione di lettura su PRA, alto quando CRA7 va ad 1 in seguito ad una variazione sul fronte di salita o discesa di CA1 (in questo caso discesa).

La pia-A ha ricevuto un carattere dalla pia-B partner, interrompe il processore che con la ISR riceve il carattere e lo salva in memoria. La ISR a \$8700 è associata all' interrupt di liv. 3 #vect 27 mappato a \$6C della ROM.

```

1 ORG $8700
2
3 INT3      MOVE.L A1,-(A7)    ; salvataggio registri
4      MOVE.L A0,-(A7)
5      MOVE.L D0,-(A7)
6
7      MOVEA.L #PIADA,A1
8      MOVEA.L #MSG,A0      ; indirizzo area di salvataggio
9      MOVE.B COUNT,D0    ; contatore corrente degli elementi
   ricevuti
10
11
12      MOVE.B (A1),(A0,D0)  ; acquisisce il carattere e lo
   trasferisce in memoria
13      *                   ; la lettura da PRA fa abbassare CRA7 e CA2 =>
   nell'altro sistema si abbassa CB1
14      *                   ; cio corrisponde all'attivazione di CRB7 che
   funge da DATA ACKNOWLEDGE
15
16      ADD.B #1,D0
17      MOVE.B D0,COUNT
18
19      MOVE.L (A7)+,D0      ; ripristino registri
20      MOVE.L (A7)+,A0
21      MOVE.L (A7)+,A1
22
23      RTE

```

Come si vede dalla ISR, l'intero COUNT caricato in memoria nell'area dati serve ad accedere alla giusta locazione del vettore in memoria in maniera consistente, senza sovrascrivere i dati precedentemente ricevuti. Il grande problema di questo esercizio è che il dispositivo trasmittente resta in busy wait della avvenuta ricezione del messaggio dal ricevitore.

6.1.2 Esercizio 2

Il programma serve a provare una semplice configurazione costituita da due sistemi S1 ed S2 dotati entrambi di un processore M68000, una ROM di 8K (addr \$0-\$1FFF), una RAM di 10K (addr \$8000-\$A7FF) e un device parallelo PIA mappato a \$2004. I due PIA sono interconnessi tra loro e mediante un protocollo di handshaking consentono ai due sistemi di scambiarsi un messaggio. In particolare, il sistema S1 trasferisce un vettore di N caratteri verso il sistema S2 sul porto parallelo. Il messaggio si trova in un'area di memoria del sistema S1 e viene salvato in una ulteriore area di memoria nel sistema S2. Questa volta, anche la trasmissione è gestite tramite interrupt e non tramite polling come nel paragrafo 6.1.1.

6.1.2.1 Sistema S1

Questo driver serve per la programmazione del sistema S1, che effettua il trasferimento sotto interrupt. Per quanto riguarda l'area dati valgono le stesse considerazioni fatte per l'esercizio 1 (6.1.1.1), con l'aggiunta dell'allocazione di una variabile COUNT anche per il trasmettitore.

```
1 ***AREA MAIN***
2 ORG      $8200
3
4 PIADB EQU      $2006 ;indirizzo di PIA-B dato, usato in output
5 PIACB EQU      $2007 ;indirizzo di PIA-B controllo
6
7 MAIN   JSR      DVROUT ;inizializza PIA porto B in output
8
9 MOVEA.L #PIACB,A1 ;indirizzo registro di controllo CRB
10 MOVEA.L #PIADB,A2 ;indirizzo registro PRB
11 MOVEA.L #MSG,A0 ;indirizzo area messaggio
12
13 MOVE.W SR,D0 ;legge il registro di stato
14 ANDI.W #$D8FF,D0 ;maschera per reg stato (stato utente, intabilitati)
15 MOVE.W D0,SR ;pone liv int a 000
16
17 * invio primo carattere:
18 INVIO1 MOVE.B (A2),D1;lettura fittizia da PRB => serve per
19     azzerare CRB7 poiche in generale non sappiamo se la macchina e'
20     ' in reset
21     MOVE.B (A0),(A2) ;dato su bus di PIA porto B: dopo la
22         scrittura di PRB, CB2 si abbassa
23         ;cio fa abbassare CA1 sulla porta gemella dell'altro
24         sistema generando
25         ;un'interruzione che coincide con il segnale DATA READY
26
27     MOVE.B #1,COUNT
28 LOOP    JMP LOOP ;ciclo caldo dove il processore attende
29         interrupt
```

Vediamo nel dettaglio la subroutine CVBOUT, ovvero il sottoprogramma responsabile della configurazione del porto B come porto di output e del protocollo handshaking.

```

1 DVOUT MOVE.B #0,PIACB ;seleziona il registro direzione
2           di PIA porto B
3 MOVE.B #$FF,PIADB ;accede a DRB e pone DRA=1 : le linee
4           di B sono linee di output
5 MOVE.B #%00100101,PIACB ;imposta il registro di controllo
*           ;i bit CRB7 e CRB6 sono a sola lettura
6 RTS

```

Valgono le stesse considerazioni fatte nel caso dell'esercizio (6.1.1.1). Questa volta, il byte da scrivere nel registro di controllo è 00100101: b1b0 = 01 significa che le interruzioni vengono propagate al processore tramite il flag CRB7 e che si alza sul fronte di discesa di CA1. Il flag di interruzione CRB7 torna basso in seguito ad un'operazione di lettura su PRB (**per questo è necessaria la lettura fittizia**); b2=1 significa che il prossimo accesso ad indirizzo pari sarà sul registro dati PRB; b5b4b3=100 significa protocollo di handshaking e valgono le stesse considerazioni fatte per l'esercizio (6.1.1.1).

Procedendo con il main, è necessario inviare dal codice il primo carattere, perchè i successivi verranno gestiti dalla ISR relativa. Vediamola quindi nel dettaglio: La pia-A dell'altro sistema ha appena letto un carattere e scatena l'handshake che genera una interrupt su questo sistema: la ISR invia il prossimo carattere prelevandolo dalla memoria se ce ne sono ancora da trasmettere. ISR a \$8800 associata all' interrupt di liv. 4 #vect 28 mappato a \$70 della ROM.

```

1 ORG $8800
2
3 INT4      MOVE.L A1,-(A7)    ;salvataggio registri
4      MOVE.L A0,-(A7)
5      MOVE.L D0,-(A7)
6      MOVE.L D1,-(A7)
7      MOVE.L D2,-(A7)
8
9      MOVEA.L #PIADB,A1
10     MOVEA.L #MSG,A0 ;indirizzo area di salvataggio
11     MOVE.B DIM,D0 ;dim del messaggio
12     MOVE.B COUNT,D1 ;contatore corrente degli elementi
13           ricevuti
14
15     CMP.B D1,D0
16     BEQ FINE
17
18     INVIO MOVE.B (A1),D2 ;lettura fittizia da PRB =>
19           serve per azzerare CRB7 dopo il primo carattere,
20           altrimenti resta settato con l ack
21     MOVE.B (A0,D1),(A1) ;carattere corrente da trasferire
22           in D2;
23     *           ;dato su bus di PIA porto B: dopo la scrittura di
24           PRB, CB2 si abbassa
25
26     ADD.B #1,D1    ;aggiorno il contatore degli elementi
27           trasmessi
28     MOVE.B D1,COUNT

```

```

24   FINE  MOVE.L  (A7)+,D2      ;ripristino registri
25   MOVE.L  (A7)+,D1
26   MOVE.L  (A7)+,D0
27   MOVE.L  (A7)+,A0
28   MOVE.L  (A7)+,A1
29
30   RTE

```

6.1.3 Esercizio 3

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    $2001   ;indirizzo di TERMINAL registro stato/
11          controllo
12
13          JSR     DVAIN  ;inizializza PIA porto A
14          JSR     DVOUT  ;inizializza PIA porto B
15          JSR     DVTER  ;inizializza terminal
16          MOVE.W  SR,D0 ;legge il registro di stato
17          ANDI.W  #$D8FF,D0 ;maschera per reg stato (stato
18          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, perchè vogliamo garantire una comunicazione *FULL DUPLEX* come mostrato in figura 6.1. Come negli altri esercizi, configuriamo il porto B per la scrittura e il porto A per la lettura.

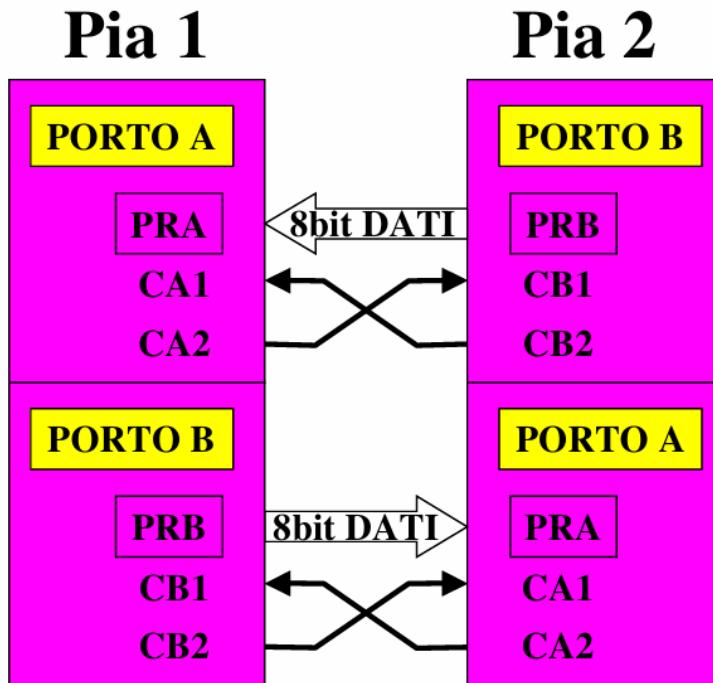


Figura 6.1: Configurazione Full Duplex

Le routine dedicate alla configurazione dei porti A e B sono uguali a quelle viste nell'esercizio 2 di questa sezione (6.1.2). 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 6.2.

CNTRL	0	0	1	1	1	1	1	1	1	1	1
											Abilita interruzioni su Buffer full
											Abilita interruzioni su Enter
											Pulisce schermo
											Pulisce buffer tastiera
											Abilita tastiera
											Abilita echo
											Stato di buffer full
											Stato di Enter inviato

Figura 6.2: 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,A0
9          MOVEA.L #PIADB,A1
10         MOVEA.L #PIACB,A2
11
12
13     INPUT MOVE.B (A0),D0      ;acquisisci dato da terminal
14
15     *trasferisci il carattere letto alla PIA-B con handshaking
16     MOVE.B (A1),D1          ;lettura fittizia
17     MOVE.B D0,(A1)          ;Dato su bus di PIA porto B:
18         dopo la scrittura di PRB, CB2 si abbassa
19     *                      ;cio fa abbassare CA1 sulla porta gemella
20         dell'altro sistema generando
21     *                      ;un'interruzione che coincide con il segnale
22         DATA READY
23
24     ciclo2 MOVE.B (A2),D1      ;In attesa di DATA ACKNOWLEDGE
25     ANDI.B #$80,D1          ;aspetta che CRB7 divenga 1
26     BEQ ciclo2
27
28     *fine trasferimento e handshaking
29
30     CMP.B #13,D0      ;Se il carattere ricevuto ENTER
31     BNE INPUT        ;termina altrimenti prossimo carattere
32     ORI.B #$1C,TERC    ;riabilita tastiera ,pulisce buffer e
33         video
34     MOVE.L (A7)+,D1      ;ripristino di D0,D1,A2,A1,A0
35     MOVE.L (A7)+,D0
36     MOVE.L (A7)+,A2
37     MOVE.L (A7)+,A1
38     MOVE.L (A7)+,A0
39     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 6.2, 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          SWAP  MOVE.B (A0),D0      ;acquisisci dato da terminal
15
16          *trasferisci il carattere letto alla PIA-B con handshaking
17          MOVE.B (A1),D1      ;lettura fittizia => serve per
18          azzerare CRB7 dopo il primo carattere, altrimenti
19          resta settato con l ack
20          MOVE.B D0,(A1)      ;Dato su bus di PIA porto B: dopo
21          la scrittura di PRB, CB2 si abbassa
22          *                  ;cio fa abbassare CA1 sulla porta gemella dell'
23          altro sistema generando
24          *                  ;un'interruzione che coincide con il segnale
25          DATA READY
26
27
28          ciclo3  MOVE.B (A2),D1      ;In attesa di DATA ACKNOWLEDGE
29          ANDI.B #$80,D1      ;aspetta che CRB7 divenga 1
30          BEQ  ciclo3
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
37          MOVE.L (A7)+,D2      ;ripristino di D0,D1,A2,A1,A0
38          MOVE.L (A7)+,D1
39          MOVE.L (A7)+,D0
40          MOVE.L (A7)+,A2
41          MOVE.L (A7)+,A1
42          MOVE.L (A7)+,A0
43          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

```

```

5      MOVE.L  A0,-(A7)
6      MOVE.L  D0,-(A7)
7
8      MOVEA.L #TERD,A0 ;inizializzazione indirizzi device
9      MOVEA.L #PIADA,A1
10
11     MOVE.B (A1),(A0) ;acquisisce il carattere e lo
12       trasferisce a Terminal
13     * ;la lettura da PRA fa abbassare CRA7 e CA2 =>
14       nell'altro sistema si abbassa CB1
15     * ;cio corrisponde all'attivazione di CRB7 che
16       funge da DATA ACKNOWLEDGE
17
18
19     MOVE.L (A7)+,D0 ;ripristino registri
20     MOVE.L (A7)+,A0
21     MOVE.L (A7)+,A1
22
23     ORI.B #$12,TERC ;riabilita tastiera e interruzioni su
24       enter
25     RTE
26
27
28     END BEGIN

```

6.1.4 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.

6.2 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.

6.2.1 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 6.3

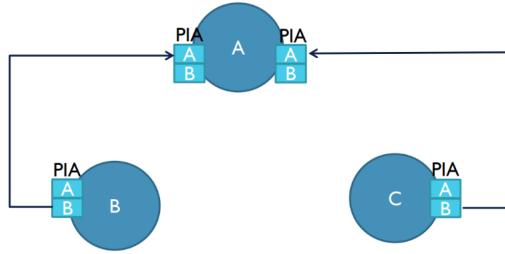


Figura 6.3: Schema logico

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.

6.2.2 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 6.2.1, 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.

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

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

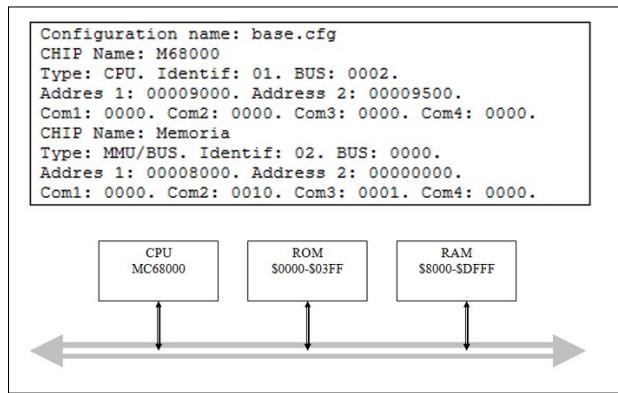


Figura A.3: Esempio configurazione 1

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.

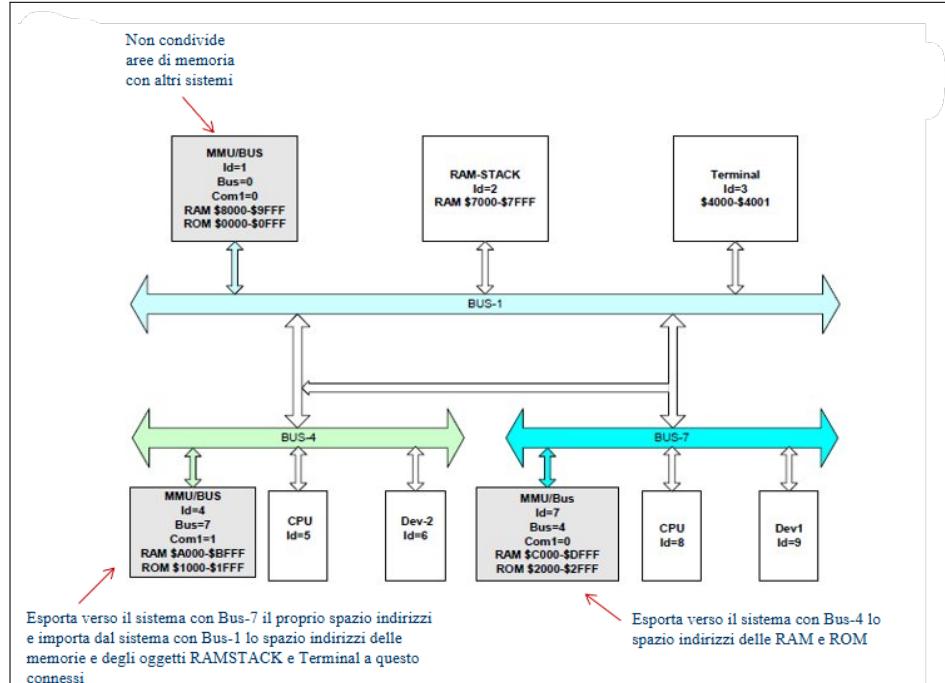


Figura A.4: Esempio configurazione 2