Appunti di

Architettura e progettazione dei calcolatori

Corso di laurea magistrale in

Ingegneria Informatica

Indice

Introduzione

FNS L'inter vincerà lo scudetto Ciccogamer Vive
FOrza Napolo Sempre

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 del 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 (Istruction 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 esecusione (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 i 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(Istruction 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-Istruction-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-Istruction-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 effetuate 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'esecuzioe (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 se non ha senso

La caratterizzazione del comando `MOVE` è la seguente:

```
    *Indirizzamento diretto D1 = D0 o D1<-D0
    MOVE D0,D1

    *Indirizzamento indiretto (sorgente), diretto (destinazione)
    *D0 = (A0), D0 = contenuto del registro in posizione A0
    MOVE.W (A0),D0

    *Indirizzamento Indiretto completo A1 = A0
    MOVE.L (A0),(A1) (istruzione non valida)
    *o
    MOVEA.L (A0),A1 (istruzione valida)
    *Indirizzamento immediato + indirizzamento Diretto
    MOVE.L #14,D0

    * Indirizzamento con spiazzamento su registro di indirizzo
    MOVE.W 4(A0), D0     * D0 = valore all'indirizzo A0 + 4

    * Indirizzamento con spiazzamento e registro indice
    MOVE.L 8(A0, D1.L), D2    * D2 = valore all'indirizzo A0 + 8 +
        D1

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

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

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

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

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

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

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

    * Ritorno da subroutine (equivalente a POP del PC)
    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 *sotterazione 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,
         D0 = D0 / 3 (quoziente in D0, resto in D1)
2    DIVU.W #3,D0      * Divisione senza segno con specifica
       grandezza valore, D0 = D0 / 3
3    DIVU.W D0,D1      * Divisione senza segno con indirizzamento
       diretto, D1 = D1 / D0
4
5    DIVS.W #3,D0      * Divisione con segno immediato + diretto,
       D0 = D0 / 3
6    DIVS.W D0,D1      * Divisione con segno tra registri, D1 = D1
       / D0
```

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 Statur Register (SR)
- **Salti a subrutine**: 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, poichè 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
        nello stack)
2    BSR label
3
4    *Salta a una subroutine con indirizzo specifico o label
5    JSR address/label
6
7    * Ritorna dalla subroutine (estrae l'indirizzo di ritorno
        dallo stack, fa il pop dallo stack)
8    RTS
```

1.2.5 Codici Logici

I codici logici sono operazioni che possono essere effettuate su degli operandi 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, D0 = D0
         & 3
2    AND.W D0, D1      * AND tra registri, D1 = D1 & D0
3    AND.L (A0), D0    * AND tra valore in memoria puntato da A0 e
        D0
4
5    OR #3, D0         * OR bit a bit con valore immediato, D0 = D0
        | 3
6    OR.W D0, D1       * OR tra registri, D1 = D1 | D0
7    OR.L (A0), D0     * OR tra valore in memoria puntato da A0 e D0
8
9    EOR #3, D0        * XOR bit a bit con valore immediato, D0 = D0
         XOR 3
10   EOR.W D0, D1      * XOR tra registri, D1 = D1 XOR D0
11   EOR.L (A0), D0    * XOR tra valore in memoria puntato da A0 e
        D0
12
13   NOT D0            * Complemento bit a bit (negazione), D0 = ~D0
```

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
```

```
    BTST #3, D0      * Testa il bit 3 di D0 (imposta Z se il bit e' 0)
    BTST #5, (A0)    * Testa il bit 5 della memoria puntata da A0

    BSET #3, D0      * Imposta il bit 3 di D0 a 1
    BSET #5, (A0)    * Imposta il bit 5 della memoria puntata da A0

    BCLR #3, D0      * Azzera il bit 3 di D0
    BCLR #5, (A0)    * Azzera il bit 5 della memoria puntata da A0

    BCHG #3, D0      * Inverte il bit 3 di D0 (0 -> 1, 1 -> 0)
    BCHG #5, (A0)    * Inverte il bit 5 della memoria puntata da A0

    TAS D0           * Testa e imposta il bit piu' alto (7) di D0
    TAS (A0)         * Testa e imposta il bit 7 del valore in memoria puntato da A0
```

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 struttura 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:

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

END_IF:         * Codice successivo...
```

Esempio di un If-Then-else

```
                    MOVE.L   D0, D1          * Carica valori nei registri
                        (supponiamo che D0 e D1 abbiano gia' valori)
                    CMP.L    D1, D0          * Confronta D0 con D1 (D0 -
                       D1)
                    BGT      THEN_BLOCK      * Se D0 > D1, salta al
                       blocco THEN

                    * ELSE block
                    MOVE.L   #0, D2          * D2 = 0
                    BRA      END_IF          * Salta oltre il blocco THEN
                        per evitare di eseguirlo

THEN_BLOCK:         MOVE.L   #1, D2          * D2 = 1

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:

```
                    MOVE.L   #0, D0          * Inizializza il contatore
                        D0 = 0

FOR_LOOP:           CMP.L    #10, D0         * Confronta D0 con 10
                    BGE      FOR_END         * Se D0 >= 10, esce dal
                       ciclo

                    * Corpo del ciclo
                    NOP                      * Istruzione di esempio (
                       da sostituire con il codice reale)

                    ADDQ.L   #1, D0          * Incrementa D0 di 1
                    BRA      FOR_LOOP        * Ripete il ciclo

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:

```
WHILE_LOOP:         CMP.L    #0, D1          * Confronta D1 con 0
                    BLE      WHILE_END       * Se D1 <= 0, esce dal
                       ciclo

                    * Corpo del ciclo
                    NOP                      * Istruzione di esempio
```

```
6
7                   SUBQ.L  #1, D1          * Decrementa D1 di 1
8                   BRA     WHILE_LOOP      * Ripete il ciclo
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
                    D0
2               MOVE.L  #10, D1         * Carica il secondo operando in
                     D1
3
4               MOVE.L  D0, -(A7)       * Push del primo operando nello
                     stack
5               MOVE.L  D1, -(A7)       * Push del secondo operando
                    nello stack
6
7               JSR     SUM_SUB         * Chiamata alla subroutine
8
9               MOVE.L  (A7)+, D2       * Il chiamante preleva il
                    risultato dallo stack
10
11              ADDQ.L  #8, A7          * Pulizia dello stack (2 valori
                     da 4 byte)
12
13              * D2 ora contiene il risultato della somma
14
15              * Codice successivo...
16
17  SUM_SUB:    MOVE.L  (A7)+, D0       * Pop del primo operando dallo
                stack
18              MOVE.L  (A7)+, D1       * Pop del secondo operando
                    dallo stack
19
20              ADD.L   D1, D0          * Somma D0 + D1, risultato in
                    D0
21
```

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

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

```
1               MOVE.L  #5, D0          * Carica il primo operando in
                    D0
2               MOVE.L  #10, D1         * Carica il secondo operando in
                     D1
3
4               MOVE.L  D0, -(A7)       * Push del primo operando nello
                     stack
5               MOVE.L  D1, -(A7)       * Push del secondo operando
                    nello stack
6
7               JSR     SUM_SUB         * Chiamata alla subroutine
8
9               MOVE.L  (A7)+, D2       * Il chiamante preleva il
                    risultato dallo stack
10
11              ADDQ.L  #8, A7          * Pulizia dello stack (2 valori
                     da 4 byte)
12
13              * D2 ora contiene il risultato della somma
14
15              * Codice successivo...
16
17  SUM_SUB:    MOVE.L  (A7)+, D0       * Pop del primo operando dallo
                stack
18              MOVE.L  (A7)+, D1       * Pop del secondo operando
                    dallo stack
19
20              ADD.L   D1, D0          * Somma D0 + D1, risultato in
                    D0
21
22              MOVE.L  D0, -(A7)       * Push del risultato nello
                    stack
23
24              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). Veodiamo degli esempi per capire meglio di cosa si sta parlando:

Istruzione	PI	PO
`MOVE.L D0,D1`	1	0
`MOVE.W D0,D1`	1	0
`MOVE.L #7,D1`	3	0
`MOVE.W (A0),(A1)`	1	2
`MOVE.W (A0),VAR`	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 pre 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 dospositivi 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 interfacciarci 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'architattura 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 difatti 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 sui 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'handshackin è una modalità di sincronizzazione 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 emanato dalla periferica che fa comprendere che il dato è pronto o che è stata presa in carico l'operazione

- **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 ha specifici comandi 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 classici comandi di spostamento già utilizzati

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 interscambiati 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
- **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. Oltretutto 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 sistema è altamente inefficiente, poichè mentre la CPU aspetta la riposta della periferica passano dei periodi di clock dove la CPU rimane ferma. 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,D0
8  *Devo prelevare N dati quindi ciclo N volte
9  FOR    CMP.W     #N,D0
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
       dello stato
16 L1     MOVE.B   S,D1
17        AND.B    #$80,D1   *Se il bit si e' alzato ho finito.
             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
             segnale di stato
24
25 FUORI  ADD.B   #1,D0              *Incremento il conteggio
26        BRA   FOR                  *Ripeti
27
28        ORG       $8100
29 D      DS.B      1    *Registro dato
30 S      DS.B      1    *Registro Stato
31 C      DS.B      1    *Registro Controllo
32
33 N      EQU       5    *Quantita' di valori da considerare
34 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
- **Polling**: L'attesa che viene svolta all'interno di tale codice non permette al processore di eseguire altri passi 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 inidirizzi 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.

2.1.4.1 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 [2.1]. In generale non è corretto parlare solo di interruzioni, poichè 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
- **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'obbiettivo 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, poichè se quando torno dalla ISR, ho cambiato qualche registro significativo si potrebbe compromettere il normale funzionamento del programma. In generale i due registri che richiedono l'obbligo di essere 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.

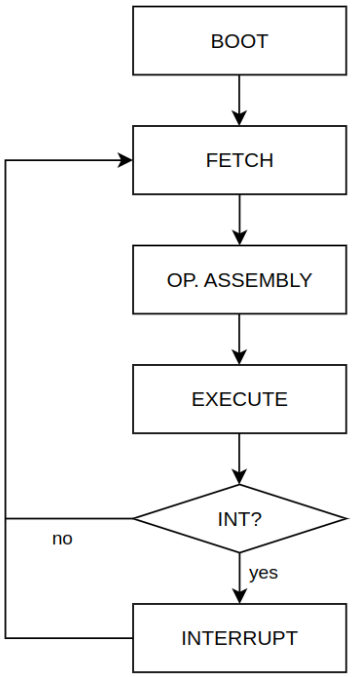

Figura 2.1: Ciclo di esecuizone con interrupt

2.1.4.2 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
- **Autovattorizzate**: A differenza del caso vettorizzato, evita la lettura del codice identificativo, poichè ogni livello di interrupt è collegato al vettore delle ISR auto-vettorizzate e permette di selezionare in maniera "ignorante" l'ISR alla locazione della tipologia di priorità inserita

2.1.4.3 PIC

In generale, nel caso di sistema **vettorizzato**, viene in aiuto il componente **PIC (Programmable Interrupt Controller)** [2.2]. 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). Più dispositivi possono essere connessi in cascata, fino a 8 per un massimo di 64 linee di interruzione. Il PIC accetta richieste di interruzione dai dispositivi di IO connessi alle sue linee e determina, a seconda dell'algoritmo di gestione prioritaria selezionato, quale delle interruzioni simultaneamente attiva ha la priorità più alta. Dopodiché trasmette un segnale sulla linea INT al processore, attende un segnale su INTA (handshaking) e poi trasmette sul bus dati il vettore di 8 bit a cui corrisponde la corretta interruzione sulla tabella delle interruzioni. Il Control Register interno al PIC permette di configurare la gestione prioritaria mediante un'opportuna modifica:

- **Fully nested**: le richieste di interruzione sono ordinate secondo uno schema a priorità fissa che va da IR0 a IR7;
- **Round Robin**: Schema prioritario a rotazione, ovvero la linea di interruzione più prioritaria appena servita diventa la meno prioritaria dopo il servizio;
- **Maschera interruzioni** consente l'inibizione o l'abilitazione delle linee di interruzione.

Il modo di operazione scelto dev essere configurato in fase di inizializzazione del PIC, ma può anche essere dinamicamente cambianto da un apposito programma di gestione. L'Interrupt Request Register (IRR) riceve in ingresso le 8 linee di interruzione provenienti dalle periferiche collegate e ne memorizza lo stato. L'input a questo registro è gestito da un circuito integrato che si occupa della gestione prioritaria delle interruzioni, mentre l'output è il registro In Service Register (ISR) in cui vengono memorizzati solo i segnali di interruzione da servire in accordo alla maschera (IMR). Il Type Register (TR) è un registro di 8 bit che memorizza nei 5 bit più significativi il valore base del vettore da scrivere in output sul bus dati, mentre nei 3 meno significativi uno spiazzamento in accordo alla linea interrompente. Dopo il servizio, l'i-esimo bit di IRR è automaticamente cancellato per riuovere la causa di interruzione.

2.1.5 Estensione del modello IO generale

Un modello di architettura dotato solo di PIA (4.1) è limitato: può gestire solo caratteri, ha a disposizione solo 7 interruzioni e può generare attese infinite con il protocollo di

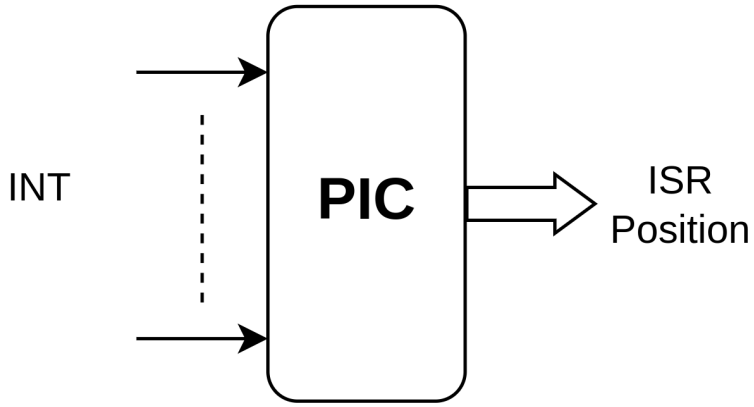

Figura 2.2: PIC(Programmable Interrupt Controller)

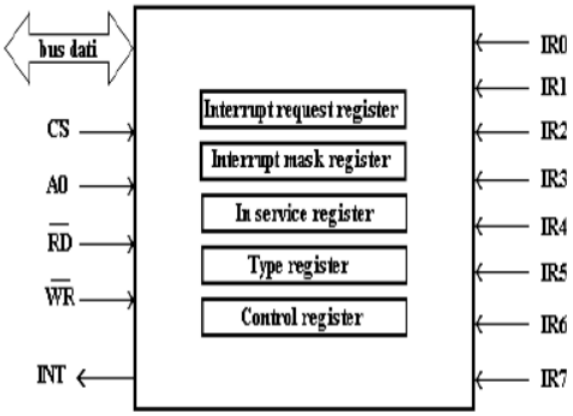

Figura 2.3: PIC: modello di programmazione

handshaking. Per risolvere questi problemi, vengono introdotti nuovi elementi nell'architettura: **DMA** per gestire il trasferimento di messaggi invece di caratteri, **PIC** (2.1.4.3) per superare la limitazione sul numero di ISR indirizzabili e **TIMER** per gestire la temporizzazione e il risolvere il problema delle attese infinite (2.4).

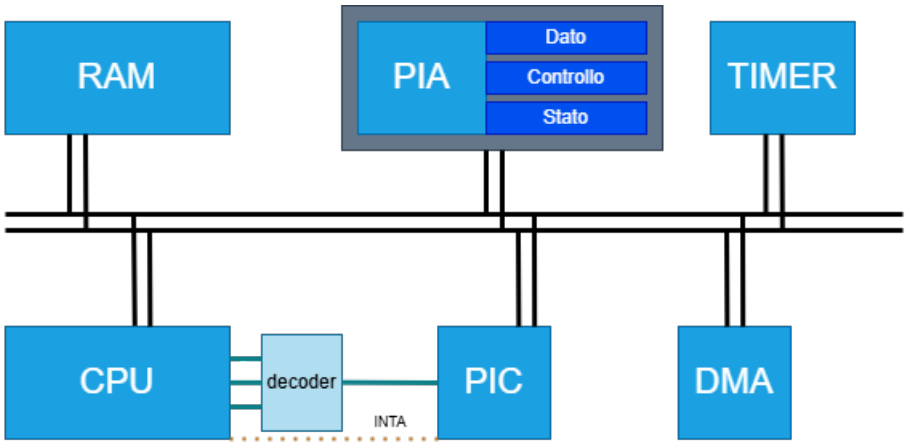

Figura 2.4: Modello IO esteso - schema logico

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 (Direct Memory Access)** è un dispositivo che permette di sollevare il processore dall'onere di trasferire i dati tra varie periferiche. Per precisare meglio tale concetto, il DMA permette di gestire il trasferimento dati tra:

- **Memoria-Periferica**
- **Periferica-Memoria**
- **Memoria-Memroia**

Il suo principio di funzionamento è semplice, ed è schematizzabile su 3 registri principali, dove principalmente si vanno a specificare:

- **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 o l'area di memoria in cui andare a trasferire i dati

La reale architettura del DMA è però ben diversa, essa risulterà più complicata. La maggior complessità dell'architettura proviene da varie tipologie di problematiche che si possono riscontrare all'interno del suo utilizzo, come, ad esempio, l'accesso al BUS dati in maniera concorrente rispetto 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 in base ai vari accessi in memoria ecc.

Il dispositivo a cui si andrà a fare riferimento nelle nostre esercitazioni sarà 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 in ASIM, tale componente è composto di soli 2 canali, quindi è un sua versione semplificata. Scendendo più nei dettagli, tale dispositivo è in grado di sostenere 4 modalità di funzionamento differenti, ovvero:

- **Single**: Ad ogni ciclo, si ferma e lascia altri cicli per il processore. Ricomincia a trasmettere quando la linea di richiesta sarà di nuovo attiva
- **Block**: La linea di richiesta viene controllata una singola volta, una volta attivato il trasferimento, il BUS, sarà rilasciato solo dopo aver finito il trasferimento
- **Demand**: Simile alla modalità Block, con l'unica differenza che il trasferimento prosegue fin tanto che la richiesta è attiva, se dovesse fermarsi, attende, e quando ricomincia, riprende da dove aveva lasciato (non si resetta)
- **Cascade**: Modalità di funzionamento che permette di collegare più DMA in cascata in modo da poter avere dispositivi con 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**

- **Block**

Guardando la figura [2.5], abbiamo il modello architetturale 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, 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 (si vanno a cancellare tutti i registri del dispositivo)
- **HRQ**: Segnale di richiesta del controllo del sistema BUS, che solitamente è collegata all'ingresso HOLD della CPU
- **HLDA**: Segnale proveniente dalla CPU che segnala l'acquisizione del sistema BUS da parte del processore
- **EOP**: Linea di interruzione 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 effetuata 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:

- **DREQ 0 e DREQ 1**: Che sono due segnali che utilizzano le periferiche per richiedere l'accesso al BUS tramite dei cicli DMA
- **DACK 0 e DACK 1**; Sono due segnali che permettono di comunicare alla periferica (da parte del DMA), che sono abilitate per un determinato ciclo DMA

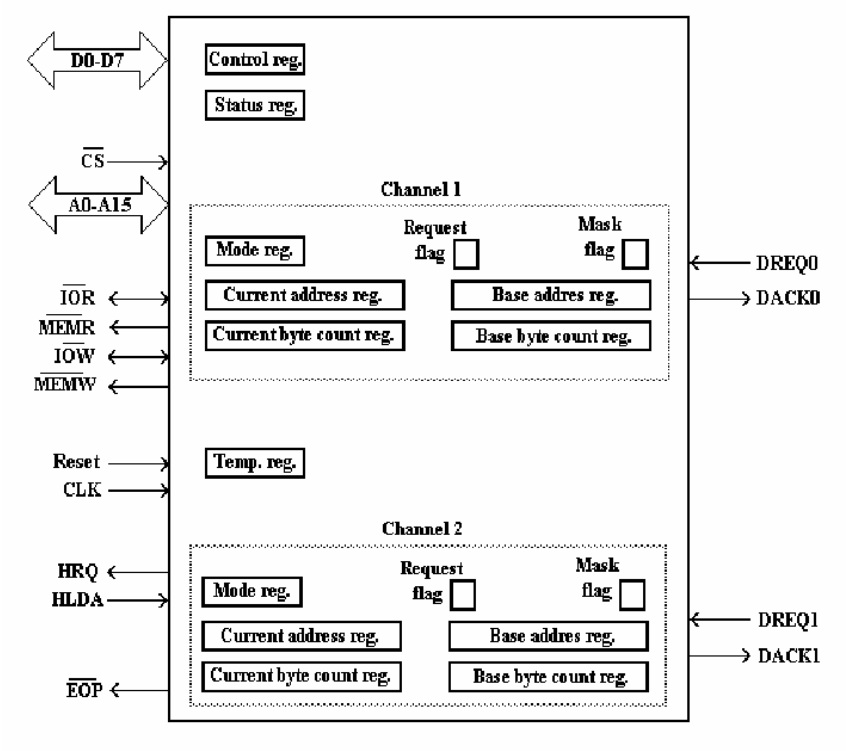

Figura 2.5: DMA a 2 canali

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 perchè 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 architetturale, 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 Istruction 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.

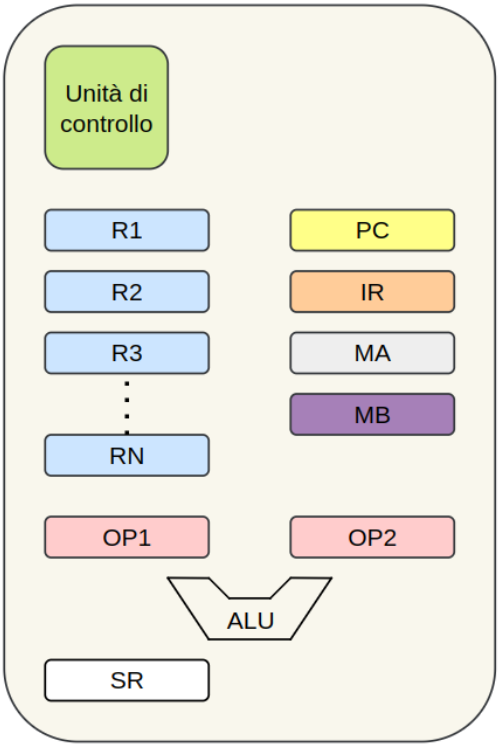

Figura 3.1: Architettura di un processore generico

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 Istruction 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 Istruction 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.

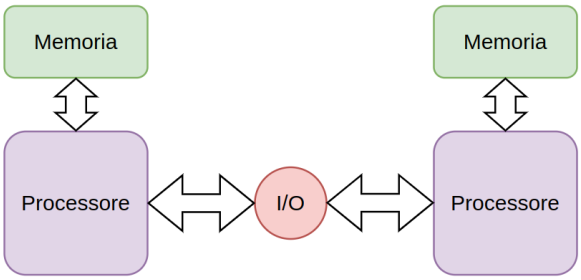

Figura 3.2: Sistemi "gemelli" o sistema multicomputer

- **MISD (Multiple Istruction 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, poichè si cerca di eseguire sempre delle operazioni contemporanee rispetto ai dati.
- **MIMD (Multiple Istruction 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 efficente, poichè 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.

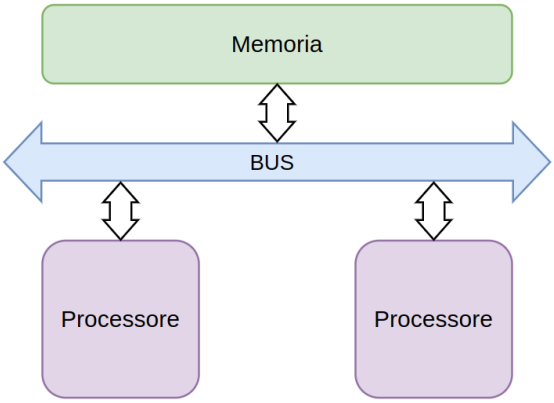

Figura 3.3: Sistema a memoria condivisa

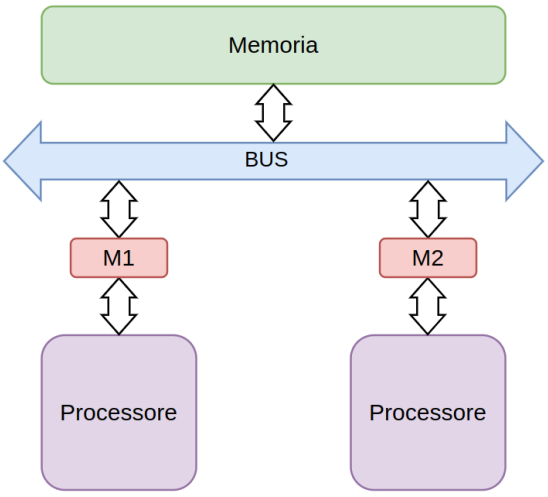

Figura 3.4: Sistema multicore

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].

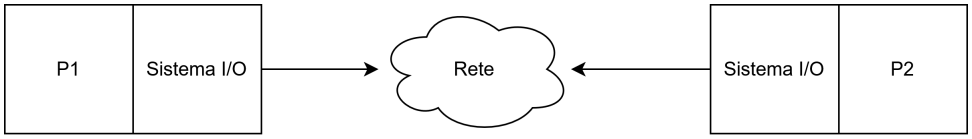

Figura 3.5: Architettura di un sistema Multi-Computer.

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 memoria. 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

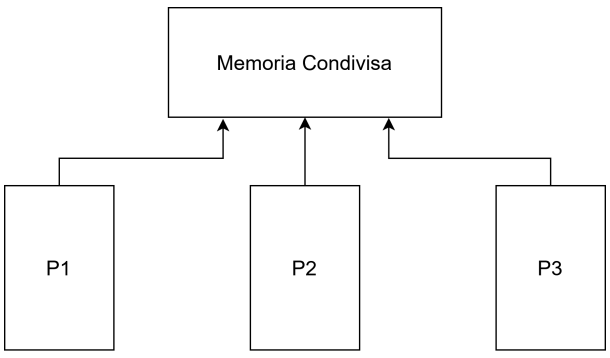

Figura 3.6: Architettura di un sistema Multi-Processore.

altro, cosa che accade tipicamente nei sistemi moderni.

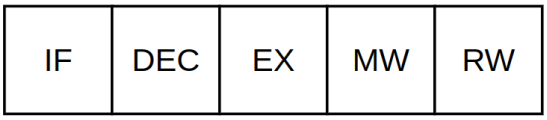

Figura 3.7: Architettura classica pipeline (ordine di esecuzione fasi)

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 [3.7]):

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

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, mente, allo stesso istante, viene caricata l'istruzione i2 (quindi viene effettuata la Istruction Fetch dell'istruzione successiva mentre viene decodificata la precedente). Se eseguiamo tale procedura per tutte le fasi noteremo che ad ogni impulso di clock (a regime), il processore darà un risultato, quindi non ci sarà bisogno di eseguire tutte le istruzioni una per volta, poichè 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 [3.8]).

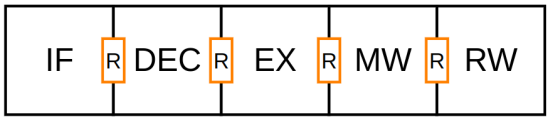

Figura 3.8: Architettura classica pipeline con registri

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*) (3.3.8);

- **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 gesite 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'istruction 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 artimetico-logiche devono essere effettuate prevalentemente tra i registri interni del processore, condizione che caratterizza fortemente i processori RISC;

- Le istruzioni devono avere tutte lunghezza fissa;

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

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

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

3.3.1 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 [3.7], 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;

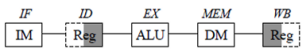

Figura 3.9: Divisone temporale impulso di clock

Come per tutte le architettura 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 3.9;

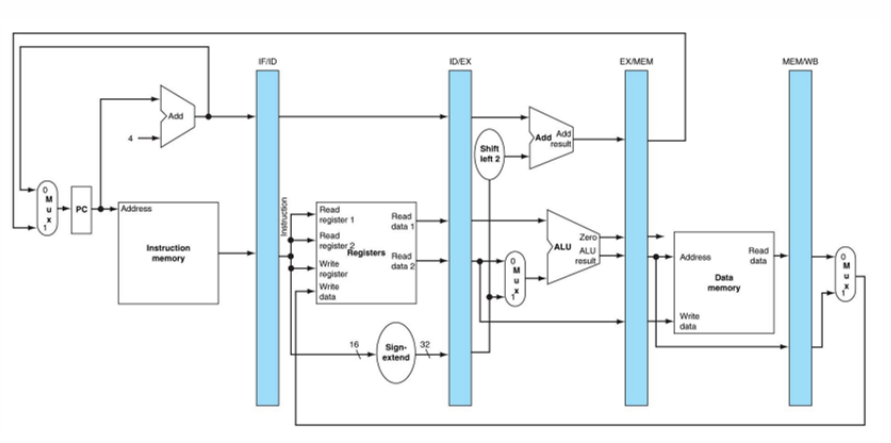

Figura 3.10: Architettura della pipeline MIPS

3.3.2.1 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, perchè 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;
- **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;
- **Istruction Memory**: Prelievo dell'istruzione da eseguire dalla memoria;

3.3.3 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 perchè 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 perchè 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).

3.3.4 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 oeprazioni 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;

3.3.5 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.

3.3.6 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.

3.3.7 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 effetture, 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.

3.3.8 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).

3.3.8.1 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 pre 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 [3.11].

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 visualizzabille all'immagine [3.12]

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

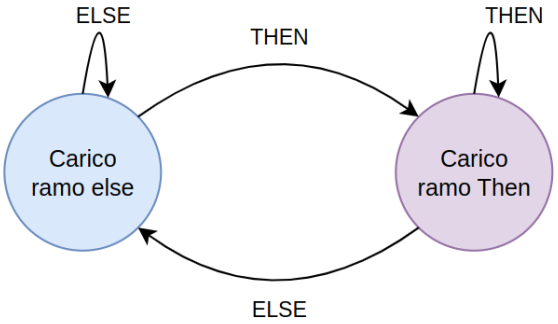

Figura 3.11: Automa della branch prediction base

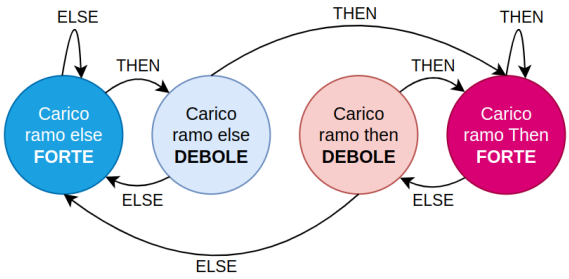

Figura 3.12: Automa della branch prediction avanzato

- 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** [3.13], e differisce dalle classiche memorie indirizzabili quali, RAM o hard disk.

3.3.9 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**

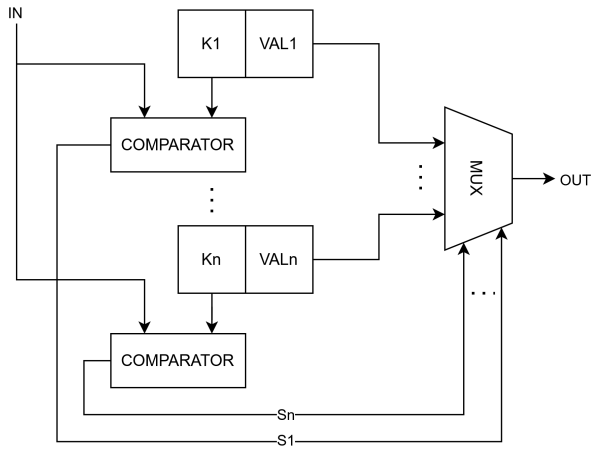

Figura 3.13: Architettura di una memoria associativa.

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 Fowarding**: 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 Fowarding**, 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. 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.

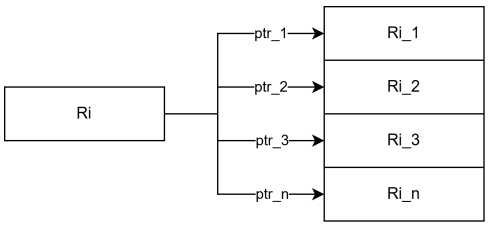

Figura 3.14: Struttura dei registri copia con indirizzamento indiretto.

3.3.10 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 opreazioni 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].

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 pù 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 tempificazione 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

	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.

	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.

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 alcune collisione. Nel nostro esempio ci siamo fermati a realizzare il collision vector per 1 solo caso, ovvero quello in cui una moltiplicazione segue una moltiplicazione. In generale, è necessario costruirlo per tutte le possibili combinazioni di operazioni (*mul* segue *add*, *add* segue *add*, eccetera). Se per ipotesi avessimo n istruzioni da realizzare, sarebbe necessario costruire 2^n vettori, complicando notevolmente il lavoro del compilatore.

Ritorniamo all'esempio e cerchiamo di capire come il compilatore gestisca le collisioni per una generica sequenza di operazioni, che supponiamo essere: *add, mul, mul.* È in-

nanzitutto 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 [3.15]. In particolare,

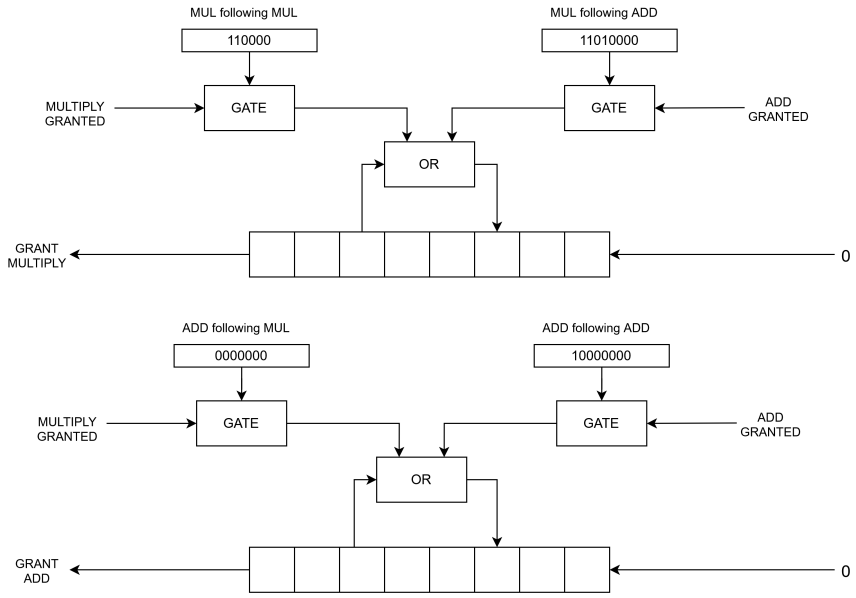

Figura 3.15: Hardware per la gestione delle collisioni.

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

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.

4.1 PIA (Periferal Interface Adapter)

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.[4.1] 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 [4.2]. 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 4.1:

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

Tabella 4.1: Indirizzamento interno

Questi registri sono selezionabili dal processore mediante le linee RS1 ed RS0. Nel dettaglio, i registri di contollo sono a 8 bit suddivisi come indicato in tabella 4.2:

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 4.2: Control Registers

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;
- **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 si 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 [4.2]. 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 tutto. Qundi, 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 sonno le seguenti:

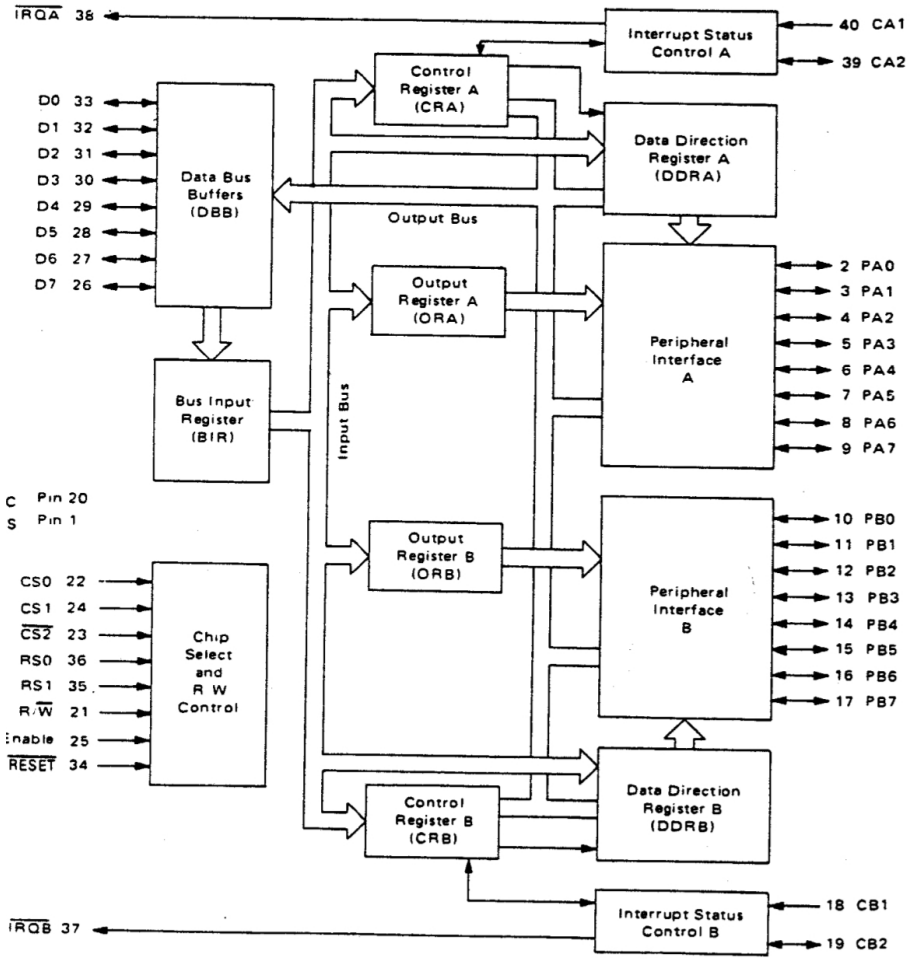

Figura 4.1: Architettura base della PIA

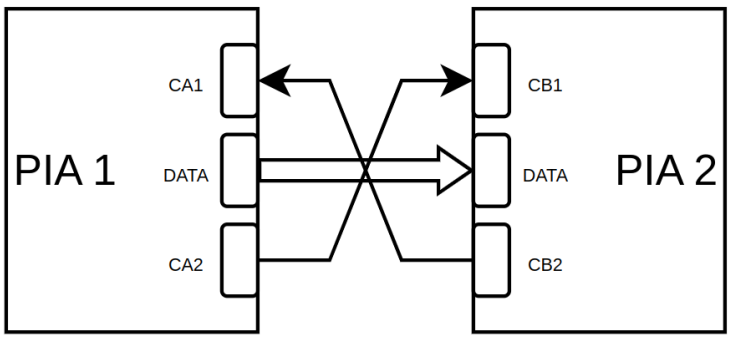

Figura 4.2: Collegamento tra due dispositivi tramite PIA

- **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), poichè 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 fittizzia), 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'interrupt 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 elle 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.

4.1.1 Configurazione della PIA

La PIA prima di essere utilizzata, richiede una propria configurazione, in parte dipendente dall'architettura e da una parte inerente 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).

4.1.1.1 Definizione del file di configurazione

< chiedere all'assistente o al professore il significato delle varie sigle del file di configurazione >

4.1.1.2 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 ad entrambe le tipologie di comportamento (interruzioni e polling) è il seguente:

```
1 *MAIN
2 MAIN   JSR     DVBOUT *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 sottocapitoli [4.1.2] e [4.1.3].

```
1 DVBOUT       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 4.2: 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).

4.1.2 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 [4.1.1.2], 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 sub-routine di configurazione la seguente schematica:

```
1  DVBOUT   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   #%00100100,PIACB     *imposta il registro di controllo
       come indicato precedentemente
4                  *i bit CRB7 e CRB6 sono a sola lettura
5  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,D0  *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,D0 *controllo se ho finito di trasmettere
24          BNE INVIO
25
26  LOOP    JMP LOOP  *ciclo caldo dove il processore ha completato
       la trasmissione
```

4.1.3 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,D0 *legge il registro di stato
4          ANDI.W  #$D8FF,D0 *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 scitto 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
```

4.2 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).

4.2.1 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.

4.2.1.1 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
6     MAIN   JSR     DVBOUT ;inizializza PIA porto B in output
7
8   MOVEA.L #PIACB,A1 ;indirizzo registro di controllo CRB
9   MOVEA.L #PIADB,A2 ;indirizzo registro PRB
10  MOVEA.L #MSG,A0 ;indirizzo area messaggio
11  MOVE.B  DIM,D0  ;dim del messaggio
12
13  CLR D1  ;appoggio
14  CLR D2  ;contatore elementi trasmessi
15
16    INVIO MOVE.B  (A2),D1     *lettura fittizia da PRB: serve per
          azzerare CRB7 dopo il primo carattere, altrimenti resta
        settato con l'ack
17          MOVE.B  (A0)+,(A2)  *carattere corrente da trasferire
                su bus di PIA porto B: dopo la scrittura di PRB,
              CB2 si abbassa
18                  *cio fa abbassare CA1 sulla porta gemella
                      dell'altro sistema generando
19                  *un'interruzione che coincide con il segnale
                      DATA READY
20          ADD.B   #1,D2   *incremento contatore elementi
              trasmessi
21
22    ciclo2  MOVE.B  (A1),D1           *In attesa di DATA
        ACKNOWLEDGE
23          ANDI.B  #$80,D1       *aspetta che CRB7 divenga 1
24          BEQ ciclo2
25
26          CMP D2,D0                 *controllo se ho finito di
              trasmettere
27          BNE INVIO
28
29    LOOP    JMP LOOP                  *ciclo caldo dove il
        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
        direzione di PIA porto B (settando il bit 2 a 0)
2         MOVE.B  #$FF,PIADB        ;accede a DRB e pone DRA=1 :
            le linee di B sono linee di output
3         MOVE.B  #%00100100,PIACB    ;imposta il registro di
            controllo
4         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.

4.2.1.2 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,
              int abilitati)
11        MOVE.W  D0,SR ;pone liv int a 000
12
13    LOOP    JMP LOOP  ;ciclo caldo dove il processore attende
          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)
```

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

```
    ORG $8700

    INT3      MOVE.L  A1,-(A7)    ;salvataggio registri
        MOVE.L  A0,-(A7)
        MOVE.L  D0,-(A7)

        MOVEA.L #PIADA,A1
        MOVEA.L #MSG,A0     ;indirizzo area di salvataggio
        MOVE.B  COUNT,D0  ;contatore corrente degli elementi
           ricevuti


        MOVE.B  (A1),(A0,D0)  ;acquisisce il carattere e lo
           trasferisce in memoria
    *               ;la lettura da PRA fa abbassare CRA7 e CA2 =>
        nell'altro sistema si abbassa CB1
    *               ;cio corrisponde all'attivazione di CRB7 che
       funge da DATA ACKNOWLEDGE

        ADD.B #1,D0
        MOVE.B  D0,COUNT

        MOVE.L  (A7)+,D0    ;ripristino registri
        MOVE.L  (A7)+,A0
        MOVE.L  (A7)+,A1

        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.

4.2.2 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 4.2.1.

4.2.2.1 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 (4.2.1.1), 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     $2007  ;indirizzo di PIA-B controllo
6
7  MAIN  JSR     DVBOUT ;inizializza PIA porto B in output
8
9    MOVEA.L #PIACB,A1 ;indirizzo registro di controllo CRB
10   MOVEA.L #PIADB,A2 ;indirizzo registro PRB
11   MOVEA.L #MSG,A0 ;indirizzo area messaggio
12
13   MOVE.W  SR,D0 ;legge il registro di stato
14   ANDI.W  #$D8FF,D0 ;maschera per reg stato (stato utente, int
        abilitati)
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
     azzerare CRB7 poiche in generale non sappiamo se la macchina e
     ' in reset
19       MOVE.B  (A0),(A2) ;dato su bus di PIA porto B: dopo la
            scrittura di PRB, CB2 si abbassa
20            ;cio fa abbassare CA1 sulla porta gemella dell'altro
                 sistema generando
21            ;un'interruzione che coincide con il segnale DATA READY
22
23       MOVE.B  #1,COUNT
24 LOOP    JMP LOOP  ;ciclo caldo dove il processore attende
     interrupt
```

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

```
1     DVBOUT   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
4 *                ;i bit CRB7 e CRB6 sono a sola lettura
5   RTS
```

Valgono le stesse considerazioni fatte nel caso dell'esercizio (4.2.1.1). 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 signfica 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 (4.2.1.1).

Procedendo con il main, è necessario inviare dal codice il primo carattere, perchè i successivi verrano gestiti dalla ISR relativa. Vediamola quindi nel dettaglio: La piaA 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 trsmettere. 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
              ricevuti
13
14        CMP.B D1,D0
15        BEQ FINE
16
17    INVIO MOVE.B  (A1),D2            ;lettura fittizia da PRB =>
          serve per azzerare CRB7 dopo il primo carattere,
          altrimenti resta settato con l'ack
18        MOVE.B  (A0,D1),(A1)  ;carattere corrente da trasferire
              in D2;
19    *         ;dato su bus di PIA porto B: dopo la scrittura di
          PRB, CB2 si abbassa
20
21        ADD.B #1,D1     ;aggiorno il contatore degli elementi
              trasmessi
22        MOVE.B  D1,COUNT
23
```

```
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
```

4.2.3 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     $2001  ;indirizzo di TERMINAL registro stato/
           controllo
11
12              JSR     DVAIN  ;inizializza PIA porto A
13              JSR     DVBOUT ;inizializza PIA porto B
14              JSR     DVTER  ;inizializza terminal
15              MOVE.W  SR,D0 ;legge il registro di stato
16              ANDI.W  #$D8FF,D0 ;maschera per reg stato (stato
                   utente, int abilitati)
```

```
17              MOVE.W  D0,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 4.3. Come negli altri esercizi, configuriamo il porto B per la scrittura e il porto A per la lettura.

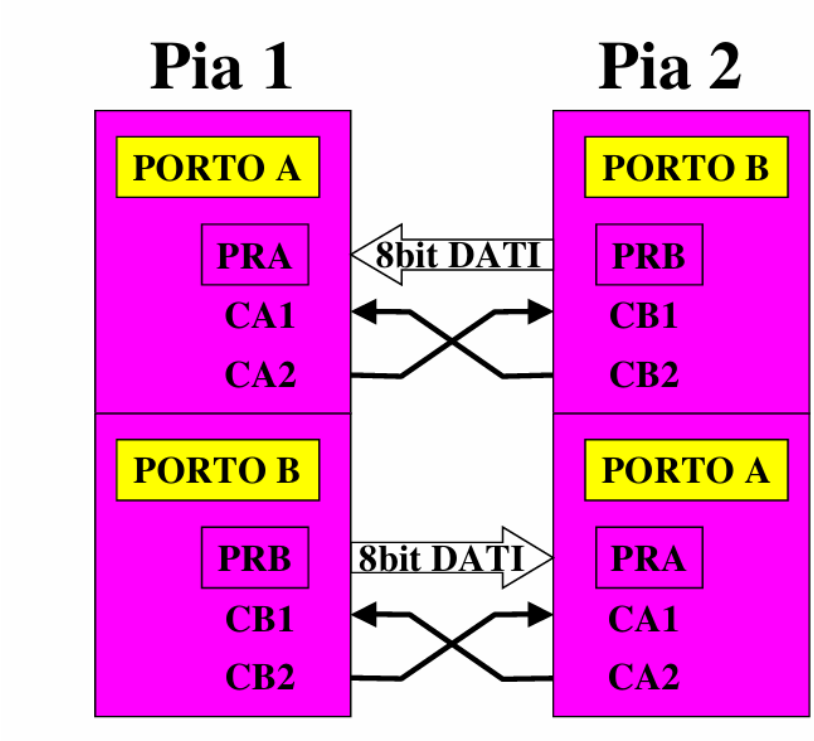

Figura 4.3: Configurazione Full Duplex

Le routine dedicate alla configurazione dei porti A e B sono uguali a quelle viste nell'esercizio 2 di questa sezione (4.2.2). 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 4.4.

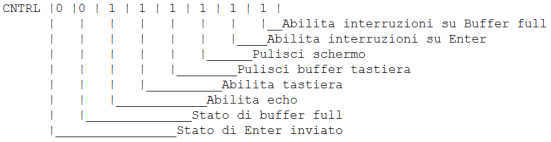

Figura 4.4: Configurazione registro controllo/stato del terminale

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 pro-

veniente 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
         supervisor
3          MOVE.L  A1,-(A7)
4          MOVE.L  A2,-(A7)
5          MOVE.L  D0,-(A7)
6          MOVE.L  D1,-(A7)
7          MOVEA.L #TERD,A0
8          MOVEA.L #PIADB,A1
9          MOVEA.L #PIACB,A2
10
11    INPUT MOVE.B  (A0),D0      ;acquisisci dato da terminal
12
13    *trasferisci il carattere letto alla PIA-B con handshaking
14             MOVE.B  (A1),D1          ;lettura fittizia
15             MOVE.B  D0,(A1)          ;Dato su bus di PIA porto B:
                  dopo la scrittura di PRB, CB2 si abbassa
16    *                ;cio fa abbassare CA1 sulla porta gemella
         dell'altro sistema generando
17    *                ;un'interruzione che coincide con il segnale
         DATA READY
18
19    ciclo2  MOVE.B  (A2),D1      ;In attesa di DATA ACKNOWLEDGE
20         ANDI.B  #$80,D1         ;aspetta che CRB7 divenga 1
21         BEQ ciclo2
22
23    *fine trasferimento e handshaking
24
25         CMP.B      #13,D0     ;Se il carattere ricevuto e' ENTER
26         BNE        INPUT    ;termina altrimenti prossimo carattere
27         ORI.B #$1C,TERC    ;riabilita tastiera ,pulisce buffer e
              video
28         MOVE.L  (A7)+,D1     ;ripristino di D0,D1,A2,A1,A0
29         MOVE.L  (A7)+,D0
30         MOVE.L  (A7)+,A2
31         MOVE.L  (A7)+,A1
32         MOVE.L  (A7)+,A0
33         RTE
```

In questo caso è presente per forza di cose un ciclo improduttivo (ciclo2) perchè bisogna procedere sequenzialmente al set di istruzioni successivo che prevede un salto a INPUT se il carattere ricevuto non è quello finale (ENTER). Dopodichè resetta il terminale riabilitando tastiera e pulendo buffer e video scrivendo nel registro di controllo in accordo a quanto esposto nella figura 4.4, 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.

```
    ORG $8600
INT2  MOVE.L  A0,-(A7)     ;push di A0,A1,A2,D0,D1 in stack supervisore
      MOVE.L  A1,-(A7)
      MOVE.L  A2,-(A7)
      MOVE.L  D0,-(A7)
      MOVE.L  D1,-(A7)
      MOVE.L  D2,-(A7)
      MOVEA.L #TERD,A0
      MOVEA.L #PIADB,A1
      MOVEA.L #PIACB,A2
      MOVE.B  #255,D2    ;#caratteri da trasferire

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

*trasferisci il carattere letto alla PIA-B con handshaking
      MOVE.B  (A1),D1          ;lettura fittizia => serve per azzerare CRB7 dopo il primo carattere, altrimenti resta settato con l'ack
      MOVE.B  D0,(A1)          ;Dato su bus di PIA porto B: dopo la scrittura di PRB, CB2 si abbassa
*             ;cio fa abbassare CA1 sulla porta gemella dell' altro sistema generando
*             ;un'interruzione che coincide con il segnale DATA READY


ciclo3  MOVE.B  (A2),D1    ;In attesa di DATA ACKNOWLEDGE
      ANDI.B  #$80,D1    ;aspetta che CRB7 divenga 1
      BEQ ciclo3

*fine trasferimento e handshaking

      DBRA      D2,SWAP ;contatore di bit inviati
      ORI.B #$1C,TERC ;riabilita tastiera ,pulisce buffer e video
      MOVE.L  (A7)+,D2  ;ripristino di D0,D1,A2,A1,A0
      MOVE.L  (A7)+,D1
      MOVE.L  (A7)+,D0
      MOVE.L  (A7)+,A2
      MOVE.L    (A7)+,A1
      MOVE.L    (A7)+,A0
      RTE
```

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

```
    ORG $8700

INT3      ANDI.B    #%11101001,TERC ;disabilita: tastiera, cancella video,interruzioni su enter
```

```
4           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
                trasferisce a Terminal
12      *           ;la lettura da PRA fa abbassare CRA7 e CA2 =>
            nell'altro sistema si abbassa CB1
13      *           ;cio corrisponde all'attivazione di CRB7 che
            funge da DATA ACKNOWLEDGE
14
15          MOVE.L  (A7)+,D0    ;ripristino registri
16          MOVE.L  (A7)+,A0
17          MOVE.L  (A7)+,A1
18
19          ORI.B #$12,TERC ;riabilita tastiera e interruzioni su
                enter
20          RTE
21
22
23          END BEGIN
```

4.2.4 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.

4.3 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.

4.3.1 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 4.5

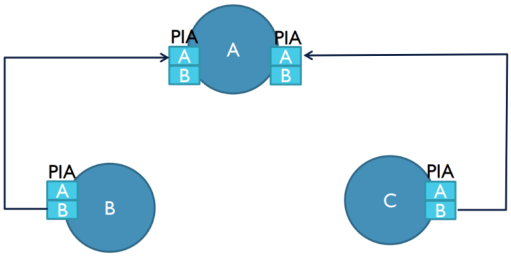

Figura 4.5: Schema logico

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:

```
    if (sem == verde){      \\istruzione atomica TAS
        sem = rosso;
        if (possesso != 1){ \\possesso non di C
            possesso = 0;
            sem = verde;
            leggo_car_b();
            car_counter_b++;
            if (car_counter_b==N){ \\fine trasmissione
                if(c_sospeso){
                    possesso = 1; \\possesso di C
                    leggo_car_c();
                    car_counter_c++;
                }
                else{
                    possesso=-1;
                }
            }
        }
        else{
            sem = verde;
            leggo_car_c();
            car_counter_c++;
        }
        return 0;
    }
    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.

4.3.2 Esercizio 2 - Prova intercorso 2023

Un sistema è compoto 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 ricebuti 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:

```
    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 4.3.1, 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 simultaore ASIMTale file va generato secondo il seguente path:
Assemble -> Assemble File <Nome_File>.a68
Fatta tale operazione, nella cartella dove vi è salvato il file à68 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 volra 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 memorià mostrerà la memoria all'indirizzo richiesto in alto.