Appunti di

Architettura e progettazione dei calcolatori

Corso di laurea magistrale in

Ingegneria Informatica

Indice

Introduzione

Ciao a tutti Forza Napoli

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 facco 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)
    *o
    MOVEA.L (A0),(A1)
    *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
```

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
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).

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. 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
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)
8    RTS
```

1.2.5 Codici Logici

I codici logici sono operazioni che possono essere effettuate su degli operandi operando bit a bit. 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

```
    ASL #1, D0        * Shift aritmetico a sinistra di 1 bit (
       mantiene il segno)
    ASR #1, D0        * Shift aritmetico a destra di 1 bit

    LSL #1, D0        * Shift logico a sinistra di 1 bit (riempie
       con 0)
    LSR #1, D0        * Shift logico a destra di 1 bit

    ROL #1, D0        * Rotazione a sinistra di 1 bit (il bit piu'
       alto rientra da destra)
    ROR #1, D0        * Rotazione a destra di 1 bit

    ROXL #1, D0       * Rotazione a sinistra con Carry
    ROXR #1, D0       * Rotazione a destra con Carry
```

1.2.7 Codici di Confronto

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

```
    CMP #5, D0        * Confronta D0 con 5 (D0 - 5, senza
       modificare D0, aggiorna i flag)
    CMP.W D0, D1      * Confronta D1 con D0 (D1 - D0, aggiorna solo
        i flag)
    CMP.L (A0), D0    * Confronta D0 con il valore in memoria
       puntato da A0

    CMPI #10, D0      * Confronto immediato con D0 (D0 - 10,
       aggiorna solo i flag)

    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

```
    TST D0            * Testa D0 (controlla se e' zero o negativo,
       senza modificarlo)
    TST.W (A0)        * Testa il valore in memoria puntato da A0

    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
```

```
8       BSET #5, (A0)     * Imposta il bit 5 della memoria puntata da
            A0
9
10      BCLR #3, D0       * Azzera il bit 3 di D0
11      BCLR #5, (A0)     * Azzera il bit 5 della memoria puntata da A0
12
13      BCHG #3, D0       * Inverte il bit 3 di D0 (0 -> 1, 1 -> 0)
14      BCHG #5, (A0)     * Inverte il bit 5 della memoria puntata da
            A0
15
16      TAS D0            * Testa e imposta il bit piu' alto (7) di D0
17      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:

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

Esempio di un If-Then-else

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

```
4
5                   * ELSE block
6                   MOVE.L  #0, D2          * D2 = 0
7                   BRA     END_IF          * Salta oltre il blocco THEN
                        per evitare di eseguirlo
8
9   THEN_BLOCK:     MOVE.L  #1, D2          * D2 = 1
10
11  END_IF:         * Codice successivo...
```

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

1.2.8.2 Ciclo FOR

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

```
1                   MOVE.L  #0, D0          * Inizializza il contatore
                        D0 = 0
2
3   FOR_LOOP:       CMP.L   #10, D0         * Confronta D0 con 10
4                   BGE     FOR_END         * Se D0 >= 10, esce dal
                        ciclo
5
6                   * Corpo del ciclo
7                   NOP                     * Istruzione di esempio (
                        da sostituire con il codice reale)
8
9                   ADDQ.L  #1, D0          * Incrementa D0 di 1
10                  BRA     FOR_LOOP        * Ripete il ciclo
11
12  FOR_END:        * Codice successivo dopo il ciclo
```

1.2.8.3 Ciclo While

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

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

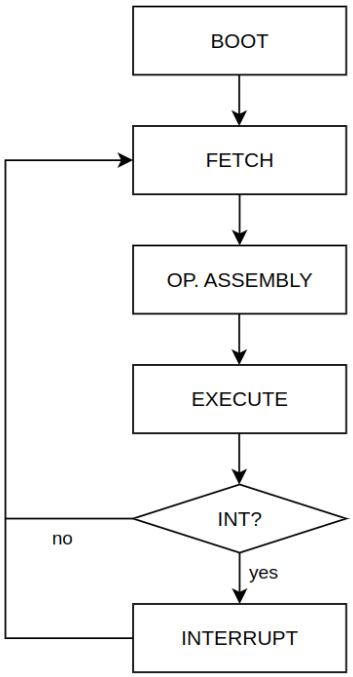

Figura 2.1: Ciclo di esecuizone con interrupt

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 (3.1) è limitato: può gestire solo caratteri, ha a disposizione solo 7 interruzioni e può generare attese infinite con il protocollo di

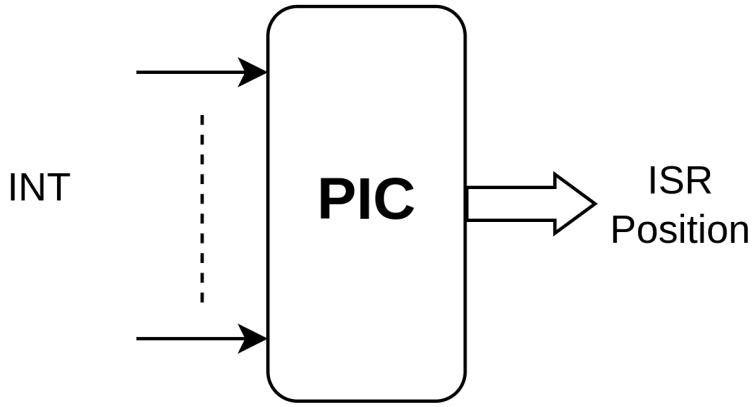

Figura 2.2: PIC(Programmable Interrupt Controller)

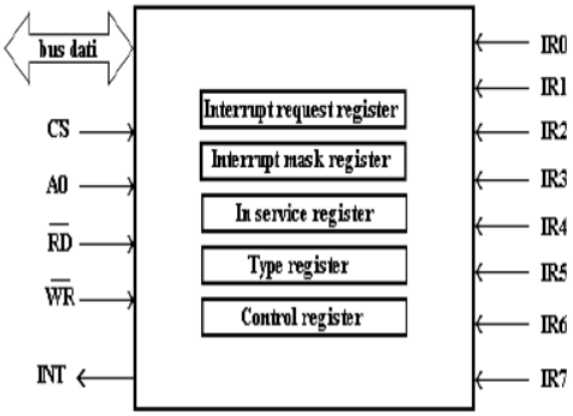

Figura 2.3: PIC: modello di programmazione

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).

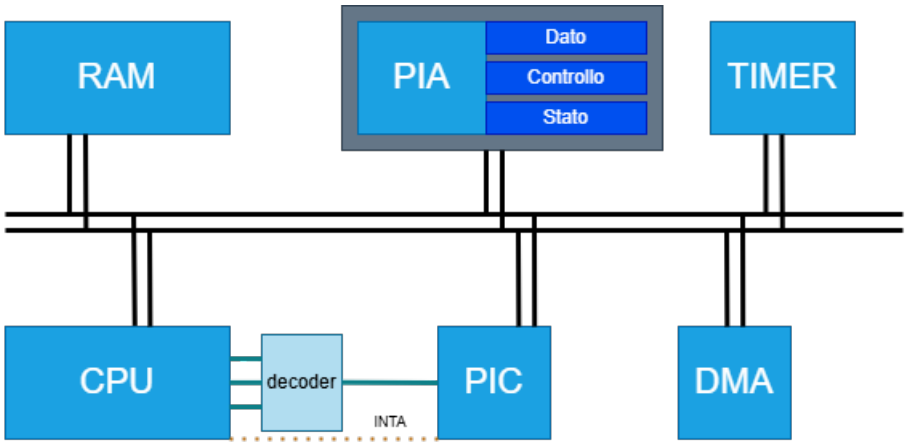

Figura 2.4: Modello IO esteso - schema logico

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

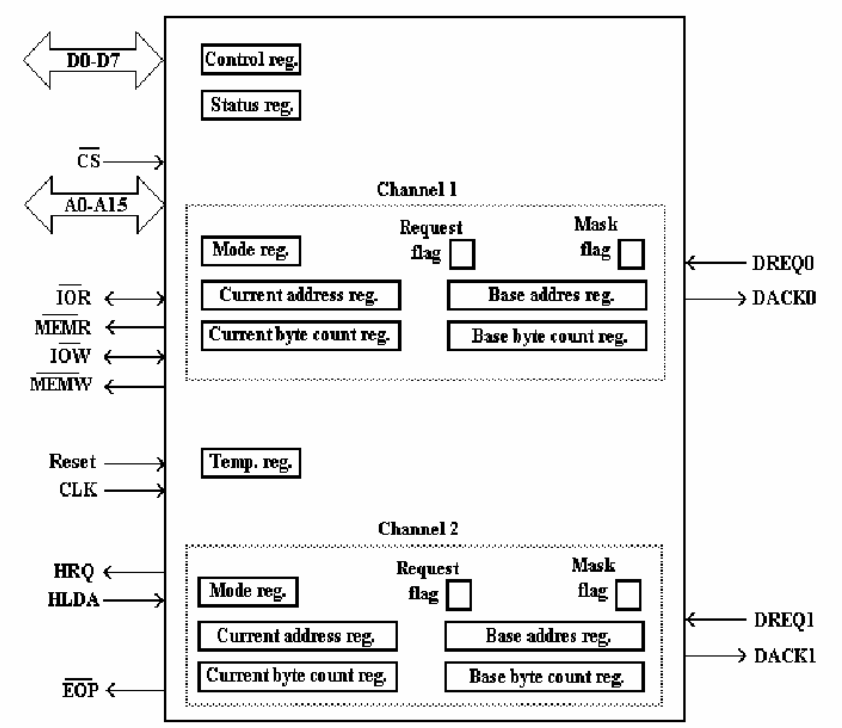

Figura 2.5: DMA a 2 canali

Capitolo 3

Esercitazioni

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

3.1 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.[3.1] 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 [3.2]. 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 3.1:

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 3.1: 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 3.2:

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**
	IRB1	*IRB2*	*controllo CB2*			*Accesso DRB*	*controllo CB1*	

Tabella 3.2: 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 devo andare a dare ai registri. Una possibile architettura di collegamento è quella visibile all'immagine [3.2]. 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 o modello di programmazione della PIA), 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

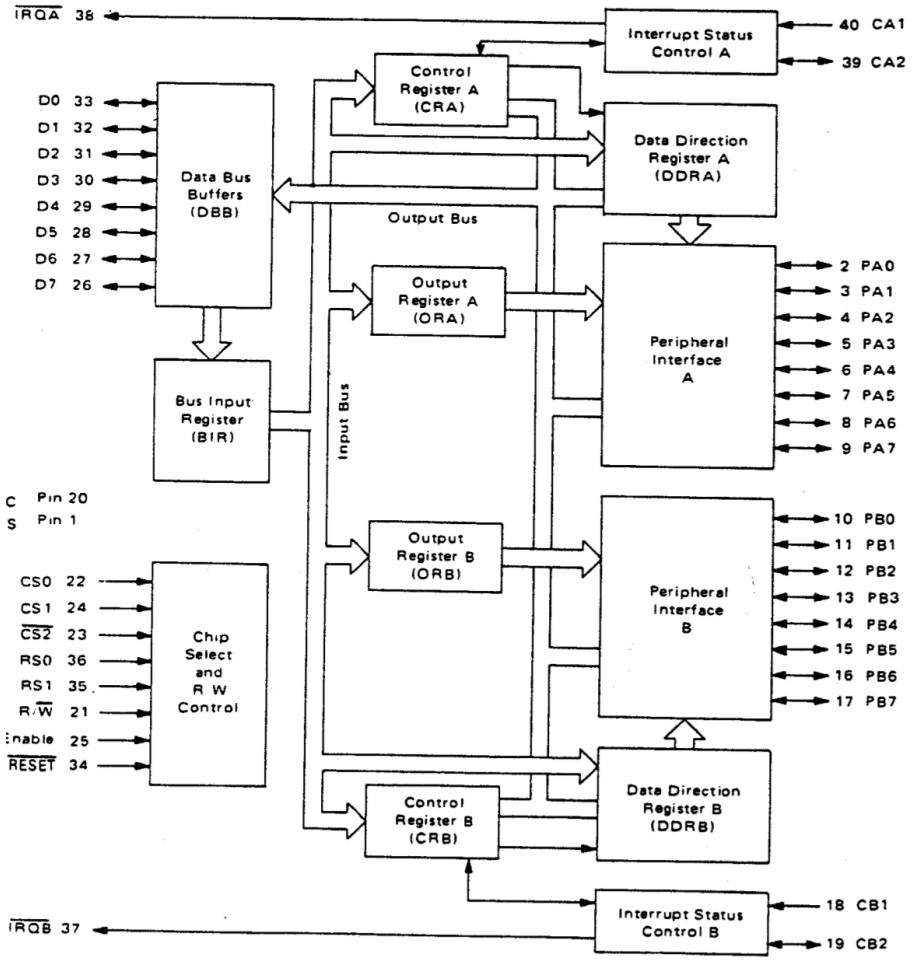

Figura 3.1: Architettura base della PIA

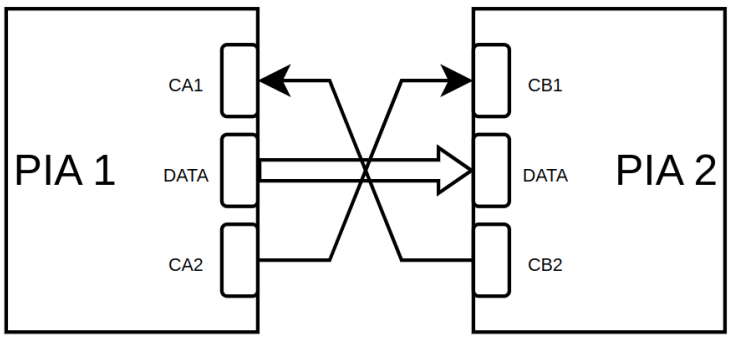

Figura 3.2: Collegamento tra due dispositivi tramite PIA

di ricezione dalla fase di trasmissione. In fase di **trasmissione** le operazioni che si vanno ad effettuare sonno le seguenti:

- **Inserimento del dato**: La periferica posiziona il dato sul bus dati collegato in maniera diretta con la PIA. Una volta effettuato l'inserimento, la periferica tramite la linea CA1 invia un segnale di *dato pronto* alla PIA, quindi sulla transizione di CA1, in accordo alla configurazione, si propaga un'interruzione al processore. Iternamente, 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 CA2, che viene interpretato dalla periferica come *registro dati libero*.
- **Lettura fittizzia**: 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 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 e di poter proseguire nelle mie 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.

3.1.1 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 senza).

3.1.1.1 Definizione del file di configurazione

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

3.1.1.2 Definizione dei registri di controllo e dato

Il registro di controllo ed il registro di stato vengono definiti 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 vanno a settare 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 [3.1.2] e [3.1.3].

```
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 3.2: 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 0) o se vengono *mascherate*, mentre il bit 1 serve a stabilire se il bit 7 si alza 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).

3.1.2 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 [3.1.1.2], dovrò impostare come controllo la seguente sequenza: 00100100, dove indichiamo che non vi è bisogno dell'uso delle interrupt e aggiungere i dettagli una volta definiti tutti i bit

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
```

3.1.3 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
```

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 L̇IS, che poi sarà inserito all'interno del simultaore ASIMṪale file va generato secondo il seguente path:
Assemble -> Assemble File <Nome_File>.a68
Fatta tale operazione, nella cartella dove vi è salvato il file ȧ68 dovrebbe essersi generato il file L̇IS
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 L̇IS 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 L̇IS. 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.