Politecnico di Torino
Laurea Triennale in Ingegneria Informatica

appunti di
Calcolatori elettronici

Autori principali: Luca Ghio
Docenti: Matteo Sonza Reorda
Anno accademico: 2011/2012
Versione: 1.0.4.1
Data: 10 maggio 2016

Ringraziamenti

Oltre agli autori precedentemente citati, quest'opera può includere contributi da opere correlate su WikiAppunti e su Wikibooks, perciò grazie anche a tutti gli utenti che hanno apportato contributi agli appunti *Calcolatori elettronici* e al libro *Calcolatori elettronici*.

Informazioni su quest'opera

Quest'opera è pubblicata gratuitamente. Puoi scaricare l'ultima versione del documento PDF, insieme al codice sorgente LaTeX, da qui: http://lucaghio.webege.com/redirs/11

Quest'opera non è stata controllata in alcun modo dai professori e quindi potrebbe contenere degli errori. Se ne trovi uno, sei invitato a correggerlo direttamente tu stesso realizzando un commit nel repository Git pubblico o modificando gli appunti *Calcolatori elettronici* su WikiAppunti, oppure alternativamente puoi contattare l'autore principale inviando un messaggio di posta elettronica a artghio@tiscali.it.

Licenza

Indice

1 L'evoluzione dei sistemi di elaborazione **7**
- 1.1 Ere . . . 7
  - 1.1.1 Era meccanica (sino al 1945) . . . 7
  - 1.1.2 Era elettronica (dal 1945 al 1975) . . . 7
  - 1.1.3 Era del VLSI (dal 1975 ad oggi) . . . 8
- 1.2 Architetture di processori . . . 8
  - 1.2.1 General-purpose . . . 8
  - 1.2.2 Special-purpose . . . 8

2 Il progetto di circuiti logici **9**
- 2.1 Progettazione . . . 9
  - 2.1.1 Livelli di progettazione . . . 9
- 2.2 Livello di porte logiche . . . 10
  - 2.2.1 Circuiti combinatori . . . 10
  - 2.2.2 Sistemi sequenziali . . . 13
- 2.3 Livello dei registri . . . 15
  - 2.3.1 Componenti generici . . . 15
  - 2.3.2 Moduli aritmetici . . . 16
  - 2.3.3 Comparatore a 4 bit . . . 17
  - 2.3.4 Registri . . . 17
  - 2.3.5 Contatori . . . 18
  - 2.3.6 Interfacce al bus . . . 19
  - 2.3.7 Memorie . . . 19
  - 2.3.8 Moduli programmabili . . . 20
- 2.4 Cenni sul consumo . . . 21

3 I processori **22**
- 3.1 Elaborazione delle istruzioni . . . 22
  - 3.1.1 Fase di fetch . . . 22
  - 3.1.2 Fase di esecuzione . . . 22

4 Architettura 8086 **24**
- 4.1 Registri . . . 24
  - 4.1.1 Classificazione . . . 24
  - 4.1.2 Status register . . . 25
- 4.2 Accesso alla memoria . . . 25
  - 4.2.1 Indirizzi di memoria . . . 25
  - 4.2.2 Segmento di stack . . . 26

5 Unità di controllo **27**
5.1 Microistruzioni . . . 27
5.1.1 Trasferimento di dati tra registri . . . 28
5.1.2 Lettura e scrittura di dati in memoria . . . 28
5.1.3 Passaggio di operandi e ritorno del risultato di un'operazione . . . 28
5.1.4 Esempi . . . 28
5.2 Tipi di unità di controllo . . . 28
5.2.1 Unità di controllo cablata . . . 29
5.2.2 Unità di controllo microprogrammata . . . 30
5.3 Unità di controllo dell'8088 . . . 31

6 Introduzione alle memorie **32**
6.1 Livelli di memorie . . . 32
6.2 Parametri . . . 33
6.2.1 Costo . . . 33
6.2.2 Prestazioni . . . 33
6.2.3 Altri parametri . . . 34
6.3 Modi di accesso . . . 34
6.4 Conservazione dei dati . . . 34

7 Le memorie ad accesso casuale **35**
7.1 Segnali di controllo e linee di stato . . . 35
7.2 Struttura interna . . . 35
7.2.1 Organizzazione a vettore . . . 36
7.2.2 Organizzazione a matrice bidimensionale . . . 36
7.3 Memorie non volatili . . . 36
7.3.1 Memorie ROM . . . 36
7.3.2 Memorie PROM . . . 37
7.3.3 Memorie EPROM . . . 37
7.3.4 Memorie EEPROM . . . 37
7.3.5 Memorie flash . . . 37
7.4 Memorie RAM . . . 37
7.4.1 Memorie RAM statiche . . . 37
7.4.2 Memorie RAM dinamiche . . . 38

8 Le memorie cache **40**
8.1 Prestazioni . . . 40
8.1.1 Gestione dei miss . . . 40
8.1.2 Aggiornamento dei dati in cache . . . 40
8.2 Mapping . . . 41
8.2.1 Direct mapping . . . 41
8.2.2 Associative mapping . . . 41
8.2.3 Set associative mapping . . . 42
8.3 Architettura Harvard . . . 42
8.4 Cache multiple . . . 42

9 Le memorie ad accesso seriale **43**
9.1 Dischi fissi . . . 43
9.1.1 RAID . . . 44
9.2 Memorie a nastro . . . 46
9.3 Memorie ottiche . . . 46

10 La memoria virtuale **47**

11 La gestione dei dispositivi input/output **48**
11.1 Modalità di indirizzamento . . . 48
11.1.1 Modalità Memory-Mapped-IO . . . 48
11.1.2 Modalità Isolated-IO . . . 48
11.2 Sincronizzazione . . . 48
11.2.1 I/O programmato . . . 48
11.2.2 Interrupt . . . 49
11.2.3 Soluzione ibrida . . . 50
11.3 DMA . . . 50
11.3.1 Fasi di trasferimento . . . 51
11.3.2 Modi di funzionamento . . . 51

12 Intel 8255 **52**
12.1 Modalità . . . 52
12.1.1 Modo 0: Basic Input/Output . . . 52
12.1.2 Modo 1: Strobed Input/Output . . . 52
12.1.3 Modo 2: Bidirectional Input/Output . . . 53

13 Intel 8259 **54**
13.1 Struttura . . . 54
13.2 Programmazione . . . 55
13.2.1 ICW . . . 55
13.2.2 OCW . . . 55

14 I bus **56**
14.1 Implementazione . . . 56
14.2 Sincronizzazione . . . 56
14.2.1 Bus sincroni . . . 56
14.2.2 Bus asincroni . . . 57
14.3 Arbitraggio . . . 57
14.3.1 Arbitraggio distribuito . . . 57
14.3.2 Arbitraggio centralizzato . . . 57

15 Le architetture a pipeline **59**
15.1 Architettura RISC . . . 59
15.1.1 Instruction set . . . 59
15.1.2 Stalli . . . 59
15.1.3 Vincoli di dipendenza . . . 60
15.1.4 Istruzioni di salto . . . 60
15.1.5 Vantaggi . . . 60
15.2 Processori superscalari . . . 60
15.2.1 Completamento non-in-ordine . . . 61
15.3 Processori multithread . . . 61
15.4 Processori multicore . . . 61

I Assembler 8086 **62**

16 Introduzione **63**
16.1 Struttura generale di un programma . . . 63
16.2 Istruzioni . . . 63
16.2.1 Input/output . . . 63

17 Informazioni generali **64**
17.1 Modi di indirizzamento . . . 64
17.2 Pseudo-istruzioni . . . 65
17.3 Operatori . . . 65

18 Istruzioni di trasferimento dati **66**

19 Istruzioni aritmetiche **67**
19.1 Istruzioni `ADD` e `SUB` . . . 67
19.2 Istruzione `CBW` . . . 67
19.3 Istruzione `CWD` . . . 67
19.4 Istruzione `ADC` . . . 67
19.5 Istruzione `SBB` . . . 67
19.6 Istruzione `INC` e `DEC` . . . 68
19.7 Istruzione `NEG` . . . 68
19.8 Istruzione `MUL` e `IMUL` . . . 68
19.9 Istruzione `DIV` e `IDIV` . . . 68

20 Istruzioni per il controllo del flusso **69**
20.1 Istruzioni di salto . . . 69
20.1.1 Tipi di salto . . . 69
20.1.2 Istruzioni di salto incondizionato . . . 69
20.1.3 Istruzioni di salto condizionato . . . 69
20.2 Istruzioni di iterazione . . . 70

21 Istruzioni per la manipolazione dei bit **71**
21.1 Istruzioni logiche . . . 71
21.2 Istruzioni di scorrimento . . . 71
21.2.1 Istruzioni di shift . . . 71
21.2.2 Istruzioni di rotazione . . . 72

22 Le procedure **73**
22.1 Struttura . . . 73
22.2 Località . . . 73
22.3 Stack . . . 73
22.3.1 Memorizzazione di variabili locali . . . 73
22.3.2 Backup del contenuto dei registri . . . 73
22.3.3 Passaggio di parametri . . . 74

23 Istruzioni per il controllo del processore **75**
23.1 Istruzioni per la gestione delle interruzioni . . . 75
23.1.1 Interrupt esterni . . . 75
23.1.2 Interrupt software . . . 75
23.2 Altre istruzioni . . . 75

24 Formato delle istruzioni macchina, tempi di esecuzione **77**
24.1 Formato delle istruzioni macchina . . . 77
24.1.1 Primo byte . . . 77
24.1.2 Secondo byte . . . 77
24.1.3 Esempi di istruzioni in base alla lunghezza . . . 77
24.2 Tempi di esecuzione . . . 78

Capitolo 1

L'evoluzione dei sistemi di elaborazione

Un **sistema di elaborazione** è un sistema, generalmente dotato di input e output, in grado di eseguire operazioni logiche ed aritmetiche su dati.

Un calcolatore deve rappresentare delle informazioni, manipolare delle informazioni tramite operazioni automatizzate (algoritmi), e memorizzare dei dati.

1.1 Ere

1.1.1 Era meccanica (sino al 1945)

I dispositivi sono di tipo **meccanico**. Le prime macchine rappresentano i numeri decimali e svolgevano le operazioni elementari.

Nell'Ottocento, nascono le macchine programmabili, ovvero delle macchine per automatizzare processi che possono essere programmati dall'utente.

Negli anni '40 si passa ai **relè** elettromagnetici, che rappresentano numeri binari tramite on/off, ma hanno ancora parti in movimento.

1.1.2 Era elettronica (dal 1945 al 1975)

Nasce l'informatica teorica: si comincia a studiare la complessità degli algoritmi e a determinare ciò che è computabile o no.

I relè sono sostituiti da **tubi a vuoto**/valvole, basati sul movimento degli elettroni, che però sono ancora pesanti e di grandi dimensioni.

L'EDVAC (progettato nel 1951) passa dalla rappresentazione decimale a quella binaria, e ha una memoria che oltre a memorizzare i dati memorizza anche il programma.

L'**architettura Harvard** si differenzia dall'**architettura di Von Neumann** per la separazione della memoria dati e della memoria codice (si veda la sezione 8.3).

Prima generazione (anni '50)

Inizia la produzione commerciale di macchine basate su tubi a vuoto, che però sono molto semplici e richiedono l'uso del linguaggio macchina (strettamente dipendente dalla macchina).

Seconda generazione (anni '60)

Compare il **transistor**: più piccolo, più risparmioso e più veloce. Le CPU introducoro il supporto alla rappresentazione dei numeri in virgola mobile. Iniziano a comparire i linguaggi di alto livello.

Compaiono i primi sistemi operativi, che possono eseguire programmi in batch (ovvero in sequenza automatica), e possono supportare più utenti per volta.

Terza generazione (anni '65)

Nascono i circuiti integrati, e si introducono le memorie a semiconduttore (simili a quelle odierne).

1.1.3 Era del VLSI (dal 1975 ad oggi)

La **Very Large Scale of Integration** (VLSI) è la possibilità di integrare in un unico chip decine di migliaia di transistor.

I **chip** in silicio vengono inseriti in un **package** con pin (DIP, PGA, BGA).

Legge di Moore Il massimo numero di transistor integrabili su un singolo circuito raddoppia ogni 18/24 mesi.

I miglioramenti di prestazioni non sono semplicemente dovuti a miglioramenti di tipo tecnologico, perché oltre all'aumento delle frequenze e del numero di transistor c'è un miglioramento dell'architettura.

1.2 Architetture di processori

1.2.1 General-purpose

I **microprocessori**, nati alla fine degli anni '70, sono dei processori integrati a bordo di singoli circuiti integrati. I computer basati su microprocessori sono detti **microcomputer**.

Nel mercato general-purpose sono importanti le prestazioni:

- **CISC:** (fino agli anni '70) per eseguire un'istruzione è necessario un certo numero di colpi di clock;
- **RISC:** (anni '80) completano un'istruzione in un solo colpo di clock, e hanno un set di istruzioni più ridotto;
- **superscalari:** (anni '90) a ogni colpo di clock completano più di un'istruzione;
- **multicore:** (anni 2000) circuiti integrati che integrano più processori (detti core).

1.2.2 Special-purpose

I sistemi special-purpose sono progettati per eseguire una sola applicazione. Nel mercato special-purpose, generalmente nei sistemi embedded, non sono fondamentali le prestazioni, ma il costo di sviluppo/progettazione e il costo di produzione dell'elettronica. I **microcontrollori** sono dei sistemi programmabili che, integrando discrete CPU, memoria e porte, si interfacciano con dei dispositivi di input/output, eseguono delle operazioni semplici e costano poco.

- **soluzione software (SW):** si compra un microprocessore/microcontrollore già esistente e si sviluppa solo il software $\Rightarrow$ costo di sviluppo basso;
- **soluzione hardware (HW):** si costruisce un hardware specifico integrando solo i dispositivi specifici per compiere una funzione, per esigenze di dimensioni o di consumo di energia $\Rightarrow$ bassa flessibilità (una volta realizzata non si può più modificare), ma personalizzabilità.

I dispositivi **System on Chip** integrano al loro interno un numero significativo di componenti (processori, memoria, moduli) che sono specifici per un'applicazione, ma sono più complicati dei microcontrollori.

Capitolo 2

Il progetto di circuiti logici

Per fare un circuito con determinate **specifiche** (costo, prestazioni...), si scelgono dei componenti, avendo informazioni di base sulla loro funzionalità, e li si assembla (**topologia**).

I **sistemi digitali** elaborano input digitali in output digitali (= binari).

Il circuito dispone di una libreria di componenti che rappresenta le **porte logiche** da interconnettere, rappresentabili con un grafo con:

- vertici = porte logiche
- archi = linee per il trasferimento dei dati tra le porte

Un circuito è definito da:

- **struttura:** da quali componenti è composto il circuito e come sono connessi questi componenti
- **comportamento:** ciò che fa

Due circuiti con strutture diverse possono avere lo stesso comportamento; il viceversa è da evitare.

2.1 Progettazione

Dati un comportamento e una struttura, il progettista deve interconnettere i componenti tenendo conto del comportamento e del costo.

Il **costo di progetto**/sviluppo si paga una sola volta, ma è significativo ⇒ deve essere recuperato sul **costo di produzione** dei componenti (vivo), che deve essere relativamente basso. Il prodotto deve richiedere un **costo di manutenzione** ragionevole ⇒ meno guasti possibile e costi di riparazione bassi.

Flusso di progetto specifiche → sintesi → verifica → realizzazione

Il progettista di un circuito realizza prima un **modello**, che descrive le porte logiche e le loro interconnessioni che soddisfano le specifiche, quindi verifica tramite simulazione che si comporti come previsto prima di metterlo in produzione.

Il costo di progettazione a volte viene fatto in maniera top-down, partendo dalle specifiche e realizzando un progetto suddividendo il problema in sottoproblemi più semplici, oppure partendo da un modello simile già esistente e modificando alcune parti.

2.1.1 Livelli di progettazione

Il progetto può avvenire a diversi livelli. Un approccio top-down prevede la discesa nei seguenti livelli:

- **sistema:** opera su blocchi di dati codificati in un certo modo (es. output allo schermo), e utilizza dei componenti già noti;
- **registri:** non opera su 0 e 1, ma sulle variabili (ad es. si progetta il circuito per il nuovo standard USB partendo da blocchi logici elementari e assemblandoli) (si veda la sezione 2.3);
- **porte logiche:** gli elementi assemblati dal progettista sono le porte logiche, in grado di produrre dei segnali di uscita su ciascuna porta logica che dipendono dai segnali in ingresso e che sono determinati da funzioni booleane implementate in hardware (si veda la sezione 2.2);
- **transistor:** livello molto basso, ma sempre su grandezze digitali;
- **elettrico:** il circuito è descritto dal progettista a livello elettrico, cioè in termini di connessioni di componenti quali condensatori, induttori, ecc. ⇒ si riduce allo studio di un sistema di equazioni differenziali in cui compaiono correnti e tensioni.

Più si scende di livello, più il dettaglio di **tempo** è importante, e più piccoli e semplici devono essere i circuiti per poter essere studiati.

2.2 Livello di porte logiche

Il progetto di un sistema sequenziale sincrono si compone delle seguenti fasi:[1]

1. si costruiscono il diagramma degli stati e la tavola degli stati;
2. minimizzazione degli stati: un algoritmo cerca di minimizzare il diagramma degli stati, ovvero di ricondurlo a un diagramma equivalente[2] riunendo più stati in uno;
3. assegnazione degli stati: data una serie di stati con nomi simbolici, si assegna a ciascuno stato una codifica binaria e si calcola il numero di flip-flop necessari per memorizzarli:

$$\sup\{\log_2 (\text{numero degli stati})\}$$

4. costruzione della tavola di verità della rete combinatoria: per ogni combinazione di ingressi e di stati correnti, si elencano le uscite e gli stati futuri;
5. sintesi della rete combinatoria: tramite le tavole di Karnaugh, si esprimono le funzioni f e h (ciascun bit è dato da una funzione booleana);
6. si disegna il circuito.

2.2.1 Circuiti combinatori

I **circuiti combinatori** implementano funzioni combinatorie. Una **funzione combinatoria** è una trasformazione:

$$z : B^n \to B$$

dove $B = \{0, 1\}$.

In un sistema combinatorio, i valori di uscita dipendono esclusivamente dai valori applicati ai suoi ingressi in quell'istante.

[1] Si vedano le sezioni 2.2.1 e 2.2.2.

[2] Due diagrammi di stato si dicono **equivalenti** se a parità di sequenze di ingresso, il comportamento non cambia.

Rappresentazione

Il sistema deve trasformare un certo numero di bit in ingresso in un certo numero di bit in uscita. Il comportamento di un sistema combinatorio può essere descritto da:

- **funzione booleana:** es. $f(a, b) = a \cdot b$ con $\cdot$ = porta AND;
- **tavola di verità:** la tabella indica l'output per ogni combinazione in ingresso.

Esempio Il codificatore prioritario indica l'indice del primo bit in ingresso di valore 1. Si può descrivere con funzioni booleane o più semplicemente con una tavola di verità.

Terminologia

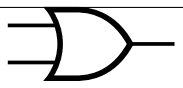
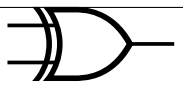
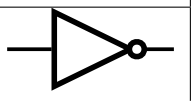
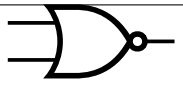
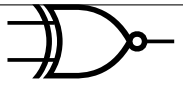
AND	OR	XOR	NOT
			
$x_1 \cdot x_2$	$x_1 + x_2$	$x_1 \oplus x_2$	$\overline{x}$
NAND	**NOR**	**XNOR**	
			
$\overline{x_1 \cdot x_2}$	$\overline{x_1 + x_2}$	$\overline{x_1 \oplus x_2}$	

Tabella 2.1: Porte logiche.[3]

fan-in numero di ingressi di una porta

fan-out numero di altre porte pilotate dall'uscita di una porta

Due porte logiche sono in connessione **wired-or** se le loro uscite vanno agli ingressi di una porta OR, che si può anche omettere.

letterale	minterm	maxterm	cubo
$\dot{x_i}$	$\dot{x_1} \cdot \dot{x_2} \cdot \cdots \cdot \dot{x_n}$	$\dot{x_1} + \dot{x_2} + \cdots + \dot{x_n}$	$\dot{x_i} \cdot \cdots \cdot \dot{x_j} \cdot \cdots \cdot \dot{x_k}$

Tabella 2.2: [4]

Circuiti combinatori ben formati

I **circuiti combinatori ben formati** (ccbf) sono caratterizzati da una delle seguenti composizioni:

- una singola linea;
- una singola porta;
- due ccbf giustapposti (cioè uno accanto all'altro);
- due ccbf tali che le uscite di uno sono gli ingressi dell'altro;
- due ccbf tali che uno degli ingressi di uno è anche uno degli ingressi dell'altro.

Sono vietati i circuiti dotati di cicli.

[3] Questa tabella contiene immagini che sono tratte da Wikimedia Commons (AND ANSI.svg, OR ANSI.svg, XOR ANSI.svg, NAND ANSI.svg, NOR ANSI.svg, XNOR ANSI.svg, NOT ANSI.svg), sono state realizzate dall'utente jjbeard e si trovano nel dominio pubblico.

[4] Il puntino su una variabile indica che la variabile può comparire affermata o negata.

Circuiti a due livelli

- somma di prodotti: le uscite di diverse porte and sono pilotate da un'unica porta or;
- prodotto di somme: le uscite di diverse porte or sono pilotate da un'unica porta and.

Minimizzazione

Data una tavola di verità, ci possono essere più **circuiti equivalenti**/funzioni che la implementano.

È possibile costruire lo schema di un circuito dalla sua tavola di verità posizionando una porta OR avente per ingressi le uscite delle porte AND che danno 1, ma il circuito risultante non è quello minimo.

Circuiti minimi Con l'**algebra booleana** si può semplificare una funzione booleana in un circuito minimo (operazione di **minimizzazione**).

Il problema di trovare un **circuito minimo** è di tipo NP-completo. Un circuito minimo è composto dal minimo numero possibile di porte logiche (al più 2 livelli). Implementa una somma di prodotti tale che il numero di prodotti è minimo (cubo), e nessun letterale può essere cancellato da un prodotto.

Dato un numero di variabili in ingresso ragionevole, il circuito minimo è ricavato dalle **tavole di Karnaugh**, che esprimono il comportamento di un circuito nella maniera più compatta possibile, cercando la copertura degli uni con il minimo numero di cubi.

Valori don't care I **valori don't care** sono quelli che non sono forniti dalla specifica nella tavola di verità, perché le relative combinazioni in ingresso non si verificano mai.

Nella **codifica BCD**, si assegnano 4 bit (4 ingressi) a ciascuna delle cifre decimali ⇒ siccome le cifre vanno da 1 a 9, avanzano dei bit che sono i valori don't care.

Ritardi

Una porta logica è composta da transistor, rappresentabili come degli interruttori che si chiudono quando la tensione applicata al segnale di controllo è alta (1 = cortocircuito) e viceversa (0 = circuito aperto). Una opportuna combinazione di transistor può far commutare la tensione di uscita V_{out} nei valori:

- 1: V_{out} viene lasciata essere uguale alla tensione di alimentazione V_{cc}, a cui è collegata attraverso la **resistenza pull-up**[5];
- 0: V_{out} risulta collegata alla massa.

Ogni transistor ha un suo tempo di commutazione ⇒ il cambiamento di un valore in ingresso si riflette con un certo **ritardo** nel valore di uscita ⇒ il valore in ingresso non viene modificato prima che sia passato il tempo $t_0 + k$ per la commutazione del valore in uscita.

Il **cammino critico** è il cammino di lunghezza massima che va da un ingresso a un'uscita, e si può trovare assegnando un livello a ciascuna porta. Come requisito per il circuito minimo si può stabilire una profondità massima. La **profondità** massima di un circuito è la lunghezza del cammino critico, ovvero il numero di porte logiche che costituiscono il cammino critico.

La profondità è legata alla velocità del circuito, cioè al numero massimo di configurazioni che è in grado di gestire nell'unità di tempo. Il ritardo massimo è un numero di unità di tempo pari alla profondità massima.

[5] Si veda la voce Resistenza pull-up su Wikipedia in italiano.

2.2.2 Sistemi sequenziali

In un **sistema sequenziale**, il valore di uscita dipende anche da ciò che è successo negli istanti precedenti.

Il numero di storie precedenti non può essere infinito, perché il numero di bit utilizzabili per memorizzare la storia precedente è finito. Una variabile interna Y detta **variabile di stato**, codificata in un certo numero di bit, memorizza un valore numerico che corrisponde a una determinata storia. L'output viene determinato in base agli ingressi correnti X e alla variabile di stato Y.

Esempio Un sistema campiona l'ingresso I con una certa frequenza di tempo. L'uscita O assume il valore 1 se durante i 3 istanti di campionamento precedenti l'ingresso I ha assunto i valori 101, e assume il valore 0 diversamente.

Rappresentazione

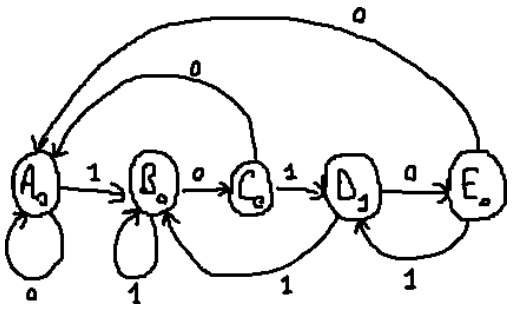


Figura 2.1: Esempio di diagramma degli stati.

Diagramma degli stati Il **diagramma degli stati** si basa su una scala dei tempi discretizzata in punti ben definiti:

- stato corrente: la variabile di stato può assumere i valori rappresentati come lettere cerchiate;
- ingresso: ogni arco rappresenta il cambio di stato, in base al valore in ingresso corrente;
- uscita: si può scegliere di rappresentare i valori di output in due modi:
  - a pedice dello stato (come in figura): questo valore è un unico bit, quindi può dipendere solo dallo stato corrente e non dagli ingressi correnti;
  - direttamente sugli archi, racchiusi in un rettangolo: facilita la rappresentazione di valori di uscita composti da più bit.

Il numero totale degli archi è uguale a: numero degli stati $\times$ numero delle combinazioni di ingresso.

Tavola degli stati La **tavola degli stati** elenca tutte le combinazioni possibili di stati correnti e valori in ingresso. Il numero delle righe si calcola: numero degli stati $\times$ $2^{\text{numero di bit in ingresso}}$ $\Rightarrow$ diventa troppo lunga in caso di alti numeri di stati e di bit in ingresso.

Funzioni di transizione e d'uscita degli stati La **funzione di transizione** e la **funzione di uscita** sono delle funzioni booleane che, dati i valori assunti dagli ingressi correnti e dalla variabile di stato corrente[6], restituiscono rispettivamente il valore futuro della variabile di stato e il valore di uscita.

[6] I valori assunti dalla variabile di stato sono rappresentati in formato binario anziché in lettere.

Implementazione

Il **flip-flop** è un elemento di memoria in grado di memorizzare un singolo bit.

Flip-flop asincroni Il **flip-flop set-reset (*SR*) asincrono** è un elemento di memoria di tipo sequenziale in cui si può forzare o mantenere un certo valore binario a seconda della scelta dei due ingressi S e R:

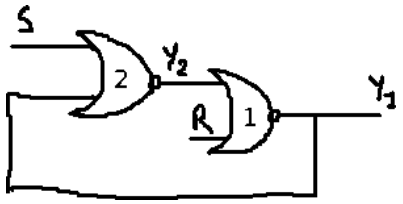


Figura 2.2: Flip-flop SR asincrono.

ingresso S	**ingresso** R	**uscita** Y_1 **e** Y_2 **= NOT** Y_1
1	0	Vengono forzati i valori 1 sull'uscita Y_1 e 0 sull'uscita Y_2.
0	1	Vengono forzati i valori 0 sull'uscita Y_1 e 1 sull'uscita Y_2.
0	0	I valori in uscita non vengono commutati ma si mantengono costanti.
1	1	È una configurazione vietata perché la transizione dallo stato $S = R = 1$ allo stato $S = R = 0$ produce un risultato logicamente non definito: l'uscita Y_1 può assumere 0 o 1 a seconda dei ritardi di commutazione Δ_1 e Δ_2 delle due porte logiche, i quali dipendono da fattori non prevedibili (per es. temperatura, umidità...): • l'uscita Y_1 rimane al valore 0 se $\Delta_1 < \Delta_2$, cioè la porta 2 commuta prima che la porta 1 abbia finito di commutare; • l'uscita Y_1 commuta al valore 1 se $\Delta_1 > \Delta_2$, cioè la porta 1 riesce a finire di commutare prima che la porta 2 ha commutato.

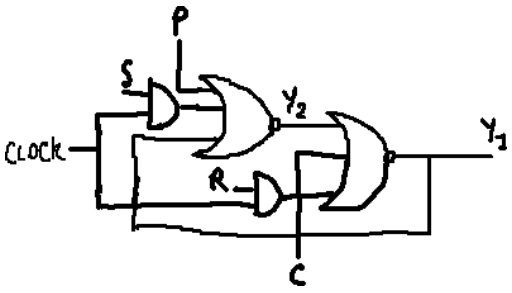


Figura 2.3: Flip-flop SR sincrono.

Flip-flop sincroni Il **clock** è un segnale di temporizzazione a onda quadra. In un **flip-flop *SR* sincrono**, il segnale di clock si duplica in due segnali posti agli ingressi di due porte AND in coppia con entrambi gli ingressi S e R:

- clock = 1: il flip-flop è sensibile alle variazioni degli ingressi S e R;

- clock = 0: il flip-flop mantiene il suo valore senza vedere gli ingressi S e R, lasciando così il tempo al nuovo valore di propagarsi.

Due ulteriori segnali P (preset) e C (clear) servono per forzare in modo asincrono il flip-flop rispettivamente a 1 o a 0.

Nel **flip-flop di tipo** D, S e R sono uniti, attraverso una porta NOT, in un unico ingresso D che permette di forzare una coppia di valori $S \neq R$, che rimarrà costante fino al successivo colpo di clock. I flip-flop di tipo D sono i più utilizzati perché sono semplici e poco costosi, e non presentano configurazioni vietate.

Un **flip-flop di tipo** T è un flip-flop di tipo D senza ingressi: il segnale di clock fa invertire il valore memorizzato (l'uscita ritorna all'ingresso tramite una porta NOT).

Un sistema sequenziale si dice **sincrono** se tutti i flip-flop sono collegati allo stesso segnale di clock.

Modello di Huffman Il **modello di Huffman** è un circuito costituito da una **rete combinatoria** e da una serie di **flip-flop** sincroni.

Tramite il modello di Huffman si può implementare un sistema sequenziale. La variabile di stato si può memorizzare in una serie di flip-flop. Gli altri blocchi combinatori ricevono in ingresso la variabile di stato e gli altri valori in ingresso.

Casi particolari

- **modello di Mealy:** tutte le uscite z_i della rete combinatoria dipendono sia dagli ingressi x_i sia dalle variabili di stato y_i;
- **modello di Moore:** almeno un'uscita z_i dipende solo dalle variabili di stato $y_i \Rightarrow$ tali uscite rimarranno costanti fino al successivo colpo di clock.

2.3 Livello dei registri

Non si lavora a livello di singoli bit ma a livello di parole (= vettori di bit). Il **parallelismo** di una parola è il suo numero di bit.

2.3.1 Componenti generici

Porta logica operante su parole

La **porta logica operante su parole** è uno scatolotto contenente una batteria di porte logiche elementari, avente ingresso e uscita con eguale parallelismo di m bit e con eguale numero di parole.

Multiplexer

Il **multiplexer**, pilotato dall'ingresso p, detto **segnale di controllo**, seleziona una delle k parole in ingresso portandola in uscita.

Connettendo più multiplexer semplici **in cascata** si può avere un maggior numero di parole in ingresso.

L'uso di un multiplexer è un modo veloce (ma non efficiente) per implementare il comportamento di una generica funzione booleana data: basta porre agli ingressi con parallelismo di $m = 1$ bit tutti i possibili 2^n valori restituiti dalla funzione, e ai segnali di controllo gli n ingressi valutati dalla funzione.

Decodificatore

Il **decodificatore** è un modulo avente 2^n linee in uscita e n linee in ingresso. Il valore in ingresso attiva una sola linea di uscita, cioè quella avente per indice lo stesso valore in ingresso.

Accanto ai normali valori in ingresso, vi è il **valore di enable** e che assume 1 o 0:

- $e = 1$: alla linea di uscita è consentito essere attivata dalla linea in ingresso corrente;
- $e = 0$: nessuna uscita può essere attivata, e in uscita vi è un valore predefinito indipendente dalla linea in ingresso.

Anche i decodificatori si possono connettere in cascata: si assegna la discriminazione dei bit più significativi ai decodificatori dei primi livelli e quella dei bit meno significativi ai decodificatori dei livelli sottostanti, e si forniscono come valori di enable le uscite dei decodificatori a livello superiore. Con la connessione in cascata bisogna però tenere in conto dei ritardi di commutazione dei decodificatori (durante il tempo di commutazione Δt i segnali in uscita sono casuali nel tempo).

Codificatore

Il **codificatore** ha un comportamento opposto a quello del decodificatore: riceve una sola delle 2^k linee in ingresso e restituisce fino a k linee in uscita, la cui parola corrisponde all'indice della linea in ingresso corrente.

Sono però vietate le combinazioni in ingresso con più di un bit 1 $\Rightarrow$ non è molto utilizzato.

Codificatore prioritario

Nel **codificatore prioritario**, a ogni linea in ingresso è associata oltre all'indice una **priorità**. L'uscita è l'indice della linea in ingresso con priorità massima tra quelle attive.

In questo modo viene ottimizzato l'impiego dei bit in ingresso.

Siccome le priorità partono da 0, bisogna distinguere il caso in cui la linea a priorità 0 è quella attiva e il caso in cui nessuna linea è attiva $\Rightarrow$ si aggiunge il bit **Input Active** (IA) in uscita:

- $IA = 1$: almeno una linea in ingresso è attiva;
- $IA = 0$: nessuna linea in ingresso è attiva.

2.3.2 Moduli aritmetici

Sommatore combinatorio

Il **sommatore combinatorio** è un modulo aritmetico che riceve in ingresso 2 parole di m bit e ne restituisce in uscita la somma in $m + 1$ bit $\Rightarrow$ non è molto utilizzato per il suo costo di progetto: anche se richiede il minimo tempo, è da progettare per ogni numero m.

Sommatore combinatorio modulare con ripple carry adder

Il **sommatore combinatorio modulare** si compone di una successione (= **ripple carry adder**) di **full-adder**[7], che sono dei sommatori combinatori in grado di gestire il riporto. Ogni full-adder riceve solo 2 bit in ingresso ($m = 1$) e il bit di riporto/**carry** c proveniente dal full-adder precedente $\Rightarrow$ maggiore flessibilità $\Rightarrow$ minore costo di progetto.

Costruendo la tavola di verità, si sintetizzano le seguenti funzioni:

$$\begin{cases} z_i = x_i \oplus y_i \oplus c_i \\ c_{i+1} = x_i y_i + x_i c_i + y_i c_i \end{cases}$$

[7] Si veda la voce Full-adder su Wikipedia in italiano.

I full-adder sono implementati con circuiti combinatori a k livelli $\Rightarrow$ il ritardo temporale Δ di ciascun livello si riflette in un ritardo temporale $\delta = k\Delta$ di ciascun full-adder.

Nel ripple carry adder, il riporto viene propagato nella catena $\Rightarrow$ i ritardi crescono al crescere del parallelismo.

Sommatore combinatorio modulare con carry-lookahead

Dalla funzione del sommatore combinatorio modulare relativa al riporto c_i:

$$\begin{cases} c_i = x_i y_i + (x_i + y_i)c_{i-1} = g_i + p_i c_{i-1} \\ c_{i-1} = x_i y_i + (x_i + y_i)c_{i-2} = g_i + p_i c_{i-2} \end{cases} \Rightarrow c_i = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-2}$$

con: $\begin{cases} g_i = x_i \cdot y_i \\ p_i = x_i + y_i \end{cases}$, si può ricavare la seguente formula in termini solo di x e y:

$$c_k = g_k + (p_k g_{k-1} + p_k p_{k-1} g_{k-2} + \cdots + p_k p_{k-1} \cdots p_2 p_1 g_0) + p_k p_{k-1} \cdots p_2 p_1 p_0 c_{in}$$

in modo che il full-adder non debba aspettare il carry del precedente. Il **carry-lookahead** invia a ciascun full-adder il corrispondente c.

Sommatore seriale

Il **sommatore seriale** contiene un singolo full-adder che tratta a ogni colpo di clock una coppia di bit alla volta; il carry corrente viene salvato in un flip-flop interno.

Vantaggi minimo hardware e minimo costo

Svantaggio i tempi di esecuzione sono più lunghi a causa dei colpi di clock

ALU[8]

La **Arithmetic Logic Unit** (ALU) è una componente combinatoria che implementa tutte le principali funzioni aritmetiche (su numeri decimali) e logiche (bit a bit). In ingresso non ha il segnale di clock, ma un segnale di controllo che specifica il tipo di operazione da effettuare. Il ritardo è dovuto solo alle porte logiche interne. Un ulteriore ingresso di carry C_{in} (con corrispettiva uscita C_{out}) permette di collegare ALU in cascata per operazioni più complesse.

2.3.3 Comparatore a 4 bit

Il **comparatore a 4 bit** permette di confrontare due numeri interi senza segno su 4 bit. Un valore di enable stabilisce se le linee sono attivabili.

Connettendo più comparatori in cascata è possibile confrontare interi composti da un numero maggiore di bit: si va dalla quaterna di bit più significativa a quella meno significativa. Il segnale di enable in ingresso a ogni comparatore è pilotato dal comparatore precedente nella catena, in modo che venga attivato solo se quest'ultimo ha verificato la condizione di uguaglianza.

2.3.4 Registri

Registro a n bit

Il **registro a n bit** serve per memorizzare un ingresso su n bit ogniqualvolta viene attivato il segnale di load; il valore memorizzato viene riportato nell'uscita z e viene mantenuto costante fino al successivo segnale di load $\Rightarrow$ è possibile stabilizzare un segnale molto variabile. Se presente, il segnale di clear permette di resettare a 0 tutti i bit memorizzati.

Internamente, il registro a n bit contiene una serie di n flip-flop di tipo D, ciascuno dei quali memorizza uno degli n bit della parola in ingresso.

[8] Per approfondire, si veda la voce Arithmetic Logic Unit su Wikipedia in italiano.

Registro a scalamento

Il **registro a scalamento** (o shift-register) memorizza le parole non parallelamente ma un bit alla volta, perché l'ingresso x e l'uscita z hanno parallelismo 1.

Il segnale di shift enable pilota il clock di ogni flip-flop: al segnale di shift enable, il contenuto corrente viene scalato di una posizione: il flip-flop i-esimo passa il bit memorizzato al flip-flop $i+1$-esimo, quindi riceve il nuovo bit dal flip-flop $i-1$-esimo, facendo entrare all'estrema sinistra il nuovo bit in ingresso e facendo uscire all'estrema destra l'ultimo bit in eccesso.

I registri a scalamento universali (più costosi) permettono lo scalamento in entrambi i sensi, e ne esistono diverse implementazioni:

- ci possono essere due segnali di shift enable, uno a destra e uno a sinistra;
- ci può essere un unico segnale di shift enable insieme a uno di left shift select, che specifica se effettuare lo scalamento a destra o a sinistra.

Vi può anche essere un segnale di load che permette di caricare parallelamente n bit in ingresso (come nel registro a n bit).

Applicazioni

- convertire dati seriali in ingresso (un bit per volta) in dati paralleli in uscita (byte), e viceversa;
- moltiplicare/dividere numeri per potenze di 2 (basta scalare di un bit);
- memorizzare dati seriali in una coda FIFO (il primo valore scala fino in fondo ed esce per primo).

2.3.5 Contatori

In un **contatore**, il valore memorizzato all'interno viene incrementato/decrementato a ogni segnale di count enable. Un contatore può avere, a seconda dell'occasione, anche degli altri segnali specifici:

- up/down: indica se il valore va incrementato o decrementato;
- step: specifica di quanto deve essere incrementato il valore;
- terminal count: fornisce 1 quando il contatore raggiunge il suo valore massimo (11...).

Viene anche detto **contatore modulo-2^k** (dove k è il numero di bit) perché al 2^k-esimo impulso (11...) ritorna al valore iniziale 0.

Contatore asincrono

Il **contatore asincrono** è costituito da una serie di flip-flop di tipo T: il segnale di count enable di ogni flip-flop è pilotato dal flip-flop precedente, così un flip-flop quando commuta da 0 a 1 iniziando il fronte di salita dell'uscita fa commutare anche il flip-flop successivo $\Rightarrow$ il contatore è asincrono perché i flip-flop non sono pilotati dallo stesso segnale di clock. La successione di bit ottenuta è discendente.

Vantaggi hardware economico e facilità di progettazione

Svantaggio siccome ogni flip-flop ha un ritardo Δ, il ritardo complessivo è proporzionale al parallelismo del contatore (numero di flip-flop)

Contatore sincrono

Il **contatore sincrono** prevede un unico segnale di count enable. La logica combinatoria è costituita da un sommatore/sottrattore che carica parallelamente tutti i bit, li elabora e restituisce direttamente tutti i nuovi bit da scrivere nei flip-flop.

Svantaggi hardware più complesso e costoso

Vantaggio il ritardo è indipendente dal numero di flip-flop, e dipende solo dai ritardi della logica combinatoria e del singolo flip-flop

2.3.6 Interfacce al bus

Un insieme di moduli è detto **in connessione a bus** se i segnali vengono trasferiti da un modulo all'altro attraverso una struttura di interconnessione/comunicazione detta **bus**[9], da cui i moduli in lettura caricano il valore corrente a ogni segnale di load. Per i moduli in scrittura invece vale la seguente regola: in ogni istante il bus può essere pilotato da uno solo dei moduli collegati in scrittura ⇒ ogni modulo deve essere collegato al bus attraverso un'interfaccia.

L'**interfaccia** è un dispositivo, posto tra il modulo e il bus, che prima di passare al bus il nuovo valore da scrivere verifica che nessun altro modulo stia scrivendo sul bus.

Il **buffer tri-state** è un'interfaccia che agisce a seconda dei valori di enable, pilotati da un decoder:

- $e = 1$: il modulo è collegato al bus, e l'interfaccia lascia passare l'uscita del modulo verso il bus;
- $e = 0$: il modulo è in **alta impedenza** (l'uscita è Z), cioè è elettricamente scollegato dal bus e la scrittura è inibita.

Il **transceiver** è un'interfaccia generica che permette la bidirezionalità del segnale (sia in lettura sia in scrittura).

2.3.7 Memorie[10]

Le **memorie** sono dei moduli/dispositivi in grado di memorizzare m[11] parole di parallelismo n e di mantenerle nel tempo. A differenza dei registri, le memorie possono leggere o scrivere da/su una sola parola per volta. Il segnale di indirizzo su $\log_2 m$ bit fornisce l'indice della parola; i segnali di read e di write specificano se la parola è da leggere o scrivere. Manca il segnale di clock ⇒ le specifiche devono indicare i parametri temporali della memoria, come il tempo minimo di attesa tra un'operazione e l'altra e il tempo di lettura della parola in ingresso, in base ai ritardi interni della memoria.

Un segnale di enable serve per accecare la memoria ai segnali di ingresso e per eventualmente mettere l'uscita a un bus in alta impedenza.

Memoria RAM

Le **memorie RAM** sono delle memorie cosiddette **volatili** perché il contenuto viene cancellato quando si scollega l'alimentazione. Le memorie RAM si possono inserire internamente in un circuito oppure su piastre.

Operazione di scrittura Un decoder riceve in ingresso il segnale di indirizzo e seleziona una parola attivando uno degli m segnali di load, il quale, per mezzo di una porta AND che lo affianca al segnale di write, attiva la scrittura nella parola di indice corrispondente.

[9] Si veda il capitolo 14.
[10] Per approfondire, si veda il capitolo 7.
[11] Solitamente il numero di parole m è una potenza di 2.

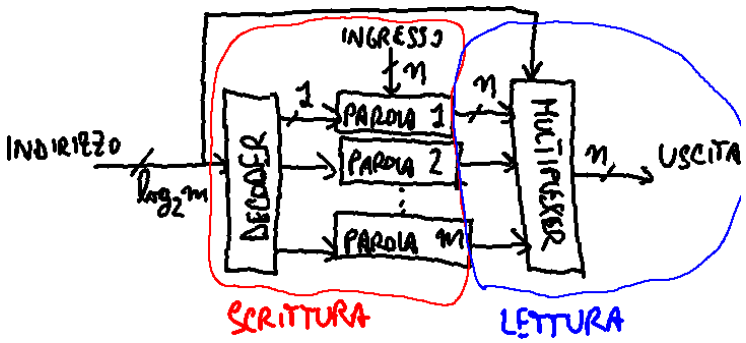


Figura 2.4: Struttura interna di una memoria RAM.

Operazione di lettura Il multiplexer, ricevendo il segnale di indirizzo (sempre accoppiato in AND con il segnale di read), seleziona una parola e ne riporta l'uscita.

Memoria ROM

A differenza delle RAM, le **memorie ROM** sono in sola lettura. Il contenuto viene determinato all'atto della fabbricazione e non può più essere modificato successivamente. La memoria ROM è un modulo combinatorio perché l'uscita dipende solo dall'ingresso dell'indice di parola corrente.

Banchi di memoria

Se la lunghezza delle parole è maggiore del parallelismo di un modulo, si possono affiancare più moduli su cui suddividere le parole, a cui fornire le opportune porzioni di bit e parallelamente il segnale di indirizzo comune.

Se invece la suddivisione è fatta sullo spazio di indirizzamento, i dati vengono forniti parallelamente a tutti i moduli tramite un bus. Come segnali di enable si devono usare direttamente i bit più significativi degli indirizzi di memoria, che possono essere elaborati da un decoder o più semplicemente, nel caso di 2 moduli, inviati in via diretta a un modulo e attraverso un inverter all'altro.

2.3.8 Moduli programmabili

PLA

Le **Programmable Logic Array** (PLA) aiutano a costruire un circuito di tipo combinatorio in modo più semplice ed economico: il costruttore può infatti programmare una PLA, cioè può collegare le singole porte logiche non direttamente a mano ma attraverso dei segnali elettrici, in modo da personalizzarla in base alla funzione booleana che vuole implementare. La programmazione di una PLA è irreversibile perché si può fare una volta sola.

Un esempio di applicazione è la codifica di un numero in una visualizzazione a trattini sul display del supermercato.

FPGA

Il **Field Programmable Gate Array** (FPGA) aggiunge alla PLA una componente sequenziale e la possibilità di riprogrammare più di una volta il circuito. Rappresenta una soluzione intermedia tra il costo di progettazione e l'efficienza della soluzione HW e la flessibilità della soluzione SW.

L'FPGA viene utilizzata spesso per i prototipi: il programmatore può sperimentare il suo codice sull'FPGA prototipo mentre aspetta che venga sviluppato il più ottimizzato circuito integrato ASIC (= soluzione HW).

2.4 Cenni sul consumo

Il consumo nel tempo di corrente da parte di un circuito è definito in termini di:

- energia totale (potenza effettiva)
- dissipazione del calore

Parametri

- **consumo statico:** (trascurabile) è fisso perché si deve solo al fatto di essere alimentato, anche se i transistor non commutano;
- **consumo dinamico:** dovuto ai transistor che commutano un numero di volte proporzionale alla frequenza di clock $\Rightarrow$ si deve ridurre il più possibile, facendo attenzione alle prestazioni.

Capitolo 3

I processori

Il **processore** è un dispositivo capace di:

- elaborare istruzioni, cioè leggere delle istruzioni dalla memoria (fase di fetch) ed eseguirle (fase di esecuzione);[1]
- interagire con il mondo esterno attraverso le porte di periferica, reagendo il più presto possibile alle segnalazioni (= richieste di interrupt) provenienti dall'esterno.[2]

L'importanza di ciascuna di queste capacità dipende dalla destinazione d'uso del processore.

3.1 Elaborazione delle istruzioni[3]

3.1.1 Fase di fetch

L'**instruction set** è l'insieme delle operazioni che il processore è in grado di eseguire. A ogni operazione corrisponde una certa istruzione in memoria (es. `ADD`). Durante la **fase di fetch** il processore legge una di queste istruzioni dalla memoria tramite un bus esterno:

1. il **Program Counter** (PC) contiene l'indirizzo di memoria della nuova istruzione da leggere;
2. l'**Address Register** (AR) passa il nuovo indirizzo dal PC alla memoria;
3. la memoria restituisce la porzione di codice corrispondente all'indirizzo specificato dal processore;
4. il **Data Register** (DR) memorizza[4] nell'**Instruction Register** (IR) la porzione di codice proveniente dalla memoria, pronta ad essere decodificata ed eseguita dal processore;
5. il PC viene aggiornato all'indirizzo dell'istruzione successiva a quella letta.

3.1.2 Fase di esecuzione

L'**unità di controllo**[5], tramite il bus interno, dopo aver ricevuto l'istruzione dall'IR, ha il compito di pilotare i vari moduli, cioè fornire segnali di controllo ad essi, per le operazioni ad es. di caricamento degli operandi, esecuzione delle istruzioni, acquisizione dei risultati, ecc. La

[1] Si veda la sezione 3.1.
[2] Si veda il capitolo 11.
[3] Per approfondire, si veda la voce Ciclo di fetch-execute su Wikipedia in italiano.
[4] Il DR viene usato in generale per il trasferimento di dati tra la memoria e il processore: un ulteriore segnale di controllo indica se l'operazione che deve effettuare la memoria è la lettura o la scrittura.
[5] Si veda il capitolo 5.

Arithmetic-Logic Unit (ALU)[6] è il modulo che esegue una serie di operazioni aritmetiche/-logiche su operandi. I registri **Accumulator** (AC) forniscono gli operandi alla ALU e ricevono i risultati delle operazioni.

All'interno di ogni istruzione si possono identificare delle **microistruzioni** elementari (es. accesso ai registri, fetch, ecc.), ciascuna delle quali richiede un colpo di clock.

Il tempo di esecuzione di un'istruzione, misurato in colpi di clock, dipende da:

- tipo di istruzione: quale istruzione eseguire (es. `MOV`, `ADD`)?
- tipo di operandi: l'operando si trova nei registri (accesso più veloce) o in memoria (ritardo del fetch)?
- modo di indirizzamento: l'indirizzo passato è diretto (es. `VAR`) o richiede una somma (es. `vett[DI]`)?

Gli **operandi** di una generica istruzione possono essere contenuti nei registri (`Rx`) o nella memoria (`M[xx]`), a seconda di ciò che è richiesto dall'operazione letta dalla memoria. I **registri** sono dei moduli di memoria costituiti da flip-flop che risiedono all'interno del processore. Anche se il numero di registri è finito, il tempo di accesso a un registro è molto minore del tempo di accesso a una cella di memoria, a causa dei colpi di clock necessari durante l'operazione di fetch. Il compilatore di C decide automaticamente se utilizzare i registri o la memoria, privilegiando quando possibile i primi: nel caso in cui l'operando si trova in memoria, durante la fase di esecuzione dell'operazione è necessario effettuare un ulteriore fetch per l'operando.

Per ridurre i tempi di esecuzione, mentre l'**execution unit** esegue un'operazione la **bus interface unit** si occupa del fetch dell'istruzione successiva.

[6] Si veda la sezione 2.3.2.

Capitolo 4

Architettura 8086

Il processore Intel 8086 è caratterizzato da:

- **spazio di indirizzamento:** la quantità di memoria che è in grado di indirizzare (2^{20} indirizzi da 1 byte = 1 MB);
- **parallelismo:** le lunghezze di parole con cui i moduli sono in grado di operare (16 bit).

La memoria è suddivisa in due porzioni, definite per dimensioni e posizione: una contiene i dati del programma, l'altra contiene il **codice macchina** del programma.

4.1 Registri

L'**instruction set architecture** (ISA) è l'insieme delle informazioni pubblicate dal produttore del processore.

4.1.1 Classificazione

- **special register:** (es. PC, IR) non possono essere acceduti dal programmatore in modo diretto, anche se ad esempio l'istruzione `JMP` accede indirettamente al PC;
- **user register:** (es. AX, DI) possono essere usati dal programmatore in modo esplicito nelle istruzioni.

- **registri di dato:** (AX, BX, CX, DX) sono 4 registri da 16 bit, ciascuno dei quali è costituito da una coppia di registri da 8 bit (H e L);
- **registri indice:** (es. DI, BX) oltre a poter contenere i dati, possono essere usati anche per passare un indirizzo (es. `vett[DI]`);
- **registri contatore** (es. CX);
- **registri di segmento** (Code Segment [CS], Data Segment [DS], Stack Segment [SS], Extra Data Segment [ES]);[1]
- **registri puntatore** (es. Instruction Pointer [IP], Stack Pointer [SP], Base Pointer [BP]).[2]

Ci possono anche essere dei registri di servizio, che il progettista ha destinato solo all'uso interno.

[1] Si veda la sezione 4.2.1.
[2] Si veda la sezione 4.2.1.

4.1.2 Status register

Lo **status register** è un modulo da 16 bit, 9 dei quali contengono i flag di condizione e di controllo, che vengono letti continuamente tra un'istruzione e l'altra:

- **flag di controllo:** definiscono alcune modalità di comportamento del processore (per es. l'Interrupt Flag [IF] acceca il processore ai segnali di interrupt);
- **flag di condizione:** il processore deve poter valutare le condizioni delle istruzioni `if` e saltare alle `sub` appropriate ⇒ a differenza dell'istruzione `JMP`, che è un'istruzione di salto incondizionato che ne scrive l'indirizzo nel PC senza valutare alcuna condizione di istruzioni `if`, le istruzioni di salto condizionato `JN<nome_flag>` decidono se scrivere o no l'indirizzo nel PC a seconda di certi flag di condizione `<nome_flag>F`, che accompagnano l'uscita della ALU al termine di un'operazione.

4.2 Accesso alla memoria

Il BHE è un segnale di enable che, nel caso di indirizzi di memoria pari, specifica se la memoria deve restituire una coppia di byte, cioè la word intera (BHE = 0), oppure un singolo byte (BHE = 1):

A_0 (bit meno significativo)	BHE	Byte restituiti
0 (pari)	0	2
	1	1
1 (dispari)	0	1
	1	don't care[3]

Un numero su 2 byte viene memorizzato in memoria con la rappresentazione little endian, ovvero il byte meno significativo precede quello più significativo.

Alcune parti della memoria predefinite sono **riservate** per esempio al bootstrap (Reset Bootstrap) e alla gestione degli interrupt (Interrupt Vector Table).

Ogni istruzione in codice macchina richiede da 1 a 6 byte, e comprende il **codice operativo** che identifica il tipo di operazione, e per ogni operando alcuni bit per identificarne il tipo (es. registro dati) e altri per il suo indirizzo.

4.2.1 Indirizzi di memoria

Il processore raggiunge ogni cella di memoria tramite un **indirizzo fisico** su 20 bit, che è la somma binaria di due sotto-indirizzi: il segment address e l'effective address. Questo meccanismo impedisce che un programma esca dai segmenti di memoria ad esso riservati, anche se le operazioni sugli indirizzi richiedono un certo tempo.

Segment address

Il **segment address** è l'indirizzo di partenza del segmento che contiene la cella di memoria.

La memoria dell'8086, di dimensioni totali pari a 2^{20} byte, è suddivisa in **paragrafi** da 16 byte ⇒ i 4 bit meno significativi dell'indirizzo di partenza di ogni paragrafo risultano sempre uguali a 0 ⇒ bastano 16 bit per identificare un paragrafo. Siccome la posizione iniziale di un paragrafo coincide anche con la posizione iniziale di un segmento da 64 KB = 2^{16} byte, il segment address ha una lunghezza pari a 16 bit anziché 20 bit.

I primi 16 bit dei segment address, tipicamente relativi ai segmenti di memoria codice (CS), dati (DS) e stack (SS), sono memorizzati nei **registri di segmento**. L'ES è usato per i segmenti di memoria maggiori di 64 KB.

[3] Si veda la sezione 2.2.1.

Effective address

L'**effective address** è l'offset della cella di memoria rispetto all'indirizzo di partenza del segmento.

L'effective address richiede 16 bit perché ogni segmento ha dimensione pari a 64 KB.

I **registri puntatore** Instruction Pointer (IP), Stack Pointer (SP) e Base Pointer (BP) memorizzano gli effective address correnti, da sommare agli opportuni segment address. Anche i registri indice SI e DI possono essere usati come registri puntatore.

4.2.2 Segmento di stack

Nella memoria, a ogni programma è assegnato, oltre ai segmenti dati e codice, un terzo segmento di memoria detto **stack**, che implementa una coda LIFO. Nello stack, il processore non opera direttamente sugli indirizzi di memoria, ma compie solo delle operazioni di push (inserimento) e pop (estrazione).

Il registro **Stack Segment** (SS) contiene il segment address, cioè l'indirizzo della testa dello stack. Lo stack si riempie a partire dalla coda verso la testa. Lo **Stack Pointer** (SP) è uno special register, invisibile al programmatore, che memorizza l'effective address, cioè l'indirizzo della prima cella correntemente libera dello stack; viene decrementato e incrementato di 2 byte rispettivamente dalle operazioni di push e pop. Il processore non si accorge se lo stack è vuoto $\Rightarrow$ esiste il rischio di eccesso di operazioni di pop.

Capitolo 5

Unità di controllo[1]

Il processore è composto da due parti:

- **unità di elaborazione** (o data path): contiene tutti i componenti attraverso cui passano i dati;
- **unità di controllo** (UC): si occupa di fornire i segnali di controllo per pilotare i componenti contenuti nell'unità di elaborazione, a seconda di volta in volta della istruzione corrente contenuta nell'IR e degli stimoli provenienti dall'esterno (es. campionamento MFC[2]).

5.1 Microistruzioni

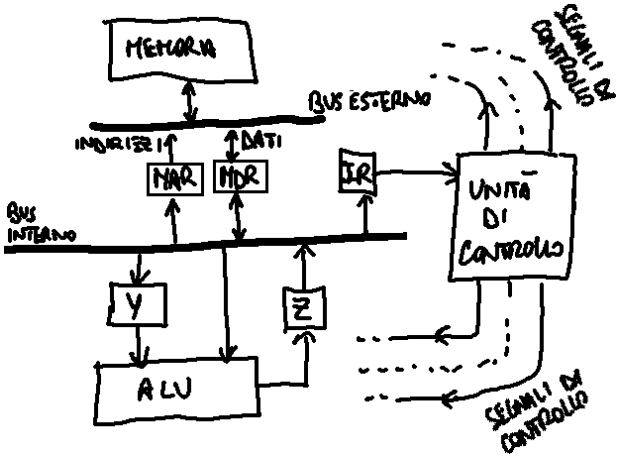


Figura 5.1: Esempio di CPU.

L'unità di controllo deve conoscere le operazioni che deve effettuare per tutte le microistruzioni dell'instruction set. L'unità di controllo può eseguire le seguenti operazioni elementari:

1. trasferimento di un dato da un registro ad un altro;
2. prelievo di un dato o di una istruzione dalla memoria e caricamento in un registro;
3. scrittura in memoria di un dato contenuto in un registro;
4. passaggio alla ALU degli operandi di un'operazione aritmetica o logica e memorizzazione del risultato in un registro.

[1] Si veda la voce Unità di controllo (informatica) su Wikipedia in italiano.
[2] Si veda la sezione 5.1.2.

5.1.1 Trasferimento di dati tra registri

I dati possono passare attraverso il **bus interno** da un registro all'altro. La unità di controllo può pilotare l'ingresso e l'uscita di ogni registro attraverso due segnali di controllo: uno pilota l'ingresso per il caricamento di dati dal bus, l'altro viceversa.

La unità di controllo deve anche preoccuparsi che l'ingresso del registro di destinazione venga attivato dopo che il registro di partenza ha finito di scrivere il dato sul bus.

5.1.2 Lettura e scrittura di dati in memoria

Dal punto di vista della comunicazione tra memoria e processore, il bus esterno è in realtà diviso in tre bus: bus dati, bus indirizzi e bus di controllo.

Il registro MAR può solo leggere dal bus interno e scrivere sul bus esterno tramite due segnali di controllo, e ha il compito di passare alla memoria l'indirizzo della cella da leggere. L'unità di controllo campiona il segnale di controllo MFC: quando la memoria ha finito di leggere il dato e il segnale è stabile, essa attiva l'MFC.

Il registro MDR è connesso con il bus interno e con il bus esterno con segnali bidirezionali. Un multiplexer pilotato dal segnale di controllo SEL decide se gli ingressi dell'MDR devono essere pilotati dal bus interno oppure da quello esterno. Due buffer tri-state decidono invece a quale bus inviare le uscite.

5.1.3 Passaggio di operandi e ritorno del risultato di un'operazione

Y e Z sono due registri **Accumulator** (AC) che forniscono gli operandi alla ALU e ricevono i risultati delle operazioni.

Nel caso di un'operazione a due operandi, siccome non si può leggere dalla memoria più un operando per volta, il primo operando va caricato temporaneamente nel registro Y, aspettando che il secondo operando venga letto in modo che i due operandi vengano passati parallelamente alla ALU.

Il risultato dell'operazione viene memorizzato nel registro Z pronto ad essere passato al bus interno.

5.1.4 Esempi

`ADD [R3], R1`

Durante la fase di fetch, il processore esegue in parallelo l'accesso in memoria e, mentre aspetta l'MFC, l'aggiornamento del PC (tramite una rapida PC_{in}), sfruttando il fatto che il PC è stato caricato sul bus interno dalla precedente PC_{out}. Inoltre, nell'attesa della lettura del secondo operando il primo viene già predisposto all'elaborazione venendo caricato nel modulo Y.

`JMP lab`

Il codice macchina dell'istruzione contiene l'offset dell'istruzione a cui saltare relativo all'indirizzo dell'istruzione di salto, pronto da sommare al valore corrente del PC.

`JE lab`

L'unità di controllo verifica il valore del generico flag N trasferito dall'unità di elaborazione all'unità di controllo ($C"_{in}$).

5.2 Tipi di unità di controllo

Gli obiettivi di progettazione di un'unità di controllo sono:

- minimizzare la quantità di hardware;

- massimizzare la velocità di esecuzione;
- ridurre il tempo di progetto (flessibilità).

A seconda delle esigenze si possono scegliere due tipi di unità di controllo: **cablata** e **microprogrammata**.

Generalmente i sistemi CISC sono dotati di un'unità di controllo microprogrammata, e i RISC di una cablata. L'unità di controllo dei microprocessori 8086 è microprogrammata.

5.2.1 Unità di controllo cablata

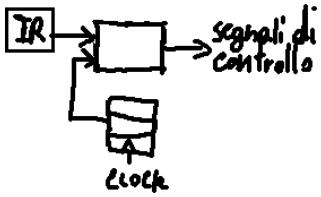


Figura 5.2: Struttura dell'unità di controllo cablata.

L'**unità di controllo cablata** (o hardwired) è un circuito sequenziale che riceve i segnali di ingresso, passa attraverso una variazione degli stati e restituisce i dati in uscita: dei flip-flop memorizzano lo stato corrente, e la rete combinatoria ha in ingresso l'IR e altri segnali e in uscita i segnali di controllo con cui pilotare l'unità di elaborazione.

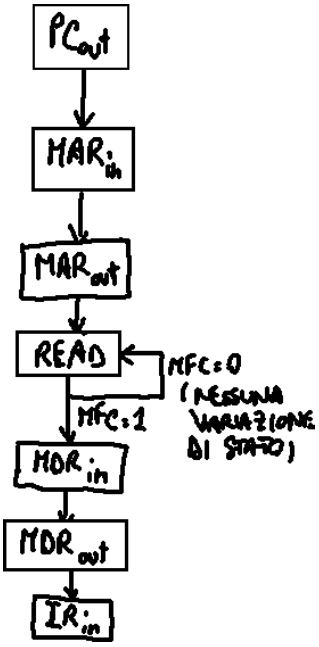


Figura 5.3: Diagramma degli stati della fase di fetch.

Le operazioni sono temporizzate da colpi di clock:

1. dopo aver ricevuto il segnale di `END` dell'istruzione precedente, inizia il fetch tramite il registro MAR;
2. continua a campionare il segnale MFC in attesa che la memoria sia pronta, finché l'MFC non è pari a 1;

3. carica l'istruzione nell'IR tramite il registro MDR;
4. esegue l'istruzione contenuta nell'IR pilotando i vari segnali di controllo.

Svantaggi

- complessità di progetto: i tempi e i costi di progetto dell'unità di controllo possono diventare troppo alti se un processore è troppo complesso e richiede troppi segnali di controllo da gestire;
- mancanza di flessibilità: dimenticando anche solo un segnale di controllo l'operazione non va a buon fine. Inoltre, se si aggiungono istruzioni bisogna rifare da capo il progetto.

5.2.2 Unità di controllo microprogrammata

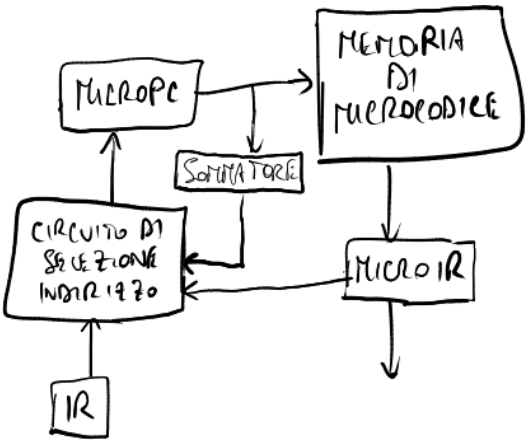


Figura 5.4: Struttura (semplificata) dell'unità di controllo microprogrammata.

L'**unità di controllo microprogrammata** ha una maggiore flessibilità di progetto, al prezzo di un maggiore costo hardware e di una minore velocità.

Memoria di microcodice

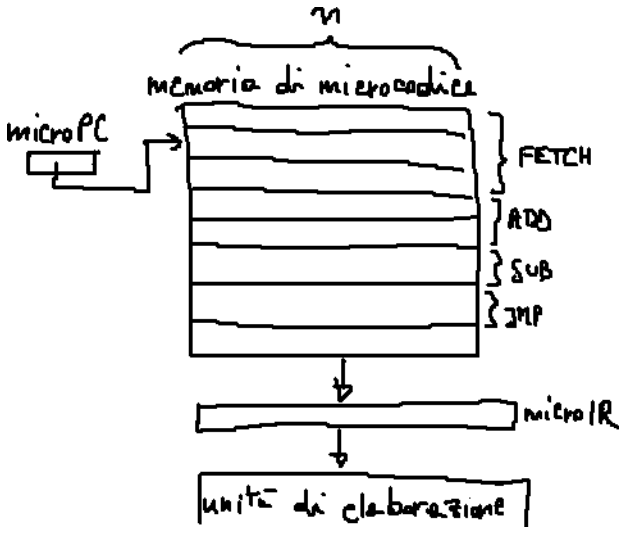


Figura 5.5: Esempio di memoria di microcodice.

Le sequenze di segnali di controllo relative a ogni microistruzione non sono generate da una logica di controllo sequenziale, ma memorizzate direttamente all'interno di una **memoria di microcodice**, di solito una ROM, pilotata da un **microPC**. Un registro **microIR** memorizza temporaneamente la sequenza di bit da inviare all'unità di elaborazione. All'inizio della memoria di microcodice si trova la sequenza per la fase di fetch; successivamente sono elencati i codici di controllo delle istruzioni a cui saltare in base al campo codice operativo specificato; altre istruzioni di salto alla fase di fetch successiva sono presenti al termine di ogni istruzione. Se il progettista si accorge di avere sbagliato un valore, può correggere quel valore direttamente nella memoria senza dover riprogettare tutto da capo.

Circuito di selezione dell'indirizzo

Il **circuito di selezione dell'indirizzo** si assicura che il microPC punti sempre alla parola corretta:

- normalmente il circuito di selezione invia il valore del microPC a un semplice sommatore che lo incrementa all'istruzione consecutiva;
- durante l'attesa dell'MFC, il circuito di selezione deve mantenere fisso il microPC fino a quando la memoria non è pronta;
- il circuito di selezione valuta i flag per le istruzioni di salto condizionato:
  - al termine della fase di fetch il microPC deve saltare alla prima microistruzione dell'istruzione letta;
  - all'ultima microistruzione dell'istruzione corrente il microPC deve tornare all'inizio della successiva fase di fetch.

All'inizio di una parola della memoria di microcodice, possono essere previsti alcuni bit riservati al circuito di selezione dell'indirizzo per aiutarlo, in termini di complessità della logica combinatoria, a decidere se la microistruzione successiva richiede un semplice incremento o un'ulteriore elaborazione del valore nel microPC. Altre soluzioni possono scegliere di specificare al termine della parola direttamente il nuovo valore da caricare nel microPC.

Microprogrammazione verticale

Il numero m di parole nella memoria di microcodice è difficile da minimizzare perché dipende dal numero di microistruzioni proprie del microprocessore e dalla loro complessità in termini di colpi di clock. Il parallelismo n è pari al numero dei segnali di controllo da pilotare a ogni colpo di clock, e può essere minimizzato attraverso tecniche di **microprogrammazione verticale**.

Per esempio, si possono individuare le classi di compatibilità aggiungendo un decoder in uscita di ciascuna. La **classe di compatibilità** è un gruppo di segnali di controllo a due a due **compatibili**, ovvero tra tutti i segnali ne viene attivato sempre e solo uno per colpo di clock. Un esempio è la classe di compatibilità dei segnali di controllo che pilotano i buffer tri-state connessi a un bus.

5.3 Unità di controllo dell'8088

L'unità di controllo dell'8088 è composta da una parte cablata e da una parte microprogrammata. La memoria di microcodice ha 504 parole di parallelismo 21.

In realtà, il codice operativo di ogni istruzione oltre a indicare il tipo di istruzione dà informazioni sul tipo di operandi (registro, memoria...). Gli operandi stessi (es. AX) vengono memorizzati all'interno dei registri M e N. Alla fine della fase di fetch, una PLA riceve il codice operativo dell'istruzione letta e restituisce il nuovo valore del microPC, selezionando l'indirizzo di partenza della sequenza di microistruzioni. Inoltre, le istruzioni che si differenziano per poche microistruzioni possono essere fatte corrispondere allo stesso blocco di memoria di microcodice, valutando quando necessario il contenuto del registro X per differenziarle.

Capitolo 6

Introduzione alle memorie

Accanto al processore e alle periferiche vi è un **sotto-sistema di memoria**. Si deve cercare un compromesso tra:

- prestazioni, ovvero il tempo di accesso a dati e istruzioni da parte del processore;
- costo.

Siccome la velocità dei processori raddoppia ogni anno e mezzo mentre quella delle memorie (in particolare i colpi di clock di attesa a ogni accesso) ogni dieci, gli accessi in memoria penalizzano sempre di più le prestazioni di processori veloci.

6.1 Livelli di memorie

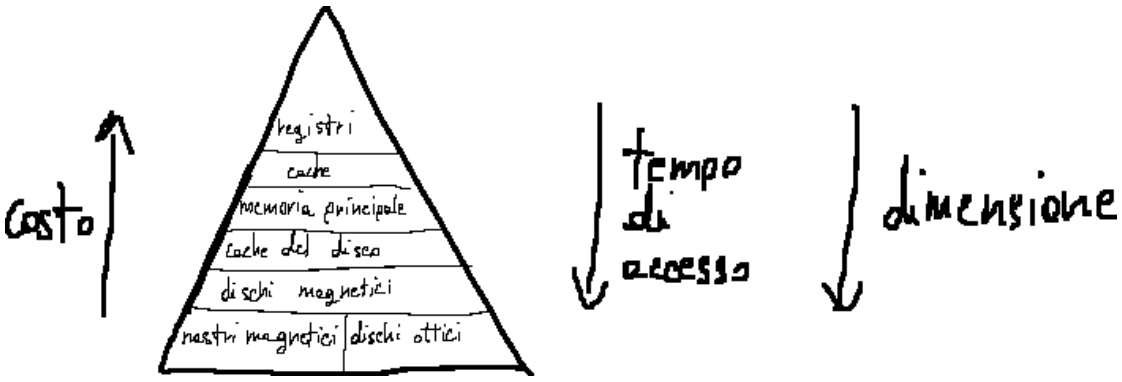


Figura 6.1: Piramide dei supporti di memorizzazione.

Le memorie sono caratterizzate dalla **località dei riferimenti** (= accessi): le applicazioni non accedono uniformemente all'intera memoria, ma statisticamente nella finestra temporale l'esecuzione delle istruzioni rimane limitata entro una certa zona della memoria (su istruzioni vicine), e analogamente la lettura/scrittura di dati interessa per un certo tempo solo alcuni dei dati. Perciò la maggior parte degli accessi in memoria non avviene in celle isolate, ma a un blocco di celle di memoria consecutive:

- **località temporale:** se all'istante t il programma accede a una cella di memoria, è probabile che nell'istante successivo $t + D$ acceda di nuovo alla stessa cella;
- **località spaziale:** se all'istante t il programma accede alla cella di memoria all'indirizzo X, è probabile che nell'istante successivo $t + D$ acceda alle celle vicine all'indirizzo $X + e$.

L'accesso allo stack è esclusivamente locale: le celle vengono possono essere accedute solo in modo consecutivo tramite le operazioni di push e pop.

Si possono quindi ottimizzare le prestazioni della memoria suddividendola in più **livelli**: le memorie di livello superiore, più piccole ma più veloci e costose per bit, sono destinate al blocco di dati e istruzioni su cui l'applicazione sta operando correntemente. Si deve però usare un meccanismo efficiente per posizionare i dati e le istruzioni nel livello corretto.

Al primo livello si trovano i registri, che costituiscono la **memoria interna** del processore: sono realizzati con RAM statiche e sono estremamente costosi.

All'interno della modulo RAM si distinguono una **memoria cache** veloce e la **memoria principale** vera e propria, realizzata con RAM dinamiche. Altre memorie cache sono utili all'interno del processore o montate sulla scheda madre.

La **memoria secondaria** è molto più capiente ma molto più lenta (es. dischi fissi). L'accesso è molto più complesso: il programma non può accedervi direttamente, ma richiede la chiamata a procedure offerte dal sistema operativo.

Memorie off-line come le chiavette richiedono anche operazioni meccaniche per l'accesso (es. inserimento nella porta USB).

6.2 Parametri

6.2.1 Costo

Più aumenta la velocità più aumenta il costo. Quando bisogna minimizzare le dimensioni fisiche (es. SoC), si somma il costo di progetto.

6.2.2 Prestazioni

Tempo di accesso

Il **tempo di accesso** (o latenza) è il tempo richiesto per soddisfare la richiesta dell'utente di lettura o scrittura:

- lettura: è il tempo che intercorre tra l'istante in cui il processore fornisce l'indirizzo e l'istante in cui la memoria fornisce l'MFC, che è il segnale di controllo che avvisa che il dato è stato letto;
- scrittura: è il tempo che intercorre tra l'istante in cui il processore fornisce il dato e l'istante in cui la memoria riceve il dato.

Un'efficiente gestione dei livelli di memoria riduce i tempi di accesso: i dischi magnetici e ottici hanno infatti un elevato tempo di accesso a causa di limitazioni meccaniche.

Alla fine dell'accesso non è detto che la memoria sia già pronta per iniziare un altro ciclo, cioè per leggere/scrivere un nuovo dato.

Tempo di ciclo

Il **tempo di ciclo** è il tempo che intercorre tra un ciclo di accesso e l'altro, cioè tra l'istante in cui la memoria principale riceve la richiesta di lettura/scrittura e l'istante in cui la memoria risulta pronta per una nuova lettura/scrittura. Ad esempio, nella scrittura è necessario aspettare che la memoria, dopo aver ricevuto il dato dal processore, lo scriva internamente. Il tempo di ciclo è quindi più lungo del tempo di accesso, o al più uguale.

Tasso di trasferimento

L'accesso alle memorie secondarie non avviene a singoli byte ma a blocchi (dell'ordine di kilobyte), perché si devono effettuare delle operazioni meccaniche preliminari di preparazione che richiedono un primo tempo di accesso per avviare la lettura/scrittura del blocco. Pertanto si deve cercare

di minimizzare il numero degli accessi sfruttando il più possibile l'accesso a blocchi ⇒ il **tasso di trasferimento** di una memoria secondaria è la velocità di lettura sequenziale del blocco, e si misura in bit per secondo (bps).

6.2.3 Altri parametri

- consumo
- portabilità
- dimensione

6.3 Modi di accesso

- **accesso casuale:** (es. RAM) ogni byte ha un indirizzo, e il tempo di accesso è lo stesso per qualsiasi byte;[1]
- **accesso sequenziale:** (es. nastri) il tempo di accesso dipende da quanto l'ordine di accesso segue l'ordine in cui sono memorizzati i dati;
- **accesso diretto:** (es. disco) ogni blocco ha un indirizzo, e l'accesso è casuale per l'individuazione del blocco e sequenziale all'interno del blocco stesso;
- **accesso associativo:** a ogni parola è associata una chiave binaria; le memorie ad accesso associativo sono molto costose.

6.4 Conservazione dei dati

L'**alterabilità** è la possibilità di modificare la memoria: le ROM non hanno alterabilità, le RAM sì.

La **volatilità** è l'incapacità di mantenere i dati memorizzati alla rimozione dell'alimentazione.

In una memoria con tecnologia **Destructive Readout**, i dati dopo che vengono letti vengono cancellati dalla memoria ⇒ vanno persi se non vengono riscritti successivamente in memoria.

Alcune memorie dinamiche hanno bisogno periodicamente di operazioni di **refreshing**, cioè di lettura e immediata riscrittura di tutti i dati, affinché non vadano persi.

Le memorie possono avere dei guasti, cioè delle differenze di comportamento da quello previsto; comprendono mancanze di risposta o bit scritti con valore sbagliato (per cause come le radiazioni). I **guasti permanenti** persistono a ogni tentativo, e non sono più riparabili; i **guasti transitori** capitano una volta sola. Le frequenze di guasto si dicono rispettivamente **Mean Time To Failures** (MTTF) e **Mean Time Between Failures** (MTBF).

[1] Si veda il capitolo 7.

Capitolo 7

Le memorie ad accesso casuale

Per le **memorie ad accesso casuale** vale:

- ogni cella può essere indirizzata indipendentemente;
- i tempi di accesso sono uguali e costanti per ogni cella.

7.1 Segnali di controllo e linee di stato

Il funzionamento della memoria è temporizzato: occorrono un certo tempo minimo per stabilizzare i segnali di dato e di indirizzo, un certo tempo minimo per accedere a quei segnali, un certo tempo minimo tra un accesso e l'altro, ecc. Il progettista può scegliere semplicemente di affidarsi ai tempi stabiliti dalle specifiche, oppure aggiungere dei circuiti appositi per valutare dinamicamente i segnali di controllo. I **segnali di controllo** servono per esempio:

- per informare la memoria sul tipo di operazione corrente (lettura o scrittura);
- per informare la memoria quando sono stabili e possono essere campionati i segnali di indirizzo sull'**Abus** e di dato sul **Dbus** (in scrittura);
- per informare il processore quando sono stabili e possono essere campionati i segnali di dato sul Dbus (in lettura);
- per informare il processore quando è possibile procedere con un nuovo ciclo di accesso alla memoria;
- per informare il processore se la memoria sta effettuando le operazioni di refreshing e i dati in lettura non sono stabili.

Accanto ai segnali di controllo vi sono le **linee di stato**, che sono dei segnali per informare l'esterno sullo stato corrente della memoria. Si usano generalmente per informare che si è verificato un guasto. Non comprendono l'MFC, perché esso non viene generato dalla memoria ma dal controllore della memoria.

7.2 Struttura interna

I bit effettivi che memorizzano le informazioni sono contenuti nella **matrice di memoria**. Il **circuito di controllo** riceve tramite il bus di controllo i segnali di controllo del processore (ad esempio i segnali di read e write), quindi pilota vari moduli all'interno della memoria, garantendo che i segnali caricati nei registri siano sempre stabili. Due registri di dato permettono la comunicazione con il bus di dati; ci può anche essere un transceiver[1] che permette la bidirezionalità del segnale.

[1] Si veda la sezione 2.3.6.

7.2.1 Organizzazione a vettore

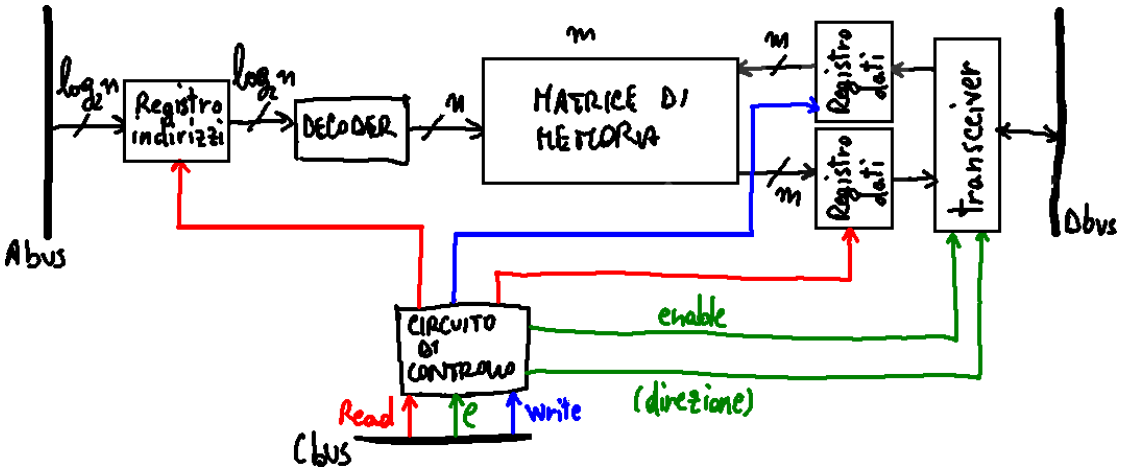


Figura 7.1: Struttura interna di una memoria ad accesso casuale con organizzazione a vettore.

La memoria è vista come un vettore di parole: l'indirizzo proveniente dal bus di indirizzi, di parallelismo $\log_2 n$, viene stabilizzato da un registro[2] degli indirizzi, pilotato tramite segnali di controllo dal circuito di controllo, e viene decodificato da un **decoder degli indirizzi** in n segnali di enable che selezionano la parola corretta all'interno della matrice di memoria. È semplice da implementare se il numero delle parole è ragionevole, altrimenti il decoder diventerebbe troppo complesso.

7.2.2 Organizzazione a matrice bidimensionale

La memoria è vista come una matrice organizzata in righe e colonne i cui elementi sono delle parole, ognuna delle quali ha un segnale di selezione di riga e uno di colonna, provenienti da due decoder. La parola viene attivata solamente se entrambi i segnali di selezione sono attivi. Due decoder di selezione risultano molto più semplici di un singolo decoder. Si può inoltre dimezzare il numero di segnali di indirizzo grazie a due registri: prima si salva l'indirizzo di riga in un registro, poi si salva l'indirizzo di colonna nell'altro registro, e infine si attivano i decoder.

I **segnali di strobe** `RAS` e `CAS` sono segnali di controllo che specificano se gli indirizzi devono essere caricati sul registro delle righe o delle colonne.

Si possono perciò velocizzare gli accessi sequenziali a blocchi di memoria (bit consecutivi), che sono statisticamente molto frequenti: la possibilità di attivare separatamente i registri di caricamento degli indirizzi permette di mantenere lo stesso indirizzo di riga al variare degli indirizzi di colonna (**Page Mode**).

7.3 Memorie non volatili

7.3.1 Memorie ROM

Il contenuto delle **memorie ROM** è definito solamente al momento della costruzione e successivamente può essere acceduto solo in lettura. Applicazioni:

- librerie di procedure frequentemente usate;
- programmi di sistema;

[2] Si veda la sezione 2.3.4.

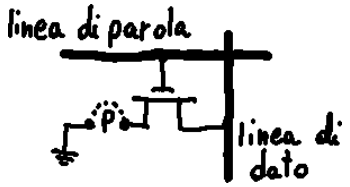


Figura 7.2: Cella di una memoria ROM.

- **tavole di funzioni:** sono gli elenchi dei risultati di funzioni che sarebbero troppo complessi da calcolare sul momento.

In fase di costruzione della memoria si agisce sul punto P che si trova tra ogni transistor e la massa: la cella assume il valore 1 se il punto P diventa un circuito aperto, e assume il valore 0 se P è un cortocircuito.

La variazione del costo è trascurabile al variare del numero di chip all'interno del singolo dispositivo rispetto al numero di dispositivi prodotti; si somma il costo di progetto. La memoria ROM è **non volatile**: il contenuto non viene cancellato all'assenza dell'alimentazione.

7.3.2 Memorie PROM

Nelle **memorie PROM**, agli incroci delle linee vi sono dei diodi che possono essere bruciati selettivamente tramite delle correnti e trasformati in circuiti aperti. Possono essere programmate una sola volta, prima di metterle sulla piastra.

7.3.3 Memorie EPROM

Le **memorie EPROM** possono essere riprogrammate più volte, anche se la cancellazione avviene grazie a raggi ultra-violetti. La programmazione può essere effettuata anche sulla piastra, poiché le correnti necessarie sono compatibili con la piastra stessa.

7.3.4 Memorie EEPROM

Le **memorie EEPROM** possono essere riprogrammate senza raggi ultra-violetti, ma la programmazione è lenta e sono costose.

7.3.5 Memorie flash

Le **memorie flash** richiedono delle operazioni di scrittura a blocchi, e il blocco deve essere ogni volta cancellato prima di poter essere riscritto.

7.4 Memorie RAM

Le memorie RAM possono essere di due tipi:

- **memorie statiche:** ogni bit richiede un flip-flop SR (6 transistor CMOS);
- **memorie dinamiche:** ogni bit richiede un condensatore e un transistor $\Rightarrow$ più densa.

7.4.1 Memorie RAM statiche

Ogni cella di una **memoria RAM statica** richiede 6 transistor: 4 costituiscono una combinazione di porte logiche, equivalente a un flip-flop SR[3] avente due uscite Y_1 e Y_2 opposte tra loro, e 2 collegano il flip-flop a due linee di dato parallele:

[3] Si veda la sezione 2.2.2.

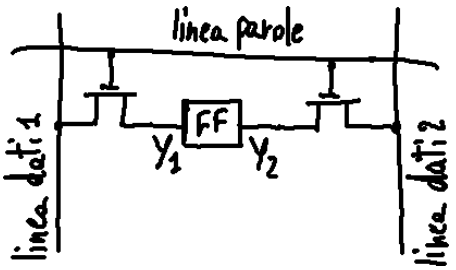


Figura 7.3: Cella di una memoria RAM statica.

- operazione di lettura: il flip-flop forza sulle linee di dato una coppia di bit opposti che viene poi elaborata dalla circuiteria associata alla colonna;
- operazione di scrittura: la circuiteria converte il bit nella corretta coppia di bit opposti che viene forzata sul flip-flop.

7.4.2 Memorie RAM dinamiche

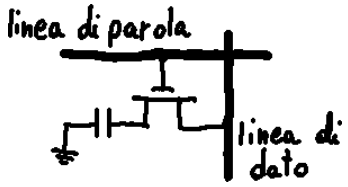


Figura 7.4: Cella di una memoria RAM dinamica.

Le celle di una **memoria RAM dinamica** contengono dei condensatori collegati alle linee di dato tramite transistor:

- operazione di scrittura: il condensatore viene caricato o scaricato alla tensione della linea di dato;
- operazione di lettura: un sensore rileva se la linea di dato rimane a un valore basso perché il condensatore è scarico (valore 0) o se sale a un valore alto perché è carico (1) ⇒ la memoria RAM dinamica è caratterizzata dal Destructive Readout[4]: se il condensatore è carico, si scarica e perde il bit 1 memorizzato ⇒ al termine di ogni ciclo di lettura è necessario riscrivere il dato appena letto.

Svantaggio le memorie dinamiche sono più lente delle memorie statiche a causa dei processi di carica e scarica del condensatore

Vantaggio le memorie dinamiche sono meno costose perché ogni cella richiede un solo transistor

I condensatori hanno una capacità infinitesima ⇒ qualunque minuscola particella carica (es. radiazione) può modificare il valore del condensatore (bit flip) ⇒ meno affidabilità.

Le memorie dinamiche richiedono quindi il refreshing: il contenuto deve essere periodicamente letto e riscritto, altrimenti i condensatori carichi lentamente si scaricherebbero per via delle inevitabili piccole correnti di dispersione che comunque scorrono. Gli accessi che capitano durante le operazioni di rinfresco richiedono quindi più tempo.

Il **codice di parità** è un codice di protezione che aumenta l'affidabilità della memoria: ogni parola ha un bit in più, che indica se il numero di bit pari a 1 dev'essere pari o dispari, permettendo di verificare se c'è stato un errore e quindi un bit è stato modificato.

[4]Si veda la sezione 6.4.

I **codici di Hamming** sono codici SECDED che permettono sia di rilevare sia di correggere gli errori singoli, ma si limitano a rilevare gli errori doppi (molto più rari). Questi codici richiedono $log_2 m$ bit, con m parallelismo.

Memorie SDRAM

Le **memorie Synchronous DRAM** (SDRAM) sono delle RAM sincrone temporizzate da un certo segnale di clock. Il processore quindi sa esattamente quanto tempo è richiesto per l'accesso ai dati.

Memorie interlacciate

Le memorie dinamiche spesso sono più lente rispetto alla velocità del processore, perché il tempo di accesso è più lungo dell'inverso del clock del processore. Con le **memorie interlacciate**, il processore può accedere contemporaneamente, tramite indirizzi di blocco, a un blocco di $n \cdot m$ parole distribuite su ognuna delle n memorie affiancate con parallelismo m bit.

Capitolo 8

Le memorie cache

Le **memorie cache** si trovano a un livello intermedio tra i registri e la memoria principale, e servono per mitigare i tempi di attesa durante gli accessi in memoria da parte del processore. Il processore non deve passare attraverso il bus per accedere alla cache.

8.1 Prestazioni

Una cache per essere gestita in modo efficiente deve contenere meno dati che non vengono acceduti possibile. Il tempo medio di accesso in memoria per la CPU è la media pesata tra M^* e C:

$$t_{\text{medio}} = hC + (1 - h)M^*$$

dove:

- h è la percentuale di accessi che avvengono con successo (**hit**) in cache;
- C è il tempo di accesso alla cache;
- M è la penalità di fallimento (**miss**), ossia il tempo di accesso in memoria quando il dato non si trova in cache;
- M^* è il tempo impiegato dal processore per accorgersi che il dato non si trova in cache sommato a M.

8.1.1 Gestione dei miss

In caso di miss, dopo che il dato è stato letto dalla memoria si può scegliere se passarlo subito al processore oppure se salvarlo prima in cache:

- il dato viene subito salvato in cache, e poi viene letto dal processore in cache ⇒ si ottiene una semplificazione dell'hardware, perché il processore comunica solo con la cache;
- il dato viene subito passato al processore, e poi viene salvato in cache dal cache controller ⇒ non c'è il ritardo della cache, ma il cache controller si complica.

8.1.2 Aggiornamento dei dati in cache

Se il processore deve modificare un dato in cache:

- **write-back:** ogni linea di cache ha un **dirty bit**, che viene impostato al momento della modifica di un dato, in modo che i dati modificati alla fine vengano salvati in memoria prima che la linea di cache venga sovrascritta;

- **write-through:** il processore scrive parallelamente sia in memoria sia in cache per mantenerle sincronizzate tra loro ⇒ le operazioni di scrittura non beneficiano della presenza della cache, ma i miss in lettura costano meno rispetto al write-back ed è comunque vantaggioso perché le operazioni di scrittura sono molto più rare di quelle di lettura.

8.2 Mapping

La cache è organizzata in **linee**, ciascuna delle quali contiene un blocco di memoria, e generalmente ha una dimensione di 8, 16 o 32 byte. Il **cache controller** si occupa di verificare rapidamente se il dato è in cache e di decidere quale linea riportare in memoria in caso di miss, ovvero quando è necessario far posto a un nuovo blocco. All'inizio di ogni linea vi è un **tag**, che memorizza l'indirizzo originario del blocco.

Di solito ogni linea termina con un **bit di validità**, che informa se è vuota o se contiene informazioni significative.

8.2.1 Direct mapping

Il **direct mapping** è caratterizzato da una corrispondenza fissa tra il blocco di memoria e la linea di cache.

Supponendo che la dimensione dei blocchi sia pari a 16 byte, l'indirizzo della cella di memoria richiesta dal processore viene prima diviso per 16, in modo da escludere i primi 4 bit meno significativi che costituiscono l'offset all'interno del blocco, e quindi viene diviso per il numero di linee[1], così i bit meno significativi che costituiscono il resto della divisione forniscono direttamente il numero della linea (ad esempio se il numero di linee è 4 si considerano solo i primi 2 bit meno significativi dell'indirizzo). In caso di miss, l'informazione viene letta dalla memoria e caricata in cache in base all'indice calcolato in questo modo, sovrascrivendo il vecchio blocco, e il tag viene aggiornato all'indirizzo del nuovo blocco privato dei bit che costituiscono l'indice della linea di cache.

Vantaggi il direct mapping richiede solo di dividere per potenze di 2, quindi con tempi di calcolo nulli, e di confrontare l'indirizzo del blocco richiesto con un solo tag

Svantaggio può capitare che il programma in esecuzione utilizzi di frequente blocchi aventi lo stesso resto della divisione, senza la possibilità di distribuire il carico di lavoro sulle altre linee di cache

8.2.2 Associative mapping

L'**associative mapping** non prevede la corrispondenza fissa, ma ogni blocco può essere memorizzato in qualsiasi linea di cache.

Vantaggio la hit ratio è maggiore

Svantaggio il cache controller deve confrontare l'indirizzo con tutti i tag ⇒ spreco di tempo

Il cache controller deve adottare una strategia che in caso di miss con cache piena decida quale blocco sovrascrivere:

- si sovrascrive un blocco a caso ⇒ economico;
- **coda FIFO:** si sovrascrive il blocco che si trova da più tempo in cache ⇒ può capitare di sovrascrivere un blocco che il processore utilizza tantissimo;

[1] Il numero di linee in una cache è sempre una potenza di due.

- **Least Recently Used (LRU):** si sovrascrive il blocco usato meno di recente ⇒ difficile da realizzare perché è necessario aggiornare la graduatoria delle linee di cache a ogni accesso in cache;
- **Least Frequently Used (LFU):** si sovrascrive il blocco usato meno di frequente ⇒ ancora più difficile da realizzare.

8.2.3 Set associative mapping

Nel **set associative mapping**, le linee di cache sono organizzate a insiemi, detti **set**; l'indirizzo di ogni set è ottenuto dal resto della divisione per direct mapping, e la scansione dei tag si limita al set ⇒ più efficiente. Anche nel set associative mapping, in caso di miss con set pieno il cache controller deve adottare una strategia.

8.3 Architettura Harvard

L'**architettura Harvard** si differenzia dall'**architettura di Von Neumann** per la separazione della memoria dati e della memoria codice.

Nell'architettura Harvard sono presenti due cache separate: una per i dati e l'altra per le istruzioni.

Vantaggio la cache per le istruzioni può evitare di supportare l'aggiornamento delle singole parole

Svantaggio si potrebbe avere uno spreco economico legato al cattivo dimensionamento della cache: soprattutto nei sistemi general-purpose, il costruttore non può prevedere se il programma utilizzerà molto una cache piuttosto che l'altra

8.4 Cache multiple

Nei sistemi a multiprocessore in cui un'unica memoria è **condivisa** attraverso un singolo bus, se ogni CPU non avesse la propria cache si verificherebbe un collo di bottiglia per l'accesso in memoria. Le attese dovute a due miss concomitanti sono compensate dalla maggiore potenza di calcolo. Se si esegue un programma distribuito su più processori, i programmi in esecuzione sui singoli processori non sono indipendenti, ma ogni tanto richiedono di comunicare tra loro attraverso dei dati condivisi ⇒ se un processore modifica un dato condiviso, anche la copia nella cache dell'altro processore dovrebbe essere aggiornata ⇒ esistono dei **protocolli di coerenza**:

- i dati condivisi possono non venire salvati in cache;
- un cache controller verifica se un altro cache controller sta accedendo a un dato condiviso tramite il bus e blocca la scrittura di quel dato all'interno della sua cache azzerando il bit di validità.

È conveniente avere più **livelli di cache**: il dato richiesto viene cercato a partire dalla cache di livello più basso (più veloce) a quella di livello più alto (più lenta), e tutto ciò che si trova in una cache deve trovarsi anche nelle cache di livello superiore. Nei sistemi a multiprocessore, le cache nei livelli più alti possono essere condivise da più processori.

Capitolo 9

Le memorie ad accesso seriale

Le **memorie ad accesso seriale** sono caratterizzate da un accesso più lento rispetto a quello delle memorie ad accesso casuale[1]: il tempo di accesso a un blocco richiede un tempo per iniziare ad accedere al blocco e un tempo per leggere/scrivere il blocco stesso. Il tempo di accesso al blocco dipende inoltre dalla posizione del blocco.

9.1 Dischi fissi

Un **disco fisso** è un disco magnetico[2] costituito da una pila di dischi. Su ogni faccia opera radialmente una testina; le testine sono collegate in modo solidale a un unico meccanismo di moto.

I bit sono memorizzati in maniera sequenziale su **tracce** disposte a circonferenze concentriche; ogni traccia è a sua volta suddivisa in **settori**. Per effettuare un'operazione di lettura o scrittura, la testina si posiziona sulla traccia, quindi il disco ruota per far scorrere la traccia sotto la testina.

I dischi magnetici hanno un'organizzazione **Constant Angular Velocity** (CAV): il disco gira a velocità angolare costante, indipendentemente dalla posizione della testina ⇒ la velocità lineare dei bit è bassa vicino al centro e alta vicino al bordo ⇒ le informazioni vanno memorizzate in modo meno denso vicino ai bordi. Vantaggio: la circuiteria che controlla la rotazione del disco è semplice.

Il tempo di accesso t_A a un generico settore è la somma di:

- **tempo di seek** t_S: la testina si posiziona sulla traccia ⇒ dipende se la traccia si trova lontana o vicina alla testina;
- **tempo di latenza** t_L: il disco ruota affinché l'inizio del settore da leggere si trovi sotto la testina ⇒ dipende dalla velocità di rotazione del disco;
- **'tempo di trasferimento dati** t_D: i dati vengono letti in modo seriale, rilevando un campo magnetico generato dalla magnetizzazione della traccia ⇒ dipende dalla distanza tra la testina e il disco (qualche micron) e dal numero di bit da leggere/scrivere.

Non si può aumentare troppo la densità di bit per unità di superficie, ma è necessario garantire una distanza minima tra una traccia e l'altra e tra la testina e la traccia per evitare le interferenze. Inoltre, se ci si basasse semplicemente sul segno della polarizzazione (il valore alto corrisponde a 1, il valore basso corrisponde a 0), non sarebbe quantificabile il preciso numero di bit in una sequenza di bit aventi uguale polarizzazione ⇒ i dati vanno codificati in modo da minimizzare le interferenze e rendere preciso il movimento della testina su ogni singolo bit: nella **codifica Manchester** (o di fase), ogni bit è codificato come una transizione di polarizzazione tra due bit (la transizione da valore basso a valore alto corrisponde a 0, e viceversa).

[1] Si veda il capitolo 7.

[2] Per dischi magnetici s'intendono i dischi fissi e i dischi floppy.

La ricerca del settore avviene da parte della testina iniziando a leggere la traccia, nella quale vi sono accanto alle informazioni delle intestazioni che specificano il settore corrente. L'inizio di un settore è codificato da una certa sequenza di byte specifici, in modo che la testina possa sincronizzarsi con l'inizio del settore e identificare nella sequenza di bit dove iniziano e finiscono i byte. Ogni settore è costituito da due **campi**, ID e dati, separati da zone della traccia, dette **gap**, in cui sono codificati i caratteri di sincronizzazione. Vi sono anche dei controlli di parità (CRC) per verificare la correttezza delle informazioni, e bit di informazione sulla traccia, sulla faccia e sul settore.

Il **disk controller** deve avere una circuiteria e una memoria (**data buffer**) per controllare le varie parti del disco, per il confronto con il CRC e per la decodifica dei dati. Il disk controller comunica con il processore attraverso un'interfaccia IDE, SATA, ecc.

Nel bootstrap, prima del primo accesso al disco fisso viene letta una ROM per la inizializzazione del sistema.

9.1.1 RAID[3]

Il **RAID** è una proposta di standard che serve per:

- rendere l'insieme fisico di più dischi equivalente a un unico disco virtuale, con parti hardware e software che gestiscono le richieste in maniera trasparente;
- migliorare l'affidabilità;

e può anche fornire qualche vantaggio in termini di prestazioni.

È possibile scegliere tra 6 configurazioni dette **livelli**: RAID 0, RAID 1, RAID 2, RAID 3, RAID 4, RAID 5.

RAID 0

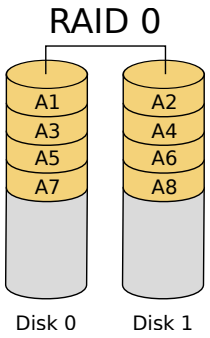


Figura 9.1: RAID 0.[4]

Nel **RAID 0**, i dati vengono distribuiti su **strisce** (strip) di dimensione variabile (dell'ordine di qualche byte), ciascuna stante su un disco. L'Array Management Software si interpone tra l'utente e i dischi fisici, che non sono legati in modo particolare tra loro.

Vantaggio la banda in lettura/scrittura è maggiore perché le operazioni di lettura/scrittura vengono effettuate in parallelo sui vari dischi

[3]Per approfondire, si veda la voce RAID su Wikipedia in italiano.

[4]Questa immagine è tratta da Wikimedia Commons (RAID 0.svg), è stata realizzata da Colin M. L. Burnett ed è concessa sotto la licenza Creative Commons Attribuzione - Condividi allo stesso modo 3.0 Unported.

RAID 1

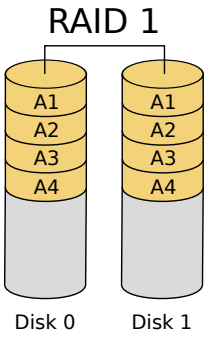


Figura 9.2: RAID 1.[5]

Anche nel **RAID 1** i dati sono distribuiti su strisce. I dischi sono suddivisi in due gruppi che sono in **mirroring** tra loro, cioè le informazioni sono replicate su entrambi i gruppi.

Vantaggio affidabile

Svantaggio lo spazio disponibile è dimezzato

RAID 2

Il **RAID 2** non è molto utilizzato perché è complicato: le strisce si riducono a singoli byte e sono lette in modo sincronizzato, il gestore è di tipo hardware, e sono riservati alcuni dischi ai codici di Hamming[6].

RAID 3 e RAID 4

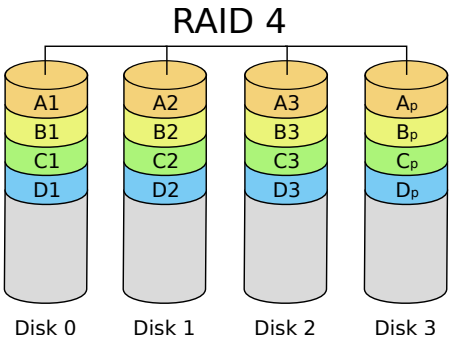


Figura 9.3: RAID 4.[7]

Il **RAID 3** e il **RAID 4** prevedono un solo disco riservato alle informazioni di parità. Il bit di parità non informa in quale disco si è verificato l'errore, ma se il sistemista riesce a sapere in quale disco si è verificato l'errore può utilizzare il disco di parità per ricostruire completamente le informazioni memorizzate in quel disco. Nel RAID 3 le strisce si riducono a singoli byte.

[5]Questa immagine è tratta da Wikimedia Commons (RAID 1.svg), è stata realizzata da Colin M. L. Burnett ed è concessa sotto la licenza Creative Commons Attribuzione - Condividi allo stesso modo 3.0 Unported.

[6]Si veda la sezione 7.4.2.

[7]Questa immagine è tratta da Wikimedia Commons (RAID 4.svg), è stata realizzata da Colin M. L. Burnett ed è concessa sotto la licenza Creative Commons Attribuzione - Condividi allo stesso modo 3.0 Unported.

Svantaggio le operazioni di lettura e scrittura in parallelo sono limitate dalle prestazioni del disco di parità, che potrebbe diventare un collo di bottiglia

RAID 5

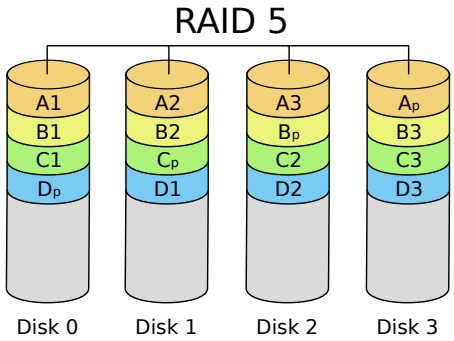


Figura 9.4: RAID 5.[8]

Il **RAID 5** distribuisce le informazioni di parità un po' su un disco un po' su un altro.

Vantaggio si evitano colli di bottiglia

9.2 Memorie a nastro

Nelle **memorie a nastro**, le informazioni sono memorizzate su tipicamente 9 tracce, di cui l'ultima contenente le informazioni di parità; la testina legge serialmente il nastro che si avvolge. Ogni traccia è organizzata in record, distanziati da gap per la sincronizzazione della lettura.

9.3 Memorie ottiche

Le **memorie ottiche** hanno una maggiore affidabilità: i guasti transitori sono più rari e la durata nel tempo è maggiore.

I dischi ottici hanno un'organizzazione **Constant Linear Velocity** (CLV): il motore ruota a velocità lineare costante, ma a velocità angolari diverse a seconda della posizione del raggio laser (che prende il posto della testina) ⇒ il controllore è più complicato.

[8] Questa immagine è tratta da Wikimedia Commons (RAID 5.svg), è stata realizzata da Colin M. L. Burnett ed è concessa sotto la licenza Creative Commons Attribuzione - Condividi allo stesso modo 3.0 Unported.

Capitolo 10

La memoria virtuale

Quando la memoria principale fisica viene riempita, è necessario estenderla a una **memoria virtuale**, così il processore può accedere a indirizzi di memoria che superano lo spazio di indirizzamento. Secondo il principio di località dei riferimenti[1], nella memoria principale fisica si trovano le informazioni più usate, e le informazioni di minor utilizzo sono archiviate temporaneamente nella memoria secondaria. La memoria virtuale è suddivisa in blocchi detti **pagine**, grandi da 1 KB a 64 KB. Il processore e i programmi in realtà vedono solo un'unica memoria di dimensione corrispondente alla memoria complessiva.

Uno strato hardware, detto **Memory Management Unit** (MMU), intercetta gli **indirizzi logici** provenienti dal processore e li converte attraverso la **Memory Address Table** (MAT) negli equivalenti indirizzi fisici. In caso di miss, ossia quando l'informazione non si trova in memoria principale, il sistema operativo, attivato dalla MMU attraverso il segnale di **page fault**, interrompe il programma in esecuzione e trasferisce l'informazione richiesta dalla memoria virtuale a quella fisica, quindi riprende l'esecuzione del programma. Il meccanismo è efficiente se i page fault sono in numero minimo.

La MMU è integrata nel processore. La conversione degli indirizzi logici in fisici, compiuta prima di arrivare alla cache, deve essere effettuata in tempi minimi.

La MAT è memorizzata in memoria principale ⇒ ogni accesso a un indirizzo fisico richiederebbe un ulteriore accesso in memoria. Statisticamente in un intervallo di tempo il processore accede frequentemente alla stessa pagina ⇒ il **Translation Lookaside Buffer** (TLB) è una cache posta direttamente a bordo della MMU che memorizza le ultime entry della MAT usate.

Siccome la MMU opera via software, la hit ratio è più elevata di quella delle cache che funzionano in modo hardware, perché le pagine in memoria principale sono effettivamente quelle di maggior utilizzo.

[1] Si veda la sezione 6.1.

Capitolo 11

La gestione dei dispositivi input/output

Il processore non comunica direttamente (tramite il bus) con i periferici, ma ci sono dei moduli di interfaccia detti **porte**. Ogni porta contiene dei registri per lo scambio di dati e per le informazioni di stato, che hanno degli indirizzi specifici: siccome il processore conosce solo quegli indirizzi, è compito della porta far comunicare il processore e il periferico.

Alcuni circuiti di controllo gestiscono il funzionamento del registro. Un segnale di read/write e il decodificatore di indirizzi specificano rispettivamente il tipo di operazione e su quale registro vuole operare il processore.

11.1 Modalità di indirizzamento

11.1.1 Modalità Memory-Mapped-IO

In **modalità Memory-Mapped-IO**, le periferiche e la memoria sono mappate in un unico spazio di indirizzamento: ogni porta di periferica ha un indirizzo, e il processore comunica con le porte di periferica in modo analogo a come comunica con le celle di memoria; la **logica di decodifica** si occupa di accedere o alla memoria o alla periferica basandosi solamente sull'indirizzo. Sebbene sia supportata dall'8086, la scelta di questa modalità esclude la possibilità di utilizzare alcuni indirizzi di memoria, perché una cella di memoria e una porta di periferica non possono avere lo stesso indirizzo.

11.1.2 Modalità Isolated-IO

La **modalità Isolated-IO** offre due spazi di indirizzamento distinti per la memoria e le periferiche. Il segnale IO/M specifica se l'indirizzo fornito dal processore è destinato alle periferiche o alla memoria, ma è un segnale aggiuntivo che deve essere decodificato.

11.2 Sincronizzazione

La comunicazione tra la periferica e il processore deve avvenire alla velocità massima, e nessuno dei due deve penalizzare le prestazioni della comunicazione. Solitamente il processore è molto più veloce delle periferiche $\Rightarrow$ serve un meccanismo di **sincronizzazione**.

11.2.1 I/O programmato

L'**I/O programmato** è un meccanismo completamente software che si basa su un driver per la gestione completa della periferica. Il processore campiona (= **polling**) il segnale di ready della

periferica finché non è pronta, in modo che il trasferimento di dati tramite i registri avvenga in modo sincronizzato.

Svantaggio il codice che effettua il polling è un semplice ciclo da cui il processore non esce finché la periferica non è pronta ⇒ fino al termine del ciclo il processore è impegnato unicamente ad effettuare il polling in attesa che la periferica sia pronta.

11.2.2 Interrupt

La **richiesta di interruzione** (o interrupt) è una situazione di "emergenza" in cui le periferiche chiedono al processore di interrompere l'esecuzione di una serie di istruzioni.

Quando la periferica è pronta, può richiamare l'attenzione del processore con una richiesta di interruzione: il segnale di INT viene campionato dal processore al termine dell'esecuzione di ogni istruzione. In questo modo, finché non arriva la richiesta di interruzione il processore può effettuare altre operazioni anziché passare tutto il tempo a testare il segnale di ready. Quando il processore rileva una richiesta di interruzione, passa alla procedura di servizio **Interrupt Service Routine** (ISR), che è una serie di istruzioni in memoria che gestiscono la richiesta.

Il processore salva via hardware i contenuti correnti del PC e della parola di stato prima di richiamare la ISR. La ISR deve all'inizio salvare via software nello stack eventuali registri e parole di memoria utilizzati.

Si dice che l'interrupt è vettorizzato perché esiste una Interrupt Vector Table che associa ogni codice di interrupt all'indirizzo della corrispondente ISR.

L'**Interrupt Controller** gestisce il flusso di richieste di interruzione provenienti da più periferiche. L'Interrupt Acknowledge (INTA) è un segnale fornito dal processore all'Interrupt Controller per informarlo che è pronto a servire una periferica e che ha bisogno dell'indirizzo della periferica che ha fatto la richiesta ⇒ l'Interrupt Controller glielo invia tramite il bus dati.

Il bit I nei registri di stato STI e CLI può disattivare temporaneamente l'ascolto delle richieste di interruzione da parte del processore. L'Interrupt Controller può essere programmato in modo da disabilitare selettivamente le periferiche, oppure associando delle priorità alle periferiche in modo da servire prima le periferiche con maggiore priorità in caso di richieste simultanee. Inoltre, grazie alle priorità si può implementare una soluzione più complicata: la procedura di servizio di una periferica con una certa priorità può venire interrotta da un'altra periferica a priorità maggiore, mentre vengono escluse le periferiche a priorità minore.

La **latenza di interrupt** è il ritardo massimo di reazione alle richieste di interruzione, cioè il tempo massimo tra l'attivazione del segnale di INT e l'avvio della ISR. Il processore 8086 ha un'elevata latenza di interrupt (l'istruzione `DIV` per esempio richiede più di 100 colpi di clock) ⇒ l'8086 non è adatto per le applicazioni real-time.

Altre cause di richiesta di interruzione

Segnali di errore Le richieste di procedure di servizio possono essere scatenate da:

- un'eccezione incontrata da un'operazione svolta dal processore (ad es. divisione per zero);
- un'eccezione di illegal instruction, cioè il codice operativo letto dalla memoria non corrisponde ad alcuna istruzione esistente (ad es. il codice $00\ldots0$ spesso non corrisponde a un'istruzione esistente ma è riservato alla terminazione del programma);
- un errore di parità durante la lettura dalla memoria.

Segnali per il debug Il processore può essere impostato in modalità debug tramite il flag di interrupt:

- modalità trace: al termine di ogni istruzione (modalità single step), il processore richiama la procedura di servizio che gestisce l'operazione di debug;

- il programmatore può impostare un breakpoint su un'istruzione, che corrisponde a una istruzioni di salto alla procedura del programma di debug ⇒ il processore, dopo aver eseguito l'ultima istruzione, si ferma e salta alla procedura del programma di debug, dopodiché se necessario ritorna a eseguire il codice;
- alcune particolari strutture dati possono essere usate dal debugger per accedere a variabili e registri.

Eccezioni di privilegio Molti processori hanno due modalità di funzionamento:

- modalità supervisore: è prevista per il sistema operativo;
- modalità utente: è più limitata, ed esclude delle istruzioni critiche che se il programma in esecuzione tenta di eseguire scatenano le eccezioni di privilegio (ad es. l'istruzione `ALT` che blocca il processore deve essere richiamata solo dal sistema operativo).

11.2.3 Soluzione ibrida

Una soluzione semplice ed economica prevede l'uso di una porta logica OR come Interrupt Controller semplificato. Per capire da quale periferica arriva la richiesta, il processore deve eseguire un'unica procedura di servizio che legga tutti i segnali di ready delle periferiche (polling).

11.3 DMA

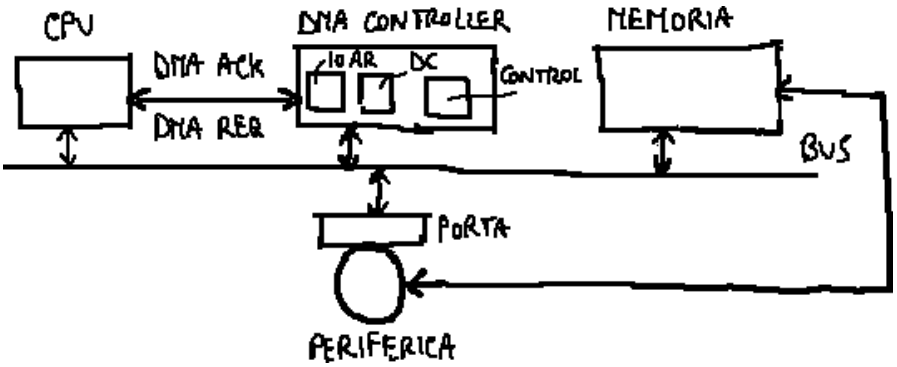


Figura 11.1: Sistema con DMA Controller.

Il **Direct Memory Access Controller** (DMA Controller) è un dispositivo, separato dal processore, in grado di gestire il trasferimento di grosse moli di dati tra una periferica e la memoria, ed evita che ogni singolo byte debba passare attraverso un registro del processore appesantendo quest'ultimo.

Devono esistere dei segnali di coordinamento tra il DMA Controller e:

- il processore: il DMA Controller è a sua volta una periferica in grado di gestire cicli di bus, cioè in grado di diventare il bus master[1] ⇒ deve esistere un meccanismo di arbitraggio del bus[2] (segnali DMA Request e DMA Acknowledge) in modo che il processore e il DMA Controller non facciano accesso al bus contemporaneamente;
- la periferica: il DMA Controller e la periferica sono collegati, oltre che attraverso il bus, anche in maniera diretta da un segnale bidirezionale di sincronizzazione, in modo che la periferica possa informare il DMA Controller quando è pronta (e viceversa).

[1] Si veda il capitolo 14.
[2] Si veda la sezione 14.3.

11.3.1 Fasi di trasferimento

1. programmazione del trasferimento: il processore ordina al DMA Controller che è necessario effettuare un certo trasferimento specificando l'indirizzo di partenza del blocco da trasferire (tramite la porta **IOAR**), la dimensione del blocco da trasferire (porta **DC**), e la direzione del trasferimento (registro di controllo);
2. il DMA Controller aspetta che la periferica sia pronta;
3. la periferica comunica al DMA Controller che è pronta;
4. segnale **DMA Request**: il DMA Controller fa sapere al processore che intende prendere il controllo del bus;
5. segnale **DMA Acknowledge**: il processore informa il DMA Controller che si è sganciato dal bus;
6. il DMA Controller diventa il bus master[3] e inizia il trasferimento;
7. alla fine del trasferimento, il DMA Controller rilascia il bus e informa il processore che è pronto per un altro trasferimento.

11.3.2 Modi di funzionamento

- **burst transfer:** il trasferimento avviene a blocchi, e la CPU non può accedere al bus fino a quando non è stato trasferito l'intero blocco ⇒ le prestazioni del sistema dipendono dall'efficienza delle cache del processore;
- **cycle stealing:** ogni tanto il DMA Controller cede temporaneamente il controller del bus al processore ⇒ il trasferimento è più lento, ma il processore non è bloccato per periodi troppo lunghi;
- **transparent DMA:** il DMA Controller è in grado di capire (tramite dei segnali di stato del bus) quando il processore sta usando il bus, e può sfruttare i momenti in cui il bus è libero per continuare il trasferimento ⇒ il processore è rallentato il meno possibile perché non c'è la negoziazione del bus.

[3] La memoria e la periferica non si accorgono che il bus master non è più il processore ma il DMA Controller.

Capitolo 12

Intel 8255

L'**Intel 8255** è un dispositivo, messo in commercio alla fine degli anni '70, per gestire le **interfacce parallele**, che permettono il trasferimento di un byte alla volta. Oggi i segnali che erano diretti a tale modulo sono supportati da un modulo inserito nel processore.

Il segnale di enable si chiama Chip Select (CS).

L'interfaccia parallela contiene 4 porte (porta A, porta B, porta C, **registro di controllo** CW) di parallelismo 8 bit. I primi 18 bit alti degli indirizzi forniti dal processore vengono ignorati (l'8086 ha un ABus su 20 bit), mentre i due bit bassi A0 e A1 identificano il registro su cui operare.

12.1 Modalità

Le periferiche possono lavorare in diverse modalità, forzate da alcuni bit del registro di controllo.

12.1.1 Modo 0: Basic Input/Output

Il **Basic Input/Output** non supporta né gli interrupt né i segnali di controllo, e ciascuna porta può essere impostata in input o in output.

Vantaggio le porte A, B e C possono così essere collegate a 3 periferiche diverse, se queste richiedono ciascuna solo una porta

12.1.2 Modo 1: Strobed Input/Output

Il **Strobed Input/Output** supporta l'interrupt tramite la porta C $\Rightarrow$ solo le porte A e B possono venire usate per il trasferimento dati.

L'Intel 8255 può inviare la richiesta di interrupt al processore tramite il segnale INTR.

Segnali di controllo tra periferica e Intel 8255

Input

- INTE: serve per ignorare temporaneamente le richieste di interrupt;
- STB (strobe): la periferica chiede all'Intel 8255 di caricare internamente il dato fornito dalla periferica;
- IBF: l'Intel 8255 comunica all'esterno che ha acquisito internamente il nuovo dato.

Output

- OBF: l'Intel 8255 informa che il dato è pronto per la lettura;
- ACK (acknowledge): la periferica informa che ha finito di acquisire il nuovo dato dall'Intel 8255.

12.1.3 Modo 2: Bidirectional Input/Output

Il **Bidirectional Input/Output** supporta segnali che sono un po' in input e un po' in output.

Può essere utilizzata solo la porta A: la porta C si occupa come nel modo 1 dei segnali di input/output (STB, IBF, OBP e ACK).

Capitolo 13

Intel 8259

L'**Intel 8259** è un interrupt[1] controller in grado di gestire fino a 8 periferiche. L'Intel 8259 supporta anche la gestione delle priorità associate alle periferiche.

13.1 Struttura

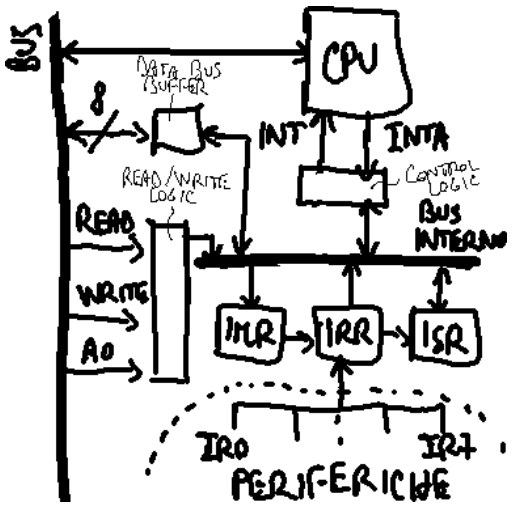


Figura 13.1: Struttura dell'Intel 8259.

Il processore può leggere e scrivere dati su 8 bit nei registri dell'Intel 8259 tramite il bus:

- i segnali di read e write specificano il tipo di operazione (lettura o scrittura);
- il segnale A0 specifica su quale registro operare.

L'Intel 8259 contiene principalmente 3 registri:

- l'**Interrupt Mask Register** (IMR) permette il mascheramento selettivo delle periferiche;
- l'**Interrupt Request Register** (IRR) riceve i segnali di richieste di interruzione dalle periferiche;
- l'**Interrupt Service Register** (ISR) tiene traccia delle periferiche la cui procedura di servizio è ancora in corso: il processore tramite l'ISR può far sapere all'Intel 8259 quando

[1] Si veda la sezione 11.2.2.

inizia e quando finisce la procedura di servizio di una periferica, così se nel frattempo una periferica a priorità inferiore fa richiesta l'Intel 8259 attiva il segnale di INT per la nuova periferica solo dopo che il processore ha terminato di servire la periferica a priorità maggiore.

13.2 Programmazione

Il processore può programmare l'Intel 8259 inviandogli alcune parole di controllo.

13.2.1 ICW

In fase di inizializzazione, il processore invia le **Initialization Command Words** (ICW):

- ICW1: specifica se l'Intel 8259 è in cascata e se esso dev'essere sensibile ai fronti o ai livelli;
- ICW2: contiene il codice associato alla prima periferica (i codici delle altre periferiche sono consecutivi);
- il processore specifica anche la modalità di gestione delle priorità (statica o rotante).

13.2.2 OCW

Durante il funzionamento, il processore può inviare le **Operation Command Words** (OCW):

- OCW1: specifica quali dispositivi mascherare selettivamente agli interrupt;
- OCW2: serve per cambiare le priorità.

Capitolo 14

I bus

Il **bus** è la struttura di interconnessione condivisa tra la CPU e il resto del sistema (memoria, periferiche). È composto da:

- bus di indirizzi: segnali di indirizzo
- bus dati: segnali di dato
- bus controllo: segnali di controllo

Solo un dispositivo per volta può scrivere sul bus; possono essere più di uno i dispositivi che leggono dal bus. Quando un dispositivo diventa il **bus master** (o master del bus), prende il controllo del bus e decide a ogni ciclo il tipo di operazione da effettuare (ad es. ciclo di lettura dalla memoria, ciclo di scrittura su un periferico, ciclo di Interrupt Acknowledge).

14.1 Implementazione

- bus interno a un circuito integrato (ad es. il processore);
- pista a bordo di una scheda su cui sono montati più circuiti integrati;
- **bus di backplane**: può collegare più schede.

In un **bus multiplexato** le stesse linee portano a seconda dei momenti segnali di dato o di indirizzo ⇒ si risparmiano linee di bus, ma è necessaria una circuiteria per la connessione alla memoria e la velocità è minore rispetto ai bus non multiplexati.

Nelle **architetture a bus multiplo**, più bus sono organizzati gerarchicamente in base alla velocità e sono connessi tra loro attraverso dei processori di IO oppure dei **bridge**, che si occupano al momento opportuno di leggere il segnale dal bus di livello superiore e di restituirlo al bus di livello inferiore.

Da/su un **bus condiviso** possono leggere/scrivere più moduli. I **bus dedicati** sono impropriamente strutture di comunicazione tra due moduli. Il bus condiviso è più economico, ma ha prestazioni meno elevate rispetto al bus dedicato, perché non può supportare più di una comunicazione in contemporanea.

14.2 Sincronizzazione

14.2.1 Bus sincroni

Tutti i dispositivi collegati a un **bus sincrono** condividono un unico segnale di clock, e la frequenza di clock è imposta dal dispositivo più lento. È preferibile che non ci siano dei dispositivi

troppo lenti o troppo lontani tra di loro o dalla sorgente di clock a causa dei ritardi fisici del segnale.

Ogni volta che un dispositivo slave non riesce a completare un'operazione sul bus entro il tempo prestabilito, il bus master deve aggiungere dei colpi di clock, chiamati **cicli di wait**, fino a quando il dispositivo non completa l'operazione e non attiva il segnale di ready. I cicli di wait devono essere però richiesti il meno possibile, ad esempio da dispositivi acceduti raramente o da memorie impegnate nel refreshing.

14.2.2 Bus asincroni

La soluzione a **bus asincrono** non prevede un segnale di clock comune, ma ogni ciclo di bus è delimitato dai segnali di controllo strobe e acknowledge:

- **strobe:** il dispositivo sorgente comunica che il segnale è stato scritto sul bus;
- **acknowledge:** il dispositivo di destinazione comunica che ha terminato la lettura del segnale dal bus.

La durata del trasferimento dipende solo dalle velocità dei due dispositivi coinvolti. La soluzione sincrona è più complessa e costosa rispetto a quella asincrona perché sono necessari dei segnali di controllo e una logica che li piloti.

14.3 Arbitraggio

Quando una risorsa è condivisa, serve un meccanismo di arbitraggio per gestire le situazioni di contesa:

- **arbitraggio del bus:** la CPU e il DMA Controller[1] vogliono diventare bus master nello stesso momento, ma ad ogni istante un solo dispositivo può funzionare da master del bus;
- più processori voglio fare accesso in contemporanea a un disco fisso;
- più dispositivi vogliono effettuare una richiesta di interruzione al processore (si veda la sezione 11.2.2).

14.3.1 Arbitraggio distribuito

L'**arbitraggio distribuito** non prevede alcun arbitro. Nel caso del **bus SCSI**, ad esempio, ogni dispositivo che vuole essere promosso a bus master deve prima accertarsi che non esista un altro dispositivo a priorità più alta (cioè su una linea `DB(i)` con numero `i` maggiore) che a sua volta intende diventare il bus master.

14.3.2 Arbitraggio centralizzato

L'**arbitraggio centralizzato** prevede un arbitro che decide chi promuovere a bus master per il prossimo ciclo di bus. Tutti i dispositivi sono collegati al segnale Bus Busy che indica se il bus è attualmente libero o occupato da un bus master.

Richieste indipendenti

Con il meccanismo delle **richieste indipendenti** ogni dispositivo è collegato all'arbitro tramite una coppia di segnali di controllo:

- **Bus Request:** il dispositivo richiede di essere promosso a bus master;
- **Bus Grant** (o Acknowledge): l'arbitro concede al dispositivo la promozione a bus master.

[1] Si veda la sezione 11.3.

È una soluzione costosa perché servono molti segnali di controllo e un arbitro che implementi una strategia intelligente.

Daisy Chaining

Nel **Daisy Chaining** i vari segnali Bus Request delle unità sono collegati in wired-or[2]. Quando arriva una richiesta il Bus Grant, segnale unico, inizia a scorrere nella catena delle unità alla ricerca di quella che ha fatto la richiesta:

- se l'unità corrente è quella che ha fatto la richiesta, essa attiva il Bus Busy e prende il controllo del bus;
- se l'unità corrente invece non ha fatto richiesta, passa il Bus Grant all'unità successiva.

È una soluzione semplice ed economica perché il numero dei segnali è indipendente dal numero di unità, ma presenta delle caratteristiche svantaggiose:

- poco flessibile: le priorità sono fisse, e l'ultimo dispositivo nella catena ha la priorità minima;
- lenta: il Bus Grant impiega un certo tempo a scorrere la catena;
- poco affidabile: se un'unità precedente nella catena smette di funzionare, la catena si interrompe.

Polling

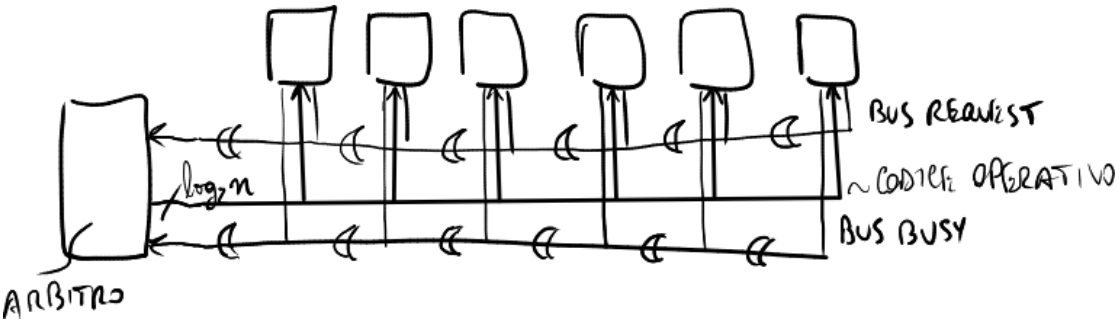


Figura 14.1: Esempio di arbitraggio centralizzato con polling.

La soluzione con **polling** differisce dal Daisy Chaining per il meccanismo del Bus Grant. Quando il bus si libera e almeno un'unità ha fatto la richiesta (tramite connessione wired-or), l'arbitro scandisce la catena interrogando i dispositivi uno alla volta, e si ferma al primo che risponde e attiva il Bus Busy. A ogni unità è associato un codice identificativo binario, che durante la scansione della catena viene fornito dall'arbitro su un'altra linea a cui sono collegate tutte le unità.

Vantaggi

- maggiore flessibilità all'interno dell'arbitro: cambiando l'ordine con cui l'arbitro scandisce i dispositivi è possibile implementare qualsiasi meccanismo di gestione delle priorità;
- maggiore tolleranza ai guasti: se un'unità si guasta le altre continuano a funzionare.

[2]Si veda la sezione 2.2.1.

Capitolo 15

Le architetture a pipeline

Per migliorare le prestazioni del processore, oltre ad aumentarne la frequenza, si può modificarne l'architettura.

I processori si differenziano in base al tempo, in termini di colpi di clock, richiesto per il completamento delle istruzioni:

- **CISC:** (fino agli anni '70) per completare un'istruzione è necessario un certo numero di colpi di clock;
- **RISC:** (anni '80) completano un'istruzione in un solo colpo di clock;
- **superscalari:** (anni '90) a ogni colpo di clock completano più di un'istruzione.

15.1 Architettura RISC

I processori con architettura RISC fanno uso delle **pipeline**.

Nelle istruzioni si possono identificare delle fasi standard (per es. Fetch, Decode, Operate, Write), ciascuna delle quali opera in un solo **stadio** del processore per colpo di clock ⇒ i vari stadi possono essere fatti lavorare in parallelo, così mentre viene completata un'istruzione ce n'è già un'altra a cui manca una fase per il completamento, e il primo stadio inizia a eseguire un'altra istruzione ancora ⇒ a ogni colpo di clock viene completata un'istruzione.

Gli stadi sono inframezzati da registri, sincronizzati dallo stesso segnale di clock.

15.1.1 Instruction set

Il set di istruzioni deve comprendere istruzioni semplici, e le istruzioni devono avere un codice operativo di dimensione regolare in modo che la decodifica sia veloce. C'è un unico modo di indirizzamento, che si basa su un registro e una costante.

Tutte le istruzioni lavorano sui registri, tranne le istruzioni `LOAD` e `STORE` che rispettivamente leggono e scrivono tra la memoria e un registro. In questo modo si evitano istruzioni che effettuano accessi pesanti alla memoria. I RISC dispongono di un numero significativo di registri (~ 100).

Tipicamente l'unità di controllo[1] è cablata e non microprogrammata, grazie al fatto che la gestione dell'unità di elaborazione è più semplice. Un'unità di controllo cablata è inoltre più veloce e più piccola di una microprogrammata.

15.1.2 Stalli

Tutte le istruzioni devono passare attraverso le stesse fasi; non sono ammessi dei fetch che richiedono più colpi di clock, ma ogni fase deve durare un solo colpo di clock.

[1] Si veda il capitolo 5.

Possono verificarsi dei casi imprevedibili, detti **stalli**, in cui non si fa in tempo a eseguire una fase entro un singolo colpo di clock (ad es. fetch).

I ritardi del fetch possono essere attenuati con una **coda delle istruzioni** subito dopo la fase di fetch, che normalmente è piena. Se si verifica un miss, lo stadio di decodifica continua a lavorare sulle istruzioni rimaste nella coda.

15.1.3 Vincoli di dipendenza

Se alcune istruzioni richiedono di operare sul risultato dell'istruzione precedente, oppure le istruzioni in esecuzione in parallelo condividono degli operandi:

- soluzione hardware: il processore aspetta che l'istruzione precedente finisca di accedere al dato condiviso;
- soluzione software: il compilatore aggiunge delle istruzioni `NOP` ⇒ l'hardware è più semplice.

15.1.4 Istruzioni di salto

Nella pipeline vengono caricate le istruzioni in modo consecutivo ⇒ quando l'istruzione di salto scrive il nuovo valore nel PC, l'istruzione consecutiva nel frattempo è già stata caricata in pipeline:

- soluzione software: il compilatore aggiunge delle istruzioni `NOP` in fase di compilazione;
- soluzione hardware: il processore deve annullare l'esecuzione delle istruzioni in pipeline da saltare.

Salti condizionati

I salti condizionati presentano un'ulteriore complicazione, perché non è detto che modifichino il PC.

Già in fase di decodifica dell'istruzione, il processore può tentare di fare una predizione, per evitare il più possibile di caricare inutilmente istruzioni, attraverso opportuni algoritmi basati sull'analisi del codice e/o su statistiche dinamiche calcolate in fase di esecuzione (ad es.: la volta precedente l'istruzione ha saltato?).

15.1.5 Vantaggi

Un codice ottimizzato per i processori RISC risulta più veloce e di dimensioni comparabili rispetto all'alternativa CISC, perché le istruzioni eseguite nonostante siano in numero maggiore sono più semplici sia in termini di colpi di clock sia in termini di lunghezza in byte. Un codice compilato per un particolare processore RISC però è poco flessibile in termini di compatibilità con altri processori RISC ⇒ per questo motivo all'inizio non vennero utilizzati molto in ambito general-purpose ma solo in ambito special-purpose. I processori general-purpose odierni invece usano comunque le pipeline, grazie a dell'hardware che traduce al volo le istruzioni generiche dell'assembler 8086 ad istruzioni più semplici di tipo RISC.

Nei RISC la **latenza**, ovvero il tempo medio di attesa per il servizio di un interrupt, è molto bassa perché a ogni colpo di clock viene completata un'istruzione.

15.2 Processori superscalari

I processori superscalari usano due pipeline in parallelo per migliorare le prestazioni: a ogni istante sono in esecuzione 8 istruzioni in parallelo. Svantaggio: i vincoli di dipendenza raddoppiano.

Le unità di fetch, decodifica e write-back elaborano due istruzioni per colpo di clock. La fase di operate è svolta da varie unità differenziate a seconda del tipo di istruzione: unità per l'aritmetica intera, unità per l'aritmetica in virgola mobile...

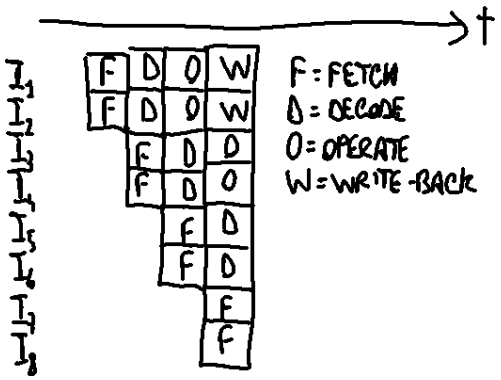


Figura 15.1: Flusso di esecuzione.

15.2.1 Completamento non-in-ordine

L'unità di decode diventa un'**unità di smistamento** (o di issue) che distribuisce le istruzioni sulle varie unità di operate, in modo che tutte le varie unità lavorino il più possibile continuamente ⇒ le istruzioni non vengono eseguite nell'ordine in cui compaiono nel codice, ma in parallelo ⇒ la **coda di completamento** usa degli algoritmi che riordinano i risultati e ne garantiscono la correttezza.

15.3 Processori multithread

Non si può aumentare troppo il numero di stadi in parallelo, perché i vincoli di dipendenza sarebbero troppi ⇒ si passa dall'Instruction Level Parallelism (ILP) al Thread Level Parallelism (TLP):

- fino agli anni '90, il processore adottava il **time sharing**, cioè a ogni programma in esecuzione il processore dedicava una parte del tempo, ma in ogni istante eseguiva un solo programma alla volta;
- un processore multithread esegue in parallelo istruzioni provenienti da più programmi in esecuzione ⇒ sfruttano al massimo le unità di operate.

15.4 Processori multicore

All'interno dello stesso circuito sono integrati più processori, detti **core**, su cui viene distribuito il lavoro.

Parte I

Assembler 8086

Capitolo 16

Introduzione

16.1 Struttura generale di un programma

Un programma è sempre composto da tre pseudo-istruzioni, dette **direttive**, che non sono delle istruzioni per il processore ma solo delle indicazioni specifiche dell'assemblatore:

- `.STACK` indica il contenuto del **segmento** (= parte di memoria) stack;
- `.DATA` indica il contenuto del segmento per le variabili, dichiarate con `<nome_simbolico> <tipo> <valore_di_inizializzazione|?>`;

16.2 Istruzioni

Le variabili simboliche evitano di richiedere l'uso diretto degli indirizzi.

- `MOV` scrive un valore in memoria: `MOV <cella_di_destinazione> <valore>`
- `ADD` somma due valori e memorizza il risultato all'interno della cella di memoria del primo operando: `ADD <primo_operando> <secondo_operando>`

16.2.1 Input/output

Il sistema operativo offre delle procedure di tipo **driver** che aiutano le applicazioni a comunicare con le periferiche. I driver si preoccupano di compiere direttamente le operazioni di input/output, conoscendo gli indirizzi di periferica.

L'istruzione `INT 21h` (interrupt) simula l'input di una periferica, interrompendo il programma in esecuzione e attivando una procedura driver, la quale recupera il valore contenuto nel registro AH:

- `1`: non appena viene digitato un carattere sulla tastiera esso viene memorizzato in AL in codifica ASCII;
- `2`: su schermo viene visualizzato il contenuto di DL.

I numeri devono sempre essere convertiti in codifica ASCII.

Capitolo 17

Informazioni generali

17.1 Modi di indirizzamento

Esistono 7 **modi di indirizzamento** che definiscono come gli operandi possono essere passati. L'assembler 8086 non è un **linguaggio ortogonale**, ma impone dei vincoli sui modi di indirizzamento nei confronti delle istruzioni: ad esempio, l'istruzione `<MOV [DI] [SI]` è vietata perché gli operandi `[DI]` e `[SI]` si trovano entrambi in memoria.

Register addressing (`AX, BX`) Gli operandi sono **registri** che contengono dati.

Immediate addressing (`7, 07h, OFFSET VAR, LENGTH VETT`) Gli operandi sono delle **costanti**, cioè sono memorizzati direttamente all'interno del codice macchina dell'istruzione ⇒ il valore massimo è limitato dalla dimensione dell'istruzione. Parole chiave come `OFFSET` e `LENGTH` non fanno parte dell'instruction set del processore; gli operandi che le contengono vengono automaticamente convertiti dall'assemblatore in indirizzi costanti.

Direct addressing (`VAR, TABLE+2, TABLE[2]`) Gli operandi sono **offset di memoria**, passati indirettamente tramite i nomi simbolici delle variabili: è compito dell'assemblatore in fase di compilazione, dopo aver assegnato a ogni variabile il proprio offset rispetto al principio del segmento dati[1], convertire il nome simbolico nel suo offset.

Register indirect addressing (`[BX], [DI], [SI]`) Gli operandi sono **offset di memoria**, passati indirettamente tramite nomi di registri racchiusi tra parentesi quadre. Incrementando di volta in volta il contenuto di un registro, è possibile scandire tutta la memoria.

Base relative addressing (`[BX]+4, 4[BX], [BX+4]`) Gli operandi sono dati dalla somma tra una costante e un valore recuperato al tempo di esecuzione.

Directed indexed addressing (`VETT[DI]`) Gli operandi sono variabili il cui offset viene sommato, al tempo di esecuzione, all'indice. È molto usato per la scansione di vettori.

Base indexed addressing (`TABLE[DI][BX]+6`) Gli operandi sono variabili il cui offset viene opportunamente sommato a due indici e facoltativamente a una costante.

Gli indici non funzionano come in C, ma si deve specificare l'offset di un elemento tenendo conto dell'effettivo numero di bit di cui sono composti gli operandi precedenti: `<riga>*<numero_colonne>*<lunghezza_word>+<colonna>*<lunghezza_word>`

[1] Il sistema operativo deciderà in fase di esecuzione dove collocare il segmento dati all'interno della memoria.

17.2 Pseudo-istruzioni

Le **pseudo-istruzioni** non si trasformano in codice macchina che verrà eseguito dal processore, ma sono solo delle direttive per l'assemblatore.

Definizione di variabili (`DB`, `DW`, `DD`) Per definire un vettore si può utilizzare l'istruzione `DUP`:

<nome> <tipo> <lunghezza_in_byte> DUP <valore_di_inizializzazione|?>

`DB` può essere utilizzato anche per memorizzare caratteri, dichiarati tra virgolette. Non esiste il carattere di terminazione `'\0'`.

Definizione di costanti (`EQU`)

Gestione dei segmenti (`.MODEL`, `.DATA`, `.CODE`, `.STACK`) A seconda delle esigenze di memoria, è possibile specificare all'assemblatore quale modello di memoria (es. `small`) richiede il programma, cioè il numero di segmenti di memoria e la loro dimensione.

Direttiva `END`

17.3 Operatori

Calcolo degli attributi di una variabile (`OFFSET`, `LENGTH`)

Costruzione di costanti (`+`, `AND`)

Modifica del tipo di una variabile (`PTR`) Anteponendo `<tipo> PTR` al nome di una variabile, si specifica la lunghezza di un operando da leggere in memoria, per rimuovere le ambiguità (è l'analogo del cast nel C).

Capitolo 18

Istruzioni di trasferimento dati

Il progettista di un processore può seguire due filosofie:

- RISC: il processore è dotato del numero minimo di istruzioni indispensabili ⇒ più semplice e più veloce;[1]
- CISC: il processore offre delle ulteriori istruzioni (es. **XCHG**) che potrebbero essere sostituite con altre istruzioni più elementari, ma sono comode per il programmatore ⇒ più commerciale.

Alcune istruzioni di trasferimento dati:

- `MOV`: ha alcune limitazioni: per esempio, non si può trasferire un dato direttamente da una cella di memoria all'altra o da un segment register all'altro, ma bisogna passare per i registri;
- `XCHG`: scambia i contenuti dei due operandi;
- `LEA <dest>, <sorg>`: copia l'offset di `<sorg>` in `<dest>`; è equivalente a:

  MOV <dest>, **OFFSET** <sorg>

- `IN` e `OUT`: specificano che l'indirizzo a cui si vuole accedere è un registro di periferica, impostando il segnale IO/M su IO.

[1] Si veda la sezione 15.1.

Capitolo 19

Istruzioni aritmetiche

Le operazioni e i formati supportati in campo numerico variano a seconda del processore. L'8086 supporta istruzioni per il calcolo binario (in complemento a 2) di numeri interi (non in virgola mobile) su 8, 16 ed in parte anche su 32 bit con segno e senza segno, e istruzioni per il calcolo tra numeri in formato BCD.

19.1 Istruzioni `ADD` e `SUB`

Le istruzioni `ADD` e `SUB` supportano numeri su 8 (byte) o 16 bit (word), a patto che gli operandi abbiano uguale lunghezza. Il primo operando oltre a essere in ingresso è anche in uscita. Gli operandi non possono essere entrambi locazioni di memoria. Queste istruzioni impostano molteplici flag che possono poi essere acceduti.

Le istruzioni `ADD` e `SUB` non fanno distinzione se i due operandi in complemento a 2 sono con segno o senza segno.

19.2 Istruzione `CBW`

L'istruzione `CBW`, priva di operandi, estende il contenuto dal registro AL su 8 bit all'intero registro AX su 16 bit. L'operazione di estensione è differente a seconda che si tratti di un numero con segno o senza segno.

19.3 Istruzione `CWD`

L'istruzione `CWD` funziona in modo analogo all'istruzione `CBW`, ma passa da 16 a 32 bit, cioè da AX a DX:AX estendendo il segno.

19.4 Istruzione `ADC`

L'istruzione `ADC` permette di sommare due operandi su 32 o 64 bit, tenendo conto del riporto lasciato nel carry flag CF dalle istruzioni `ADD` applicate in precedenza sulle word meno significative.

19.5 Istruzione `SBB`

L'istruzione `SBB` funziona in maniera analoga a `ADC`, tenendo conto del carry flag CF lasciato dalle istruzioni `SUB`.

19.6 Istruzione `INC` e `DEC`

Le istruzioni `INC` e `DEC`, di tipo CISC, incrementano o decrementano l'operando di un'unità; aggiornano tutti i flag di stato tranne il carry flag CF.

19.7 Istruzione `NEG`

L'istruzione `NEG` cambia segno all'operando (in complemento a 2).

19.8 Istruzione `MUL` e `IMUL`

Le istruzioni `MUL` e `IMUL` effettuano la moltiplicazione di numeri rispettivamente senza segno e con segno. L'unico operando viene moltiplicato per il contenuto di AL o di AX a seconda se l'operando è su 8 o su 16 bit, e il risultato viene memorizzato rispettivamente in AX e in DX:AX.

I flag CF e OF cambiano il loro significato: se sono entrambi 0, la parte alta del risultato è nulla.

19.9 Istruzione `DIV` e `IDIV`

Le istruzioni `DIV` e `IDIV`, aventi un unico operando, effettuano la divisione:

- operando su 8 bit: il contenuto del registro AX viene diviso per l'operando, e vengono memorizzati il quoziente in AL e il resto in AH;
- operando su 16 bit: il contenuto del registro DX:AX viene diviso per l'operando, e vengono memorizzati il quoziente in AX e il resto in DX.

Se si cerca di effettuare una **divisione per zero** (cioè il divisore è troppo piccolo e genererebbe un quoziente troppo grande rispetto alle dimensioni del registro di destinazione), viene generato un errore e viene interrotto il programma.

Capitolo 20

Istruzioni per il controllo del flusso

20.1 Istruzioni di salto

Le **istruzioni di salto** modificano il valore del PC in modo che il fetch successivo venga effettuato in un'altra posizione di memoria.

Si distinguono istruzioni di salto condizionato (es. `JNE`) e di salto incondizionato (es. `JMP`).

20.1.1 Tipi di salto

Short (2 byte)

Le istruzioni dei processori destinati ad applicazioni a basso costo sono progettate per avere un codice macchina il meno lungo possibile, in modo che siano sufficienti memorie più piccole ed economiche. In particolare, le istruzioni di salto condizionato sono di tipo **short**, cioè sono ottimizzate per i salti che vengono effettuati in una destinazione vicina alla stessa istruzione: bastano infatti 2 byte complessivi, di cui il secondo byte contiene l'offset dell'istruzione a cui saltare non relativo all'inizio del segmento di memoria, ma relativo alla stessa istruzione di salto.

Near (3 byte)

Le istruzioni di salto di tipo **near** richiedono 3 byte: il primo contiene il codice operativo dell'istruzione, gli altri due l'offset della procedura a cui saltare.

Far (5 byte)

Le istruzioni di salto di tipo **far** (5 byte) possono saltare da un segmento di memoria all'altro, modificando oltre all'IP anche il registro di segmento.

20.1.2 Istruzioni di salto incondizionato

L'istruzione `JMP` è un'istruzione di salto incondizionato di tipo near.

20.1.3 Istruzioni di salto condizionato

Per ogni flag esistono due **istruzioni di salto condizionato**, che eseguono il salto una se il flag vale 1, l'altra se vale 0.

Confronto tra numeri con segno	Confronto tra numeri senza segno	Salta se
JL o JNGE	JB o JNAE	destinazione $<$ sorgente
JG o JNLE	JA o JNBE	destinazione $>$ sorgente
JLE o JNG	JBE o JNA	destinazione $\leq$ sorgente
JGE o JNL	JAE o JNB	destinazione $\geq$ sorgente
JE	JE	destinazione $=$ sorgente
JNE	JNE	destinazione $\neq$ sorgente

Altre istruzioni di salto condizionato (ad es. `JZ` e `JNZ`) sono in grado di testare flag come SF e ZF lasciati dall'istruzione `CMP <sorgente> <destinazione>`.

L'istruzione `JCXZ` salta se in CX è contenuto il valore 0.

Le istruzioni di salto condizionato sono di tipo short.

20.2 Istruzioni di iterazione

L'istruzione `LOOP` serve per la gestione dei cicli: decrementa di un'unità il registro CX, confronta il CX con 0, e se è diverso da 0 salta all'etichetta usata come parametro.

Il codice:

```
LOOP <etichetta>
```

è equivalente a:

```
DEC CX
CMP CX, 0
JNE <etichetta>
```

Capitolo 21

Istruzioni per la manipolazione dei bit

Le **istruzioni per la manipolazione dei bit** si suddividono in:

- **istruzioni logiche:** permettono di modificare o controllare uno o più bit;
- **istruzioni di scorrimento:** permettono di cambiare la posizione dei bit.

21.1 Istruzioni logiche

Oltre ai registri per il trasferimento dei dati tra il processore e la periferica, vi sono dei registri di controllo/stato in cui ciascun bit informa il processore secondo un parametro sul funzionamento della periferica ⇒ il processore deve poter operare su singoli bit.

Le **istruzioni logiche** (bit a bit) permettono di operare su un singolo bit di una word per volta. La **maschera** seleziona i bit su cui operare: `OR` forza a 1 i bit selezionati dalla maschera, `NOT` inverte tutti i bit, e così via.

L'istruzione `TEST` esegue l'AND bit a bit restituendo attraverso il flag ZF se un bit vale 1.

21.2 Istruzioni di scorrimento

Le **istruzioni di scorrimento** permettono di effettuare lo shift dei bit. Si suddividono in:

- **istruzioni di shift**
- **istruzioni di rotazione**

Il secondo operando `contatore` può essere un immediato o il registro CL.

21.2.1 Istruzioni di shift

Le istruzioni `SHL` e `SHR` effettuano lo scalamento puro rispettivamente a sinistra e a destra, inserendo il valore 0 nel bit liberato e salvando il bit espulso nel flag CF.

Se i numeri sono senza segno, lo scalamento coincide con la moltiplicazione/divisione per 2.

Le istruzioni `SAL` e `SAR` effettuando moltiplicazioni/divisioni per 2 tenendo conto del segno del numero in complemento a 2: l'istruzione `SAL` è equivalente alla `SHL`, ma la `SAR` (verso destra) opera sui numeri negativi inserendo 1 nel bit liberato.

21.2.2 Istruzioni di rotazione

Le istruzioni `ROL` e `ROR` reimmettono il bit espulso nel bit liberato, e lo salvano nel CF sovrascrivendo il valore precedente. Le istruzioni `RCL` e `RCR` sovrascrivono il CF dopo aver inserito lo stesso suo valore precedente nel bit liberato.

Capitolo 22

Le procedure

Una **procedura** è una porzione di codice richiamabile tramite il suo nome in qualsiasi punto del programma. Il salto alla procedura supporta il ritorno al chiamante. Migliora la leggibilità del codice e risparmia memoria.

22.1 Struttura

Una procedura si definisce in questo modo:

```
<etichetta> PROC <tipo>
...
<etichetta> ENDP
```

La procedura può essere richiamata con l'istruzione `CALL <etichetta>`. In realtà nell'istruzione `CALL` l'assemblatore converte il simbolo `<etichetta>` nell'indirizzo della prima istruzione della procedura. Alla chiamata, l'indirizzo del chiamante viene salvato dall'IP allo stack.

L'istruzione `RET`, priva di parametri, permette di ritornare alla procedura chiamante.

22.2 Località

Una procedura di tipo `NEAR` (predefinito) può essere richiamata solo all'interno dello stesso segmento di codice, e viene richiamata da una `CALL` su 3 byte tramite l'offset della sua prima istruzione. Una procedura di tipo `FAR` può essere chiamata da procedure appartenenti a segmenti di codice diversi; la `CALL` su 5 byte prima di caricare l'offset della procedura deve modificare il registro CS, effettuando il push nello stack del vecchio valore di CS.

Anche l'istruzione `RET` si distingue per i tipi `NEAR` e `FAR`, effettuando la pop dallo stack di uno o due word.

22.3 Stack[1]

22.3.1 Memorizzazione di variabili locali

La procedura può memorizzare temporaneamente nello stack le variabili dichiarate in modo locale al suo interno.

22.3.2 Backup del contenuto dei registri

Le procedure vengono richiamate tramite l'istruzione `CALL`, che salta alla prima istruzione della procedura. Al termine della procedura, l'istruzione `RET` deve poter ritornare al chiamante ⇒

[1] Si veda la sezione 4.2.2.

l'istruzione `CALL`, prima di sovrascrivere il PC con il nuovo indirizzo, deve quindi temporaneamente salvare (push) nello stack il PC del chiamante, che verrà letto (pop) dall'istruzione `RET`. La struttura LIFO garantisce che l'ordine dei PC letti da una serie di istruzioni `RET` appartenenti a procedure annidate sia corretto, cioè dall'ultima procedura chiamata fino al primo chiamante.

I valori dei registri possono essere salvati prima della prima istruzione e ripristinati dopo l'ultima istruzione, in modo che la procedura non "sporchi" i registri.

22.3.3 Passaggio di parametri

I linguaggi assembler non hanno alcun meccanismo apposito per il passaggio dei parametri.

Prima della chiamata della procedura il chiamante può salvare nello stack i parametri, che verranno letti durante l'esecuzione della procedura. Il problema è che l'istruzione `CALL` li coprirebbe con il valore dell'IP. Il registro BP serve per salvare temporaneamente il valore dell'SP subito dopo la chiamata come prima istruzione della procedura (lo stack si riempe a partire dal fondo), in modo che il primo parametro sia accessibile all'indirizzo `[BP+2]`. Un semplice ripristino dell'SP al valore originario segue il ritorno al chiamante. I parametri di ritorno possono essere salvati dalla procedura e letti dal chiamante in una posizione destinata appositamente nello stack tramite una istruzione `SUB` dell'SP che precede le push dei parametri. Non c'è limite al numero di parametri, e i parametri rimangono in memoria solo per il tempo di esecuzione della procedura. Questa soluzione non può essere applicata a procedure annidate, perché il registro BP può contenere un solo valore di SP; per ovviare a questo problema, si può salvare il contenuto di BP a sua volta nello stack subito prima di sovrascriverlo con il valore di SP, e quindi ripristinarlo con una pop subito prima della istruzione `RET`, avendo l'accortezza di accedere al primo parametro all'indirizzo `[BP+4]`.

Altri modi

Variabili globali La procedura può usare direttamente delle variabili globali, che sono visibili a tutte le procedure compresa quella chiamante.

Svantaggio non è una soluzione flessibile

Registri Lo scambio di informazioni tra la procedura e il chiamante avviene tramite i registri.

I registri che non sono restituiti al programma chiamante possono essere usati internamente alla procedura, salvando temporaneamente nello stack i loro vecchi valori.

Svantaggio il numero di registri è limitato

Capitolo 23

Istruzioni per il controllo del processore

23.1 Istruzioni per la gestione delle interruzioni

23.1.1 Interrupt esterni[1]

Tramite gli **interrupt esterni**, un dispositivo esterno (es. tastiera) può in qualunque momento richiedere l'attenzione del processore attraverso un segnale sul pin INT, in modo che il processore, verificando questo segnale al termine di ogni istruzione, possa interrompere l'esecuzione del programma e saltare al codice che serve il dispositivo esterno. La richiesta di interruzione è affiancata da un codice su 8 bit che identifica il dispositivo tra quelli elencati in memoria nella **Interrupt Vector Table** (IVT), la quale associa ogni codice di periferica all'indirizzo di partenza della procedura che serve il dispositivo, detta **Interrupt Service Routine** (ISR). L'indirizzo di partenza della ISR è composto dal nuovo valore del CS[2] (16 bit) e dal nuovo valore dell'IP (16 bit). La ISR deve sporcare il meno possibile la memoria che era in uso dal programma interrotto; inoltre, a differenza di una normale procedura, la ISR all'avvio salva temporaneamente nello stack, oltre al vecchio indirizzo di ritorno IP e al vecchio valore di CS, anche il registro di stato PSW che contiene i flag, che alla fine viene ripristinato tramite un'istruzione `IRET`. L'installazione di un nuovo dispositivo, prima di attivare gli interrupt, richiede di caricare in memoria la ISR e aggiungere il codice del nuovo dispositivo nella IVT.

23.1.2 Interrupt software

L'8086 supporta anche gli **interrupt software**, che simulano un interrupt esterno e saltano a una certa procedura identificata dal parametro di un'istruzione `INT` (es. `INT 21h`). Le procedure richiamate da interrupt software sono offerte dal sistema operativo ⇒ il programmatore non deve preoccuparsi di conoscere l'effettivo indirizzo o il nome della procedura.

23.2 Altre istruzioni

- L'istruzione `INTO` equivale ad una `INT 4` quando l'Overflow Flag (OF) è settato, altrimenti equivale ad un'istruzione nulla.
- Se il processore entra nello stato idle tramite l'istruzione `HLT`, aspetta una richiesta di interrupt senza fare nulla.

[1] Per approfondire, si veda la sezione 11.2.2.
[2] La ISR non può stare nello stesso segmento di codice del programma che è stato interrotto.

- In un sistema a più di un microprocessore, uno dei microprocessori può segnalare tramite l'istruzione `LOCK` agli altri di non occupare il bus mentre sta effettuando una serie di particolari operazioni che devono essere eseguite in modo consecutivo.
- L'istruzione `NOP` non fa nulla, e serve per:
  - introdurre un tempo di attesa di durata nota prima di eseguire l'istruzione successiva, ad esempio per rallentare un ciclo;
  - sovrascrivere delle istruzioni in eccesso senza dover ricompilare tutto il programma;
  - craccare i programmi!
- Le istruzioni `STC`, `CLC` e `CMC` operano sul Carry Flag (CF).
- Le istruzioni `STI` e `CLI` operano sull'Interrupt Flag (IF) che determina se il processore è sensibile agli interrupt.

Capitolo 24

Formato delle istruzioni macchina, tempi di esecuzione

24.1 Formato delle istruzioni macchina

In ambiente 8086 il **formato delle istruzioni macchina** è variabile, cioè le istruzioni hanno codici macchina di lunghezze diverse: al codice operativo sono riservati 1 o 2 byte, agli operandi fino a 4 byte. L'unità di controllo, dopo aver letto il primo byte dell'istruzione, deve capire la lunghezza dell'istruzione.

24.1.1 Primo byte

Oltre ad accogliere una parte del codice operativo, il **primo byte** può contenere a seconda dell'istruzione alcuni bit particolari:

- ***W*:** specifica se l'istruzione opera su byte ($W = 0$) o su word ($W = 1$);
- ***D*:** specifica l'ordine degli operandi (ad. es vale 0 se il registro è l'operando sorgente o 1 se è l'operando destinazione);
- ***S*:** specifica se l'operando immediato è rappresentato su 1 o 2 byte.

24.1.2 Secondo byte

Nelle istruzioni con due operandi, il **secondo byte** contiene i campi *MOD*, *REG* e *R/M*:

- il campo *REG* specifica un registro su 3 byte;
- il campo *MOD* specifica il significato dell'operando *R/M* (ad es. se *MOD* vale 11, anche il secondo operando è un registro che è specificato in *R/M*).

24.1.3 Esempi di istruzioni in base alla lunghezza

- 1 byte: `NOP`, `PUSH`, `POP`
- 2 byte: `MOV AX, BX`, `MOV AX, [BX]`
- 3 byte: `MOV AX, imm`
- 4 byte: `MOV AX, var`, `MOV [BX], imm`
- 6 byte: `MOV var, imm`

24.2 Tempi di esecuzione

Il linguaggio assembler viene usato quando è richiesto un tempo di esecuzione minimo in termini di colpi di clock e non è disponibile un compilatore sufficientemente efficiente in termini di ottimizzazione del tempo di esecuzione. I **tempi di esecuzione** delle istruzioni sono specificati nel manuale in base a:

- tipo di istruzione;
- posizione degli operandi: gli operandi in memoria richiedono un tempo di accesso maggiore di quello richiesto dagli operandi immediati, e oltretutto l'indirizzo potrebbe dover essere calcolato (es. `[BX]var`) o un'istruzione potrebbe richiedere un accesso sia in lettura sia in scrittura (es. `ADD var, AX`);
- allineamento degli operandi in memoria: si veda la sezione 4.2.

Le istruzioni lunghe occupano il processore per tutto il tempo di esecuzione, durante il quale le richieste di interrupt non vengono ascoltate.