)untatore viene sommato
zo del dato.

ıte alla generazione degli
: lavorano in background.
icienti come ad esempio:

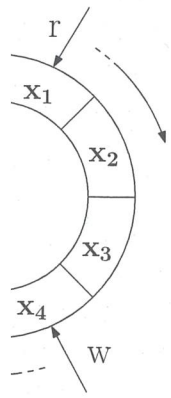
ntatore è contenuto in un

stione di vettori di dati,

lla DFT di un vettore di

one assai spedita rispet-
ıso, la struttura dati che
ıo elemento della lista è,
elemento è, automatica-
:odice, tutte le istruzioni
a e per riposizionare il

a associato agli algoritmi
restituiscono un vettore
uttavia sufficiente che il
i bit, cioè da destra verso
ccedendo al vettore delle
sciato, automaticamente



)sto in modo da consentire
ıra venga automaticamente
ıta così una struttura dati
tremo opposto.

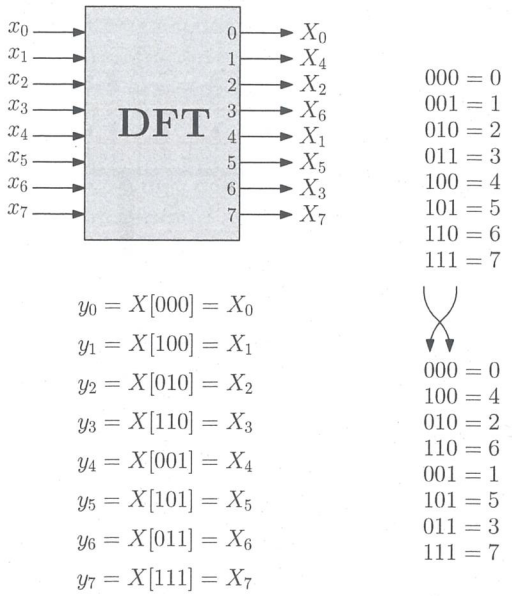


Figura 5.12: Indirizzamento a bit rovesciati: gli algoritmi DFT ottimizzati producono un vettore della trasformata X che risulta non ordinato. L'ordine corretto si può però facilmente ripristinare leggendo gli indirizzi delle componenti del vettore della trasformata a bit rovesciati.

e in modo trasparente al programmatore, si recupera direttamente l'i-esimo componente della trasformata.

Anche se sembrano particolari di secondaria importanza, questi modi di indirizzamento semplificano notevolmente il lavoro del compilatore o di chi debba realizzare il codice per le funzioni appena descritte nel linguaggio assembly della macchina. Per questo motivo, molti DSP li implementano. Come esempi si possono citare i dsPIC© della serie 33F, che li supportano entrambi, oppure i DSP della serie 563XX di NXP, che ne implementano anche altri, ancora più specifici.

5.3 Strategie di controllo

Siamo a questo punto in grado di discutere un altro componente fondamentale per il funzionamento di un processore, ovvero la sua unità di controllo. Per facilitare l'esposizione seguente, conviene fin d'ora fare riferimento alla Fig. 5.13, che mostra lo schema a blocchi di un ipotetico processore a 8 bit, che, seppure estremamente semplificato, permetterà di illustrare alcuni aspetti essenziali del funzionamento dei sistemi di controllo.

Come è possibile osservare, il processore si compone di poche unità funzionali, ciascuna governata da un limitato numero di segnali di controllo. Questi, che sono in tutto soltanto 16, sono raggruppati in un bus e impostati dall'unità di controllo durante lo svolgimento di ogni ciclo macchina. È inoltre visibile il bus a 8 bit che permette il trasferimento dei dati tra le varie unità funzionali. Alcune connessioni sono bidirezionali, altre unidirezionali, come indicato dalle frecce. Dobbiamo supporre, naturalmente, che la connessione del bus alle unità

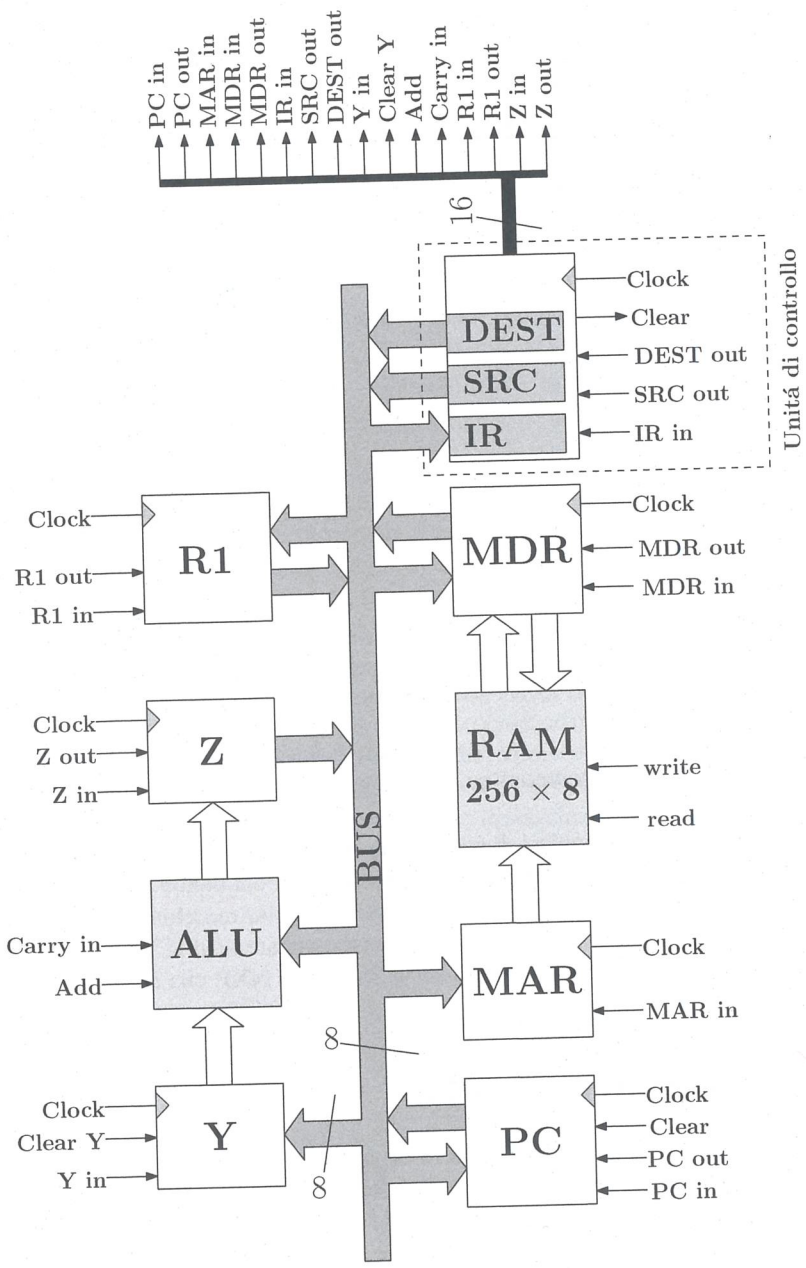


Figura 5.13: Un ipotetico processore a 8 bit.

funzionali della CPU avvenga attraverso opportune porte tri-state, come illustrato in Fig. 5.3. Le unità funzionali sono principalmente rappresentate da registri, tutti a 8 bit: Y e Z, che servono di supporto alla ALU, $R1$, che è il registro di lavoro, MAR e MDR, che gestiscono

la memoria come "Memory Address Register" e "Memory Data Register" rispettivamente, e, infine, PC, che è il "Program Counter" del processore. Quest'ultimo, come noto, quando una data istruzione del codice utente è in esecuzione, deve contenere l'indirizzo della successiva, in modo che la CPU segua *automaticamente* lo sviluppo logico del programma.

I segnali di controllo sono generati dall'ultimo blocco circuitale che è anche il più complesso. Esso si occupa della decodifica e dell'esecuzione delle istruzioni che compongono il codice dell'utente. Inoltre, predispone l'aggiornamento del PC, svolgendo la cosiddetta fase di "fetch" dell'istruzione successiva. Nella nostra ipotetica organizzazione, il circuito di controllo si avvale di tre registri, che contengono ciascuno una frazione del codice binario corrispondente all'istruzione in corso di esecuzione, cioè alla "Instruction Word", IW.

Il primo, IR, contiene il codice binario identificativo dell'istruzione, il cosiddetto "Operation Code" o più semplicemente, $OpCode$.

Il secondo, SRC, può essere usato per accogliere un parametro, ovvero un valore numerico non soggetto a variazioni durante l'esecuzione del codice, assemblato all'interno della codifica binaria dell'istruzione stessa, un accorgimento spesso usato per aumentare la compattezza e la velocità del repertorio di istruzioni. In altri repertori, il valore numerico corrisponde invece ad un indirizzo di memoria, da cui deve essere letto uno degli operandi, e che viene spesso espresso in termini relativi, ovvero come incremento da sommare al PC.

Il terzo, $DEST$, è usato, come il precedente, per accogliere un indirizzo oppure, altre volte, il codice identificativo di un registro, dove depositare il risultato di una operazione. Ancora una volta, si tratta di un accorgimento che permette di assemblare nella IW informazioni aggiuntive rispetto al semplice $OpCode$, risparmiando spazio in memoria.

Dall'analisi dell'$OpCode$, cioè dalla decodifica dell'istruzione corrente, il circuito di controllo deve ricavare la *sequenza di segnali di controllo* necessaria per ottenere due effetti: *i)* lo svolgimento della fase di "fetch" con l'aggiornamento del PC, e *ii)* l'esecuzione dell'istruzione corrente. Storicamente, si sono sviluppati due diversi approcci per realizzare questi effetti, ovvero:

1. l'approccio *microprogrammato;*
2. l'approccio *cablato.*

Tra i microcontrollori di progettazione più datata come, ad esempio, il modello $80XC196$ di Intel, che risale agli anni '70 del secolo scorso, si vedono ancora esempi di unità di controllo del tipo microprogrammato. I prodotti più recenti, specialmente se ispirati alla filosofia RISC, presentano invece controllori di tipo cablato.

Una ulteriore differenziazione tra le unità di controllo riguarda la scelta della strategia di "clocking", o temporizzazione, delle operazioni del processore. Sono possibili, ancora una volta, due soluzioni:

1. controllo su molti cicli;
2. controllo su un solo ciclo.

Nel primo caso la sequenza delle operazioni di prelievo, decodifica e esecuzione delle istruzioni occupa *più periodi* di clock.

Nel secondo, tutte le operazioni avvengono all'interno di un solo periodo di clock, che dovrà quindi essere sufficientemente lungo da consentire di gestire anche l'istruzione *più complessa* presente nel repertorio del processore.

Unità di controllo
Clock
Clear
DEST out
SRC out
IR in
Clock
MDR out
MDR in
write
read
Clock
MAR in
Clock
Clear
PC out
PC in

come illustrato in Fig. 5.3.
tutti a 8 bit: Y e Z, che
R e MDR, che gestiscono

Solitamente, la strategia su un singolo ciclo si impiega solo nei sistemi di controllo cablati, mentre esistono processori con temporizzazione multi-ciclo sia a controllo microprogrammato, dove questa è spesso anche l'unica soluzione praticabile, sia a controllo cablato, dove si pone in alternativa alla strategia a singolo ciclo.

Prima di analizzare nel dettaglio i diversi approcci, torniamo al processore di Fig. 5.13 e vediamo come una ipotetica istruzione del suo repertorio possa essere tradotta nella giusta sequenza di segnali di controllo che la realizza. Supponiamo, quindi, che, all'inizio di un ciclo macchina, il program counter del processore punti ad una locazione di memoria dove si trovi il codice binario della seguente istruzione[4]:

$$ADD \quad R1, R1, numero \tag{5.1}$$

che si immagina determini la somma del contenuto del registro $R1$ con la quantità il cui *indirizzo in memoria* è indicato dal simbolo $numero$, depositando il risultato nello stesso $R1$. Supponiamo anche che il codice della nostra istruzione sia stato progettato in modo da contenere su alcuni dei suoi bit, per esempio accanto all'$OpCode$, il valore di $numero$, che sarà quindi automaticamente reso disponibile nel registro SRC.

In una forma più astratta, ma di diretta interpretazione, l'istruzione di nostro interesse si può scrivere come:

$$R1 \leftarrow R1 + M[numero],$$

dove la scrittura $M[\cdot]$ indica l'accesso, in lettura o scrittura, ad una specifica locazione della memoria.

La prima sequenza di operazioni che il processore deve eseguire corrisponde alla lettura dalla memoria dell'istruzione il cui indirizzo è contenuto nel PC e nel *successivo incremento* del PC stesso. In termini simbolici, ciò corrisponde alle due assegnazioni seguenti:

$$IR \leftarrow M[PC];$$

$$PC \leftarrow PC + 1.$$

L'istruzione letta, nel nostro caso "ADD R1, R1, numero", verrà successivamente decodificata ed eseguita. La fase di prelievo dell'istruzione si sviluppa grazie all'attivazione dei seguenti segnali:

PC out: *l'indirizzo dell'istruzione corrente viene copiato sul bus*;

MAR in: *l'indirizzo viene trasferito sul MAR* (*e la memoria attivata in modalità* read);

Clear Y: *il registro Y viene posto a* 0;

Add: *la ALU somma Y con quanto presente sul bus* (*i.e.* $PC + 0$);

Carry in: *la ALU aggiunge* 1 *alla somma precedente*;

Z in: *il risultato* (*i.e.* $PC + 1$) *viene scritto su Z* (*che funge da accumulatore*).

Tutti questi segnali vengono asseriti dall'unità di controllo *in modo simultaneo* all'inizio di un ciclo macchina. Perché ciò sia possibile, non devono prodursi conflitti nell'utilizzo del bus o di altre risorse. Pertanto, le attività seguenti:

[4]Si suppone, evidentemente, che il codice risieda nella memoria RAM, sebbene questa condizione non si riscontri sempre nella pratica.

Z out: *il contenuto di Z è trasferito sul bus*;

PC in: *PC è ora pari al suo valore iniziale* $+1$;

MDR in: *il contenuto della locazione puntata dal vecchio valore di PC viene letto dalla memoria e, appena disponibile, salvato su MDR*;

dovranno essere eseguite nel periodo di clock *successivo*.

Se così non fosse, l'asserimento simultaneo dei segnali $PC\,out$ e $Z\,out$, ad esempio, determinerebbe un conflitto sul bus. Per lo stesso motivo, anche le attività:

MDR out: *il contenuto di MDR è trasferito sul bus*;

IR in: *il contenuto del bus* (*i.e. l'OpCode dell'istruzione corrente*) *viene salvato in* IR;

che completano la fase di fetch, devono essere eseguite in un ciclo di clock ancora successivo.

Dovrebbe risultare chiaro, a questo punto, che è la presenza di un solo bus ad imporre la *segmentazione* delle operazioni di prelievo dell'istruzione e che la stessa segmentazione sarà in generale richiesta anche in fase di esecuzione. Infatti, ogni volta che lo stesso registro è chiamato ad una operazione di lettura seguita da una di scrittura, oppure quando due diversi registri intendono trasferire un valore al bus, non è possibile asserire simultaneamente i relativi segnali di controllo, perché ciò porterebbe ad una indeterminazione dei livelli logici assunti dal bus oltre che ad un probabile danneggiamento delle porte logiche di interfaccia. Ciò significa che l'organizzazione del semplice processore di Fig. 5.13 richiede, di sicuro, una strategia di "clocking" del tipo multi-ciclo, in cui il completamento di una istruzione occupa, in generale, più periodi di clock.

Alla fase di prelievo segue, naturalmente, quella di esecuzione. Anch'essa sarà sviluppata in più fasi successive, ovvero:

SRC out: *l'indirizzo del primo addendo* (*i.e.* $numero$) *è trasferito sul bus*;

MAR in: *il contenuto del bus viene copiato nel registro MAR* (*e la memoria attivata in modalità* read);

cui segue, nel periodo di clock seguente

*R*1 *out*: *il contenuto di R*1 *è sul bus*;

Y in: *il contenuto del bus viene scritto sul registro Y* (*che serve da buffer*);

MDR in: *l'addendo M*[*numero*] *viene letto dalla memoria e copiato sul registro MDR*;

e, nel periodo di clock ancora successivo

MDR out: *il contenuto di MDR è sul bus*;

Add: *la ALU somma Y con quanto presente sul bus* (*i.e.* $R1 + M[numero]$);

Z in: *il risultato, che compare all'uscita della ALU dopo un tempo di attesa relativamente piccolo, viene copiato in Z.*

Fino a questo punto, si è ottenuto che

$$Z \leftarrow R1 + M[numero],$$

quindi, per concludere, basta introdurre un ulteriore segmento

Z out: *il contenuto di Z è trasferito sul bus*;

*R*1 *in*: *R*1 *copia il contenuto di Z dal bus*;

che corrisponde allo spostamento in $R1$ del valore contenuto in Z, ovvero a

$$R1 \leftarrow Z,$$

che rappresenta il risultato finale desiderato.

La fase di esecuzione, o "execute", della nostra istruzione è quindi stata suddivisa in 4 segmenti. In totale, abbiamo quindi avuto bisogno di 7 segmenti (3 per la fase di *fetch*, 4 per quella di *execute*) che dovranno essere posti in essere in 7 successivi cicli di clock.

Bisogna sottolineare che la numerosità dei segmenti necessari per una istruzione dipende fortemente dalla struttura circuitale del processore, in particolare dal *numero* e dalla *qualità* delle risorse a disposizione. Il nostro esempio, come conseguenza della sua intenzionale semplicità, offre prestazioni piuttosto limitate e dunque tende ad accrescere il numero di passi necessari al completamento di istruzioni anche semplici. In casi reali, la situazione è molto più favorevole e il numero di segmenti richiesti assai minore.

Ammesso che il lettore abbia avuto la pazienza di seguire lo sviluppo di questo semplice esempio, dovrebbe essere ora chiaro come una qualunque istruzione del repertorio si debba *sempre* poter scomporre in una opportuna sequenza di attivazione dei segnali di controllo del processore. È esattamente in questo aspetto che si realizza l'interazione tra la progettazione dell'architettura e quella dell'organizzazione circuitale di una CPU. Tale interazione, e le molte possibili ottimizzazioni che ne scaturiscono, rappresenta uno degli elementi più importanti, dal punto di vista ingegneristico, nella definizione di un nuovo processore.

Il problema del controllo del processore si "riduce", quindi, all'individuazione di tecniche che permettano di eseguire tali sequenze, una volta che siano state definite per una assegnata organizzazione circuitale, in modo esatto e ripetibile, per tutte le istruzioni del repertorio.

Nel seguito, verranno illustrati i due sistemi che consentono di ottenere questo risultato e quindi di automatizzare il funzionamento di un qualunque processore.

5.3.1 Controllo microprogrammato

Il controllo microprogrammato viene realizzato attraverso unità di controllo la cui struttura riproduce, di fatto, quella di una *ulteriore*, ma più semplice, CPU, con una memoria, un program counter e una semplice ALU. Questa "micro CPU" viene anche indicata come "microcode engine" o "motore del microcodice". Ogni istruzione del repertorio a disposizione del programmatore, che in questo contesto indicheremo con il termine *macro-istruzione*, corrisponde ad un microprogramma o microcodice, composto da un certo numero di parole. I microprogrammi sono contenuti in una memoria di tipo ROM, ovvero a sola lettura.

Sono possibili due organizzazioni dell'unità di controllo. La più semplice si basa sul concetto di microprogrammazione *orizzontale*. Nella microprogrammazione orizzontale, l'unità di controllo esegue il microcodice in modo rigidamente sequenziale, leggendo una microistruzione in ciascun ciclo di clock, a partire da un indirizzo della memoria ROM che è associato in modo univoco all'*OpCode* identificativo della macroistruzione corrente e fino al completamento della sua esecuzione. Ogni microcodice si apre dunque con la sequenza di microistruzioni che realizza la fase di fetch della macroistruzione *successiva*,

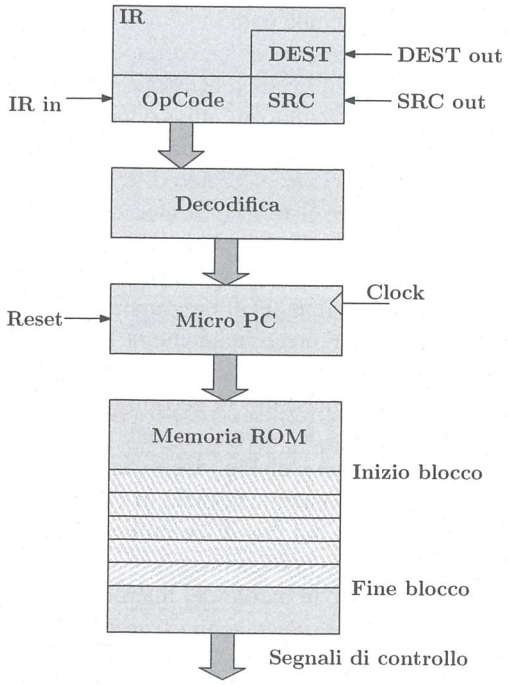


Figura 5.14: Schema a blocchi semplificato di una unità di controllo microprogrammata.

che viene letta e caricata, e si chiude con l'aggiornamento del "micro Program Counter", in modo da posizionarlo sulla locazione della memoria ROM corrispondente al nuovo *OpCode*. La microprogrammazione orizzontale richiede, però, più memoria ROM di quanto sarebbe strettamente necessario, dato che segmenti di microcodice, ad esempio quelli relativi alla fase di fetch, vengono ripetuti molte volte.

Un approccio alternativo è basato sul concetto di microprogrammazione *verticale*. In questo caso il microcodice non viene eseguito solo sequenzialmente, ma sono anche possibili spostamenti o "salti" tra regioni non contigue della memoria ROM. Questo permette di eliminare le ripetizioni di segmenti di microcodice, introducendo in pratica delle vere e proprie "micro-subroutine". D'altra parte, l'unità di controllo diviene più complessa.

La Fig. 5.14 rappresenta lo schema a blocchi di principio di una unità di controllo a microprogrammazione verticale, che potrebbe essere usata dal processore di Fig. 5.13.

Nella microprogrammazione verticale, la fase di fetch trasferisce l'istruzione *puntata* dal program counter, *non* la successiva, nell'unità di controllo e, soltanto dopo, incrementa il program counter. La lettura dell'*OpCode* determina il posizionamento del motore del microcodice sul blocco di memoria pertinente all'istruzione in carico. In altre parole, il motore del microcodice, scrivendone l'indirizzo sul micro program counter, realizza un "salto" alla locazione della memoria ROM desiderata. Da questo momento, ha inizio l'esecuzione dell'istruzione, che procede come nel caso precedente. All'inizio di ogni periodo di clock il circuito accede ad una nuova parola di microcodice, la quale, a sua volta, determina l'attivazione degli opportuni segnali di controllo. Dopo la lettura di ogni parola, il micro

program counter viene incrementato di una unità.

L'ultima parola di ogni blocco di microcodice corrisponde però a un codice "speciale", che non ha effetto sulle linee di controllo del processore, ma agisce su un segnale interno, indicato come "Reset" in Fig. 5.14. In seguito all'attivazione di tale segnale, il micro program counter si posiziona all'inizio del blocco di microcodice che sviluppa la fase di fetch, ora condiviso, dando il via ad una nuova sequenza di decodifica e esecuzione.

Questa azione, di fatto, costituisce il ritorno al blocco principale di microcodice dopo l'esecuzione di una micro subroutine. Si dovrebbe cogliere il vantaggio offerto dalla microprogrammazione verticale, che di fatto implementa sul microcodice le strutture di salto a subroutine e ritorno tipiche di tutti i linguaggi di programmazione.

Tanto nella microprogrammazione orizzontale che in quella verticale, il microcodice è composto di parole i cui bit corrispondono, ciascuno direttamente, ad un particolare segnale di controllo, così da semplificare al massimo la decodifica. La lunghezza della parola di microcodice è quindi pari al numero dei diversi segnali di controllo richiesti dalla CPU, ad esempio 16, nel caso del processore di Fig. 5.13. La presenza di un valore 1, in un certo bit della parola, attiva il corrispondente segnale di controllo, o, come si dovrebbe più correttamente dire, *asserisce* la linea del bus di controllo che ha la stessa posizione nella parola. Un valore 0, ovviamente, lascia la linea inattiva.

Ricordando l'esempio precedente, le parole del microcodice relativo ad una data istruzione, saranno complessivamente tante quanti sono i segmenti in cui sono state scomposte la fase di fetch e di esecuzione dell'istruzione corrente.

È anche possibile verificare, sempre riferendosi all'esempio precedente e considerando la Fig. 5.13, come l'esecuzione dell'istruzione "ADD R1, R1, numero" corrisponda alle seguenti 7 parole di microcodice, rappresentate in formato esadecimale su 16 bit:

$$0x6072;\ 0x9001;\ 0x0C00;\ 0x2200;\ 0x1084;\ 0x0822;\ 0x0009;$$

dove ogni bit si riferisce alla corrispondente linea del bus di controllo. Le prime tre parole costituiscono il preambolo, comune a tutte le macro-istruzioni, che realizza la fase di fetch, le successive, invece, attivano i segnali necessari per l'esecuzione dell'istruzione.

5.3.2 Controllo cablato

Nelle unità di controllo cablate il motore del microcodice è rimpiazzato da un *circuito logico combinatorio* che genera direttamente i segnali di controllo a partire dall'*OpCode* dell'istruzione corrente, gestendo anche le temporizzazioni.

Una ipotetica unità di controllo cablata è schematicamente rappresentata in Fig. 5.15. Essa comprende alcuni circuiti fondamentali. Il primo che esaminiamo è un generatore di clock *secondari*, ovvero derivati da quello principale del processore, il cui scopo è permettere la distribuzione nel tempo dell'attivazione dei segnali di controllo, evitando conflitti nell'uso del bus e delle altre risorse. Il suo funzionamento può essere illustrato, sempre in riferimento al processore di Fig. 5.13, considerando la Fig. 5.16.

Come sappiamo, il ciclo macchina del nostro processore deve essere suddiviso in 7 segmenti, ciascuno della durata di un periodo di clock. Il circuito che realizza questa funzione è piuttosto semplice, basandosi su un contatore e alcuni flip-flop di tipo D.

Il contatore riceve in ingresso il clock principale e ne conta il numero di periodi. Trascorsi 7 periodi, la sua uscita cambia stato, tipicamente assumendo il valore alto. Al successivo

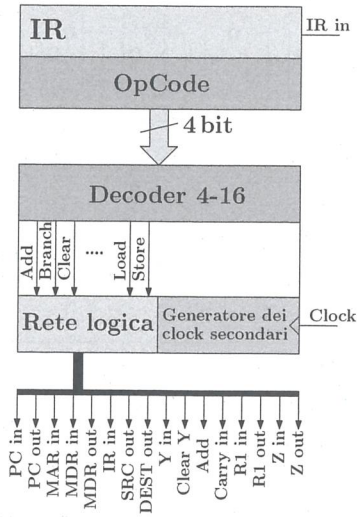


Figura 5.15: Unità di controllo cablata: è visibile un decodificatore delle istruzioni (a 4 bit), un generatore dei clock secondari e una rete logica combinatoria che asserisce i segnali di controllo del processore, elaborando sia i clock secondari che le linee del bus gestito dal decodificatore.

fronte del clock, l'uscita del contatore torna al valore basso, il conteggio riparte e inizia un nuovo ciclo macchina. Si realizza così un divisore di frequenza, che genera un impulso ogni 7 del segnale di clock principale. L'uscita del contatore fornisce il primo dei clock secondari, Clk_1. Per produrre gli ulteriori clock secondari, è poi sufficiente alimentare con il primo una catena di flip-flop di tipo D, ciascuno dei quali introduce un ritardo pari a un periodo del clock principale sul segnale di ingresso. Si ottengono così i 7 segnali che sono necessari per abilitare ciascuno dei 7 segmenti dello sviluppo di una istruzione.

Una unità di controllo cablata si avvale poi di un *circuito di decodifica* per l'identificazione delle istruzioni. Lo scopo di questo circuito è associare in modo univoco a ciascun $OpCode$ una delle linee di un bus, la cui dimensione è pari al numero di istruzioni presenti nel repertorio. Si tratta di un circuito logico elementare, detto anche *demultiplexer*, di cui, solo per completezza, viene ricordato uno schema di principio in Fig. 5.17.

È chiaro che l'uso di un demultiplexer offre la possibilità di usare un numero relativamente basso di bit per rappresentare le istruzioni, lasciando più spazio nella IW per la codifica di parametri, operandi, indirizzi, ... Con riferimento al controllo microprogrammato, questo circuito sostituisce il caricamento nel micro program counter dell'indirizzo di inizio del blocco di microcodice dedicato all'istruzione.

Il cuore dell'unità di controllo è la rete logica, puramente *combinatoria*, che gestisce le linee di controllo del processore. Una parte di tale rete, compatibile con il processore di Fig. 5.13 e quindi, a dispetto dell'apparenza, molto semplice, è visibile in Fig. 5.18.

Si osserva come il circuito consista in una matrice combinatoria con un vettore di porte AND e uno di porte OR, non necessariamente completi, in cascata. Ogni segnale di controllo è connesso, attraverso una porta AND a due ingressi, alla linea di *ciascuna* istruzione che ne richieda l'attivazione. L'altro ingresso della porta è connesso alla linea di clock secondario corrispondente al segmento del ciclo macchina durante il quale si vuole che il segnale sia

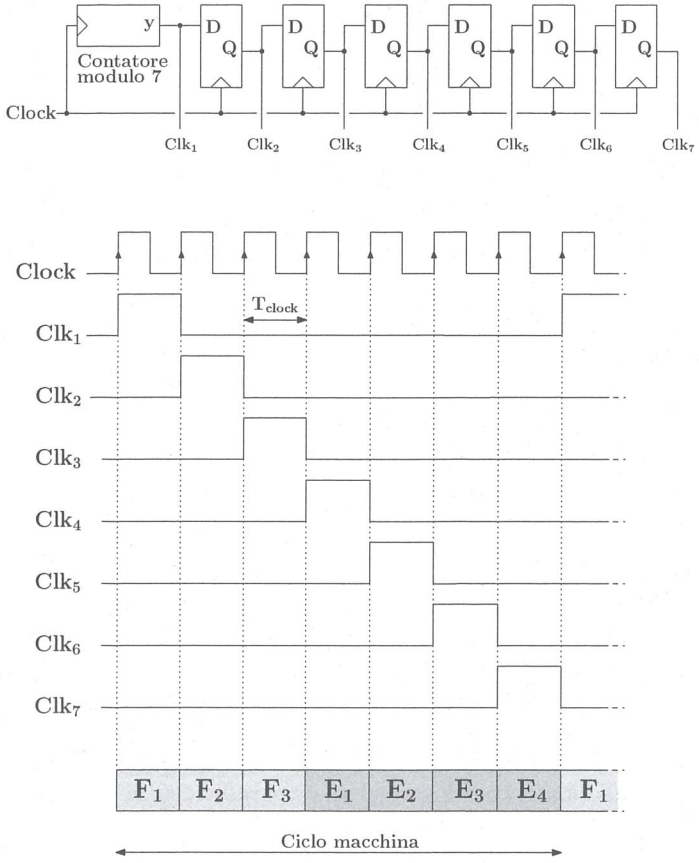


Figura 5.16: Circuito per la generazione dei segnali di clock secondari richiesti da una unità di controllo cablata.

attivo. Il risultato di questa connessione è che il segnale si attiverà ogni volta che una data istruzione lo richiede, ma solo nel segmento di ciclo macchina previsto dallo sviluppo di quella istruzione.

Si noti che, per mantenere un minimo di leggibilità, nella costruzione della Fig. 5.18 si sono volutamente trascurati alcuni segnali di controllo e alcune istruzioni di repertorio indicate nella Fig. 5.15.

Armandosi, ancora una volta, di una buona dose di pazienza, è possibile verificare *direttamente sullo schema* il funzionamento della rete, per esempio in corrispondenza dell'istruzione ipotetica che abbiamo già considerato, ovvero "ADD R1, R1, numero".

Immaginando, ad esempio, che sia attivo il clock secondario relativo alla prima parte della fase di fetch, ovvero Clk_1, si vede che saranno direttamente asseriti i seguenti segnali: $PC\,out$, $MAR\,in$, Add, $Clear\,Y$, $Carry\,in$ e $Z\,in$. Come è facile verificare, questi sono tutti e soli i segnali necessari per questa fase, secondo l'espansione dell'istruzione discussa in precedenza.

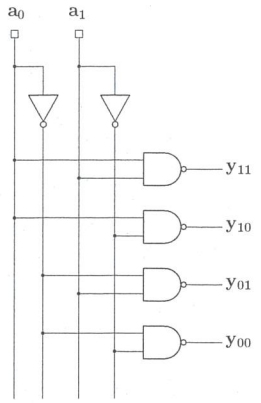


Figura 5.17: Schema di principio di un decodificatore binario, o demultiplexer, a 2 bit. Ciascuna delle 4 combinazioni di ingresso attiva, portandola, in questo caso, a livello logico *basso*, una e una sola delle 4 uscite.

Allo stesso modo, durante il periodo in cui è attivo Clk_2, saranno asseriti i segnali $PC\,in$, $MDR in$ e $Z\,out$, e così via, procedendo attraverso tutti i segmenti, fino al completamento dell'istruzione.

Dovrebbe essere ora chiaro il principale vantaggio di un'unità di controllo cablata rispetto ad una microprogrammata. La prima ha tempi di risposta *molto bassi*, in quanto i segnali di controllo vengono asseriti agendo su una rete combinatoria, che presenta solo due array di porte in cascata. In confronto, un controllo microprogrammato deve accedere ad una memoria, leggere una parola di microcodice e asserire le linee del bus di controllo i cui bit di stato sono pari a 1. I tempi di risposta che ne derivano sono decisamente superiori. Oltre a questo, l'occupazione di area su silicio da parte di un controllore cablato è *nettamente minore* rispetto a quella di un motore per microcodice.

Appare però evidente anche il principale svantaggio del controllo cablato, ovvero la totale *assenza di flessibilità*. Una volta che la rete logica è predisposta, modificarla, per esempio per aggiungere al repertorio una nuova istruzione, richiede una sua completa riprogettazione e la realizzazione di un nuovo chip. Nel caso si abbia invece a che fare con un motore per microcodice, l'aggiunta di una istruzione, come anche la correzione di un errore di progetto delle sequenze di attivazione dei segnali di controllo, comporta appena una riprogrammazione della memoria ROM.

Un ulteriore svantaggio è rappresentato dal fatto che, nel caso una istruzione non richieda tutti i periodi di clock del ciclo macchina, può essere piuttosto complesso evitare che il processore rimanga *inerte* durante i cicli inutilizzati. La temporizzazione, essendo cablata nel circuito, è infatti anch'essa rigida. Invece, un motore a microcodice può agevolmente saltare una fase, dato che ogni istruzione può avere la sua espansione realizzata nel numero minimo di parole di microcodice. Non c'è alcun obbligo, infatti, che i segmenti di microcodice abbiano tutti la stessa lunghezza.

A questo punto, sembrerebbe che il controllo cablato sia difficilmente preferibile a quello microprogrammato. In effetti, ciò è stato vero in passato, agli albori della tecnologia digitale, quando la progettazione dei circuiti logici era effettuata in modo manuale e, quindi, la probabilità di commettere un qualche errore non appariva affatto trascurabile. Oggigiorno,

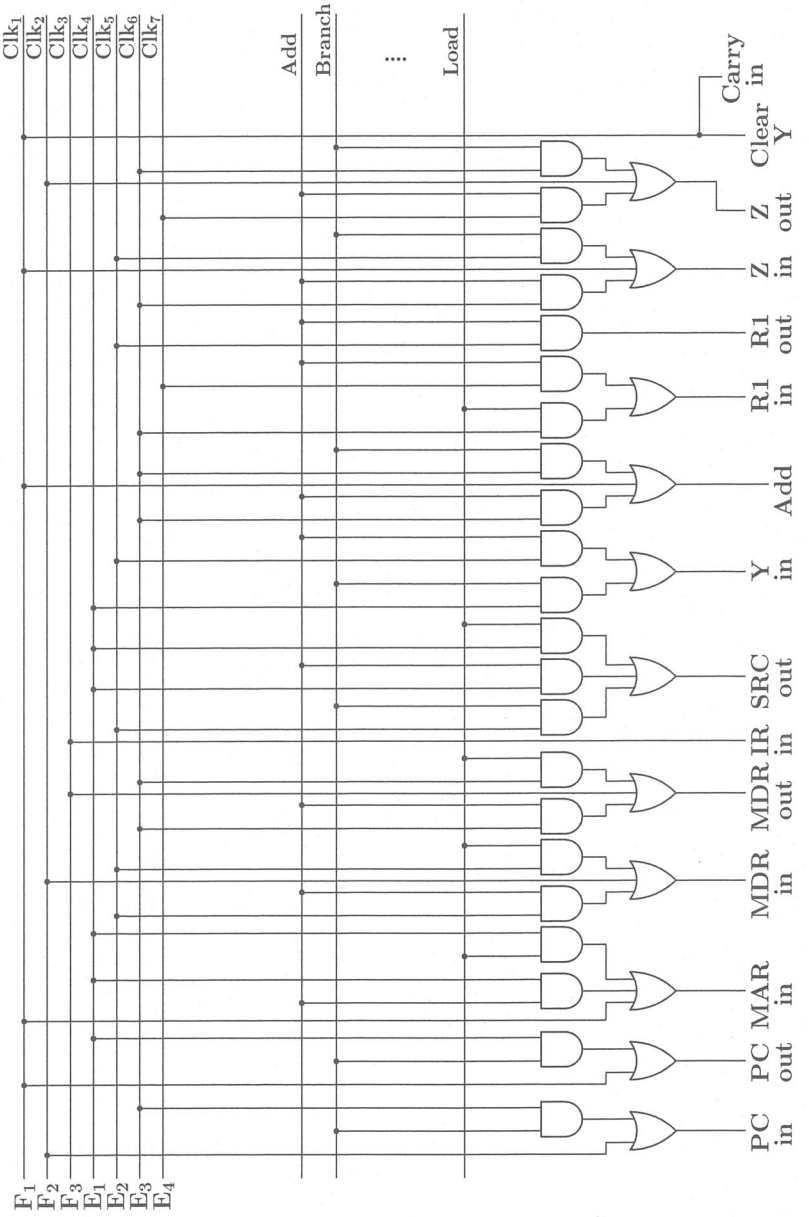


Figura 5.18: Una parte della rete combinatoria richiesta dal controllo del processore di Fig. 5.13.

la progettazione di circuiti digitali VLSI avviene mediante strumenti CAD estremamente sofisticati, capaci non solo di verificare la correttezza logica di una rete, ma anche di testarne la *robustezza* rispetto alle inevitabili, per quanto minime, imprecisioni del processo di realizzazione. Ciò permette di progettare circuiti estremamente complessi, come i moderni microprocessori, mantenendo la probabilità di malfunzionamento del prodotto finito su valori

veramente molto bassi. In questo contesto, la superiore velocità dei sistemi di controllo cablati e la loro minore occupazione di area sono divenuti fattori determinanti nelle scelte di progetto. Il risultato finale è che *tutti* i processori di recente progettazione adottano controllori cablati.

5.3.3 Organizzazioni a ciclo singolo

Come precedentemente accennato, solo la disponibilità di risorse multiple può rendere possibile una strategia di temporizzazione su un ciclo singolo. In base a quanto visto finora, risulta infatti chiaro che una organizzazione a risorse condivise, e.g. con un solo bus, una sola memoria per dati e programmi, etc. non può operare in un solo ciclo.

L'esecuzione delle istruzioni in un solo ciclo diventa invece possibile quando si passa ad organizzazioni più complesse. Questa richiede, come minimo, che la fase di fetch dell'istruzione possa essere svolta "simultaneamente" alla sua decodifica ed esecuzione. Possiamo, in effetti, definire esattamente i requisiti *minimi* necessari affinché una CPU possa funzionare con una temporizzazione a ciclo singolo. Essi sono i seguenti:

1. le memorie dati e istruzioni, con i relativi bus, devono essere *separate*;
2. deve essere disponibile una ALU separata per l'incremento del PC;
3. la gestione degli incrementi del PC deve essere flessibile, in modo da permettere l'esecuzione di istruzioni di salto senza coinvolgere la ALU principale della CPU.

Consideriamo, a titolo di esempio, che un processore, con tutte le caratteristiche appena indicate, debba eseguire in un solo ciclo la seguente istruzione:

$$ADD \quad somma, R1, R2$$

il cui significato, prevedibilmente, è

$$M[somma] \leftarrow R1 + R2.$$

Le operazioni da svolgere sono le seguenti:

1. prelievo dell'istruzione e incremento del PC;
2. trasferimento dei valori presenti in $R1$ e $R2$ nella ALU;
3. esecuzione della somma;
4. scrittura del risultato su $M[somma]$.

È importante osservare che, nei processori a ciclo singolo, le CPU sono spesso del tipo "a molti registri" e dunque le ALU hanno molte più possibilità operative rispetto a quella del processore di Fig. 5.13.

Una tipica organizzazione è illustrata, schematicamente, in Fig. 5.19. La fase di fetch può essere svolta nello stesso ciclo di clock della fase di esecuzione perché, per ipotesi, la memoria istruzioni e la memoria dati sono separate. Quindi, le operazioni:

$$MAR_I \leftarrow PC,$$

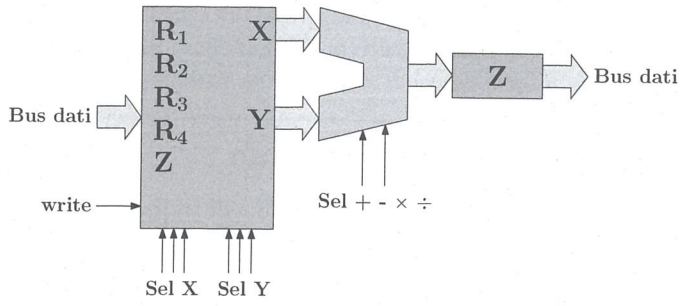


Figura 5.19: ALU di una CPU con organizzazione a molti registri. I comandi di selezione determinano quali registri fungono da operandi, X e Y, e quale operazione aritmetica viene eseguita, sia essa la somma, la differenza, il prodotto o, in qualche caso, addirittura la divisione.

che usa il bus *indirizzi* della memoria *riservata al codice*, e

$$IR \leftarrow MDR_I,$$

che usa il bus *dati* della memoria *riservata al codice*, non creano conflitti con l'elaborazione che si serve, eventualmente, del bus *riservato ai dati*. Inoltre, l'operazione

$$PC \leftarrow PC + 1,$$

che applica a PC l'incremento predefinito, è svolta da una ALU *distinta* da quella riservata ai dati. Mentre il controllore incrementa il PC, è già possibile che l'istruzione venga decodificata e siano svolte le operazioni che la realizzano, ovvero, riferendosi alla Fig. 5.19:

$$Z \leftarrow R1 + R2,$$

$$MAR_D \leftarrow somma,$$

usando il bus *indirizzi* della memoria *riservata agli operandi*, e

$$MDR_D \leftarrow Z,$$

usando il bus *dati* della memoria *riservata agli operandi*. È importante ricordare che l'esecuzione delle operazioni non è sequenziale, ma parallela, ovvero simultanea. Sarà comunque necessario attendere un certo tempo prima che tutti i segnali si assestino.

A questo punto è bene osservare che, a meno che non si abbia a che fare con repertori di istruzioni molto semplici, come quelli di alcuni processori sviluppati per una architettura RISC davvero *rigida*, l'organizzazione a singolo ciclo tende, paradossalmente, ad essere piuttosto *inefficiente*. Infatti, è l'istruzione più onerosa, in termini di tempo di esecuzione, a determinare il minimo periodo di clock possibile. Questo penalizza l'esecuzione delle istruzioni più veloci, che, sebbene vengano eseguite in una frazione di tale periodo, lasciano poi il processore *inutilizzato* per il resto del tempo.

Invece, nelle organizzazioni a molti cicli, è il tempo di risposta dell'unità funzionale più lenta, sia essa la ALU, i registri o, più verosimilmente, la memoria, a determinare il minimo periodo di clock possibile, che sarà certamente più breve di quello necessario per il funzionamento a ciclo singolo. Questo, insieme al fatto che non tutte le istruzioni richiedono lo stesso numero di periodi di clock per essere decodificate ed eseguite, consente, di norma, un migliore sfruttamento del processore. È comunque possibile combinare i vantaggi dei due approcci in organizzazioni più complesse, quelle basate su *pipeline*, di cui parleremo estesamente nel prossimo capitolo.

La *potenziale* inefficienza delle organizzazioni a ciclo singolo può essere resa evidente grazie ad un semplice esempio, ancora una volta brillantemente concepito dagli autori di [1]. Consideriamo un processore che abbia i seguenti tempi di risposta:

- memoria: 2 ns;
- ALU: 2 ns;
- registri: 1 ns.

Consideriamo, inoltre, un programma di test che sia composto da istruzioni di vario tipo, nelle proporzioni indicate:

- letture da memoria (load): 24%
- scritture in memoria (store): 12%
- operazioni (ALU su registri): 44%
- salti (tipo branch): 20%

Una implementazione del controllo a singolo ciclo deve permettere comunque l'esecuzione dell'istruzione più lenta. Questa è, di solito, la lettura da memoria, che appare in istruzioni del tipo "load R1, numero", e che richiede le seguenti attività:

1. fetch dell'istruzione: e.g. 2 ns;
2. eventuale operazione ALU (offset sull'indirizzo): e.g. 2 ns;
3. scrittura sul registro MAR: e.g. 1 ns;
4. accesso alla memoria dati: e.g. 2 ns;
5. scrittura sul registro: e.g. 1 ns.

Una implementazione del controllo di tipo multiciclo è invece limitata nel periodo di clock solo dalla più lenta delle unità funzionali, ossia dall'accesso alla memoria.

Immaginiamo ora che la versione multiciclo del controllo richieda la suddivisione del ciclo macchina in 4 segmenti[5], ciascuno di durata pari a un periodo di clock ovvero, visti i tempi di risposta, almeno pari a 2 ns.

[5]La fattibilità di questa segmentazione dipende dall'organizzazione circuitale del processore, che assumiamo essere adeguata.

Nelle due implementazioni, l'istruzione "load" avrà allora la stessa durata, cioè 8 ns. Le istruzioni più semplici, e.g. i salti, potranno però avere durata minore nel caso multiciclo rispetto al caso a ciclo singolo.

Supponendo quindi che le istruzioni abbiano la seguente durata, espressa sia in termini di numero di segmenti del ciclo macchina che di tempo:

- load: 4 segmenti, 8 ns;
- store: 4 segmenti, 8 ns;
- operazioni ALU: 3 segmenti, 6 ns;
- salti: 2 segmenti, 4 ns;

il programma di test richiederà un numero di periodi di clock per istruzione mediamente pari a:

$$4 \cdot (0.24 + 0.12) + 3 \cdot 0.44 + 2 \cdot 0.2 = 3.16$$

quindi ben minore di 4, valore corrispondente al periodo di clock richiesto dall'organizzazione a ciclo singolo. Rispetto all'organizzazione a ciclo singolo, quindi, l'organizzazione multiciclo garantisce un risparmio di tempo del 21% sul programma di test.