4.5 Un'implementazione multi-ciclo

Implementazione multi-ciclo: chiamata anche *implementazione a più cicli*, è un'implementazione nella quale un'istruzione è eseguita in cicli di clock multipli.

Negli esempi precedenti ciascuna istruzione è stata decomposta in una serie di passi corrispondenti alle operazioni richieste dalle diverse unità funzionali: tali passi possono essere impiegati per creare un'**implementazione a più cicli** (o **multi-ciclo**), dove ciascun passo richiede un ciclo di clock. L'implementazione multi-ciclo permette all'unità funzionale di essere utilizzata più di una volta per istruzione, purché in cicli di clock diversi. La condivisione delle unità funzionali permette di ridurre la quantità di hardware richiesto. La possibilità di avere istruzioni con un diverso numero di cicli di clock e la possibilità di possedere in comune unità funzionali per la stessa istruzione sono i vantaggi principali dei progetti multi-ciclo. La **Figura e4.5.1** mostra una versione astratta di un'unità operativa multi-ciclo. Se confrontiamo la Figura e4.5.1 con l'unità operativa a ciclo singolo della Figura 4.15, possiamo osservare le seguenti differenze:

- è sufficiente una sola unità di memoria per istruzioni e dati;
- è presente una sola ALU, anziché una ALU e due sommatori;
- uno o più registri sono aggiunti a ciascuna unità funzionale per memorizzarne l'uscita dell'unità fino a quando, in un ciclo di clock successivo, il valore verrà utilizzato.

Al termine di ogni ciclo di clock, tutti i dati utilizzati nei cicli di clock successivi dovranno essere memorizzati in un elemento di stato. I dati utilizzati da istruzioni successive sono memorizzati negli elementi di stato visibili al programmatore (ossia il register file, il PC o la memoria). Al contrario, i dati utilizzati nella stessa istruzione ma in un ciclo di clock successivo sono memorizzati in uno dei registri addizionali.

La posizione dei registri supplementari è quindi determinata da due fattori: quali unità funzionali riescono a lavorare in un solo ciclo di clock e quali dati saranno richiesti nei cicli successivi per implementare le istruzioni. Nel progetto multi-ciclo si assumerà che il ciclo di clock possa contenere al più una delle seguenti operazioni: accesso in memoria, accesso al register file (due letture o una scrittura), operazione della ALU; tutti i dati prodotti da una di queste tre unità funzionali (memoria, register file, ALU) dovranno essere salvati in un registro temporaneo per essere utilizzati nei cicli di clock seguenti.

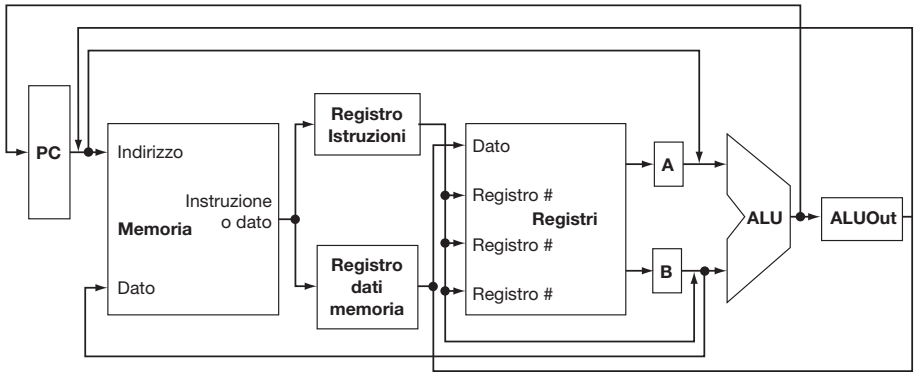

Figura e4.5.1 Vista ad alto livello dell'unità di elaborazione multi-ciclo. La figura mostra gli elementi principali dell'unità di elaborazione: un'unità di memoria condivisa, una sola ALU condivisa dalle istruzioni, le connessioni tra tali unità condivise. L'adozione di unità funzionali condivise richiede l'aggiunta o l'espansione dei multiplexer e l'utilizzo di nuovi registri temporanei che memorizzino i dati tra i cicli di clock della stessa istruzione: i registri aggiuntivi sono l'*Instruction Register* (IR, registro istruzioni), il *Memory Data Register* (MDR, letteralmente: registro dati della memoria), A, B e ALUOut.

Se essi non fossero salvati, allora la possibilità di utilizzare valori non corretti sarebbe molto probabile. Per soddisfare tali requisiti sono quindi stati aggiunti i seguenti registri:

- il registro istruzioni (o *Instruction Register*, IR) e il registro dei dati della memoria (o *Memory Data Register*, MDR) memorizzano l'output della memoria rispettivamente durante le operazioni di lettura di un'istruzione e di un dato; sono utilizzati due registri separati in quanto, come vedremo tra breve, i due valori saranno necessari nello stesso ciclo di clock;
- i registri A e B memorizzano i valori dei registri letti dal register file;
- il registro ALUOut memorizza l'uscita della ALU.

Tutti i registri, fatta eccezione per IR, memorizzano i dati solamente tra un ciclo di clock e quello immediatamente successivo e pertanto non richiedono un segnale di controllo della scrittura. Il registro IR, invece, deve memorizzare l'istruzione fino alla fine della sua esecuzione, richiedendo quindi un segnale di controllo per la scrittura. Questa distinzione sarà più chiara quando si esamineranno i diversi cicli di clock per le varie istruzioni.

Poiché alcune unità funzionali sono state condivise da differenti funzioni, sarà necessario aggiungere dei multiplexer e, in alcuni casi, espandere quelli esistenti. Per esempio, poiché si utilizza una sola memoria per istruzioni e dati, è necessario un multiplexer per selezionare l'origine degli indirizzi della memoria tra le due sorgenti possibili: PC (per leggere istruzioni) e ALUOut (per i dati).

Avendo rimpiazzato le tre ALU dell'unità di elaborazione a singolo ciclo con una sola ALU, quest'ultima dovrà ricevere tutti gli ingressi che erano indirizzati alle diverse unità. Per trattare questi ingressi addizionali, l'unità di elaborazione è stata modificata come segue:

1. Si è aggiunto un nuovo multiplexer sul primo ingresso della ALU, che seleziona il registro A oppure PC;
2. Il multiplexer a due vie presente sul secondo ingresso della ALU è sostituito da uno a quattro vie. I due ingressi aggiuntivi sono la costante 4 (per incrementare il PC) e il valore del campo offset, esteso nel segno e fatto scorrere (utilizzato per il calcolo dell'indirizzo di salto).

La **Figura e.4.5.2** mostra i dettagli dell'unità di elaborazione completa di questi multiplexer aggiuntivi: aggiungendo pochi registri e multiplexer si è potuto ridurre il numero di unità di memoria da due a una, ed eliminare due sommatori. Essendo i registri e i multiplexer relativamente piccoli, la modifica può ridurre in modo sostanziale il costo dell'hardware.

Poiché l'unità di elaborazione mostrata in Figura e4.5.2 richiede diversi cicli di clock per istruzione, i segnali di controllo saranno necessariamente differenti. Gli elementi di stato visibili al programmatore (il Program Counter, la memoria e i registri) e il registro IR necessitano di segnali di controllo per la scrittura. La memoria invece richiede anche un segnale di controllo per la lettura. Possiamo utilizzare l'unità di controllo della ALU progettata per l'unità di elaborazione a singolo ciclo (si vedano la Figura 4.13 e l'Appendice D). Infine, ciascun multiplexer a due vie richiede un singolo segnale di controllo, mentre il multiplexer a quattro vie ne richiede due.

La **Figura e4.5.3** riporta l'unità di elaborazione di Figura e4.5.2 dopo l'aggiunta dei segnali di controllo.

L'unità di elaborazione multi-ciclo richiede degli elementi supplementari, necessari a supportare i salti condizionati e incondizionati; dopo queste aggiunte si vedrà come le istruzioni vengono sequenzializzate e come generare la logica di controllo.

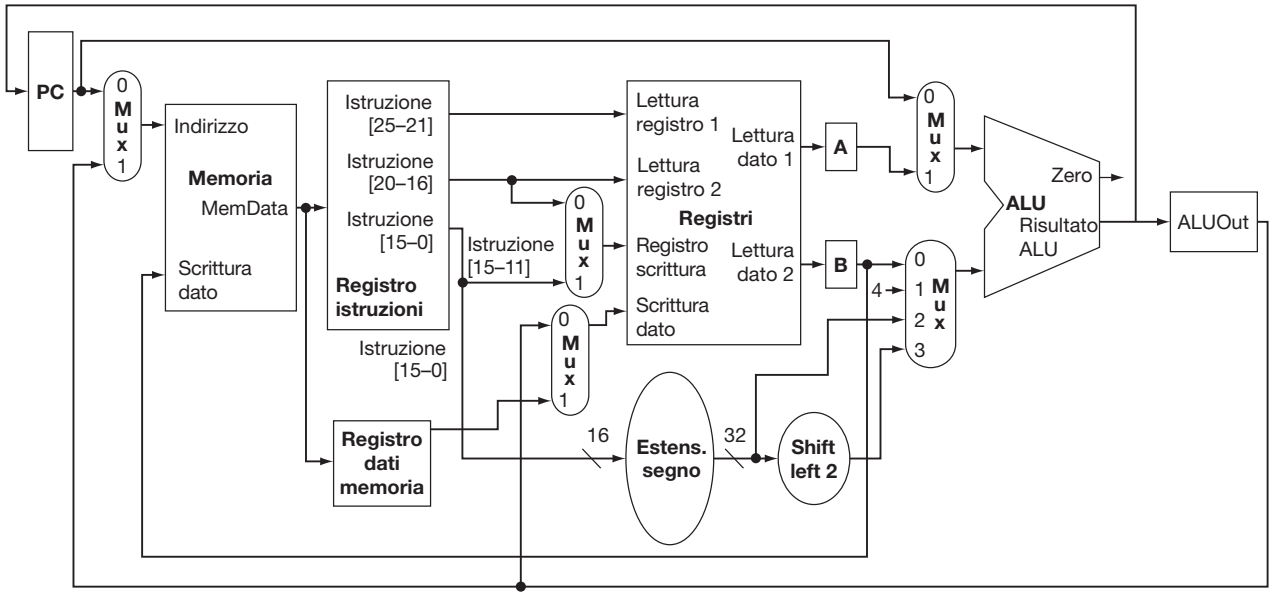

Figura e4.5.2 L'unità di elaborazione multi-ciclo che gestisce le principali istruzioni MIPS. Mentre questa unità di elaborazione supporta il normale incremento del PC, saranno necessarie alcune interconnessioni aggiuntive e un multiplexer per implementare i salti condizionati e incondizionati, come vedremo tra breve. Rispetto all'unità di elaborazione a singolo ciclo, le aggiunte comprendono diversi registri (IR, MDR, A, B, ALUOut), un multiplexer per l'indirizzo di memoria, un multiplexer per l'ingresso superiore della ALU e la trasformazione del multiplexer sull'ingresso inferiore della ALU in uno a quattro vie. Queste piccole aggiunte permettono di eliminare due sommatori e un'unità di memoria.

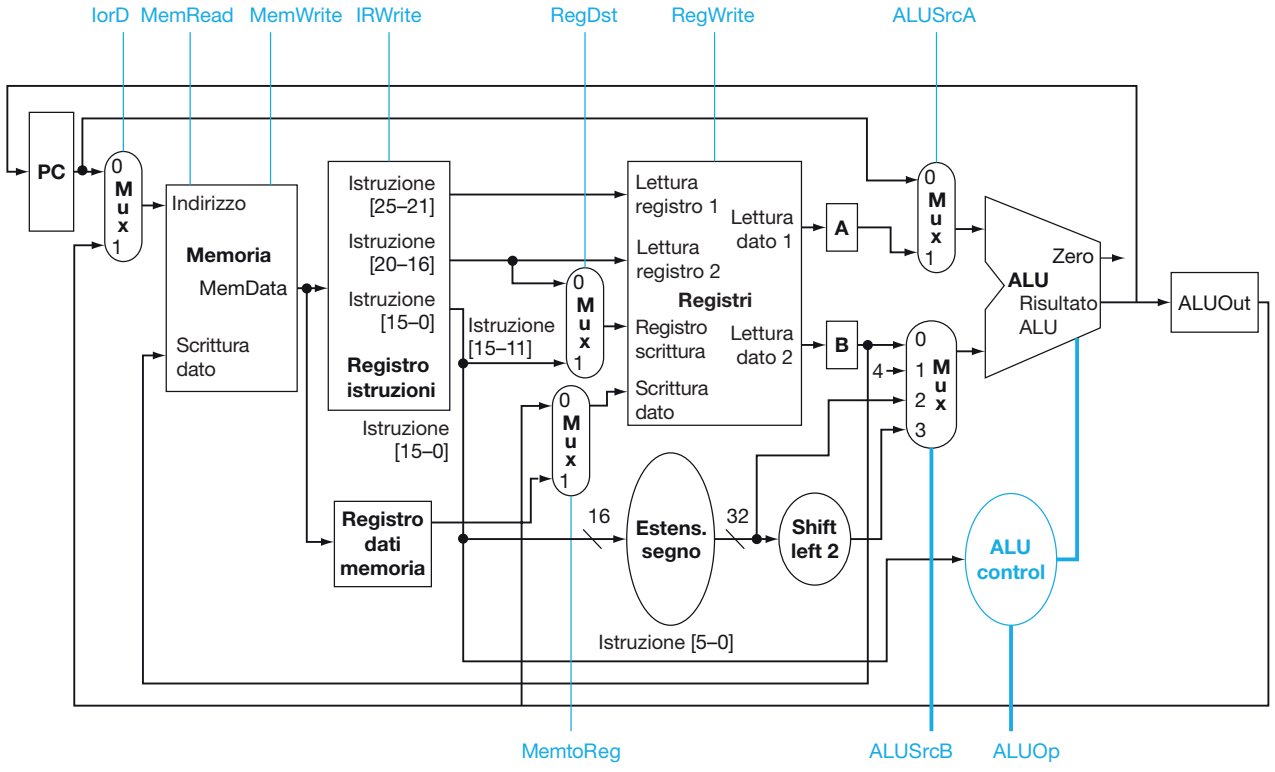

Figura e4.5.3 L'unità di elaborazione multi-ciclo della figura 5.26 con l'aggiunta dei segnali di controllo. I segnali ALUOp e ALUSrcB sono segnali di controllo a 2 bit, mentre tutti gli altri sono segnali di 1 bit; i registri A e B non esigono un segnale di scrittura in quanto il loro contenuto viene letto solo nel ciclo immediatamente successivo a quello in cui è scritto. Il Memory Data Register (MDR) è stato aggiunto per conservare il dato letto dalla memoria durante una load: il dato non può essere scritto direttamente nel register file perché il ciclo di clock non è sufficiente a contenere sia il tempo richiesto dall'accesso in memoria sia quello della scrittura nel register file. Il segnale MemRead è stato trasferito verso l'alto per semplificare la figura. La parte del circuito richiesta per eseguire e controllare i salti sarà aggiunta tra breve.

Aggiungendo le istruzioni branch e jump, vi sono tre possibili sorgenti per il valore da scrivere nel PC:

1. L'uscita della ALU, che vale PC + 4 durante il reperimento dell'istruzione; questo valore viene scritto direttamente dentro il registro PC;
2. Il registro ALUOut, che conterrà l'indirizzo del salto condizionato dopo che il calcolo è stato eseguito;
3. I 26 bit meno significativi del Registro Istruzioni (*Instruction Register*, IR), fatti scorrere di due posizioni a sinistra e concatenati con i 4 bit superiori di PC incrementato, che è la sorgente quando l'istruzione è una jump.

Come già osservato quando abbiamo implementato l'unità di elaborazione a singolo ciclo, il PC viene scritto sia in modo incondizionato sia condizionato. Durante i normali incrementi e le istruzioni jump il PC è scritto incondizionatamente. Se l'istruzione è un salto condizionato, il valore di PC incrementato è sostituito dal valore presente in ALUOut solamente se i due registri selezionati sono uguali. Quindi la nostra implementazione utilizzerà due segnali di controllo separati per la scrittura di PC: PCWrite che causa una scrittura incondizionata sul PC e PCWriteCond che causa una scrittura del PC solo se la condizione del salto è vera.

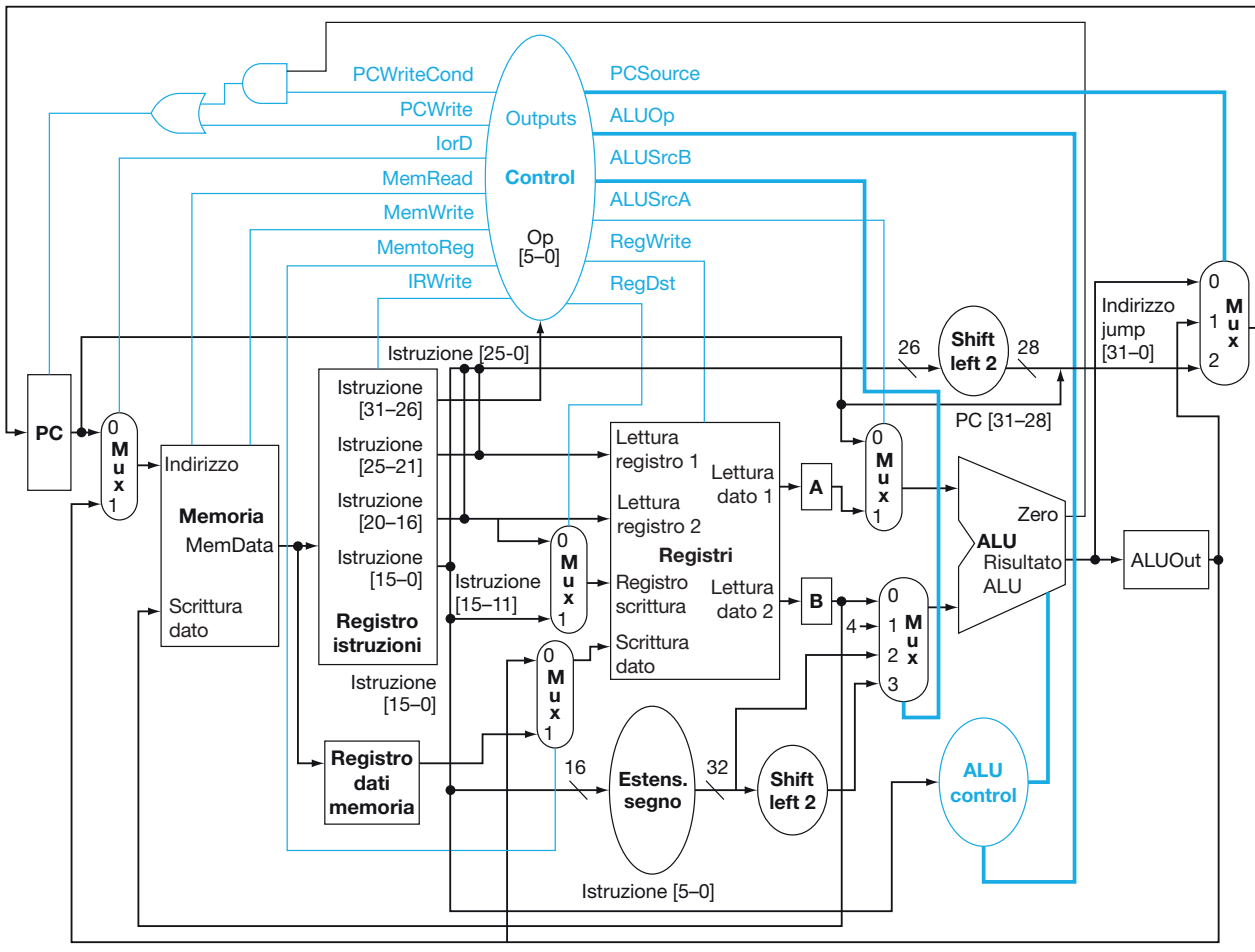

Figura e4.5.4 L'unità di elaborazione completa e i relativi segnali di controllo nell'implementazione multi-ciclo. I segnali di controllo della figura e4.5.3 sono connessi all'unità di controllo e sono stati inclusi tutti gli elementi necessari per la modifica di PC. Le aggiunte principali rispetto alla figura e4.5.3 sono il multiplexer utilizzato per selezionare la fonte del nuovo valore di PC, due porte logiche utilizzate per combinare i segnali di scrittura di PC e i segnali di controllo PCSource, PCWrite e PCWriteCond. Il segnale PCWriteCond è utilizzato per decidere se un salto condizionale deve essere preso. Il supporto per i salti incondizionati è incluso.

Dobbiamo collegare questi due segnali alla logica di controllo per la scrittura di PC. Come già fatto nell'implementazione a singolo ciclo, si utilizzeranno alcune porte logiche per ricavare il segnale di controllo per la scrittura del registro PC a partire da PCWrite, PCWriteCond e dall'uscita Zero della ALU, utilizzata per stabilire se i due registri, operandi della `beq`, sono uguali. Per determinare se PC debba essere scritto durante un salto condizionale, si collegano con una porta AND il segnale Zero della ALU e PCWriteCond. L'uscita della porta AND è a sua volta messa in OR con PCWrite, che è il segnale di controllo per la scrittura incondizionata. L'uscita della porta OR viene collegata al segnale di controllo della scrittura del registro PC.

La **Figura e4.5.4** mostra l'unità di elaborazione multi-ciclo completa e la relativa unità di controllo, compresi i segnali di controllo aggiuntivi e il multiplexer per implementare l'aggiornamento di PC.

Prima di esaminare in dettaglio i passi necessari per eseguire ciascuna istruzione, esaminiamo informalmente gli effetti dei segnali di controllo (come abbiamo fatto per il progetto a singolo ciclo in Figura 4.16). La **Figura e4.5.5** mostra l'operazione compiuta da ciascun segnale di controllo quando asserito e quando non asserito.

Azioni dei segnali di controllo di 1 bit

Nome del segnale	Effetto quando asserito	Effetto quando non asserito
RegDst	Il numero del registro destinazione per Registro scrittura proviene dal campo rt.	Il numero del registro destinazione per Registro scrittura proviene dal campo rd.
RegWrite	Nessuno.	Nel registro specificato sull'ingresso Registro scrittura è scritto il valore presente sull'ingresso Scrittura dati.
ALUSrcA	Il primo operando della ALU è PC.	Il primo operando della ALU è il registro A.
MemRead	Nessuno.	Il contenuto della cella di memoria determinata dall'ingresso Indirizzo è posto sull'uscita MemData.
MemWrite	Nessuno.	Il contenuto della cella di memoria determinata dall'ingresso Indirizzo è sostituito dal valore presente sull'ingresso Scrittura dati.
MemtoReg	Il valore inviato all'ingresso Scrittura dati dei registri proviene da ALUOut.	Il valore inviato all'ingresso Scrittura dati dei registri proviene da MDR.
IorD	L'indirizzo fornito alla memoria proviene da PC.	L'indirizzo fornito alla memoria proviene da ALUOut.
IRWrite	Nessuno.	L'uscita della memoria viene scritta in IR.
PCWrite	Nessuno.	PC viene scritto; la provenienza del valore è controllata da PCSource.
PCWriteCond	Nessuno.	PC viene scritto se anche l'uscita Zero della ALU è attiva.

Azioni dei segnali di controllo di 2 bit

Nome del segnale	Valore (binario)	Effetto
ALUOp	00	La ALU calcola una somma.
	01	La ALU calcola una sottrazione.
	10	La ALU calcola l'operazione determinata dal campo funct dell'istruzione.
ALUSrcB	00	Il secondo ingresso della ALU proviene dal registro B.
	01	Il secondo ingresso della ALU è la costante 4.
	10	Il secondo ingresso della ALU è il valore dei 16 bit meno significativi di IR, estesi a 32 bit con segno.
	11	Il secondo ingresso della ALU è il valore dei 16 bit meno significativi di IR, estesi a 32 bit con segno e scalati a sinistra di 2 bit.
PCSource	00	In PC viene scritta l'uscita della ALU (PC + 4).
	01	In PC viene scritto il contenuto di ALUOut (l'indirizzo di destinazione del salto condizionato).
	10	In PC viene scritto l'indirizzo di destinazione del salto incondizionato (IR[25-0] scalato a sinistra di 2 bit e concatenato con PC + 4[31-28]).

Figura e4.5.5 Azioni causate dai valori dei segnali di controllo della figura e4.5.4. La tabella superiore descrive i segnali di controllo di 1 bit, mentre quella inferiore riporta i segnali di controllo di 2 bit. Soltanto i segnali di controllo che agiscono sui multiplexer possono avere effetto quando non asseriti. Queste informazioni sono simili a quelle di figura 4.16 per l'unità di elaborazione a singolo ciclo, ma vi sono parecchi nuovi segnali (IRWrite, PCWrite, PCWriteCond, ALUSrcB e PCSource), mentre altri non utilizzati sono stati eliminati o sostituiti (PCSrc, Branch e Jump).

Approfondimento Per ridurre il numero di segnali che connettono le diverse unità funzionali, i progettisti possono utilizzare dei bus condivisi. Un bus condiviso è un insieme di linee che connettono diverse unità; nella maggior parte dei casi, comprendono le varie sorgenti che possono porre dei dati sul bus e le diverse destinazioni che possono leggere tali valori. In modo analogo alla riduzione del numero di unità funzionali nell'unità di elaborazione, è possibile ridurre il numero di bus che connettono le unità, condividendoli. Per esempio, vi sono sei sorgenti di dati che raggiungono la ALU, ma solo due di esse sono adoperate in ogni istante, per cui due bus sono sufficienti per trasportare i valori che devono raggiungere la ALU. Invece di utilizzare un grande multiplexer davanti alla ALU, un progettista può utilizzare un bus condiviso, purché garantisca che in ogni istante di tempo solo una delle sorgenti stia inviando dei dati verso il bus. Questa tecnica può permettere di risparmiare delle linee di dati, ma sarà indispensabile un pari numero di linee di controllo per determinare quale valore venga scritto sul bus. Un importante effetto collaterale di questa tecnica è una possibile riduzione delle prestazioni, poiché difficilmente un bus sarà veloce quanto una connessione da punto a punto.

Suddividere l'esecuzione delle istruzioni in cicli di clock

Data l'unità di elaborazione della Figura E4.5.4, per determinare quali nuovi segnali di controllo dovranno essere introdotti e indicare il valore che i segnali di controllo devono assumere, si deve esaminare che cosa deve accadere in ciascun ciclo di clock dell'esecuzione multi-ciclo. L'obiettivo nel decomporre l'esecuzione delle operazioni in cicli di clock dovrebbe essere quello di bilanciare la quantità di lavoro svolta in ciascun ciclo, in modo da minimizzare il periodo di clock. Si può iniziare suddividendo l'esecuzione di ciascuna istruzione in una serie di passi, ciascuno corrispondente a un ciclo di clock, che siano all'incirca della stessa lunghezza. Per esempio, possiamo fare in modo che ciascun passo contenga al più un'operazione con la ALU, un accesso al register file o un accesso alla memoria. Con questo vincolo il ciclo di clock dovrebbe risultare non più lungo della più lenta di queste operazioni.

Si ricordi che alla fine di ciascun ciclo di clock tutti i valori richiesti nel ciclo successivo dovranno essere memorizzati in un registro, che può essere uno degli elementi di stato principali (per esempio PC, il register file o la memoria), un registro temporaneo scritto a ogni ciclo di clock (per esempio A, B, MDR o ALUOut), oppure un registro temporaneo dotato di segnale di controllo della scrittura (per esempio IR). Si ricordi anche che, essendo il circuito sensibile ai fronti, si leggerà sempre il valore corrente dei registri, in quanto il nuovo valore non apparirà fino al ciclo di clock successivo.

Nel caso dell'unità di elaborazione a singolo ciclo, ciascuna istruzione utilizza durante l'esecuzione un insieme di elementi, di cui molti lavorano in serie, utilizzando come ingresso l'uscita di un altro elemento. Alcuni elementi invece operano in parallelo; per esempio, l'incremento di PC e la lettura dell'istruzione sono contemporanee. Una situazione analoga si verifica anche nell'unità di elaborazione multi-ciclo: tutte le operazioni attribuite a un certo passo vengono eseguite in parallelo in un ciclo di clock, mentre i passi successivi vengono eseguiti in serie in diversi cicli di clock. La limitazione a una sola operazione di tipo ALU, accesso alla memoria o accesso al register file determina ciò che può essere contenuto nello stesso passo.

Si noti che verrà fatta distinzione tra la lettura e la scrittura del PC o degli altri registri individuali rispetto alle letture e alle scritture nel register file. Mentre nel primo caso la lettura o la scrittura è interna al ciclo di clock, per leggere o scrivere un dato nel register file è necessario un ciclo di clock aggiuntivo. Il motivo di questa distinzione risiede nel fatto che il register file ha una logica di controllo e un overhead di accesso superiori rispetto ai singoli registri. Quindi, riducendo il periodo del clock, comporta il dover impiegare cicli di clock aggiuntivi per gli accessi al register file.

I possibili passi di esecuzione e le relative azioni compiute vengono riportati di seguito. Ciascuna istruzione MIPS richiede da tre a cinque passi.

1. Passo di fetch dell'istruzione

Caricamento dell'istruzione dalla memoria e calcolo dell'indirizzo dell'istruzione successiva in ordine sequenziale:

```
IR <= Memoria[PC];
PC <= PC + 4;
```

Operazione. Si utilizza PC come indirizzo di memoria, si esegue una lettura e si scrive l'istruzione nel registro istruzioni (IR), dove risiederà per i passi successivi, si incrementa inoltre PC di 4. Abbiamo utilizzato il simbolo <= dal Verilog; esso indica che tutti termini alla destra sono valutati e quindi assegnati, il che è effettivamente quello che l'hardware compie durante il ciclo di clock.

Per implementare questo passo dovranno essere asseriti i segnali di controllo MemRead e IRWrite e dovrà essere posto IorD a 0 per selezionare PC come sorgente dell'indirizzo. L'incremento di PC, nello stesso passo, richiede di porre il segnale ALUSrcA a 0 (inviando PC alla ALU) e ALUSrcB a 01 (inviando 4 alla ALU), con ALUOp pari a 00 (per forzare una somma). Infine, occorrerà memorizzare nuovamente in PC l'indirizzo incrementato, asserendo PCWrite. L'incremento di PC e l'accesso alla memoria possono avvenire in parallelo e il nuovo valore di PC non sarà visibile fino al successivo ciclo di clock. (Il valore incrementato di PC viene anche memorizzato in ALUOut, ma tale azione non provoca disfunzioni.)

2. Passo di decodifica dell'istruzione e caricamento dei registri

In questo passo e nel precedente non si conosce ancora quale sia l'istruzione, quindi si possono compiere solamente azioni applicabili a tutte le istruzioni (come prelevare l'istruzione nel passo 1) o che non siano dannose, nel caso in cui l'istruzione non sia quella supposta. Quindi in questo passo si possono leggere i due registri indicati dai campi rs e rt dell'istruzione, poiché non è dannoso leggerli anche nel caso in cui non siano richiesti: i valori letti dal register file potrebbero servire in passi successivi, quindi vengono letti dal register file e memorizzati nei registri temporanei A e B. Viene anche calcolato l'indirizzo di destinazione dei salti condizionati utilizzando la ALU, il che non è dannoso, poiché si può ignorare tale valore nel caso in cui l'istruzione non sia branch. Il potenziale indirizzo di destinazione viene memorizzato in ALUOut.

Svolgere anticipatamente in maniera "ottimistica" queste operazioni ha il beneficio di ridurre il numero di cicli di clock richiesti per eseguire un'istruzione. Tali operazioni possono essere eseguite così presto grazie alla regolarità del formato delle istruzioni. Per esempio, se l'istruzione ha due registri in ingresso, questi sono sempre nei campi rs e rt, mentre se l'istruzione è branch, l'offset è sempre nei 16 bit meno significativi:

```
A <= Reg[IR[25-21]];
B <= Reg[IR[20-16]];
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
```

Operazione. Si accede al register file per leggere i registri rs e rt e memorizzare il risultato nei registri A e B. Poiché A e B sono sovrascritti a ogni ciclo di clock, il register file deve essere letto a ogni ciclo. In questo passo si calcola anche l'indirizzo di destinazione dei salti condizionati e lo si memorizza in ALUOut, da dove verrà prelevato nel ciclo di clock successivo se l'istruzione è branch: ciò richiede di portare ALUSrcA a 0 (inviando PC alla ALU), ALUSrcB a 11 (inviando il campo offset, scalato ed esteso, alla ALU) e ALUOp a 00 (spe-

cificando una somma). L'accesso al register file e il calcolo dell'indirizzo del salto avvengono in parallelo. Terminato questo ciclo di clock, si determina la successiva azione in funzione dal valore dell'istruzione.

3. Esecuzione, calcolo dell'indirizzo di memoria o completamento del salto

Questo è il primo passo in cui l'operazione compiuta dall'unità di elaborazione è determinata dalla classe di istruzione. In ogni caso la ALU elabora gli operandi preparati nei passi precedenti svolgendo una funzione tra le tre possibili, a seconda della classe di istruzione. Si possono descrivere le azioni svolte per le diverse classi di istruzioni:

Accesso alla memoria:

```
ALUOut <= A + sign-extend(IR[15-0]);
```

Operazione. La ALU somma gli operandi in modo da calcolare l'indirizzo della memoria. Questo richiede di impostare ALUSrcA a 1 (inviando il registro A al primo ingresso della ALU) e ALUSrcB a 10 (inviando sul secondo ingresso della ALU l'uscita dell'unità di estensione del segno); il segnale ALUOp viene portato a 00 (facendo sì che la ALU esegua l'operazione di somma).

Istruzione logico-aritmetica (tipo-R):

```
ALUOut <= A op B;
```

Operazione. La ALU esegue l'operazione specificata dal codice di funzione sui due valori letti dal register file nel ciclo precedente. Questo richiede di impostare i valori ALUSrcA = l e ALUSrcB = 00 (la cui combinazione invia i registri A e B agli ingressi della ALU), mentre il segnale ALUOp viene portato a 10 (in modo che i segnali di controllo della ALU vengano determinati a partire dal campo funct).

Salto condizionato (branch):

```
if (A==B) PC <= ALUOut;
```

Operazione. Si utilizza la ALU per verificare l'uguaglianza tra i due registri letti nel passo precedente. Il segnale Zero della ALU verrà utilizzato per determinare se si debba eseguire il salto. In questo caso si pone ALUSrcA = l e ALUSrcB = 00 (mandando i valori provenienti dal register file alla ALU), con il segnale ALUOp posto a 01 (facendo sì che la ALU sottragga) per controllare l'eguaglianza. Il segnale PCCondWrite deve essere asserito per aggiornare PC nel caso in cui l'uscita Zero della ALU sia asserita; portando PCSource a 01, il valore scritto in PC proviene da ALUOut, che contiene l'indirizzo di destinazione del salto calcolato nel ciclo precedente. Per i salti condizionati che vengono eseguiti, in pratica PC è scritto due volte: la prima dall'uscita della ALU (nel passo di decodifica dell'istruzione e caricamento dei registri) e la seconda da ALUOut (durante il passo di completamento del salto). Il valore scritto per ultimo è quello che verrà utilizzato nella successiva fase di reperimento dell'istruzione.

Salto incondizionato (jump):

```
# {x, y} notazione Verilog per la concatenazione dei bit di
# x e y
PC <= {PC[31-28], IR[25-0], 2'b00)};
```

Operazione. Il valore di PC è sostituito dall'indirizzo del salto. PCSource è pilotato in modo da inviare l'indirizzo del salto a PC, e PCWrite viene asserito per scrivere l'indirizzo in PC.

4. Passo di accesso alla memoria o completamento dell'istruzione di tipo-R

In questo passo le istruzioni load o store accedono alla memoria e le istruzioni logico-aritmetiche scrivono il risultato. Quando un valore è recuperato dalla

memoria, questo è posto nel Memory Data Register (MDR), dal quale deve essere utilizzato nel ciclo successivo.

Accesso alla memoria:

```
MDR <= Memoria [ALUOut];
```

oppure

```
Memoria [ALUOut] <= B;
```

Operazione. Se l'istruzione è load, si recupera una parola di dato dalla memoria e la si scrive nel MDR. Se l'istruzione è store, il dato viene scritto in memoria. In entrambi i casi si utilizza l'indirizzo calcolato nel passo precedente e memorizzato in ALUOut. Per l'istruzione store, l'operando sorgente è memorizzato in B (in realtà B è stato letto due volte, una volta nel passo 2 e l'altra nel passo 3; fortunatamente nei due casi viene letto lo stesso valore, dal momento che il numero di registro, memorizzato in IR, non cambia). Si asserisce il segnale MemRead per load e MemWrite per store; inoltre, per le istruzioni load, il segnale IorD è posto a 1 in modo da usare l'indirizzo di memoria proveniente dalla ALU invece che quello prelevato da PC. Poiché MDR viene scritto a ogni ciclo, non richiede alcun segnale di controllo.

Istruzione logico-aritmetica (tipo-R):

```
Reg[IR[15-ll]] <= ALUOut;
```

Operazione. Si pone il contenuto di ALUOut, corrispondente al risultato dell'operazione della ALU del ciclo precedente, nel registro rd. Il segnale RegDst deve essere posto a 1 forzando l'utilizzo del campo rd (bit 15-11) nella selezione della cella del register file in cui scrivere. Il segnale RegWrite deve essere asserito, mentre MemtoReg viene impostato a 0 in modo da scrivere l'uscita della ALU anziché i dati provenienti dalla memoria.

5. Passo di completamento della lettura da memoria

In questo passo vengono completate le istruzioni load scrivendo il valore proveniente dalla memoria.

Load:

```
Reg[IR[20-16]] <= MDR;
```

Operazione. Si scrive il dato letto, che è stato memorizzato in MDR nel ciclo precedente, all'interno del register file. Per fare ciò si pone MemtoReg = 1 (per scrivere il dato proveniente dalla memoria), si asserisce RegWrite (causando una scrittura) e si asserisce RegDst = 0 per scegliere il campo rt (bit 20-16) come numero di registro.

La sequenza dei cinque passi è riassunta in **Figura e4.5.6**; da tale sequenza è possibile dedurre le azioni dell'unità di controllo in ciascun ciclo di clock.

Definizione dell'unità di controllo

A questo punto, avendo individuato quali sono i segnali di controllo e quando devono essere asseriti, si può procedere all'implementazione dell'unità di controllo. Nel progettare l'unità di controllo per l'unità operativa a singolo ciclo si è utilizzato un insieme di tabelle di verità che specificano i valori dei segnali di controllo in funzione della classe di istruzioni. L'unità di controllo dell'unità operativa multi-ciclo è più complessa, poiché ciascuna istruzione viene eseguita in più passi: l'unità di controllo deve dunque specificare sia i segnali da asserire in ciascuno dei passi sia il passo successivo della sequenza.

Nome del passo	Azione per istruzione di tipo-R	Azione per salti di accesso alla memoria condizionati	Azione per istruzione	Azione per salti incondizionati
Prelievo dell'istruzione	IR <= Memoria[PC] PC <= PC + 4			
Decodifica dell'istruzione/ caricamento dei registri	A <= Reg[IR[25-21]] B <= Reg(IR[20-16]] ALUOut <= PC + (sign-extend(IR[15-0]) << 2)			
Esecuzione, calcolo dell'indirizzo, completamento dei salti	ALUOut <= A op B	ALUOut <= A +sign-extend (IR[15-0])	if (A == B) then PC <= ALUOut	PC <= {PC[31-28], IR[25-0],2'b00)};
Accesso alla memoria o completamento dell'istruzione di tipo-R	Reg [IR(15-11]] <= ALUOut	Load: MDR <= Memoria[ALUOut] oppure Store: Memoria[ALUOut] <= B		
Completamento della lettura da memoria		Load: Reg[IR[20-16]] <= MDR		

Figura e4.5.6 Riassunto dei passi intrapresi durante l'esecuzione delle diverse classi di istruzioni. Le istruzioni possono richiedere da tre a cinque passi di esecuzione. I primi due passi sono indipendenti dalla classe di istruzioni, mentre in quelli successivi l'istruzione richiede da uno a tre cicli aggiuntivi per essere completata, a seconda della classe. Le caselle vuote nei passi di accesso alla memoria e di completamento della lettura indicano che le classi di istruzioni corrispondenti richiedono un numero minore di cicli. In un'implementazione multi-ciclo, si inizia l'esecuzione di una nuova istruzione non appena l'istruzione corrente viene completata, quindi tali cicli non sono di attesa e non vengono sprecati. Come già citato, il register file di fatto viene letto a ogni ciclo, ma finché IR non cambia i valori letti sono sempre uguali. In particolare, il valore letto nel registro B durante lo stadio di decodifica dell'istruzione per le istruzioni branch o di tipo-R è uguale al valore che B assume durante il passo di esecuzione e che viene usato nell'accesso alla memoria per l'istruzione store word.

Il paragrafo 4.14 mostra come vengono utilizzati i linguaggi di progettazione dell'hardware con entrambi gli esempi di unità di elaborazione multi-ciclo e di controllo a stati finiti. Nel progetto dei moderni sistemi digitali, il passo finale relativo al passaggio dalla descrizione dell'hardware alla realizzazione vera e propria tramite porte logiche è gestito da strumenti di sintesi logica. L'Appendice D mostra come funziona questo processo attraverso la traduzione dell'unità di controllo multi-ciclo in una implementazione hardware dettagliata. I concetti principali sull'unità di controllo si possono intuire da questo capitolo senza esaminare il materiale contenuto nell'Appendice D. Il paragrafo 5.9 è utile per coloro che vogliono effettivamente realizzare un progetto hardware, e l'Appendice C mostra come potrà apparire l'implementazione a livello di porte logiche.

Calcolo di CPI nelle CPU multi-ciclo

ESEMPIO

Utilizzando la combinazione di istruzioni relativa al benchmark SPECINT2006 mostrata in Figura 3.24, qual è il valore del CPI se si assume che ciascuno stato richieda un ciclo di clock?

SOLUZIONE

La combinazione è di un 20% di load, 8% di store, 10% di branch e 62% ALU (si suppone che tutte le altre istruzioni della combinazione siano istruzioni aritmetico-logiche). Dalla Figura 4.5.6 si deduce il numero di cicli di clock per ciascuna classe di istruzioni come segue:

- Load: 5
- Store: 4
- Istruzioni ALU: 4
- Salti condizionati: 3
- Salti incondizionati: 3

Il valore del CPI è dato dalle espressioni seguenti:

$$\text{CPU} = \frac{\text{Cicli di clock CPU}}{\text{Numero istruzioni}} = \frac{\sum \text{Numero istruzioni}_i \text{CPI}_i}{\text{Numero istruzioni}} =$$

$$= \sum \frac{\text{Instruction count}_i}{\text{Instruction count}} \times \text{CPI}_i$$

(continua)

(continua)

Il rapporto fra Numero istruzioni$_i$ e Numero istruzioni non è altro che la frequenza delle istruzioni della classe *i*. Si possono quindi sostituire i valori e ottenere

$$\text{CPI} = 0{,}20 \times 5 + 0{,}08 \times 4 + 0{,}62 \times 4 + 0{,}11 + 0{,}10 \times 3 = 4{,}10$$

Questo valore del CPI è migliore di quello che si sarebbe ottenuto nel caso peggiore, cioè se tutte le istruzioni avessero richiesto lo stesso numero di cicli di clock (ossia 5). Ovviamente l'overhead in entrambi i tipi di progetto può ridurre o incrementare questa differenza. L'implementazione multi-ciclo è più conveniente, dal momento che utilizza meno componenti all'interno dell'unità di elaborazione.

Il metodo che abbiamo utilizzato per specificare l'unità di controllo multi-ciclo fa uso delle **macchine a stati finiti** (*finite state machine*, FSM). Una macchina a stati finiti consiste in un insieme di stati e di istruzioni su come cambiare stato. Le istruzioni sono definite tramite una **funzione stato prossimo** (letteralmente successivo), che mette in corrispondenza lo stato presente e gli ingressi con lo stato futuro. Nell'utilizzo delle macchine a stati finiti per il progetto delle unità di controllo, ciascuno stato specifica inoltre l'insieme delle uscite che vengono asserite quando la macchina si trova in quello stato; nell'implementazione della macchina si assume normalmente che le restanti uscite siano non asserite. Il corretto funzionamento dell'unità di elaborazione dipende dal fatto che i segnali non esplicitamente asseriti siano effettivamente non asseriti, anziché indifferenti. Per esempio, il segnale RegWrite deve essere asserito solo quando si vuole scrivere un elemento del register file. Quando questo segnale non è esplicitamente asserito, deve essere pari a zero per evitare scritture indesiderate.

Macchina a stati finiti: una funzione logico-sequenziale costituita da un insieme di ingressi e uscite, una funzione stato prossimo che trasforma lo stato corrente e gli ingressi in un nuovo stato, e una funzione di uscita che trasforma lo stato corrente e gli ingressi in un insieme di uscite asserite.

Funzione stato prossimo: una funzione combinatoria che, dati gli ingressi dello stato corrente, determina lo stato prossimo di una macchina a stati finiti.

I segnali di controllo dei multiplexer sono trattati diversamente, dal momento che selezionano uno degli ingressi sia che valgano 0 sia che valgano 1. Nella macchina a stati finiti, quindi, verranno sempre specificati i valori di tutti gli ingressi di controllo dei multiplexer che ci interessano. In alcuni tipi di implementazione dell'unità di controllo, forzare di un segnale a 0 potrebbe essere un'azione automatica e potrebbe non richiedere alcuna porta logica. Un esempio di questo tipo di macchine a stati finiti è riportato nell'Appendice D e si invitano i lettori che non abbiano sufficiente familiarità con il concetto di macchina a stati finiti a consultare l'Appendice D prima di proseguire.

L'unità di controllo a stati finiti corrisponde fondamentalmente ai cinque passi di esecuzione mostrati nelle pagine precedenti; ogni passo della macchina a stati finiti corrisponde a un ciclo di clock. La macchina a stati finiti è composta di diverse parti: dato che i primi due passi di esecuzione sono identici in tutte le istruzioni, i primi due stati della macchina a stati finiti saranno comuni a tutte le istruzioni, i passi da 3 a 5 dipendono invece dal codice operativo. Al termine dell'esecuzione dell'ultimo passo di una particolare classe di istruzioni, la macchina a stati finiti dovrà ritornare allo stato iniziale per iniziare a prelevare l'istruzione successiva.

La **Figura e4.5.7** mostra una rappresentazione astratta della macchina a stati finiti. Per completarne i dettagli, si espanderà per prima la parte relativa al prelievo dell'istruzione e alla decodifica, mentre in seguito si esamineranno gli stati e le azioni relativi alle diverse classi di istruzioni.

Nella **Figura e4.5.8** sono riportati i primi due stati della macchina a stati finiti utilizzando la tradizionale rappresentazione grafica. Gli stati sono numerati per semplificare l'esposizione, sebbene i numeri siano arbitrari; lo stato 0, corrispondente al passo 1, è lo stato iniziale della macchina.

I segnali asseriti in ciascuno stato sono indicati all'interno del cerchio che lo rappresenta. Gli archi tra gli stati definiscono lo stato futuro e, nel caso in cui ne siano possibili molti, sono etichettati con le condizioni che selezionano uno specifico stato futuro. Dopo lo stato 1, i segnali affermati dipendono dalla classe

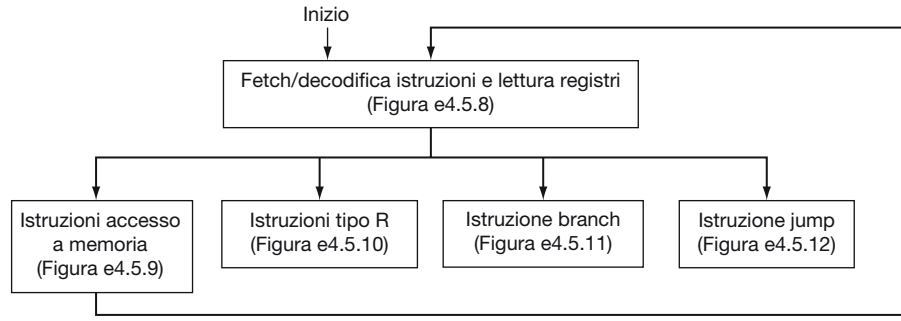

Figura e4.5.7 Vista ad alto livello dell'unità di controllo come macchina a stati finiti. I primi passi sono indipendenti dalla classe dell'istruzione, in seguito il completamento delle istruzioni delle diverse classi richiede delle sequenze dipendenti dal codice operativo; dopo aver terminato le azioni richieste per le diverse classi di istruzioni, l'unità di controllo inizia a prelevare una nuova istruzione. Ciascun rettangolo della figura rappresenta uno o più stati; l'arco etichettato con Inizio indica lo stato in cui inizia l'esecuzione al momento di prelevare la prima istruzione.

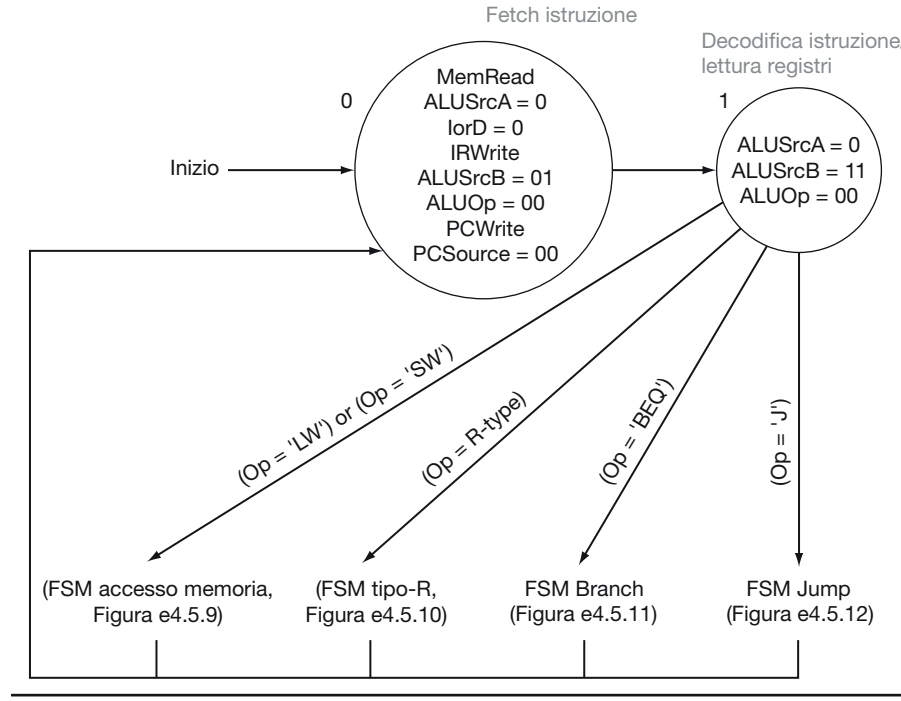

Figura e4.5.8 La porzione di fetch e decodifica dell'istruzione è identica per tutte le istruzioni. Gli stati riportati in questa figura corrispondono al rettangolo superiore della macchina a stati finiti astratta della Figura e4.5.7. Nel primo stato si asseriscono due segnali per leggere un'istruzione dalla memoria e scriverla nell'Instruction Register (MemRead e IRWrite) e si pone IorD a 0 per scegliere PC come sorgente dell'indirizzo. I valori dei segnali ALUSrcA, ALUSrcB, ALUOp, PCWrite e PCSource fanno sì che PC + 4 venga calcolato e memorizzato in PC (oltre a memorizzarlo in ALUOut, da cui però non viene mai letto). Nello stato successivo viene calcolato l'indirizzo di destinazione del salto forzando ALUSrcB a 11 (in modo che i 16 bit meno significativi, estesi e fatti scorrere, vengano inviati alla ALU), ALUSrcA a 0 e ALUOp a 00: il risultato viene memorizzato nel registro ALUOut, il quale viene comunque scritto a ogni ciclo di clock. Vi sono poi quattro stati futuri possibili, in funzione della classe di istruzione che è nota in questo stato. Il valore di un ingresso dell'unità di controllo, chiamato Op, determina quale arco debba essere seguito. Si ricordi che tutti i segnali non esplicitamente asseriti sono non asseriti; questo è particolarmente importante per i segnali che controllano le scritture. Per i segnali di controllo relativi ai multiplexer, la mancanza di specifiche al riguardo indica che è indifferente come questi vengono impostati.

dell'istruzione, pertanto la macchina a stati finiti ha quattro archi uscenti dallo stato 1, corrispondenti alle quattro classi: accesso alla memoria, tipo-R, branch on equal e jump. L'azione di portarsi in stati diversi in funzione del tipo di istruzione è chiamato *decodifica*, in quanto la scelta dello stato futuro, e quindi le azioni che ne seguiranno, dipende dalla classe dell'istruzione.

La **Figura e4.5.9** riporta la porzione di macchina a stati finiti utilizzata nell'implementazione delle istruzioni di accesso alla memoria. Per tali istruzioni nel primo stato, dopo il prelievo dell'istruzione e dei registri, viene calcolato l'indirizzo di memoria (stato 2). Per calcolare tale indirizzo occorrono dei valori sui multiplexer in ingresso alla ALU tali da far sì che sul primo ingresso vada il registro A e che il secondo ingresso sia ottenuto tramite estensione del segno dal campo di spiazzamento; il risultato è scritto nel registro ALUOut. Dopo il calcolo dell'indirizzo di memoria, quest'ultima deve essere letta o scritta, il che richiede due stati differenti. Se il codice operativo dell'istruzione è `lw`, allora lo stato 3 (corrispondente al passo di accesso alla memoria) effettua la lettura (viene asserito MemRead). L'uscita della memoria è quindi sempre scritta in MDR. Se l'opcode è invece `sw`, lo stato 5 effettua una scrittura in memoria (asserendo MemWrite). Negli stati 3 e 5, il segnale IorD è posto a 1 per forzare la provenienza dell'indirizzo di memoria dalla ALU. Dopo l'esecuzione della scrittura, l'istruzione `sw` è completa e lo stato futuro è lo stato 0. Se l'istruzione è una load, è necessario un altro stato (stato 4) per scrivere il risultato proveniente

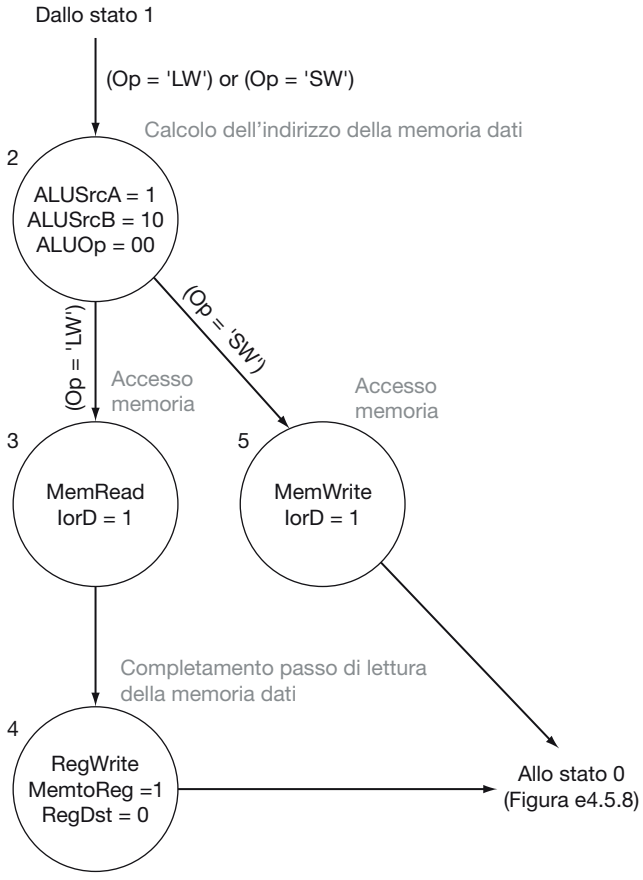

Figura e4.5.9 La macchina a stati finiti per il controllo delle istruzioni di accesso alla memoria ha quattro stati. Tali stati corrispondono al rettangolo denominato "Istruzioni di accesso alla memoria" di figura e4.5.7. Dopo il calcolo dell'indirizzo di memoria, che avviene nello stato 2 grazie ai segnali di controllo ALUSrcA, ALUSrcB e ALUOp, sono richieste sequenze diverse per le istruzioni load e store. Le operazioni di load richiedono uno stato aggiuntivo per copiare il risultato da MDR (in cui è stato scritto nello stato 3) al register file.

dalla memoria nel register file. Tale operazione viene ottenuta impostando i segnali di controllo dei multiplexer come MemtoReg = 1 e RegDst = 0, in modo da inviare il valore di MDR al register file, utilizzando rt come numero di registro. Dopo tale stato, corrispondente al passo di completamento della lettura da memoria, lo stato futuro sarà lo stato 0.

L'implementazione delle istruzioni di tipo-R richiede due stati, corrispondenti ai passi 3 (esecuzione) e 4 (completamento dell'istruzione di tipo-R). La **Figura e4.5.10** mostra questa parte, composta da due stati, della macchina a stati finiti. Lo stato 6 asserisce il segnale ALUSrcA e pone ALUSrcB a 00, facendo sì che i due registri letti dal register file vengano utilizzati come ingressi della ALU. Viene inoltre impostato ALUOp a 10, in modo da pilotare l'unità di controllo della ALU a utilizzare il campo funzione nella determinazione dei segnali di controllo della ALU. Nello stato 7 viene asserito il segnale RegWrite per provocare una scrittura nel register file, viene asserito RegDst in modo da utilizzare il campo rd come numero di registro destinazione, mentre MemtoReg è non asserito, affinché ALUOut sia selezionato come sorgente del valore che verrà scritto nel register file.

Nel caso dei salti condizionati, occorre solo uno stato aggiuntivo, poiché questi terminano l'esecuzione già al terzo passo. In questo stato occorre impostare i segnali di controllo in modo da causare il confronto dei registri A e B e la scrittura condizionale di PC con l'indirizzo presente nel registro ALUOut. L'esecuzione del confronto richiede di asserire ALUSrcA e impostare ALUSrcB a 00, ponendo ALUOp a 01 per pilotare la sottrazione (verrà utilizzata solo l'uscita Zero della ALU trascurando il risultato della sottrazione). La scrittura di PC richiede invece di asserire PCWriteCond e di impostare PCSource = 01, provocando la scrittura del valore del registro ALUOut (che contiene l'indirizzo del salto calcolato nello stato 1, come da Figura e4.5.8) in PC, questo ovviamente se l'uscita Zero della ALU è asserita. La **Figura e4.5.11** mostra questo unico stato.

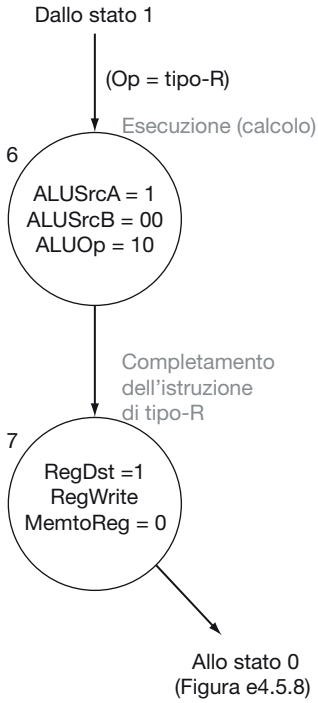

Figura e4.5.10 Le istruzioni di tipo-R si implementano con una semplice macchina con due stati. Tali stati corrispondono al rettangolo denominato "Istruzioni di tipo-R" di figura e4.5.7. Il primo stato esegue l'operazione logico-aritmetica, mentre il secondo provoca la scrittura del risultato (che si trova in ALUOut) nel register file. Nello stato 7 i tre segnali di controllo asseriti determinano la scrittura del contenuto di ALUOut nel registro del register file specificato dal campo rd del registro istruzioni.

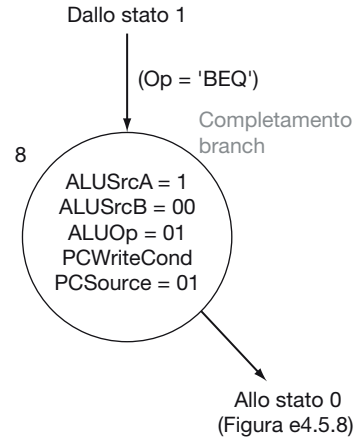

Figura e4.5.11 L'istruzione branch richiede un solo stato. Le prime tre uscite che vengono asserite in tale stato (ALUSrcA, ALUSrcB e ALUOp) specificano alla ALU di eseguire il confronto, mentre i segnali PCSource e PCWriteCond determinano la scrittura condizionale nel caso in cui la condizione di salto sia verificata. Si noti che non viene utilizzato il valore memorizzato nel registro ALUOut, ma direttamente l'uscita Zero della ALU. L'indirizzo di destinazione del salto è invece letto da ALUOut, dove era stato memorizzato al termine dello stato 1.

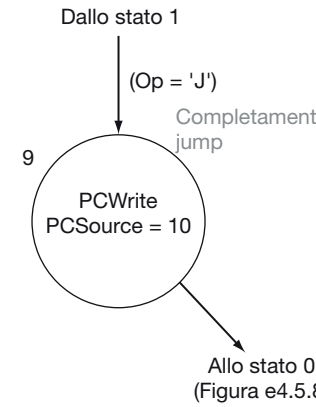

Figura e4.5.12 L'istruzione jump richiede uno stato per asserire i due segnali di controllo che scrivono in PC i 26 bit meno significativi dell'instruction register fatti scorrere a sinistra di 2 posizioni e concatenati ai 4 bit più significativi del PC dell'istruzione.

L'ultima classe di istruzioni è costituita dai salti incondizionati (jump). Come il branch, richiede un solo stato per il proprio completamento (**Figura e4.5.12**), nel quale si asserisce il segnale PCWrite per provocare la scrittura di PC. Si imposta PCSource a 10 in modo che il valore scritto sia costituito dai 26 bit dell'instruction register, a cui si concatenano 00_{due} come bit meno significativi e i 4 bit più significativi di PC.

A questo punto è possibile unire insieme le parti della macchina a stati finiti in modo da fornire la specifica dell'unità di controllo, come mostrato in **Figura e4.5.13**. In ogni stato sono riportati i segnali asseriti, mentre lo stato futuro dipende dai bit del codice operativo dell'istruzione, quindi gli archi sono etichettati con operazioni di confronto con i relativi codici operativi.

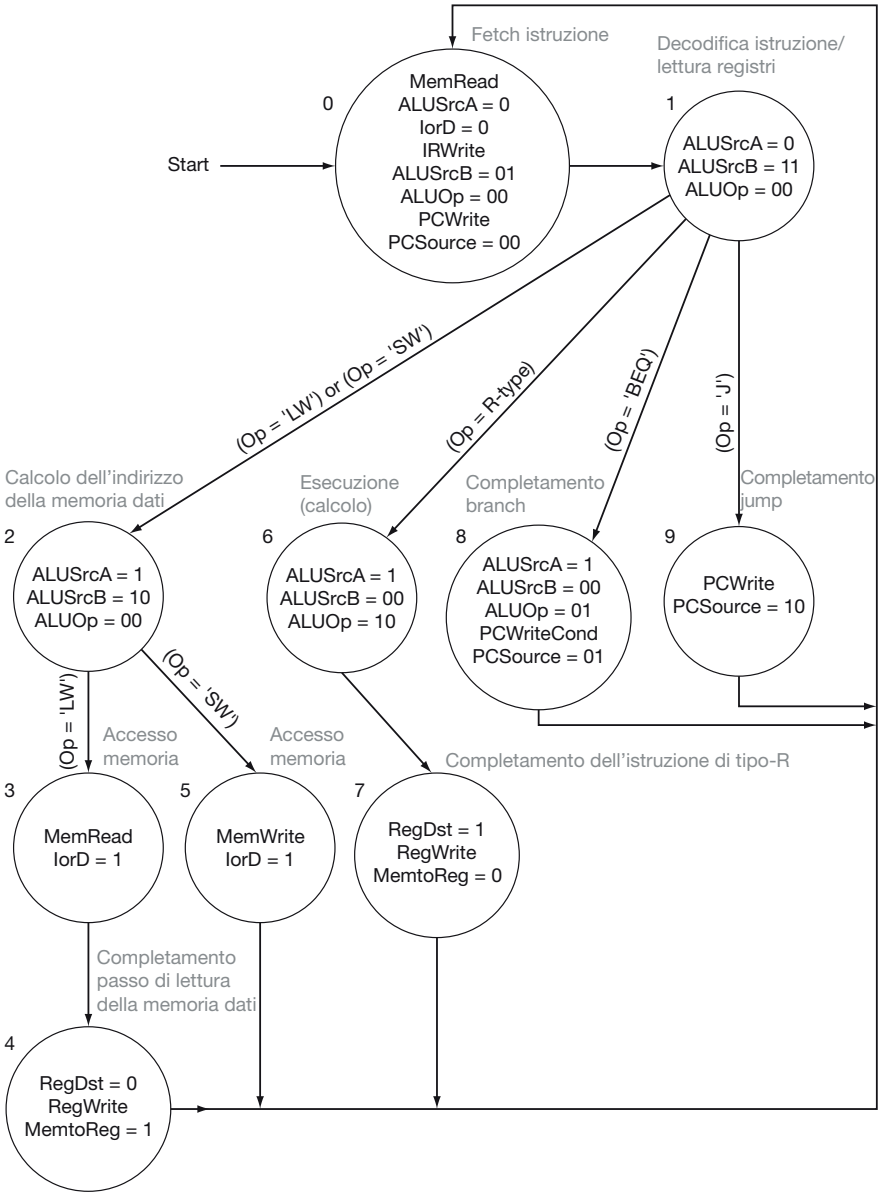

Figura e4.5.13 La macchina a stati finiti per il controllo relativa all'unità di elaborazione di Figura e4.5.14. Le etichette sugli archi sono le condizioni da verificare nella determinazione dello stato prossimo. Nel caso in cui tale stato prossimo non dipenda da una condizione, l'etichetta è assente. Le etichette interne ai nodi indicano i segnali di uscita dell'unità di controllo asseriti durante lo stato corrispondente; gli ingressi dei multiplexer vengono sempre specificati quando la loro uscita è richiesta, quindi in alcuni stati si troveranno ingressi di multiplexer posti a 0.

Una macchina a stati finiti può essere implementata mediante un registro temporaneo che memorizza lo stato corrente e un blocco di logica combinatoria che determina sia i segnali che devono essere asseriti nell'unità di elaborazione, sia lo stato prossimo. Una rappresentazione di tale struttura è riportata in **Figura e4.5.14**. L'Appendice D mostra in dettaglio come si possa implementare una macchina a stati finiti con questa struttura. Nel paragrafo D.3 viene implementata la logica combinatoria di controllo per la macchina a stati finiti di Figura e4.5.13, sia tramite una memoria a sola lettura (ROM, *Read-Only Memory*) sia mediante una matrice logica programmabile (PLA, *Programmable Logic Array*). (Si veda l'Appendice B per una descrizione di questi elementi). Il prossimo paragrafo propone invece un approccio alternativo per rappresentare l'unità di controllo. Le due tecniche esaminate sono semplicemente rappresentazioni alternative delle stesse informazioni di controllo.

La pipeline, che è l'argomento del Capitolo 6, è quasi sempre utilizzata per accelerare l'esecuzione delle istruzioni. Per istruzioni semplici la pipeline è capace di raggiungere frequenze di clock elevate tipiche di un progetto multi-ciclo e, allo stesso tempo, un CPI di un singolo ciclo come i progetti a singolo ciclo di clock. In molti processori che utilizzano la pipeline, comunque, l'esecuzione di alcune istruzioni impiega più di un singolo ciclo di clock e richiede un controllo multi-ciclo. Le istruzioni in virgola mobile sono un tipico esempio. Ci sono molti esempi nell'architettura IA-32 che richiedono l'uso di un controllo multi-ciclo.

Approfondimento Il tipo di macchina a stati finiti riportato in Figura e5.4.14 è detto *macchina di Moore*, dal nome di Edward Moore. La sua caratteristica principale è che le uscite dipendono solamente dallo stato presente. In una macchina di Moore il rettangolo denominato *logica combinatoria di controllo* può essere

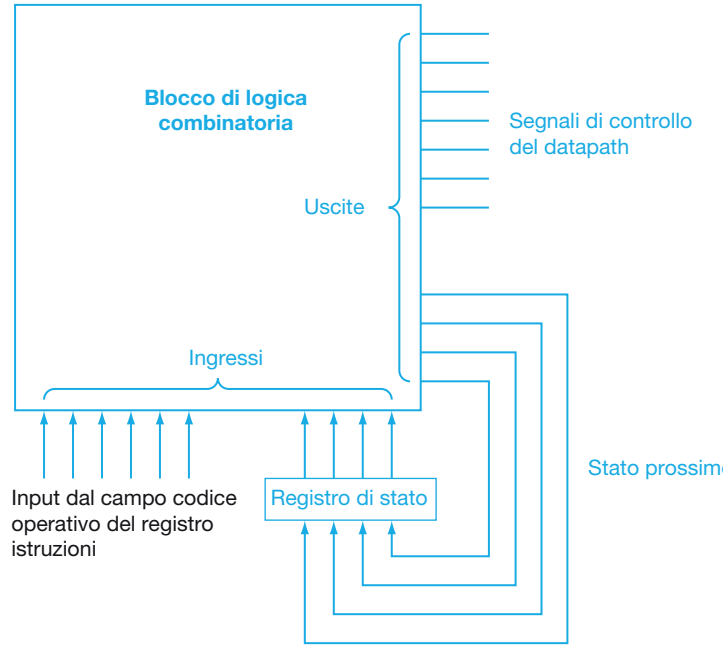

Figura e4.5.14 Le unità di controllo realizzate come macchine a stati finiti sono tipicamente implementate utilizzando un blocco di logica combinatoria e un registro per memorizzare lo stato corrente. L'uscita della logica combinatoria è il numero dello stato successivo e i segnali di controllo da asserire durante il ciclo corrente, mentre gli ingressi sono lo stato presente e altri ingressi necessari a determinare lo stato prossimo; in questo caso i bit del campo codice operativo. Si noti che nelle macchine a stati finiti utilizzate in questo capitolo le uscite dipendono soltanto dallo stato presente e non dagli ingressi. L'approfondimento che segue spiega con maggior dettaglio questo concetto.

diviso in due parti: la prima contiene le uscite di controllo e riceve in ingresso solo lo stato, mentre la seconda ha una sola uscita, quella di stato futuro.

Un tipo alternativo di macchina a stati finiti è la *macchina di Mealy*, dal nome di George Mealy. Tale macchina permette di utilizzare sia lo stato presente sia il valore degli ingressi per determinare le uscite. Le macchine di Moore presentano dei vantaggi teorici di implementazione in termini di velocità e dimensioni dell'unità di controllo. La velocità si giustifica notando che le uscite di controllo, che devono essere disponibili il più presto possibile nel ciclo di clock, non dipendono dagli ingressi ma solo dallo stato presente. Il vantaggio nelle dimensioni sarà evidente nell'Appendice D, quando si procederà nell'implementazione fino a ottenere le porte logiche. Un possibile svantaggio delle macchine di Moore è che possono richiedere un numero maggiore di stati. Per esempio, in situazioni in cui vi sia la differenza in un solo stato tra due sequenze di stati, in una macchina di Mealy si possono fondere tali stati rendendo le uscite dipendenti dagli ingressi.

Capire le prestazioni dei programmi

Per un processore con una data frequenza di clock, le prestazioni relative tra due segmenti di codice saranno determinate dal prodotto del CPI e del numero di istruzioni eseguite in ogni segmento. Come abbiamo visto in questo paragrafo, le istruzioni possono avere un CPI differente, anche per un semplice processore. Nei prossimi due capitoli vedremo che l'introduzione della pipeline e l'utilizzo di cache creeranno ulteriori condizioni per la variazione del CPI. Sebbene molti fattori che influenzano il CPI siano controllati dal progettista hardware, il programmatore, il compilatore e il sistema software determinano quali istruzioni sono eseguite, ed è questo processo che determina quale sarà il CPI effettivo per il programma. I programmatori che cercano di incrementare le prestazioni devono capire il ruolo del CPI e i fattori che lo influenzano.

Autovalutazione

1. Vero o falso: poiché l'istruzione jump non dipende dal valore dei registri o dal calcolo dell'indirizzo di destinazione del salto, può essere completata durante il secondo stato, invece che aspettare fino al terzo.
2. Vero o falso o può darsi: il segnale di controllo PCWriteCond può essere sostituito da PCSource[0].