Struttura della CPU

Struttura interna della CPU

Data Path di una semplicissima CPU

Per analizzare in dettaglio la struttura interna di una CPU facciamo un esempio didattico con la CPU NS-0 (Nello Scarabottolo).

Caratteristiche:

- Macchina RISC: opcode da 2 bit, per un totale di 2^2=4 istruzioni;
  - Le istruzioni macchina non sono tutte di uguale lunghezza (una delle istruzioni usano due parole di memoria).
- Metodi di indirizzamento: solamente immediato e diretto;
- Macchina a 16bit:
  - Data Bus a 16bit e dunque celle di memoria da 16 bit;
  - Address Bus a 14bit dunque 2^14=16K celle di spazio di indirizzamento.

Elementi interni:

- 1 registro GPR denominato R0;
- 1 registro di CC che puo' contenere solo il bit Z che ci dice se l'ultima operazione di somma da parte delle'ALU ha prodotto un risultato nullo;
- ALU che fa solo l'operazione di ADD;
- 1 Internal Bus di input che va verso i registri;
- 2 Internal Bus di output che vanno fuori dai registri.

Set di istruzioni:

- LOAD (opcode 00): porta nel registro R0 il contenuto della cella di memoria il cui indirizzo a 14bit e' contenuto nell'istruzione stessa;
- STORE (opcode 01): porta in memoria il contenuto di R0;
- ADD (opcode 10): occupa due parole di memoria:
  - La prima solo per indicare l'opcode 10;
  - La seconda per contenere una costante a 16 bit che sommata a R0 produce il risultato scritto su R0.
- BRZ (opcode 11): istruzione di salto per cui:
  - Se trova nel CC il bit Z a 1, forza l'indirizzo a 14 bit presente nell'istruzione nel PC.

Un registro della CPU NS-0 e' costituito da un certo numero di bistabili di tipo D, tanti quanti i bit del registro. Esiste un comando Rin che abilita tutti i bistabili e fa si cio' che e' presente nell'Internal Bus di input venga scritto nel registro. Ed esiste di riflesso un comando Rout1-2 (a seconda di quale Internal Bus di

output va) che serve a abilitare/disabilitare i buffer TRI-STATE che contentono al contenuto del registro di essere emesso sul bus interno di destinazione, oppure di rimanere celato perche' le uscite sono ad alta impedenza.

Il registro di interfaccia al Data Bus esterno e' leggermente differente in quanto puo' ricevere le informazioni da campionare sia dall'Internal Bus in input sia dal Data Bus e puo' emettere le proprie informazioni sia sugli Internal Bus di output sia sul Data Bus. Esistono in questo caso 4 comandi:

- MDRin: campiona nel registro MDR il contenuto dell'Internal Bus di input;
- MDRsample: campiona nel registro MDR cio' che compare nel Data Bus;
- MDRout1: emette nell'Internal Bus di output il contenuto del registro MDR;
- MDRoutenable: emette nel Data Bus esterno (verso i dispositivi Slave) il contenuto dei MDR.

Lo scopo degli Internal Bus e' quello di consentire ai registri di emettere il suo contenuto e di campionare il contenuto scritto. Se l'ALU e' in grado di propagare cio' che si presenta ai suoi ingressi A o B in uscita O, questo consente di far circolare le informazioni tra i registri.

Vediamo quindi quali sono le istruzioni che l'ALU deve essere in grado di fare:

- **NOP** - comando 00: non chiediamo all'ALU di fare nulla;
- **PASS** - comando 01: funzione che consente all'ALU di prendere il contenuto del proprio ingresso A e propagarlo internamente in uscita (sull'Internal Bus di input);
- **INC** - comando 10: funzione di incremento che porta in output il valore dell'ingresso A aumentato di 1;
- **ADD** - comando 11: porta in uscita il risultato della somma dei due ingressi A e B e nel caso il risultato sia 0 porta 1 nella propria uscita di esito (**F**lag) di modo che i CC possano campionarlo.

Abbiamo definito dunque il **Data Path**, ossia il percorso di dati all'interno della CPU, che ci consente di muovere il contenuto dei vari registri e tramite MAR e MDR accedere anche informazioni presenti nei vari dispositivi Slave.

Per muovere i dati lungo il Data Path ci servono, oltre i comandi all'ALU, i comandi per i registri:

- **Zsample**: comando per campionare il valore di uscita di esito dell'ALU nel bit Z dei CC;
- **R0in, R0out1, R0out2**: comandi per consentire al contenuto del registro R0 di essere campionato oppure essere emesso sugli Internal Bus di output out1 e out2;
- **PCin, PCout1**: comandi per campionare il valore ed emettere il contenuto del Program Counter;
- **IRin, IRout1**: comandi per campionare il valore ed emettere il contenuto dell'Instruction Register;

- **MARin, MARoutenable**: comandi in entrata e in uscita verso gli Slave;
- **MDRin, MDRout1, MDRout2, MDRoutenable**: come il MAR puo' parlare anche agli Slave.

Includiamo anche i comandi **MEMread** e **MEMwrite** del Control Bus per accedere a memoria.

Control Path di una semplicissima CPU

Col Data Path abbiamo visto come orchestrare i trasferimenti di dati nella CPU NS-0, vediamo ora come controllare tali trasferimenti mediante il Control Path.

Iniziamo dalla **fase di fetch** della prossima istruzione macchina e vediamo quale passo in sequenza deve essere fatto dalla CPU e quali comandi devono essere attivati:

- Passo S0: PCout1, ALUpass, MARin
  - Passiamo il valore del Program Counter nel registro di indirizzamanto, per consentire di indirizzare la cella di memoria contenente l'istruzione da acquisire;
- Passo S1: MARoutenable, MEMread, ALUnop
  - Emettiamo sull'Address Bus il contenuto nel MAR e dare alla memoria il comando di lettura (l'ALU non fa niente);
- Passo S2: MARoutenable, MEMread, MDRsample, ALUnop
  - Si conferma il comando precedente per dare il tempo alla memoria di trasmettere sul Data Bus il contenuto della cella indirizzata e con il comando MDRsample si fa in modo che questa istruzione da eseguire venga campionato nel MDR;
- Passo S3: MDRout1, ALUpass, IRin
  - Si trasferisce il contenuto del MDR nel registro istruzioni dove si potra' effettuare la decodifica;
- Passo S4: PCout1, ALUinc, PCin
  - Si intcrementa di 1 mediante l'ALU il contenuto del Program Counter per puntare all'istruzione macchina successiva (per far si' e' necessario che tutti i bistabili del registro PC siano Flip-Flop Master-Slave non trasparenti, in modo che il valore emesso in uscita e quello campionato in ingresso non si inseguano all'interno dei bistabili stessi).

Passiamo dunque alla **fase di decodifica**: in base all'opcode acquisito nella fase di fetch (che nella CPU NS-0 e' costituito dai 2 bit piu' significativi del registro IR) si decide quale tra le 4 istruzioni macchina deve essere eseguita.

Fase di execution.

Se l'istruzione decodificata e' la **LOAD**:

- Passo S5: IRout1, ALUpass, MARin

  - Portare i 14bit meno significativi dell'IR nel MAR in modo da poter indirizzare la cella da cui leggere il dato;
- Passo S6: MARoutenable, MEMread, ALUnop
- Passo S7: MARoutenable, MEMread, MDRsample, ALUnop
  - Lettura da memoria identica alla fase di fetch;
- Passo S8: MDRout1, ALUpass, ROin
  - Trasportare il contenuto del Memory Data Register in R0 che e' la destinazione del valore della cella letta.

Se l'istruzione decodificata e' la **STORE**:

- Passo S5: IRout1, ALUpass, MARin
  - Portare i 14bit meno significativi dell'IR nel MAR;
- Passo S6: R0out1, ALUpass, MDRin
  - Portare il contenuto di R0 nel Memory Data Register;
- Passo S7: MARoutenable, MEMwrite, MDRoutenable, ALUnop
  - Il MAR emette nell'Address Bus l'indirizzo da 14bit e contemporaneamente l'MDR emette nel Data Bus il dato da scrivere il memoria, dunque tramite il comando MEMwrite avviene la scrittura;
- Passo S8: MARoutenable, MEMwrite, MDRoutenable, ALUnop
  - Ripetere il passo una seconda volta per dare tempo al dispositivo di memoria di campionare il Data Bus e di scrivere tale valore nella cella indirizzata;

Se l'istruzione decodificata e' l'**ADD**:

- Passo S5: PCout1, ALUpass, MARin
- Passo S6: MARoutenable, MEMread, ALUnop
- Passo S7: MARoutenable, MEMread, MDRsample, ALUnop
- Passo S8: PCout1, ALUinc, PCin
  - La costante da sommare e' ancora presente in memoria, dobbiamo quindi ripetere una sorta di fase di fetch ovvero portare il PC nel registro di indirizzamento, fare una lettura da memoria, incrementare il PC per portarci alla prossima istruzione ed infine effettuare la somma vera e propria;
- Passo S9: MDRout1, R0out2, ALUadd, R0in, Zsample
  - Il registro MDR che e' la costante da sommare viene viene portato sul Internal Bus di output1, il contenuto di R0 va sul bus di output2, l'ALU somma i dati presenti ai suoi ingressi A e B e pone il risultato sull'Internal Bus di input e il registro CC campiona l'esito della somma per tenere traccia qualora il risultato fosse nullo.

Se l'istruzione decodificata e' **BRZ**:

- Passo S5: if(Z=1) IRout1, ALUpass, PCin
  - Se il registro CC ha il suo unico bit a 1 allora i 14bit di indirizzo presenti nel registro istruzioni mediante un passaggio verso l'ALU devono essere caricati nel Program Counter per saltare alla destinazione dell'istruzione di salto. Se nel CC c'e' 0, non si fa nulla e si procede

con il fetch dell'istruzione successiva come da normale comportamento di salto condizionato dove la condizione non e' verificata.

Struttura della CU

CU Cablata

Riassumiamo quali sono gli ingressi e le uscite della CU:

- **Ingressi**:
  - Codice operativo dell'istruzione acquisita durante la fase di fetch (nel caso della CPU NS-0, qualcosa che ci dica se dobbiamo fare Load, Store, Add, BranchZ);
  - Situazione del registro CC (nel nostro caso sapere se Z e' 0 oppure 1);
  - Lo stato di avanzamento nel Control Path, ossia il numero dello step a cui siamo arrivati per capire quale e' il prossimo da eseguire.
- **Uscite**:
  - Comandi ai registri della CPU (quei comandi che ci servono per muovere le informazioni tra i registri nel Data Path);
  - Comandi all'ALU;
  - Comandi al Control Bus per gestire le interazioni tra la CPU e i dispositivi Slave.

In una CU cablata per capire di quale istruzione si tratta, possiamo prendere i 2 bit piu' significativi dell'IR e inserirli come ingressi in una decoder 2 a 4 che attivi una delle 4 linee (L, S, A, B) a seconda dell'istruzione.

Per quanto riguarda i CC e' sufficiente far uscire il valore del bistabile Z.

Per conoscere lo stato di avanzamento possiamo realizzare un contatore che viene incrementato dal Clock della CPU e fare uso di un decoder che trasforma il valore conteggiato nell'accensione successiva di una delle 10 linee che ci dicono a quale step siamo arrivati nell'avanzamento della nostra sequenza di comandi. I comandi poi saranno inviati alle uscite, verso l'ALU, versi i registri o verso il Control Bus.

Nella realizzazione della CU cablata si segue un approccio HW. Se facciamo riferimento ale sequenze che indicano lo stato di avanzamento nel Control Path, ogni output della CU e' attivo in determinati step di esecuzione. Possiamo allora costruire un'espressione logica di tipo combinatorio per ogni output in funzione degli input elencati prima. Per la realizzazione di queste espressioni logiche usiamo un circuito detto **Logic Array**, cioe' un circuito in grado di sintetizzare diverse espressioni booleane. Tale circuito, realizzato su circuito integrato e' particolarmente veloce ed efficiente che e' in grado di fornire alla

velocita' necessaria per il funzionamento della CPU le uscite della CU in funzione dei suoi ingressi.

Alcuni esempi di **espressioni logiche**:

- PCout1: comando che dice al PC di emettere sull'Internal Bus di output1 il proprio contenuto avviene:
  - allo step S0;
  - allo step s4;
  - agli step S5 e S8 seoltanto se l'istruzione e' ADD
  - dunque l'espressione logica diviene: **PCout1 = S0 + S4 + (S5 + S8).A**
- **MARoutenable = S1 + S2 + S6.(L + A) + S7.(L + S + A) + S8.S**
- **IRout1 = S5.(L+S+B.Z)**
  - Il comando avviene al passo S5 dell'istruzione Branch se il CC di Z e' 1.
- **ALUmsb = S4 + (S8 + S9).A**
  - (Most Significant Bit deve essere 1)
- **ALUlsb = S0 + S3 + S5.(L+S+A+B.Z) + S6.S + S8.L + S9.A**
  - (Least Significant Bit deve essere 1)
- **MEMread = S1 + S2 + (S6 + S7).(L + A)**

Procedendo in modo analogo possiamo costruire tutte le espressioni logiche di tutti gli output della CU che stiamo realizzando.

Caratteristiche della CU Cablata:

- Una struttura particolarmente efficente:
  - La rete combinatoria Logic Array e' il modo piu' rapido per generare le uscite a partire dagli ingressi, dunque assicura la massima velocita' di esecuzione;
- Particolarmente adatta a CPU RISC con poche istruzioni:
  - La complessita' del Logic Array e' "dominabile", ed e' questo il motivo per cui inizialmente le macchine RISC hanno avuto un gran successo;
- Non e' di facile modifica:
  - Evoluzioni dell'architettura che portino a modifiche della struttura della CPU richiedono la riprogettazione completa della CU Cablata;
- Non e' particolarmente facile usare una CU Cablata per macchine CISC:
  - Il Logic Array che realizza le espressioni combinatorie aumenta esponenzialmente di complessita' col crescere degli ingressi e delle uscite;
- Solo recentemente, grezie all'evoluzione tecnologica del silicio, e' applicabile a CU complesse.

CU Microprogrammata

La differenza rispetto alla CU Cablata e' l'assenza tra gli input del passo al quale siamo arrivati. Rimangono gli ingressi:

- I bit del registro IR: opcode dell'istruzione acquisita durante il fetch (L, S, A, B);
- Situazione del registro CC (Z on/off).

Gli output rimangono invariati:

- Comandi ai registri;
- Comandi all'ALU;
- Comandi al Control Bus.

La CU Microprogrammata segue un approccio progettuale di tipo SW, ogni output e' attivati in base alle microistruzioni del microprogramma che viene eseguito. Dunque ogni istruzione del microprogramma produce comandi.

La memoria di microprogramma, tipicamente una ROM, contiene tutte le sequenze di passi, quindi il Control Path deve essere scritto sulle celle di questa ROM.

E siccome e' necessario ramificare il flusso di esecuzioned el microprogramma, e' presente una logica di salto che decide la prossima microistruzione da eseguire.

Principali caratteristiche:

- Ha una struttura particolarmente regolare
  - La ROM cambia di dimensioni in base alla complessita' del microprogramma
  - Il circuito, il microProgramCounter e la logica di gestione dei microSalti sono molto semplici;
- E' particolarmente adatta a CU complesse (CISC) quando la tecnologia HW non e' cosi' evoluta da consentire il progetto di CU cablate estremamente sofisticate
  - Si puo' inoltre pensare di avere un microAssemblyLanguage in modo da poter operare direttamente sulla CU;
- Particolarmente in voga negli anni '80;
- Ha problemi di velocita' di esecuzione dovuti alla presenza della memoria di microprogramma: e' sempre necessario un ciclo di lettura della prossima microistruzione prima di emettere i comandi;
- L'approccio cablato tende a soppiantare l'approccio microprogrammato non appena la tecnologia di integrazione lo consente.

Architettura dell'ALU

Compiti e organizzazione dell'ALU

L'ALU ha essenzialmente 3 compiti:

- Chiudere il Data Path tra gli Internal Bus della CPU;
- Svolgere operazioni logiche sugli operandi A e B:
  - AND bit a bit;
  - OR bit a bit;
  - NOT bit a bit.
- Svolgere operazioni aritmetiche sui numeri A e B:
  - Confronti;
  - Somme e sottrazioni in complemento a 2;
  - A volte moltiplicazioni e sottrazioni (in modulo e segno).

Ingressi e uscite:

- **A e B**: ingressi degli operandi il cui parallelismo e' tipicamente il parallelismo della CPU, ovvero A e B hanno ciascuno un numero di fili pari al numero dei bit dei registri di lavoro dei registri GPR della CPU;
- **R**: risultato (anche qui parallelismo della CPU);
- **C**: bit di comando il cui numero e' legato al numero di operazioni che l'ALU e' capace di svolgere (2^C = numero di operazioni disponibili);
- **F**: flag di esito, per ogni condizione che l'ALU e' in grado di verificare esiste un flag di esito (NZP nella CPU LC-2, Z nella CPU NS-0).

L'ALU e' al suo interno una **rete combinatoria** che:

- In base al comando C inviato:
  - Opera sugli operandi A e B;
  - Produce un risultato R;
  - Emette un flag di esito F.

Al suo interno puo' avere un comportamento sequenziale in alcune sue componenti (es. moltiplicatore) ma all'esterno presenta comunque un comportamento combinatorio, cioe': a fronte di richieste e di operandi, produca, in un tempo piu' o meno lungo (a seconda della complessita' dell'operazione svolta), il proprio risultato.

Circuiti sommatori

Abbiamo gia' visto i circuiti Half Adder e Full Adder.

Per sommare 2 numeri di n bit, ci basta concatenare n Full Adder:

- Ciascun Full Adder riceve i 2 bit di posto omologo nei due addendi A e B, cioe' i 2 bit coefficienti della stessa potenza di 2;
- Somma i 2 bit A e B per generare il risultato Sn (con posto omologo ai due addendi);
- Riceve in ingresso l'eventuale riporto della somma di peso inferiore;

- Genera il riporto per la somma dei bit di peso superiore.

In realta' il Full Adder in posizione 2^0 puo' essere tranquillamente un circuito Half Adder, poiche' non avra' mai riporto d'ingresso. Si inserisce comunque un Full Adder per avere un circuito piu' regolare da realizzare, vedendo quindi il Full Adder in 2^0 avere il bit di riporto collegato a massa.

Il problema principale di questa soluzione per la somma di 2 numeri a n bit, e' legato al fatto che ogni Full Adder e' una rete combinatoria a 2 livelli, realizzata nella forma ottimizzata "somma di prodotti" o "prodotto di somme".

Dunque per la somma di 2 numeri a n bit servono n Full Adder: ne consegue che la rete combinatoria risultate e' una rete a 2n livelli di porte logiche(il sommatore di peso massimo ha bisogno di ricevere in ingresso il riporto da parte del sommatore precedente, etc.), duqnue piuttosto lenta.

|Una possibile soluzione a questo problema e' l'espressione logica **Carry Look Ahead**.

Espressione logica del riporto (i+1)esimo: Ri+1 = AiRi + BiRi + AiBi Che puo' essere fattorizzata: Ri+1 = AiBi + (Ai + Bi)Ri

Possiamo indicare: Gi = AiBi: prodotto logico tra Ai e Bi => prende il nome di funzione **generate** poiche' genera un contributo al riporto Ri+1 con i due bit Ai e Bi; Pi = Ai + Bi: somma logica tra Ai e Bi => prende il nome di funzione **propagate** poiche' propaga al riporto Ri+1 il risultato proveniente dalle somme precedenti (rappresentato dalla presenza di Ri nel Full Adder di posto i).

Abbiamo dunque: Ri+1 = Gi + PiRi

Possiamo iterare questa espressione, tenendo conto che Ri e' dato dalla funzione generate di posto i-1 piu' la funzione propagate di posto i-1 per il riporto di posto i-1: Ri = Gi-1 + Pi-1Ri-1 Ri+1 = Gi + Pi(Gi-1 + Pi-1Ri-1) Ri+1 = Gi + PiGi-1 + PiPi-1Ri-1 ... Ri+1 = Gi + PiGi-1 + PiPi-1Gi-2 + PiPi-1...P1G0 + PiPi-1...P0R0

Se andiamo avanti fino a 0, vediamo che il Ri+1 e' la somma di tanti contribuiti che via via vedono crescere il numero di fattori in cascun prodotto logico ma in cui ogni termine e' sempre o una funzione propagate o una funzione generate. Duqnue ogni addendo di questa lunga somma e' sempre un prodotto di funzioni generate e propagate.

Il risultato e' che il circuito capace di calcolare il Carry secondo la tecnica del Carry Look Ahead e' una rete a 3 livelli di porte logiche capace di generare direttamente il riporto i+1esimo. E' una somma di due prodotti (2 livelli di porte logiche), ma poiche i termini della somma sono somme o prodotti dei bit di dato abbiamo bisogno di un altro livello di porte logiche. Si puo' dunque anticipare il riporto generando con soli 3 livelli un riporto in posizione qualsiasi nella somma di 2 numeri a n bit. I limiti sono solo tecnologici ed elettronici di una rete combinatoria con porte logiche a troppi ingressi per fare la somma finale di tutti gli addendi del Carry Look Ahead. I tipici circuiti Carry Look Ahead

lavorano ad anticipare il Carry in una sequenza di 8 bit o inferiore. Se pero' pernsiamo che il Carry di una sequenza ad 8 bit richiede 16 livelli nel sommatore normale e solo 3 con la tecnica del Carry Look Ahead, possiamo intuire l'enorme vantaggio in termini di velocita'.

Un'altra variante del sommatore a n bit e' il sommatore/sottrattore di numeri a n bit in complemento a 2: Se complichiamo un circuito Carry Look Ahead o un circuito facente uso dei Full Adder tradizionali, facendo si che tutti i bit del secondo addendo (B) non entrino direttamente nei Full Adder ma in una porta XOR che abbia come secondo ingresso un bit di controllo -ADD/SUB. Se il segnale -ADD/SUB vale 0, il valore di B viene propagato al Full Adder e viene inserito uno 0 nel riporto del primo Full Adder. Se il segnale -ADD/SUB vale 1 (vogliamo dunque fare una sottrazione): gli ingressi B di tutte le posizioni viene negato, dunque propagato **complementato** al Full Adder (abbiamo invertito tutti i bit di B). Il segnale -ADD/SUB a 1 entra anche come segnale di riporto del primo Full Adder, che significa di fatto fare il complemento a 2 del numero B, che vuol dire fare l'operazione aritmetica A-B.

Circuiti moltiplicatori

La moltiplicazione tra 2 numeri da n bit genera un prodotto con 2n bit. Si costruisce la matrice diagonale dei prodotti parziali:

- Dove il moltiplicatore vale 1 si copia il moltiplicando;
- Dove il moltiplicatore vale 0 si inseriscono gli zeri; Si effettua la somma per colonna dei prodotti parziali e ad ogni generazione di riporto si scrive un 1 nella colonna immediatamente piu' significativa.

I bit del moltiplicatore (mj) si devono propagare per riga. I bit del moltiplicando (qi) si devono propagare in diagonale.

La matrice di calcolo puo' essere realizzata da:

- Porta AND: che effettua il prodotto parziale del singolo bit del moltiplicando per il singolo bit del moltiplicatore;
- Sommare il prodotto parziale ottenuto precedentemente con il risultato parziale della somma in colonna tramite un Full Adder;
- Tenendo conto del riporto in ingresso e generando l'eventuale riporto in uscita.

La generazione di prodotti parziali richiede 1 livello di porte logiche AND. Ogni cella introduce ulteriori 2 livelli: il circuito di 2 livelli. Dopo aver completato la prima riga della matrice diagonale (n celle), i riporti devono discendere lungo la diagonale (n-1 celle). NlivelliTot = 1 + 2 x (n+(n-1)) = 4n -1

La complessita' del moltiplicatore cresce con il quadrato del numero di bit dei fattori.