Domande di calcolatori t

Calcolatori Elettronici

Alma Mater Studiorum – Università di Bologna (UNIBO)

11 pag.

DOMANDE DI CALCOLATORI T

Ingegneria Informatica 2014/2015
Andrea Hade

1 Split Cycle

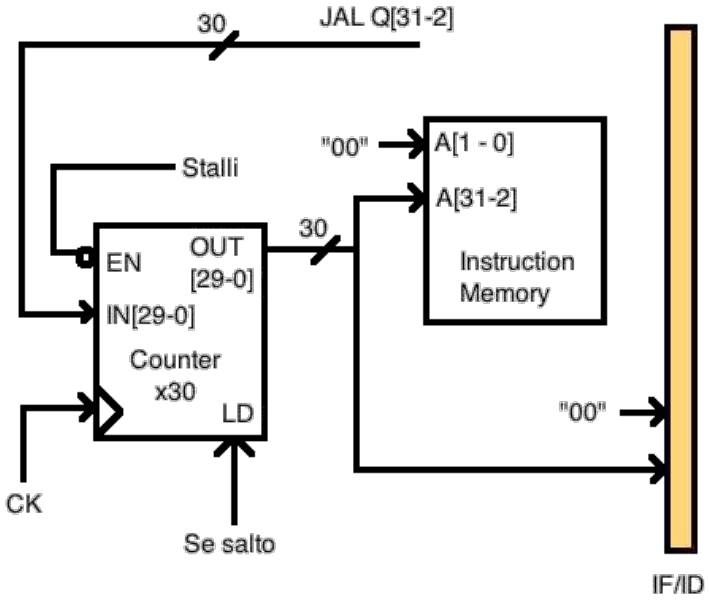
1.1 Cos'è e a cosa serve

Permette di eliminare le alee di dato (RAW) in alternativa al MUX davanti al RF. Consiste nello scrivere sul RF nel primo semiperiodo e leggere in esso nel secondo semiperiodo. In questo modo l'istruzione che si trova in ID non avrà mai registri obsoleti, perché sono stati aggiornati dalla fase di WB nel semiperiodo precedente. Si può realizzare attraverso un RF negative edge-triggered.

1.2 Svantaggi

L'operazione che prima si faceva in un ciclo di clock adesso va fatta in un mezzo, quindi la frequenza è doppia. Le altre soluzioni (es MUX) permettono invece di lavorare a frequenza standard, facendo tutto quello che fa lo split cycle: non è la soluzione migliore.

2 Contatore in IF al posto del MUX, PC e ADDER



I 30 bit di OUT li utilizzo per portare nella pipeline il PC + 4 per i salti.

3 Forwarding

3.1 Cos'è e a cosa serve

Permette di eliminare quasi tutte le alee di dato (o RAW) senza stallare la pipeline. Nelle fasi di IF e ID leggo il dato non aggiornato, ma nella fase di EX una rete combinatoria (FU) permette di fornire all'ALU il dato più aggiornato (attraverso dei MUX), in base a dei confronti. Il dato fornito può provenire:

- dal RF, quindi dalla fase di ID dell'istruzione attuale (forwarding non usato)
- dallo stadio di MEM dell'istruzione prima
- dallo stadio di WB di 2 istruzioni prima.

Se proviene dalla fase di WB è anche necessario intercettare il dato prima che venga scritto nel RF, attraverso un MUX così da fornirlo aggiornato al pipeline register ID/EX.

3.2 Forwarding Unit: cosa fa, che tipo di rete è, quali segnali entrano/escono

È l'unità che si occupa di realizzare il forwarding. Si tratta di una rete combinatoria che confronta il codice operativo e i registri **sorgente** delle istruzioni in EX con i codici operativi e i registri **destinazione** delle istruzioni in MEM e in WB. Se c'è un match tra le istruzioni (detto anche *hit*), il FU porta alla ALU il dato più aggiornato. Se tutte gli stadi hanno aggiornato il registro, chi ha il dato più aggiornato? La fase di MEM. Dalla FU escono i segnali per governare i MUX in entrata alla ALU così da scegliere il più aggiornato.

4 Se facessi a meno del FU e dello split cycle?

È necessario stallare la pipeline fino al WB dell'istruzione che scrive sul registro dal quale si vuole leggere (alea di dato o RAW). Come realizzo stallo? Uso un MUX e nel caso di stallo ricampiono lo stesso dato. In questo modo l'istruzione viene bloccata fino all'aggiornamento del RF, così da riprendere l'esecuzione.
Stallo vs NOP:

- Stallo → pipeline si blocca fino al WB dell'istruzione che ha causato l'alea, all'uscita continuano
- NOP → serve per uccidere le istruzioni sbagliate entrate nelle pipeline, all'uscita vengono perse

5 Alee nel DLX

Si presentano tutte le volte che un'istruzione non può essere eseguita correttamente.

5.1 Alee strutturali

Si verificano quando si tenta di accede contemporaneamente alla stessa risorsa, ovvero:

- IF e MEM accedono entrambi in memoria in un ciclo di clock, risolvo usando due cache (che fanno riferimento ad un'unica DRAM), due bus e due indirizzi diversi (instruction mem e data mem)
- Una LOAD seguita immediatamente da una STORE avendo un solo registro MDR causerebbe un sovrapporsi di dati, risolvo con due registri separati: LMRD e SMRD
- Ho bisogno di un ADDER per l'IF e un ALU per l'EX
- Doppia porta di uscita nel RF altrimenti sarebbe necessario sequenzializzare gli accessi

In altre parole si raddoppiano le risorse.

5.2 Alee di dato o RAW - Read After Write

Si verificano quando un dato dipende da un altro che non è stato ancora aggiornato. Ad esempio se un istruzione scrive su un registro che verrà utilizzato dalle istruzioni immediatamente successive, bisogna far sì che queste ricevano il registro aggiornato entro la fase EX (dove effettivamente lo usano). I metodi che si possono utilizzare sono:

- Forwarding:
  - Con FU e MUX davanti al RF
  - Con FU e split cycle
- Stallo: blocco l'istruzione in uno stadio per 1, 2 o 3 cicli di clock, fino alla modifica del RF

Se subito dopo l'istruzione LOAD su un registro, viene utilizzato quel registro si ha una particolare alea dato detta di LOAD che si può risolvere o con una delle soluzioni dette prima, oppure con una soluzione software detta **DELAYED LOAD**. Il compilatore sa che l'istruzione è una LOAD e quella successiva è problematica, quindi la sostituisce con un'istruzione utile (cioè che non modifica il codice e non usa il registro destinazione della LOAD), o se non la trova utilizza una NOP SW. In questo modo evito di stallare.

5.3 Alee di controllo

Si verificano in corrispondenza dei Branch (salti condizionati). Sono imprevedibili, il loro effetto è noto solo a run-time. Se è preso o no lo sappiamo solo in face di EX e potrebbero essere entrate delle istruzioni che non c'entrano nulla. Se il salto non è preso le istruzioni entrate vanno benissimo. I metodi per risolvere questo tipo di alee sono:

1. **ALWAYS STALL**: appena so che l'istruzione è un salto (lo so in ID: viene decodificata), lo considero sempre preso e stallo sempre fino al raggiungimento della fase di WB dell'istruzione di Branch (che insieme alle precedenti continua l'esecuzione, altrimenti **deadlock**: non avanza più nessuno e tutto si ferma). Soluzione poco efficiente, ma semplice da un punto di vista di HW.

2. **PREDICT NOT TAKEN**: considero che il salto non sia mai preso e nel caso sia preso uso delle NOP.

3. **BRANCH TARGET BUFFER (BTB)**: Una tabella ovvero una cache, memoria associativa molto veloce, costituita da comparatori ai quali forniamo l'indirizzo (detto TAG) e se coincide con uno della tabella la BTB fornisce un HIT e ci dice anche qual è l'indirizzo del prossimo fetch. Fornisce un MISS se non è nella tabella. Necessaria una cache per velocizzare e non allungare troppo i tempi di clock, perché comunque l'uso della BTB ha un costo. Esempio loop: ultima predizione sarà sbagliata per forza perché finisce il loop. Riassume la storia dei salti, realizzata durante la fase di fetch, per prevederli nella maniera più efficiente possibile e per non stallare (quindi va fatto entro la fase di fetch; ma come capisco senza la fare decodifica? Guardo l'indirizzo, se è nella tabella è per forza un salto). In caso di errore utilizzo delle NOP. Struttura BTB:
    a. Indirizzo dell'istruzione salto (detto TAG): massimo 30 e a multipli di 4 (altrimenti eccezione per non allineamento)
    b. Se taken o untaken: uso 2 bit, che codificano 4 stati, perché lo stato cambia dopo 2 errori consecutivi.
    c. Indirizzo di salto (o BTA – Branch Target Address)

4. **DELAYED BRANCH**: il compilatore sa che l'istruzione è un branch e cerca di sostituire le 3 istruzioni successive al branch con delle altre che non modificano l'esito del codice e devono essere sempre eseguite (tipicamente da sopra) e le mette dopo il branch. Così non vengono mai usate delle NOP HW e nel caso peggiore, cioè se non trova delle istruzioni utili, il compilatore mette delle NOP SW (diverse da quelle HW) da 1 a 3. Simile all'esecuzione fuori ordine dei calcolatori moderni.

5.4 NOP

Vengono messe nella pipeline dalla UDC tramite dei MUX e si utilizzano per cancellare le istruzioni sbagliate che sono entrate nella pipeline (nel caso di Branch):

3 NOP: uccido le istruzioni sbagliate (3) entrate nella pipeline, appena so che il salto è preso; gestione di salto il fase di MEM.
[elimino istruzione in if/id, id/ex, ex/mem]

2 NOP: anticipo la gestione del salto alla fase di EX per mettere una NOP in meno, collego l'uscita dell'ALU e il risultato della condizione del salto al MUX del fetch con il rispetto dei Tsu e Th => allungo tempo di clock (potrebbe andarci bene o no, in base agli obiettivi che abbiamo).
[elimino istruzione in if/id, id/ex]

1 NOP: vado a gestire il salto in EX (prima cosa critica) e bypasso il PC (seconda cosa critica), quindi alla memoria non arriva più l'uscita campionata da PC ma arriva in modo combinatorio l'indirizzo di destinazione (ovvero l'esito combinatorio di un confronto) ottenuto dall'ALU. Quindi ogni volta che un salto viene preso spreco solo l'istruzione successiva al salto ma allungherei di molto il tempo di clock (ancora più del caso di prima), cioè diminuirei la frequenza dell'intero sistema.
[elimino istruzione in id/ex]

NOP SW: Sono delle no-operation, ovvero del codice vero e proprio. Diverse dalle NOP HW (immesse nella pipeline con dei MUX gestiti dall'UDC distribuita).

5.5 Quali clock devo spegnere per stallare un'istruzione?

If/id, id/ex, ex/mem; mem/wb mi serve per scrivere il dato.

5.6 Soluzioni alea dato al RF nello stadio di ID?

Quelli dell'alea dato: Forwarding (FU + MUX o FU + split cycle) o stallo.

5.7 Jump vs Branch

	Jump	Branch
Cos'è?	Salto sempre taken	Salto taken o unteaken (=0?)
Cosa fa l'HW?	IF: IR<-M[PC] PC<-PC+4 ID: A<-RS1 B<-RS2 EX: x <- PC+(IR$_{15}$)^{16}##IR$_{15\ldots0}$[**JL**] x <- A [**JR, JLR**] MEM: PC<-x (= BTA) WB: **JR:** NOP; **JL, JLR:** R31<-PC	IF: IR<-M[PC] PC<-PC+4 ID: A<-RS1 B<-RS2 EX: x<- PC+(IR$_{15}$)^{16}##IR$_{15\ldots0}$ MEM: PC<-x WB: NOP: non modifico RF
Rimedio	ALWAYS STALL o delayed slot	Quelle sopra al punto 5.3

6 Interrupt

6.1 Cos'è? Che tipi? Come vengono gestiti?

L'interrupt è un evento imprevedibile, non sincronizzato col clock e che non scade. Se ho abilitate le interruzioni, le devo gestire attraverso l'interrupt handler. Dopo di che il codice deve riprendere la normale esecuzione. Al BOOT le interruzioni sono disabilitate. Vi sono 3 macro-gruppi di interruzioni:

1. **Interrupt hardware**: provengono dall'esterno della CPU e possono essere ignorabili/mascherabili, se arrivano dal pin **INT** (attraverso IEN: DLX e IF: Intel), o non ignorabili sul pin **NMI** (pochi casi, situazioni critiche come la caduta di tensione dell'alimentazione che non garantisce il corretto funzionamento delle reti, allora NMI scatta e mette tutto in sicurezza);
2. **Exception**: nascono da circuiti all'interno del sistema stesso e possono essere scatenati volontariamente da un'istruzione; esempi: divisione per 0, overflow, segmentation fault, breakpoint, page-fault;
3. **Interrupt software**: eventi indotti dal programmatore per richiedere al SO l'esecuzione di una system call (nel DLX tramite istruzione TRAP); non è inatteso ma è comunuqe un interrupt perrché usa la stessa modalità di trasferimento di controllo dell'int hw ed exception. La chiamata è anonima cioè non si specifica l'indirizzo della procedura chiamata.

6.2 Nel DLX c'è il PIC? Che tipo di gestione a Interrupt usa il DLX?

NO! Al suo posto c'è un software che gestisce ciò che dovrebbe fare il PIC. Il DLX gestisce l'I/O a interrupt con trasferimento di controllo NON vettorizzato. Il PIC è mappato in memoria e può avere un registro interno per capire quali interrupt sono attivi per generare l'interrupt handler più corretto (Intel), oppure il tutto fatto via software (DLX).

6.3 Come fa la CPU a capire che c'è un'interruzione? Come ritorna all'istruzione di prima?

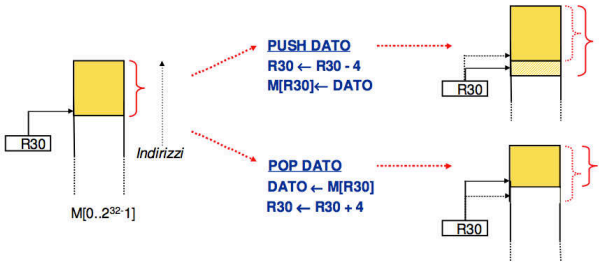
6.3.1 Senza annidamento

Nella fase di fetch il DLX controlla se è presente un interrupt da gestire (se Interrupt Enable – IEN = 1):

- se NON lo trova o gli interrupt non sono abilitati → fetch normale
- se lo trova:
  1. salvo in IAR il PC dell'istruzione successiva (PC+4, perché le istruzioni sono sempre lunghe 4 byte)
  2. il flusso del programma si sposta sul **handler a indirizzo PC = 0** relativo all'interrupt 1
  3. disabilito gli interrupt ponendo IEN = 0 (di default), quindi questo esegue fino al RFE (Return From Exception)
  4. ritorna ponendo IAR nel PC e riprendendo il normale flusso del programma

6.3.2 Con annidamento

Se invece abilito gli interrupt durante l'handler, succede che se mi arriva un altro interrupt più prioritario, l'indirizzo di ritorno verrebbe sovrascritto in IAR. È bene quindi salvare il vecchio IAR in un area della memoria gestita a stack via software. Quindi bisogna scrivere le funzioni di push/pop (tipo I), riservando un registro da usare come stack-pointer in cima allo stack,



per es R30. Le due funzioni andranno a leggere (pop) o scrivere (push) in memoria e aggiorneranno R30.

6.4 Dove l'IRET?

È un istruzione che si trova alla fine di una procedura di servizio che nell'8088/8086 ripristina dallo stack:

1. Il CS e IP della prossima istruzione da eseguire (nel normale flusso del main)
2. Registro dei FLAG

6.5 Cos'è l'INT TYPE e a cosa serve?

È un numero naturale ad 8 bit, quindi 256 interruzioni da 0 a 255. Esso che permette di trovare nella IVT la prima istruzione della procedura di servizio di ciascun interrupt. La CPU effettua un ciclo speciale di bus detto INTA (Aknowledge), a questo punto il PIC (Programmable Interrupt Controller) mette l'int-type sul bus dati. E la CPU tramite la IVT (in memoria ad indirizzo 0) ottiene l'indirizzo dell'handler e lo mette nel PC: PC ← IVT[int-type]. Il PIC gestisce anche le scale di priorità e ogni int-type occupa una dimensione di 4 byte.

6.6 IVT

Tabella che contiene i vari int-type con i relativi handler, in totale è composta da 256 elementi. La IVT inizia all'indirizzo fisico 0 della memoria. Si trova nei processori che sfruttano il controllo vettorizzato delle interruzioni (es 8088/8086).
Input: int-type
Output: IVT[int-type]

6.7 Perché quattro byte per ogni INT-TYPE?

Perché sono 2 byte per il selettore del handler e 2 byte per l'offeset del handler (= 4 byte).

6.8 NMI: cos'è? E' interna o esterna? Si trova nell'IVT? Fai un esempio

Si tratta di un particolare interrupt hardware, quindi esterno, non mascherabile **NMI**. Si verifica in pochi casi, in situazioni critiche: come la caduta di tensione dell'alimentazione. Essa non garantisce il corretto funzionamento delle reti, allora NMI scatta e mette tutto in sicurezza. Non si trova nella IVT.

6.9 Come maschero le interruzioni?

Attraverso due flag: Interrupt Enable = IEN = 0 (DLX), Interrupt Flag IF = 0 (Intel).

6.10 Cos'è IEN?

Interrupt Enable: permette di attivare o disattivare gli interrupt hardware nel DLX.

6.11 Se ho un solo interrupt?

Tutto normale, lo gestisco direttamente e basta.

6.12 Se ho gli interrupt disabilitate?

Non potendo sfruttare la gestione dell'I/O a interrupt utilizzo la **gestione a polling** (a controllo di programma): la CPU testa continuamente il registro di stato fino a quando BIF=1 (per leggere) o BOF=1 (epr scrivere), così da completare il trasferimento. Per fare handshake dovrei testare lo stato sul pin INTR. È poco efficiente.

7 Mapping

7.1 Mapping a 32 bit: spiegare e fare un esempio

Ci sono 4 bus, ciascuno è condizionato da un segnale che si chiama BE (bus enable o bank enable) che viene emesso dal processore. Se si vuole leggere una word sono tutti attivi, 16 bit: 2 BE adiacenti attivi (o asseriti), 1 byte: 1 BE asserito. Inoltre, non ha senso che 2 byte non adiacenti come be0 e be3 siano asseriti insieme. Se abbiamo una memoria logica da 128k e un sistema con un bus a 32 bit, avremo 4 memorie fisiche che realizzano la memoria logica e ciascuna è da 32 k: cioè ¼ della taglia che vogliamo realizzare. Supponiamo di avere 2 MB di EPROM, 32bit di bus dati e indirizzi. I 2 MB dobbiamo suddividerli in 4 blocchi da 512KB che mappiamo tra 4000 0000h e 401F FFFFh. NB: non è detto che esistano le memorie che avremmo

bisogno di utilizzare (le RAM sono più vincolate: multipli di 4 → RAM più piccola 128/4, più grande 128*4). 512 K = 19 bit di decodifica interna perché ho 2^{19}=512K oggetti indirizzabili. Le 4 memorie si attiverebbero tutte contemporaneamente, perché gli indirizzi sono tutti uguali, a meno del BE.
Se ci fosse una sola memoria non servirebbe la decodifica, facendo quella semplificata si userebbe tutto lo spazio di indirizzamento e ogni indirizzo sarebbe per lei.
Come collego il bus indirizzi/controllo/dati agli ingressi della memoria? RD è OE, CE tramite la decodifica, BA[20-**2**]→A[18-0] x 4 banchi, divido poi il bus dati:

D[7-0] → BD[7-0], D[15-8] → BD[15-8], D[16-23] → BD[16-23], D[31-24] → BD[31-24].

BA0 e **BA1** non sono usati perché non sfrutterei il parallelismo e i dati finirebbero sempre sulla stessa memoria. Per sfruttare il parallelismo devo buttare $\log_2$(Num_Bus) bit. Così, per esempio, ho che i primi 4 indirizzi logici corrispondono allo stesso indirizzo fisico. Poi tramite i BE seleziono il banco su cui voglio leggere/scrivere.

7.2 Mapping di una periferica agli stessi indirizzi di una memoria: come si gestisce?

Si può fare solo se si ha uno spazio di indirizzamento **separato** per le memorie e per i dispositivi di I/O. Si mappa tranquillamente la periferica e la CPU discrimina le varie modalità (I/O o memoria) con un pin aggiuntivo che si affianca al bus indirizzi (MEM/IO*). Oppure attraverso il bus dei controlli, come: MRDC, MWRC, IORDC, IOWRC.

7.3 Il DLX permette la lettura o la scrittura di dati disallineati?

Dipende. Visto che il DLX sfrutta un parallelismo a 32 bit, se leggo/scrivo:

- 8 bit → Sì, sempre: 1 byte è sempre allineato
- 16 bit → Sì, ad indirizzi pari
- 32 bit → No, non è possibile (le word devono essere sempre allineate)

7.4 Perché il DLX non accetta word disallineate?

Perché dovrebbe generare 2 indirizzi fisici diversi contemporaneamente: essendo una word da 32 bit, per sfruttare il parallelismo deve essere per forza a indirizzi multipli di 4.

7.5 Il DLX ha un doppio spazio di indirizzamento?

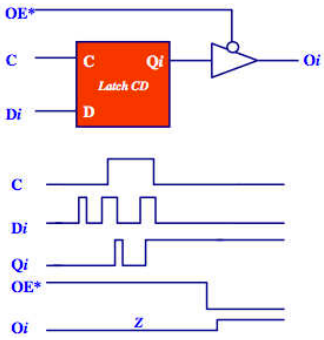
NO! È memory mapped I/O, cioè i dispositivi di I/O sono mappati nello stesso spazio di indirizzamento della memoria.

7.6 Dispositivi e sigle

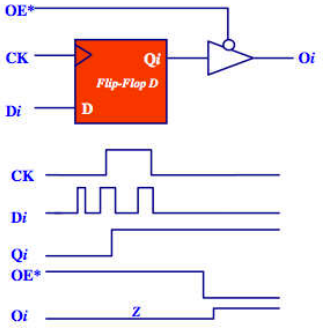
244: tri-state (con OE)

245: tri-state bidirezionale (transceiver, OE e DIR)

373: latch CD con uscite in 3-state
(campiona sempre se C=1)



374: FFD con uscite in 3-state
(campiona una volta: quando CK = 0→1)



8 RAM e Cache

8.1 Descrizione ai morsetti della RAM

Tutte le RAM per indirizzare le informazioni che contengono sfruttano un **decoder interno**, con i seguenti morsetti:

1. Indirizzi: A[k-1…0] che selezionano un determinato byte nel chip
2. Dato: D[7…0]
3. Chip select (!CS) è l'enable del decoder, in modo da attivare uno specifico chip
4. Comandi: !WR e !RD

Tutti i decoder ricevono lo stesso indirizzo e in base al !CS vengono attivati o meno.
Indirizzo: chip##byte_nel_chip

8.2 Perché le RAM vanno a multipli dispari di 2^n?

Perché hanno una decodifica di 2bit per riga e 2 bit per colonna. È quindi logico pensare che siano multipli di 4.

8.3 Passaggio da Ai -> BEi e viceversa: quando si può e come?

Ai → BEi non si può fare in quanto sappiamo solo l'indirizzo di partenza ma non quanti dati voglio leggere/scrivere
BEi → Ai si può visto che abbiamo 4 bus a 8 bit (32 totali): $A_0 = !BE_0\ !BE_1$, $A_1 = !BE_0\ !BE_2$

8.4 Cache

È un layer tra DRAM e CPU. Sfrutta il **principio di località**: più ci si avvicina al processore, più piccola e veloce è la cache (si abbassa il livello). Le informazioni più usate vengono messe più vicine al processore, ma col tempo si perderanno. Inoltre:

- CPU accede contemporaneamente a tutti i livelli di cache e ascolta la prima che mi risponde;
- i dati vengono salvati partendo dai livelli più vicini alla CPU e mano a mano che finisce lo spazio mi ci si allontana.

9 Gestione dell'I/O

9.1 Perché il segnale WR del FFD deve essere in logica negata?

Perché così il clock del FFD non arriva con dei glitch, visto che è campionato sul fronte positivo del clock, viene leggermente ritardato e si rispettano i tempi di set-up e di hold dei bus dati.

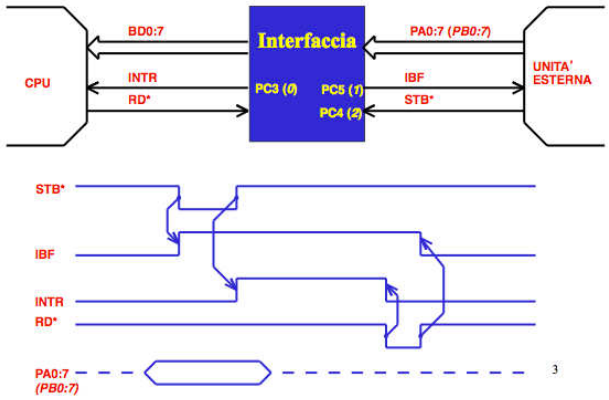
9.2 Protocollo di handshake e forme d'onda

9.2.1 Generale

È una stretta di mano, ovvero un metodo per sincronizzare due dispositivi che non hanno clock condiviso per trasferire dei dati. Il processore non viene disturbato direttamente, c'è un'**interfaccia** che fa da tramite con i pin classici ([A0, A1, A2] → in base a quanti registri interni ha, RD, WR, CS). Interfaccia può essere nell'UE ma non nella CPU e la si mappa nello spazio di indirizzamento. È costituita da un buffer (8 FFD) di 8 bit (per esempio) per l'input e un buffer da 8 bit per l'output.
NB: la CPU emette indirizzi, segnali di controllo, bus dati e all'esterno c'è una rete di decodifica.

9.2.2 INPUT handshake

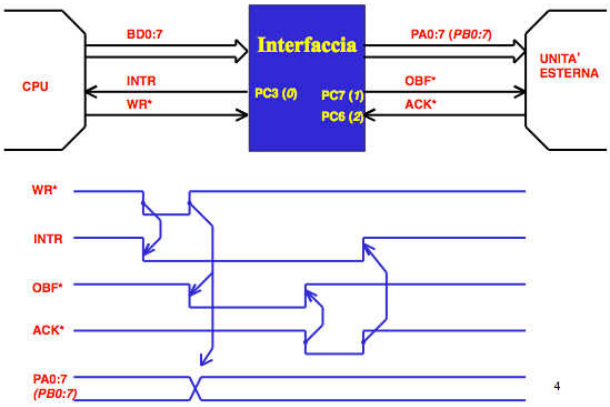
(UE => CPU)



1. UE controlla se **IBF** = 0, altrimenti aspetta per no sovrascrivere il dato già presente (IBF va a zero quando la CPU legge il dato)
2. Se IBF = 0, allora scrive **8 bit** di dato e attiva **STB** (è un read)
3. Interfaccia notifica tramite **INTR** (interrupt) alla CPU la presenza di un dato da leggere
4. La CPU durante la lettura attiva **RD** (INTR=0)
5. Finita la lettura IBF = 0 (per evitare problemi di lentezza dell'UE)

Se IBF = 1 e la CPU ha gli interrupt disabilitati? L'UE non scrive mai.

9.2.3 OUTPUT Handshake



(CPU => UE)

1. CPU viene **notificata** tramite INTR che **OBF**=0 e quindi può scrivere
2. CPU mette **8 bit** sul bus dati e attiva **WR**
3. Interfaccia attiva OBF
4. UE durante la lettura attiva **ACK** (OBF=0)
5. Fine lettura si disattivano ACK e INTR

9.3 Gestione a interrupt vs gestione a polling

La **gestione a polling** (a controllo di programma): la CPU testa continuamente il registro di stato fino a quando BIF=1 (per poter leggere) o BOF=1 (per poter scrivere), così da completare il trasferimento.
La **gestione a interrupt**: Quando l'interfaccia è pronta manda un interrupt alla CPU (ingresso INT), segnale INTR. La CPU interrompe l'attività e passa il controllo all'interrupt handler che realizza il traferimento del dato. Quando IBF =1, il PIC si occupa di rintracciare chi ha generato l'interrupt e lo comunica alla CPU. Lo fa attraverso 2 modalità:

1. Trasferimento di controllo **vettorizzato** (Intel): PIC manda int-type e la CPU andando nella IVT all'indice fornito (int-type) ottiene l'indirizzo della prima istruzione del handler.
2. Trasferimento di controllo **non vettorizzato** (DLX): Arriva un interrupt, CPU esegue sempre lo stesso handler (esempio all'indirizzo 0) ed esso si occupa di chiedere a polling alla periferica chi ha mandato l'interrupt.

9.4 Come gestisce l'I/O il DLX?

Ad INTERRUPT con trasferimento di controllo NON vettorizzato (vedi sopra).

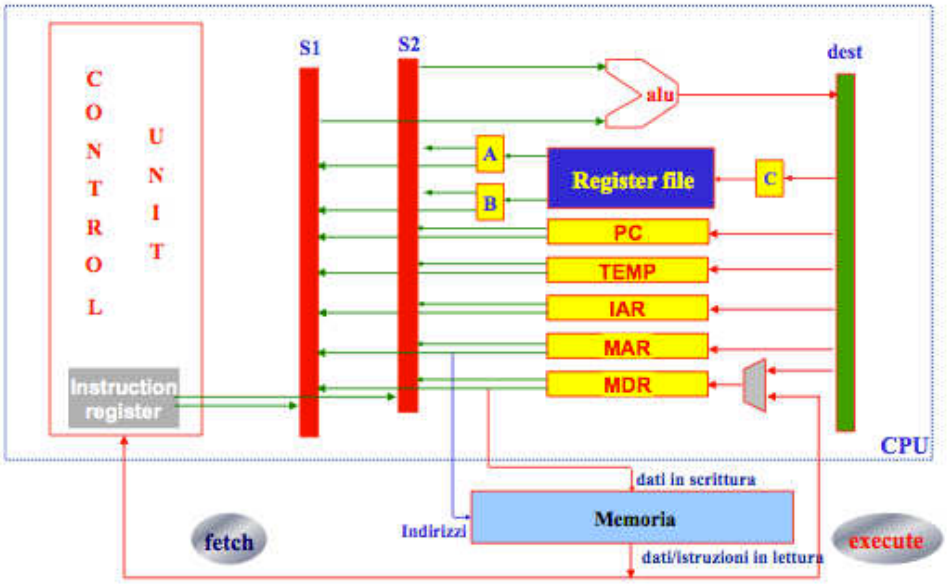
10 DLX

10.1 Sequenziale vs Pipeline: descrivere e fare gli schemi

Modalità indirizzamento: registro + offeset/immediato

10.1.1 Sequenziale

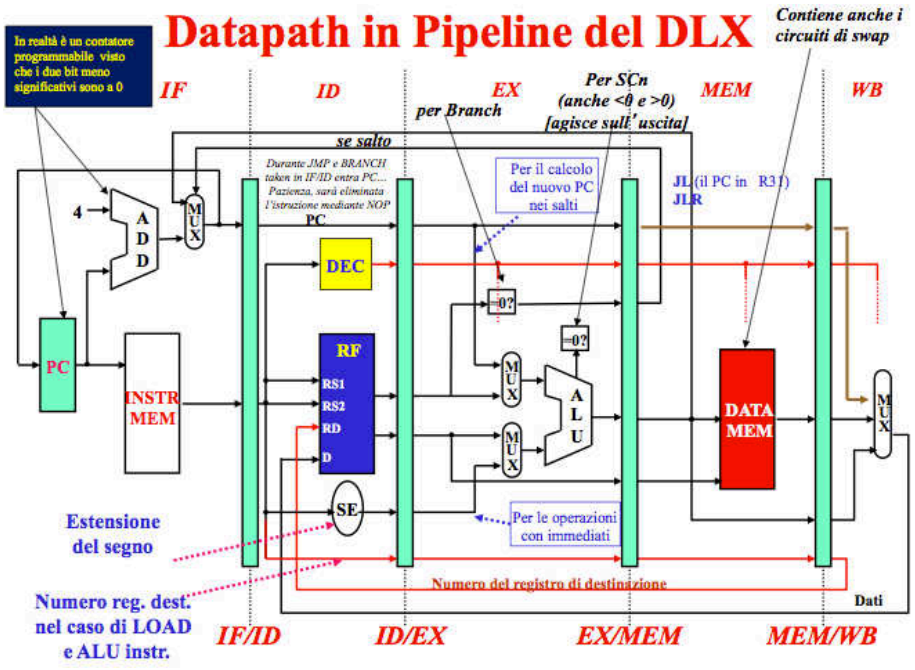
- 32 registri di uso generale (R0…R31) da 32 bit
- Bus dati a 32 bit → parallelismo a 32 bit, bus indirizzi a 32 bit (2^{32}=4GB)
- 32 registri Floating Point da 32 bit
- Set d'istruzioni ortogonale → massima libertà utilizzo dei registri
- R0 = 0 sempre, read-only
- R31 = salvo indirizzo di ritorno per le chiamate (istruzioni di tipo JL)
- Architettura **RISC** (R-R): istruzioni a lunghezza fissa di 32 bit (**4 byte** → ecco perché si fa PC+4), allineate (lettura di un'istruzione per ogni ciclo di bus) e con campi a lunghezza fissa
- 3 formati d'istruzione (codice operativo di 6 bit):
  - **I**mmediato: 2 registri da 5 bit (2^5=32 registri), immediato a 16 bit (Load/Store,Branch,JR)
  - **R**egistri: 3 registri da 5 bit, 11 bit per estensione cod. op. (istruzioni aritmetico-logiche)
  - **J**ump: 26 bit di immediato (JL)
- Gestione stack: non presente → implementabile via software
- Registro di flag: solo per floting point testabile a polling + 2 flag di uscita aggiornati ad ogni clock
- MOVS2I e MOVI2S (2=to) spostare registro speciale (es. IAR) da/verso registro general purpose
- UDC concentrata in un unico luogo.



Si tratta sempre di **schemi logici**.

10.1.2 Pipeline

- Esegue lo stesso set d'istruzioni
- Cambia la rete logica che la compone: migliora l'efficienza (CPI teorico = 1)
- Si basa sul concetto di catena di montaggio
- Latency: tempo per realizzare il prodotto (T_A) → non cambia
- Throughput: frequenza di completamento ($1/T_A$) → cambia
- Ci metto meno una volta che è entrata in regime (CPI = 1, per ogni stadio ho un'istruzione diversa)
- 5 sottoperazioni eseguibili in parallelo: IF, ID, EX, MEM, WB
- Il clock: vincolato dall'operazione più lunga
- Problemi: alee strutturali, dato e controllo
- UDC distribuita.



10.2 CISC/RISC

RISC (DLX):

- Set istruzioni ridotto (CPI=1)
- HW semplice
- Molti registri
- Unico modalità indirizzamento
- Istruzioni a lunghezza fissa: campi fissi
- Architettura load/store
- Interruzioni NON vettorizzate
- Registri ortogonali

CISC (Intel):

- Complesso set istruzioni
- HW costoso
- Pochi registri
- Più modalità indirizzamento
- Istruzione a lunghezza variabile
- Qualunque istruzione può accedere in memoria
- Interruzioni vettorizzate

RISC è più veloce!

10.3 Istruzioni RISC: vantaggi istruzioni lunghezza fissa e campi sempre stessa posizione

- Si fa tutto a priori
- Decodifica istruzioni veloce → so già dove sono
- Più veloce
- Anche un CISC trasforma in RISC appena può

10.4 Architettura Harvard: descrivi, perché si usa?

Si usa per evitare le alee strutturali: ho 2 cache separate (codice, dati), bus indirizzi/dati/controllo.

10.5 Perché MUX all'ingresso dell'ALU?

Per eseguire tutti i tipi di istruzioni. Il MUX sceglie tra:

- calcolo del PC nel caso dei salti (J)
- un o più registri provenienti dal RF (R)
- le operazioni con immediato (I)

10.6 Schema a blocchi del RF: uno semplificato