

I 5 PASSI BASE PER IMPLEMENTARE UN ISA

- 1) INSTRUCTION FETCH: PRELEVO L'ISTRUZIONE DA ESEGUIRE ED INCREMENTO IL PC;
- 2) INSTRUCTION DECODE: DECODIFICO L'ISTRUZIONE E LEGGO GLI EVENTUALI OPERANDI DAL REGISTER FILE;
- 3) EXECUTION: USO LA ALU PER IL CALCOLO (OPERAZIONI ARITMETICO LOGICA/INDIRIZZO DI MEMORIA PER LOAD/STORE);
- 4) MEMORY: ACCEDO ALLA MEMORIA DATI (RAM) SE LOAD E STORE;
- 5) WRITE BACK: SCRIVO IL RISULTATO (INDIETRO) NEL REGISTER FILE SE SONO OPERAZIONI ARITMETICO LOGICA.

LA 4 E LA 5 SONO DUNQUE OZIONALI.

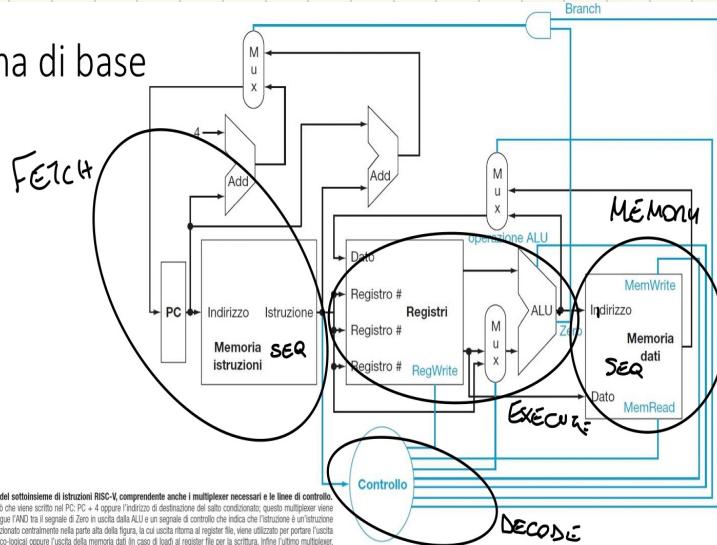
NELLA NOSTRA IMPLEMENTAZIONE CONSIDEREREMO SOLO: LOAD, STORE, ADD, SUB, AND, OR, BEQ.

Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo SB	immed[12,10:5]	rs2	rs1	funz3	immed[4:1,11]	codop	Istruzioni di salto condizionato
Tipo UJ	immediato[20, 10:1, 11, 19:12]				rd	codop	Istruzioni di salto incondizionato
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

Queste no

Figura 2.19 Formati delle istruzioni RISC-V.

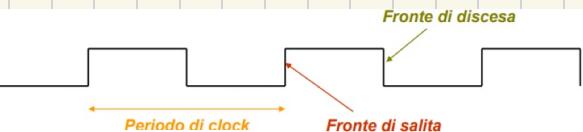
Schema di base



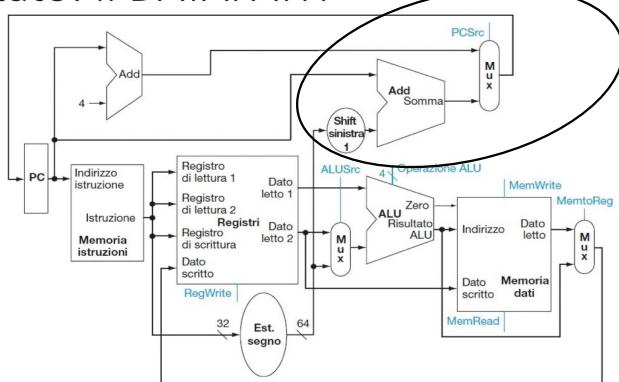
SCHEMA BASE, PER L'ESECUZIONE DELLE NOSTRE ISTRUZIONI.
VENGONO PRELEVATE LE ISTRUZIONI DAL PC (CHE VIENE INCREMENTATO), L'ISTRUZIONE VIENE PRELEVATA E RICONOSCIUTA, VENGONO EFFETTUATI I SALTI E TUTTE LE ALTRE OPERAZIONI.
IL MUX IN ALTO DETERMINA COSA VIENE SCRITTO NEL PC, QUELLO IN BASSO, DETERMINA LA PROVENIENZA DEL SECONDO OPERANDO.

TEMPORIZZAZIONE

QUANDO PROCESSIAMO UN DATO ALL'INTERNO DI UNA RETE COMBINATORIA, BISOGNA ATTENDERE, CHE QUESTO SIA VALIDO, IL VALORE DI UN ELEMENTO DI STATO, È AGGIORNATO IN CORRISPONDENZA DI UN FRONTE DI SALITA. IL TEMPO NECESSARIO PER LA PROPAGAZIONE DI QUESTI SEGNALI, È DETERMINATO DAL PERIODO DI CLOCK.



Risultato: il DATAPATH



NEL DATAPATH, A DIFFERENZA DI QUELLO BASE, CON L'AGGIUNTA DI UN MUX, CI CONSENTE DI ESEGUIRE LE NOSTRE OPERAZIONI CON UN SINGOLO CICLO DI CLOCK.

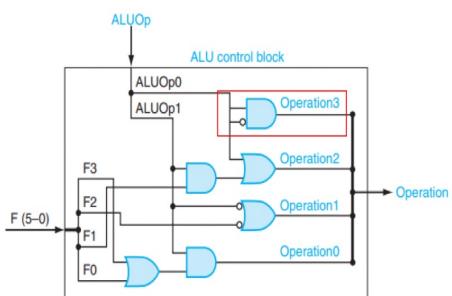
Figura 4.11 Una semplice unità di elaborazione per il nucleo dell'architettura RISC-V che combina tra loro gli elementi richiesti dalle istruzioni appartenenti ai diversi tipi. I componenti sono mostrati nelle Figure 4.6, 4.9 e 4.10. Questa unità di elaborazione può eseguire le istruzioni di base (load/store di un registro, operazioni con la ALU e salti condizionati) in un singolo ciclo di clock. Per integrare i salti condizionati si è resa necessaria solamente l'aggiunta di un altro multiplexer.

Controllo

Figura 4.13 Tabella della verità dei 4 bit di controllo della ALU (detti "operazione"). Gli ingressi della tabella della verità sono i campi ALUOp e i campi funz. Vengono riportate solamente le righe della tabella per cui i segnali di controllo della ALU devono essere asserti. Sono mostrate anche alcune righe associate a valori indifferenti: per esempio, ALUOp non utilizza il codice 11, quindi la tabella della verità conterrà le combinazioni 1X e X1 di ALUOp anziché 10 e 01. Anche se mostriamo tutti e dieci i bit dei campi funz, notate che gli unici bit che assumono valori diversi per le quattro istruzioni di formato R sono i bit 30, 14, 13 e 12. Quindi, occorrono solamente questi quattro bit dei campi funzione come input all'unità di controllo della ALU.

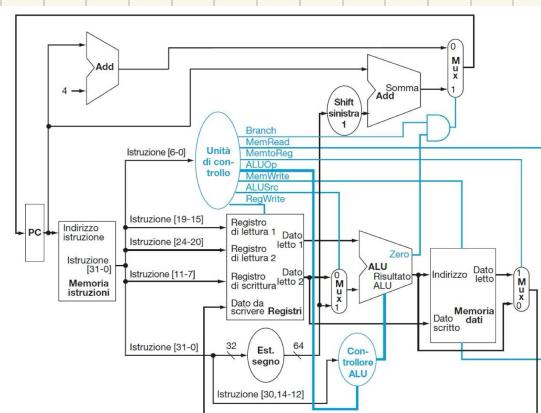
Codice operativo istruzione	ALUOp	Operazione eseguita dall'istruzione	Campo funz7	Campo funz3	Operazione dell'ALU	Ingresso di controllo alla ALU
ld	00	load di 1 parola doppia	XXXXXX	XXX	somma	0010
sd	00	store di 1 parola doppia	XXXXXX	XXX	somma	0010
beq	01	salto condizionato all'uguaglianza	XXXXXX	XXX	sottrazione	0110
Tipo R	10	add	0000000	000	somma	0010
Tipo R	10	sub	0100000	000	sottrazione	0110
Tipo R	10	and	0000000	111	AND	0000
Tipo R	10	or	0000000	110	OR	0001

ALUOp		Campo funz7							Campo funz3			Operazione
ALUOp1	ALUOp2	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001



QUESTO È IL CIRCUITO RELATIVO AL CONTROLLO DELLA ALU RIPORTATO QUI SOPRA.

UNITÀ DI CONTROLLO



Nome del segnale	Effetto quando non asserto	Effetto quando asserto
RegWrite	Nullo	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 12 bit del campo immediato dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nullo	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea "Dato letto"
MemWrite	Nullo	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea "Dato scritto"
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

Figura 4.16 Effetto di ciascuno dei sette segnali di controllo. Quando il segnale di controllo a 1 bit di un multiplexer a due vie è assertivo, il multiplexer seleziona l'ingresso etichettato con 1; in caso contrario, cioè se il segnale di controllo non è assertivo, il multiplexer seleziona l'ingresso etichettato con 0. Si ricordi che tutti gli elementi di stato ricevono il clock come ingresso implicito e che il clock controlla le operazioni di scrittura. Far commutare il clock mediante un segnale esterno a un elemento di stato potrebbe causare problemi di temporizzazione (vedi l'Appendice A 6 per la discussione di questo problema).

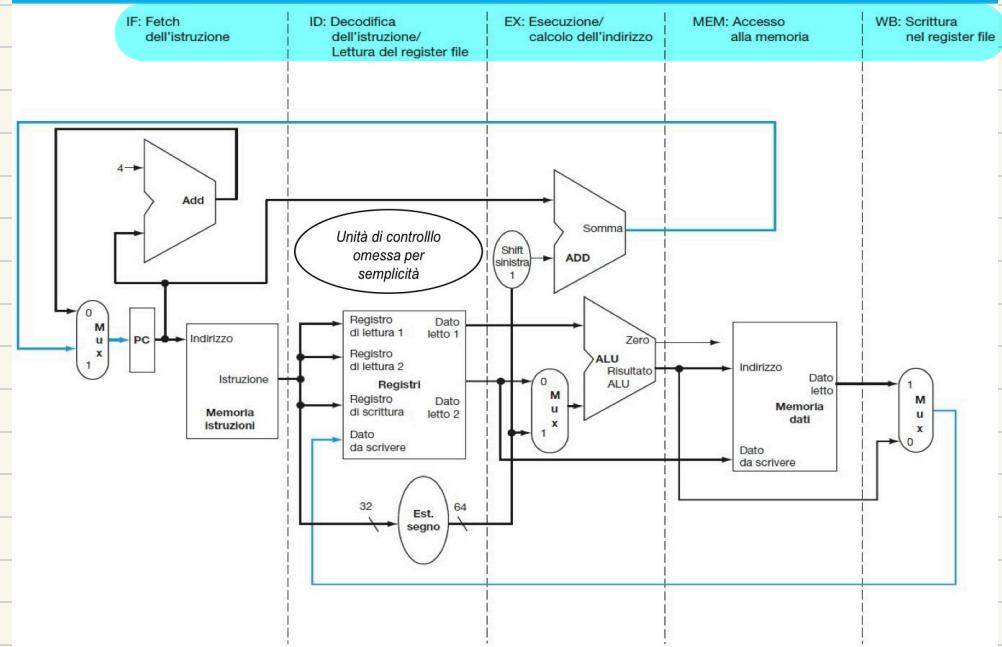
Istruzione	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Tipo R	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

**TABELLA PER
IL CONTROLLORE
DELLA ALU.**
**I SOLI INGRESSI
SONO ALUOP E
I CAMPI FUNZ.
NEI CAMPI FUNZ,
SONO NECESSARI
SOLO: 30, 14, 13, 12
ESIMO BIT, GLI ALTRI
RIMANGONO INVARIATI.**

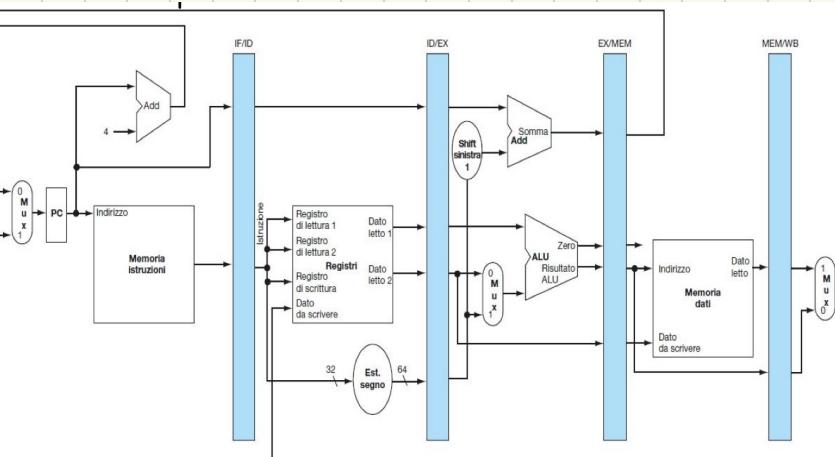
Figura 4.17 L'unità di elaborazione costituita finora, con la sua unità di controllo. L'input dell'unità di controllo è il campo 7 bit dell'istruzione che costituisce il codice operativo. L'output dell'unità di controllo è formato da due segnali a 1 bit utilizzati per controllare un multiplexer (ALUSrc e MemtoReg), tre segnali a 1 bit per controllare lettura e scrittura del register file e della memoria dati (RegWrite, MemRead e MemWrite), un segnale a 1 bit utilizzato come segnale di controllo per i salti condizionati (Branch) e un segnale di controllo a 2 bit per la ALU (ALUOp). Una porta AND viene utilizzata per combinare il segnale di Branch e l'uscita Zero della ALU; l'uscita di questa porta determina da dove prendere il valore successivo del PC e genera il segnale di controllo PCSrc per il multiplexer posizionato in alto (Figura 4.15). Si noti che PCSrc viene derivato e non proviene direttamente dall'unità di controllo; per questo motivo, metteremo di indicarlo nelle figure successive.

CPU CON PIPELINE IL PIPELING NON SERVE A DIMINUIRE I TEMPI DI ESECUZIONE, BEN SI' DI AUMENTARE IL THROUGHPUT; QUINDI I VARI STADI LAVORERANNO SEPARATAMENTE UTILIZZANDO RISORSE DIFFERENTI.

Idea di base: suddivisione della CPU in stadi



REGISTRI PER LA PIPELINE



NECESSITANO DUNQUE DI REGISTRI AGGIUNTIVI, PER PRESERVARE IL VALORE DELL'ISTRUZIONE IN ESECUZIONE, DEL PC, E DI ALCUNI VALORI SPECIFICI CHE NECESSITANO LE VARIE FASI DELLA PIPELINE.

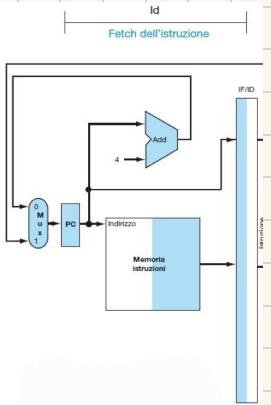
64 99 36
BIT

261
BIT

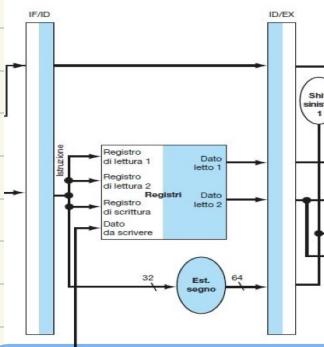
198
BIT

133
BIT

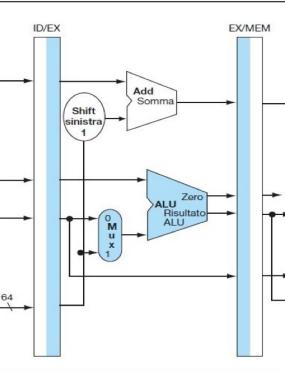
QUI VIENE COUNTATO IL +5 BIT DEL RD.



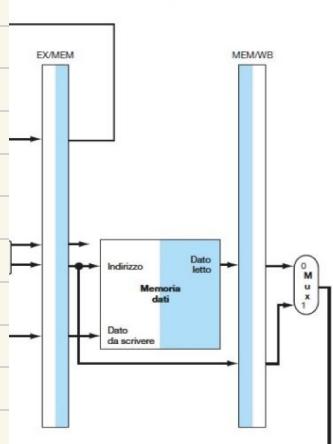
FETCH DELL'ISTRUZIONE : IL PC VIENE INCREMENTATO E RISCRITTO, IL VECCHIO VALORE VA SALVATO IN IF/ID, PERCHÉ PUÒ ESSERE RICHIESTO PIÙ AVANTI, AD ESEMPIO IN UNA BEQ.



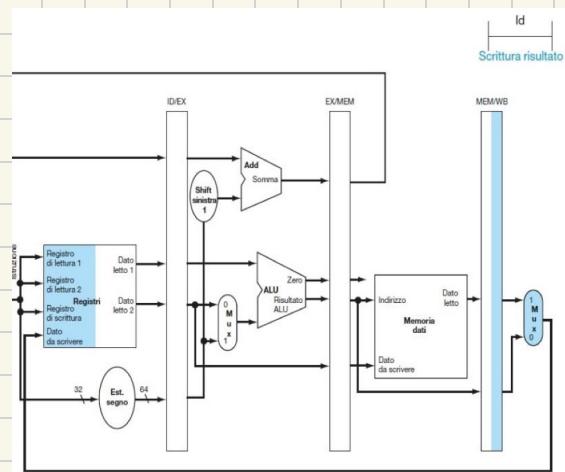
DECODIFICA DELL'ISTRUZIONE : ENTRAMBI I DUE REGISTRI VENGONO LETTI E SCRITTI NELLA PIPELINE ID/EX, INSIEME ALL'ESTENSORE DEL SEGNO



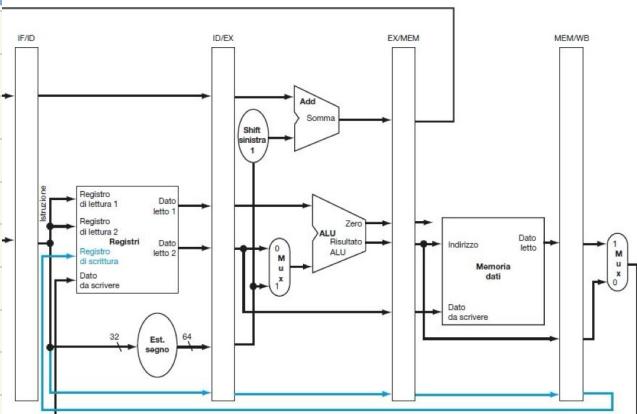
ESECUZIONE ARITMETICA O CALCOLO INDIRIZZO : IL RISULTATO DEL CALCOLO DELLE ALU, VIENE INSERITO IN EX/MEM



ACCESSO ALLA MEMORIA: IL DATO LETTO, USANDO IL RISULTATO COME INDIRIZZO, E IL RISULTATO STESSO ENTRANO NELLA PIPELINE MEM/WB



SCRITTURA NEL REGISTER FILE:
IN CASO SI DEBBA INSERIRE UN RISULTATO NEL REGISTER FILE, QUEST'ULTIMO VIENE LETTO E SCRITTO NEL REGISTER FILE.

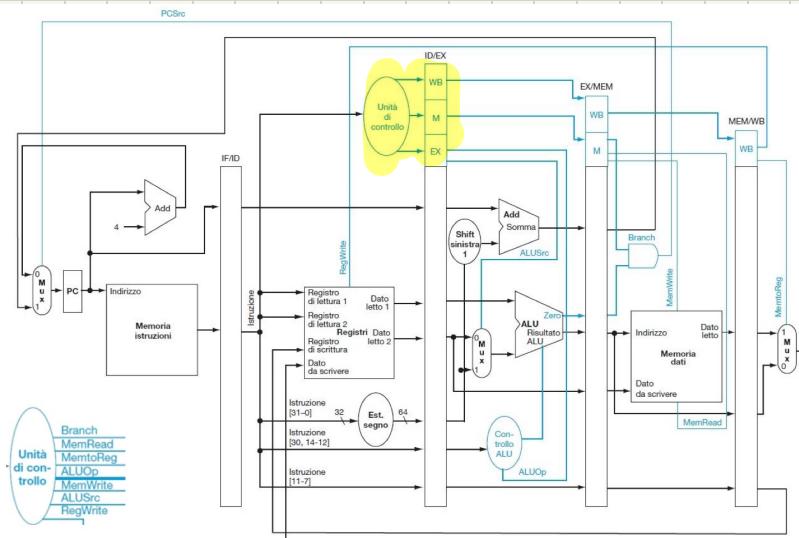


LA FIGURA VIENE MODIFICATA PER LA MANCANZA DEL REGISTRO DI DESTINAZIONE (S BIT), CHE DEVONO ESSERE PRESI DAL MEM/WB (VISTO CHE SI È PROTRATTO FINO ALL'ULTIMO).

ALL'INTERNO DELLA PIPELINE DOBBIANO AGGIUNGERE L'UNITÀ DI CONTROLLO.

Istruzione	Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo		Segnali di controllo dello stadio di accesso alla memoria dati			Segnali di controllo dello stadio di scrittura	
	ALUOp	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
Tipo R	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Figura 4.47 I segnali di controllo sono gli stessi di Figura 4.18, ma qui sono suddivisi in tre gruppi, ciascuno attivo in uno degli ultimi tre stadi della pipeline.

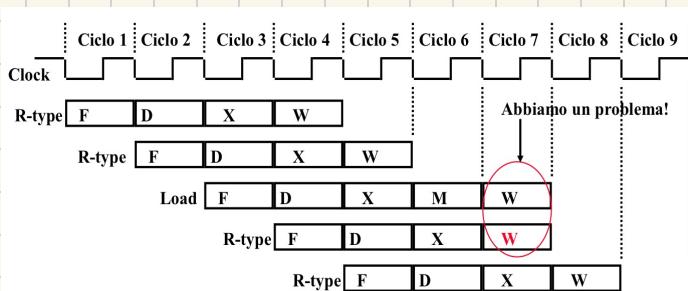


OCCHIO DUNQUE ESTENDERE LA PIPELINE, DALLO STADIO DI ID/EX.

QUESTO SCHEMA PERO' PRESENTA UN PROBLEMA NELLA BEQ, E BISOGNA FAR IL FLUSH DELLE 3 ISTRUZIONI PRECEDENTI.

CRITICITÀ STRUTTURALE

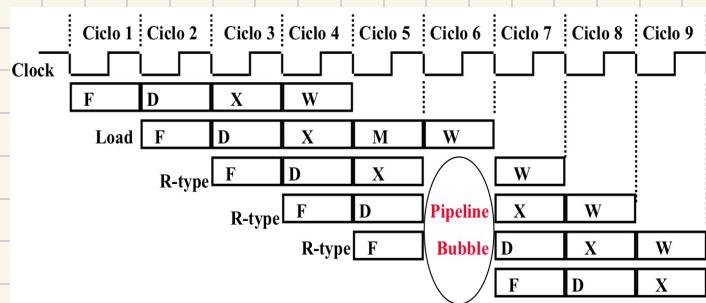
SUPPONIAMO DI AVERE ISTRUZIONI DI TIPO R A 6 CICLI.



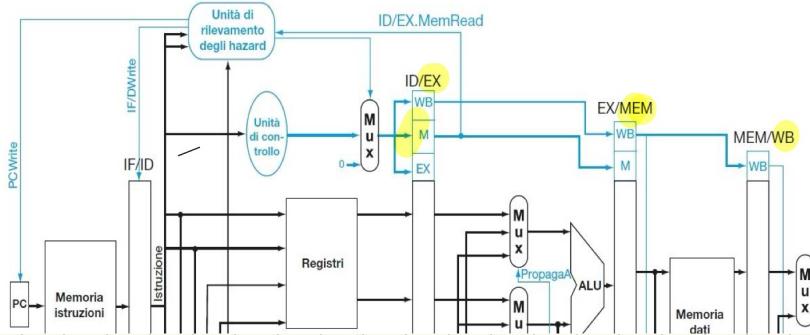
DUE ISTRUZIONI TENTANO DI SCRIVERE NEI REGISTRI CONTEMPORANEAMENTE, MA LA PORTA DI SCRITTURA È UNICA, DUNQUE ABBIANO UNA CRITICITÀ STRUTTURALE.

ESISTONO DUE MODI PER RISOLVERE LE CRITICITÀ NELLA PIPELINE:

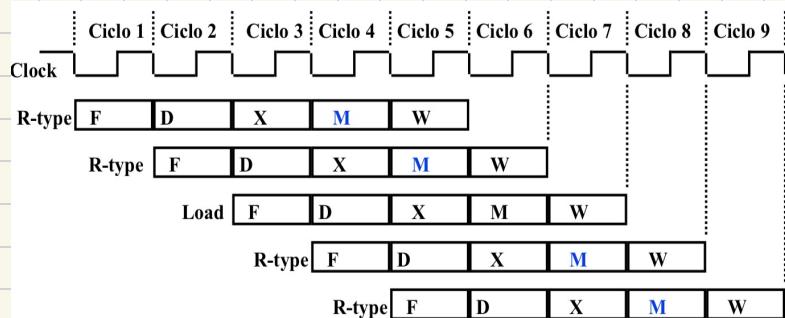
1) INSERIMENTO DI UNA BUBBLE



NOP (NOT OPERATION): SE IMPOSTO A 0 I 7 SEGNAli DI CONTROLLO, GLI STADI EX, MEM, WB
CREO UN ISTRUZIONE CHE NON FA NULLA Bisognerà RIPETERE LO STADIO IF/ID, PERCHÉ BLOCCO IL
PC E LA SCRITTURA DI IF/ID.

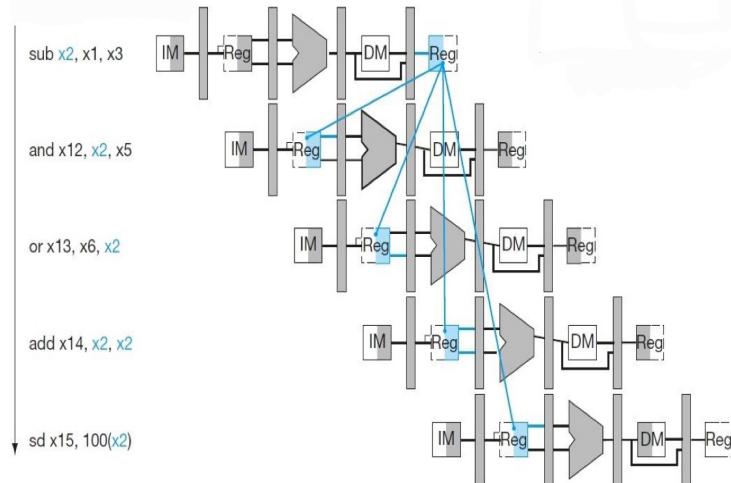


2) RIMANDARE LA SCRITTURA DEL REGISTRO PER UN CICLO (ISTRUZIONI R): VIENE AGGIUNTA LA MEM,
CHE NELLE ISTRUZIONI DI TIPO R NON FARANNO NULLA. IL RISC-V ADOTTÀ QUESTA SOLUZIONE, INFATTI
NON PRESENTA CRITICITÀ STRUTTURALI.



CRITICITÀ SUI DATI

SI TRATTA DEL TENTATIVO DI UTILIZZARE UN'UNITÀ PRIMA CHE SIA PRONTA, AD ESEMPIO UN'ISTRUZIONE DIPENDE DAL RISULTATO DELL'ISTRUZIONE PRECEDENTE, ANCH'ESSA IN PIPELINE CHE NON HA ANCORA SCRITTO NEL REGISTER FILE.



I CALCOLI SAREBBERO SBAGLIATI FINO ALLA ADD.

CLASSIFICAZIONE CRITICITÀ SUI DATI

- **Row (or Flow dependency)**: UN'ISTRUZIONE SUCCESSIVA LEGGE UN REGISTRO PRIMA CHE L'ISTRUZIONE PRECEDENTE LO ABbia SCRITTO.

- **WAW (or Anti dependency)**: UN'ISTRUZIONE SUCCESSIVA CERCA DI SCRIVERE UN REGISTRO CHE È STATO LETTO DALL'ISTRUZIONE PRECEDENTE.

- **WAW (o Output dependency)**: DUE ISTRUZIONI CONSECUTIVE, SCRIVONO SULLO STESO REGISTRO.

SOLUZIONI PER LE CRITICITÀ SUI DATI

1) INSERISCO DELLE BOLLE

2) PROPAGO IL DATO TRA DUE STADI EX DELLA PIPELINE.

3) RIORDINO LE ISTRUZIONI CHE NON SONO DIPENDENTI FRA LORO.

PROPAGAZIONE

Ordine di esecuzione del programma (sequenza delle istruzioni)

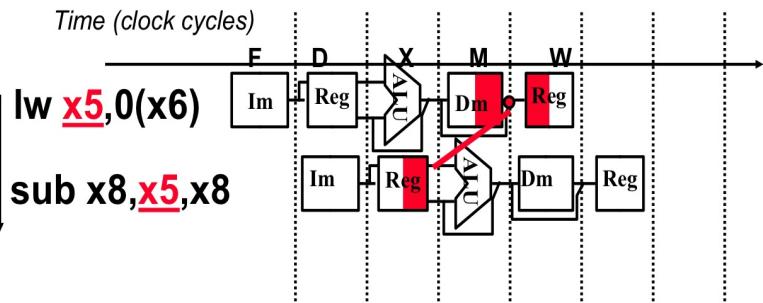
add x1, x2, x3



sub x4, x1, x5

QUESTO NON FUNZIONA PER LA LOAD.

LE ISTRUZIONI CHE DIPENDONO DALLA LOAD, DEVONO ESSERE RITARDATE/MANDATE IN STALLO VIA HARDWARE (INTERLOCK)



RIORDINO

- | | |
|------------------------------|------------------------------|
| 1. ld x1, 0(x31) // load b | 1. ld x1, 0(x31) // load b |
| 2. ld x2, 8(x31) // load e | 2. ld x2, 8(x31) // load e |
| 3. add x3, x1, x2 // a=b+e | 5. ld x4, 16(x31) // load f |
| 4. sd x3, 24(x31) // store a | 3. add x3, x1, x2 // a=b+e |
| 5. ld x4, 16(x31) // load f | 4. sd x3, 24(x31) // store a |
| 6. add x5, x1, x4 // c=b+f | 6. add x5, x1, x4 // c=b+f |
| 7. sd x5, 32(x31) // store c | 7. sd x5, 32(x31) // store c |

LA 3 DIPENDE DALLA 2, LA PROPAGAZIONE ELIMINA LA DIPENDENZA DALLA 1, E ANCHE LA 6 DIPENDE DALLA 5.

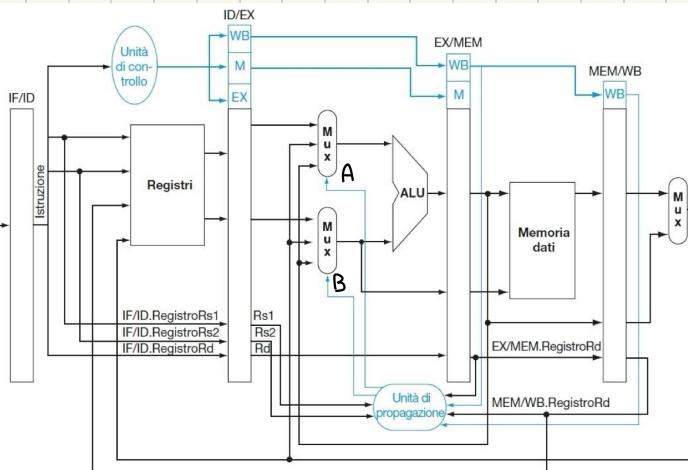
RILEVAMENTO DELLE DIPENDENZE

1a) EX/MEM. REGISTRO RD = ID/EX. REGISTRO RS1 2a) MEM/WB. REGISTRO RD = ID/EX. REGISTRO RS1

1b) EX/MEM. REGISTRO RD = ID/EX. REGISTRO RS2 2b) MEM/WB. REGISTRO RD = ID/EX. REGISTRO RS2

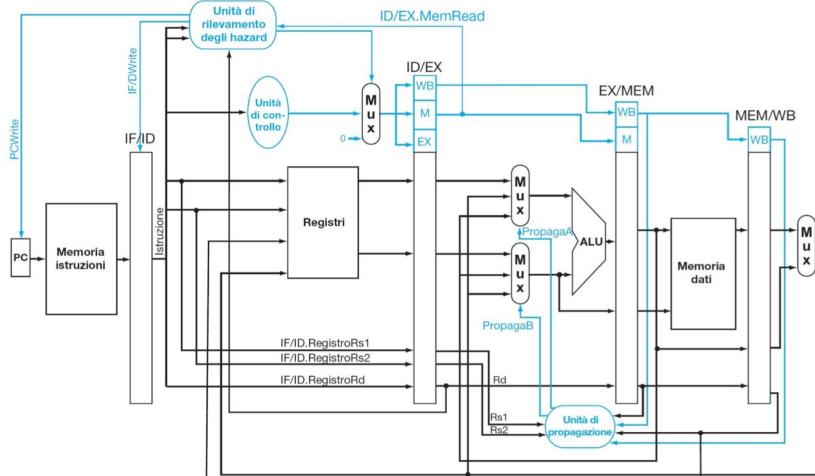
IMPLEMENTAZIONE DI UNA SOLUZIONE

Controllo multiplexer	Sorgente	Spiegazione
PropagaA = 00	ID/EX	Il primo operando della ALU proviene dal register file
PropagaA = 10	EX/MEM	Il primo operando della ALU viene propagato dal risultato della ALU nel ciclo di clock precedente
PropagaA = 01	MEM/WB	Il primo operando della ALU viene propagato dalla memoria dati o da un precedente risultato della ALU
PropagaB = 00	ID/EX	Il secondo operando della ALU proviene dal register file
PropagaB = 10	EX/MEM	Il secondo operando della ALU viene propagato dal risultato della ALU nel ciclo di clock precedente
PropagaB = 01	MEM/WB	Il secondo operando della ALU è propagato dalla memoria dati o da un precedente risultato della ALU



QUESTA È LA NOSTRA UNITÀ DI ELABORAZIONE, DOPO LE MODIFICA PER RISOLVERE GLI HAZARD, TRAMITE LA PROPAGAZIONE. LA FIGURA RAPPRESENTA UNO SCHEMA STILIZZATO IN CUI MANCANO I SALTI E L'ESTENSORE DEL SEGNO.

MA PER LA LW BISOGNA INSERIRE UNA BOLLA, NECESSITIAMO DI UN'UNITÀ RILEVAMENTO HAZARD CHE GENERA UNA NOP, QUANDO ID/EX.RD = IF/ID.RS1 OR .RS2.



SCHEMA AGGIORNATO!

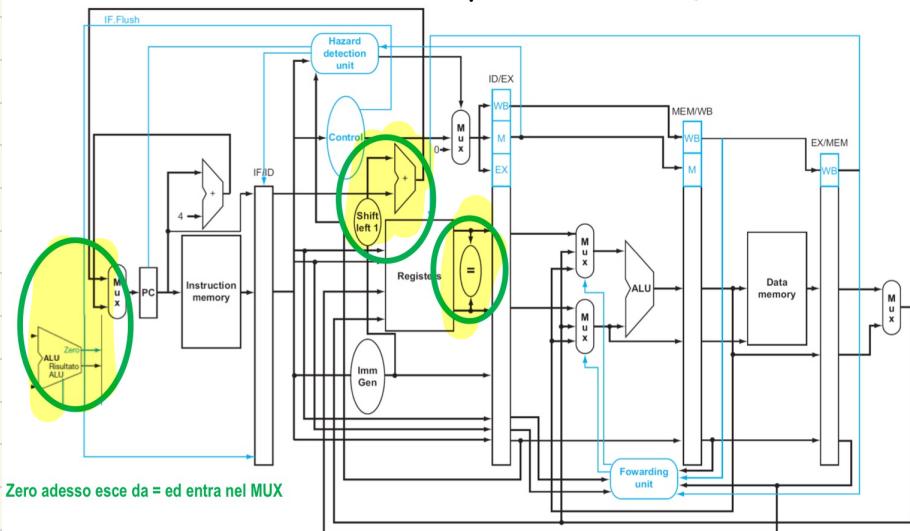
Criticità sul controllo

Si tratta del tentativo di saltare pur avendo caricato le istruzioni del ramo non preso. Per risolverlo, o aspettiamo o modifichiamo l'architettura. Nello schema finora esaminato, la decisione di un salto viene presa alla fine dello stadio Mem, perciò le istruzioni PC+6, PC+8, PC+12 dovranno essere sostituite con una NOP in caso il branch saltasse in una zona differente.

Le soluzioni sono due (ma non risolvono completamente il problema):

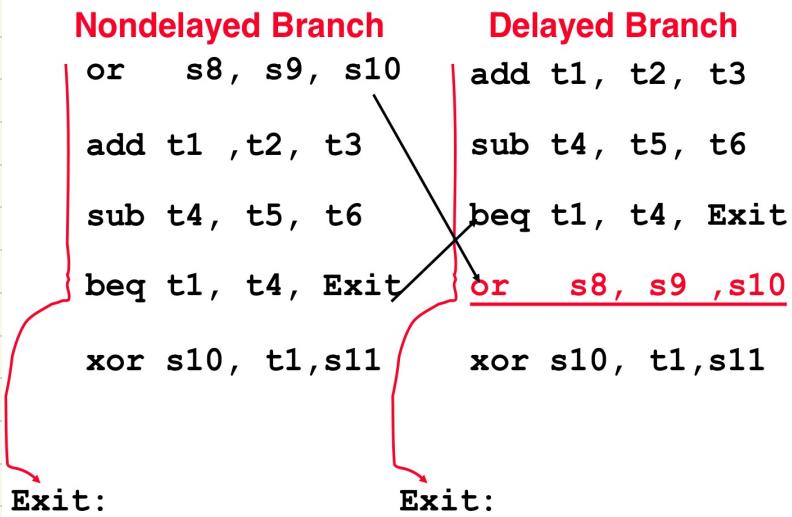
- 1) Cerco di predire quale ramo prenderò
- 2) Riduco la penalità associata al salto anticipando la scelta.

#1 ottimizzazione, anticipo il salto allo stadio ID



#2 ottimizzazione, ridefinizione del salto, ovvero, sia che si faccia, sia se non si faccia il salto, viene eseguita l'istruzione che segue immediatamente il salto (branch-delay-slot).

Osservazioni sul Branch-Delay-Slot: caso peggiore è l'inserimento di una NOP, caso migliore trovo un'istruzione che può essere inserita nello slot senza influenzare il programma (procedura effettuata dal compilatore).



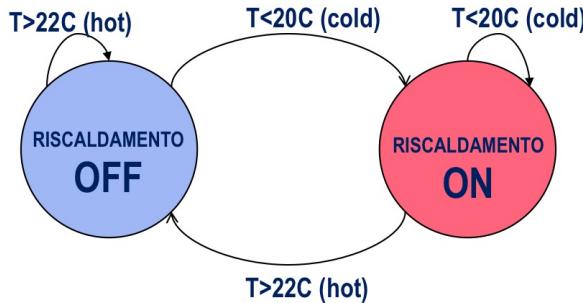
PREDIZIONE DEL SALTO

SI POSSONO UTILIZZARE UNO O DUE BIT SU UNA PARTE DELL'INDIRIZZO DELL'ISTRUZIONE PER RICORDARMI SE HO SALTATO OPURE NO (CON DUE BIT HO PIÙ POSSIBILITÀ DI ACCURATEZZA 80%).

PARTE D'AGOSTINO IN PAUSA

IN SEGUITO PARTE DELZANDO

MACCHINE A STATI FINITI

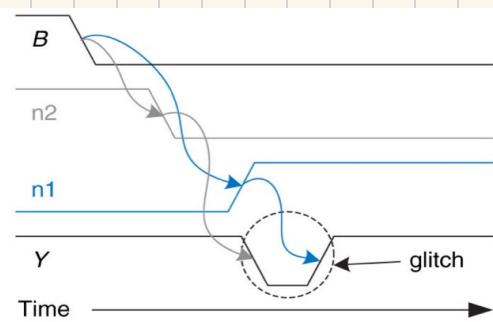
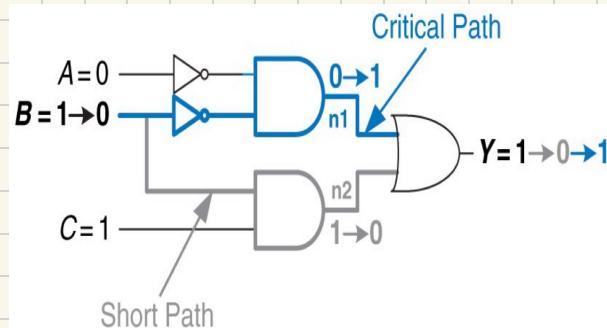


UN INSIEME DI STATI E TRANSIZIONI, CI SI SPOSTA FRA UNO STATO E UN ALTRO SOLO SE LA CONDIZIONE È RISPETTATA.

LE MACCHINE A STATI SONO COMPLETE, CIOÈ TUTTI GLI STATI DEFINISCONO TRANSIZIONI PER TUTTI GLI INPUT, E SONO DETERMINISTICHE CIOÈ CON UN CERTO DATO SEGUO SEMPRE UN CERTO ARCO. INOLTRE LE MACCHINE A STATI POSSONO ESSERE IMPLEMENTATE TRAMITE RETI SEQUENZIALI.

ALEE / GLITCH / HAZARD

PUÒ ACCADERE CHE VARIANDO UN INGRESSO SI CREINO VARIAZIONI MULTIPLE DI USCITA, È IMPORTANTE SAPER ANALIZZARE IL COMPORTAMENTO TEMPORALE DELLA RETE LOGICA.

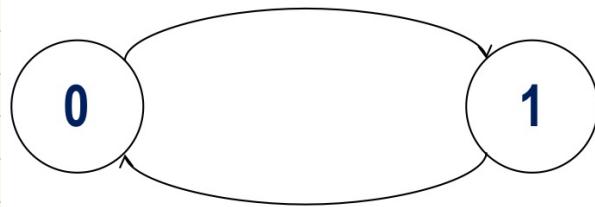


LE RETI SEQUENZIALI POSSONO ESSERE:

SINCRONE (RICHIEDE UN SEGNALE DI CLOCK);

ASINCRONE (NON RICHIEDE IL CLOCK).

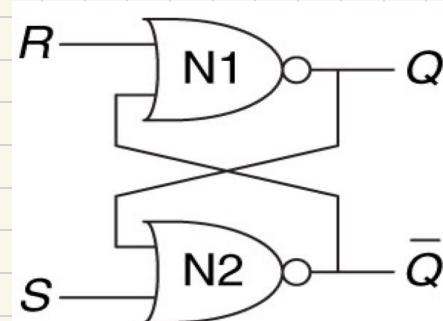
COME SI MEMORIZZA UN BIT?



SEMPLICEMENTE È UNA RETE A DUE STATI.

LATCH SR

USA UNA LOGICA ASINCRONA, ED È L'ESEMPIO DI CIRCUITO DI MEMORIA AD 1 BIT PIÙ SEMPLICE, L'USCITA DIPENDE DAI SEGNALI IN INGRESSO E DAI SEGNALI PRECEDENTEMENTE PROCESSATI.

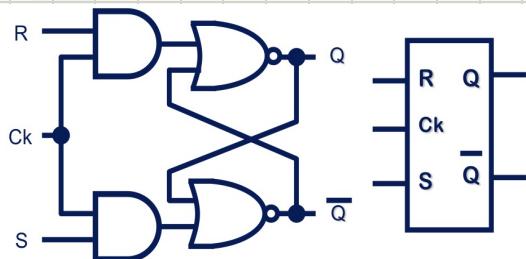


R SI DICE "RESET" E S SI DICE "SET", QUINDI QUANDO $R=1$ RESETTO $Q=0$, QUANDO $S=1$ SETTO $Q=1$. QUANDO SIA $R=Q=0$, SI DICE "CONSERVA" OVVERO NON FA NÉ UN SET, NE UN RESET. $R=Q=1$ NON SI PUÒ FARE.

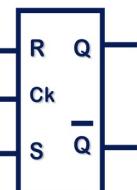
S	R	Q	/Q
0	0	Q	/Q
0	1	0	1
1	0	1	0
1	1	---	---

S-R CLOCKED

PER FARE IN MODO CHE S E R VENGANO ASCOLTIATI SOLO IN DETERMINATI ISTANTI, UTILIZZIAMO UN CLOCK ($CK=1$).



Schema logico



Simbolo

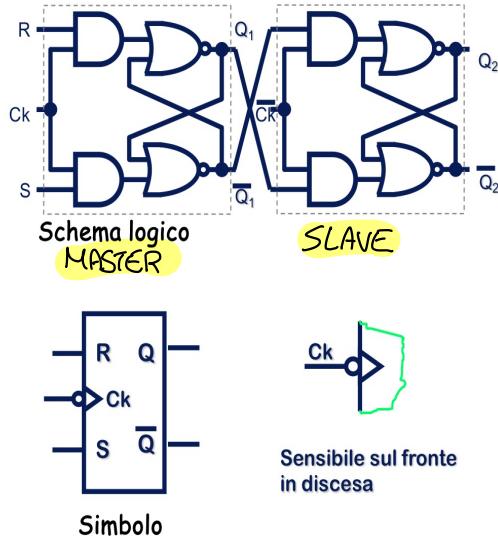
Ck	S	R	Q	/Q
1	0	0	Q	/Q
1	0	1	0	1
1	1	0	1	0
1	1	1	---	---
0	x	x	Q	/Q

S-R EDGE TRIGGERED

IN QUESTO CASO DIAMO IMPORTANZA SUL FRONTE DI DISCESA.

Ck	S	R	Q	/Q
0	0	0	Q	/Q
0	1	0	0	1
1	0	1	1	0
1	1	1	---	--
0	X	X	Q	/Q
1	X	X	Q	/Q
X	X	X	Q	/Q

Tabella di Verità



CK=1, MASTER È SENSIBILE A S E R, SLAVE MANTIENE IL VALORE;

CK=0, MASTER MANTIENE IL VALORE (Q₁), SLAVE PRENDE IN INPUT Q₂=Q₁;

FRONTE CK=1 → 0, MASTER TRASMETTE IL DATO SALVATO (QUANDO ERA CK=1), SLAVE MEMORIZZA;

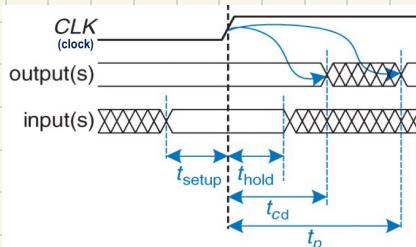
FRONTE CK=0 → 1, MASTER RIMANE ATTIVO, SLAVE LASCIA INVARIATO IL DATO SALVATO.

TIMING

BISOGNA TENERE CONTO DEI TEMPI DI PROPAGAZIONE DEL SEGNALE FRA INGRESSO E USCITA.

IL T_P È IL TEMPO NECESSARIO AFFINCHÉ L'USCITA SIA STABILE DAL MOMENTO IN CUI È VARIATO L'INGRESSO.

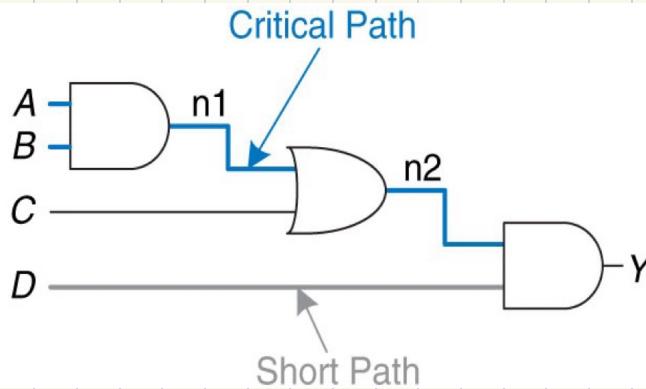
DATO CHE I SEGNALI SR DEVONO ESSERE STABILI DURANTE IL FRONTE DEL CLOCK, VANNO DUNQUE PREPARATI PRIMA (TSETUP), DOPO IL FRONTE I SEGNALI ATTENDERANNO (THOLD) PRIMA DI POTER VARIARE ULTERIORMENTE.



TCD (TEMPO DI CONTAMINAZIONE) È IL TEMPO TRA IL VARIAMENTO DI INGRESSO E IL VALORE DI USCITA.

Critical Path

È IL CAMMINO PIÙ LUNGO FRA UN INGRESSO E UN USCITA.



D-LATCH

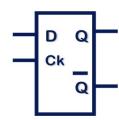
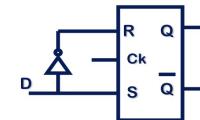
(ONVERO COSA DEVE MEMORIZZARE)

IL SEGNALE D CONTROLLO COSA DEVE ESSERE L'USCITA, IL SEGNALE CLK CONTROLLO QUANDO LO STATO DEVE ESSERE CAMBIATO.

(ONVERO QUANDO DEVO MEMORIZZARE)

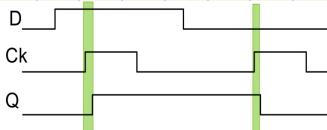
D	Ck	Q
X	0	Q
0	1	0
1	1	1

Tabella di Verità



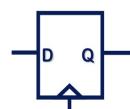
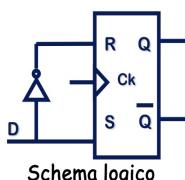
D-EDGE TRIGGERED

L'USCITA CAMBIA SOLO SUL FRONTE DI SALITA.



D	Ck	Q
X	1	Q
X	0	Q
X	-	Q
0	-	0
1	-	1

Tabella di Verità

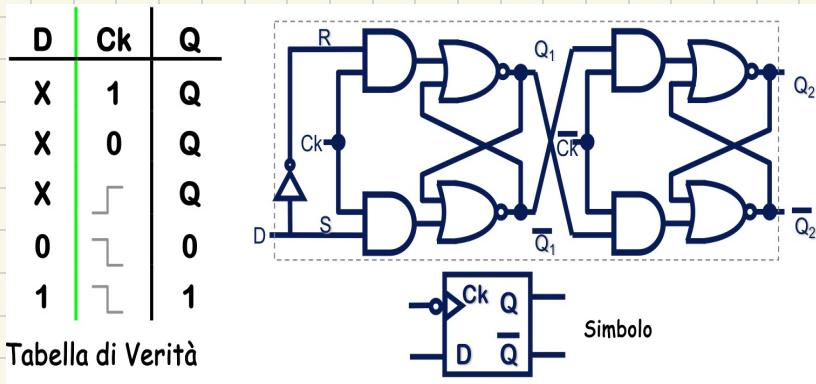


Simbolo del D-Edge-Triggered

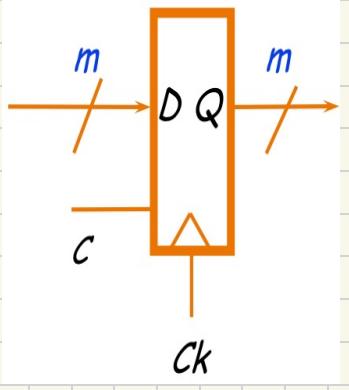
Layout simbolico: 44 transistors
(42 per SR-ET + 2 per NOT)

FLIP FLOP D

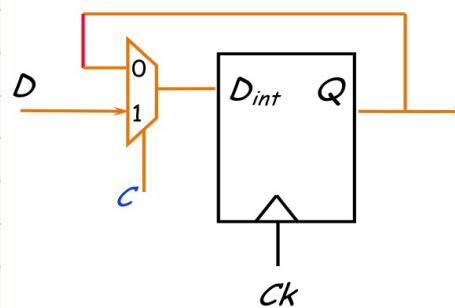
IN QUESTO CASO VOGLIAMO SENSIBILITÀ SOLO SUL FRONTE DI DISCESA.



LOGICA DI REGISTER FILE

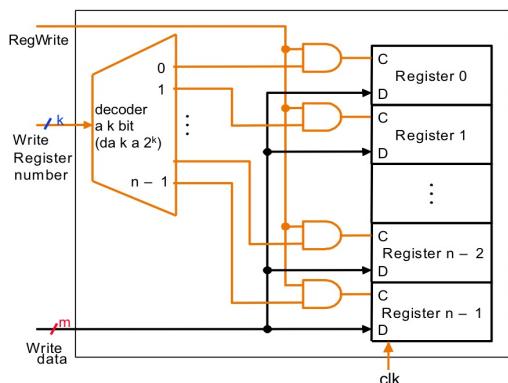


QUESTA È LA CELLA DI UN SINGOLO REGISTRO A m BIT.

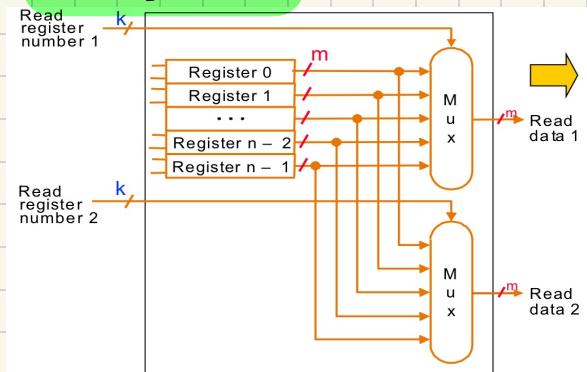


QUESTO È LA CELLA DI UN SINGOLO REGISTRO A 1 BIT.

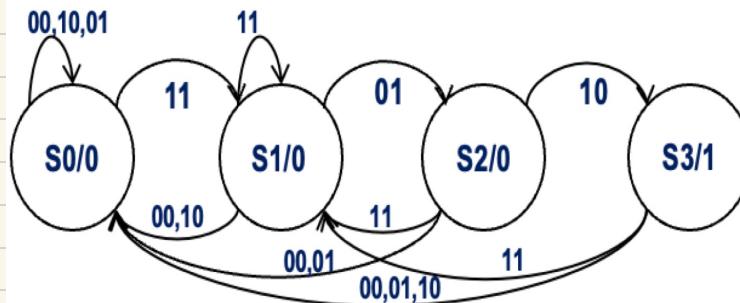
LOGICA DI SCRITTURA



LOGICA DI LETTURA



SINTESI DI RICONOSCIMENTO CIRCUITO SEQUENZIALE CON MOORE



ATTENDIAMO L'ARRIVO
DELLA SEQUENZA ESATTA
 $11, 01, 10$.

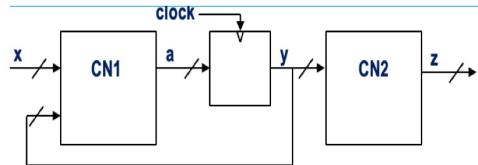
ASSEGNAVO AD OGNI STATO UNA SEQUENZA DI BIT.

$$\begin{array}{llll} S_0 = 00 & S_1 = 01 & S_2 = 11 & S_3 = 10 \end{array}$$

ABBIANO A CHE HA DUE BIT $A_1 A_0$, E Z UN BIT SOLO.
X È IL NOSTRO INPUT A DUE BIT $X_1 X_0$.

INPUT

	$X_1 X_0$	00	01	11	10
$Y_1 Y_0$					
S_0	00 _{S0}	00 _{S0}	01 _{S1}		00 _{S0}
S_1	00 _{S0}	11 _{S2}	01 _{S1}	00 _{S0}	
S_2	00 _{S0}	00 _{S0}	01 _{S1}	10 _{S3}	
S_3	00 _{S0}	00 _{S0}	01 _{S1}	00 _{S0}	



CREO LA TABELLA DEI PERCORSI.

PER I RISULTATI DI A, GUARDIAMO PER A₁ DOVE IL PRIMO BIT È 1, PER A₀ DOVE IL SECONDO BIT È A 1.

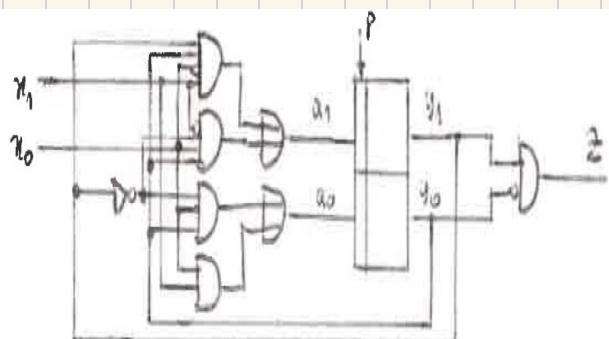
$$A_1 = (\text{NOT}(Y_1) * Y_0 * \text{NOT}(X_1) * X_0) + (Y_1 * Y_0 * X_1 * \text{NOT}(X_0))$$

$$A_0 = (\text{NOT}(Y_1) * Y_0 * X_0) + (X_1 * X_0)$$

$Y_1 Y_0$	Z
00	0
01	0
11	0
10	1

IL RISULTATO DI Z È
DATO DA QUALE STATO
RAGGIUNGE LA SEQUENZA
ESATTA.

DOPODICHE' REALIZZARE
IL CIRCUITO.



GERARCHIE DI MEMORIA

I SISTEMI DI MEMORIZZAZIONE SONO ORGANIZZATI IN GERARCHIE IN BASE A VELOCITÀ, COSTO, VOLATILITÀ.

- MEMORIA PRINCIPALE (RAM): DOVE LA CPU ACCDE DIRETTAMENTE.
- MEMORIA SECONDARIA (DISK): UNA MEMORIA NON VOLATILE.

LE MEMORIE POSSONO ESSERE:

- SRAM (STATIC), MEMORIZZA UN GRAN NUMERO DI PAROLE IN UN INSIEME DI FLIP-FLOP, FINCHÉ C'È ALIMENTAZIONE I DATI VENGONO MANTENUTI.

- DRAM (DYNAMIC), I BIT VENGONO ESPRESI SOTTO FORMA DI CARICA DEI TRANSISTOR, CHE VA RINFRESCATO PERIODICAMENTE CON UN CIRCUITO DI CONTROLLO. USANDO MENO TRANSISTOR LA DENSITÀ DI BIT MEMORIZZATI IN DRAM È PIÙ ALTA RISPETTO ALLA SRAM.

- HDD: MEMORIA PERSISTENTE CON DISCHI MAGNETICI.

- SSD: TECNOLOGIA BASATA CON CELLE SEMICONDUTTORI, E NON RICHIENDE SPOSTAMENTI MECCANICI.

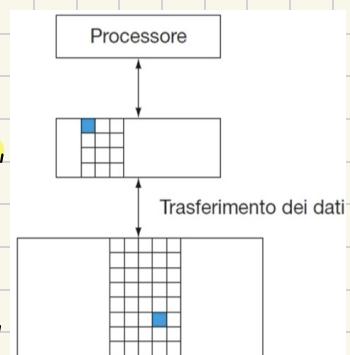
LOCALITÀ TEMPORALE = QUANDO SI FA RIFERIMENTO AD UN ELEMENTO DI MEMORIA, SI EFFETTUERANNO ALTRE OPERAZIONI SULLO STESSO ELEMENTO.

LOCALITÀ SPAZIALE = // // // // // // // // SUBITO DOPO OPERAZIONI ANCHE SU ALTRI ELEMENTI VICINI.

LIVELLO SUPERIORE ED INFERIORE DELLA MEMORIA

IL LIVELLO SUPERIORE È AD ACCESSO RAPIDO, CON CAPACITÀ MINORE, MENTRE, IL LIVELLO INFERIORE È PIÙ LENTO MA HA PIÙ CAPACITÀ.

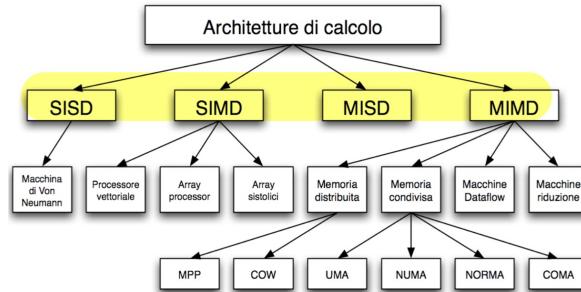
IN LETTURA SI CERCA PRIMA IL DATO A LIVELLO SUPERIORE, SE L'ELEMENTO È GIÀ PRESENTE (CACHE HIT) SI PROCEDE CON LA LETTURA, ALTRIMENTI BISOGNA TRASFERIRE I DATI A LIVELLO INFERIORE (CACHE MISS).



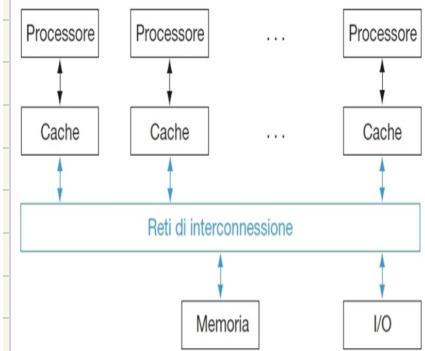
In SCRITTURA INVECE PONE IL PROBLEMA DI CONSISTENZA TRA DATI IN CACHE E DATI A LIVELLO INFERIORE.
SI ADOTTANO STRATEGIE DIVERSE:

- SCRIVERE IN CACHE E IN MEMORIA (WRITE-THROUGH)
- SCRIVERE IN CACHE E SCHEDULARE SCRITTURA SU DISCO TRAMITE BUFFER (WRITE BUFFER)
- SCRIVERE IN CACHE E COPIARE I DATI IN MEMORIA SOLO QUANDO LA CACHE DEVE ESSERE SOVRASCritto (WRITE BACK)

ARCHITETTURE DI CALCOLO



MIMD



S=Single, M=Multiple, I=Instruction stream(s), D=Data stream(s)

Due classi di MIMD: Shared Memory MIMD e Distributed Memory MIMD

LE ARCHITETTURE SHARED MEMORY (MIMD) SONO ALLA BASE DEL MULTICORE PROGRAMMING.

DIVERSI PROGRAMMI ESEGUITI SIMULTANEALEMENTE POSSONO CONDIVIDERE I DATI IN MEMORIA PRINCIPALE ATTRAVERSO TECNICHE DI THREADING HW E SW. IN QUESTO CASO GESTIRE LA CACHE DIVENTA COMPLESSO, E RICHIEDE LA COERENZA TRA I VALORI IN CACHE E QUELLI IN MEMORIA.

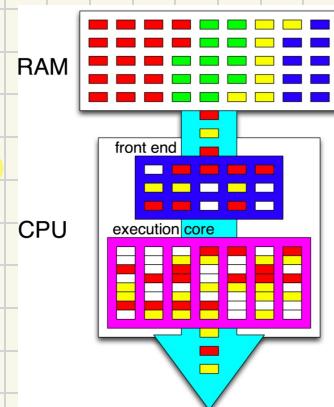
SOFTWARE MULTITHREADING

A LIVELLO DI SOFTWARE VIENE GESTITO DAL SISTEMA OPERATIVO, UN SINGOLO PROGRAMMA HA DIVERSI FLUSSI DI ISTRUZIONI (QUINDI PIÙ DI UN PC), CHE CONDIVIDONO LA STESSA MEMORIA ED EFFETTUANO OPERAZIONI SIMULTANEALEMENTE (PARALLELISMO REALE O VIRTUALE)

PARALLELISMO VIRTUALE: THREAD ESEGUITI SULLO STESSO PROCESSORE.

PARALLELISMO REALE: // // SU DIVERSI CORE.

L'HARDWARE MULTI-THREADING È SUPPORTATO SOLO DAI PROCESSORI SUPERSCALARI CON INSTRUCTION LEVEL PARALLELISM, E SIMULTANEOUS MULTITHREADING; IN CASO DI UN CACHE MISS (STALL) È POSSIBILE PASSARE ALL'ESECUZIONE DI UN ALTRO THREAD SULLO STESSO CORE



PROCESSI E PROGRAMMI

UN PROCESSO È UN'ATTIVITÀ CONTROLLATA DA UN PROGRAMMA CHE SI SVOLGE SUL PROCESSORE, UN PROGRAMMA SPECIFICA UNA SEQUENZA DI ESECUZIONI DI UN INSIEME DI ISTRUZIONI. PIÙ PROCESSI POSSONO ESEGUIRE LO STESSO PROGRAMMA, POSSANO CONDIVIDERE LO STESSO CODICE, I DATI E LO STATO RIMANGONO SEPARATI.

DAL CODICE SORGENTE AL CODICE ESEGUITIBILE

- COMPILE-TIME: COMPILAZIONE, CODICE PASSA DA ALTO A BASSO LIVELLO; LINKING, IL CODICE DELLE LIBRERIE IMPORTATE VIENE INCLUSO.
- LOADING-TIME: IL CODICE BINARIO VIENE CARICATO IN RAM.
- EXECUTION-TIME: IL CODICE OGGETTO VIENE ESEGUITO DALLA CPU.

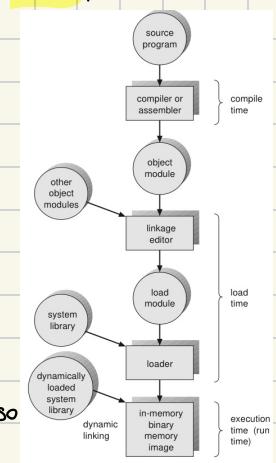
INDIRIZZI LOGICI E FISICI

• SPAZIO DI INDIRIZZAMENTO LOGICO: OGNI PROCESSO È ASSOCIATO AD UNO SPAZIO DI INDIRIZZAMENTO LOGICO; GLI INDIRIZZI USATI IN UN PROCESSO SONO INDIRIZZI LOGICI (INDIRIZZI DI MEMORIA GENERATI DALLA CPU).

• SPAZIO DI INDIRIZZAMENTO FISICO: OGNI INDIRIZZO LOGICO CORRISPONDE AD UN INDIRIZZO FISICO, OS SI OCCUPA DELLA TRADUZIONE DA LOGICO A FISICO.

BINDING

SI INDICA L'ASSOCIAZIONE DI INDIRIZZI DI MEMORIA AI DATI E ALLE ISTRUZIONI (VIENE EFFETTUATO IN CARICA E/O ESECUZIONE).



BINDING DURANTE CARICAMENTO

IL CODICE GENERATO DAL COMPILATORE NON CONTIENE INDIRIZZI ASSOLUTI, MA RELATIVI, QUESTO CODICE È DETTO RILOCABILE. IL PROGRAMMA LOADER SI PREOCCUPA DI AGGIORNARE TUTTI I RIFERIMENTI AGLI INDIRIZZI DI MEMORIA COERENTEMENTE (PERMETTE DI GESTIRE MULTIPROGRAMMAZIONE), RICHIEDE UNA TRADUZIONE DEGLI INDIRIZZI DA PARTE DEL LOADER.

BINDING DURANTE L'ESECUZIONE

L'INDIVIDUAZIONE DELL'INDIRIZZO DI MEMORIA EFFETTIVO VIENE FATTA CON L'AVVIO DI UN COMPONENTE HW, MEMORY MANAGEMENT UNIT (MMU), IL PROGRAMMA POTRÀ SPOSTARSI DA UNA ZONA ALL'ALTRA DELLA MEMORIA.

LOADING (DINAMICO)

CONSENTE DI POTER CARicare alcune routine di libreria SOLO QUANDO VENGONO CHIAMATE, TUTTE LE ROUTINE RISIEDONO SUL DISCO, MENTRE QUELLE DI POCO IMPORTANZA NON VENGONO CARICATE.

LINKING

- STATICO: LE ROUTINE DI LIBRERIA VENGONO CARicate DA OGNI PROGRAMMA CHE LE USA.
- DINAMICO: VIENE POSTICIPATO IL LINKING AL PRIMO RIFERIMENTO DELLE ROUTINE.

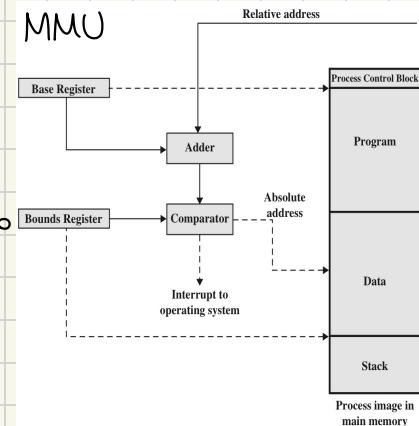
PROCESSI E SISTEMA OPERATIVO

IL SISTEMA OPERATIVO TIENE TRACCIA DEI PROCESSI DA UNA TABELLA DEI PROCESSI (PCB PROCESS CONTROL BLOCK MANTIENE LE INFORMAZIONI SUI PROCESSI).

THREAD (PROCESSO LEGGERO)

È UN'UNITÀ DI ESECUZIONE CON: PC, REGISTRI, STACK, STATO DI ESECUZIONE.

UN THREAD CONdivide CON I THREAD SUOI PARI, UNA UNITÀ DI ALLOCAZIONE RISORSE: CODICE ESEGUITIBILE, DATI, RISORSE DEL SISTEMA OPERATIVO.



UN TASK = UNITÀ DI RISORSE + THREAD CHE VI ACCEDONO.

CONDIVISIONE DI RISORSE TRA I THREAD

- VANTAGGI: MAGGIORE EFFICIENZA, CREARE/CANCELLARE THREAD È MOLTO PIÙ VELOCE, COME LO È L'ESECUZIONE DI UNA TASK CHE È OPERATA DA PIÙ THREAD.
- SVANTAGGI: UNA MAGGIORE COMPLESSITÀ DI PROGRAMMAZIONE, DIFFICOLTÀ DI PROTEZIONE DEI DATI.

USER-LEVEL THREAD

STACK, PC E OPERAZIONI SUI THREAD SONO IMPLEMENTATI TRAMITE LIBRERIE A LIVELLO UTENTE.

- VANTAGGI: EFFICIENTI (ASSSENZA DI SYSTEM CALL), SEMPLICI DA IMPLEMENTARE, PORTABILI.
- SVANTAGGI: ASSSENZA DI SCHEDULING AUTOMATICO TRA I THREAD, LE SYSTEM CALL SONO BLOCCANTI, NON SFRUTTA IL MULTIPROCESSORE.

KERNEL-LEVEL THREAD (UNIX)

IL KERNEL GESTISCE DIRETTAMENTE I THREAD, ATTRAVERSO SYSTEM CALL.

- VANTAGGI: LO SCHEDULING È PER THREAD (NON PER PROCESSO), UN THREAD CHE SI BLOCCA NON BLOCCA TUTTO IL PROCESSO, UTILE PER PROCESSI DEL OS E SISTEMI MULTIPROCESSO.
- SVANTAGGI: MENO EFFICIENTE, RISCRITTURA DELLE SYSTEM CALL, MENO PORTABILE, SCHEDULING NON MODIFICABILE.

PROCESSI IN UNIX

Ogni processo è identificato dal PID, ha uno spazio di indirizzamento separato (estraneo a quello degli altri processi).

Un processo è formato da: STACK, DATI (STATICI E HEAP), E IL CODICE.

Il codice, dati e heap sono nella parte iniziale di memoria virtuale, lo stack nella parte finale.

PER CREARE UN PROCESSO IN UNIX, SI UTILIZZA LA FUNZIONE `FORK()`, CHIAMATA DAL PROCESSO P(PADRE) CREA IL NUOVO PROCESSO F(FIGLIO). P E F CONDIVIDONO LO STESSO CODICE, F HA UNA COPIA DEI DATI E DEL PC DI P. F PROSEGUITA L'ESECUZIONE ALLA RIGA DOPO LA FORK.

PER CAPIRE SE `FORK()` HA FUNZIONATO RESTITUISCE IL PID DEL FIGLIO ALTRIMENTI 0.

LA FUNZIONE `EXECVE()` SOSTITUISCE DATI, CODICE, STACK AL FIGLIO, COSÌ CHE POSSA ESEGUIRE UN NUOVO PROGRAMMA.

IL PADRE PUÒ ASPETTARE LA TERMINAZIONE DEL FIGLIO CON `WAITPID`.

USER AND KERNEL MODE

I PROCESSI IN UNIX LAVORANO IN USER E KERNEL MODE, CIOÈ IL KERNEL ESEGUE NEL CONTESTO DI UN PROCESSO LE OPERAZIONI PER GESTIRE LE CHIAMATE DI SISTEMA, IN MODO CHE IL PROCESSO POSSA PASSARE AL CONTESTO UTENTE QUANDO GLI SERVE.

LA PARTE DINAMICA DI UN CONTESTO DI PROCESSO È ORGANIZZATA COME UNO STACK.

Livello 0: User

Livello 1: Chiamate di sistema

Livelli 2-6: Interrupt (l'ordine dipende dalla priorità associata alle interrupt)

ALGORITMO DI GESTIONE DELLE INTERRUZIONI

- 1) SALVA IL CONTESTO DEL PROCESSO CORRENTE
- 2) DETERMINA LA FONTE DELL'INTERRUPT
- 3) RECUPERA L'INDIRIZZO DI PARTENZA DELLA ROUTINE DI GESTIONE INTERRUPT
- 4) INVOCÀ LE ROUTINE DI GESTIONE
- 5) RECUPERA IL LIVELLO DI CONTESTO PRECEDENTE

PER MOTIVI DI EFFICIENZA PARTE DI INTERRUPT È GESTITA DALLA CPU.

IL MODO USER-KERNEL PERMETTE AL KERNEL DI LAVORARE NEL CONTESTO DI UN ALTRO PROCESSO SENZA DOVER CREARE ALTRI PROCESSI KERNEL. IL CONTROLLO INTERRUPT PUÒ PASSARE DA UN PROCESSO ALL'ALTRO IN 4 SCENARI (CONTEXT SWITCH): IL PROCESSO SI SOSPENDE, QUANDO TERMINA, QUANDO TORNA A MODO USER, QUANDO TORNA A MODO USER DOPO CHE IL KERNEL HA GESTITO UN INTERRUPT.

PROGRAMMA SEQUENZIALE

IL MODELLO DI ESECUZIONE SEQUENZIALE PREVEDE:

- PROGRAMMI FORMATI DA SEQUENZE DI ISTRUZIONI (ORDINATE)
- UN'UNICA MEMORIA VIRTUALE (PER DATI STATICI E DINAMICI)
- ESECUZIONE DI UNA SINGOLA ISTRUZIONE ALLA VOLTA.

SISTEMI CONCORRENTI

SUPPONIAMO CHE IL NOSTRO ELABORATORE FORNISCA UNA QUALCHE FORMA DI PARALLELISMO:

- PARALLELISMO REALE: MULTI-PROCESSORE, SISTEMA DISTRIBUITO
- PARALLELISMO VIRTUALE: MULTI-PROGRAMMAZIONE

PROGRAMMAZIONE CONCORRENTE

RIPASSIAMO IL MODELLO DI CALCOLO IN MODO DA PREVEDERE L'ESECUZIONE DI UN INSIEME DI ISTRUZIONI, CHE POSSONO ESSERE PARZIALMENTE ORDINATE. IN QUESTO MODO SFRUTTIAMO LE ARCHITETTURE MULTI-PROCESSORE, MIGLIORIAMO LE INTERFAZIE GRAFICHE E COMPRENSIBILITÀ DEL CODICE.

ESEMPI SUI THREAD

VAR1=x₁, VAR2=x₂ //DICHIARAZIONE VARIABILI GLOBALI CONDIVISE.

THREAD T₁ {PROG-i}

THREAD T₂ {PROG-m}

DENTRO PROG-i TROVIAMO UN PROGRAMMA SEQUENZIALE. OGNI THREAD VIENE ESEGUITO IN CONCORRENZA CON GLI ALTRI CONDIVIDENDO LE VARIABILI GLOBALI (L'ESECUZIONE IN PARALLELO NON IMPLICA L'ESECUZIONE RIPETUTA DEI SOTTO PROGRAMMI)

Esempio 1

Risorse
var x=0

Thread P1 { x:=500; }
Thread P2 { x:=0; }
Thread P3 { write(x); }

HA DUE POSSIBILI RISULTATI: 0 o 500. L'ESECUZIONE DI TALE PROGRAMMA DA ORIGINE AD UN SOLO VALORE.
UN PROGRAMMA CONCORRENTE DEFINISCE UNA FUNZIONE CHE DATO UN INPUT PRODUCE UN INSIEME DI POSSIBILI OUTPUT (NON DETERMINISTICO).

Esempio 2

var x=0

Thread P1 {
 while (true) x:=500; endwhile;
}

Thread P2 {
 while (true) x:=0; endwhile
}

Thread P3 {
 while (true) write(x); endwhile;
}

HA UN INSIEME INFINTO DI OUTPUT.
POTREI ESEGUIRE P1 E P2 MA MAI P3, ESEGUIRE UNA SOLA VOLTA P1 E UNA SOLA VOLTA P3 ... ECC...
L'ESECUZIONE DARA' ORIGINE AD UNA DELLE POSSIBILI SEQUENZE DI RISULTATO, OVVIAMENTE ESECUCIONI DIVERSE DANNO ORIGINE A DIVERSI RISULTATI.

Esempio 3

Risorse
var x=100

Thread P1 {
 x:=x+1;
}

Thread P2 {
 x:=x-1;
}

HA 3 POSSIBILI RISULTATI.
SE LI ESEGUE IN SEQUENZA OTTENGO 100.
SE ESEGUE P1 FINO ALLA VALUTAZIONE X+1 E PRIMA DELL'ASSEGNAZIONE (=) ESEGUE P2, OTTENGO 101.
CASO CONTRARIO A QUESTO OTTENGO 99.

RACE Condition

Più THREAD ACCEDONO CONCORRENTEMENTE AGLI STESSI DATI, E IL RISULTATO DIPENDE DALL'ORDINE DI ESECUZIONE DI ISTRUZIONI SELEZIONATE DAI DIVERSI THREAD. NEI SISTEMI MULTITASKING I RACE CONDITION SONO MOLTO FREQUENTI, QUINTO MOLTO PERICOLOSI.

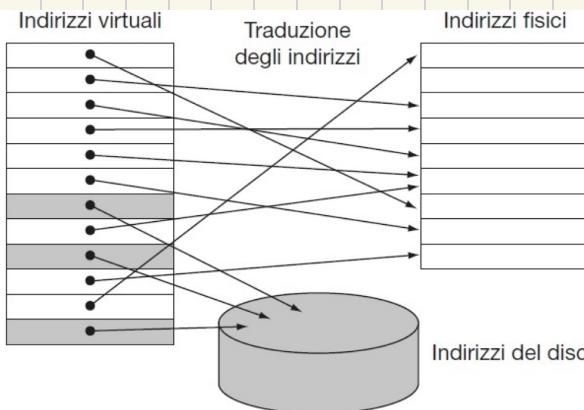
MEMORIA VIRTUALE

È LA TECNICA CHE PERMETTE L'ESECUZIONE DI PROCESSI NON COMPLETAMENTE CARICATI IN MEMORIA PRINCIPALE. PERMETTE DUNQUE DI ESEGUIRE IN CONCORRENZA I PROCESSI CHE NECESSITANO DI MAGGIORE MEMORIA, QUINDI C'È UNA SEPARAZIONE FRA MEMORIA LOGICA E FISICA.

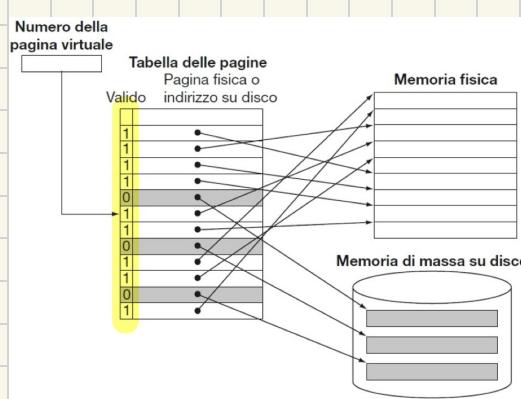
Ricordiamo l'architettura di Von Neumann che dice che istruzioni e dati da eseguire devono essere caricati in RAM, tuttavia non è necessario che tutto lo spazio di indirizzamento logico sia in RAM, in quanto non viene utilizzato tutto contemporaneamente.

IMPLEMENTAZIONE MEMORIA VIRTUALE

PAGINAZIONE SU RICHIESTA (DEMAND PAGING), SI UTILIZZA LA TECNICA DELLA PAGINAZIONE (ALCUNE POTREBBERO TROVARSI IN MEMORIA SECONDARIA); NELLA TABELLA DELLE PAGINE DI OGNI SINGOLO PROCESSO SI UTILIZZA UN BIT DI VALIDITÀ (PER DIRE SE È PRESENTE IN MEMORIA PRINCIPALE); QUANDO UN PROCESSO TENTA DI ACCEDERE AD UNA PAGINA NON IN RAM IL PROCESSORE GENERA UN TRAP (PAGE FAULT) E IL PAGER CARICA LA PAGINA.



SIA LA PARTE VIRTUALE SIA QUELLA FISICA È MAPPATA IN PAGINE, COSÌ CHE OGNI PAGINA VIRTUALE È MAPPATA SU QUELLA FISICA, MENTRE LE PAGINE VIRTUALI NON MAPPATE SU QUELLE FISICHE SI TROVANO SU DISCO. PIÙ PAGINE VIRTUALI POSSONO ESSERE MAPPATE SU QUELLE FISICHE, COSÌ I PROGRAMMI CONDIVIDERANNO DATI E PARTI DI CODICE.



LA TABELLA DELLE PAGINE PUÒ ESSERE A PIÙ LIVELLI, QUINDI PAGINIAMO LA TABELLA DELLE PAGINE.

pagine a due

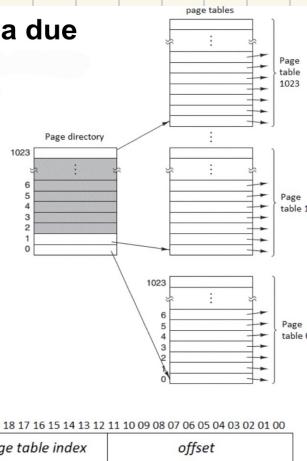
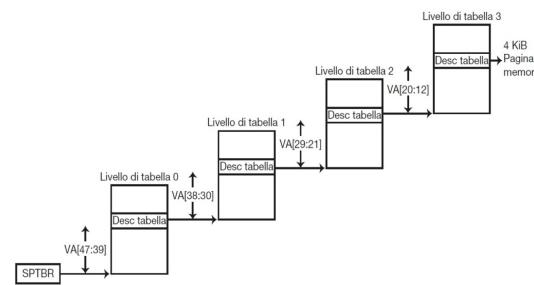


Tabella delle pagine a più livelli con indirizzi a 64 bit



Nei processori a 64 bit si usano quattro livelli ed ogni indirizzo logico è diviso in cinque campi:

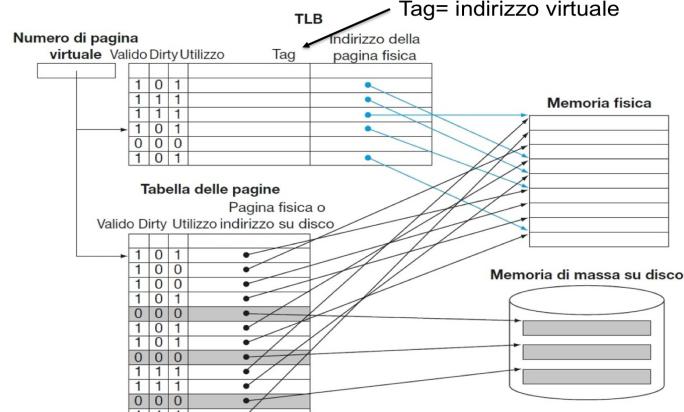
- 1.un campo di 9 bit che contiene un numero utilizzato per accedere alla **page directory** di primo livello,
- 2.un campo di 9 bit che contiene un numero utilizzato per accedere alla **page directory** di secondo livello,
- 3.un campo di 9 bit che contiene un numero utilizzato per accedere alla **page directory** di terzo livello,
- 4.un campo di 9 bit che contiene un numero utilizzato per accedere alla tabella delle pagine di quarto livello,
- 5.un campo di 12 bit che contiene l'offset che rappresenta la cella all'interno del **frame** della memoria centrale che contiene la pagina.

IL CACHING APPLICATO ALLA TABELLA DELLE PAGINE (TLB)

LA PAGINAZIONE COME ABBIANO VISTO È UTILE PER RAZIONALIZZARE L'USO DELLA RAM, MA QUESTO INTRODUCE UN OVERHEAD NELLA FASE DI RISOLUZIONE DEGLI INDIRIZZI VIRTUALI.

VIENE INTRODOTTA UNA MEMORIA CACHE CHIAMATA **TRANSACTION LOOKAHEAD BUFFER (TLB)** PER MANTENERE UN ACCESSO DIRETTO AD UN NUMERO FISSO DI ELEMENTI DELLA TABELLA DELLE PAGINE. NELLA TBL SI ASSOCIA AD UN INDIRIZZO VIRTUALE IL CORRISPONDENTE FISICO, E DEI FLAG, "USE BIT" SE LA PAGINA È IN USO, "DIRTY BIT" SE CI SONO STATE MODIFICHE RECENTI.

MEMORIA VIRTUALE E CACHE LAVORANO INSIEME, NON CI POSSONO ESSERE ELEMENTI IN CACHE CHE NON SONO IN MEMORIA. ESISTONO VARIE STRATEGIE PER DECIDERE COSA TENERE IN CACHE, AD ESEMPIO LRU (LEAST RECENTLY USED). SELEZIONA INDIRIZZI IN TBL CON BIT DI USO A 0, OPPURE IN MODO CASUALE.



Con le TLB bisogna gestire due diversi tipi di miss:

- miss nella TLB (indirizzo logico non è nella cache) → page table
- miss nella tabella delle pagine (validità=0) → page fault

CONFLITTI NELL'USO DELLA CACHE

SE UN BLOCCO CHE DEVO SCRIVERE IN CACHE DEVE ESSERE SOVRASCRITTO SU UN'ALTRO BLOCCO GIÀ PRESENTE IN CACHE COSA SUCCIDE AL VECCHIO BLOCCO?

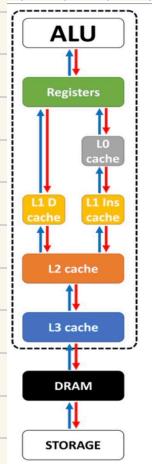
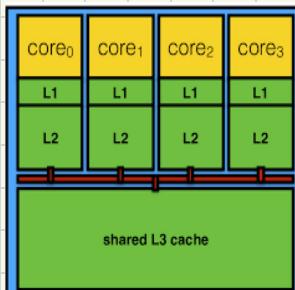
- SE IL VECCHIO BLOCCO È STATO ACCEDUTO IN LETTURA (READ), SI RIMPIAZZA
 - // " " " " " MODIFICATO (READ/WRITE), DIPENDE DALLE POLITICHE DI COERENZA TRA LIVELLI DI MEMORIA.
- WRITE THROUGH (SCRIVO SIA IN CACHE CHE IN MEMORIA), IL BLOCCO DI CONFLITTO PUÒ ESSERE RIMPIAZZATO SENZA PROBLEMI.
- WRITE BACK (SCRIVO SOLO IN CACHE, E RITARDO LA SCRITTURA IN MEMORIA), PRIMA DI RIMPIAZZARE IL BLOCCO QUESTO DEVE PRIMA ESSERE SCRITTO IN MEMORIA.

LE WRITE DIVENTANO TROPPO LUNGHE (ANCHE SE CI SONO SOLO WRITE HIT).

- SOLUZIONE: USIAMO UN WRITE BUFFER COME MEMORIA TAMPONE VELOCE TRA CACHE E MEMORIA, PER NASCONDERE LA LATENZA DI ACCESSO ALLA MEMORIA. QUINDI I BLOCCI VENGONO SCRITTI NEL WRITE BUFFER IN ATTESA DELLA SCRITTURA ASINCRONA IN MEMORIA.

CENNI AL PROBLEMA DELLA COERENZA DELLA CACHE

LIVELLI DI CACHE NEI MULTICORE



- CACHE L1 È LA CACHE PIÙ ALTA DI GERARCHIA, HA DIMENSIONE MINORE MA È LA PIÙ VELOCE (TEMPO DI ATTESA ZERO, È DI SOLITO INTEGRATA NEL CHIP)

- CACHE L2, SI TROVA ACCANTO A L1, DI SOLITO SI ACCDE A L2 SE IN L1 NON SI È TROVATO CIÒ CHE SI CERCAVA. VIENE UTILIZZATA PER COLMARE IL DIVARIO DI PRESTAZIONI DEL PROCESSORE E LA MEMORIA.

COERENZA DELLA MEMORIA

UN SISTEMA DI MEMORIA È COERENTE SE LA LETTURA DI UN DATO RESTITUISCE IL VALORE CHE È STATO SCRITTO PIÙ RECENTE. ABBIAMO VISTO CHE IN MULTICORE PROGRAMMING SI POSSONO VERIFICARE DATA RACE, E IN PRESENZA DI PIÙ LIVELLI DI CACHE LE COSE SI COMPLICANO.

FUNZIONALITÀ RICHIESTE DA UN PROCESSORE MULTICORE :

- MIGRAZIONE, UN DATO VIENE TRASFERITO DA UNA CACHE ALL'ALTRA;
- REPPLICAZIONE, QUANDO UN DATO VIENE LETTO SI MEMORIZZA UNA COPIA IN OGNI CACHE.

SNOOPY CACHE COHERENCE PROTOCOL

L'IDEA È DI ASSICURARE CHE UN PROCESSORE ABBIA ACCESSO ESCLUSIVO AD UN DATO PRIMA DI MODIFICARLO (MUTUA ESCLUSIONE). VIENE IDEALIZZATA LA WRITE INVALIDATE PROTOCOL. CONSISTE CHE QUANDO UNA CPU DEVE MODIFICARE IL DATO X, MANDA UN SEGNALE DI INVALIDAZIONE PER TALE DATO, COSÌ CHE ESISTE UNA SOLA COPIA VALIDA DEL DATO.

LE CACHE IMPLEMENTANO ALGORITMI DI COERENZA ATTRAVERSO DEI CONTROLLER (CIRCUITI DEDICATI SULLA CACHE)

FINE PARTE DELZANO

CONTINUA PARTE D'AGOSTINO

INSTRUCTION LEVEL PARALLELISM

SI TRATTA DI ESEGUIRE PIÙ DI UNA ISTRUZIONE PER CICLO DI CLOCK. SI AUMENTANO I COMPONENTI INTERNI AGLI STADI DELLA PIPELINE.

PARALLELIZZAZIONE

- **STATICA**, LE DECISIONI SONO PRESE DURANTE LA COMPILAZIONE
- **DINAMICA**, " " " " L'ESECUZIONE.
- **IBRIDA**, LA PIÙ USATA

SPECULAZIONE

SPECULARE VUOL DIRE "INDOVINARE" LE CARATTERISTICHE DI UNA ISTRUZIONE, IN MODO DA PERMETTERE L'INIZIO DI ALTRE ISTRUZIONI CHE DIPENDONO DA QUESTA.

SE LA SPECULAZIONE SI RIVELA ERRATA DEVO ESSERE IN GRADO DI TORNARE INDIETRO.

NELLA PARALLELIZZAZIONE STATICA IL COMPILATORE INSERISCE ISTRUZIONI AGGIUNTIVE PER RIPARARE GLI ERRORI.

NELLA PARALLELIZZAZIONE DINAMICA I DATI VENGONO MEMORIZZATI IN UN BUFFER E COPIATI IN MEMORIA IN CASO DI UNA SPECULAZIONE CORRETTA.

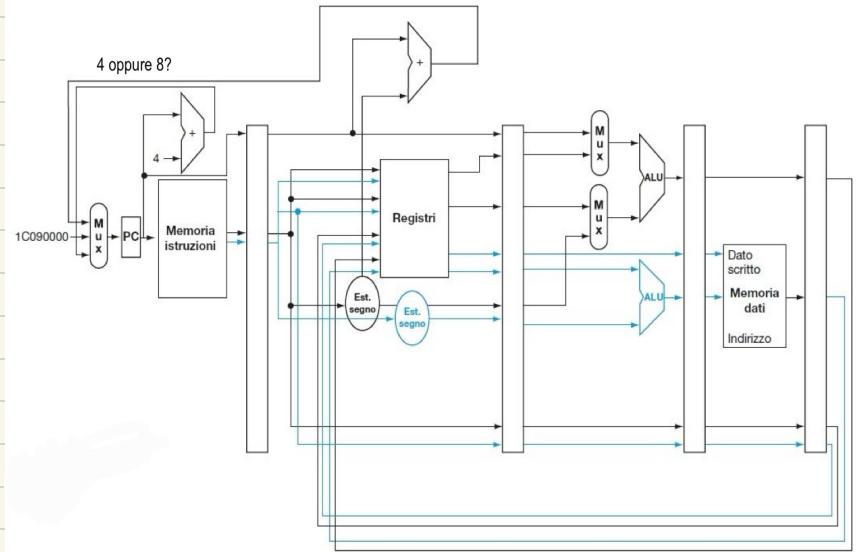
PARALLELIZZAZIONE STATICÀ

IL COMPILATORE SI OCCUPA DI FORMARE PACHETTI DI ISTRUZIONI, E PREVENIRE O RIDURRE HAZARD.

UN PACHETTO DI ISTRUZIONI È UN'ISTRUZIONE PIÙ LUNGA IN CUI I COMPONENTI SONO PRESENTI CAMPI PREDEFINITI. SE LE ISTRUZIONI NON SI POSSONO ACCOPPIARE USO UNA NOP.

Tipo di istruzione	Stadi della pipeline				
Istruzione ALU o di salto	IF	ID	EX	MEM	WB
Istruzione di load o di store	IF	ID	EX	MEM	WB
Istruzione ALU o di salto	IF	ID	EX	MEM	WB
Istruzione di load o di store	IF	ID	EX	MEM	WB
Istruzione ALU o di salto	IF	ID	EX	MEM	WB
Istruzione di load o di store	IF	ID	EX	MEM	WB

VERY INSTRUCTION LONG WORD (VLIW).



HW AGGIUNTIVO PER LA PARALLELIZZAZIONE.

ESEMPIO ESECUZIONE DI ISTRUZIONI

Ciclo: ld x31, 0(x20) ld **x31, 0(x20)**
 add x31, x31, x21 add **x31, x31, x21**
 sd x31, 0(x20) sd **x31, 0(x20)**
 addi x20, x20, -8 addi **x20, x20, -8**
 blt x22, x20, Ciclo blt **x22, x20, Ciclo**

Istruzioni ALU o di salto condizionato	Istruzioni trasferimento dati	Ciclo di clock
Ciclo:	ld x31, 0(x20)	1
addi x20, x20, -8	<i>Data hazard con la add</i>	2
add x31, x31, x21		3
blt x22, x20, Ciclo	sd x31, 8(x20)	4

Figura 4.67 Codice riordinato come apparirebbe in un processore RISC-V con pipeline a due vie. Si noti che avendo spostato la addi prima della sd, abbiamo dovuto aggiungere 8 all'offset della sd.

5 istruzioni per ciclo, 4 cicli di clock.
 CPI = 0,8 o, alternativamente, IPC = 1,25. Poco

RENAMEING
PER ACCOPPIARE

Istruzioni ALU o di salto condizionato	Istruzioni trasferimento dati	Ciclo di clock
Ciclo: addi x20, x20, -32	ld x28, 0(x20)	1
	ld x29, 24(x20)	2
	ld x30, 16(x20)	3
	ld x31, 8(x20)	4
add x28, x28, x21	sd x28, 32(x20)	5
add x29, x29, x21	sd x29, 24(x20)	6
add x30, x30, x21	sd x30, 16(x20)	7
add x31, x31, x21	sd x31, 8(x20)	8
blt x22, x20, Ciclo		

Figura 4.68 Il codice di Figura 4.67 dopo l'espansione del ciclo e il riordinamento del codice, come apparirebbe nell'esecuzione su un RISC-V dotato di parallelizzazione statica a due vie. Gli spazi vuoti sono occupati da istruzioni nop. Dato che x20 viene decrementato di 32 dalla prima istruzione del ciclo, gli indirizzi contenuti nel PC saranno rispettivamente il valore originale di x20, x20 meno 8, meno 16 e meno 24.

14 istruzioni, 8 cicli, IPC = 1,75 e CPI = 0,57. Molto meglio.
 Notate che uso più registri e il segmento testo è più grande.

```
for (i=0; i<n; i++)          for (i=0; i<n; i++)
  A[i] = A[i] + 5            A[i] = A[i] + 5
                           A[i+1] = A[i+1] + 5
```

Se il ciclo di prima fosse multiplo di 4 avrei
 Ciclo: ld **x31, 0(x20)**, add **x31, x31, x21**, sd **x31, 0(x20)**,
 ld **x31, -8(x20)**, add **x31, x31, x21**, sd **x31, -8(x20)**,
 ld **x31, -16(x20)**, add **x31, x31, x21**, sd **x31, -16(x20)**,
 ... blt x22, x20 Ciclo

Scritto così però cambia poco
 risparmio 3 blt e 3 addi

In realtà gli x31 di iterazioni successive sono indipendenti

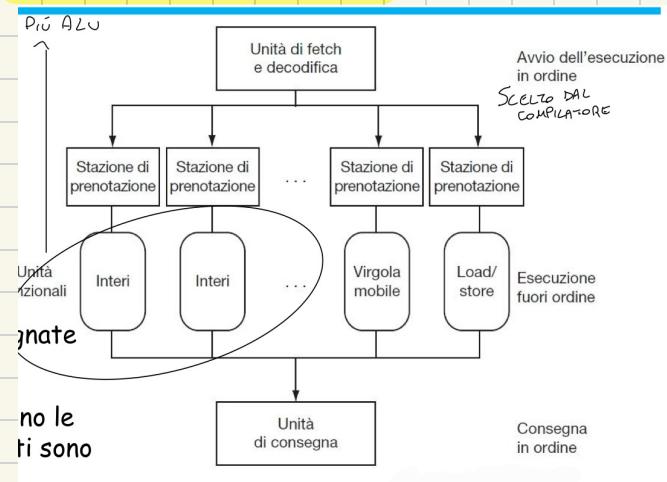
Il compilatore per evitare le false dipendenze (nominali) ridenomina i registri

Loop Unrolling

Loop vengono srotolati in modo
 da creare pacchetti di istruzioni.

PARALLELIZZAZIONE DINAMICA

I PROCESSORI SUPER SCALARI SI BASANO SU QUESTA TECNICA, LE ISTRUZIONI SONO ESEGUITE IN ORDINE E LA CPU DECIDE QUANTE ESEGUIRNE. NEI PROCESSORI SUPER SCALARI È L'HARWARE CHE GARANTISCE LA CORRETTEZZA DELL'ISTRUZIONE



LE ISTRUZIONI VANO NELLA STAZIONE CORRISPONDENTE, SOLO QUANDO GLI OPERANDI SONO DISPONIBILI L'ISTRUZIONE POTRÀ ESSERE ESEGUITA. POSSONO ESSERE ESEGUITE FUORI ORDINE, MA SEMPRE CONSEGNATE IN ORDINE. APPENA I DATI SONO DISPONIBILI LE UNITÀ FUNZIONALI ESEGURANNO. I DATI ARRIVANO O DA REGISTER FILE, O MEMORIA O DAL BUFFER. SE LA SPECULAZIONE È ERRATA CANCELLA I DATI DAL BUFFER E RIORDINO.