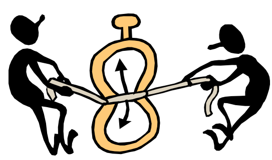
Principali Innovazioni nei Computer (1)

- Il concetto di famiglia
  - IBM System/360, anno 1964
  - DEC PDP-8
  - Separa l'architettura dall'implementazione
- Unità di Controllo Microprogrammata
  - Idea iniziale di Wilkes, anno 1951
  - Introdotta nell' IBM System/360, anno 1964
- Memoria Cache
  - IBM System/360 model 85, anno 1969

Principali Innovazioni nei Computer (2)

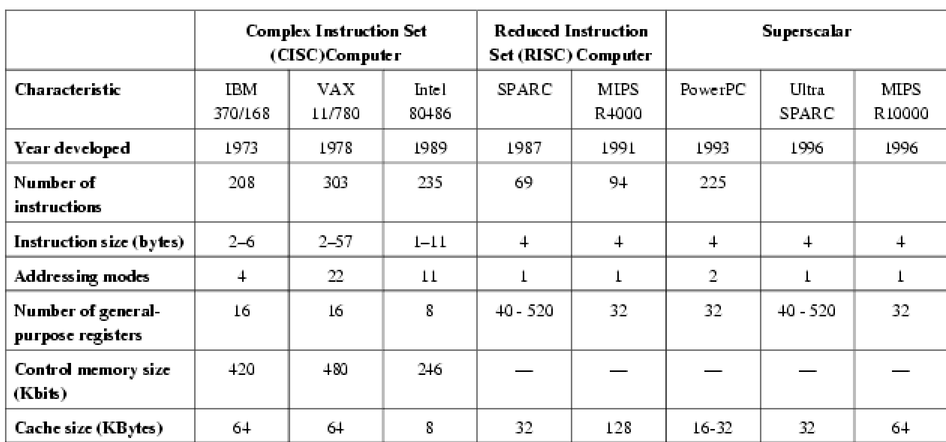
- RAM a Semiconduttori
  - anno 1970
- Microprocessori
  - Intel 4004, anno 1971
- Pipeline
  - già vista a lezione
- Processori Multipli

Passo successivo - RISC



- **R**educed **I**nstruction **S**et **C**omputer
- Caratteristiche chiave
  - Numero elevato di registri ad uso generale
  - … oppure utilizzo di compilatori per ottimizzare l'uso dei registri
  - Set istruzioni semplice e limitato
  - Ottimizzazione della pipeline (basata sul formato fisso per le istruzioni, metodi indirizzamento semplici,…)

Comparazione fra vari processori



	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Perchè erano nate le architetture **C**omplex **I**struction **S**et **C**omputer (CISC) ?

- Costo del software molto maggiore del costo dell' hardware
- Linguaggi ad alto livello sempre più complessi
- Gap semantico

conseguenze:

- Set di istruzioni ampio
- Svariati modi di indirizzamento
- Implementazione hardware di costrutti di linguaggi ad alto livello
  - Ad esempio: CASE (switch) su VAX

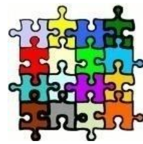
Scopi del CISC

- Facilitare la scrittura del compilatore
- Migliorare l'efficienza dell' escuzione
  - Operazioni complesse implementate tramite microcodice
- Supportare i linguaggi ad alto livello più complessi

Studio delle caratteristiche di esecuzione delle istruzioni

- Operazioni eseguite
  - determinano le funzioni da eseguire e le modalità di interazione con la memoria
- Operandi usati
  - tipo e frequenza determinano le modalità di salvataggio e i modi di indirizzamento
- Serializzazione dell'esecuzione
  - determina l'organizzazione della pipeline e del controllo
- Studi sviluppati a partire dalle istruzioni macchina generate dai programmi scritti in un linguaggio ad alto livello
- Utilizzate misure dinamiche raccolte durante l'esecuzione di programmi

Operazioni



- Assegnamento
  - Trasferimento di dati
- Costrutti condizionali (IF, LOOP)
  - Controllo di serializzazione
- Chiamata/ritorno da procedura impiega molto tempo
- Alcuni costrutti dei linguaggi ad alto livello richiedono molte operazioni macchina

Frequenza relativa di istruzioni ad alto livello [PATT82a]

	Occorrenza Dinamica		Occorrenza ponderata sulle istruzioni macchina		Occorrenza ponderata sugli accessi a memoria	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

Operandi

- Principalmente variabili scalari locali
- L’ottimizzazione si deve concentrare sull’accesso alle variabili locali

	Pascal	C	Media
Costanti Intere	16%	23%	20%
Variabili scalari	58%	53%	55%
Array/Strutture	26%	24%	25%

Chiamate di Procedura

- Consumano molto tempo
- Dipendono dal numero di parametri passati
- Dipendono dal livello di annidamento
- La maggior parte dei programmi non eseguono chiamate multiple annidate di procedure
- La maggior parte delle variabili sono locali

Implicazioni

Il miglior supporto si ottiene ottimizzando le caratteristiche più utilizzate e più onerose dal punto di vista del consumo di tempo

1. Ampio numero di registri o loro uso ottimizzato dal compilatore
   - Ottimizzazione dei riferimenti agli operandi
2. Progettazione accurata della pipeline
   - Predizione dei salti condizionali, etc.
3. Set istruzioni semplificato (ridotto)

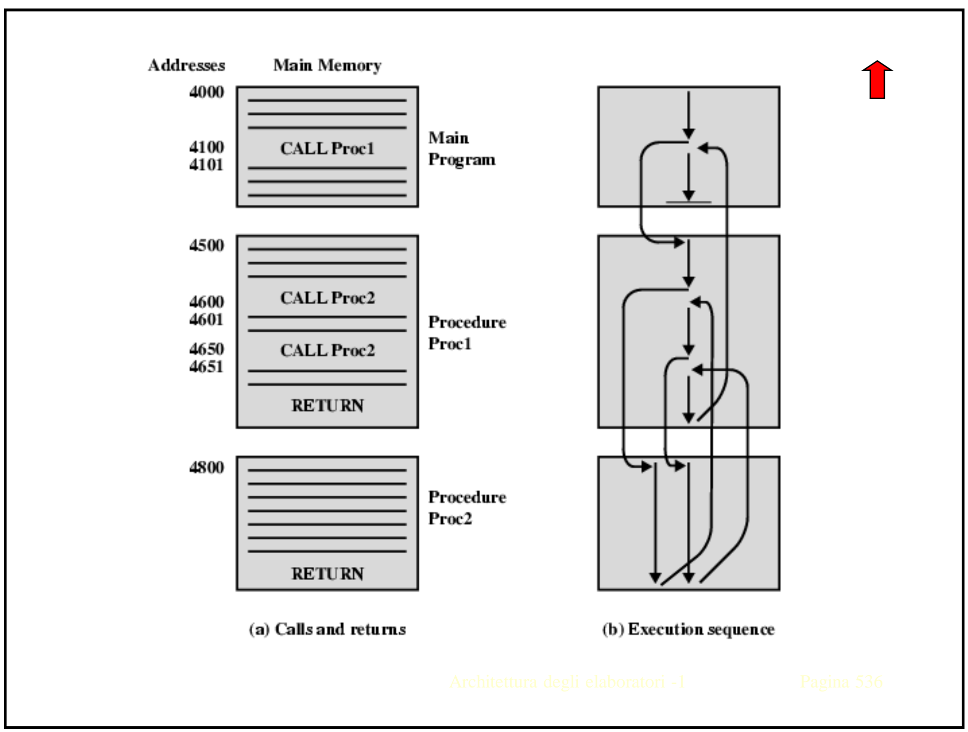
Trattamento dei registri

- Soluzione hardware
  - Usare più registri
  - In questo modo si possono mantenere più variabili nei registri

- Soluzione software
  - Registri allocati dal compilatore
  - Allocazione basata sulle variabili più usate per ogni intervallo di tempo
  - Richiede l’utilizzo di tecniche di analisi dei programmi molto sofisticate

Registri per variabili locali

- Strategia: memorizzare variabili scalari locali nei registri
- Vantaggio: riduce l’accesso alla memoria
- Problemi:
  - ogni chiamata a procedure (o funzione) cambia la località (scope delle variabili)
  - si devono passare i parametri della chiamata
  - al ritorno dalla procedura si devono ritornare i risultati
  - si devono ripristinare (i valori del)le variabili del programma chiamante al ritorno della procedura



(a) Calls and returns (b) Execution sequence

Finestre di registri

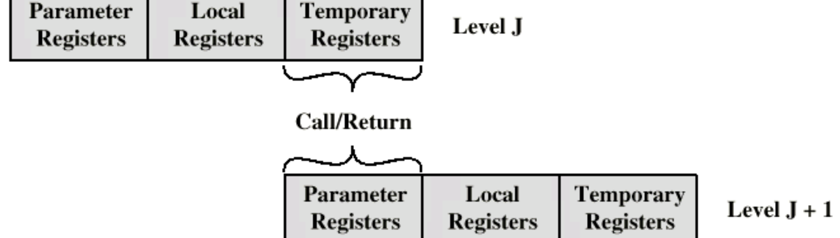
- Osservazioni: tipicamente le chiamate di procedura
  - coinvolgono pochi parametri (< 6 nel 98% dei casi)
  - non presentano grado di annidamento elevato

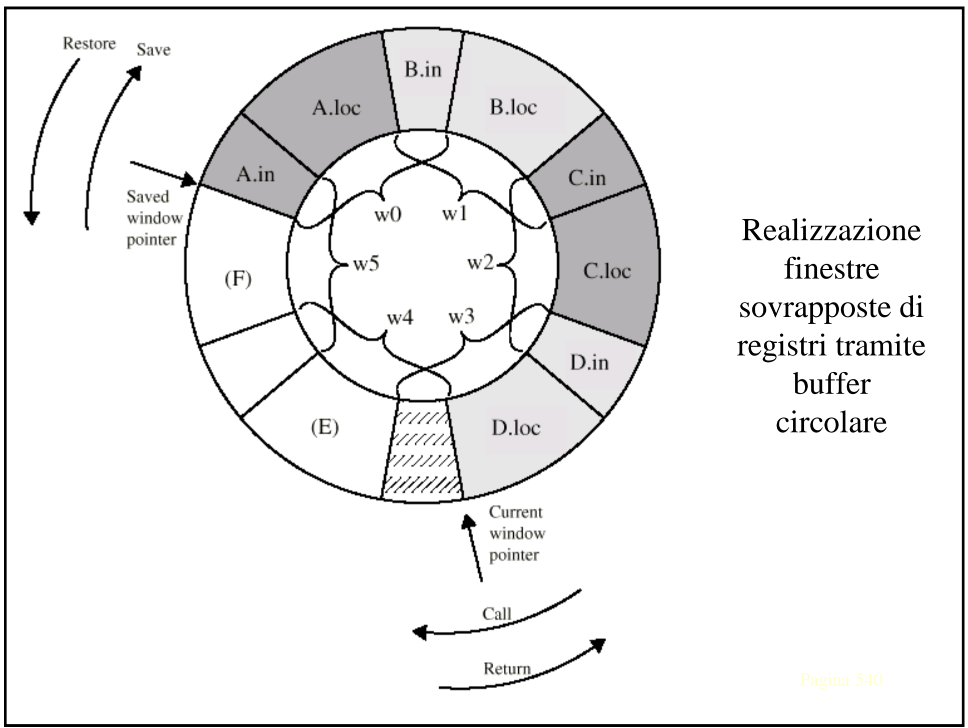
- Suggerisce la seguente soluzione: usare molti gruppi (con cardinalità limitata) di registri:
  - una chiamata di procedura seleziona automaticamente un nuovo gruppo di registri
  - il ritorno da una procedura (ri)seleziona il gruppo di registri assegnato precedentemente alla procedura chiamante

Finestre di registri

- Ogni gruppo di registri suddiviso in tre sottogruppi
  - **registri** che contengono i **parametri** passati alla procedura chiamata
  - **registri** che memorizzano il contenuto delle **variabili locali** alla procedura
  - **registri temporanei**
  - i registri temporanei di un gruppo si sovrappongono perfettamente con quelli che contengono i parametri del gruppo successivo (cioè, sono fisicamente gli stessi registri)
  - Questo permette il passaggio dei parametri senza spostare i dati

Finestre sovrapposte di registri





Operazioni sul buffer circolare

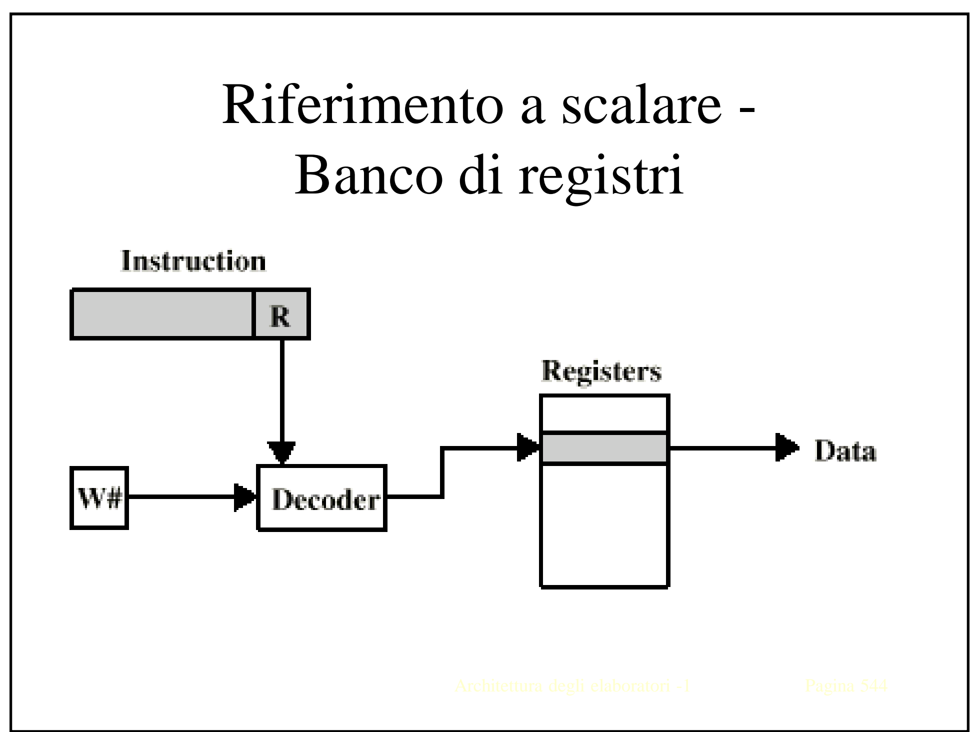
- Quando avviene una chiamata, il puntatore alla finestra corrente (**C**urrent **W**indow **P**ointer) viene aggiornato per mostrare la finestra di registri corrente attiva
- Se si esaurisce la capacità del buffer (tutte le finestre sono in uso a causa di chiamate annidate), viene generata una interruzione e la finestra più “vecchia” viene salvata in memoria principale
- Un puntatore (**S**aved **W**indow **P**ointer) indica dove si deve ripristinare l’ultima finestra salvata in memoria principale

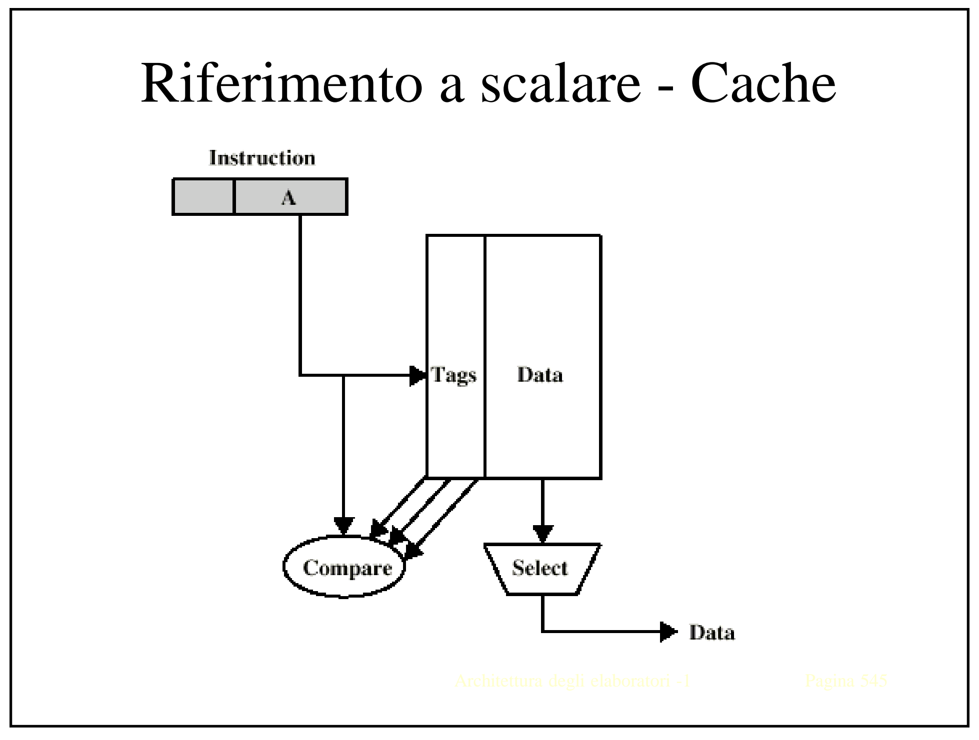
Variabili globali

- Vengono allocate dal compilatore nella memoria
  - scelta inefficiente per variabili riferite frequentemente
- Soluzione: utilizzare un gruppo di registri per memorizzare le variabili globali

Registri “contro” Cache

Banco di Registri Ampio	Cache
Tutti gli scalari locali	Scalari locali utilizzati di recente
Variabili individuali	Blocchi di memoria
Variabili globali assegnate dal compilatore	Variabili globali usate di recente
Save/Restore basato sulla profondità di annidamento delle procedure	Save/Restore basato sull’algoritmo di sostituzione adottato dalla cache
Indirizzamento a registro	Indirizzamento a memoria


Riferimento a scalare -
Banco di registri
Instruction
R
Registers
W#
Decoder
Data

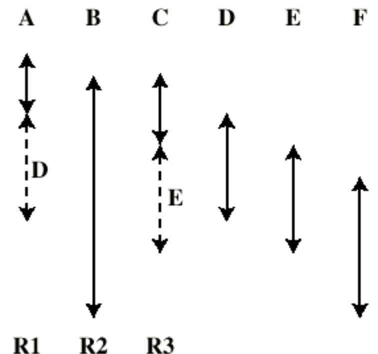

Riferimento a scalare - Cache
Instruction
A
Tags
Data
Compare
Select
Data

Ottimizzazione dei registri tramite compilatore

- Assume un numero limitato di registri (16-32)
- L'ottimizzazione è lasciata al compilatore
- Linguaggi ad alto livello non fanno riferimento esplicito ai registri
  - eccezione in C: `register int`
- Il compilatore assegna un registro simbolico (o virtuale) ad ogni variabile candidata…
- …quindi mappa (un numero virtualmente illimitato di) registri simbolici su registri reali del processore
- Registri simbolici il cui uso non si sovrappone temporalmente possono condividere lo stesso registro reale, cioè possono essere mappati sullo stesso registro reale
- Se i registri reali non sono sufficienti per contenere tutte le variabili riferite in un dato intervallo di tempo, alcune variabili vengono mantenute in memoria principale

Mapping: equivale a risolvere un problema di "colorazione" di un grafo

[problema difficile da risolvere in generale]



B
A
C
D
E
F

(a) Time sequence of active use of registers

(b) Register interference graph

Colorazione di un grafo

- Dato un grafo, costituito da nodi connessi da archi…
- …si assegni un colore per ogni nodo, in modo tale che
  - nodi adiacenti (connessi da un arco) abbiano colori diversi
  - Si usi il numero minore possibile di colori
- Nel nostro caso, i nodi corrispondono a registri simbolici
- Due registri che sono "in vita" all'interno di uno stesso frammento di codice sono connessi da un arco
- Idea di fondo: colorare il grafo con *n* colori, dove *n* è il numero di registri reali
- Nodi che non possono essere colorati sono memorizzati in memoria principale

Valutazione critica del CISC

- Semplifica il compilatore ?
  - Controverso …
  - Istruzioni macchina complesse difficili da sfruttare
  - Ottimizzazione più difficile
- Programmi più piccoli ?
  - I programmi occupano meno memoria, ma …
  - La memoria è diventata economica
  - Possono non occupare meno bit, ma semplicemente sembrano più corti in forma simbolica (codice mnemonico)
    - numero maggiore di istruzioni → codici operativi più lunghi
    - riferimenti a registri richiedono meno bit …

Valutazione critica del CISC

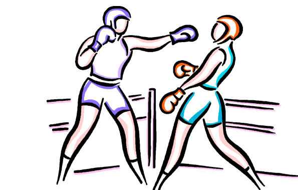
- Esecuzione dei programmi più veloce ?
  - Si tende ad usare instruzioni più semplici
  - Unità di controllo più complessa
  - Controllo microprogrammato necessita di più spazio …
  - … e quindi le istruzioni più semplici (e più usate) diventano più lente
- Non è ovvio che una architettura CISC sia la soluzione migliore

Caratteristiche RISC

- Un’istruzione per ciclo
- Operazioni da registro a registro
- Pochi e semplici modi di indirizzamento
- Pochi e semplici formati per le istruzioni
- Formati fissi per le istruzioni
- Controllo cablato (hardware, meno flessibile ma più veloce) e non microprogrammato (più flessibile ma meno veloce)
- Maggior utilizzo della ottimizzazione a livello del compilatore

RISC “contro” CISC

- Non emerge un “vincitore” netto
- Molti processori utilizzano idee da entrambe le filosofie:
  - ad esempio, PowerPC e Pentium II



Controversia tra RISC e CISC

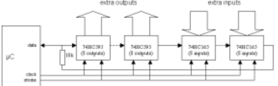
- Criterio quantitativo
  - Paragonare dimensione dei programmi e loro velocità
- Criterio qualitativo
  - Esame del merito nel supportare linguaggi ad alto livello o l’ottimizzazione dell’area di integrazione del chip
- Problemi
  - Non esistono architetture RISC e CISC che siano direttamente confrontabili
  - Non esiste un set completo di programmi di test
  - Difficoltà nel separare gli effetti dovuti all’hardware rispetto a quelli dovuti al compilatore
  - Molti confronti sono stati svolti su macchine prototipali e semplificate e non su macchine commerciali
  - Molte CPU commerciali utilizzano idee provenienti da entrambe le filosofie

Esempio di architettura RISC: famiglia MIPS

- Riferimenti bibliografici: **Stallings + Hennessy & Patterson**
- Architettura molto regolare con insieme di istruzioni semplice e compatto
- Architettura progettata per una implementazione efficiente della pipeline (lo vedremo più avanti)
- Codifica delle istruzioni omogenea: 32 bit
- Co-processore per istruzioni a virgola mobile e gestione delle eccezioni

MIPS (a 32 bit)

Registri



- 32 registri di 32 bit (registro 0 contiene sempre il valore 0)
- Architettura Load / Store
  - Istruzioni di trasferimento per muovere i dati tra memoria e registri
  - Istruzioni per la manipolazione di dati operano sui valori dei registri
  - Nessuna operazione memoria ↔ memoria
- Quindi: le istruzioni operano su registri (registro i riferito con $i)
- Esempio: add $1, $2, $3

MIPS

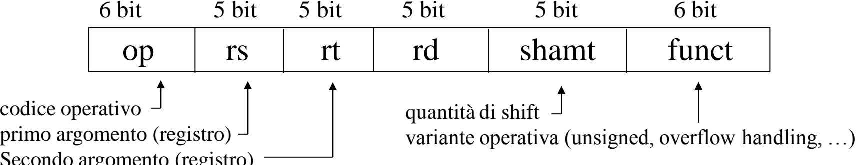
Dati e modi di indirizzamento

- Registri possono essere caricati con byte, mezze parole, e parole (riempiendo con 0 quando necessario o estendendo, cioè replicando, il segno sui bit non coinvolti del registro)
- Modalità di indirizzamento ammesse (con campi di 16 bit):
  - Immediata es. add $2, $2, 0004
  - Displacement es. sw $1, 000c($1)
- Altre modalità derivabili:
  - Indiretta registro (displacement a 0) es. sw $2, 0000($3)
  - Assoluta (registro 0 come registro base) es. lw $1, 00c4($0)

MIPS

Formato Istruzioni

- 32 bit per tutte, 3 formati diversi (formato R, formato I, formato J)
- **Formato R (registro)**



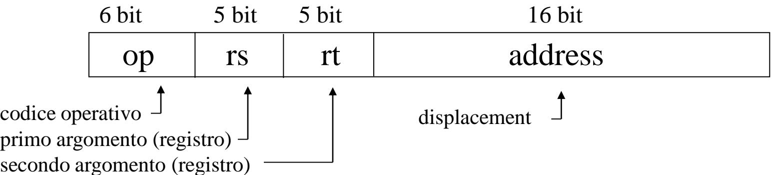
Es: `0x02484020` (`add $8,$18,$8` `[R8] ← [R18] + [R8]`)

op	rs	rt	rd	shamt	funct
000000	10010	01000	01000	00000	100000

MIPS

Formato Istruzioni

- **Formato I (istruzioni load / store)**



Es: **`0x8D2804B0`** (**`lw $8, 1200($9)`** **`[R8] ← Mem[1200+[R9]]`**)

op	rs	rt	address
100011	01001	01000	0000 0100 1011 0000

MIPS

Formato Istruzioni

- **Formato J (istruzioni jump)**

6 bit	26 bit
op	address

codice operativo ↑ indirizzo ↑

Es: **`0x0800AFFE`** (**`j 45054`** **`[PC] ← 45054`**)

op	address
000010	00 0000 0000 1010 1111 1111 1110

Fasi (MIPS)

Fasi senza pipeline:

IF (istruction fetch):

- IR ← Mem[PC] ;
- NPC ← PC + 4 ;

Dove NPC è un registro temporaneo

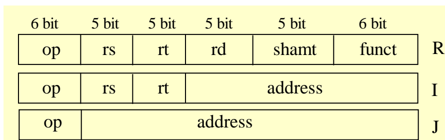
PC è il program counter

Fasi (MIPS)

ID (instruction decode/register fetch cycle):

- A ← Regs[rs] ;
- B ← Regs[rt] ;
- Imm ← campo immediato di IR con segno esteso ;

Dove A, B, Imm sono registri temporanei



Fasi (MIPS)

EX (execution/effective address cycle):

1. ***Riferimento a memoria*** `lw $8, 1200($9)`
   - ALUOutput ← A + Imm ;
2. ***Istruzione ALU registro-registro*** `add $8,$18,$8`
   - ALUOutput ← A ***func*** B ;
3. ***Istruzione ALU registro-immediato*** `addi $8,$18,4`
   - ALUOutput ← A ***op*** Imm ;
4. ***Salto*** `j 45054`
   - ALUOutput ← NPC + (Imm << 2); (shift a sinistra)
   - Cond ← (A = = 0) ;

6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	
op	rs	rt	rd	shamt	funct	R
op	rs	rt	address			I
op	address					J

Fasi (MIPS)

MEM (memory access/branch completion cycle):

- PC ← NPC ; in tutti i casi

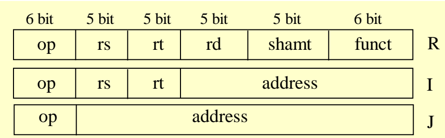
1. ***Riferimento a memoria*** `lw $8, 1200($9)`
   - LMD ← Mem[ALUOutput] or
     Mem[ALUOutput] ← B; `sw $5, 1700($4)`
2. ***Salto***
   - if (Cond) PC ← ALUOutput ; `j 45054`

6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	
op	rs	rt	rd	shamt	funct	R
op	rs	rt	address			I
op	address					J

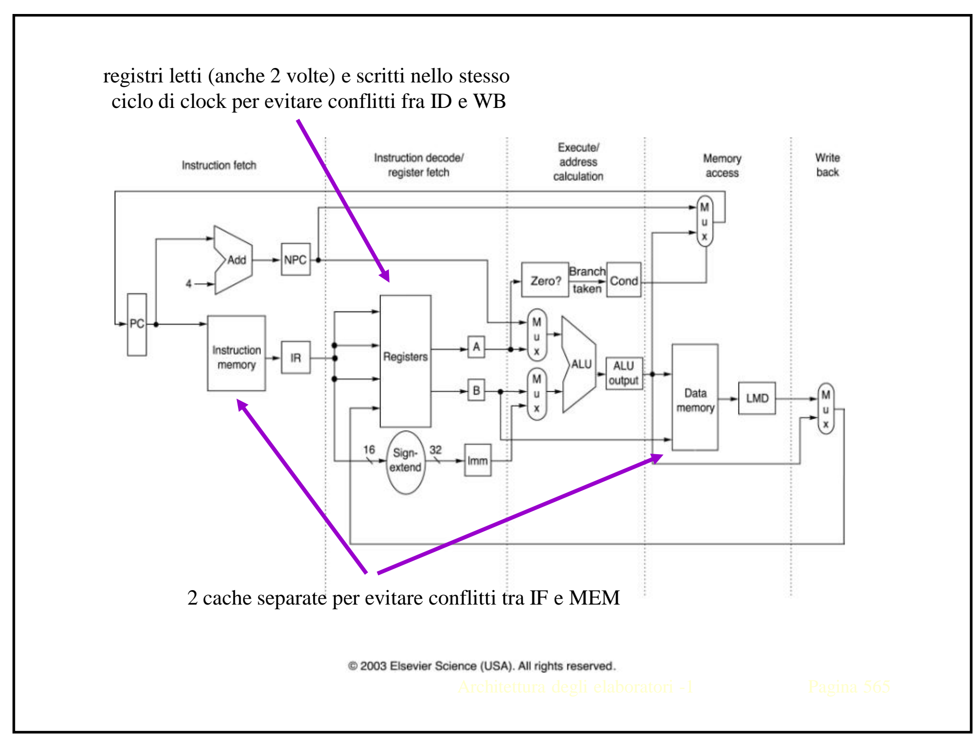
Fasi (MIPS)

WB (write/back cycle):

1. ***Istruzione ALU registro-registro*** `add $8,$18,$8`
   - Regs[rd] ← ALUOutput ;
2. ***Istruzione ALU registro-immediato*** `addi $8,$18,4`
   - Regs[rt] ← ALUOutput ;
3. ***Istruzione Load*** `lw $8, 1200($9)`
   - Regs[rt] ← LMD ;

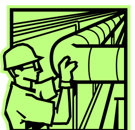


6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	
op	rs	rt	rd	shamt	funct	R
op	rs	rt	address			I
op	address					J



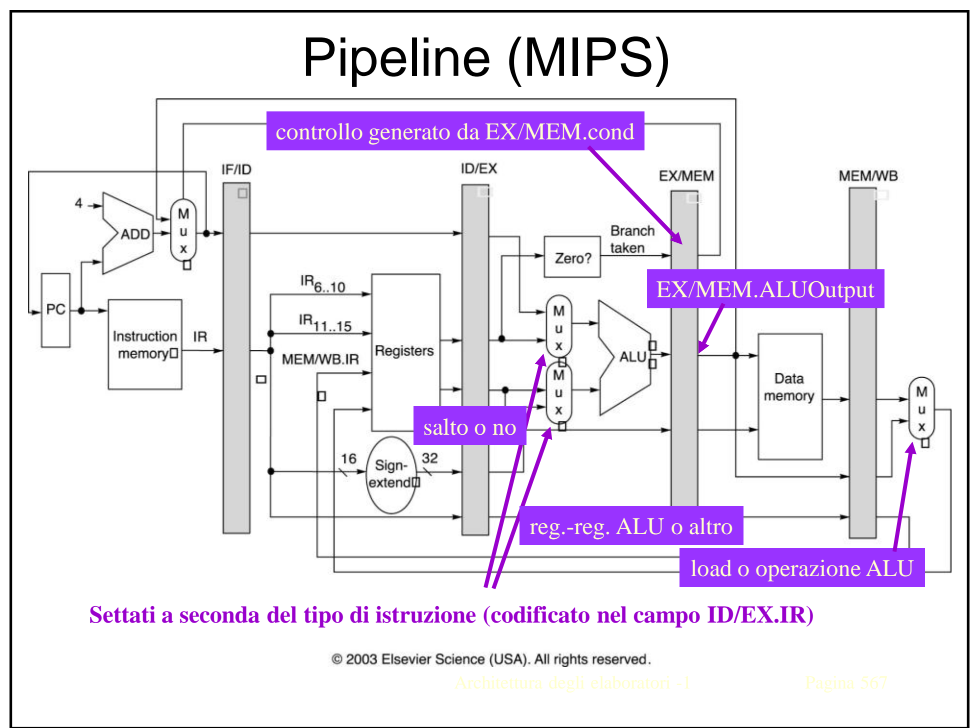
© 2003 Elsevier Science (USA). All rights reserved.

Pipeline (MIPS)



- Architettura che si presta ad una facile introduzione della pipeline: uno stadio per fase, 1 ciclo di clock per stadio
- Occorre memorizzare i dati fra una fase e la successiva: si introducono opportuni registri (denominati pipeline registers o pipeline latches) fra i vari stadi della pipeline
- Tali registri memorizzano sia dati che segnali di controllo che devono transitare da uno stadio al successivo
- Dati che servono a stadi non immediatamente successivi vengono comunque copiati nei registri dello stato successivo per garantire la correttezza dei dati

Pipeline (MIPS)



© 2003 Elsevier Science (USA). All rights reserved.

Pipeline (MIPS)

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC] IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) && EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend (IF/ID.IR[immediate field]);		
	ALU instruction	**Load or store instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOputput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

Pipeline (MIPS)

- Quando una istruzione passa dalla fase ID a quella EX si dice che la istruzione è stata “rilasciata” (issued)
- Nella Pipeline MIPS **è possibile individuare tutte le dipendenze dai dati nella fase ID**
- Se si rileva una dipendenza dai dati per una istruzione, questa **va in stallo prima di essere rilasciata**
- Inoltre, sempre nella fase ID, è possibile determinare che tipo di **data forwarding** adottare per evitare lo stallo ed anche predisporre gli opportuni segnali di controllo
- Vediamo di seguito come realizzare un forwarding nella fase EX per una dipendenza di tipo RAW (Read After Write) con sorgente che proviene da una istruzione load (load interlock)

Pipeline (MIPS)

Possibili casi

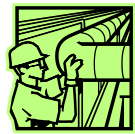
Situazione	Esempio di codice	Azione
Nessuna dipendenza	LD **$1**, 45($2) DADD $5, $6, $7 DSUB $8, $6, $7 OR $9, $6, $7	Non occorre fare nulla perché non c'è dipendenza rispetto alle 3 istruzioni successive
Dipendenza che richiede uno stallo	LD **$1**, 45($2) DADD $5, **$1**, $7 DSUB $8, $6, $7 OR $9, $6, $7	Opportuni comparatori rilevano l'uso di $1 in DADD ed evitano il rilascio di DADD
Dipendenza risolvibile con un forwarding	LD **$1**, 45($2) DADD $5, $6, $7 DSUB $8, **$1**, $7 OR $9, $6, $7	Opportuni comparatori rilevano l'uso di $1 in DSUB e inoltrano il risultato della load alla ALU in tempo per la fase EX di DSUB
Dipendenza con accessi in ordine	LD **$1**, 45($2) DADD $5, $6, $7 DSUB $8, $6, $7 OR $9, **$1**, $7	Non occorre fare nulla perché la lettura di $1 in OR avviene dopo la scrittura del dato caricato

Pipeline (MIPS)

Condizioni per riconoscere le dipendenze

Opcode (ID/EX)	Opcode (IF/ID)	Matching operand fields
Load	R-R ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	R-R ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, Store, Imm ALU, branch	ID/EX.IR[rt] == IF/ID.IR[rs]

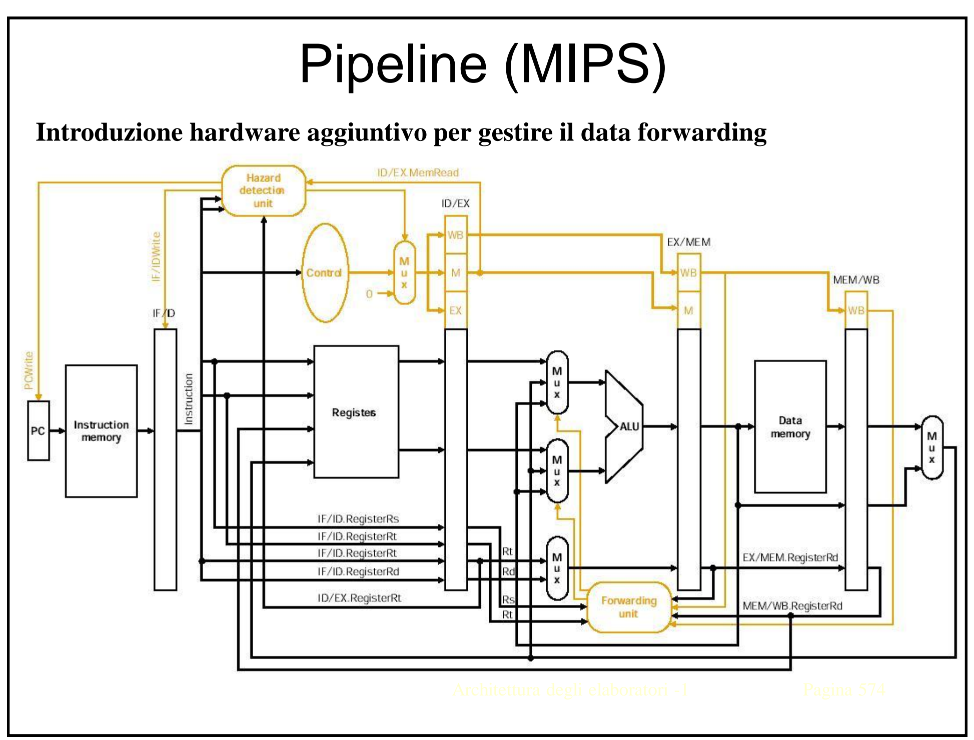
Pipeline (MIPS)

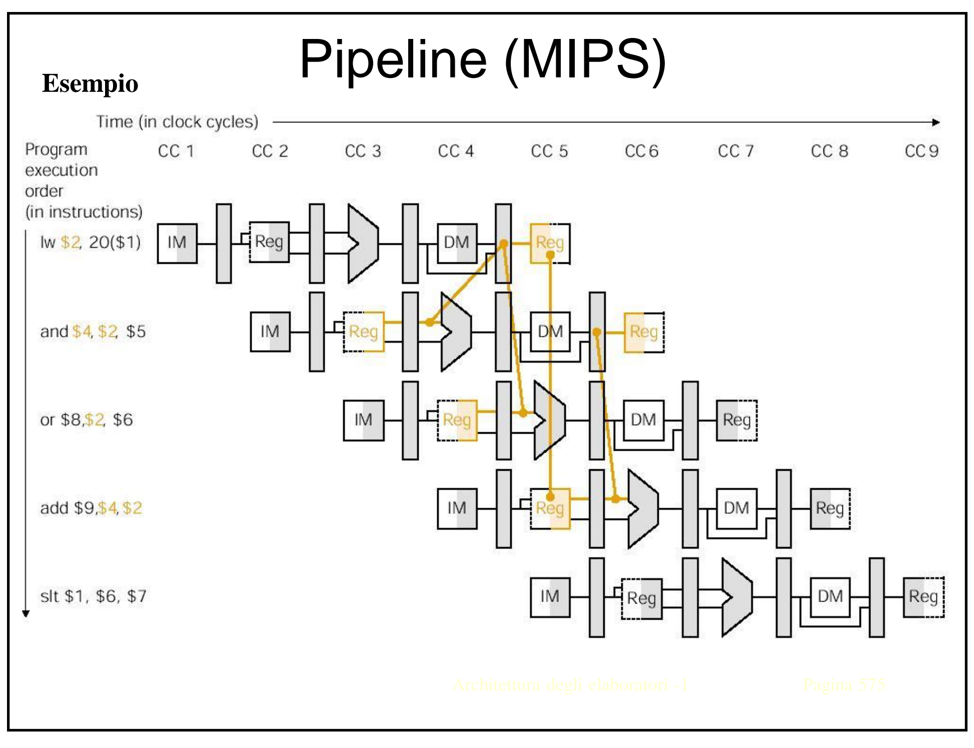


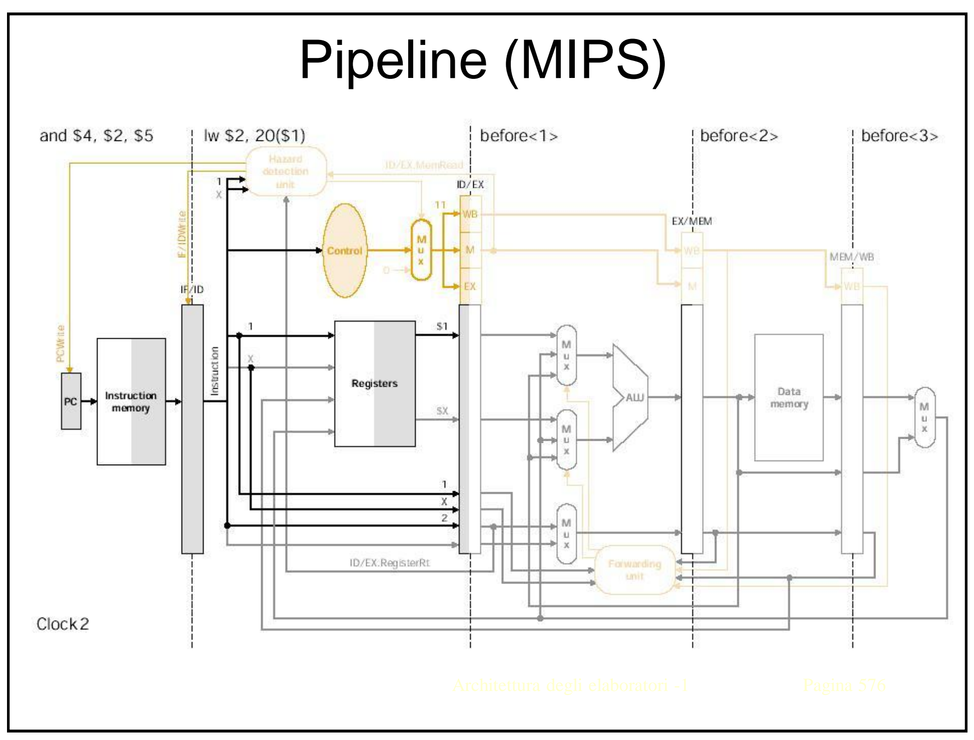
- La logica per decidere come effettuare il forwarding è simile a quella appena vista per individuare le dipendenze, ma considera molti più casi
- Una osservazione chiave è che i registri di pipeline contengono:
  - dati su cui effettuare il forwarding
  - i campi registro sorgente e destinazione
- Tutti i dati su cui effettuare il forwarding provengono:
  - dall'output della ALU
  - dalla memoria dati
- … e sono diretti verso:
  - l'input della ALU
  - l'input della memoria dati
  - il comparatore con 0
- Quindi occorre confrontare i registri destinazione di IR in EX/MEM e MEM/WB con i registri sorgente di IR in ID/EX e EX/MEM

Pipeline (MIPS)

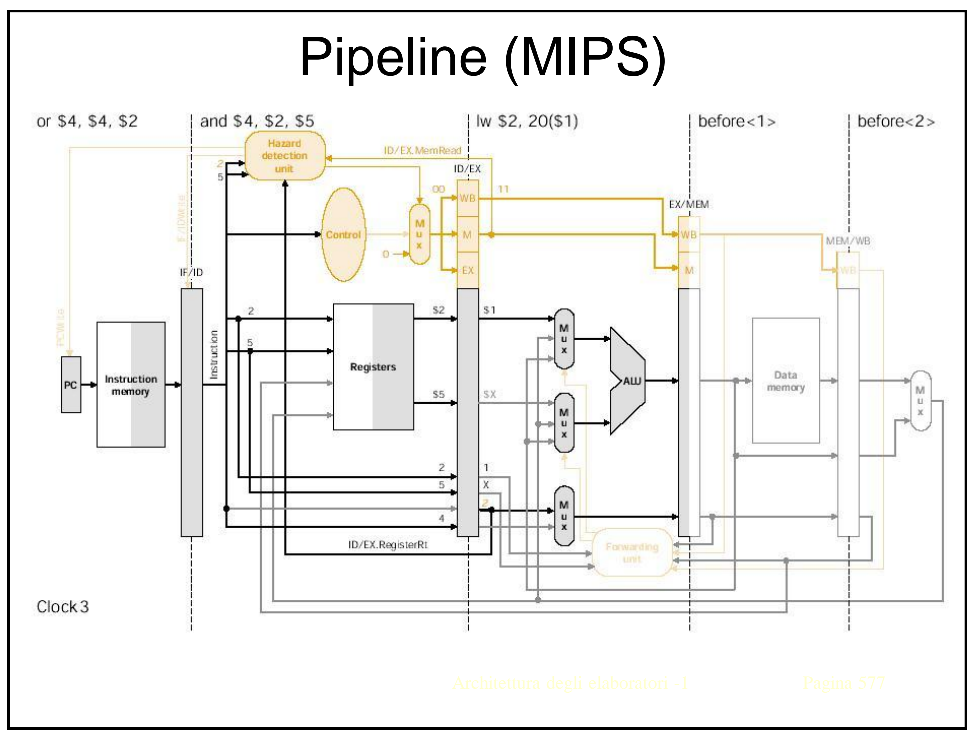
Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rt] == ID/EX.IR[rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rt] == ID/EX.IR[rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]


Pipeline (MIPS)
Introduzione hardware aggiuntivo per gestire il data forwarding
Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
0
M u x
WB
M
EX
EX/MEM
MEM/WB
IF/ID
PCWrite
PC
Instruction memory
Instruction
Registers
ALU
Data memory
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
ID/EX.RegisterRt
Rt
Rd
Rs
Rt
Forwarding unit
EX/MEM.RegisterRd
MEM/WB.RegisterRd

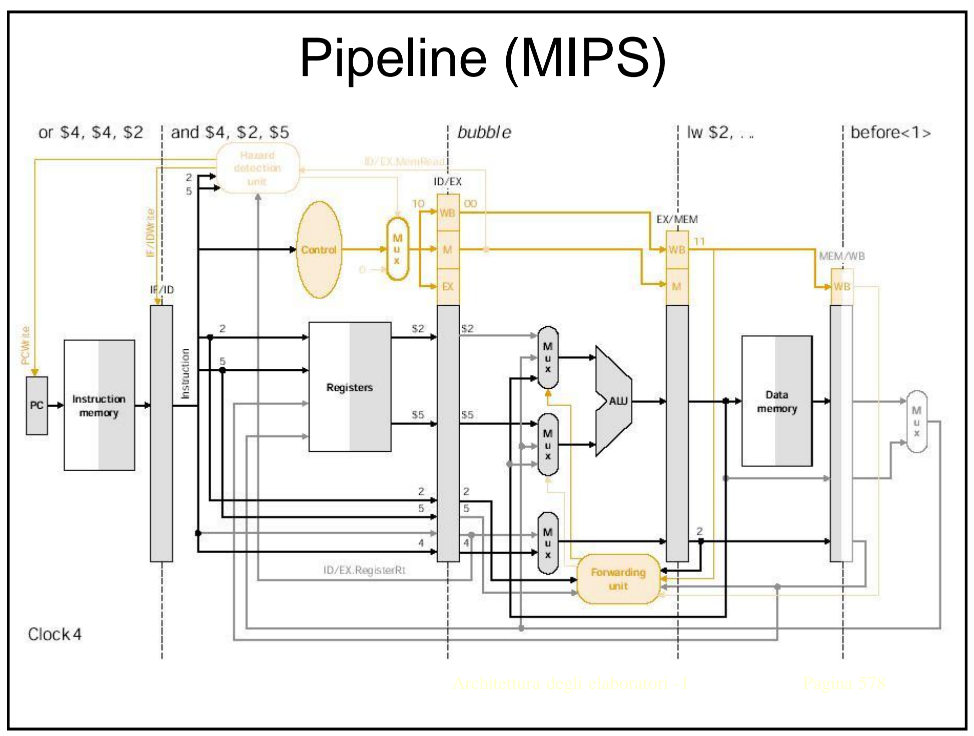

Pipeline (MIPS)
Esempio
Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
lw $2, 20($1)
and $4, $2, $5
or $8,$2, $6
add $9,$4, $2
slt $1, $6, $7
IM
Reg
DM



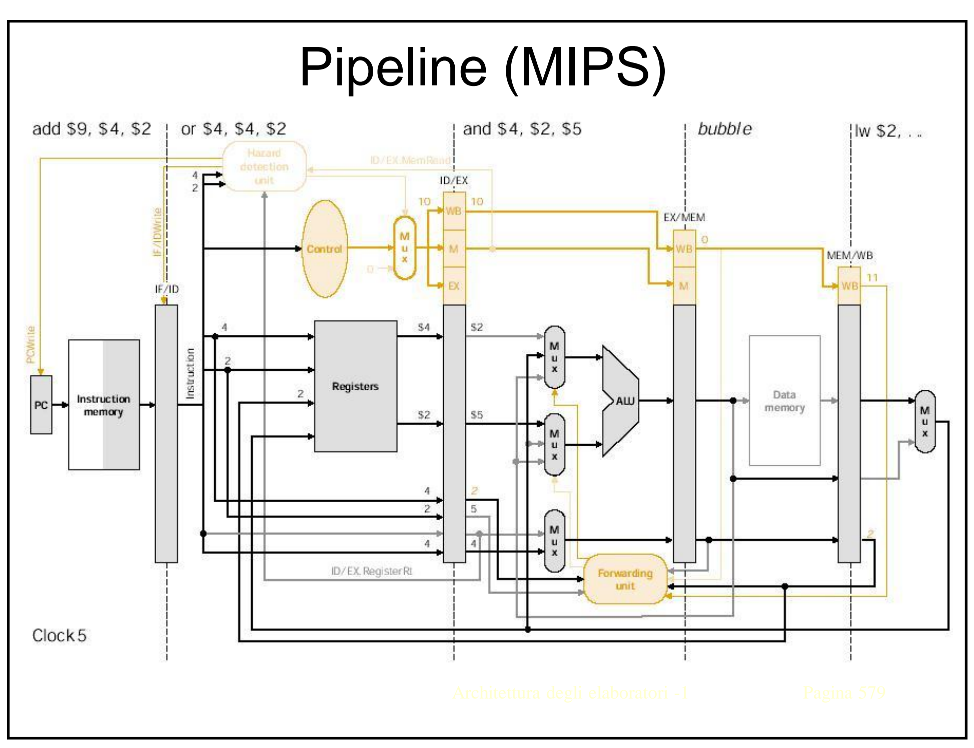
Pipeline (MIPS)
and $4, $2, $5
lw $2, 20($1)
before<1>
before<2>
before<3>
Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
Mux
WB
M
EX
EX/MEM
MEM/WB
IF/ID
PCWrite
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 2



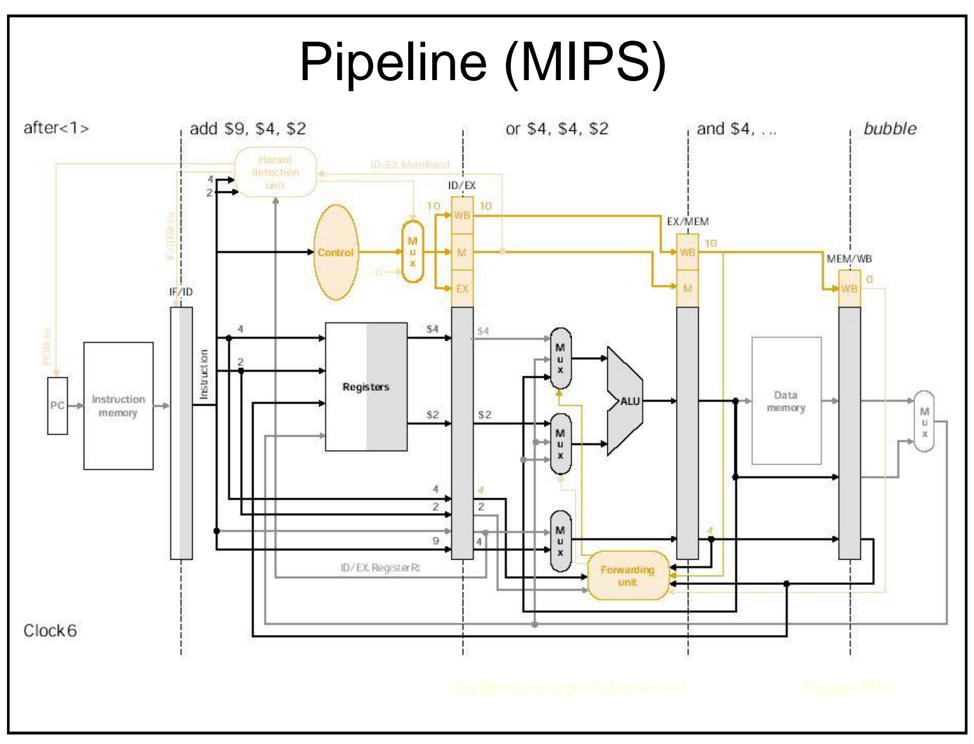
Pipeline (MIPS)
or $4, $4, $2
and $4, $2, $5
lw $2, 20($1)
before<1>
before<2>
Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
Mux
WB
M
EX
EX/MEM
MEM/WB
IF/ID
PCWrite
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 3

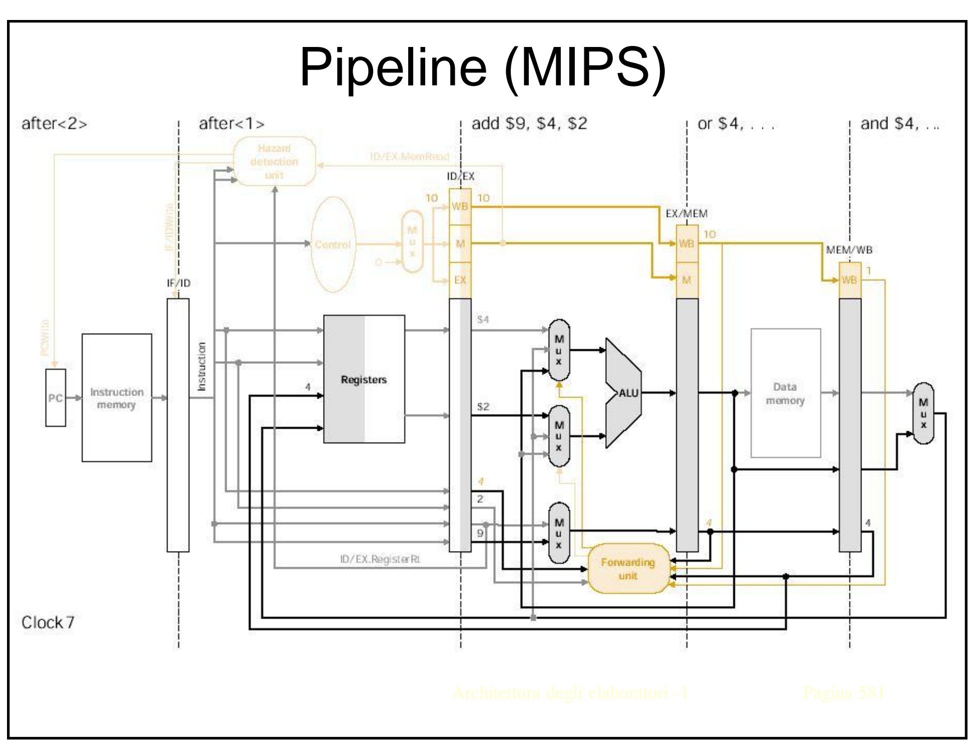


Pipeline (MIPS)
or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
before<1>
Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
EX/MEM
MEM/WB
PCWrite
IF/ID
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 4



Pipeline (MIPS)
add $9, $4, $2
or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
EX/MEM
MEM/WB
PCWrite
IF/ID
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 5


Pipeline (MIPS)
after<1>
add $9, $4, $2
or $4, $4, $2
and $4, . . .
bubble
Clock 6


Pipeline (MIPS)
after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
Clock 7