Appunti AdE

- Memorie a semiconduttore
- Dischi rigidi
- RAID
- Metodi di indirizzamento
- Pipeline
- Pipeline Hazards
- Proprietà architetture
- Architetture CISC e RISC
- MIPS Instruction set

Memorie a semiconduttore

MEMORIE A SEMICONDUTTORE

Hanno la possibilità di memorizzare e leggere valori che possono assumere gli stati 0 e 1.
Sono ad accesso casuale: permettono la lettura tramite circuito dedicato.
Possono essere:

- **RAM** (RANDOM ACCESS MEMORY)
  Permettono una lettura/scrittura veloce, tramite segnali elettrici e richiedono alimentazione per il mantenimento dei dati (VOLATILITÀ).
  Possono essere di tipo:

  - DINAMICA (DRAM)
    Chiamate dinamiche perché composte da condensatori che nel tempo lasciano decadere la carica anche se alimentati, rendendo necessario un circuito di refresh.
    Presenza di carica indica lo stato 1, assenza indica 0.

    L'operazione di scrittura viene eseguita applicando tensione alle linee bit (l'intensità determina il valore del bit) e alla linea indirizzo, trasferendo la carica al condensatore.
    In lettura si seleziona la linea indirizzo e la carica viene trasferita dalla linea bit ad un amplificatore che confronta la tensione con un valore di riferimento per determinare lo stato; viene poi effettuato un refresh per ripristinare la carica nel condensatore.

  - STATICA (SRAM)
    Sono volatili ma non necessitano di operazioni di refresh in quanto composte da porte logiche stabili, formate da transistor.
    La lettura/scrittura avviene dei/sui transistor tramite la linea indirizzo.

  Confronto:
  Entrambe sono memorie volatili ovvero richiedono una continua alimentazione per conservare i valori dei bit.
  Una cella di memoria dinamica (2 condensatori) è più semplice di una cella statica (2 porte logiche), rendendo le DRAM più dense e meno costose.
  Le DRAM sono favorite per grandi capacità (come la mem. centrale) considerando anche il lento tempo di accesso e la necessità di un circuito di refresh mentre le SRAM vengono utilizzate per le cache.

- **ROM** (READ ONLY MEMORY)
  Hanno uno schema di dati definito in produzione e non modificabile ma solo leggibile e non richiedono alimentazione per mantenere i dati (NON-VOLATILITÀ).
  Dati e programmi di piccole dimensioni non devono quindi essere caricati da dispositivi esterni.

  Esistono varianti di ROM programmabili una sola volta (ProgrammableROM) o "principalmente di lettura" (v. più frequente di W.):
  - EPROM: programmabili otticamente, richiedono cancellazione completa per modifica
  - EEPROM: programmabili elettricamente anche piccole porzioni di byte
  - Flash: programmazione elettrica a velocità notevole.

Dischi rigidi

DISCHI RIGIDI

Nei dischi magnetici la lettura e scrittura avviene tramite la bobina conduttiva della testina (HEAD) che passa sopra al disco rotante, testina muove radialmente. I dati sono disposti in anelli concentrici (TRACCE) con la larghezza della testina. Tracce adiacenti sono separate da spazi (GAPS).
Il trasferimento avviene per SETTORI, un contiguo per traccia, anche questi separati da spazi.

Lo spazio tra i bit di tracce differenti può aumentare per mantenere una velocità di lettura costante dato che i bit al centro vanno più lenti rispetto a quelli periferici.
Per mantenere la stessa densità si può in alternativa utilizzare un motore sofisticato che cambi velocità a seconda della posizione della testina.

Formattare un disco significa renderlo idoneo all'archiviazione determinando dei criteri per identificare inizio e termine di settori e tracce.

Un cilindro considera le tracce nella stessa posizione su dischi diversi rendendo possibile la lettura simultanea da più superfici.
Ordine selezione cilindro:
- FIRST COME FIRST SERVED: ordine di arrivo
- SHORTEST SEEK FIRST: servita la richiesta più vicina al cilindro corrente
- ELEVATOR ALGORITHM: servito cilindro più vicino senza cambiare direzione

$$T_t = \frac{b}{r \cdot N}$$

T_t tempo trasferimento in secondi
b # byte da trasferire
N # byte per traccia
r vel. rotazione [rps]

$T_{ACCESSO} = T_{POSIZIONAMENTO}(SEEK) + T_{LATENZA}(LATENCY)$ $\quad T_{TOT} = T_S + T_L + T_t$

$T_L = (1000 / (rpm/60))/2$ ← metà tempo rotazione, media tra caso migliore (testina poco prima settore) e peggiore (testina poco dopo settore)
(rpm/60): giri al secondo; 1000 / (rpm/60): ms per giro

es.1 DISCHI

Trasferire 64KB

$T_{TOT} = 9,728571$ ms

524288 tracks

512B/settore

T_S 0,8 ms

r 4200 rpm = 70 rps

B memorizzabili?

$T_L = (1000/70)/2 \cong 7,142857$ ms

$T_t = \frac{b}{r \cdot N}$

$N = \frac{b}{T_t \cdot r}$

$T_t = T_{TOT} - T_S - T_L$
$= 9,728571 - 0,8 - 7,142857 \cong 1,785714$ ms

$= \frac{64K}{1,785714 \cdot 70} \cdot 1000 = 524288$

Capacità $= 524288 \cdot 524288 \cong 274$ GB

RAID

RAID (REDUNDANT ARRAY of INDIPENDENT DISKS)

Sistema utile a aumentare le prestazioni del calcolatore tramite l'utilizzo di più dischi in parallelo garantendo una gestione veloce delle operazioni I/O ed implementando la ridondanza che rende più sicuri i dati e presenta un immediato backup in caso di guasti.

- RAID 0: non implementa ridondanza, dati organizzati in strisce (STRIPS) distribuite a rotazione su ciascun disco (ROUND ROBIN).
  Dati non recuperabili in caso di guasto
  Richieste servite in parallelo in caso di blocchi in dischi diversi

- RAID 1: ridondanza ottenuta scrivendo le strisce di dati su due dischi (MIRRORING). In caso di guasto i dati sono accessibili in uno dei due dischi.
  La lettura viene effettuata dal disco con minor latenza mentre la scrittura è condizionata dal disco con minori prestazioni
  Grande affidabilità ma alti costi.

- RAID 2: configurazione non commerciale - sfrutta l'accesso in parallelo: le testine di dischi si trovano nella stessa posizione in ogni momento. Strisce molto piccole (byte o parole), generati codici di correzione di Hamming e salvati su più dischi rendendo tempo di correzione minimo.

- RAID 3: solo un disco ridondante, accesso in parallelo e striping
  Calcolo bit di parità su stessa posizione di tutti i dischi utile all'eventuale ricostruzione di dati tramite XOR con informazioni di parità.
  Trasferimento veloce ma serve solo una richiesta alla volta.

- RAID 4: non commercializzato - dischi indipendenti, striping strisce grandi con strisce di parità salvate su corrispondente disco di parità che può diventare collo di bottiglia.

- RAID 5: simile a RAID 4, strisce di parità distribuite su tutti i dischi tramite algoritmo ROUND ROBIN evitando collo di bottiglia

- RAID 6: vengono effettuati due diversi calcoli della parità memorizzati su dischi differenti permettendo la rigenerazione dei dati anche alla perdita di due dischi.
  Alta disponibilità dei dati ma penalità in scrittura

Metodi di Indirizzamento

METODI DI INDIRIZZAMENTO

- **IMMEDIATO:** l'operando è parte dell'istruzione, non sono quindi necessari accessi in memoria ma il valore è limitato alla dimensione del campo indirizzo
- **DIRETTO:** il campo indirizzo è riempito con l'indirizzo in M dell'operando, singolo accesso in M, indirizzamento limitato da lunghezza istr.
- **INDIRETTO:** il campo indirizzo contiene un indirizzo di M contenente l'indirizzo dell'operando.
  Si ha quindi più capacità di indirizzamento ma sono necessari due accessi in memoria.
- **REGISTRO:** l'operando è in un registro, necessari pochi bit per indirizzare, accesso veloce e nessun accesso in M ma numero di registri limitato.
- **REGISTRO INDIRETTO:** il campo indirizzo contiene l'indirizzo di un registro che punta alla locazione M dell'operando.
  Grande possibilità di indirizzamento (2^N) e necessita di un accesso in M in meno rispetto all'indirizzamento indiretto.
- **SPIAZZAMENTO:** il campo indirizzo contiene un indirizzo M base al quale viene sommato lo spiazzamento contenuto dal registro indicato per ottenere l'indirizzo completo dell'operando. (o viceversa)
- **RELATIVO:** operando indirizzabile sommando l'indirizzo nel program counter (prossima istruzione) ad uno spiazzamento specificato.
- **REGISTRO-BASE:** indirizzo operando dato dalla somma di uno spiazzamento specificato e l'indirizzo base M contenuto in un registro specificato. Il registro può essere o meno esplicitato.
- **INDICIZZAZIONE:** gli operandi sono memorizzati in M a partire da un indirizzo specificato nell'istruzione, l'indice del dato è contenuto in un registro specificato il quale valore incrementa ad ogni accesso in M.
- **PILA:** implementazione di registro indiretto, gli operandi sono nella cima di una pila (serie contigua di locazioni di M) puntata da un registro specificato che funge da stack pointer.

- **TIPI DI ISTRUZIONI:**
  - Elaborazione dati - istruzioni aritmetico-logiche
  - Trasferimento dati tra registri, o in memoria o da I/O
  - Controllo del flusso del programma (salto in/condizionato)

Pipeline

PIPELINE: è un sistema utilizzato per completare più istruzioni in un periodo di tempo (maggiore throughput) scomponendo il lavoro in fasi da eseguire in sequenza.
Le attività delle istruzioni vengono eseguite in PARALLELO da più esecutori: nello stesso istante istruzioni diverse stanno in fasi diverse e nell'istante successivo ogni fase ripeterà lo stesso passaggio sull'istruzione successiva.

La suddivisione in fasi richiede un'attenta progettazione e analisi del trade-off in quanto un aumento del numero delle fasi comporta anche un aumento della complessità logica e di overhead.
L'overhead, ovvero il carico aggiuntivo di dati utili al cambiamento di fase che avviene tramite buffer, diventa particolarmente significativo in caso di dipendenze logiche tra istruzioni, salti e conflitti nell'utilizzo di risorse.

Suddivisione fasi fondamentali:
- FETCH INSTRUCTION (IF): prelievo istruzione
- INSTRUCTION DECODE (ID): decodifica istruzione
- FETCH DATA (CO-FO): prelievo operandi
  - CALCOLO IND. OPERANDI (CO)
  - FETCH OPERANDI (FO)
- EXECUTE INSTRUCTION (EX): esecuzione (EI)
- WRITE OPERAND (WO): memorizzazione dati

→ Ordini e comandi precisi dipendono da instruction set e architettura

Il parallelismo della pipeline è TOTALE se tutti gli esecutori lavorano a pieno regime con istruzioni indipendenti tra loro, situazione ideale

- **PREFETCH**: l'istruzione successiva può essere prelevata mentre l'istruzione corrente è in esecuzione in ALU se non ci sono conflitti tra le componenti coinvolte.
  Le prestazioni non raddoppiano perché, sebbene la fase di fetch risulta più breve, deve comunque essere terminata prima la fase di esecuzione.
  L'operazione è inoltre resa vana in caso di jump e branch.

- **FORWARDING**: Alla rilevazione di una dipendenza di dati, appositi circuiti e sistemi di bypass e multiplex inoltrano il risultati dall'uscita della ALU alla sua entrata per evitare cicli di stalla.
  Nell'architettura MIPS è implementata da tre circuiti:
  - FORWARDING UNIT: stabilisce se attuare il forward e attiva i MUX.
  - HAZARD DETECTION UNIT: riconosce le dipendenze e nei casi irrisolvibili genera stalli.
  - CONTROL UNIT: genera segnali di controllo per dirigere il forward
  Il MIPS supporta forward da EX a EX e da MEM a EX.

Pipeline Hazards

PIPELINE HAZARDS: problemi delle istruzioni in CPU che avvengono quando non è possibile eseguire l'istruzione caricata successiva alla corrente nel ciclo immediatamente successivo per evitare la produzione di risultati scorretti; fenomeni che pregiudicano il raggiungimento del parallelismo totale.
(CRITICITÀ)

Possono essere comportati da:
(es. accesso a R o a M)

- SBILANCIAMENTO DELLE FASI: le fasi possono richiedere diversi tempi di esecuzione anche a seconda dell'istruzione, rendendo possibile lo scenario in cui un esecutore resta in attesa forzata perché l'esecutore della fase precedente deve ancora passargli i dati. (invece scrivere i risultati su un registro intermedio tra le due fasi).
  → SOLUZIONI: • Scomposizione fasi onerose in più sottofasi (costose)
  • Implementazione di più esecutori in parallelo

STRUCTURAL HAZARDS

- PROBLEMI STRUTTURALI: Concorrenza tra più istruzioni nell'accesso alla stessa risorsa nello stesso ciclo di clock (es. FI, FO, WO)
  → SOLUZIONI: • Introduzione di fasi non operative (NOP/BUBBLE), il ritardo introdotto si replica su tutte le fasi successive.
  • Suddivisione delle memorie permettendo accessi paralleli es. cache istruzioni / cache dati

DATA HAZARDS

- DIPENDENZA DAI DATI: una fase non può essere eseguita in un certo ciclo di clock perché i dati di cui ha bisogno non sono ancora disponibili

Possono presentarsi in situazioni di:
- READ AFTER WRITE (RAW): un'istruzione fa riferimento ad un risultato che non è ancora stato completamente processato e salvato dalla precedente istruzione
- WRITE AFTER READ (WAR): un'istruzione cerca di scrivere in una una locazione prima che l'istruzione precedente abbia letto. L'istruzione precedente troverà valori troppo recenti.
- WRITE AFTER WRITE (WAW): un'istruzione tenta di eseguire una scrittura prima che l'abbia eseguita l'istruzione precedente (caso raro, più frequente in ambienti concorrenti)

→ SOLUZIONI: • Introduzione fasi NOP/BUBBLE, ritardare le fasi garantendo che le istruzioni facciano riferimento ai valori intesi in fase di programmazione.
• Propagazione dei dati (DATA FORWARDING), apposito circuito riconosce le dipendenze e propaga i risultati dall'uscita della ALU alla fase FO dell'istruzione successiva, riducendo il periodo NOP.
• Riordino delle istruzioni (OUT OF ORDER EXEC.): le istruzioni da eseguire non seguono l'ordine impostato dal programmatore ma vengono riordinate dal processore in modo da ridurre le dipendenze

es. add $1, $2, $3 R1 ← R2+R3
RAW
sub $4, $1, $5 R4 ← R1−R5

	1	2	3	4	5	6	7	8
add	FI	DI	CO	FO	EI	WO		
sub		FI	DI	CO	FO	EI	WO	

legge $1 ma deve ancora essere scritto dal W.O. di add

add	FI	DI	CO	FO	EI	WO			
sub		FI	DI	CO	////	////	FO	EI	WO

2 cicli di stallo

valore già disponibile all'uscita dell'ALU, data forwarding

add	FI	DI	CO	FO	EI	WO			
sub		FI	DI	CO	////	////	FO	EI	WO

es. riordino ist.

lw	$1	0	($t0)	
lw	$2	4	($t0)	
add	$3	$1	$2	
sw	$3	12	($t0)	
lw	$4	8	($t0)	
add	$5	$1	$4	
sw	$5	16	($t0)	

⇒ dipendenze RAW ridotte

lw	$1	0	($t0)
lw	$2	4	($t0)
lw	$4	8	($t0)
add	$3	$1	$2
sw	$3	12	($t0)
add	$5	$1	$4
sw	$5	16	($t0)

CONTROL HAZARDS

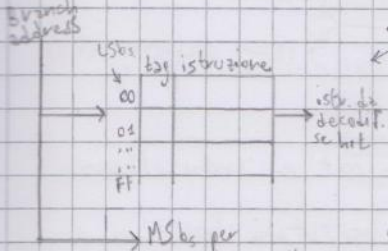
- DIPENDENZE DAL CONTROLLO: date da istruzioni che alterano la sequenzialità ed impediscono al processore di sapere in anticipo quali istruzioni caricare (es. in caso di branch e jump.) e causando lo scarto delle istruzioni successive caricate in caso di salto (BRANCH PENALTY).

oltre a jump abilitata a procedure, ritorni, interrupt

È possibile mettere in stallo la pipeline fino al calcolo della prossima istruzione (INEFFICIENTE) o implementare un'apposita logica di controllo delle istruzioni critiche (aumento complessità compilatore e hardware)

→ SOLUZIONI:

- Flussi multipli (MULTIPLE STREAMS): vengono replicate le pipeline (fino a F.I. esclusa) di entrambi i rami possibili del branch. Possono esserci conflitti di accesso alle risorse e se una delle istruzioni caricate contiene un salto vengono aggiunti ulteriori stream. La pipeline rimane però a regime.
- Prefetch istruzione target: anticipato il fetch dell'istruzione target o dell'istruzione successiva al jump.
- Buffer circolare (LOOP BUFFER): utile in caso di loop, piccola memoria contenente le ultime n istruzioni dalla quale viene verificata la presenza dell'istruzione target. Riduce overhead accessi a memoria
Può essere implementato anche il prefetch, riempiendo quindi il buffer con alcune istruzioni successive alla corrente. (TAG)
Nel buffer i LSBs sono indice buffer, MSBs per verificare presenza.



- Predizione dei salti (BRANCH PREDICTION): si cerca di predire se salto sarà intrapreso attraverso un approccio STATICO (previsione di saltare sempre, mai o in base al codice) oppure un approccio DINAMICO (bit taken/not taken o branch history).
In un approccio dinamico si utilizzano 2 bit dei quali uno indica se saltare o meno, il secondo se l'ultima previsione è stata corretta. Si cambia previsione dopo due errori di seguito.
Alla fase di fetch viene quindi associato un BUFFER DI PREDIZIONE DEI SALTI (o BRANCH HISTORY TABLE): una memoria veloce contenente indirizzi dell'istruzione salto, bits di predizione e indirizzi di destinazione dei salti o l'istruzione destinazione stessa (risparmiando il tempo di decodifica di questa)

previsione effettuata prima dell'esecuzione del programma in base a casi e statistiche

In bit taken/not taken a 2 bit → si cambia predizione ogni due errori di seguito per ridurre gli errori in caso di cicli annidati. Si possono usare più bit per codificare più storia

- Salto ritardato (DELAYED BRANCH): fino a che non si sa se ci sarà o no il salto, invece che restare in stallo il processore esegue un'istruzione opportuna precedente al salto (istruzione che può rivelarsi inutile ma non dannosa), posta dal compilatore (se la trova) nello slot BRANCH DELAY.
In generale, il compilatore sceglie un'istruzione opportuna da inserire nel branch delay slot che viene eseguita mentre si cerca di recuperare l'informazione per sapere se saltare o meno.
Quando non è presente nessuna istruzione indipendente lo slot viene riempito con l'istruzione target, utile se è probabile che il salto venga preso (es. loop) e perché l'istruzione potrebbe essere accessibile da altri cammini.

- BRANCH PENALTY: indica le istruzioni caricate in pipeline che devono essere scartate (PIPELINE FLUSH) a causa del salto ad una nuova locazione data da un branch.
La pipeline viene quindi svuotata e continua il lavoro dall'istruzione target.

Proprietà architetture

- I calcolatori sono caratterizzati da una COEVOLUZIONE (sviluppo parallelo) tra software e hardware.
  Va però mantenuto ridotto il GAP SEMANTICO, ovvero la differenza di complessità tra linguaggio macchina e linguaggi ad alto livello (HLL) che permettono di esprimere l'algoritmo risolutivo in modo semplice e conciso lasciando la gestione dei dettagli al compilatore.

  A livello hw si può implementare un set di istruzioni più ampio e ulteriori modi di indirizzamento oppure agire semplificando la struttura sottostante ai HLL.

  Analisi sofisticate e dinamiche dei programmi scritti in HLL e delle loro performance sono state effettuate per semplificare le fasi di esecuzione (funzionalità processore e interazione con memoria), gestione degli operandi (organizzazione memoria e indirizzamento) e serializzazione dell'esecuzione (pipeline).

→ È stato quindi possibile migliorare le performance dei pattern più frequenti e time-consuming implementando:
  - un ampio numero di registri per ottimizzare gli accessi agli operandi (molto frequente gestione di variabili scalari e locali)
  - una progettazione accurata della pipeline, in particolare delle dipendenze dal controllo (frequenti chiamate a procedure e salti)
  - set di istruzioni semplificato, ridotto

  Variabili SCALARI: contengono un singolo valore (contrapposte a var. vettoriali)
  LOCALI: valore cambia a seconda della località, ad ogni chiamata/return (SCOPE)
  ↓
  Ad ogni chiamata vengono salvate le variabili locali in memoria e ridistribuiti i registri per le nuove variabili locali, vengono passati i parametri. Al termine della procedura vengono ripristinati i valori locali del chiamante nei registri e ritornati i risultati.

- Nel caso dei processori RISC, che dispongono di un grande numero di registri general-purpose, questi vengono raggruppati a piccoli gruppi, ognuno dedicato ad una procedura. Alla chiamata di una procedura viene cambiato automaticamente il gruppo di registri da usare invece che salvare i dati del chiamante in memoria.
  I registri per il passaggio e ritorno dei parametri tra due procedure adiacenti sono fisicamente gli stessi, in modo da facilitare il trasferimento.

  Registri organizzati a BUFFER CIRCOLARE: struttura dati a lunghezza fissa usata per gestire le window.
  Caratterizzato da un CURRENT WINDOW POINTER che punta alla procedura corrente (fine del buffer) e un SAVED WINDOW POINTER che indica dove si deve ripristinare l'ultima finestra salvata in memoria.
  Al riempimento del buffer (cwp = swp mod n finestre) un interrupt salva il buffer, salva le procedure iniziali in memoria e avanza il SWP.
  Al termine di tutte le window un interrupt ripristina le prime window e il SWP viene riportato all'inizio.
  Il buffer circolare non è molto capiente perché il nesting è generalmente di basso livello.

  Variabili GLOBALI: accessibili da tutte le procedure, memorizzate in registri ad hoc

  I registri simbolici/virtuali sono potenzialmente infiniti mentre i registri fisici sono finiti. Vanno quindi ottimizzati in modo che registri simbolici, se usati in momenti diversi, vengano mappati sugli stessi registri reali.

Architetture RISC e CISC

Se in un certo periodo di tempo i registri fisici non sono abbastanza da gestire tutte le variabili necessarie, alcune vengono mantenute in memoria principale

ARCHITETTURA CISC (COMPLEX INSTRUCTION SET COMPUTER)

- Caratterizzata da un set esteso di istruzioni complesse a lunghezza variabile per semplificare il compilatore e migliorare le performance. Riempire il gap semantico risulta comunque complesso per il compilatore perché l'ottimizzazione del linguaggio macchina rimane dipendente anche dal contesto in cui è inserita un'istruzione.
- Vengono utilizzati pochi registri ad uso generale e nonostante il tentativo del compilatore di produrre istruzioni più corte, sono comunque necessari più cicli d'esecuzione per istruzione e i codici operativi risultano più lunghi.
- La taglia dei programmi risulta simile all'architettura RISC.
- Un'istruzione complessa risulta essere eseguita più velocemente di una serie di istruzioni semplici ma viene rallentata l'esecuzione delle istruzioni più semplici (che risultano essere le più frequenti).

ARCHITETTURA RISC (REDUCED INSTRUCTION SET COMPUTER)

- Ciclo esecutivo di un MACHINE CYCLE: se la pipeline è piena viene terminata un'istruzione ad ogni ciclo di clock.
- Utilizzo ottimizzato per trasferimento dati con registri ed utilizzo di pochi modi di indirizzamento per semplificare l'unità di controllo.
- Presenti pochi e semplici formati fissi per le istruzioni in modo da rendere simultanei decodifica, accesso ai registri ed ottimizzare la fase di fetch.
- La regolarità facilita le ottimizzazioni del compilatore.
- Unità di controllo cablata: meno flessibile ma più veloce. (hardwire) cablata=

→ **CONFRONTO CISC-RISC**

Non è possibile fare un adeguato confronto in quanto i risultati variano in funzione di hardware e compilatori usati

È difficile valutare se il lavoro dei compilatori sia efficacemente semplificato dall'architettura CISC in quanto l'ottimizzazione di istruzioni macchina complesse dipende anche dal contesto delle stesse, risultando quindi difficile la loro organizzazione.
Inoltre un'istruzione complessa risulta essere eseguita più velocemente di una serie di istruzioni semplici, che risultano essere le più frequenti.
Tuttavia l'utilizzo di istruzioni semplici comporta l'impiego di più istruzioni e quindi programmi più lunghi
La taglia dei programmi prodotti da RISC e CISC risulta infatti simile

È complesso confrontare le due architetture perché non esiste un set completo ed efficace di programmi test e risulta difficile valutare quali risultati siano dovuti al compilatore e quali all'hardware.
Molti confronti sono stati portati a termine su macchine semplificate e prototipali.
Molti processori commerciali moderni implementano architetture con elementi provenienti da entrambi RISC e CISC.

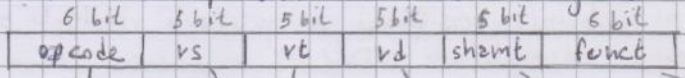
MIPS Instruction set

ARCHITETTURA MIPS (MICROPROCESSOR without INTERLOCKED PIPELINE STAGES)

Instruction set RISC per un'implementazione efficiente della pipeline.

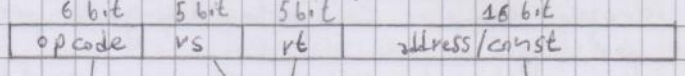
- Caratteristiche: (MIPS a 32 bit)
  - ISTRUZIONI:
    - Tutte le istruzioni a dimensione 32 bit, 3 formati
    - operazioni sui dati eseguiti solo sfruttando i registri
    - LOAD e STORE per trasferire memoria-registro ma nessuna operazione memoria-memoria
    - tutte le operazioni operano sui registri
  - REGISTRI:
    - 32 registri a 32 bit
    - registri indicati con $1, $2, ... $0 contiene sempre 0
  - DATI:
    - registri caricabili con byte, parole, mezze parole
    - spazio registri riempito con padding di 0 o viene replicato il segno
  - INDIRIZZAMENTO:
    - immediato
    - spiazzamento
    - Derivati: registro indiretto (offset a 0 0000($3)), assoluto (registro 0 come base 0004($0))

- Formati istruzioni:
  3 diversi formati a 32 bit, opcode e campi registri a dimensione fissa per semplificare Instruction Decode e Fetch di Operandi.
  - FORMATO R: operazioni aritmetiche e logiche



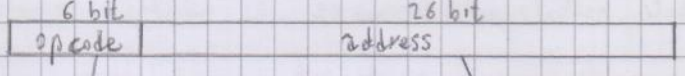
codice istruzione — registri primo e secondo argomento — registro destinazione — shift amount — codice funzione variante operativa

  - FORMATO I: load/store, immediate e salto condizionato



cod. istruzione — registri argomenti — spiazzamento o costante

  - FORMATO J: istruzioni salto incondizionato



cod. istruzione — (pseudo) indirizzo target

- Ciclo esecutivo:

1. INSTRUCTION FETCH (IF): IR ← Mem[PC], NPC ← PC+4
2. INSTRUCTION DECODE/REGISTER FETCH (ID) campi immediati da estendere a 32 bit
3. EXECUTE/ADDRESS CALCULATION (EX) ALU usata da R, I ma J da shiftare a sx di 2 → <<2
4. MEMORY ACCESS/BRANCH COMPLETION (MEM) PC ← NPC PC ← Target if(cond) PC ← Target
5. WRITE BACK (WB)

1 ciclo di clock per fase, a regime termina un'istruzione a ciclo

- PIPELINE REGISTERS (LATCHES): dati e segnali di controllo utili alle fasi successive sono memorizzati nel registro successivo, tra le fasi.
  In ogni istante i registri contengono dati relativi ad istruzioni diverse.
  Registri: IF/ID, ID/EX, EX/MEM, MEM/WB

- BANCHI REGISTRI

IF/ID
- IF/ID.IR ← Mem[PC]
- IF/ID.NPC, PC ← if ((EX/MEM.IR == Branch) && (EX/MEM.Cond)) {EX/MEM.Target} else {PC+4}

ID/EX
- ID/EX.IR ← IF/ID.IR
- ID/EX.A ← Regs[IF/ID.IR[rs]]
  ID/EX.B ← Regs[IF/ID.IR[rt]]
- ID/EX.NPC ← IF/ID.NPC ✓estesa a 32 bit
- ID/EX.Imm ← Regs[IF/ID.IR[campo Imm]]

EX/MEM
- EX/MEM.IR ← ID/EX.IR
- EX/MEM.ALUOutput ← ID/EX.A op ID/EX.B (o ID/EX.Imm) } Aritm.-Log.
- EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm } Load/Store
- EX/MEM.B ← ID/EX.B
- EX/MEM.Target ← ID/EX.NPC + (ID/EX.Imm <<2) } Salti
- EX/MEM ← Zero (ID/EX.A - ID/EX.B)

MEM/WB
- MEM/WB.IR ← EX/MEM.IR LMD: Load Memory Data
- MEM/WB.ALUOutput ← EX/MEM.ALUOutput
- MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] } Load
- Mem[EX/MEM.ALUOutput] ← EX/MEM.B } Store

FASE WB
- Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput } Aritm-Log
- Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD } Load

Le fasi IF e ID non dipendono da segnali di controllo perché questi vengono calcolati in ID e poi propagati tramite i registri

Le dipendenze dei dati vengono individuate in fase ID prima di essere rilasciate a EX (issued) e viene valutato se impostare uno stallo o un forward.

La scrittura avviene nella prima metà del clock, la lettura nella seconda; in questo modo non c'è conflitto tra una fase che tenta di scrivere in un registro dal quale un'altra fase deve leggere nello stesso ciclo.

Dipendenze dei dati rilevate da HAZARD DETECTION UNIT in fase ID comprendo il registro source di IF/ID con il registro target MEM/WB

Formato I → rt non può essere modificato