13. Architetture RISC e Processori RISC-V

Contesto storico e innovazioni nei computer

L'evoluzione dei computer ha visto diverse innovazioni fondamentali: il concetto di famiglia di processori (IBM System/360, 1964), l'unità di controllo microprogrammata (idea di Wilkes 1951, implementata nel System/360), la memoria cache (IBM System/360 model 85, 1969), la RAM a semiconduttori (1970), i microprocessori (Intel 4004, 1971), la pipeline e i processori multipli.

Architetture RISC (Reduced Instruction Set Computer)

Riconoscimento e origini

John L. Hennessy e David A. Patterson hanno ricevuto il Turing Award 2017 per aver sviluppato un approccio sistematico e quantitativo alla progettazione delle architetture dei computer. Il loro lavoro ha portato a principi duraturi e ripetibili utilizzati da generazioni di architetti. Oggi il 99% degli oltre 16 miliardi di microprocessori prodotti annualmente sono processori RISC, presenti in smartphone, tablet e dispositivi IoT.

Il problema del gap semantico

I linguaggi di alto livello (HLL) permettono di esprimere algoritmi in modo conciso, delegando al compilatore la gestione dei dettagli implementativi. Supportano paradigmi come imperativo, funzionale, logico e object-oriented. Tra istruzioni HLL e istruzioni macchina esiste un gap semantico che causa esecuzione inefficiente, dimensione eccessiva dei programmi compilati e complessità del compilatore.

Risposta tradizionale (CISC)

I progettisti hardware hanno inizialmente risposto ampliando il set di istruzioni, introducendo svariati modi di indirizzamento e implementando in hardware costrutti di linguaggi ad alto livello (esempio: istruzione CASE su architettura VAX). Questo semplificava il compilatore e migliorava l'efficienza tramite microcodice.

Approccio alternativo (RISC)

L'alternativa consiste nell'individuare caratteristiche e pattern di esecuzione delle istruzioni macchina generate dai programmi HLL per semplificare l'architettura sottostante anziché complicarla. L'analisi si concentra su operazioni eseguite, operandi utilizzati e flusso dell'esecuzione, utilizzando misure dinamiche raccolte durante l'esecuzione dei programmi.

Analisi delle istruzioni HLL

Le misurazioni mostrano che in HLL predominano le istruzioni di assegnamento (45% in Pascal, 38% in C) e le istruzioni condizionali IF (29% in Pascal, 43% in C). Tuttavia, ponderando per le istruzioni macchina generate, i loop consumano il 42% in Pascal e 32% in C, mentre le chiamate di procedura (CALL) rappresentano il 31-33%. Ponderando per gli accessi a memoria, le CALL raggiungono il 44-45%.

Analisi degli operandi

Gli operandi sono principalmente variabili scalari locali (55% in media), seguite da array/strutture (25%) e costanti intere (20%). L'ottimizzazione deve quindi concentrarsi sull'accesso alle variabili locali scalari.

Chiamate di procedura

Le chiamate di procedura sono le istruzioni più costose in termini di tempo. Le misurazioni indicano che coinvolgono tipicamente meno di 6 parametri, meno di 6 variabili locali e presentano poco annidamento.

Implicazioni progettuali

La strategia ottimale non è rendere le istruzioni macchina più simili a quelle HLL, ma ottimizzare le performance dei pattern più frequenti e costosi:

1. **Ampio numero di registri** o uso ottimizzato dal compilatore per ridurre gli accessi alla memoria

2. **Progettazione accurata della pipeline** per gestire dipendenze dal controllo dovute a salti e chiamate
3. **Set di istruzioni semplificato** implementato in maniera efficiente

---

Uso dei registri nelle architetture RISC

Principi fondamentali

I registri sono memoria interna alla CPU ad accesso molto rapido con indirizzi più brevi rispetto a cache e memoria principale. L'obiettivo è mantenere gli operandi nei registri il più possibile, minimizzando i trasferimenti memoria-registro.

La soluzione hardware consiste nell'aumentare il numero di registri per mantenere più variabili più a lungo. La soluzione software prevede che il compilatore massimizzi l'uso dei registri allocando le variabili più usate nei registri attraverso sofisticate tecniche di analisi.

Finestre di registri

Le variabili scalari locali cambiano ad ogni chiamata/rientro da procedura (scope). L'idea è suddividere i registri in piccoli gruppi di taglia fissa, dove ogni procedura ha la propria finestra. In ogni momento è visibile solo il gruppo corrispondente alla procedura attiva.

Una chiamata di procedura cambia automaticamente il gruppo di registri invece di provocare salvataggio in memoria. Al ritorno viene riselezionato il gruppo precedente. Le finestre relative a procedure adiacenti sono parzialmente sovrapposte per facilitare il passaggio dei parametri.

Struttura delle finestre

Ogni finestra è divisa in tre sottogruppi:

- **Parameter registers**: contengono i parametri passati all'invocazione e il valore da restituire
- **Local registers**: memorizzano le variabili locali
- **Temporary registers**: usati per scambiare parametri con eventuali procedure chiamate

I temporary registers di una procedura coincidono fisicamente con i parameter registers della procedura chiamata, permettendo il passaggio parametri senza trasferire dati.

Buffer circolare

I registri sono organizzati a buffer circolare. Il Current Window Pointer (CWP) punta alla finestra della procedura attiva e i riferimenti ai registri sono offset a partire dal CWP. Il Saved Window Pointer (SWP) indica dove ripristinare l'ultima finestra salvata in memoria.

Quando CWP raggiunge SWP (modulo N), si genera un interrupt, la finestra più vecchia viene salvata in memoria principale, SWP viene incrementato e la chiamata può procedere. Un buffer a N finestre può contenere N-1 attivazioni di procedure.

Per le variabili globali, accessibili da qualunque procedura, si usa un gruppo di registri dedicato disponibile a tutte le procedure.

Ottimizzazione software dei registri

Il compilatore associa ogni variabile a un registro simbolico (potenzialmente infiniti) e li mappa su registri reali (finiti). Se due registri simbolici si usano in momenti diversi, possono essere mappati sullo stesso registro reale. Se i registri reali non sono sufficienti, alcune variabili vengono mantenute in memoria principale.

Il problema equivale alla colorazione di un grafo: i nodi rappresentano i registri simbolici, due nodi sono collegati se le variabili sono "in vita" nello stesso intervallo di tempo, i colori rappresentano i registri reali. Decidere se un grafo è colorabile con k colori è NP-completo nel caso generale, ma esistono algoritmi efficienti per casi specifici. 32-64 registri fisici si dimostrano sufficienti.

CISC vs RISC

Argomenti a favore di CISC

CISC propone un ampio insieme di istruzioni complesse per semplificare il compilatore e migliorare le performance, generando sequenze brevi e veloci.

Criticità dell'approccio CISC

Semplificazione del compilatore: le istruzioni complesse sono difficili da sfruttare perché il compilatore deve trovare match precisi considerando il contesto. Con un set complesso è più difficile ottimizzare il codice macchina per migliorare la pipeline. Le misurazioni mostrano che le istruzioni più frequenti sono le più semplici.

Sequenze più brevi: meno istruzioni non significa meno bit occupati poiché molte istruzioni implicano codici operativi più lunghi e i riferimenti a registri richiedono meno bit. La dimensione dei programmi compilati per RISC o CISC risulta simile.

Sequenze più veloci: un'istruzione complessa può essere eseguita più velocemente di una serie semplice, ma l'unità di controllo diventa più complessa, il controllo microprogrammato necessita di più spazio, rallentando l'esecuzione delle istruzioni semplici che restano le più frequenti.

Caratteristiche delle architetture RISC

- **Un'istruzione per ciclo di clock**: il ciclo esecutivo dura un solo machine cycle, quindi a regime ogni ciclo termina un'istruzione
- **Operazioni da registro a registro**: tranne LOAD e STORE, aumentando o ottimizzando i registri la maggior parte degli operandi rimane nei registri
- **Pochi e semplici modi di indirizzamento**: indirizzo di registro e spiazzamento relativo a PC
- **Pochi e semplici formati fissi**: campi e opcode a dimensione fissa permettono decodifica e accesso ai registri simultanei; istruzioni allineate con la lunghezza delle parole ottimizzano il fetch
- **Unità di controllo cablata**: meno flessibile ma più veloce rispetto a quella microprogrammata

Confronto tra processori

Caratteristica	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000
Anno	1973	1978	1989	1987	1991
Numero istruzioni	208	303	235	69	94
Dimensione istruzione (byte)	2-6	2-57	1-11	4	4
Modi di indirizzamento	4	22	11	1	1
Registri general-purpose	16	16	8	40-520	32

Valutazione comparativa

Non è evidente quale architettura sia nettamente migliore. I problemi nel confronto includono: assenza di architetture direttamente confrontabili, mancanza di set completi di programmi di test, difficoltà nel separare effetti hardware da quelli del compilatore, confronti su macchine prototipali. Molte CPU commerciali utilizzano idee da entrambe le filosofie: PowerPC è RISC con elementi CISC, Pentium II è CISC con elementi RISC.

Il futuro

Le architetture domain-specific sono l'unica opzione per migliorare le prestazioni man mano che i vantaggi della legge di Moore svaniscono. Come afferma Patterson, i circuiti non diventano più veloci, quindi occorre cambiare l'architettura del set di istruzioni. Esempi includono GPU e TPU per machine learning.

Famiglia RISC-V

Introduzione

RISC-V è stato sviluppato alla UC Berkeley dal 2010, ispirato per molti aspetti a MIPS, un'architettura sperimentale sviluppata a Stanford negli anni '80. È un'architettura molto regolare, progettata per un'implementazione efficiente della pipeline.

Caratteristiche generali (32 bit)

Istruzioni: tutte di dimensione 32 bit, tutte le operazioni sui dati sono da registro a registro. Le operazioni sulla memoria sono solo load e store per trasferire dati tra memoria e registri, senza operazioni memoria-memoria.

Registri: 32 registri di 32 bit, indicati con x0-x31. Il registro x0 contiene sempre 0.

Dati: i registri possono essere caricati con byte, mezze parole e parole. Un dato più corto viene allungato riempiendo i bit rimanenti con 0 o estendendo il segno.

Modi di indirizzamento: immediato (es. `addi x2, x2, 004`) e displacement (es. `sw x1, 00c(x2)`). Altre modalità sono derivabili: indiretta registro con displacement a 0, assoluta usando x0 come registro base.

Formati delle istruzioni

RISC-V utilizza 6 formati diversi (4 principali) con dimensione fissa di opcode e riferimenti a registri. I campi hanno semantica specifica e si trovano sempre nella stessa posizione, semplificando Instruction Decode e Fetch degli operandi.

Formato R (istruzioni aritmetiche e logiche): 6 campi fissi - funz7 (7 bit), rs2 (5 bit), rs1 (5 bit), funz3 (3 bit), rd (5 bit), codop (7 bit). Esempio: `add x1, x2, x3` esegue x1 ← [x2] + [x3].

Formato I (load, operazioni immediate, shift): immediato (12 bit), rs1 (5 bit), funz3 (3 bit), rd (5 bit), codop (7 bit). Usato per `lw x8, 1200(x9)` che esegue x8 ← Mem[1200 + [x9]], e per `addi x1, x8, 4` che esegue x1 ← [x8] + 4. Include anche JALR (jump and link register) per salti incondizionati.

Formato S (store): immediato diviso in due parti (7+5 bit), rs2 (5 bit), rs1 (5 bit), funz3 (3 bit), codop (7 bit). Esempio: `sw x1, 1000(x2)` esegue [x1] → Memoria[[x2]+1000].

Formato SB (salti condizionali, sottoformato di S): l'immediato a 12 bit codifica offset con segno in multipli di 2, aggiunto al PC corrente (intervallo ±4096 rispetto al PC). BEQ e BNE confrontano uguaglianza/disuguaglianza, BLT/BLTU e BGE/BGEU confrontano minore/maggiore con o senza segno.

Formato U (caricamento costanti): immediato (20 bit), rd (5 bit), codop (7 bit). LUI inserisce il valore nei primi 20 bit del registro destinazione riempiendo i 12 bit bassi con zeri. AUIPC forma un offset di 32 bit e lo aggiunge al PC.

Formato UJ (salti incondizionati, sottoformato di U): JAL codifica offset con segno in multipli di 2 byte, aggiunto al PC (intervallo ±1 MB). Memorizza PC+4 nel registro rd. La convenzione usa x1 come registro di ritorno e x5 come alternativo.

Ciclo esecutivo

Il ciclo si articola in 5 fasi:

1. **IF (Instruction Fetch)**: IR ← Mem[PC]; PC, NPC ← PC+4
2. **ID (Instruction Decode / register fetch)**:
   - Formato R: A ← Regs[rs1]; B ← Regs[rs2]
   - Formato I: A ← Regs[rs1]; Imm ← campo immediato esteso a 32 bit
   - Formato U: Imm ← campo immediato esteso
3. **EX (Execute / address calculation)**: tutte le istruzioni usano la ALU
   - Riferimento a memoria: ALUOutput ← A + Imm
   - ALU registro-registro: ALUOutput ← A funct B
   - ALU registro-immediato: ALUOutput ← A op Imm
   - Salto condizionato: Cond ← (A==B); Target ← NPC + (Imm << 2)
   - Salto incondizionato: Target ← PC + (Imm << 2)
4. **MEM (Memory access / branch completion)**:
   - Load: LMD ← Mem[ALUOutput]
   - Store: Mem[ALUOutput] ← B
   - Salto incondizionato: PC ← Target
   - Salto condizionato: if (Cond) PC ← Target
5. **WB (Write Back)**:
   - Load: Regs[rd] ← LMD
   - ALU: Regs[rd] ← ALUOutput

Datapath

Il datapath comprende diverse unità funzionali: memoria condivisa, singola ALU condivisa tra le istruzioni, banco di registri. L'uso di unità funzionali condivise richiede multiplexer e registri temporanei (IR, MDR, A, B, ALUOut) che trattengono i dati tra i cicli di clock della stessa istruzione.

Due cache separate (istruzioni/dati) evitano conflitti tra IF e MEM. I registri sono letti in ID e scritti in WB: la scrittura avviene nella prima metà del ciclo di clock, la lettura nella seconda metà.

Segnali di controllo

I segnali principali sono:

- **RegWrite**: abilita scrittura nel register file
- **ALUSrc**: seleziona secondo operando ALU tra registro B o immediato
- **PCSrc**: seleziona tra PC+4 o indirizzo di salto in base al segnale Zero
- **MemRead/MemWrite**: abilitano lettura/scrittura memoria dati
- **MemtoReg**: seleziona se il dato da scrivere nel registro proviene da ALU o memoria

---

Pipeline RISC-V

Principi fondamentali

La pipeline prevede 1 ciclo di clock per fase (IF, ID, EX, MEM, WB). Un'istruzione richiede 5 cicli, ma a regime ogni ciclo completa 5 fasi di 5 istruzioni diverse e termina un'istruzione.

I pipeline registers (pipeline latches) memorizzano i dati utili alle fasi successive. In ogni istante registri diversi contengono dati relativi a istruzioni diverse. I registri memorizzano sia dati che segnali di controllo.

Registri di pipeline

IF/ID: memorizza IR e NPC. Il PC viene aggiornato considerando eventuali branch precedenti.

ID/EX: memorizza IR, A, B, NPC, Imm (esteso), oltre ai segnali di controllo per le fasi successive.

EX/MEM: memorizza IR, ALUOutput, Target, Cond, B (per store).

MEM/WB: memorizza IR, ALUOutput o LMD a seconda dell'istruzione.

Unità di controllo

Le fasi IF e ID non dipendono dai segnali di controllo. In fase ID si calcolano i segnali corretti per le fasi successive, che vengono propagati attraverso i registri di pipeline: due segnali per EX, cinque per EX/MEM (tre usati in MEM), due per MEM/WB (usati in WB).

Dipendenze dai dati e forwarding

Rilevamento delle dipendenze

Nella pipeline RISC-V tutte le dipendenze dai dati possono essere individuate nella fase ID. Se si rileva una dipendenza, l'istruzione va in stallo prima di essere rilasciata (issued). In fase ID è anche possibile determinare il tipo di data forwarding per evitare lo stallo.

Dipendenze RAW da istruzione Load

Consideriamo la sequenza:

```
LW x1, 45(x2)
ADD x5, x1, x7
```

```
SUB x8, x6, x7
OR x9, x6, x7
```

Nessuna dipendenza: se le istruzioni successive non usano x1, non occorre fare nulla.

Dipendenza con accessi in ordine: se OR usa x1, la lettura avviene dopo la scrittura del dato caricato (WB della load coincide temporalmente con ID della quarta istruzione, con scrittura nella prima metà e lettura nella seconda).

Dipendenza risolvibile con forwarding: se SUB usa x1, opportuni comparatori rilevano l'uso e inoltrano il risultato della load alla ALU in tempo per la fase EX di SUB. Il dato viene propagato da MEM/WB.LMD all'input della ALU.

Dipendenza che richiede stallo: se ADD (istruzione immediatamente successiva) usa x1, serve uno stallo perché il dato non è ancora disponibile. Dopo lo stallo, il forwarding completa la risoluzione.

Condizioni per riconoscere le dipendenze

Le condizioni sono verificate dalla Hazard Detection Unit:

EX hazard:

- Se EX/MEM.RegisterRd = ID/EX.RegisterRs1 → ForwardA = 10
- Se EX/MEM.RegisterRd = ID/EX.RegisterRs2 → ForwardB = 10

MEM hazard:

- Se MEM/WB.RegisterRd = ID/EX.RegisterRs1 (e non c'è EX hazard sullo stesso registro) → ForwardA = 01
- Se MEM/WB.RegisterRd = ID/EX.RegisterRs2 (e non c'è EX hazard sullo stesso registro) → ForwardB = 01

In tutti i casi si verifica anche che RegWrite sia attivo e che il registro destinazione non sia x0.

Hardware per il forwarding

L'Unità di Propagazione controlla i multiplexer sugli ingressi della ALU. I segnali PropagaA e PropagaB selezionano tra:

- 00: valore dal register file (ID/EX)
- 10: valore propagato da EX/MEM (risultato ALU del ciclo precedente)
- 01: valore propagato da MEM/WB (dato dalla memoria o risultato ALU precedente)

Gestione degli stalli

L'Unità di Rilevamento degli Hazard imposta Stall a 1 quando è necessario uno stallo. Questo azzera tutti i segnali di controllo (inserendo una bubble) e impedisce l'aggiornamento di PC e del registro IF/ID, causando la ripetizione delle fasi IF e ID.

Forwarding per operazioni store

Per operazioni di copia memoria (`lw x6, 16(x3)` seguito da `sw x6, 416(x3)`), il valore letto è disponibile dopo la fase MEM della load, mentre la store lo richiede al più tardi nella fase EX. Si utilizzano gli stessi circuiti di forward dopo aver inserito uno stallo. Senza forwarding servirebbero 2 stalli.

Register File vs Cache

Aspetto	**Ampio banco di registri a finestre**	**Cache**
Contenuto	Tutti gli scalari locali	Scalari locali usati di recente
Granularità	Variabili individuali	Blocchi di memoria
Variabili globali	Assegnate dal compilatore a registri dedicati	Usate di recente
Save/Restore	Basato sulla profondità di annidamento	Basato sull'algoritmo di sostituzione
Indirizzamento	A registro	A memoria

Per riferirsi a scalari locali i registri sono più veloci: con finestre basta combinare numero di finestra e numero di registro relativo, mentre con cache serve un intero indirizzo di memoria con confronto dei tag.