Principali Innovazioni nei Computer (1)

- Il concetto di famiglia
  - IBM System/360, anno 1964
  - DEC PDP-8
  - Separa l'architettura dall'implementazion
- 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

Architetture RISC

Reduced Instrucion Set Computer

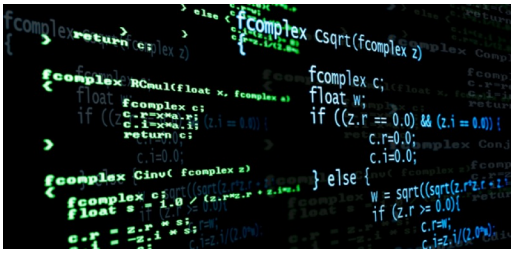
John L. **Hennessy** and David A. **Patterson**
received the 2017 ACM A.M. **Turing Award**

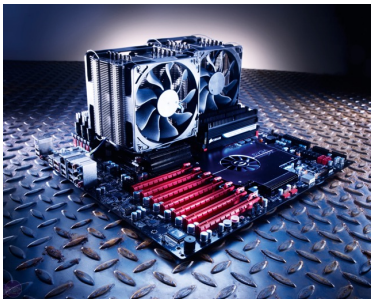
for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

Hennessy and Patterson created a systematic and quantitative approach to designing faster, lower power, and reduced instruction set computer (RISC) microprocessors. Their approach led to *lasting* and *repeatable principles* that generations of architects have used for many projects in academia and industry.
Today, 99% of the more than 16 billion microprocessors produced annually are RISC processors, and are found in nearly all smartphones, tablets, and the billions of embedded devices that comprise the Internet of Things (IoT).

Evoluzione

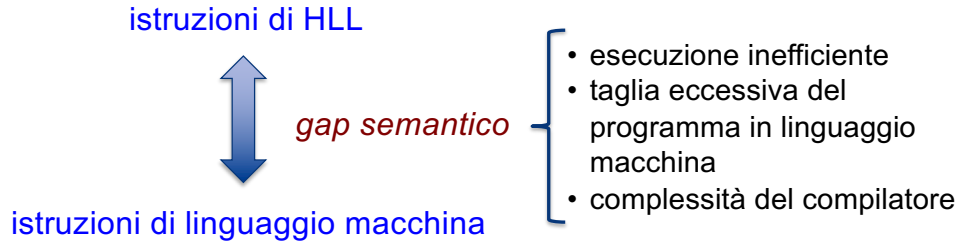
- Co-evoluzione tra hardware e linguaggi di programmazione





Evoluzione

- **High-level languages**
  - permettono di esprimere l'algoritmo risolutivo in modo più conciso: ***cosa***
  - lasciano al compilatore il compito di gestire i dettagli: ***come***
  - supportano costrutti di programmazione strutturata: paradigmi *imperativo, funzionale, logico, object-oriented*



- Risposta dei progettisti hardware:
  - set di istruzioni più ampio
  - svariati modi di indirizzamento
  - implementazione hardware di costrutti di linguaggi ad alto livello (es. CASE (switch) su architettura VAX)

✓ si semplifica il lavoro del compilatore
✓ migliora l'efficienza dell'esecuzione (sequenze di operazioni complesse implementate tramite microcodice)
✓ si supportano linguaggi HL più complessi

- esecuzione inefficiente
- taglia eccessiva del programma in linguaggio macchina
- complessità del compilatore

Alternativa:

- individuare le caratteristiche e i **pattern di esecuzione** delle istruzioni macchina generate dai programmi in HLL
- per **semplificare** l'architettura sottostante ad HLL, non complicarla

Semplificare, ***cosa?***

- operazioni eseguite
  - semplificare le *funzionalità del processore* e la sua *interazione con la memoria*
- operandi
  - tipo e frequenza d'uso degli operandi determinano *l'organizzazione della memoria* e i *modi di indirizzamento*
- flusso dell'esecuzione
  - organizzazione della *pipeline* e del controllo

come?

- fare un' **analisi** delle istruzioni macchina generate dai programmi scritti in HLL
- misure **dinamiche**: raccolte eseguendo il programma e contando il *numero di occorrenze* di una certa proprietà o di una certa caratteristica. (le misure **statiche** si basano solo sul programma sorgente, che non dice quante volte è eseguita un'istruzione)

Operazioni

- in HLL predominanza di istruzioni di **assegnamento**
  - quindi il trasferimento dei dati deve essere efficiente
- in HLL molte istruzioni **condizionali** (IF, LOOP)
  - quindi il controllo delle dipendenze dai salti deve essere efficiente
- oltre a ***frequenza*** di istruzioni, quali istruzioni richiedono più ***tempo di esecuzione***?
  - quali istruzioni del HLL causano l'esecuzione della maggior parte delle istruzioni macchina, e in quanto tempo?

Frequenza relativa di istruzioni ad alto livello

[PATT82a]

	Occorrenza Dinamica	
	Pascal	C
ASSIGN	**45%**	**38%**
LOOP	5%	3%
CALL	15%	12%
IF	**29%**	**43%**
GOTO	—	3%
OTHER	6%	1%

Frequenza relativa di istruzioni ad alto livello

[PATT82a]

	Occorrenza Dinamica		Occorrenza ponderata sulle istruzioni	
	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%
LOOP	5%	3%	**42%**	**32%**
CALL	15%	12%	**31%**	**33%**
IF	29%	43%	11%	21%
GOTO	—	3%	—	—
OTHER	6%	1%	3%	1%

moltiplicato **per il numero di istruzioni macchina prodotte** dal compilatore (normalizzato)

Frequenza relativa di istruzioni ad alto livello

[PATT82a]

	Occorrenza Dinamica		Occorrenza ponderata sulle istruzioni		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%

Frequenza relativa di istruzioni ad alto livello

[PATT82a]

dipende da

- quale linguaggio HL
- quale tipo di applicazione
- quale architettura sottostante
- resta rappresentativa delle contemporanee architetture **CISC (Complex Instruction Set Computer)**

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 scalari

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

Chiamate di procedura

- sono le istruzioni la cui esecuzione consuma più tempo, va quindi trovata un'implementazione efficiente
- due aspetti significativi:
  - il numero di parametri e variabili gestite
  - il livello di annidamento (*nesting*)
- misurazioni:
  - meno di 6 parametri, meno di 6 variabili locali
  - la maggior parte degli operandi sono variabili locali
  - poco annidamento di chiamate di procedure

Implicazioni dell'analisi

Strategia migliore per supportare i linguaggi di alto livello:

- **non** rendere le istruzioni macchina più simili alle istruzioni di HLL
- ottimizzare le performance dei pattern più usati e più time-consuming

1. ampio numero di **registri** o loro uso ottimizzato dal compilatore
   - per ottimizzare gli accessi agli operandi (abbiamo visto che sono istruzioni molto frequenti, con operandi perlopiù ***scalari e locali***, quindi è utile ridurre gli accessi alla memoria aumentando gli accessi ai registri)
2. progettazione accurata della **pipeline**
   - gestione delle dipendenze dal controllo dovute a (frequenti) salti e chiamate di procedure evitando i prefetch errati
3. set di istruzioni **semplificato (ridotto)** e implementato in maniera efficiente.

architetture RISC

Comparazione fra vari processori

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

Uso dei registri

- memoria interna a CPU ad accesso molto rapido
- hanno indirizzi più brevi di quelli per l'uso di cache e memoria principale
- bisogna assicurare che gli operandi usati siano il più possibile matenuti nei registri, minimizzando i trasferimenti memoria-registro
- **Soluzione hardware:**
  - **aumentare il numero di registri**,
  - così si mantengono più variabili per più tempo
- **Soluzione software**:
  - il **compilatore massimizza l'uso** dei registri
  - le variabili più usate per ogni intervallo di tempo sono allocate nei registri
  - richiede sofisticate tecniche di *analisi dei programmi*

Uso dei registri

- memorizzare nei registri le variabili scalari ***locali*** (le più frequenti)
- pochi registri per le variabili ***globali***

```
int main(){
  int z;
  z = foo(2,5);
  cout << "The result is" << z;
}

int foo(int x, int y){
  int s = add(x,8);
  int t = add(y,3);
  return s+t;
}

int add(int a, int b){
  int r;
  r = a+b;
  return r;
}
```

che significa locali?

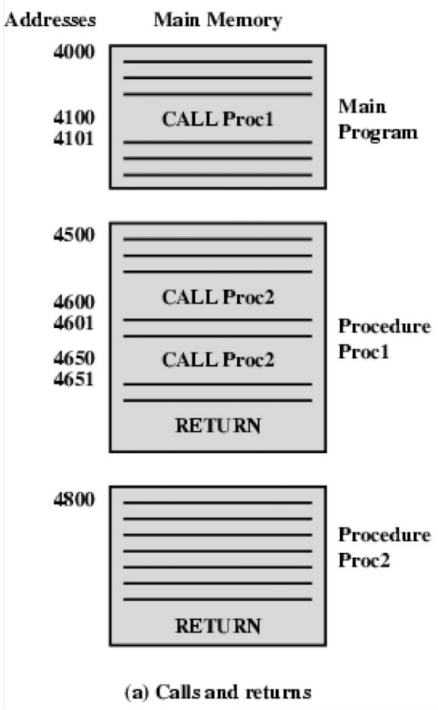
la località **cambia**
ad ogni chiamata/rientro da procedura
(scope)

Uso dei registri

```
int main(){
  int z;
  z = foo(2,5);
  cout << "The result is" << z;
}

int foo(int x, int y){
  int s = add(x,8);
  int t = add(y,3);
  return s+t;
}

int add(int a, int b){
  int r;
  r = a+b;
  return r;
}
```



(a) Calls and returns

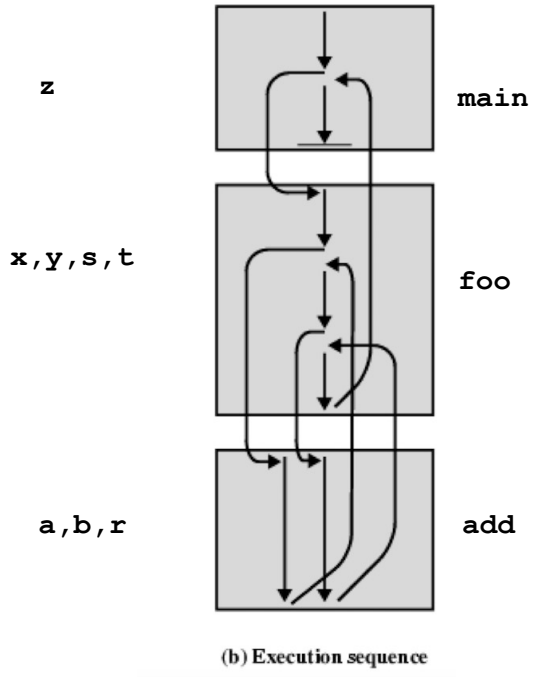
Uso dei registri

```
int main(){
  int z;
  z = foo(2,5);
  cout << "The result is" << z;
}

int foo(int x, int y){
  int s = add(x,8);
  int t = add(y,3);
  return s+t;
}

int add(int a, int b){
  int r;
  r = a+b;
  return r;
}
```

variabili locali



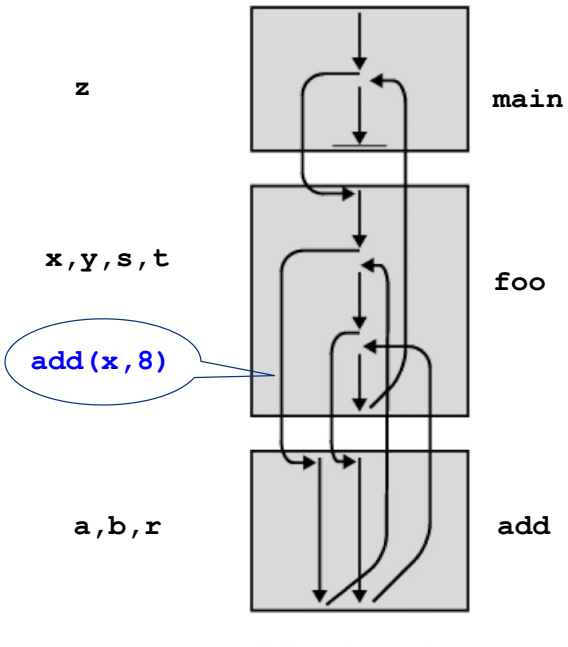
(b) Execution sequence

Uso dei registri

```
int foo(int x, int y){
  int s = add(x,8);
  int t = add(y,3);
  return s+t;
}

int add(int a, int b){
  int r;
  r = a+b;
  return r;
}
```

variabili locali



(b) Execution sequence

ogni **chiamata** di procedura:

- salva le variabili locali dai registri in memoria
- può riusare i registri per le nuove variabili locali
- passa i parametri

al **termine** della precedura:

- ripristina nei registri (i valori del) le variabili locali del chiamante
- restituisce il risultato

Uso dei registri

- misurazioni: tipicamente le chiamate di procedura
  - coivolgono pochi parametri (meno di 6)
  - non presentano grado di annidamento elevato

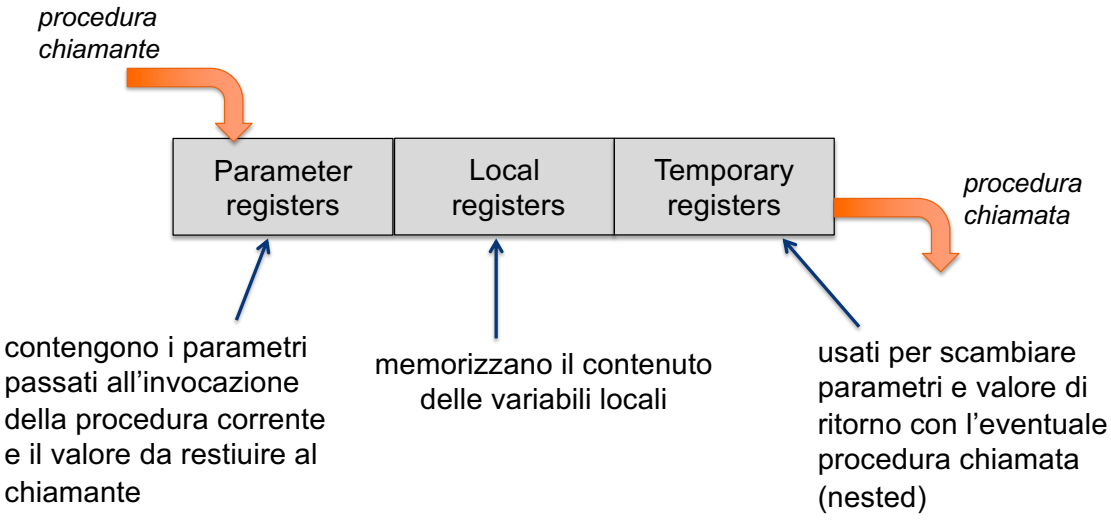
Idea per usare al meglio i *(tanti)* registri general-purpose:

- suddividere i registri in molti piccoli gruppi (di taglia fissa)
- ogni procedura ha il proprio gruppo/**finestra** di registri
- in ogni momento è visibile (indirizzabile) un solo gruppo/finestra: quello corrispondente alla procedura attiva in quel momento

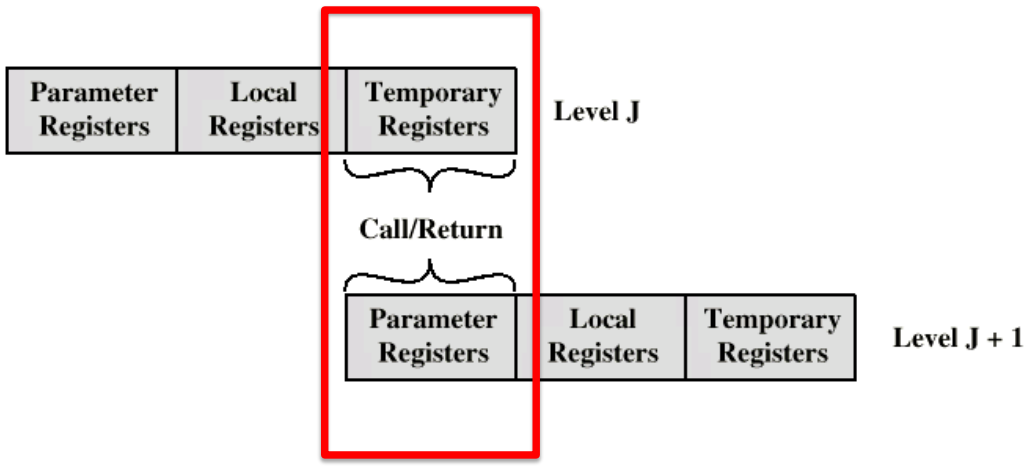
- una chiamata di procedura
  - cambia automaticamente il gruppo di registri da usare
  - invece di provocare il salvataggio dei dati in memoria
  - al ritorno viene riselezionato il gruppo di registri assegnato in precedenza alla procedura chiamante
  - le finestre relative a procedure adiacenti sono parzialmente sovrapposte, in modo da facilitare il passaggio dei parametri

Finestre di registri

- Ogni gruppo/finestra di registri è diviso in tre sottogruppi



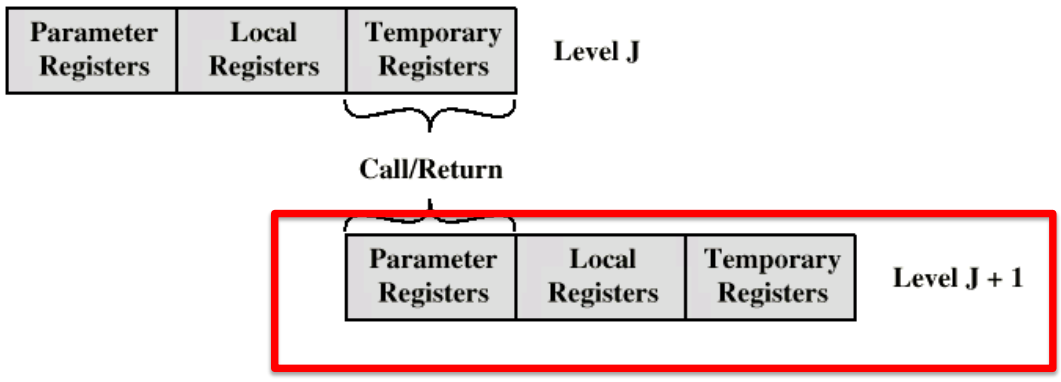
Finestre di registri



sono fisicamente gli stessi registri

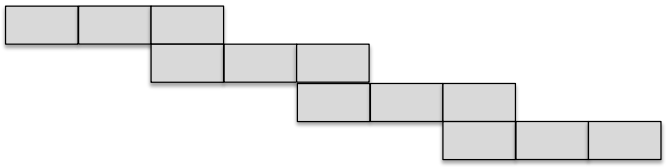
si possono passare i parametri **senza trasferire dati**

Finestre di registri

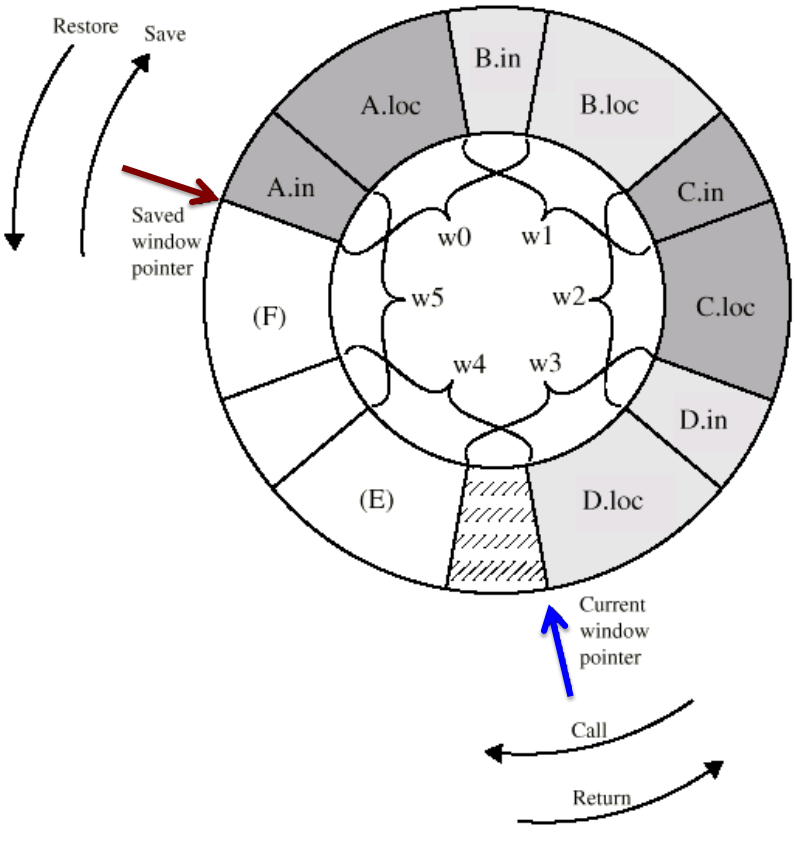


in ogni momento è visibile, e **indirizzabile**, **un solo gruppo/finestra**, usando indirizzi da 0 a N-1

Finestre di registri



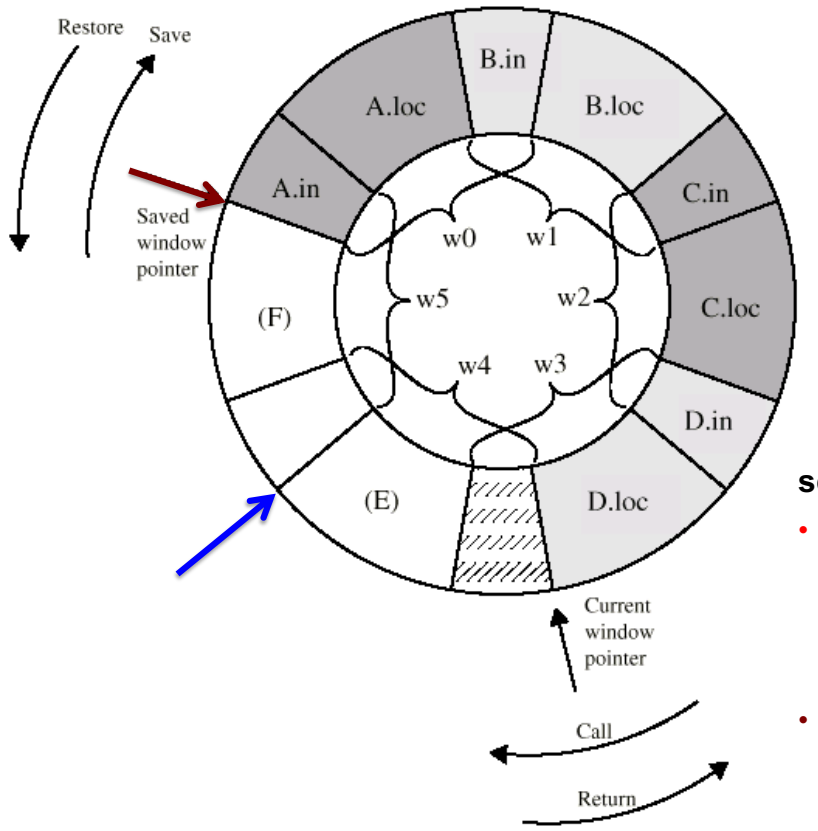
- **quante** finestre di registri?
  - una per chiamata di procedura attivata (nesting)
  - c'è spazio per un numero limitato: solo le più recenti
  - le attivazioni precedenti vanno salvate in memoria e poi recuperate quando diminuisce il nesting
- registri organizzati a **buffer circolare**



4 procedure annidate:
- A -> B -> C -> D
- D procedura attiva

Current Window Pointer
- punta alla finestra della procedura correntemente attiva
- i riferimenti ai registri usati nelle istruzioni macchina sono **offset a partire dal CWP**

Saved Window Pointer
- indica dove si deve ripristinare l'ultima finestra salvata in memoria

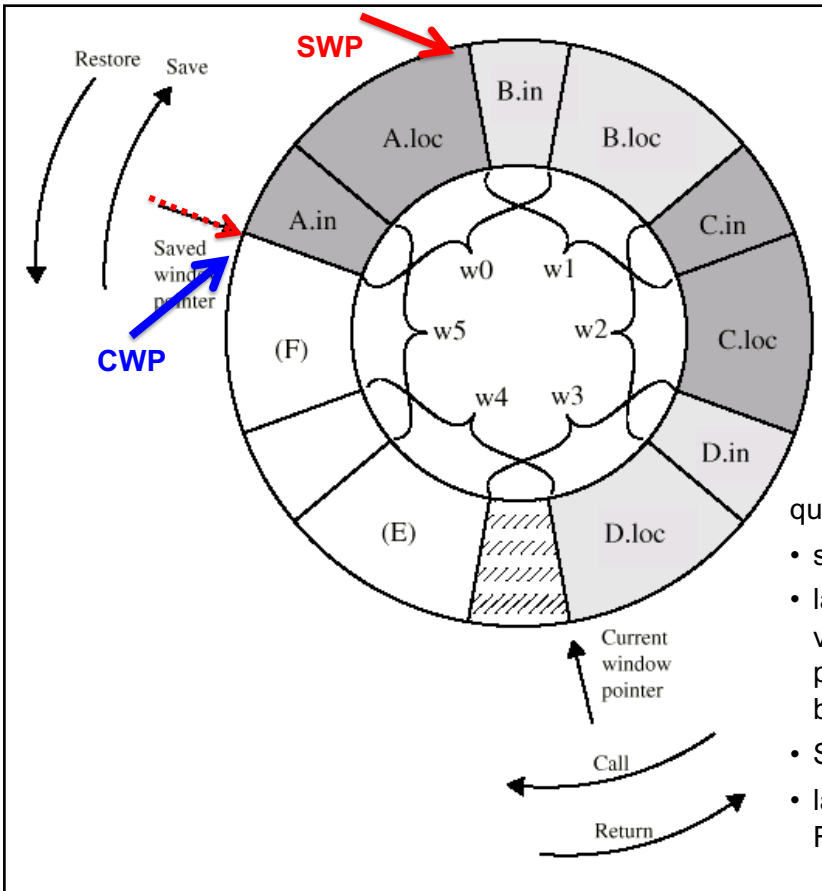


se D chiama E

- i parametri per E sono messi nei temporary registrers di D (= parameter reg. di E)
- il CWP avanza di una finestra

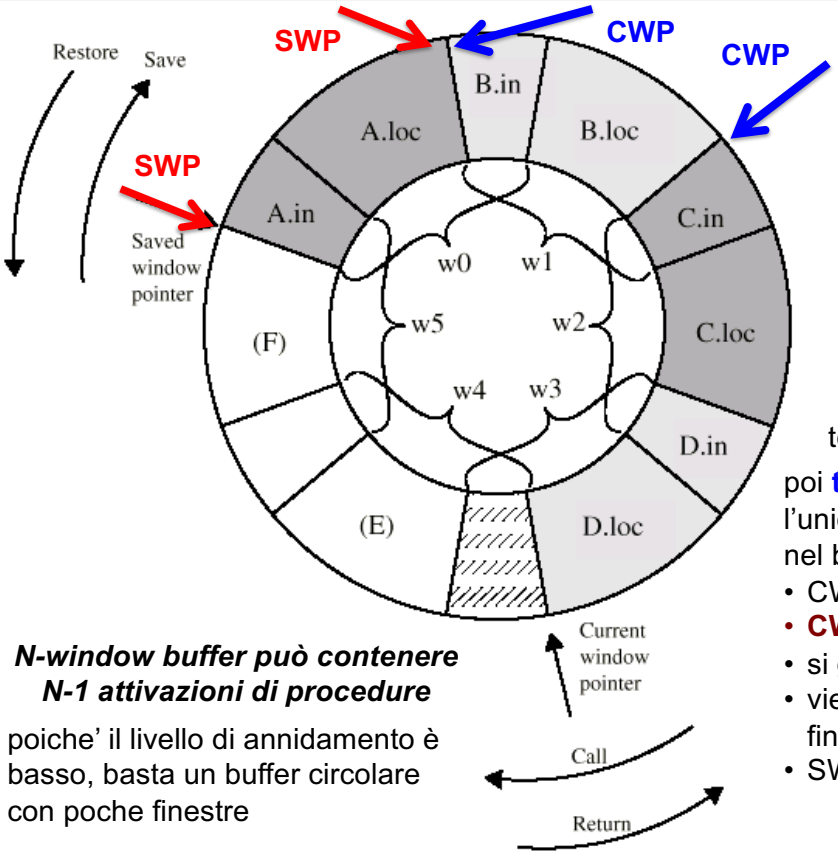
se E chiama F

- **non è possibile**: la finestra di F si sovrappone a quella di A rischia di sovrascrivere i parametri di A
- CWP = SWP (mod 6)



quando **CWP = SWP** (mod 6)

- si genera un interrupt
- la finestra più vecchia (A) viene salvata in memoria principale (bastano i primi 2 blocchi)
- SWP viene incrementato
- la chiamata della procedura F può procedere



N-window buffer può contenere N-1 attivazioni di procedure

poiche' il livello di annidamento è basso, basta un buffer circolare con poche finestre

terminano F, E, D e C

poi **termina anche B**, che è l'unica procedura rimasta nel buffer

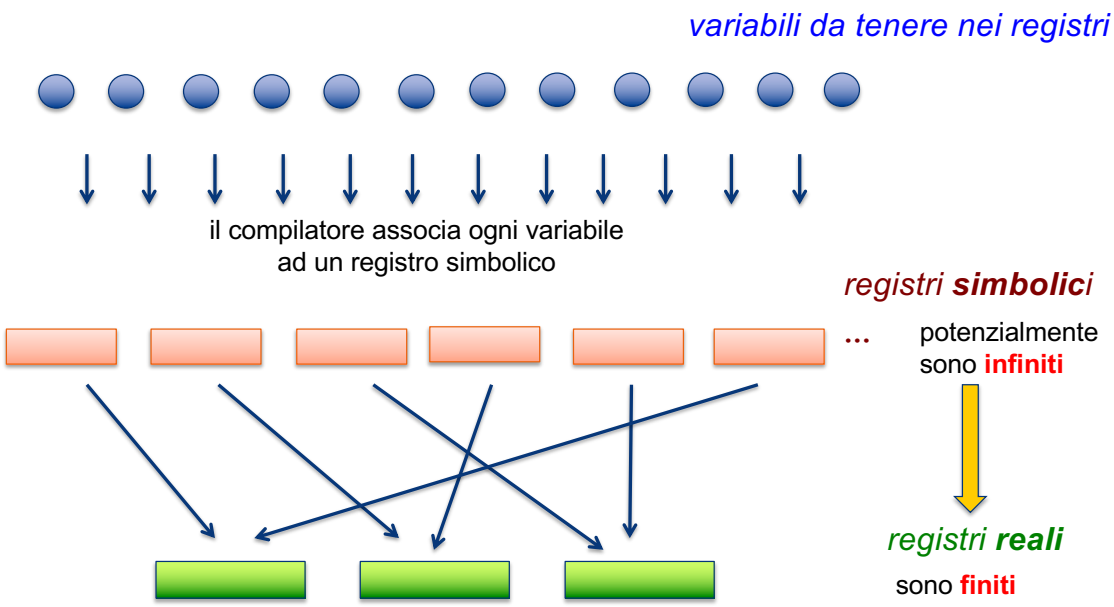
- CWP viene decrementato
- **CWP = SWP**
- si genera un interrupt
- viene ripristinata la finestra di A
- SWP viene decrementato

variabili globali

- variabili accessibili da qualunque procedura, e più di una
- dove memorizzarle?
  - il compilatore le alloca **in memoria**, ma è poco efficiente se sono usate spesso
  - Soluzione: usare un **gruppo di registri ad hoc**, disponibili a tutte le procedure

ottimizzazione dei registri

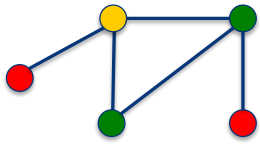
- **scopo**: trovare gli operandi il più possibile nei registri e minimizzare le operazioni di load/store
- **soluzione software**: l'architettura RISC può avere **pochi registri** (16-32) il cui uso viene **ottimizzato dal compilatore**
  - Linguaggi ad alto livello non fanno riferimento esplicito ai registri, eccezione in C: `register int`



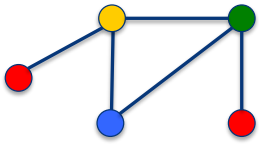
- il compilatore mappa ogni registro simbolico ad un registro reale
- se due registri simbolici **si usano in momenti diversi**, possono essere **mappati sullo stesso registro reale**
- se in un certo intervallo di tempo i **registri reali non sono in numero sufficiente** per contenere tutte le variabili riferite in quell'intervallo, alcune variabili vengono **mantenute nella memoria principale**

ottimizzazione dei registri

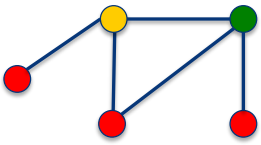
- decidere quale registro simbolico (quale variabile) assegnare a quale registro reale in ogni momento
- **m** compiti da eseguire, **n** risorse, con **m >> n**. Decidere quale compito assegnare a quale risorsa in ogni momento (*es. m voli da effettuare, n aerei*)
- equivale a risolvere un problema di **colorazione di un grafo:**
  - assegnare un colore ad ogni nodo in modo che
  - nodi adiacenti abbiano colori diversi
  - usare il minimo numero di colori



NO

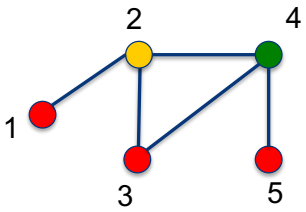


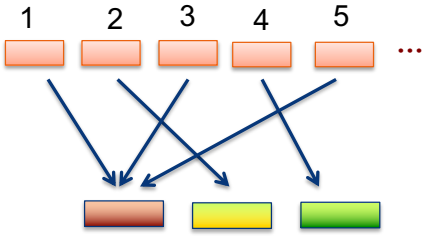
NO



SI

ottimizzazione dei registri

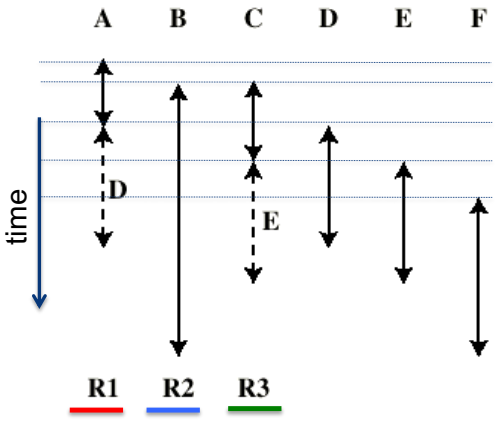




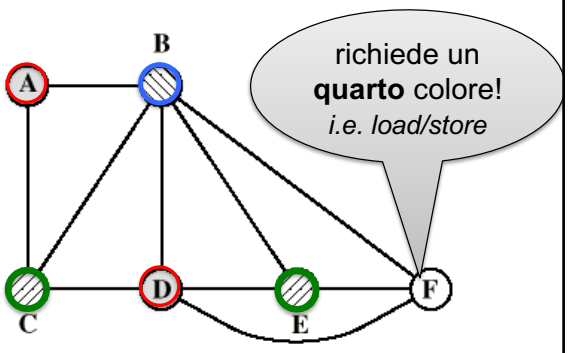
- i **nodi** (tanti) corrispondono ai **registri simbolici**
- due nodi sono collegati da un **arco** se i due registri simbolici (variabili) sono **"in vita" nello stesso intervallo** di tempo/porzione di codice
- i **colori** (pochi) corrispondono ai **registri reali**
- i nodi dello stesso colore possono essere assegnati allo stesso registro reale
- se servono più colori di quanti sono i registri reali, allora i nodi che non riescono ad essere colorati vanno memorizzati in memoria principale

grafo di interferenza

A,B,C,D,E,F registri simbolici e R1, R2, R3 registri reali



(a) Time sequence of active use of registers



(b) Register interference graph

colorazione del grafo di interferenza

- decidere se un grafo è colorabile con k colori è un problema "difficile" (NP-completo) nel caso generale.
- algorimi efficienti per casi specifici
- 32-64 registri fisici si dimostrano sufficienti

CISC o RISC?

- **CISC: ampio insieme di istruzioni, istruzioni più complesse**
  - per semplificare compilatore e migliorare *performance*
  - il compilatore deve generare "*buone" sequenze* di istruzioni macchina, cioè ***brevi*** e ***veloci*** da eseguire

CISC o RISC?

- **CISC: ampio insieme di istruzioni, istruzioni più complesse**
  - per semplificare compilatore e migliorare performance
  - il compilatore deve generare "*buone" sequenze* di istruzioni macchina, cioè ***brevi*** e ***veloci*** da eseguire
- **Sempifica il compilatore?**
  - istruzioni macchina complesse (quindi più simili a quelle HL) sono ***difficili da sfruttare***, perche' il compilatore deve trovare dei match precisi (è importante anche il **contesto i**n cui è inserita un'istruzione!)
  - con un set di istruzioni complesse è più ***difficile ottimizzare il codice macchina prodotto***, cioè **ridurlo** e **riorganizzarlo** per migliorare la pipeline

CISC o RISC?

- **CISC: ampio insieme di istruzioni, istruzioni più complesse**
  - per semplificare compilatore e r
  - il compilatore deve generare "*b* china, cioè brevi e veloci da eseguire
- **Sempifica il compil**
  - istruzioni macchina uelle HL sono ***difficili da sfrutta*** vare dei match precisi (è importante anche il **contesto** in inserita un'istruzione!)
  - con un set di istruzioni complesse è più ***difficile ottimizzare il codice macchina prodotto***, cioè **ridurlo** e **riorganizzarlo** per migliorare la pipeline
  - le misurazioni dinamiche dicono che le istruzioni più frequenti sono le più semplici

si osserva un'istruzione di HL **nel suo contesto** (porzione di programma HL)

ottimizza **una sequenza di** istruzioni macchina

CISC o RISC?

- **CISC: ampio insieme di istruzioni, istruzioni più complesse**
  - per semplificare compilatore e migliorare performance
  - il compilatore deve generare "*buone" sequenze* di istruzioni macchina, cioè ***brevi*** e ***veloci*** da eseguire
- **Sequenze di istruzioni più brevi?**
  - sequenze brevi occupano meno memoria, ma la memoria è meno costosa
  - meno istruzioni implica meno fetch e più cache hit, quindi esecuzione più veloce
  - **ma**: meno istruzioni **non significa meno bit di memoria occupata**:
    - molte istruzioni implicano codici operativi più lunghi
    - riferimenti a registri richiedono meno bit dei riferimenti alla memoria
  - la **taglia** dei programmi compilati per RISC o CISC si dimostra **simile**

CISC o RISC?

- **CISC: ampio insieme di istruzioni, istruzioni più complesse**
  - per semplificare compilatore e migliorare performance
  - il compilatore deve generare "*buone" sequenze* di istruzioni macchina, cioè ***brevi*** e ***veloci*** da eseguire
- **Sequenze di istruzioni più veloci?**
  - **un**' istruzione *complessa* può essere eseguita più velocemente di una **serie** di istruzioni più *semplici,*
  - **ma**:
    - l'unità di controllo diventa più complessa
    - il controllo microprogrammato necessita di più spazio
    - quindi **si rallenta l'esecuzione delle istruzioni più semplici**, che restano le **più frequenti**

CISC o RISC?

caratteristiche di architetture **RISC**:

- **un' istruzione per ciclo di clock**
  - (*instruction cycle*): tempo impiegato per fare fetch-decode-execute-write di un'istruzione elementare.
  - RISC: hanno un ciclo esecutivo che dura un solo machine cycle, quindi se la pipeline è piena, ad ogni ciclo di clock termina un'istruzione
  - istruzioni CISC richiedono più di un ciclo;
- **operazioni da registro a registro, tranne LOAD e STORE**
  - CISC attuali hanno anche operazioni *memory-to-memory* e *register/memory*
  - poiche' si usano di frequente scalari locali, *aumentando o ottimizzando i registri* la maggior parte degli operandi stanno a lungo nei registri.
- **pochi e semplici modi di indirizzamento**
  - indirizzo di registro, spiazzamento (relativo a PC)
  - si semplifica l'istruzione e l'unità di controllo

CISC o RISC?

caratteristiche di architetture **RISC**:

- **pochi e semplici formati fissi per le istruzioni**
  - campi e opcode a *dimensione fissa*, così la **decodifica dell'opcode** e **l'accesso ai registri** per gli operandi **possono essere simultanei**
  - istruzioni a lunghezza fissa sono **allineate con la lunghezza delle parole**, quindi il **fetch è ottimizzato** per prelevare (multipli di) una parola
  - la regolarità facilita le ottimizzazioni del compilatore
  - più responsivo agli interrupt, controllati tra due istruzioni più semplici
- **unità di controllo cablata**:
  - se cablata (cioè hardware) è meno flessibile ma più veloce
  - se microprogrammata più flessibile ma meno veloce

CISC o RISC?

- **non è evidente quale sia l'architettura nettamente migliore**
- **Problemi per fare un confronto:**
  - 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:
    - PowerPC architettura RISC con elementi CISC
    - Pentium II architettura CISC con elementi RISC

Il Futuro

- **domain-specific architectures** sono l'unica opzione per migliorare le prestazioni man mano che i vantaggi della legge di Moore iniziano a svanire:
- "the circuits aren't getting faster anymore, so you are going to have to change the instruction set architecture", ***Patterson says***.

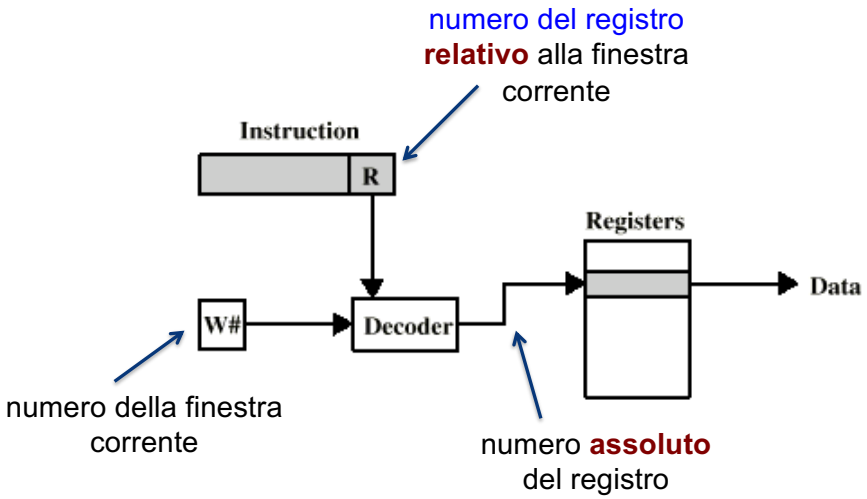
- GPU (Graphic Processing Unit)
- TPU (Tensor Processing Unit) Neural network processors for ML (con DSL TensorFlow for ML)

Register File "contro" Cache

Ampio Banco di Registri (*Register file*) a finestre	**Cache**
Tutti gli scalari locali	Scalari locali **usati di recente**
Variabili individuali	Blocchi di memoria
Variabili globali assegnate dal compilatore ai registri ad hoc	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

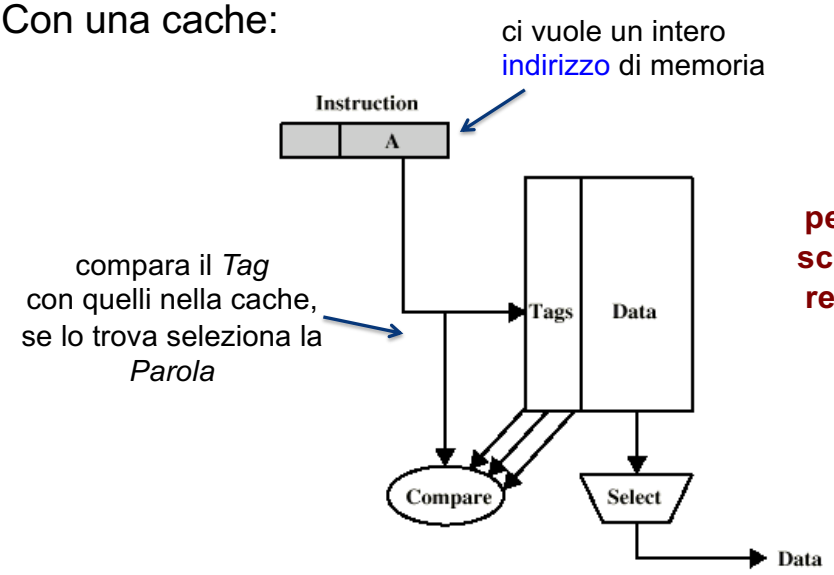
Riferimento ad uno scalare

Con un banco di registri organizzato a finestre:



Riferimento ad uno scalare

Con una cache:



per riferirsi a scalari locali i registri sono più veloci