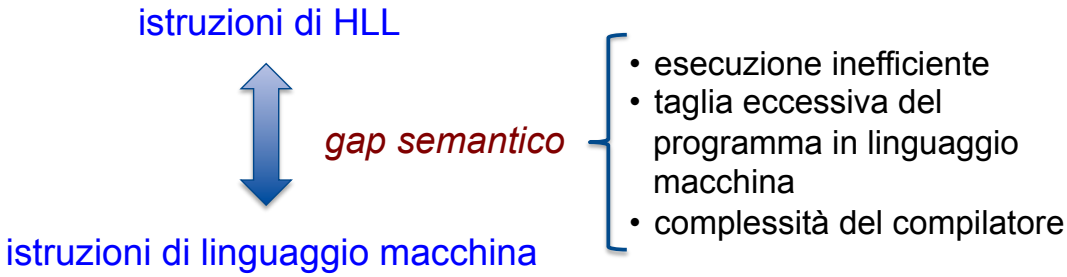
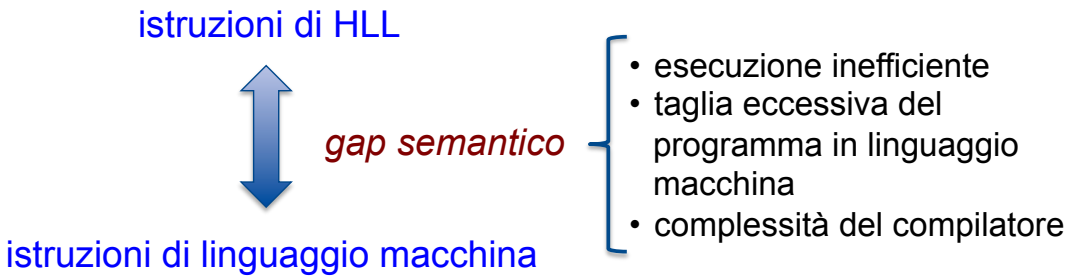


- Risposta dei progettisti hardware:
  - set di istruzioni più ampio
  - svariati modi di indirizzamento
  - implementazione hardware di costrutti di linguaggi ad alto livello (es. CASE (switch) su architettura VAX)

✓ si semplifica il lavoro del compilatore

✓ migliora l'efficienza dell'esecuzione (sequenze di operazioni complesse implementate tramite microcodice)



Alternativa:

- individuare le caratteristiche e i **pattern di esecuzione** delle istruzioni macchina generate dai programmi in HHL

- per **semplificare** l'architettura sottostante ad HHL, 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*
- serializzazione 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

- predominanza di istruzioni di **assegnamento**
  - quindi il trasferimento dei dati deve essere efficiente
- 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 (normalizato)

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

	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 salti e chiamate di procedure evitando i prefetch errati
3. set di istruzioni **semplificato (ridotto)** e implementato in maniera efficiente.

architetture RISC

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?

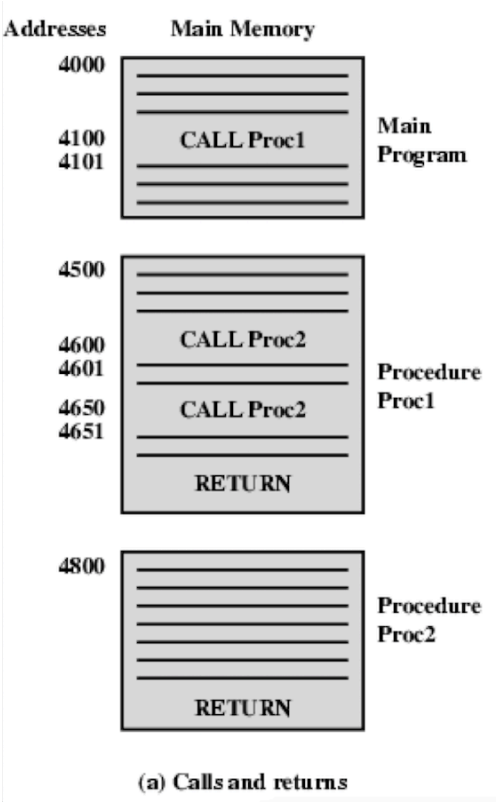
la località **cambia**
ad ogni chiamata/rientro da procedura
(scope)

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

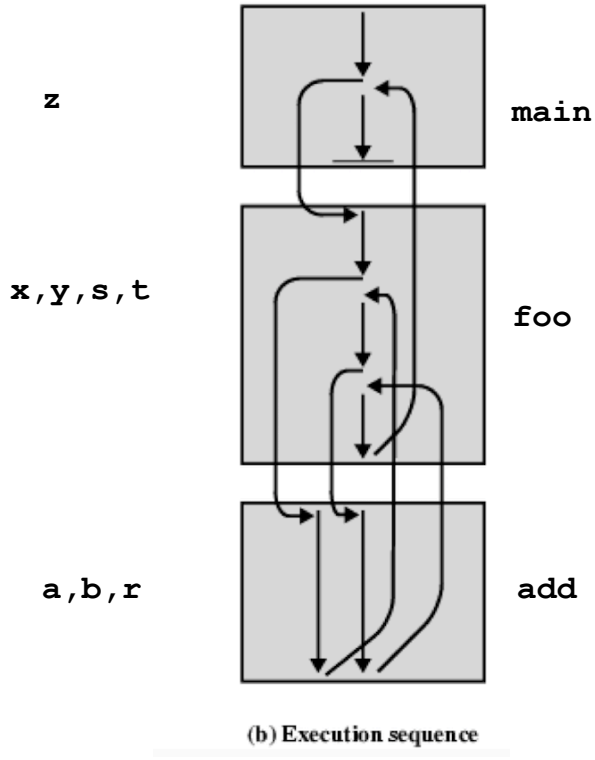
Uso dei registri

```
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



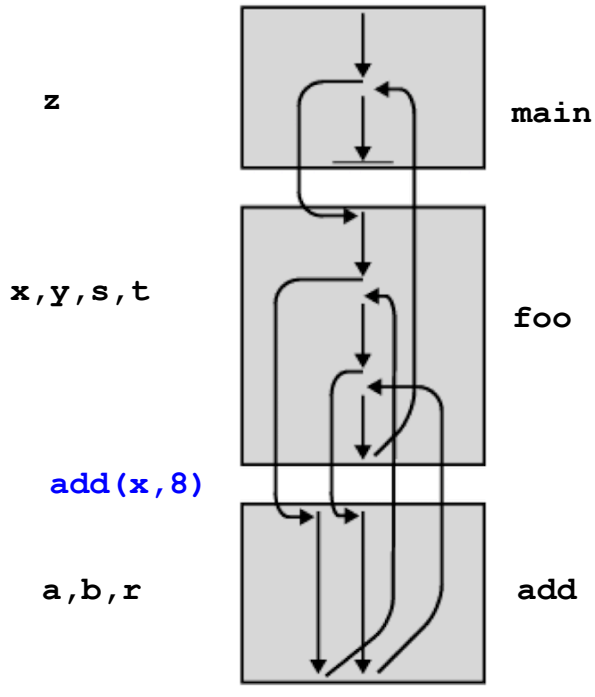
(b) Execution sequence

Uso dei registri

ogni **chiamata** di procedura:
- salva le variabili locali dai registri in memoria
- riusa i registri per le nuove variabili locali
- passa i parametri

al **termine** della precedura:
- restituisce il risultato
- ripristina (i valori del) le variabili locali del chiamante

variabili locali



(b) Execution sequence

Uso dei registri

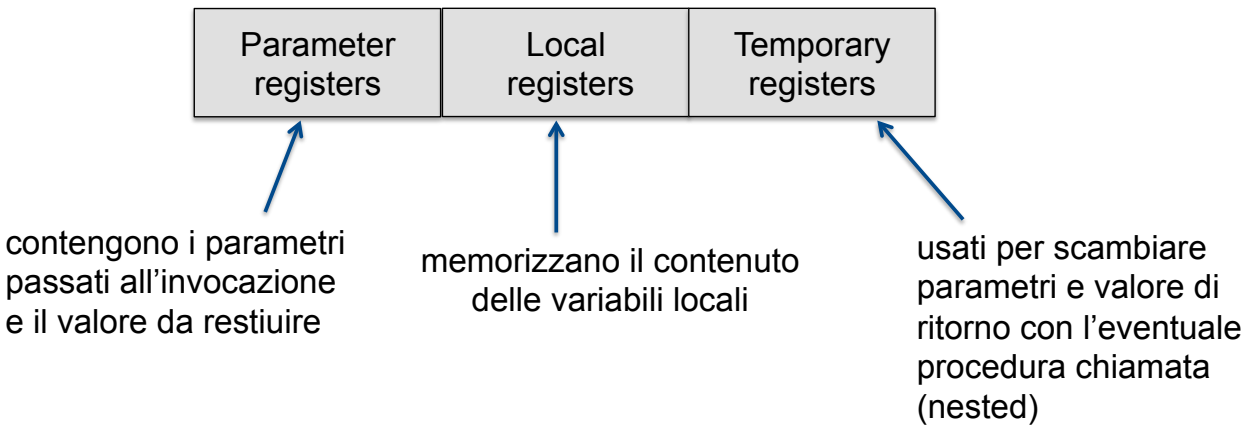
- misurazioni: tipicamente le chiamate di procedura
  - coivolgono pochi parametri (meno di 6)
  - non presentano grado di annidamento elevato

Idea:

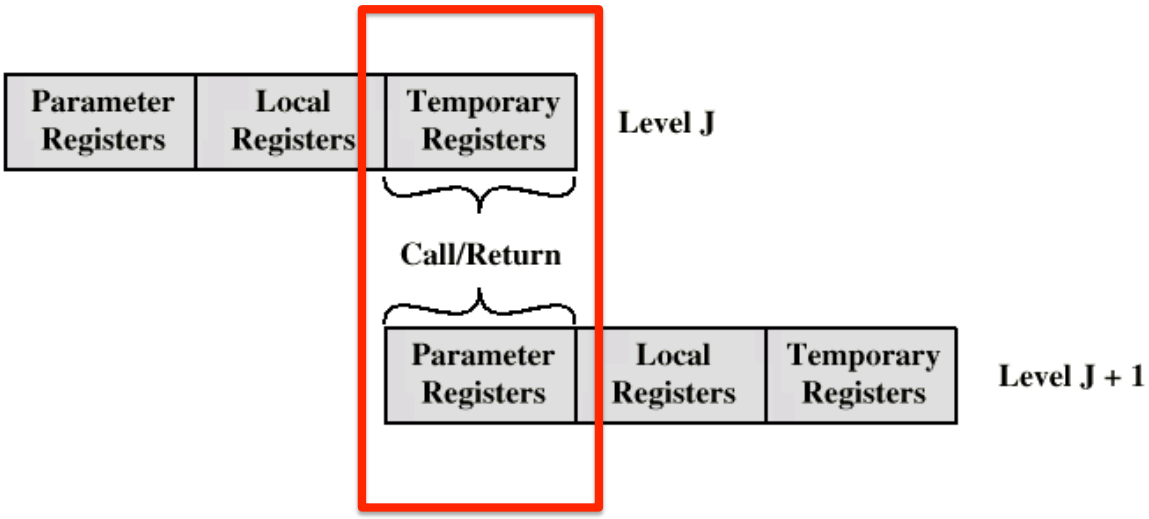
- suddividere i registri in molti piccoli gruppi (di taglia fissa)
- ogni procedura ha il suo gruppo/finestra di registri
- in ogni momento è visibile (indirizzabile) un solo gruppo/finestra
- una chiamata di procedura
  - cambia automaticamente il gruppo di registri da usare
  - invece di provocare il salvataggio dei dati in memoria
  - al ritorno viene riselezionato il gruppo di registri assegnato in precedenza alla procedura chiamante

Finestre di registri

- Ogni gruppo di registri è diviso in tre sottogruppi



Finestre di registri

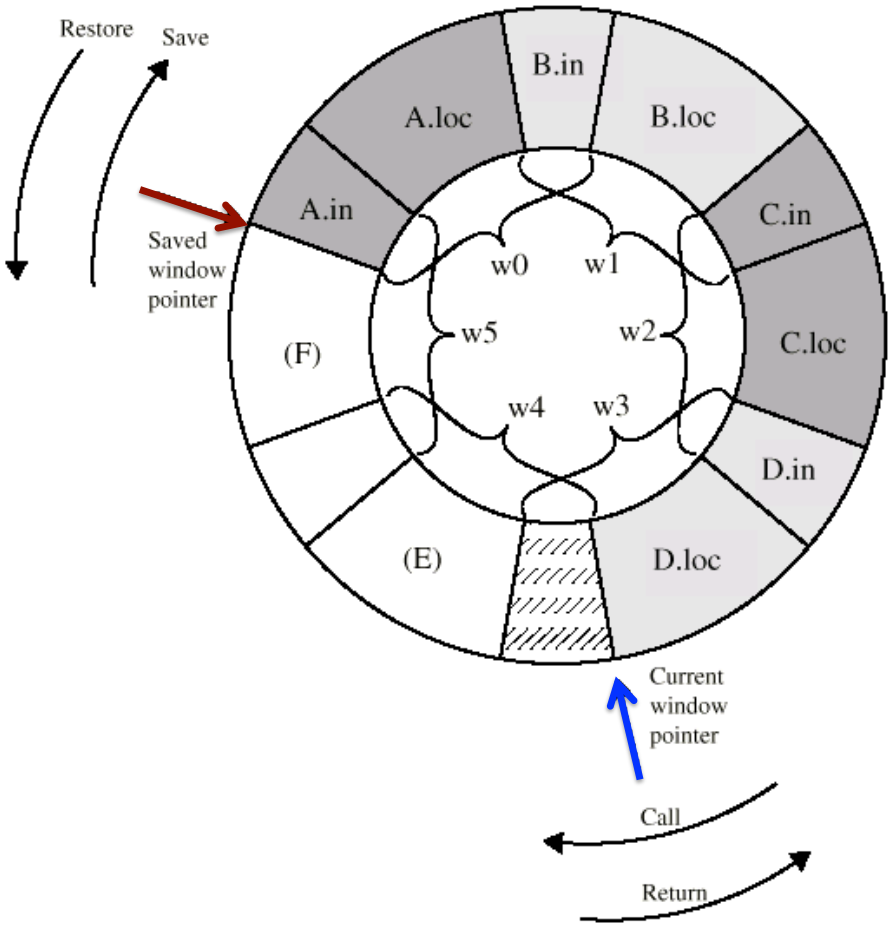


sono fisicamente gli stessi registri

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

Finestre di registri

- 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:

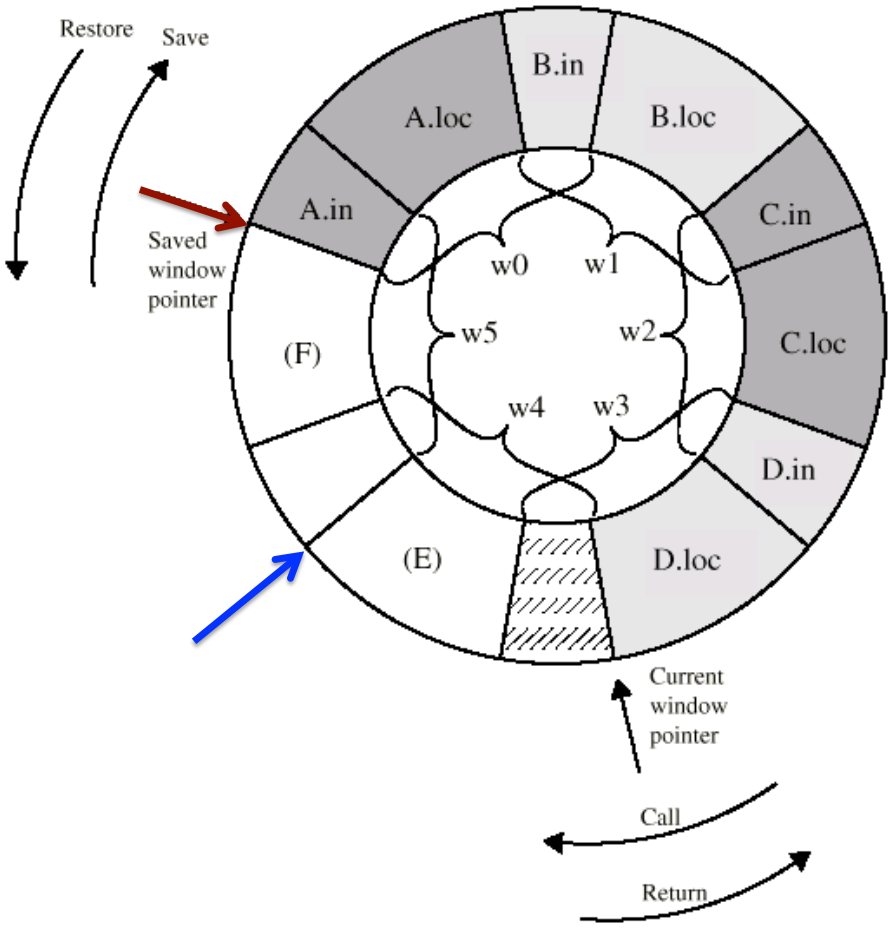
- A -> B -> C -> D
- D procedura attiva

Current Window Pointer

- punta alla finestra della procedura correntemente attiva
- i riferimenti ai registri sono offset a partire dal CWP

Saved Window Pointer

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

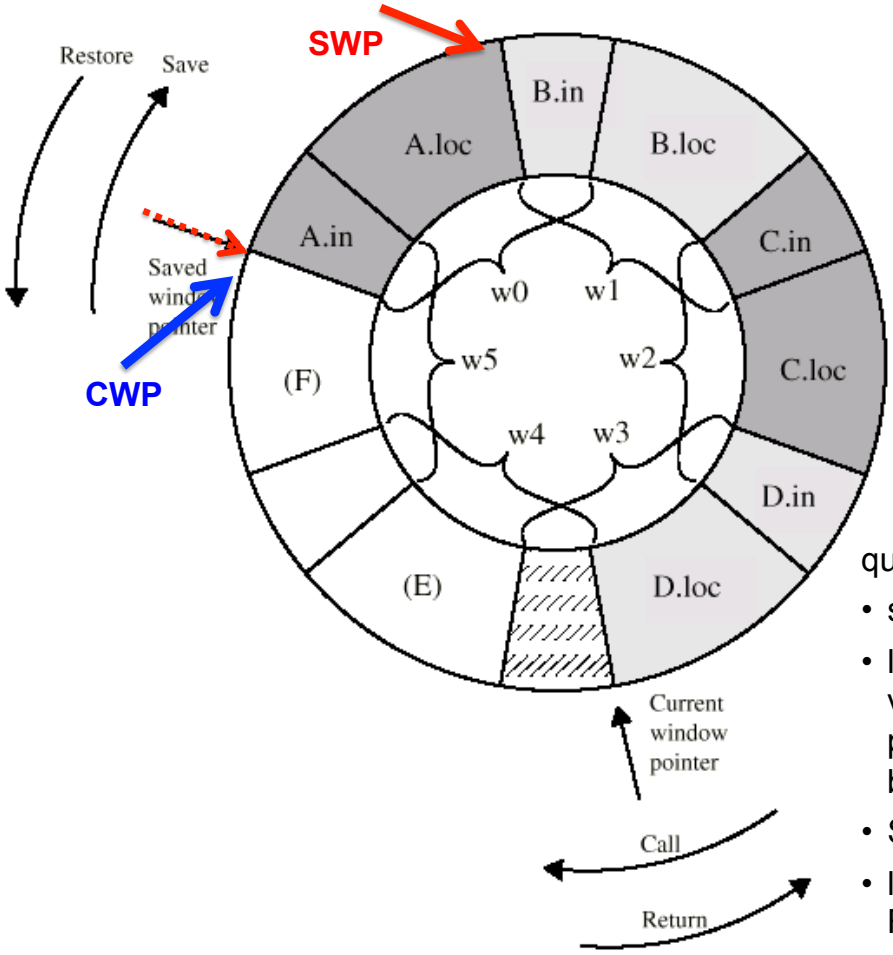


se D chiama E

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

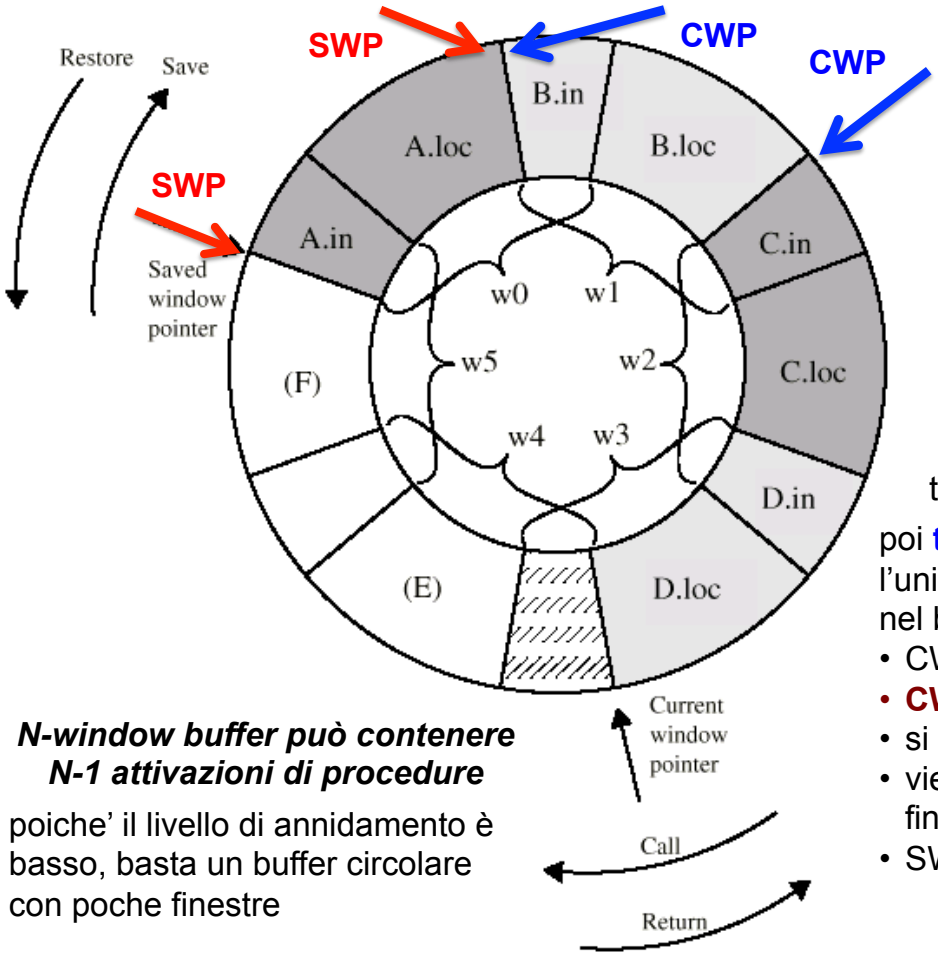
se E chiama F

- **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

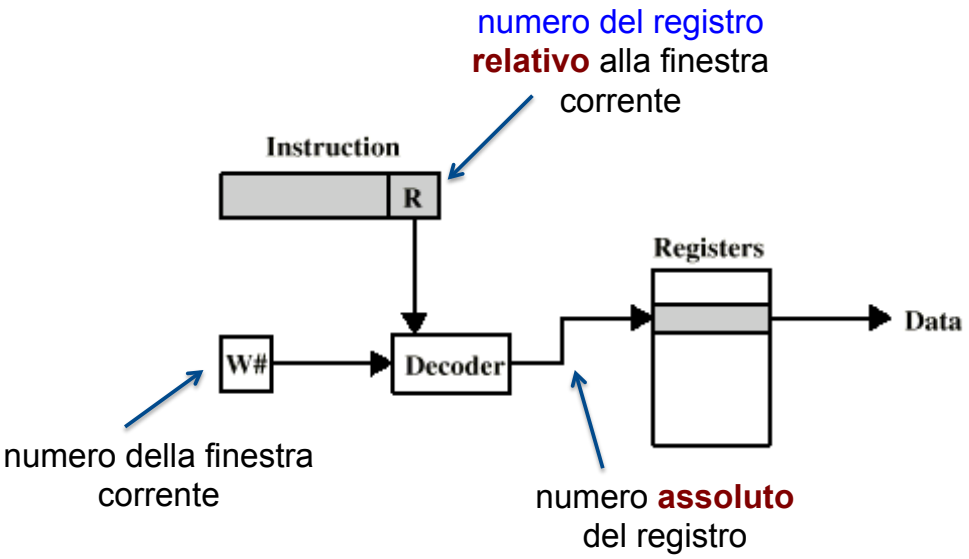
- dove memorizzarle?
  - il compilatore le alloca **in memoria**, ma è poco efficiente se sono usate spesso
  - Soluzione: usare un **gruppo di registri ad hoc**, disponibili a tutte le procedure

Registri “contro” Cache

Banco di Registri Ampio	**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

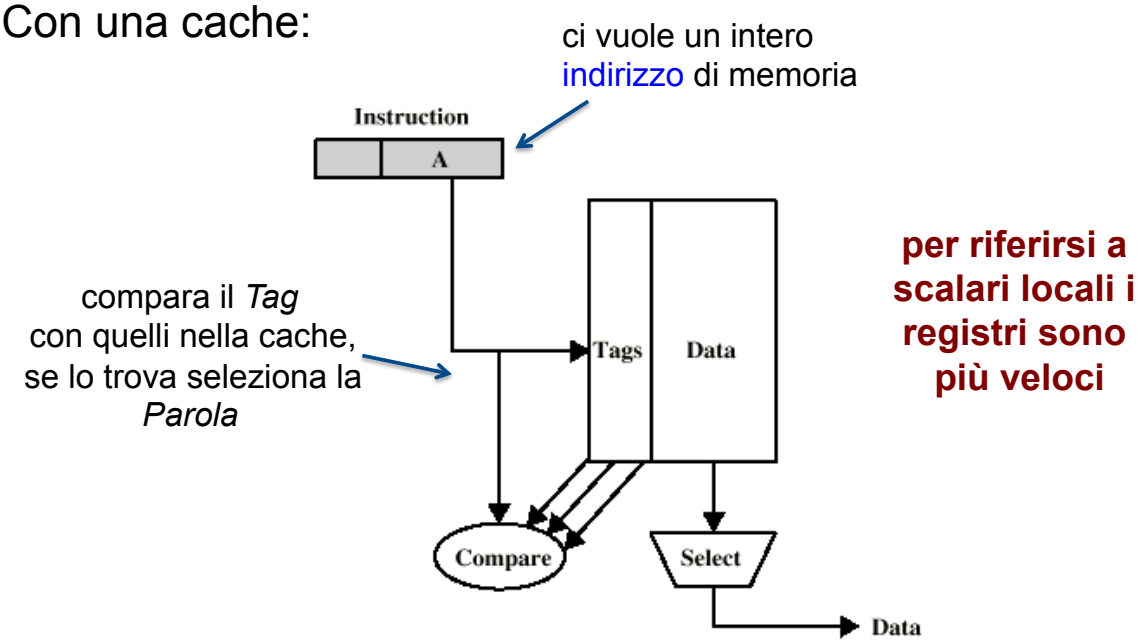
Riferimento ad uno scalare

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