

# z64: Unità di Controllo

*Alessandro Pellegrini*  
*[a.pellegrini@ing.uniroma2.it](mailto:a.pellegrini@ing.uniroma2.it)*

# Esecuzione di un programma

- Un programma è mantenuto in memoria come una sequenza di istruzioni codificate in codice macchina
- Il registro RIP punta alla prossima istruzione da eseguire
- Se non viene eseguita alcuna istruzione di controllo (condizionale) del flusso di programma, il processore esegue un'istruzione dopo l'altra in memoria
- Lo *stato del programma* viene mantenuto in:
  - Registri della CPU
  - Memoria (variabili globali e stack)
- La corretta esecuzione di un programma richiede l'orchestrazione di tutte queste risorse da parte della CPU

# Esecuzione di un programma

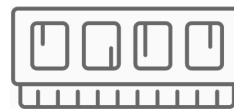
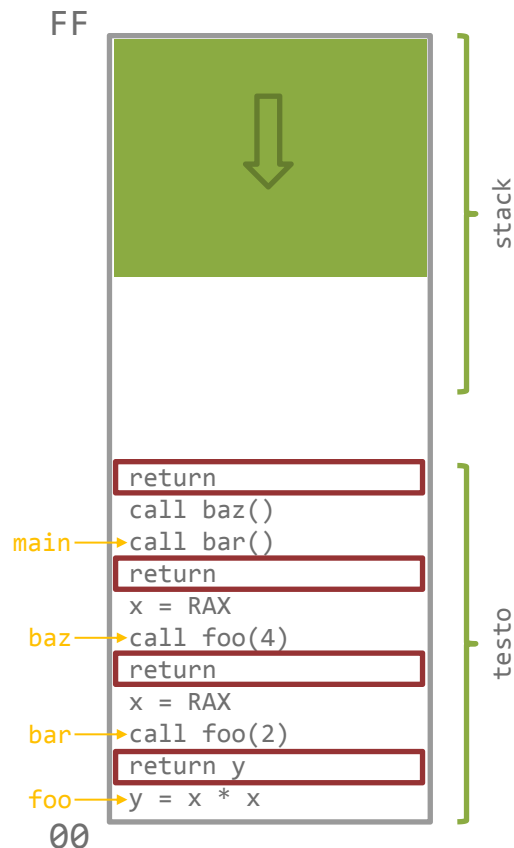
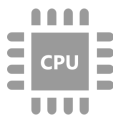
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	
RBX	
...	
IR	
PC	



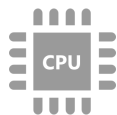
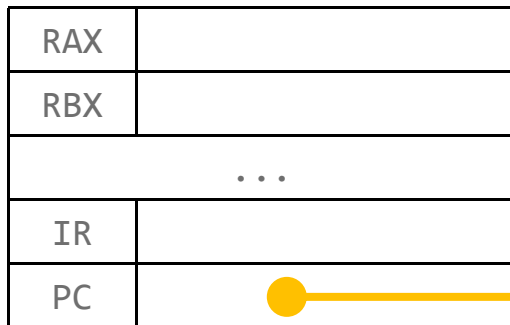
# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

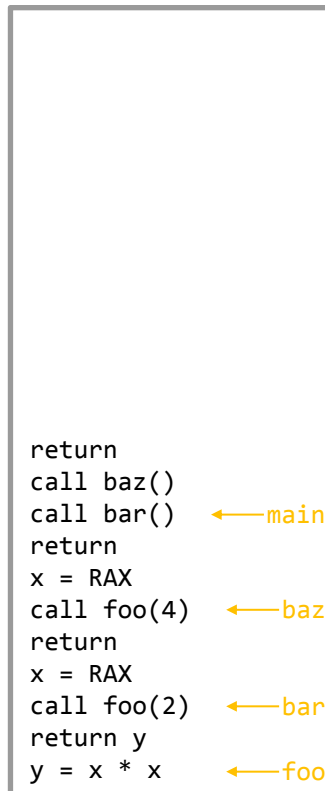
```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

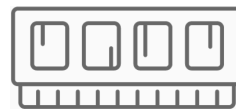
```
int main(void) {  
    bar();  
    baz();  
}
```



FF



00



# Esecuzione di un programma

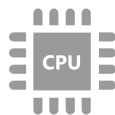
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

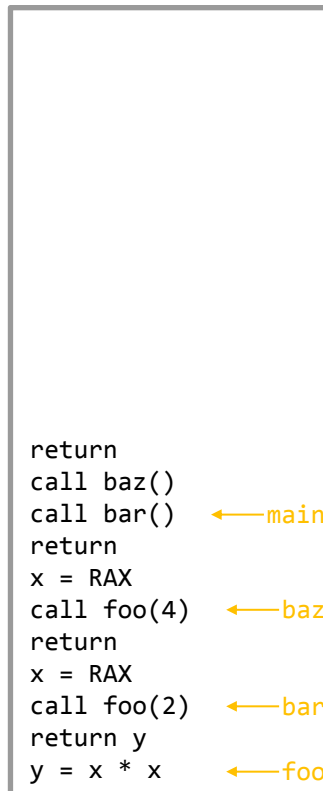
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

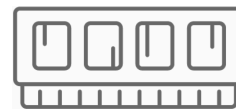
RAX	
RBX	
...	
IR	call bar()
PC	



FF



00



# Esecuzione di un programma

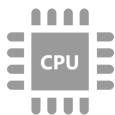
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

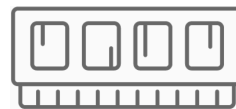
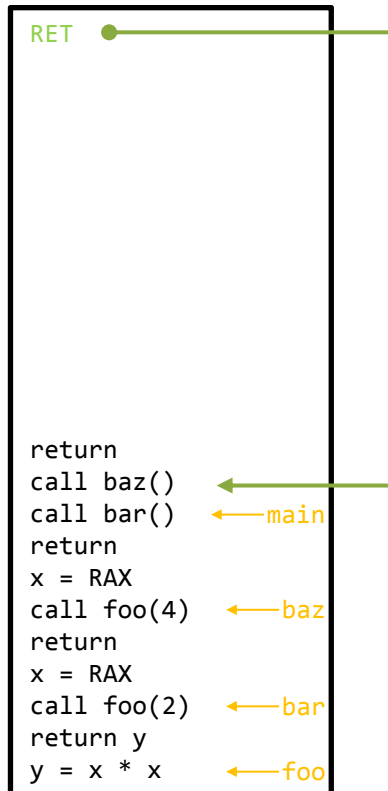
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	
RBX	
...	
IR	call foo(4)
PC	



FF




# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

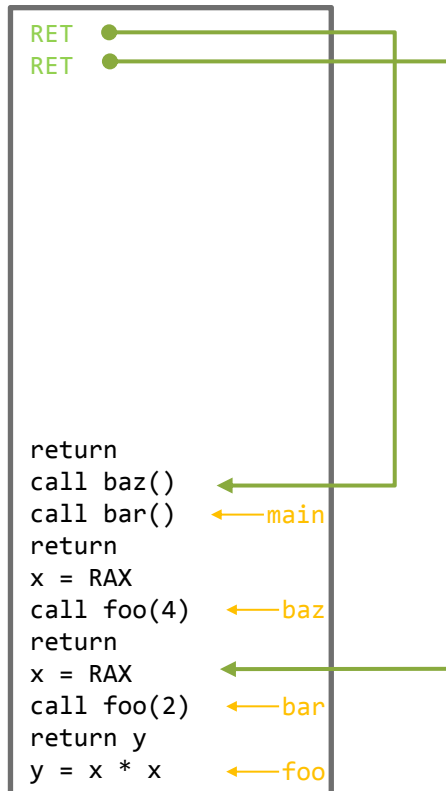
```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

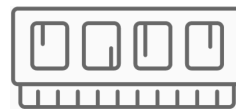
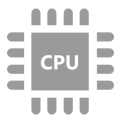
```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	
RBX	
...	
IR	y = x * x
PC	

FF



00




# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

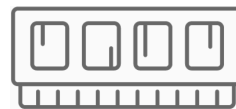
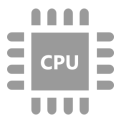
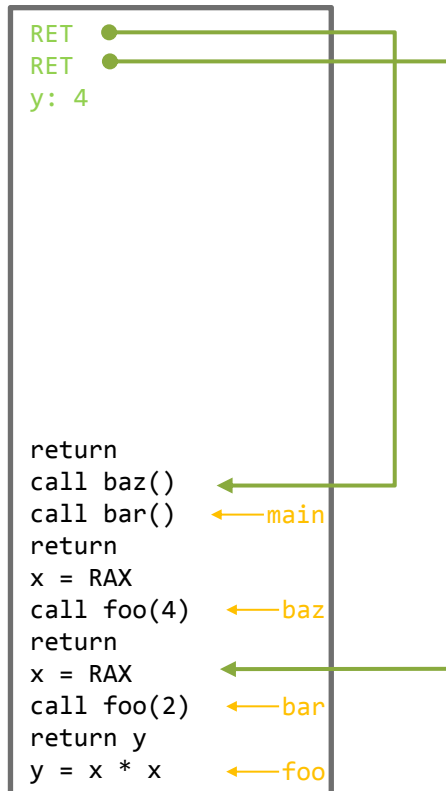
```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	
RBX	
...	
IR	y = x * x
PC	

FF





# Esecuzione di un programma

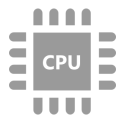
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

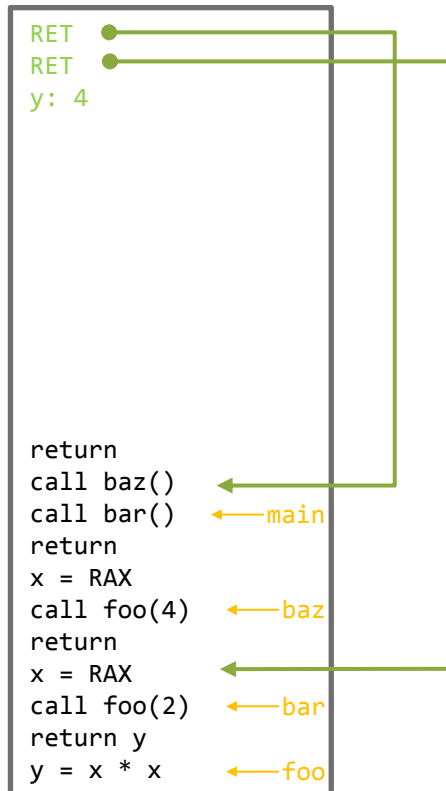
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

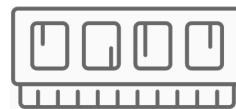
RAX	
RBX	
...	
IR	return y
PC	



FF



00



# Esecuzione di un programma

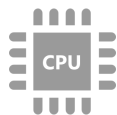
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

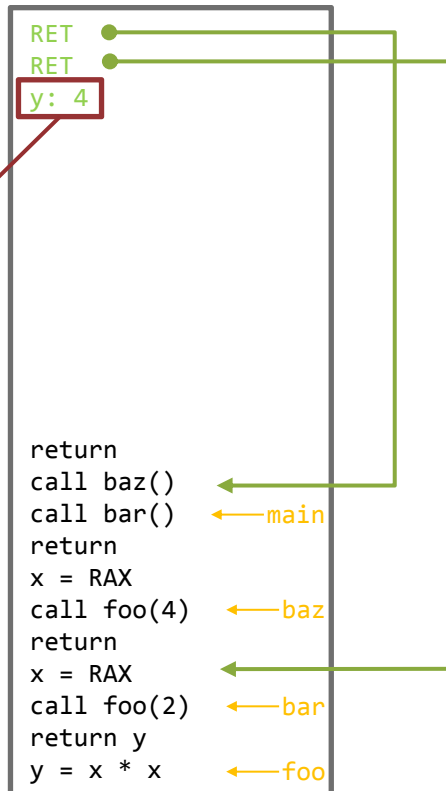
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

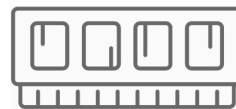
RAX	4
RBX	
...	
IR	return y
PC	



FF



00



# Esecuzione di un programma

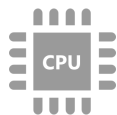
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

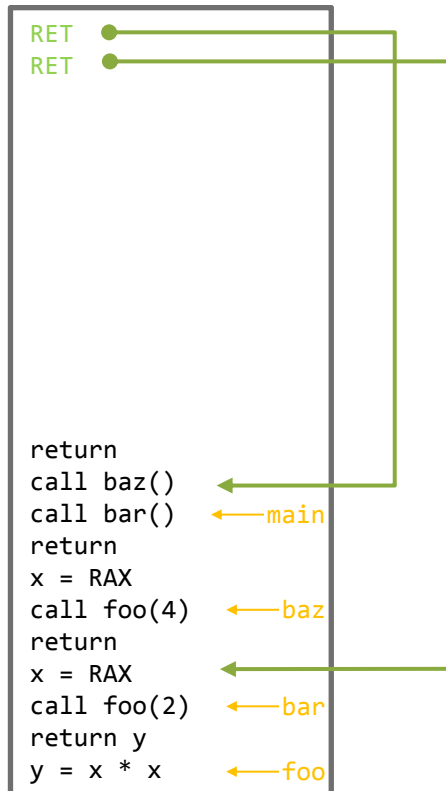
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

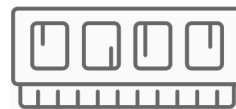
RAX	4
RBX	
...	
IR	return y
PC	



FF

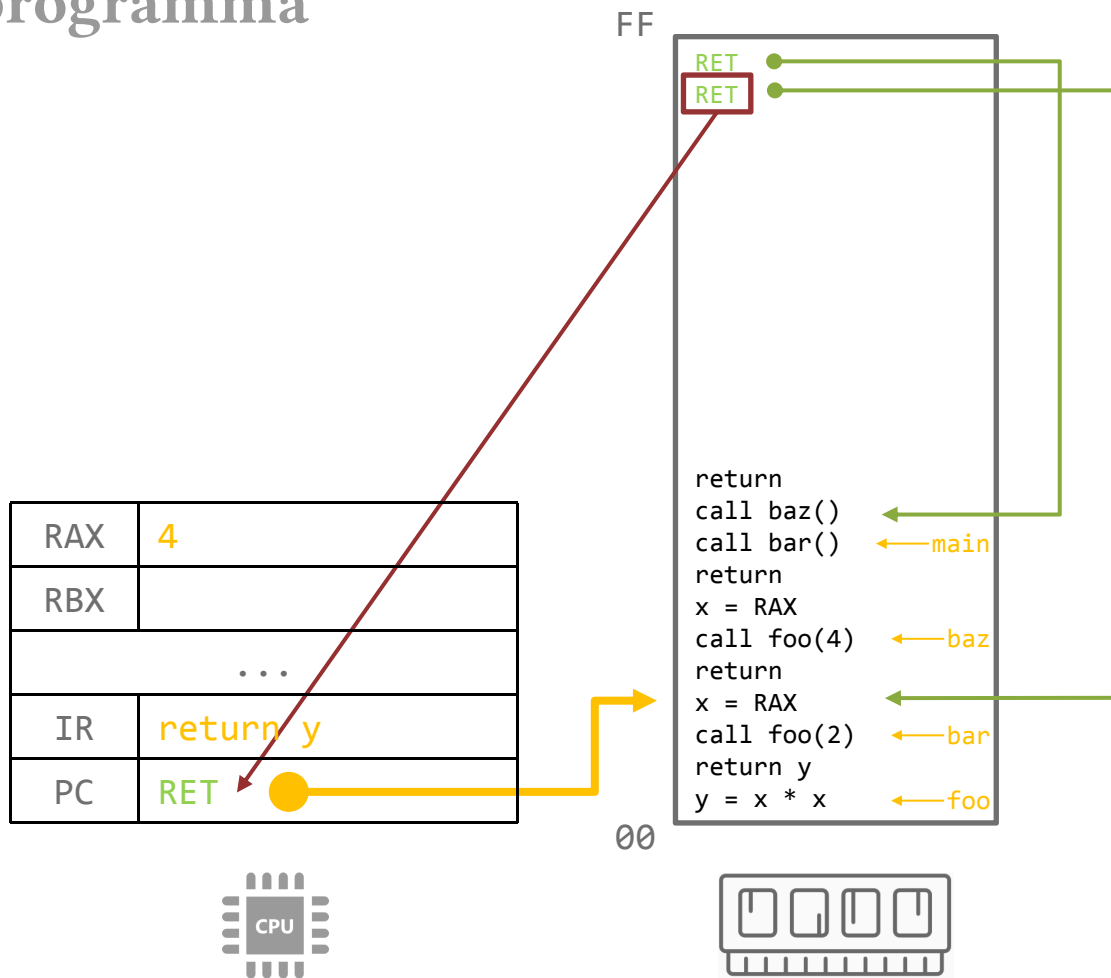


00



# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}  
  
void bar(void) {  
    int x = foo(2);  
}  
  
void baz(void) {  
    int x = foo(4);  
}  
  
int main(void) {  
    bar();  
    baz();  
}
```



# Esecuzione di un programma

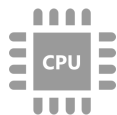
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

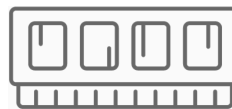
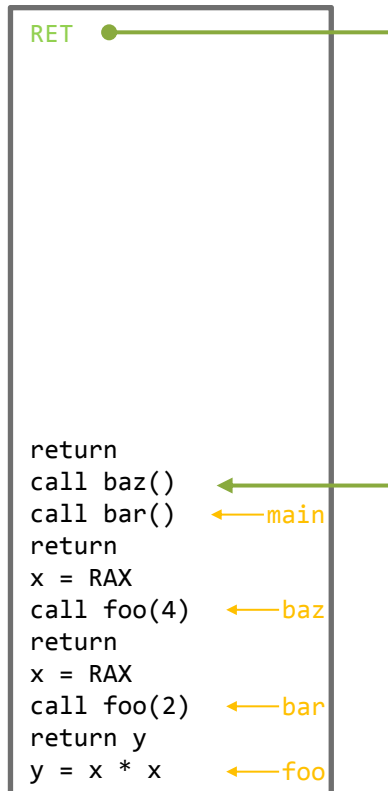
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	return y
PC	



FF



# Esecuzione di un programma

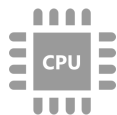
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

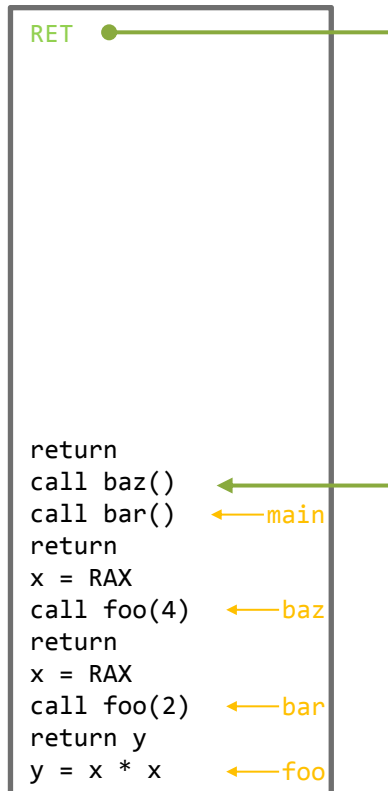
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

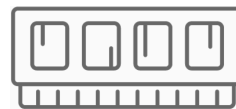
RAX	4
RBX	
...	
IR	x = RAX
PC	



FF



00



# Esecuzione di un programma

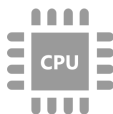
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

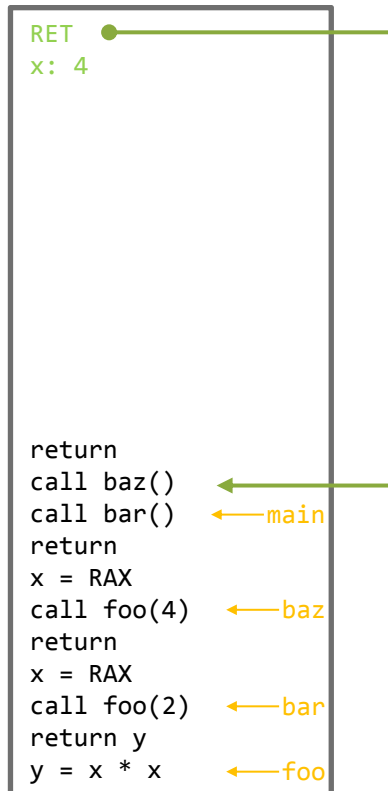
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

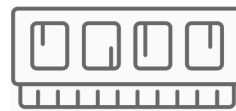
RAX	4
RBX	
...	
IR	x = RAX
PC	



FF



00



# Esecuzione di un programma

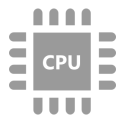
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

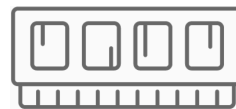
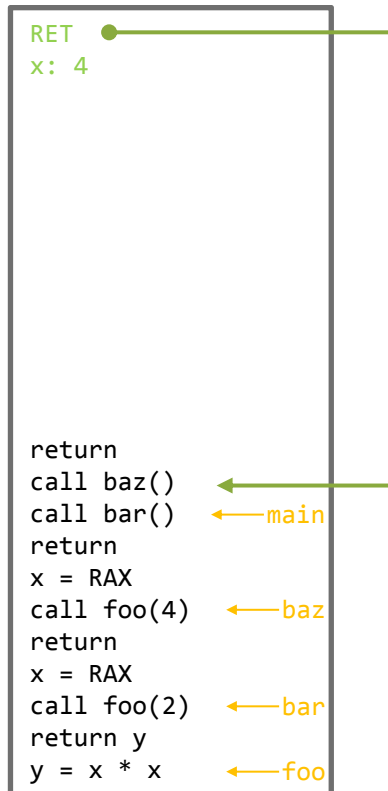
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	return
PC	



FF





# Esecuzione di un programma

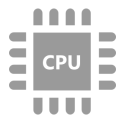
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

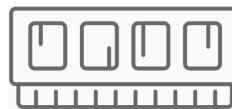
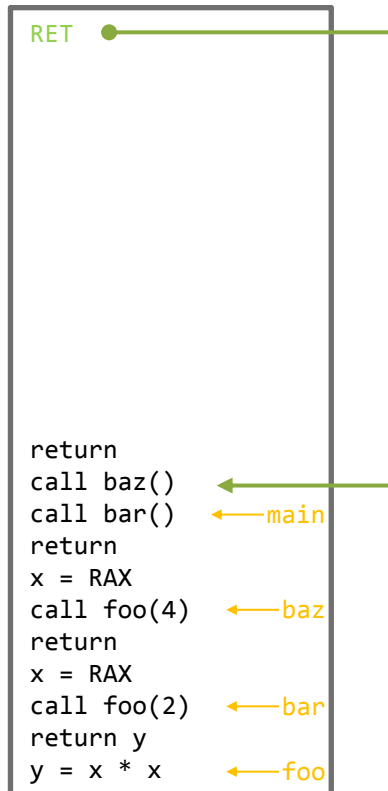
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	return
PC	



FF



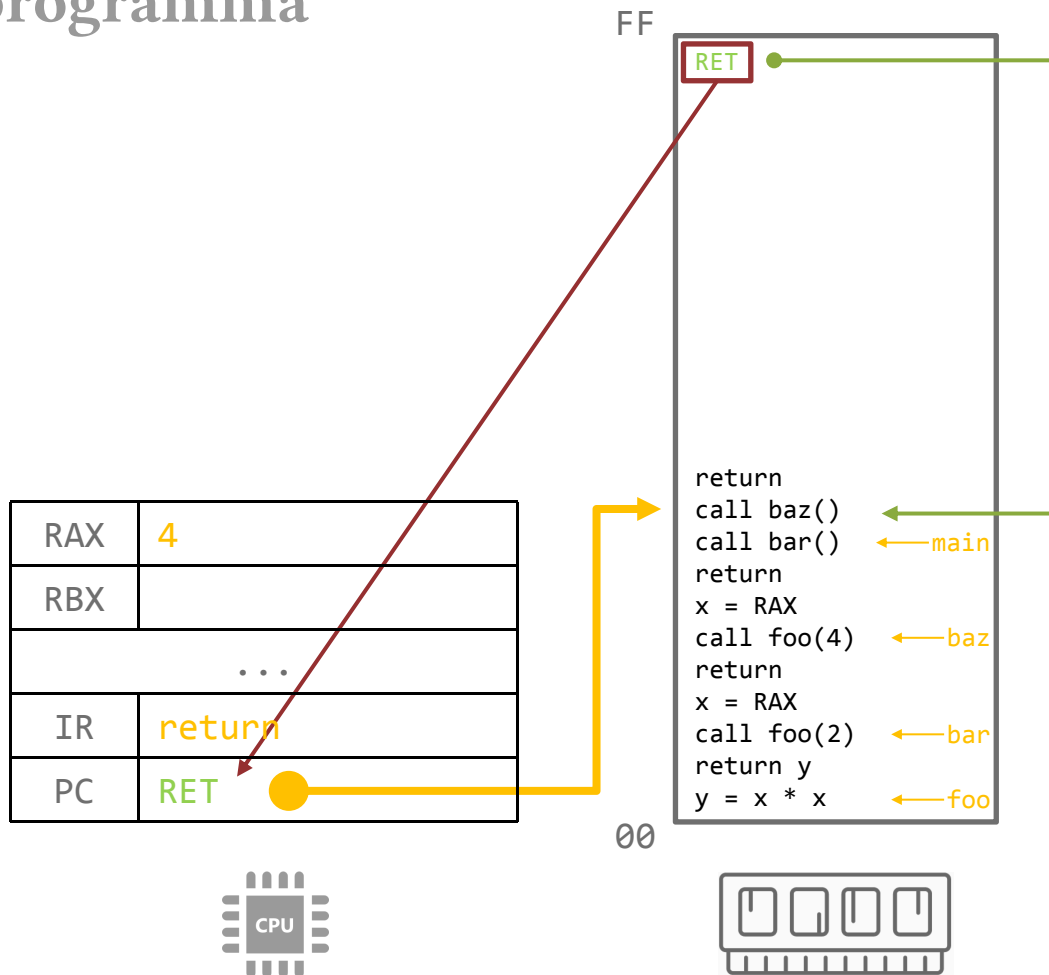
# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```



# Esecuzione di un programma

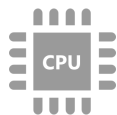
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

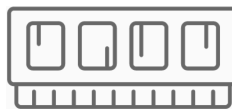
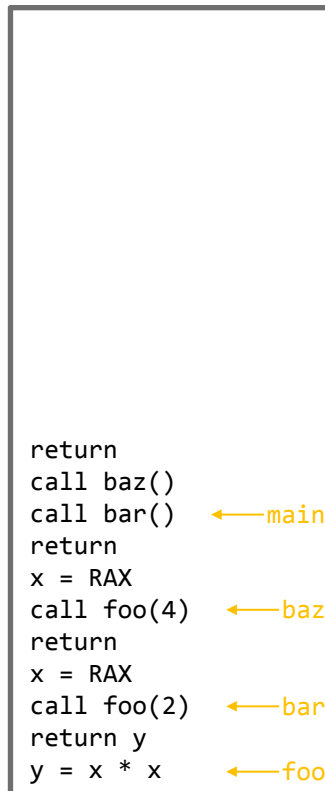
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	call baz()
PC	



FF



# Esecuzione di un programma

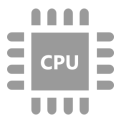
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

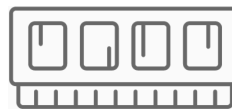
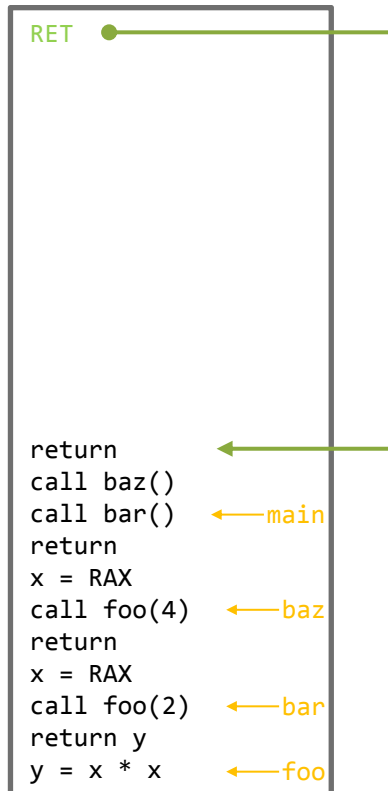
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	call foo(4)
PC	



FF




# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

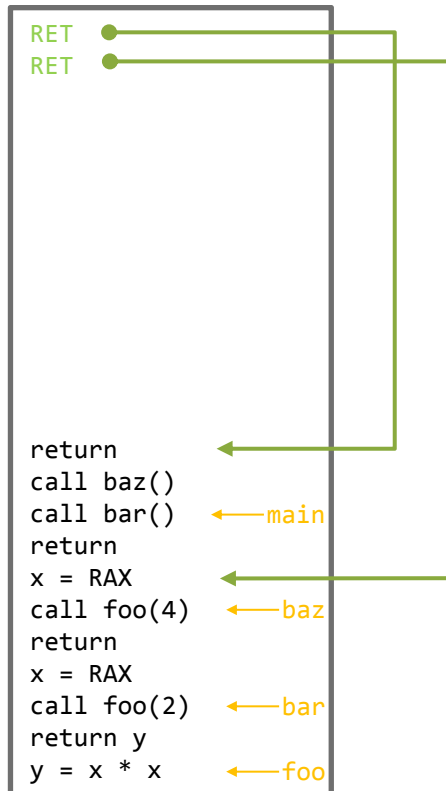
```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

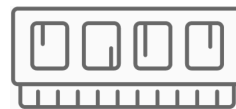
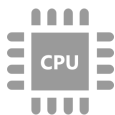
```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	y = x * x
PC	

FF



00




# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

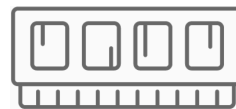
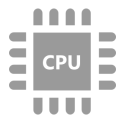
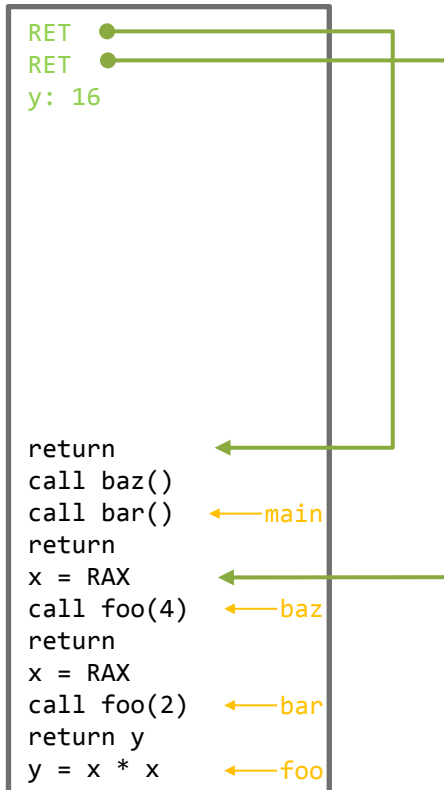
```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	y = x * x
PC	

FF



# Esecuzione di un programma

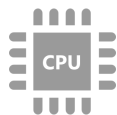
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

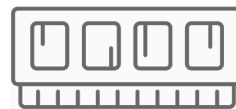
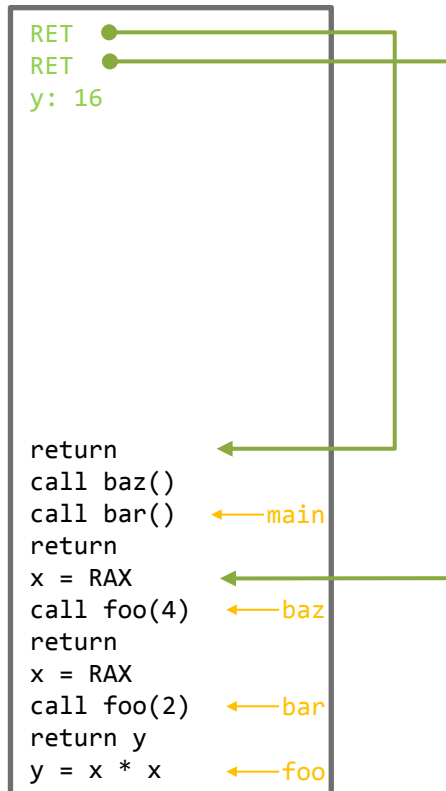
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	4
RBX	
...	
IR	return y
PC	



FF



# Esecuzione di un programma

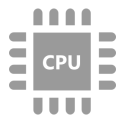
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

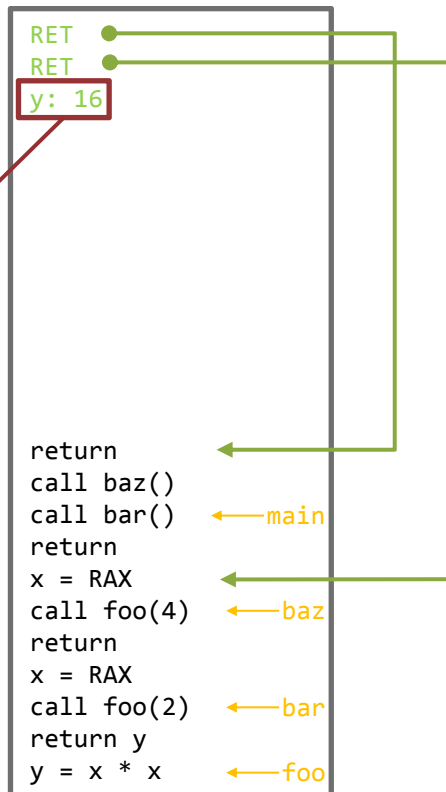
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

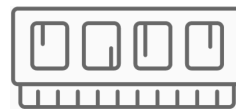
RAX	16
RBX	
...	
IR	return y
PC	



FF



00





# Esecuzione di un programma

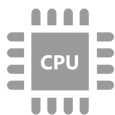
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

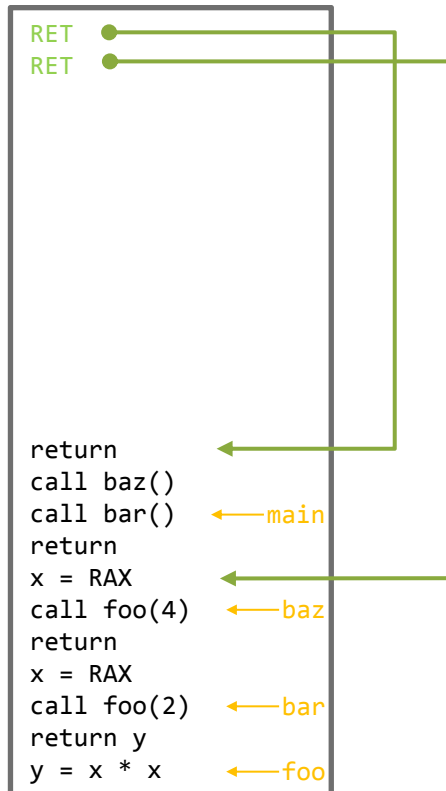
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

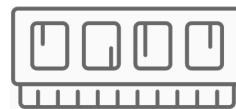
RAX	16
RBX	
...	
IR	return y
PC	



FF

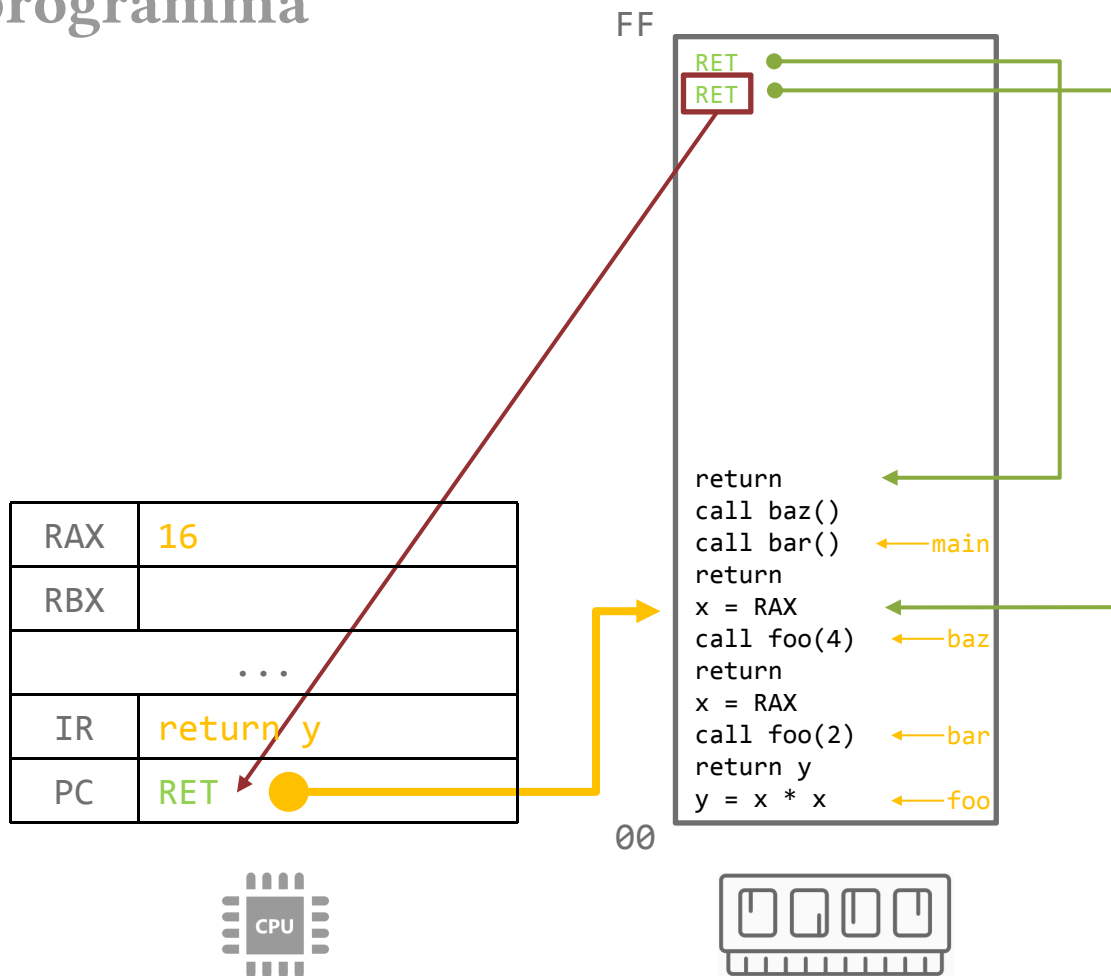


00



# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}  
  
void bar(void) {  
    int x = foo(2);  
}  
  
void baz(void) {  
    int x = foo(4);  
}  
  
int main(void) {  
    bar();  
    baz();  
}
```



# Esecuzione di un programma

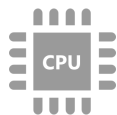
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

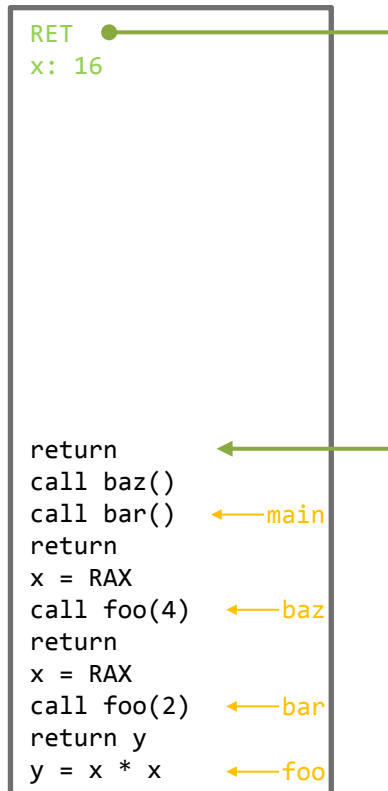
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

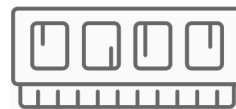
RAX	16
RBX	
...	
IR	return y
PC	



FF



00



# Esecuzione di un programma

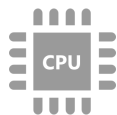
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

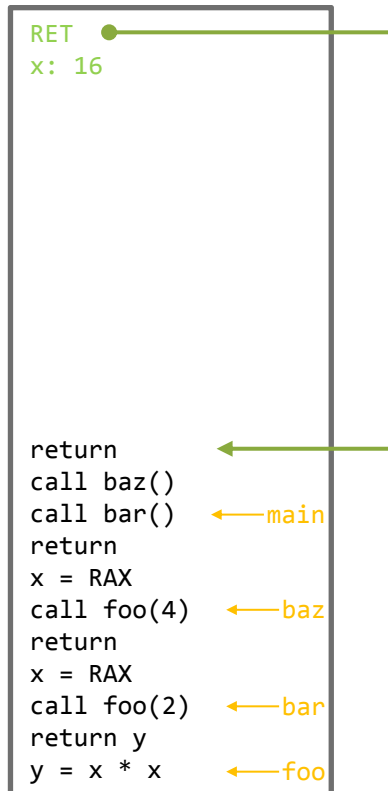
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

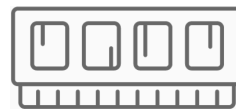
RAX	16
RBX	
...	
IR	return y
PC	RET



FF



00



# Esecuzione di un programma

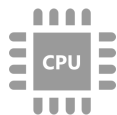
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

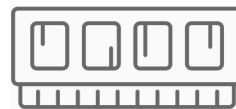
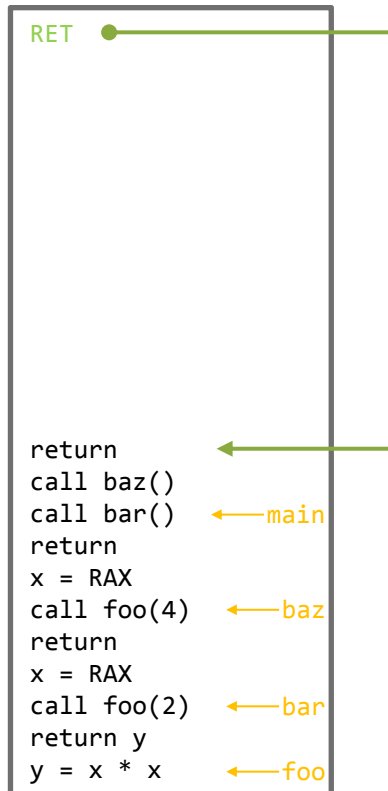
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	16
RBX	
...	
IR	return y
PC	



FF



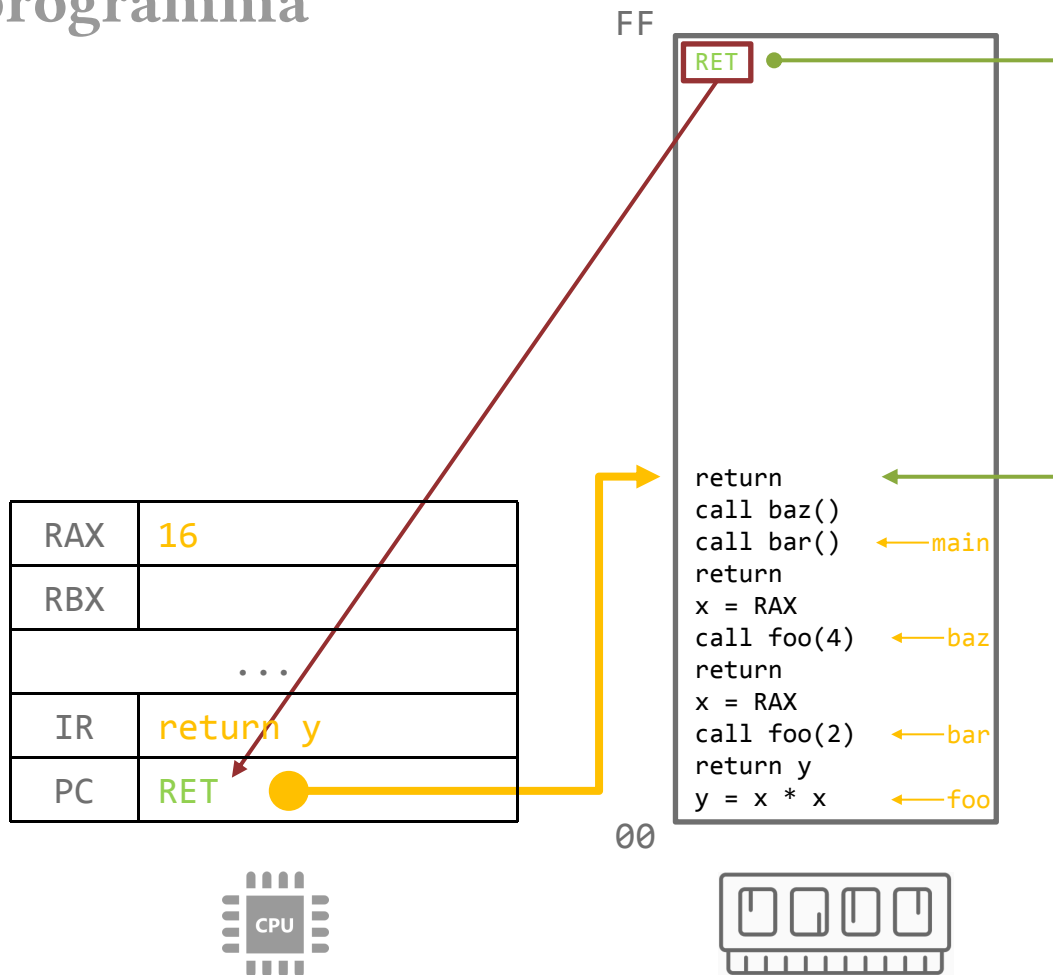
# Esecuzione di un programma

```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```



# Esecuzione di un programma

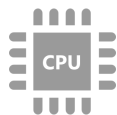
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

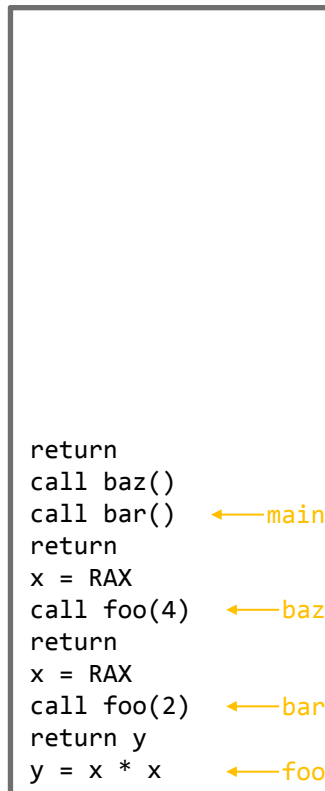
```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

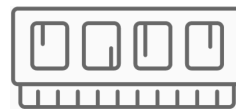
RAX	16
RBX	
...	
IR	return y
PC	



FF



00



# Esecuzione di un programma

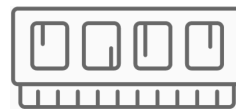
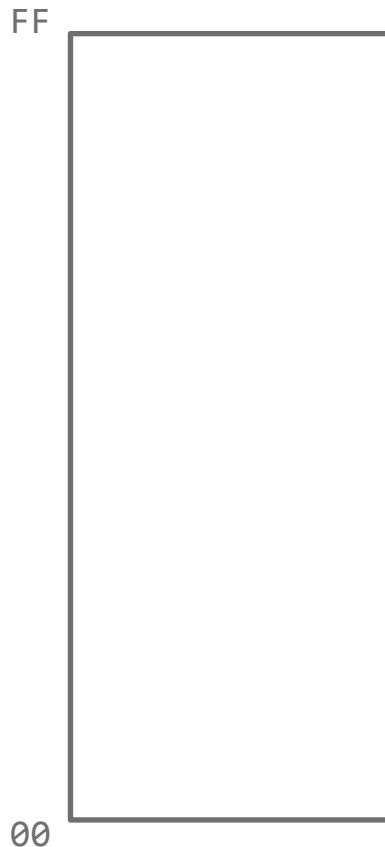
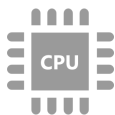
```
int foo(int x) {  
    int y = x * x;  
    return y;  
}
```

```
void bar(void) {  
    int x = foo(2);  
}
```

```
void baz(void) {  
    int x = foo(4);  
}
```

```
int main(void) {  
    bar();  
    baz();  
}
```

RAX	16
RBX	
...	
IR	return y
PC	



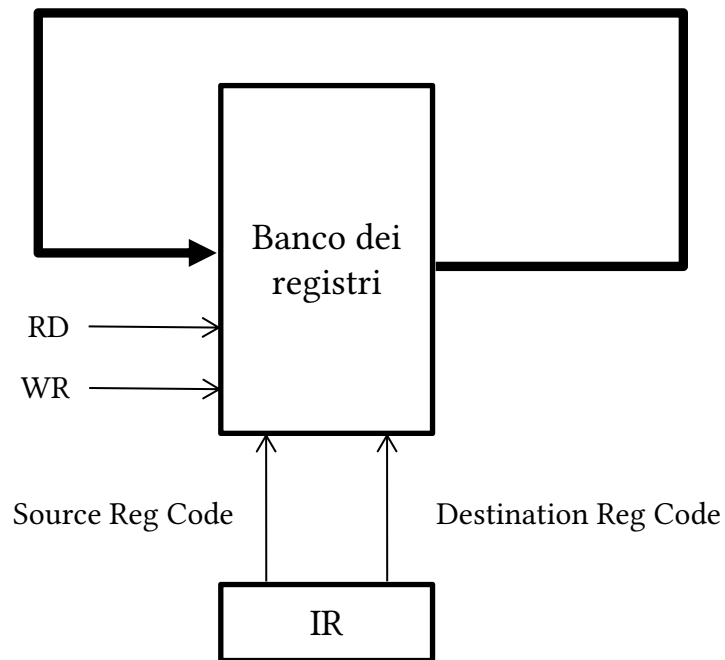


# Processamento delle istruzioni

- Le istruzioni che compongono un programma possono avere un comportamento complesso
  - es: accesso a memoria, utilizzo di più componenti hardware, ...
- L'unità di controllo governa il funzionamento di tutto il sistema, anche delle componenti esterne
- Ad alto livello, il processamento delle istruzioni prevede tre fasi:
  - *fetch*: prelievo della rappresentazione binaria dell'istruzione dalla memoria di lavoro
  - *decodifica*: l'istruzione prelevata viene interpretata per capire *come* questa dovrà essere eseguita
  - *esecuzione*: la semantica dell'istruzione viene effettivamente implementata
- L'unità di controllo deve essere progettata coerentemente con il *datapath* della CPU
  - Modello *uniciclo* o *multiciclo*

# Uniciclo: un esempio di datapath

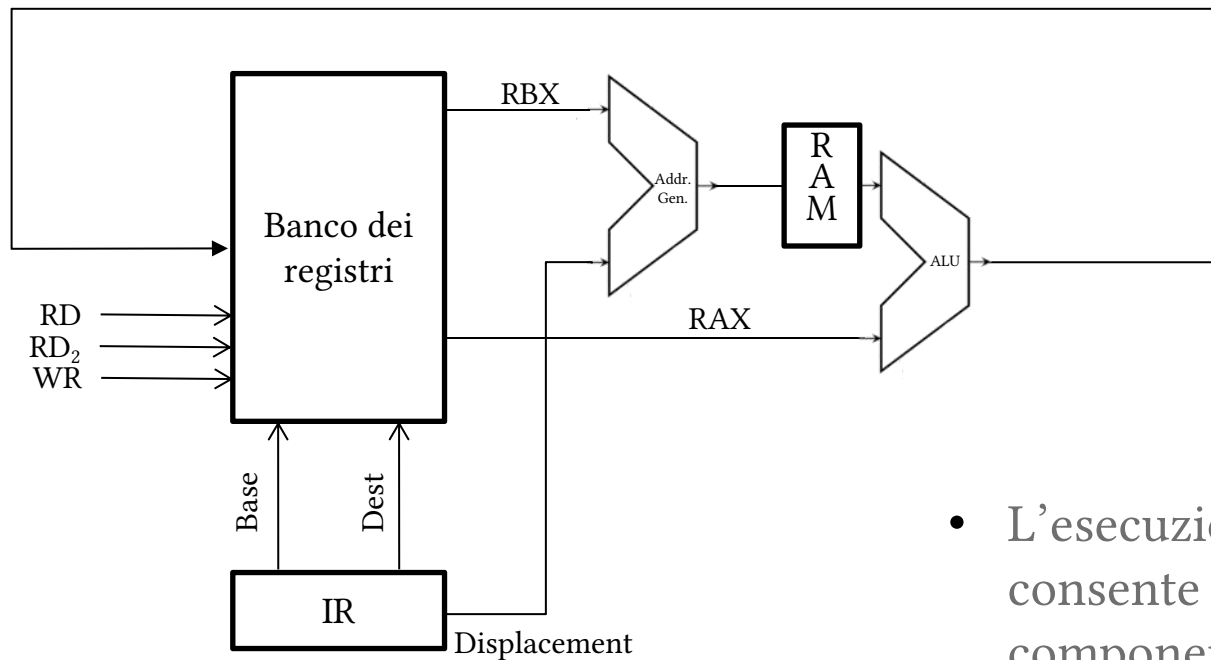
`movw %ax, %bx`



- Si può campionare il valore in lettura sul *fronte di salita* ed effettuare la scrittura sul *fronte di discesa*

# Uniciclo: un altro esempio di datapath

`add 0xabc(%rbx), %rax`



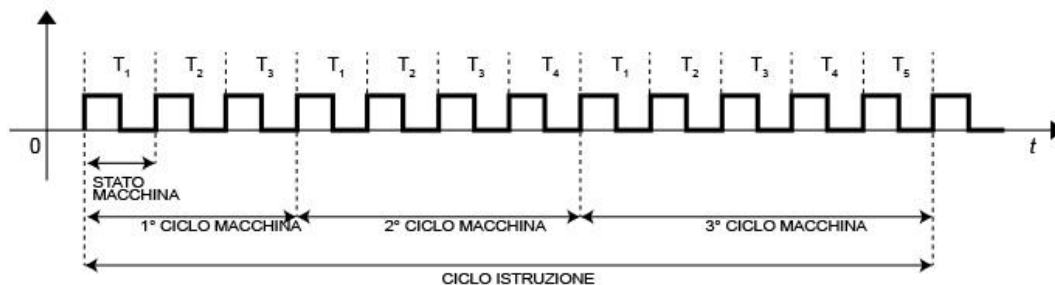
- L'esecuzione più semplice non consente il riutilizzo delle componenti hardware

# Modelli di esecuzione

- Modello *uniciclo*
  - Un'istruzione viene eseguita in un solo colpo di clock
  - Il periodo di clock deve essere tarato per garantire che tutta la rete si stabilizzi
  - È necessario prevedere componenti hardware dedicate perché non è possibile la condivisione delle componenti
    - Ad esempio, se devo eseguire due somme differenti per eseguire un'istruzione, avrò bisogno di due ALU
- Modello *multiciclo*
  - Un'istruzione è eseguita in più colpi di clock
  - Le componenti possono essere riutilizzate tra un ciclo ed il successivo
- Il datapath che abbiamo implementato per lo z64 segue il modello multiciclo: la CU deve essere implementata di conseguenza

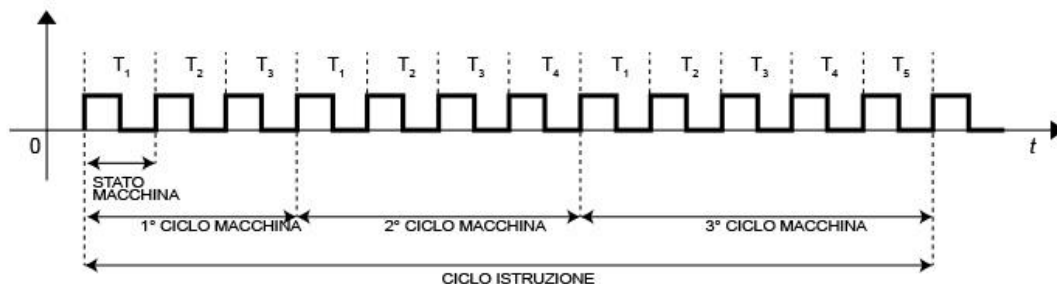
# Ciclo istruzione, ciclo macchina, stato macchina

- *Ciclo istruzione*: intervallo temporale necessario ad eseguire una istruzione nella sua interezza
- *Ciclo macchina*: intervallo temporale necessario ad eseguire una fase (fetch, decode, execute)
  - A seconda del tipo di istruzione, possono essere necessari un numero diverso di cicli macchina (es: più accessi in memoria)
- *Stato macchina*: periodo di tempo necessario per stabilizzare la rete delle unità di controllo e calcolo (corrispondente al clock)



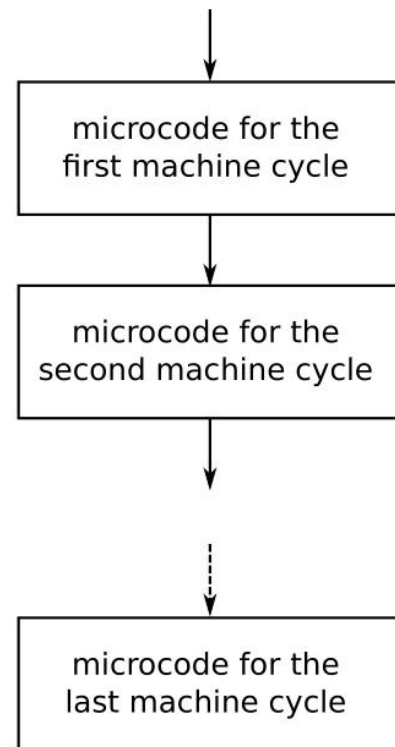
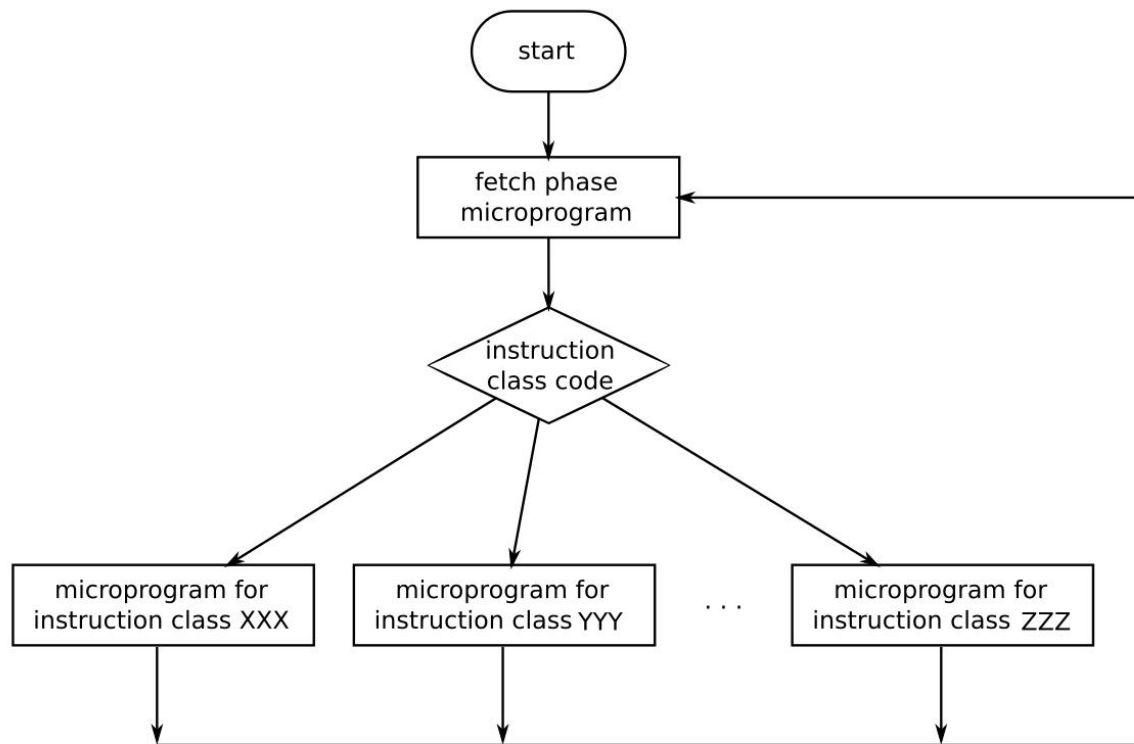
# Implementazione della CU: le Microoperazioni

- La CU implementa un'istruzione con un insieme di *microoperazioni*
- Ogni microoperazione è definita dai segnali di controllo abilitati in uno specifico stato macchina



- Ogni stato macchina corrisponde all'aggiornamento di una *macchina a stati* che implementa i *microprogrammi*

# Realizzazione della CU



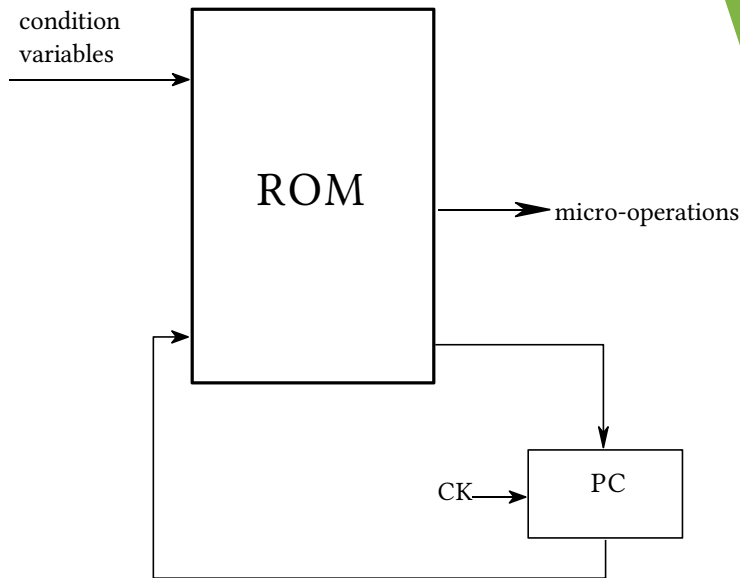
# Organizzazione della CU

- Per eseguire il microprogramma di un'istruzione, la CU userà come input:
  - La classe e il tipo dell'istruzione — contenuta in IR
  - Le variabili di condizione che arrivano da fuori la CPU — interazione con dispositivi di I/O
  - Le variabili di condizione che arrivano da dentro la CPU — ad esempio i bit di FLAGS
  - La modalità di indirizzamento degli operandi dell'istruzione — dedotta da IR
- Il numero di segnali di output della CU dipende dall'implementazione della PU e dei moduli esterni
- L'organizzazione della CU dipende dal costo implementativo, dalle prestazioni desiderate e dal tipo di macchina a stati finiti scelta (Mealy o Moore)



# La CU come macchina di Mealy

- La CU è una rete sequenziale complessa, con molte variabili in input e output
- Le funzioni  $\delta$  e  $\omega$  della macchina possono essere rappresentate tramite una ROM
  - paginata in funzione dei microprogrammi
- Ad ogni colpo di clock, vengono emessi i segnali di controllo per implementare una microoperazione
- La posizione nel microprogramma (indirizzo della ROM) è mantenuto da un registro che opera come program counter



# Paginazione dei microprogrammi

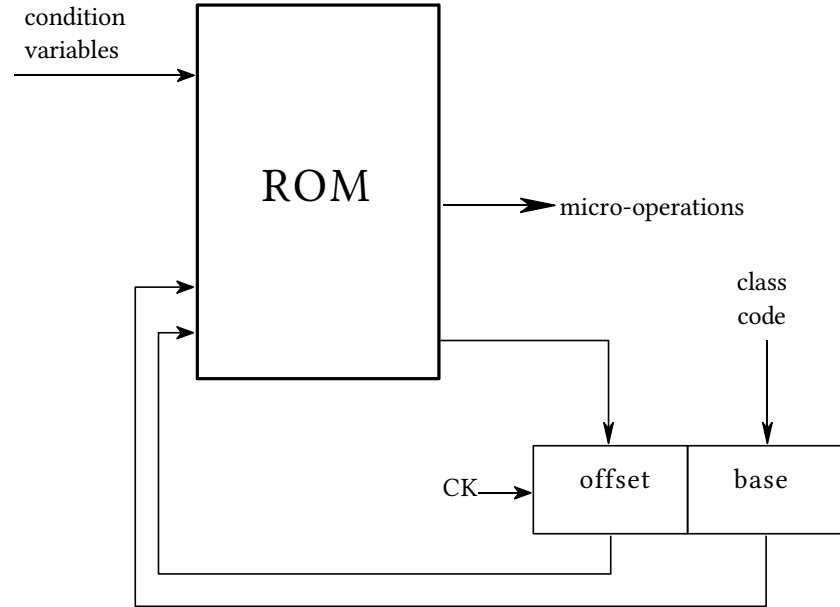
- Ogni pagina è associata ad un microprogramma di una classe di operazioni
  - Per questo le operazioni “simili” sono raggruppate nella stessa classe
- Non tutti i microprogrammi hanno la stessa lunghezza
  - Frammentazione *interna*

ROM dei microprogrammi

Pagina 0
Pagina 1
Pagina 2
Pagina 3
Pagina 4
Pagina 5
Pagina 6
Pagina 7

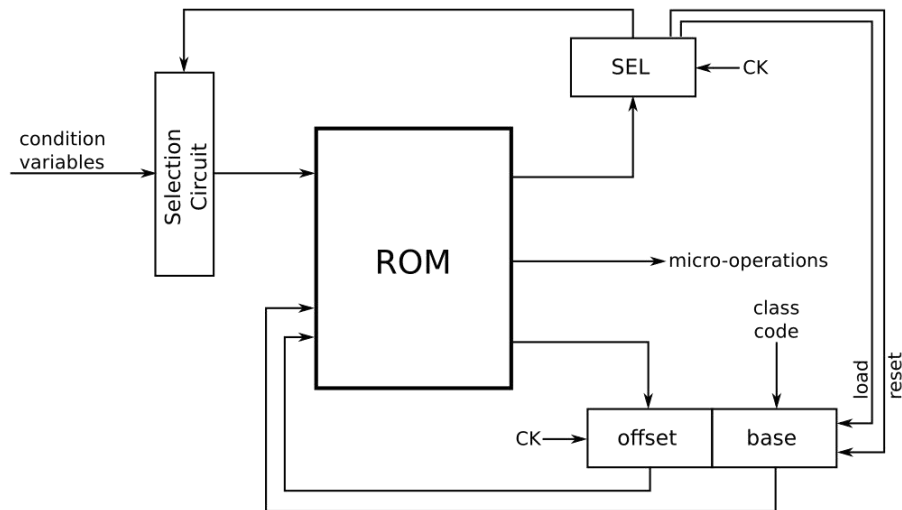
# Fase di decodifica

- Con la paginazione, la posizione interna al microprogramma può diventare un offset a partire da una base
- La decodifica di un'istruzione può quindi essere svolta inizializzando il registro di base con il codice della classe con il codice della classe



# Riduzione delle variabili in input

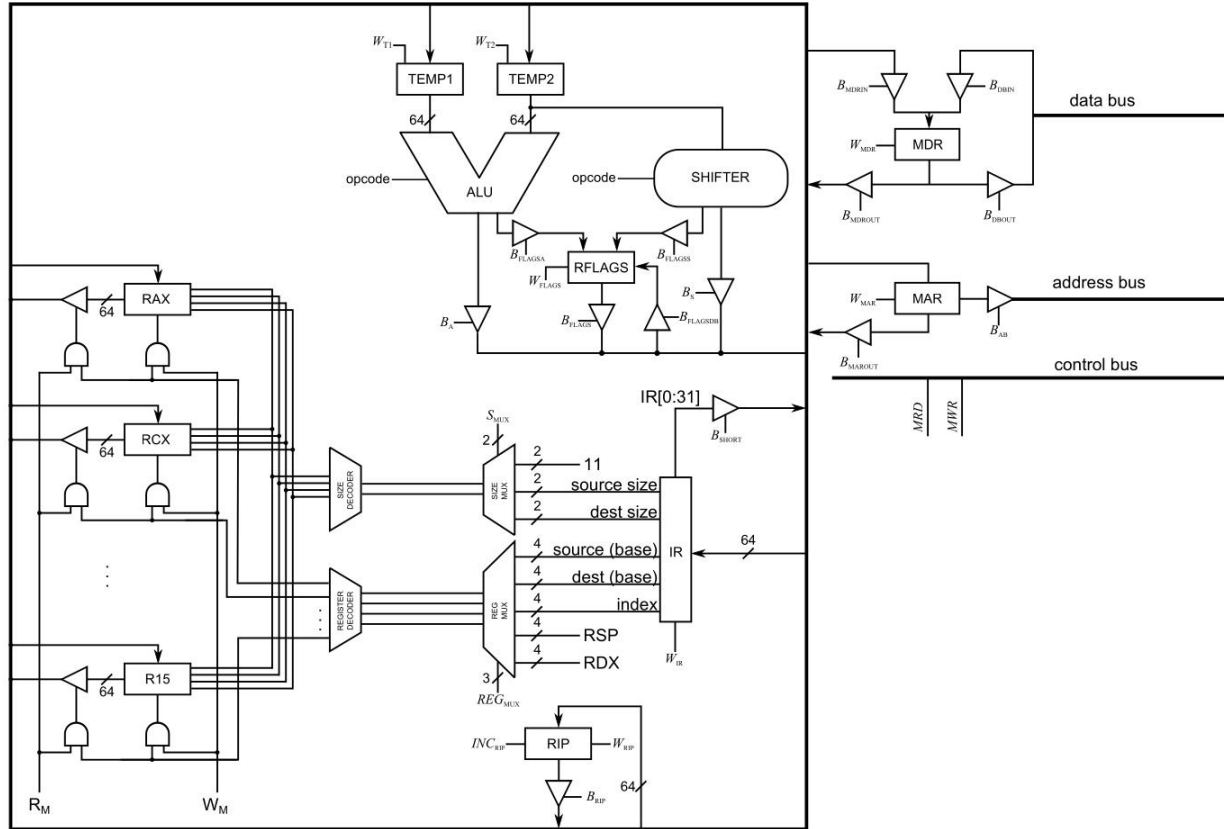
- Il numero di variabili in input può essere grande
  - Complessità di sintesi della macchina elevata
- Non tutti i segnali servono per eseguire tutti i passi dei microprogrammi
- Si può implementare una strategia di selezione
  - Si identifica la microoperazione con il numero massimo di variabili in input
  - Si redirezionano le variabili in input su un numero minore di segnali



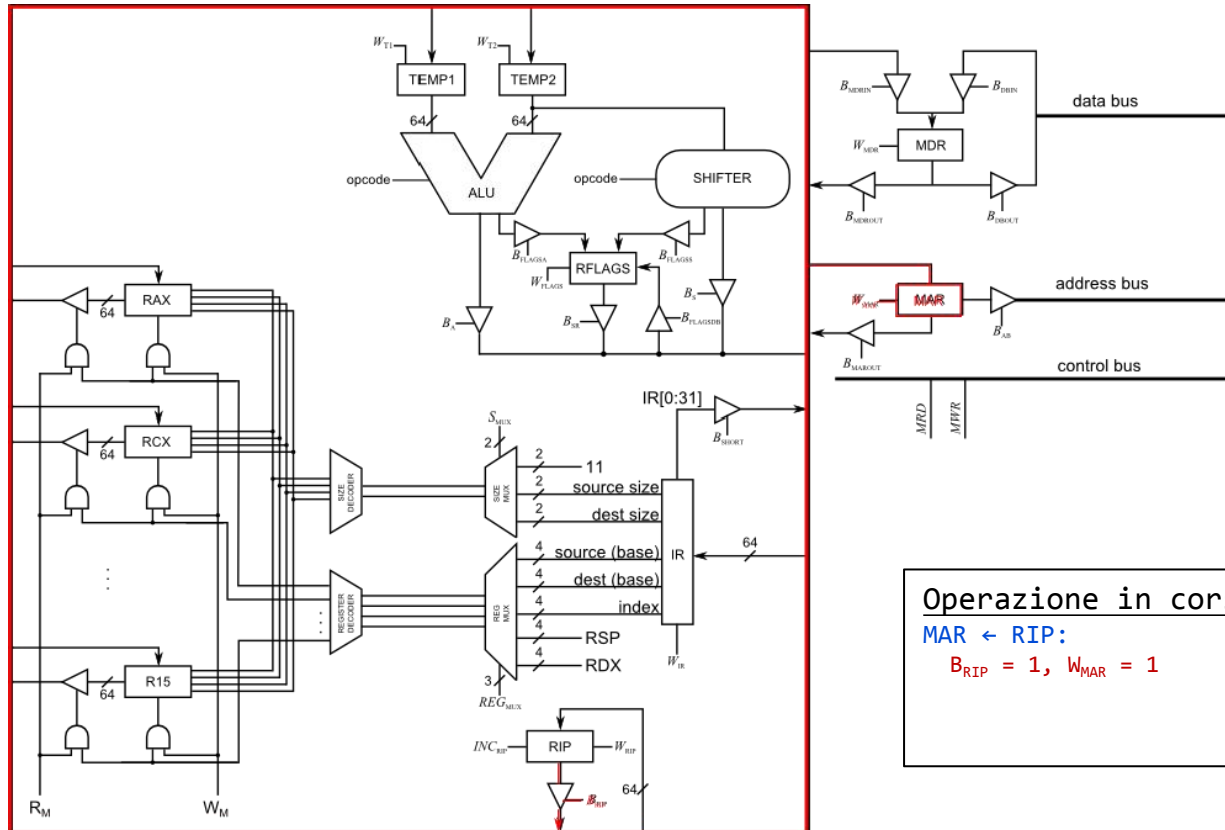
# La fase di fetch

- L'esecuzione di ogni istruzione incomincia con la fase di fetch
- Le microoperazioni associate alla fase di fetch sono:
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
- In questo modo, l'istruzione successiva viene caricata nel registro IR (così da poterla interpretare ed eseguire) e il valore di RIP viene incrementato (così da puntare alla prossima istruzione/dato)
- L'utilizzo della classe come base dei microprogrammi implementa automaticamente la decodifica (non è necessaria alcuna microoperazione dedicata)

# La fase di fetch



# La fase di fetch

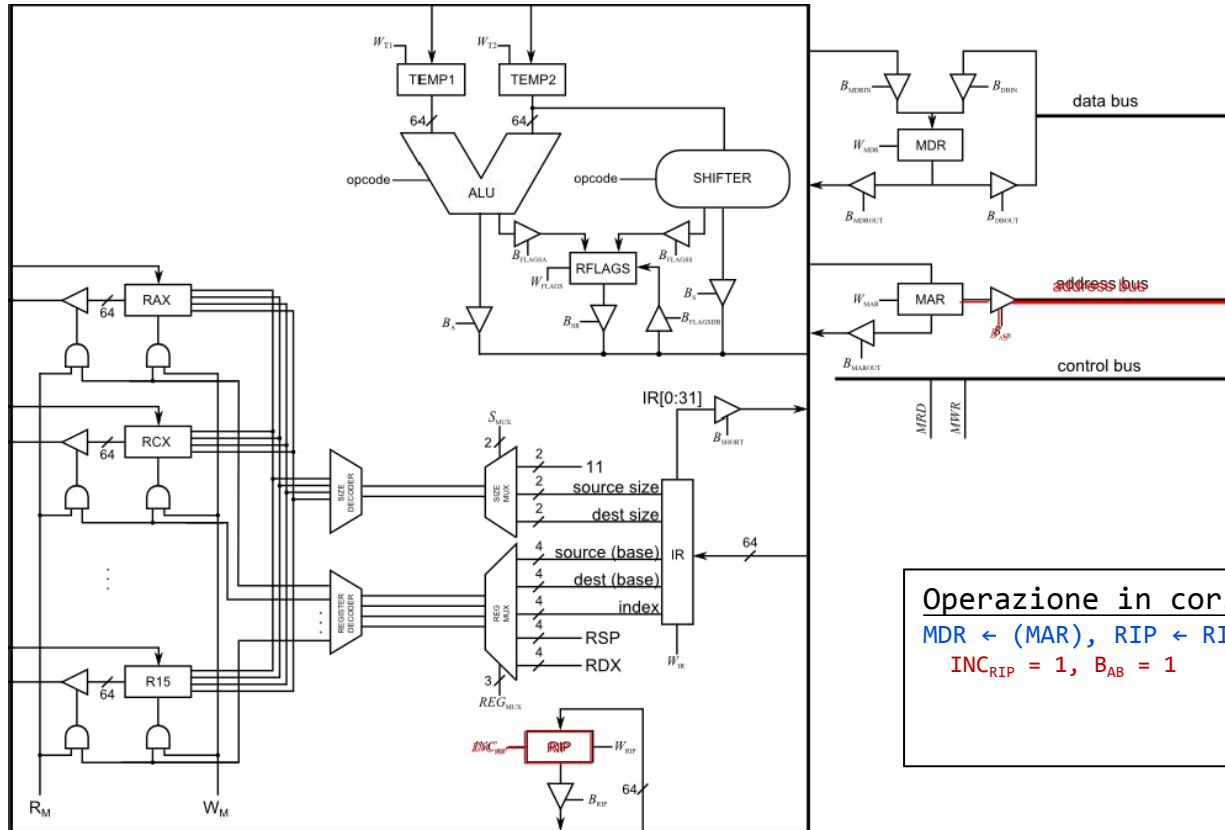


Operazione in corso:

MAR ← RIP:

$$B_{RIP} = 1, W_{MAR} = 1$$

# La fase di fetch



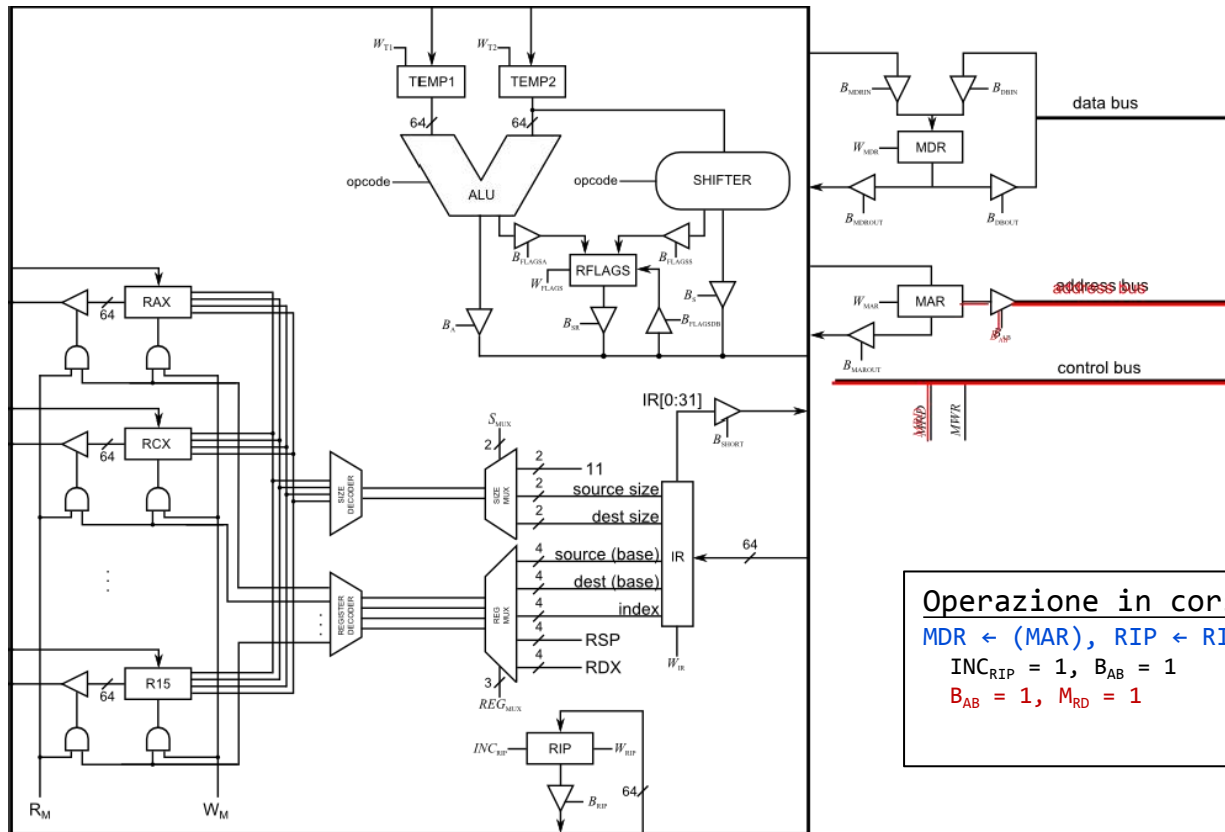
Operazione in corso:

$MDR \leftarrow (MAR), RIP \leftarrow RIP + 8:$

$INC_{RIP} = 1, B_{AB} = 1$



# La fase di fetch



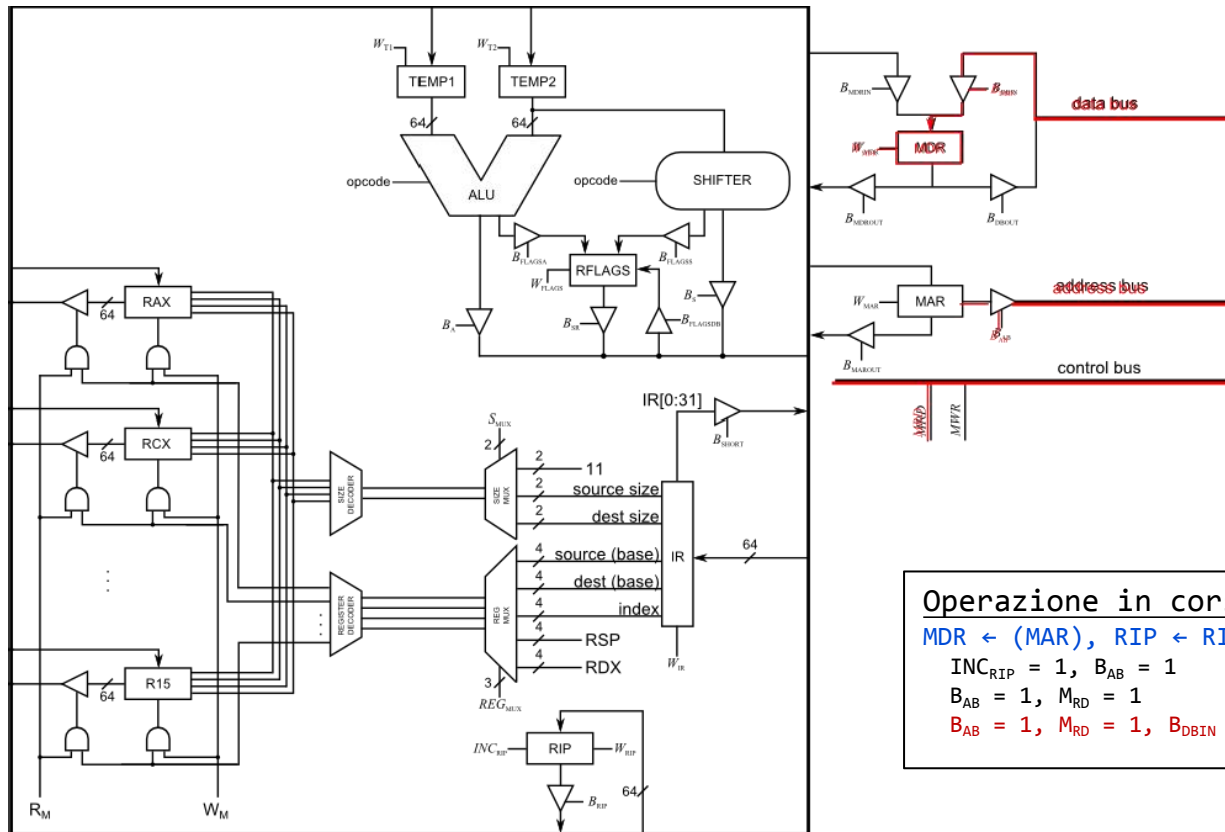
Operazione in corso:

$MDR \leftarrow (MAR), RIP \leftarrow RIP + 8:$

$INC_{RIP} = 1, B_{AB} = 1$

$B_{AB} = 1, M_{RD} = 1$

# La fase di fetch



Operazione in corso:

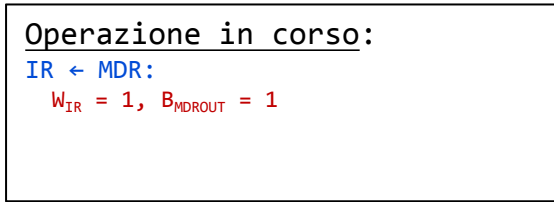
$MDR \leftarrow (MAR), RIP \leftarrow RIP + 8:$

$INC_{RIP} = 1, B_{AB} = 1$

$B_{AB} = 1, M_{RD} = 1$

$B_{AB} = 1, M_{RD} = 1, B_{DBIN} = 1, W_{MDR} = 1$

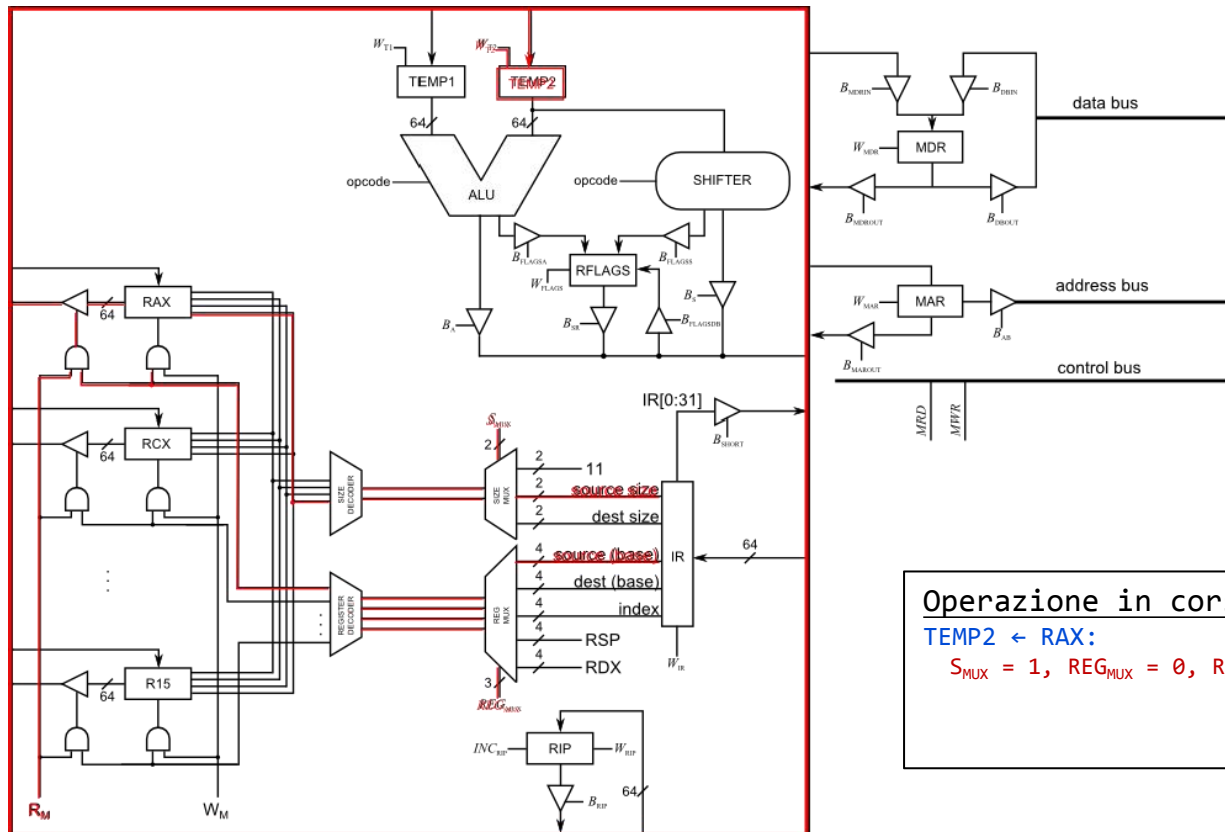
## 51



# Istruzioni di movimento dati

- Le microoperazioni associate al movimento dati dipendono dalla modalità di indirizzamento utilizzato
- Accedere in memoria utilizzando la modalità di indirizzamento dello z64 è un'attività costosa
- Tipicamente le istruzioni di movimento dati che accedono in memoria richiedono più cicli macchina
- `movq %rax, %rcx`:
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
  - $TEMP2 \leftarrow RAX$
  - $RCX \leftarrow TEMP2$

# Istruzioni di movimento dati

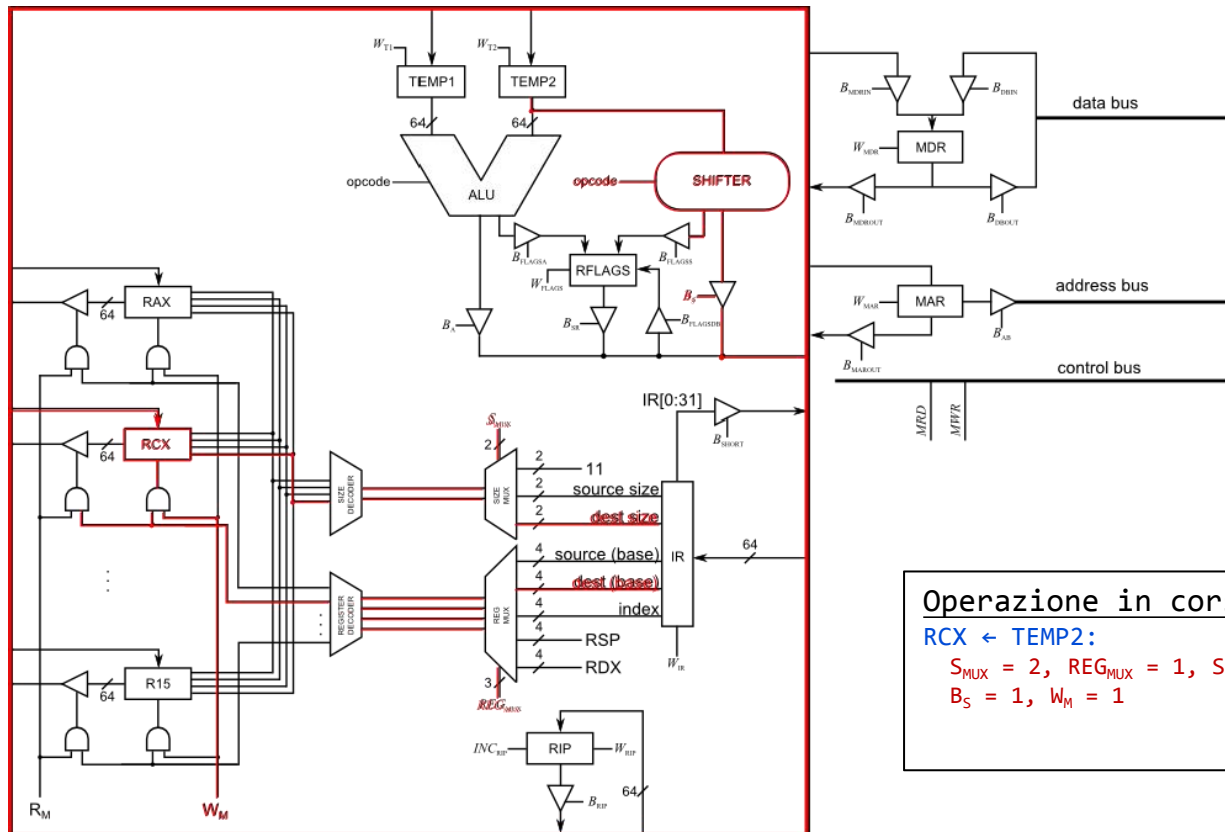


Operazione in corso:

$TEMP2 \leftarrow RAX$ :

$S_{MUX} = 1, REG_{MUX} = 0, R_M = 1, W_{T2} = 1$

# Istruzioni di movimento dati



Operazione in corso:

**RCX ← TEMP2:**

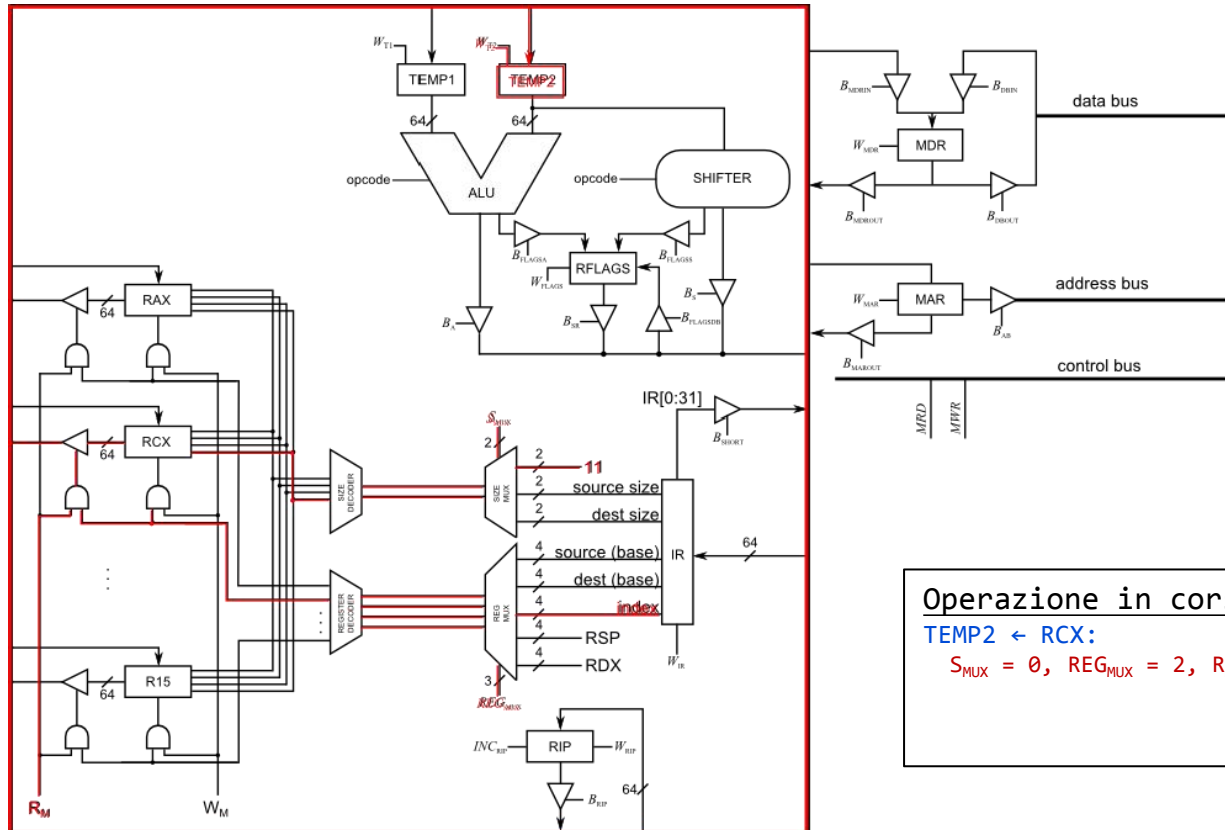
$S_{MUX} = 2$ ,  $REG_{MUX} = 1$ ,  $S_{opcode} = 000000$ ,

$B_S = 1$ ,  $W_M = 1$

# Istruzioni di movimento dati: full addressing

- `movq %rax, 0xaaaa(%rax, %rcx, 8):`
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
  - $TEMP2 \leftarrow RCX$
  - $TEMP1 \leftarrow \text{Shifter Out}[SHL, 000011]$
  - $TEMP2 \leftarrow RAX$
  - $MAR \leftarrow ALU\ OUT[ADD]$
  - $TEMP1 \leftarrow IR[0:31]$
  - $TEMP2 \leftarrow MAR$
  - $MAR \leftarrow ALU\ OUT[ADD]$
  - $MDR \leftarrow RAX$
  - $(MAR) \leftarrow MDR$

# Istruzioni di movimento dati: full addressing



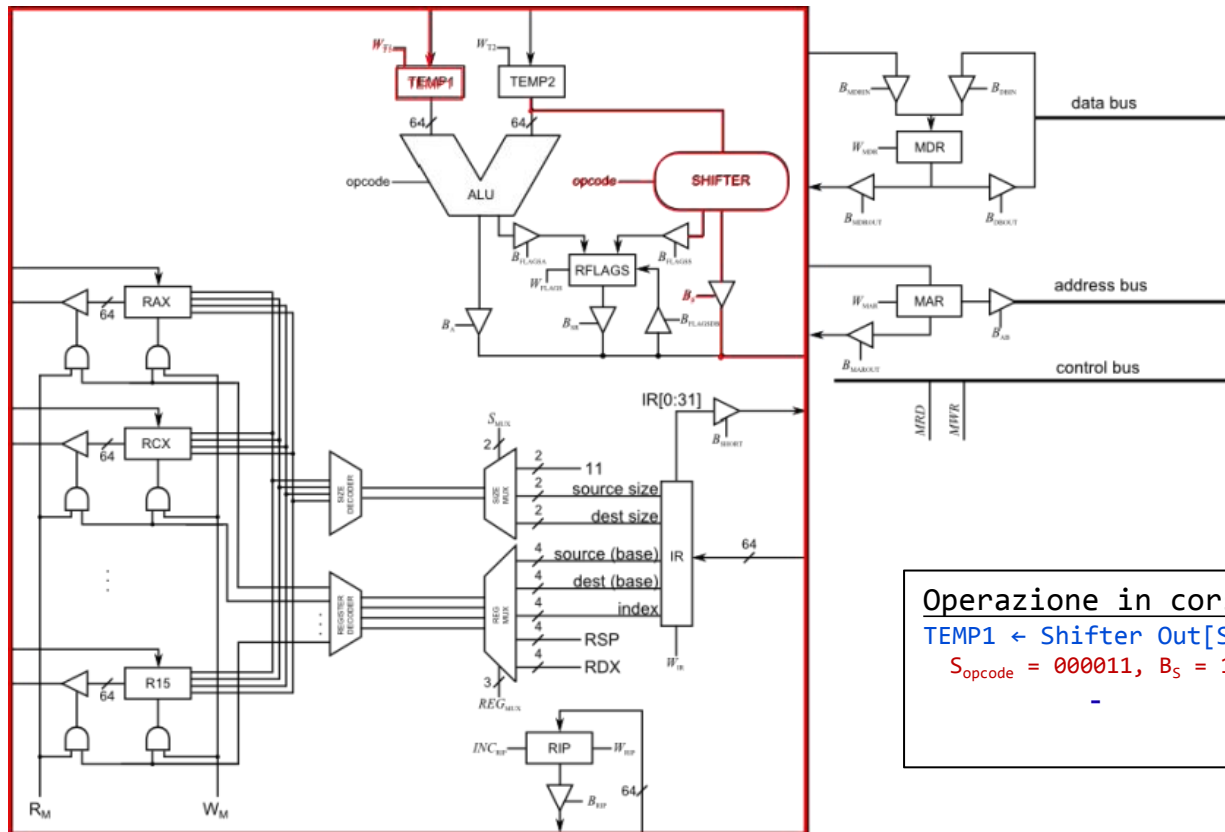
Operazione in corso:

$TEMP2 \leftarrow RCX$ :

$S_{MUX} = 0, REG_{MUX} = 2, R_M = 1, W_{T2} = 1$



# Istruzioni di movimento dati: full addressing



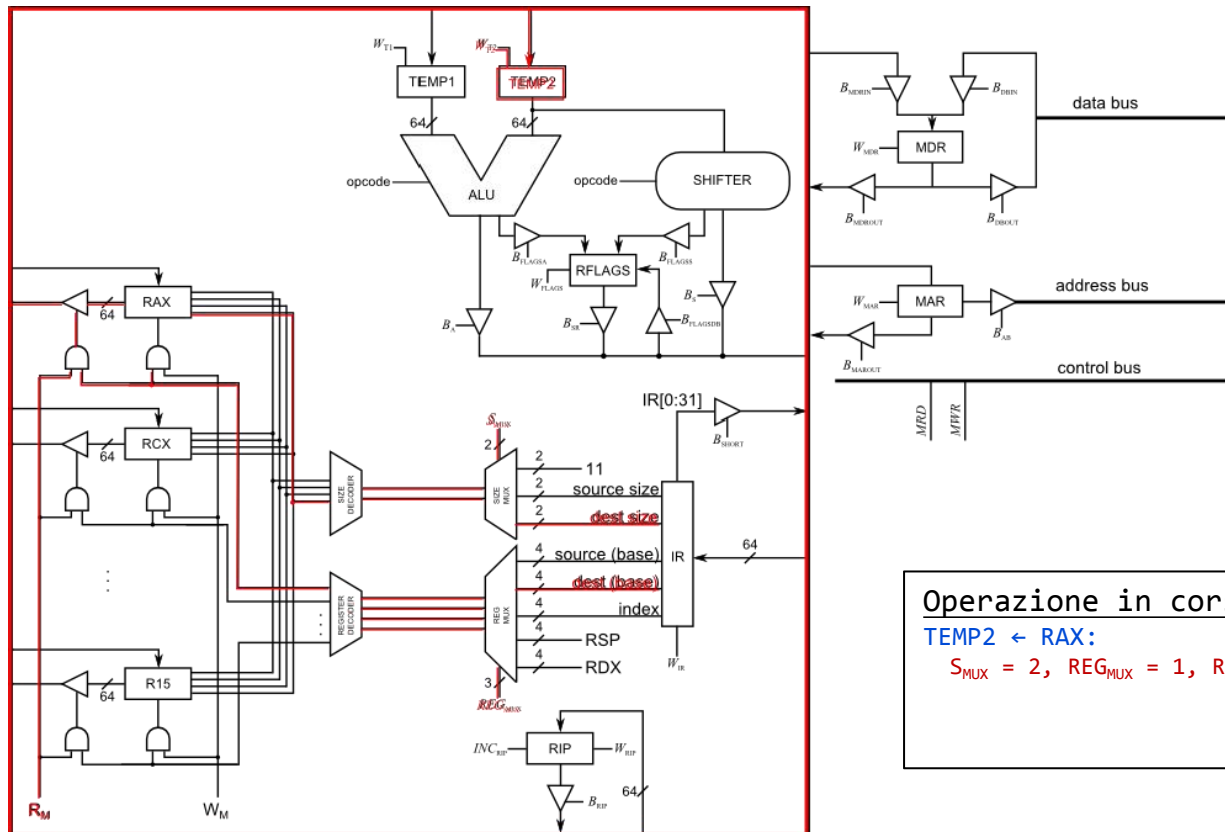
Operazione in corso:

$TEMP1 \leftarrow \text{Shifter Out}[\text{SHL}, 000011]:$

$S_{opcode} = 000011, B_S = 1, W_{T1} = 1$

-

# Istruzioni di movimento dati: full addressing

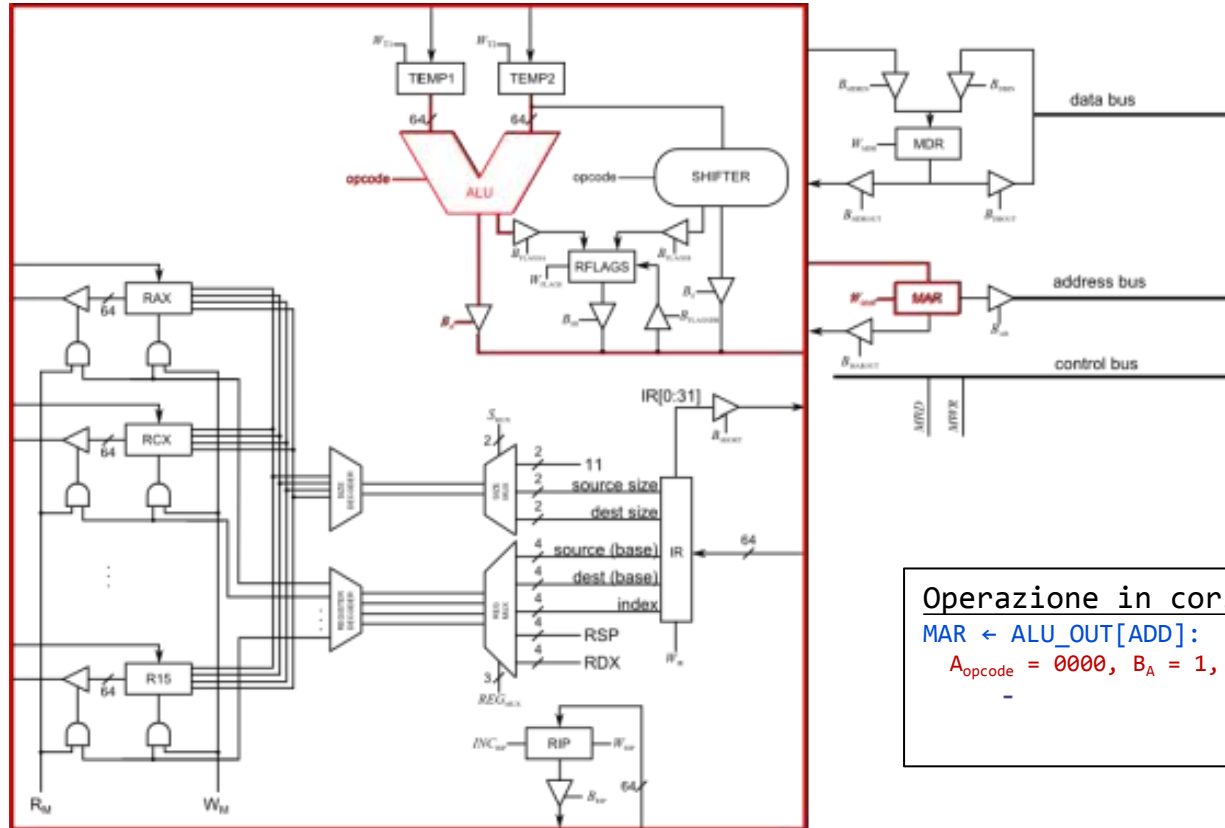


Operazione in corso:

$TEMP2 \leftarrow RAX$ :

$S_{MUX} = 2, REG_{MUX} = 1, R_M = 1, W_{T2} = 1$

# Istruzioni di movimento dati: full addressing



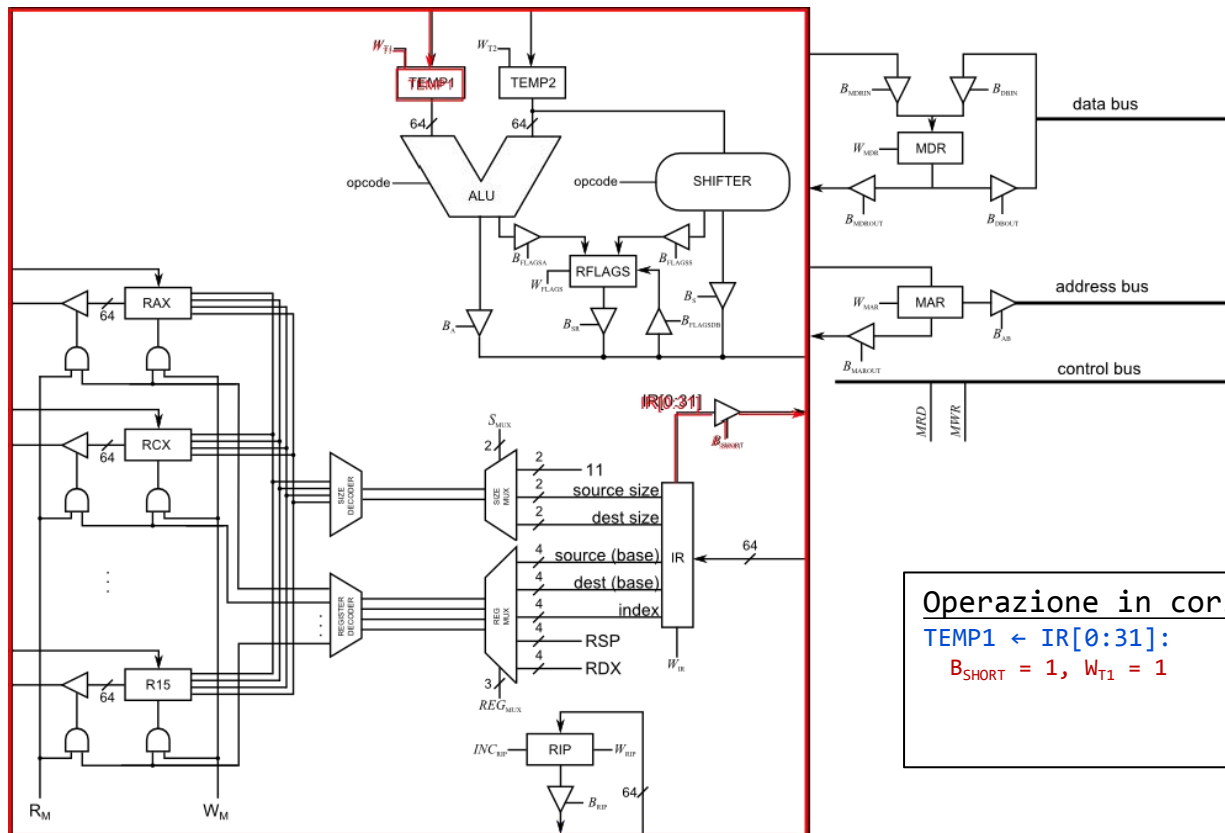
Operazione in corso:

$MAR \leftarrow ALU\_OUT[ADD]:$

$A_{opcode} = 0000, B_A = 1, W_{MAR} = 1$

-

# Istruzioni di movimento dati: full addressing

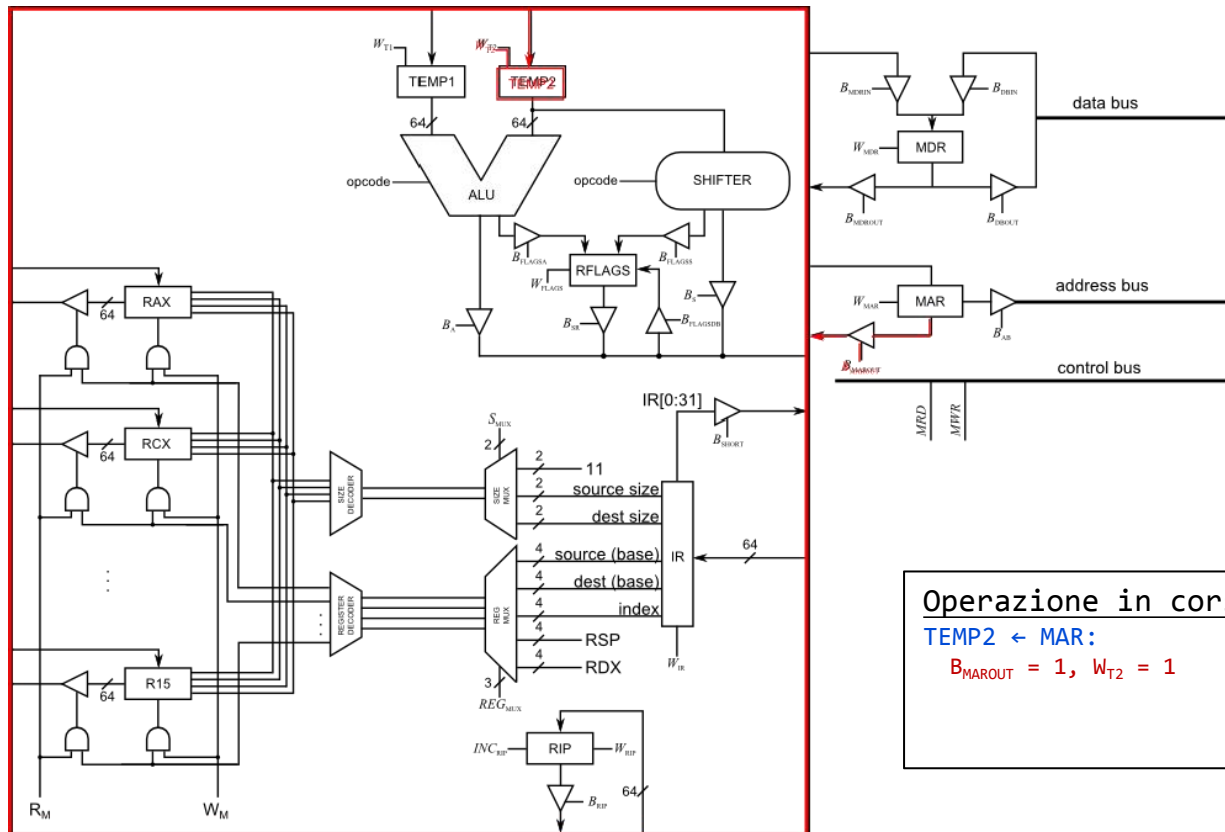


Operazione in corso:

$TEMP1 \leftarrow IR[0:31]$ :

$B_{SHORT} = 1$ ,  $W_{T1} = 1$

# Istruzioni di movimento dati: full addressing

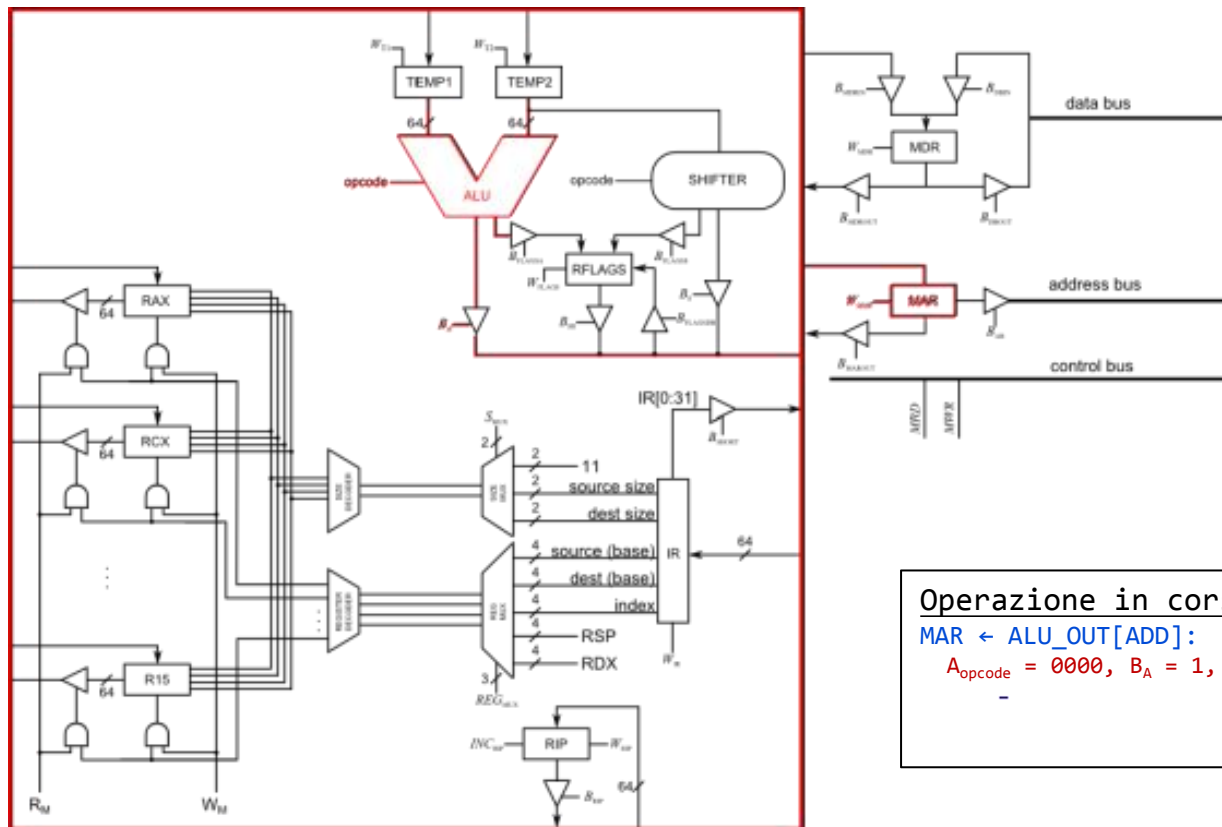


Operazione in corso:

$TEMP2 \leftarrow MAR$ :

$B_{MAROUT} = 1, W_{T2} = 1$

# Istruzioni di movimento dati: full addressing



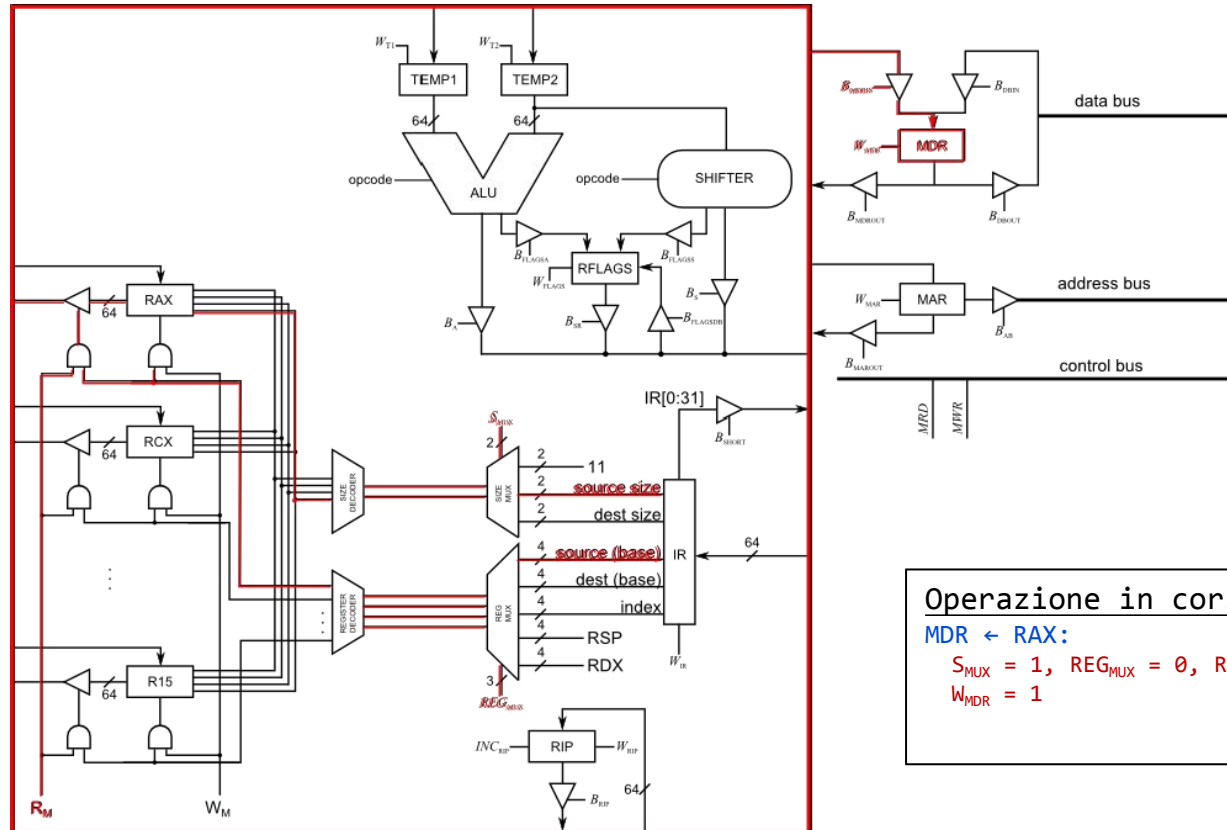
Operazione in corso:

$MAR \leftarrow ALU\_OUT[ADD]:$

$A_{opcode} = 0000, B_A = 1, W_{MAR} = 1$

-

# Istruzioni di movimento dati: full addressing



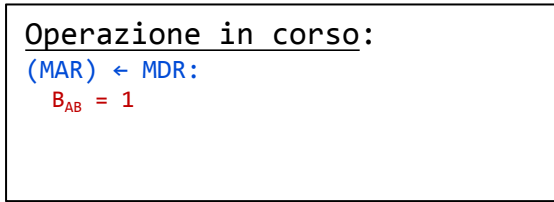
Operazione in corso:

**MDR ← RAX:**

$S_{MUX} = 1$ ,  $REG_{MUX} = 0$ ,  $R_M = 1$ ,  $B_{MDRIN} = 1$ ,

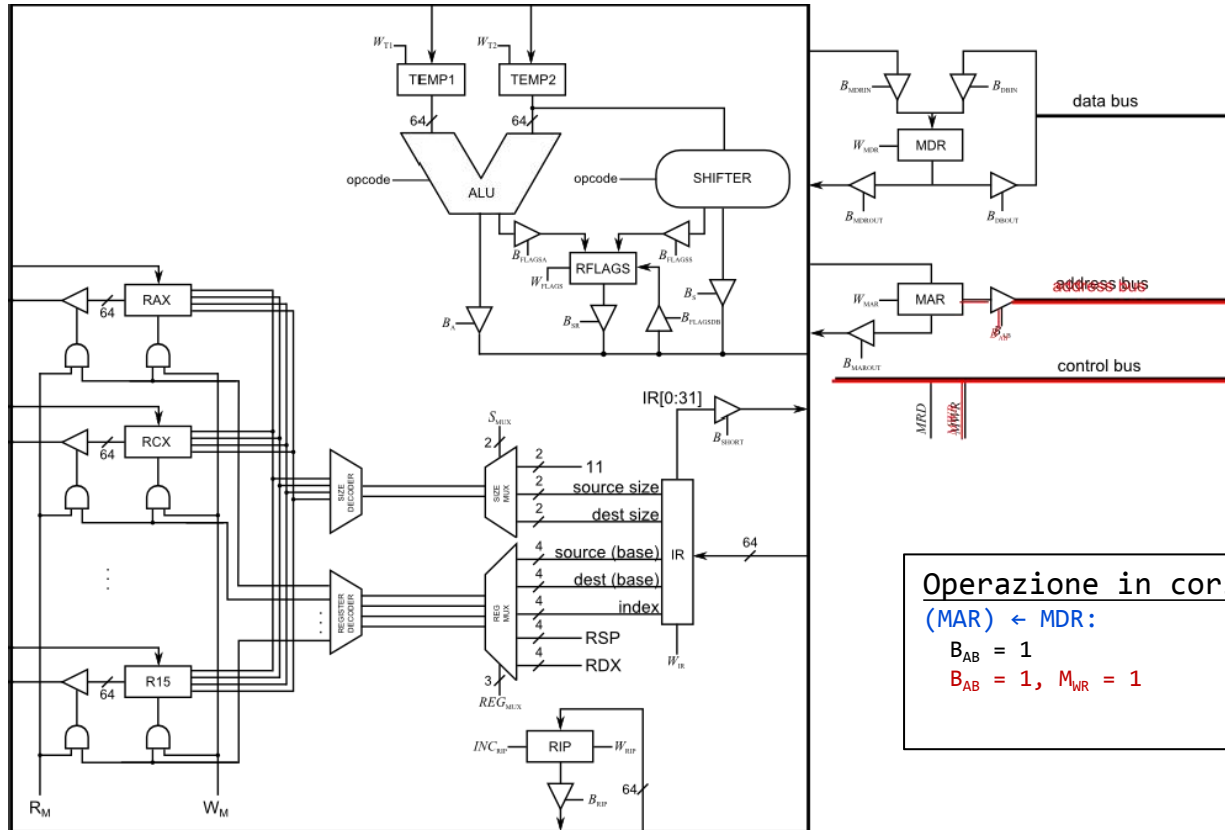
$W_{MDR} = 1$

## 64





# La fase di fetch



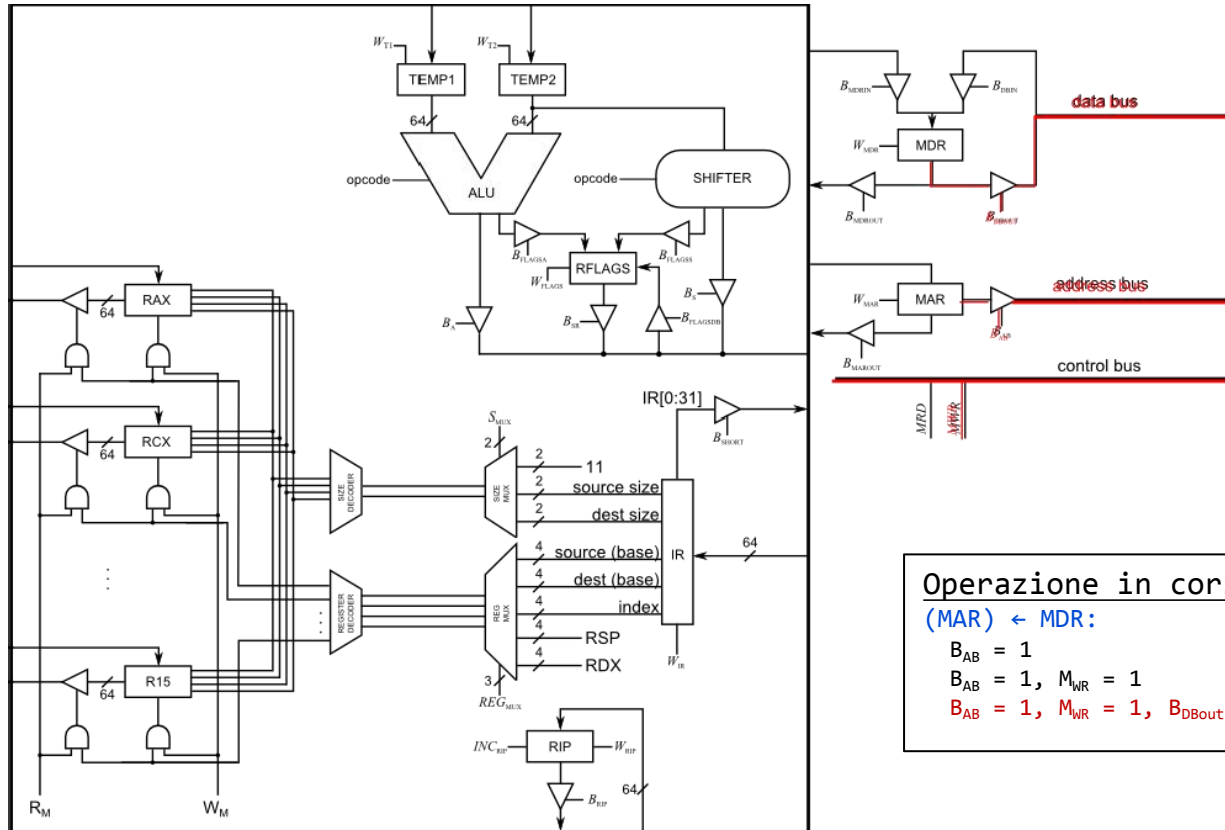
Operazione in corso:

$(MAR) \leftarrow MDR:$

$B_{AB} = 1$

$B_{AB} = 1, M_{WR} = 1$

# La fase di fetch



Operazione in corso:

$(MAR) \leftarrow MDR:$

$B_{AB} = 1$

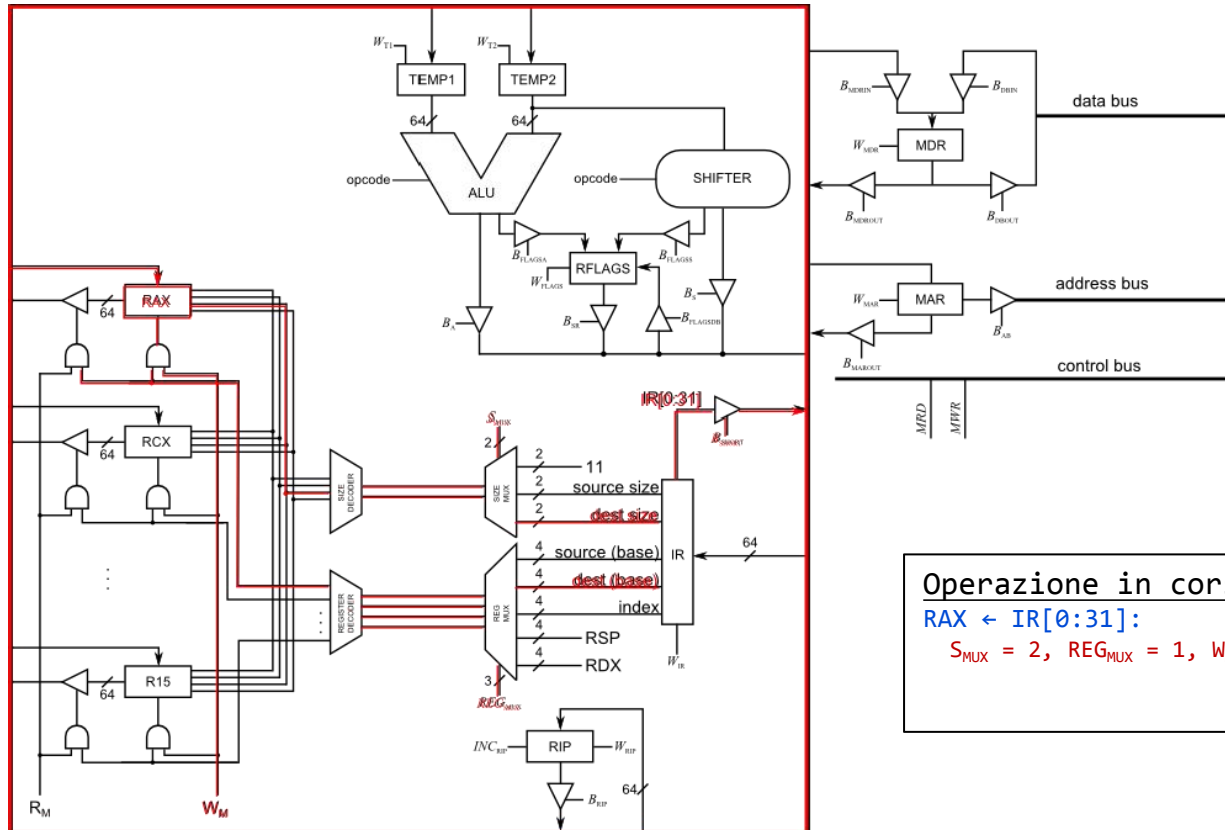
$B_{AB} = 1, M_{WR} = 1$

$B_{AB} = 1, M_{WR} = 1, B_{DBout} = 1$

# Istruzioni di movimento dati: immediati “piccoli”

- `movl $0xaaaa, %eax:`
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
  - $EAX \leftarrow IR[0:31]$

# Istruzioni di movimento dati: immediati “piccoli”



Operazione in corso:

$RAX \leftarrow IR[0:31]:$

$S_{MUX} = 2, REG_{MUX} = 1, W_M = 1, B_{SHORT} = 1$

# Istruzioni di movimento dati: immediati “grandi”

- Il microprogramma di fetch che abbiamo implementato legge dalla memoria soltanto 64 bit per volta
- Nel caso di immediati grandi, la costante si trova nei 64 bit immediatamente successivi
- È necessario un secondo accesso a memoria per prelevare la costante
- `movq $0xaaaa, %rax:`
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $RAX \leftarrow MDR$

# Unifichiamo le modalità di indirizzamento

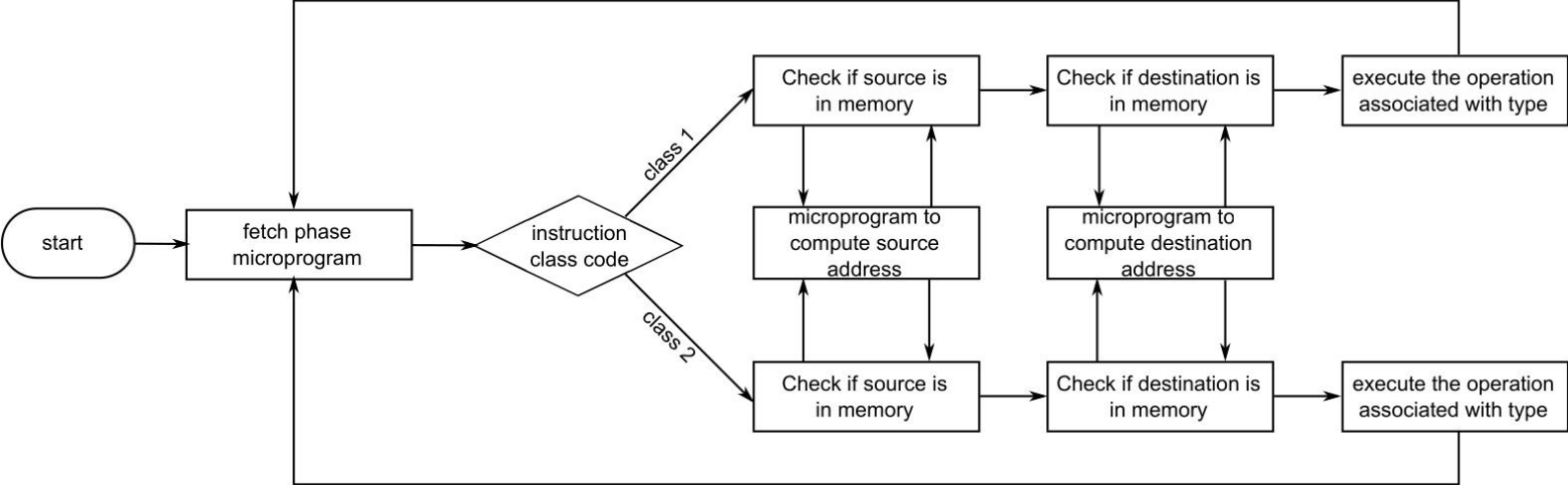
- In funzione del tipo degli operandi, un'istruzione di movimento dati o logico/aritmetica potrebbe dover accedere in memoria
  - L'accesso in memoria è parametrico:  $\text{base} + \text{indice} \cdot \text{scala} + \text{spiazzamento}$
- Avere più copie identiche delle microoperazioni per calcolare l'indirizzo effettivo nei microprogrammi di classi diverse non è efficiente
  - Viene sprecato molto spazio nella ROM per memorizzare le stesse microoperazioni
- Una modifica all'organizzazione della CU permette di “chiamare” sottoprogrammi cablati nel microcodice
  - Occorre implementare una microoperazione di “call” che aggiorna il “microprogram counter”
- Tutte le istruzioni che devono calcolare un indirizzo di memoria possono chiamare quel sottoprogramma

# Sottoprogramma firmware per calcolare gli indirizzi

- Nell'IR sono presenti dei bit che indicano quali componenti della modalità di indirizzamento sono utilizzate
- Questi bit possono essere dati in input alla CU, per determinare quali

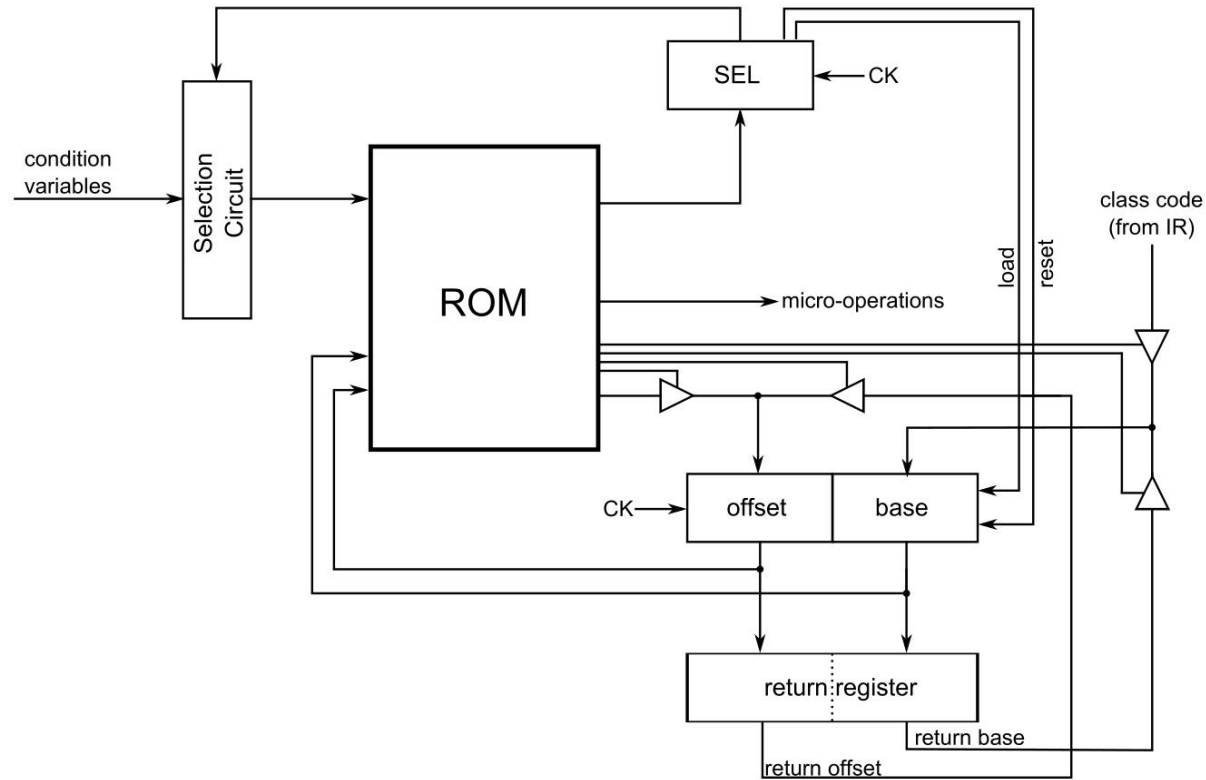
```
1  if D == 1 and Bp == 0 and Ip == 0
2      MAR ← IR[0:31]
3  else if D == 0 and Bp == 1 and Ip == 0
4      MAR ← B
5  else if Ip == 1
6      TEMP2 ← I
7      MAR ← SHIFTER_OUT[SHL, T]
8      TEMP1 ← MAR
9      if D == 1
10         TEMP2 ← IR[0:31]
11         MAR ← ALU_OUT[ADD]
12         TEMP1 ← MAR
13     endif
14     if Bp == 1
15         TEMP2 ← B
16         MAR ← ALU_OUT[ADD]
17     endif
18 endif
```

# Chiamata a sottoprogramma firmware





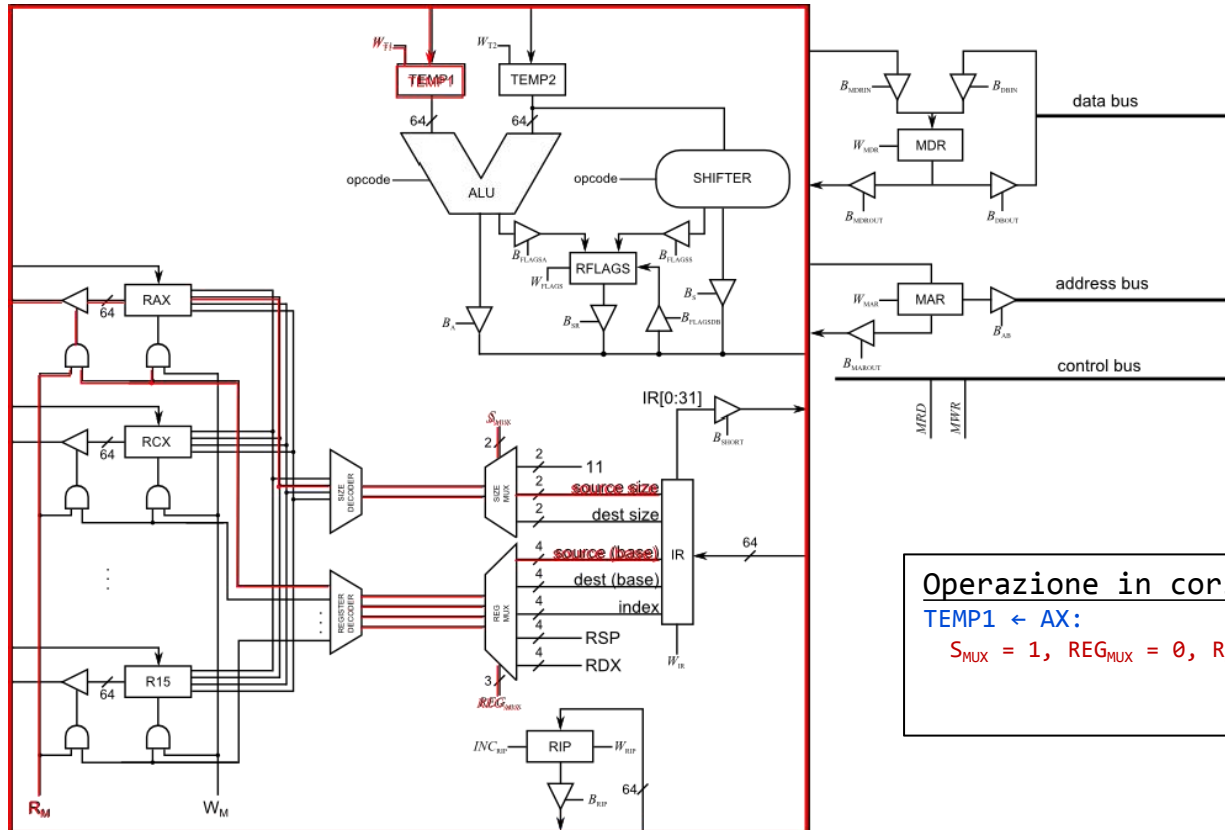
# Schema della CU modificata



# Istruzioni aritmetiche e logiche

- L'esecuzione di una determinata operazione aritmetica o logica dipende dall'opcode passato alla ALU
- `addw %ax, %cx:`
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
  - $TEMP1 \leftarrow AX$
  - $TEMP2 \leftarrow CX$
  - $CX \leftarrow ALU\ OUT[ADD]$

# Istruzioni aritmetiche e logiche

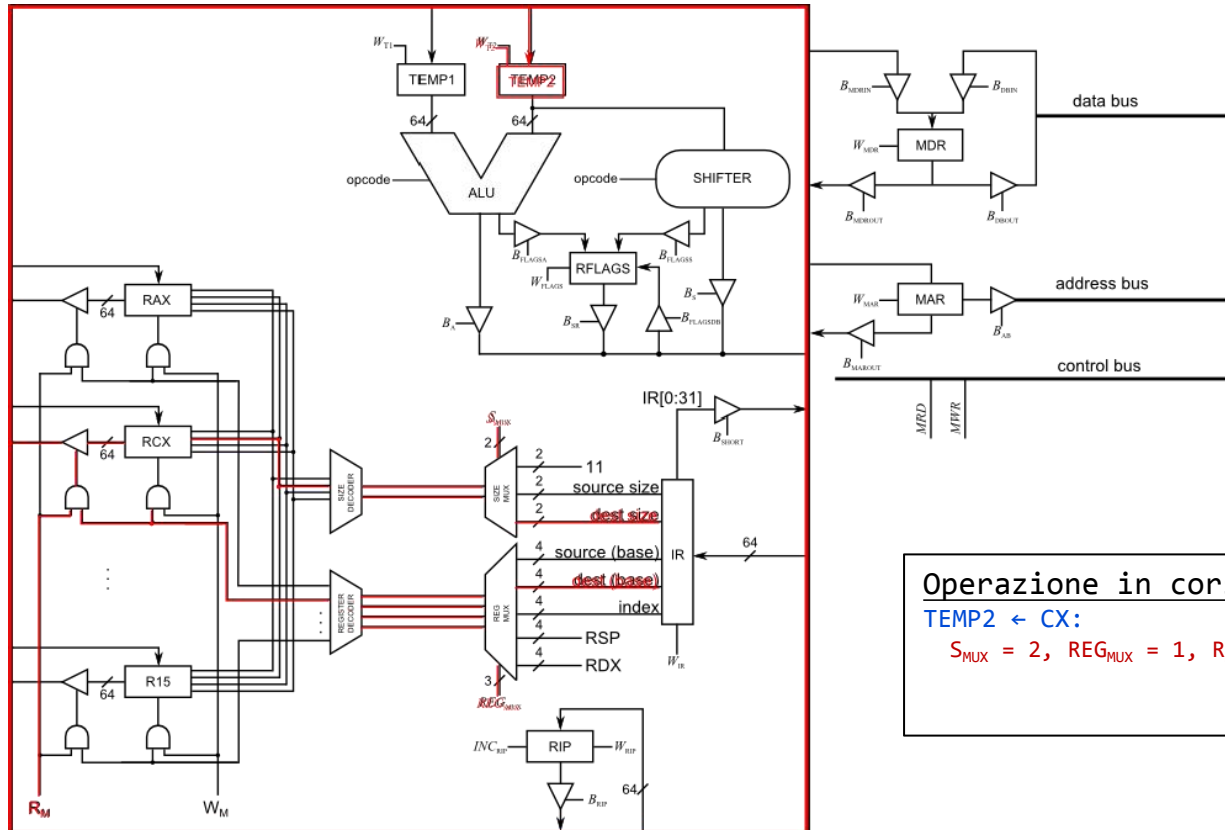


Operazione in corso:

**TEMP1  $\leftarrow$  AX:**

$S_{MUX} = 1, REG_{MUX} = 0, R_M = 1, W_{T1} = 1$

# Istruzioni aritmetiche e logiche

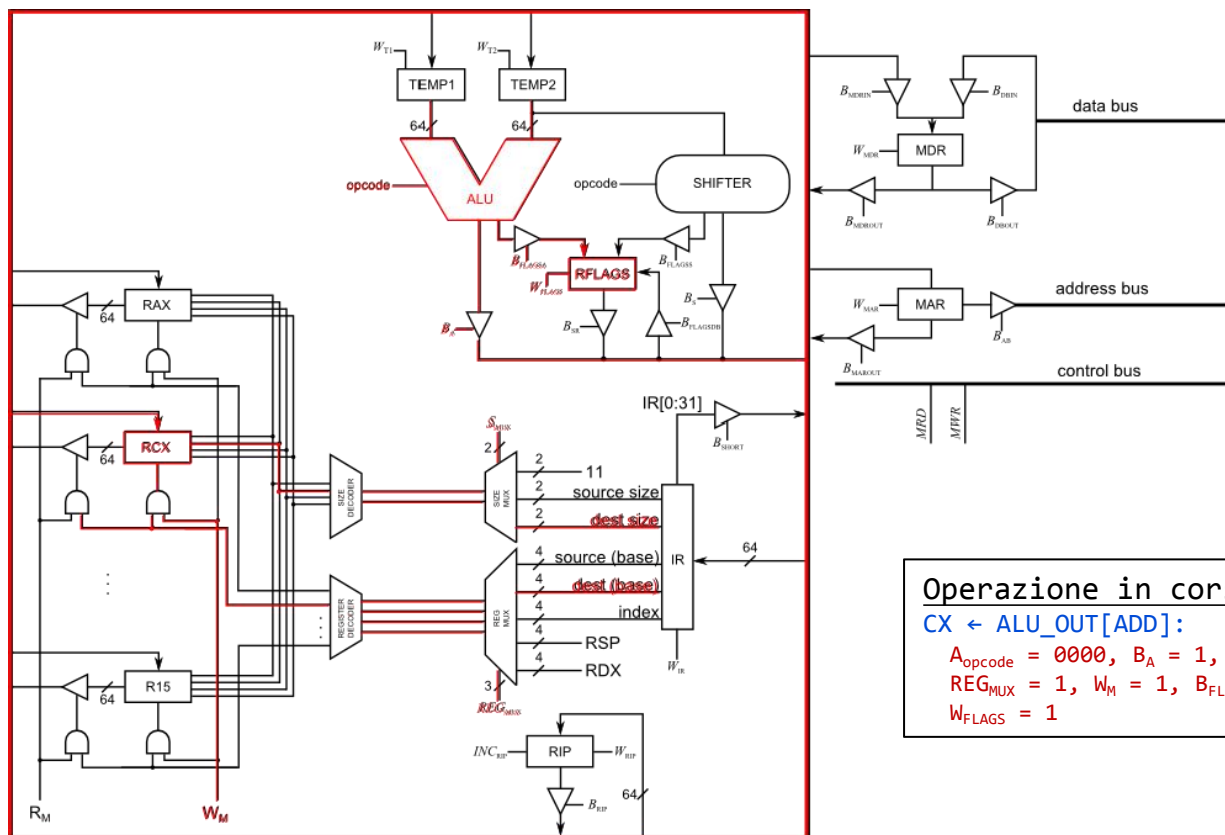


Operazione in corso:

$TEMP2 \leftarrow CX$ :

$S_{MUX} = 2, REG_{MUX} = 1, R_M = 1, W_{T2} = 1$

# Istruzioni aritmetiche e logiche



Operazione in corso:

$CX \leftarrow ALU\_OUT[ADD]:$

$A_{opcode} = 0000, B_A = 1, S_{MUX} = 2,$

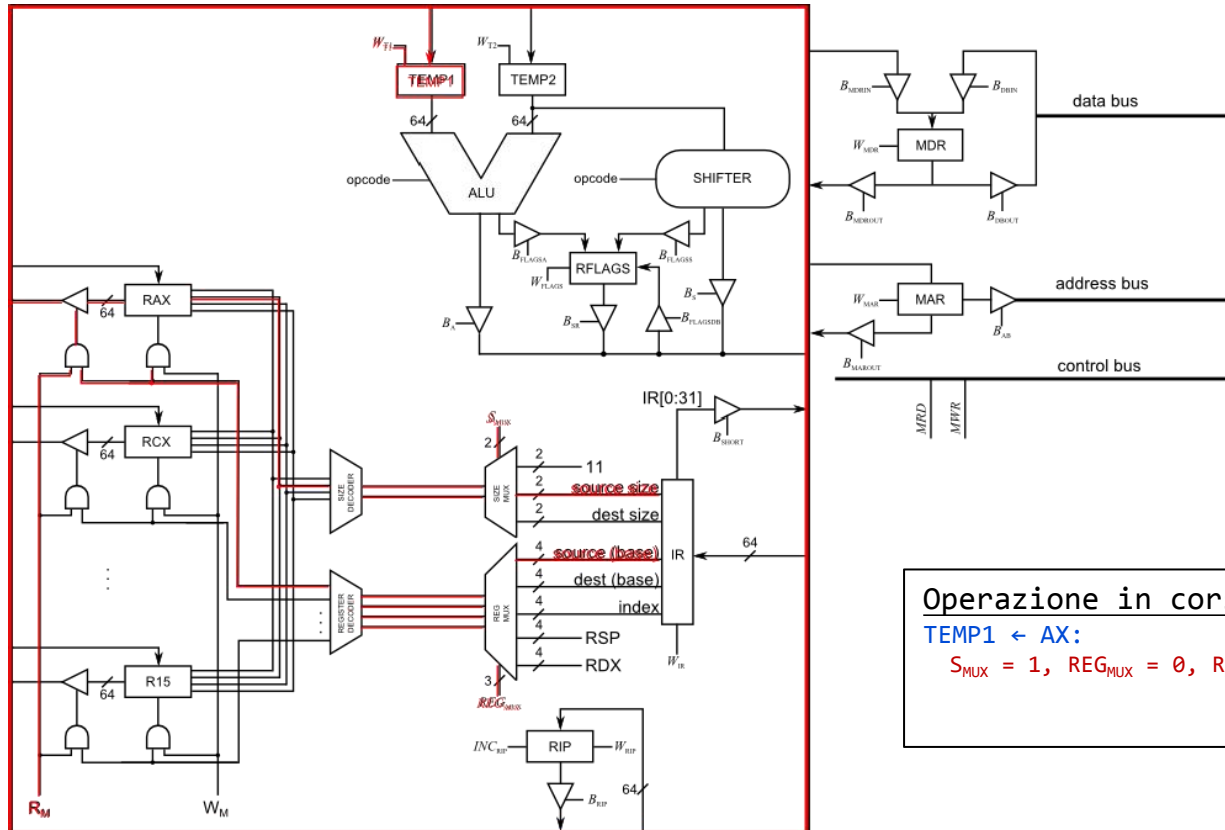
$REG_{MUX} = 1, W_M = 1, B_{FLAGSA} = 1,$

$W_{FLAGSA} = 1$

# Istruzioni aritmetiche e logiche

- Il microprogramma per le istruzioni logico/aritmetico è sostanzialmente lo stesso
- Cambia l'opcode da inviare alla ALU
- Si può utilizzare il tipo dell'istruzione (prelevato da IR) come opcode alla ALU ed eseguire sempre lo stesso microprogramma
- `andw %ax, %cx:`
  - $MAR \leftarrow RIP$
  - $MDR \leftarrow (MAR); RIP \leftarrow RIP + 8$
  - $IR \leftarrow MDR$
  - $TEMP1 \leftarrow AX$
  - $TEMP2 \leftarrow CX$
  - $CX \leftarrow ALU\ OUT[AND]$

# Istruzioni aritmetiche e logiche

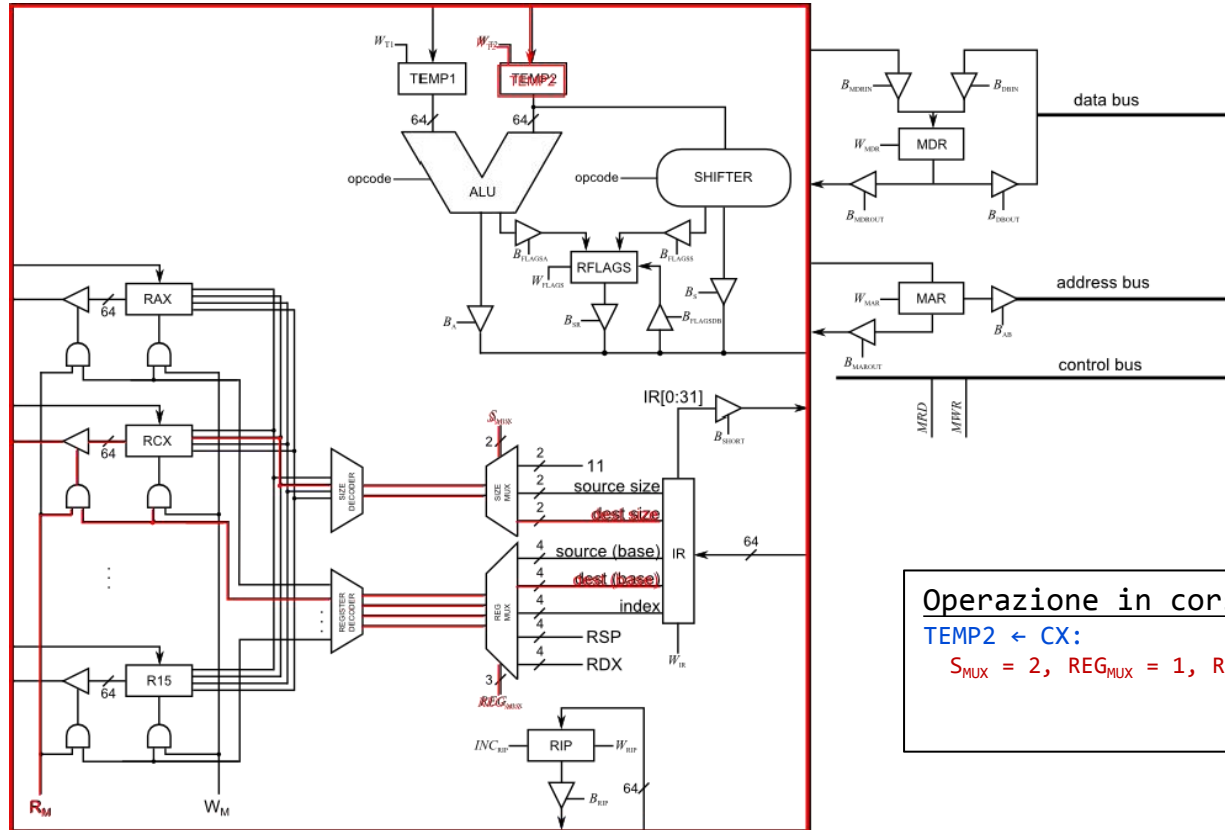


Operazione in corso:

**TEMP1  $\leftarrow$  AX:**

$S_{MUX} = 1, REG_{MUX} = 0, R_M = 1, W_{T1} = 1$

# Istruzioni aritmetiche e logiche



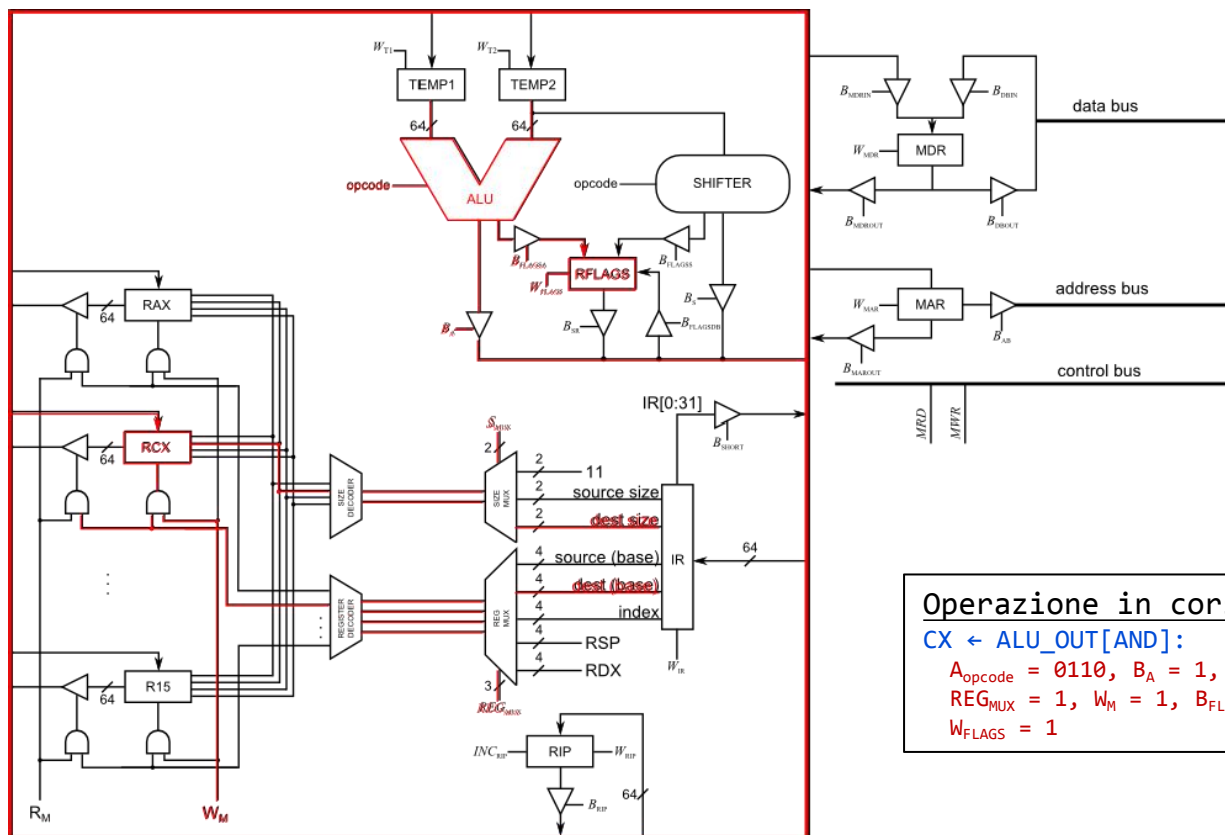
Operazione in corso:

$TEMP2 \leftarrow CX$ :

$S_{MUX} = 2, REG_{MUX} = 1, R_M = 1, W_{T2} = 1$



# Istruzioni aritmetiche e logiche



Operazione in corso:

$CX \leftarrow ALU\_OUT[AND]:$

$A_{opcode} = 0110, B_A = 1, S_{MUX} = 2$

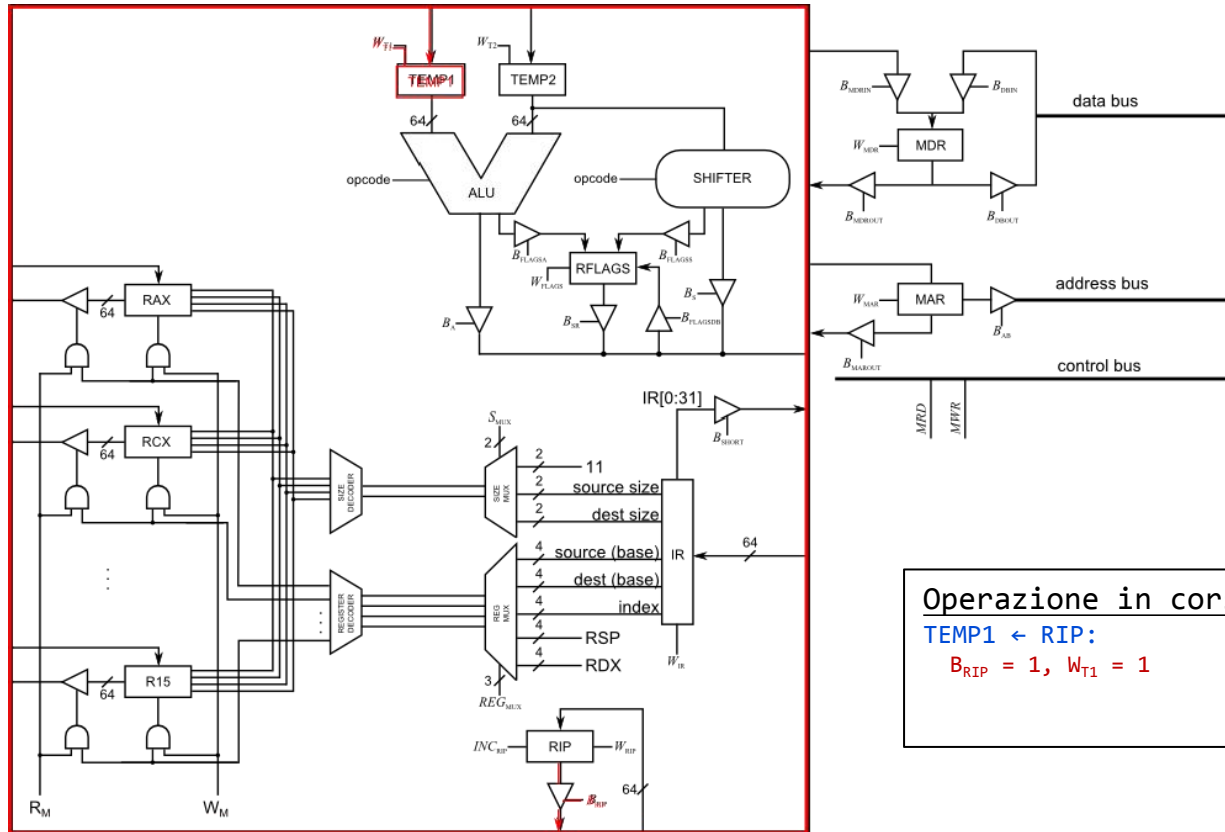
$REG_{MUX} = 1, W_M = 1, B_{FLAGS} = 1,$

$W_{FLAGS} = 1$

# Istruzioni di salto condizionale

- Nel caso di salto condizionale, il salto viene effettuato solamente se una condizione è verificata
- Il bit di FLAGS di interesse è dato in input alla CU
- Se la condizione non è verificata, non si esegue l'aggiornamento di RIP
  - È sufficiente introdurre una microoperazione di “jmp”
- `jz displacement:`
  - `MAR ← RIP`
  - `MDR ← (MAR); RIP ← RIP + 8`
  - `IR ← MDR`
  - `IF FLAGS[ZF] == 1 THEN`
    - `TEMP1 ← RIP`
    - `TEMP2 ← IR[0:31]`
    - `RIP ← ALU OUT[ADD]`
  - `ENDIF`

# Istruzioni di salto condizionale

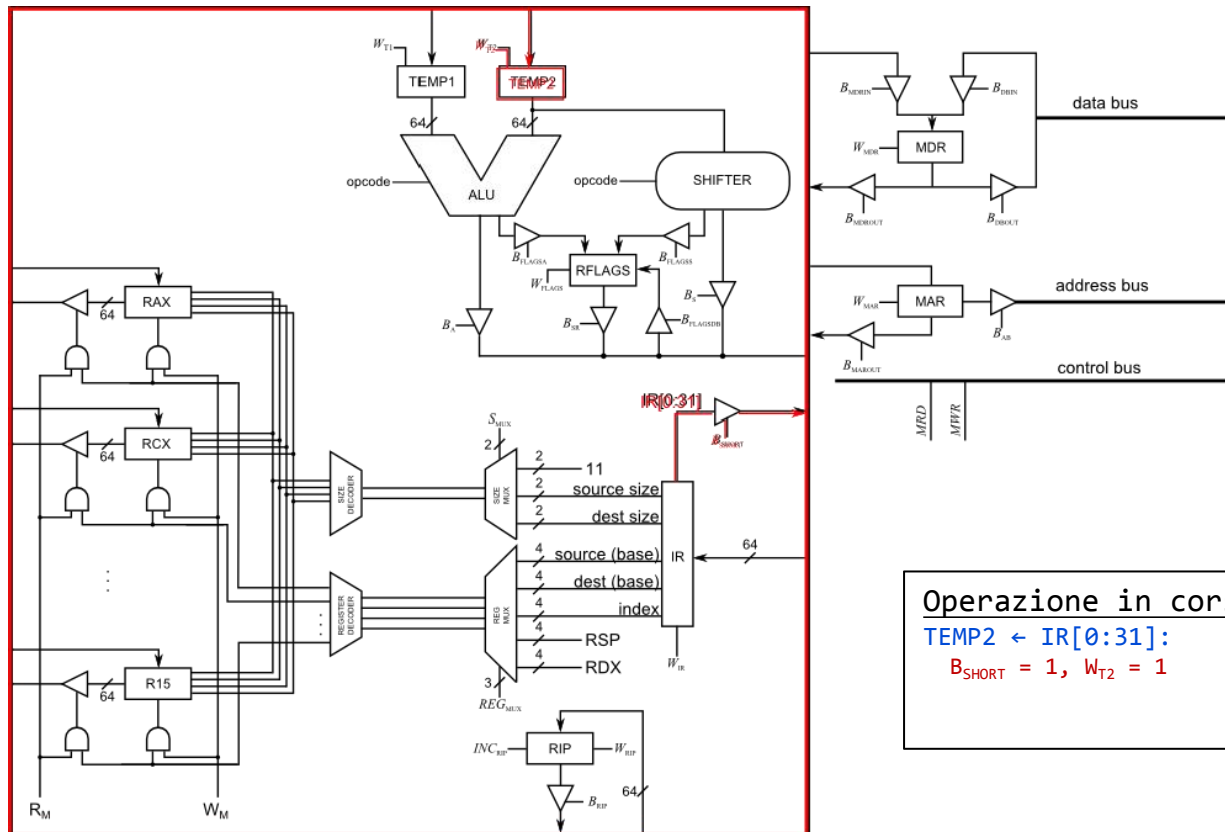


Operazione in corso:

$TEMP1 \leftarrow RIP:$

$B_{RIP} = 1, W_{T1} = 1$

# Istruzioni di salto condizionale

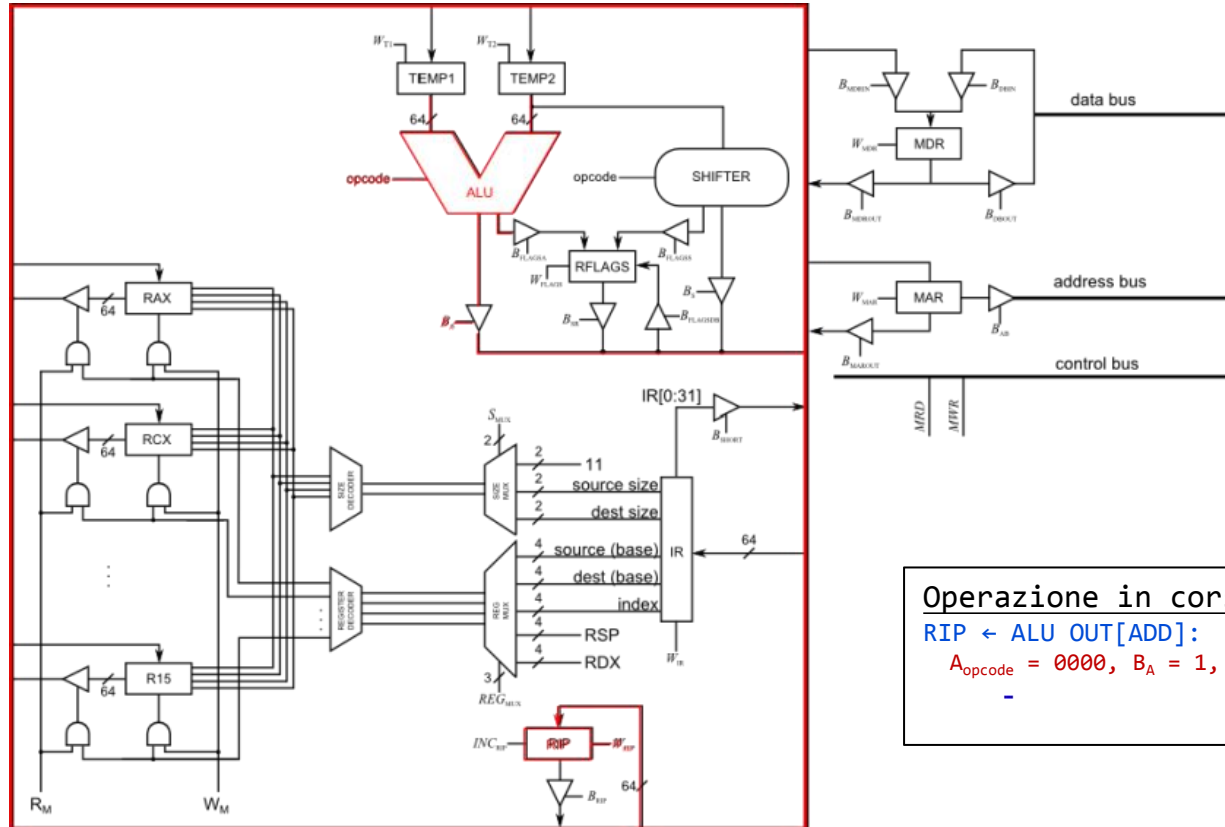


Operazione in corso:

$TEMP2 \leftarrow IR[0:31]:$

$B_{SHORT} = 1, W_{T2} = 1$

# Istruzioni di salto condizionale



Operazione in corso:

$RIP \leftarrow ALU\ OUT[ADD]:$

$A_{opcode} = 0000, B_A = 1, W_{RIP} = 1$

-