

Scheduling dinamico – algoritmo di tommasulo

Tommasulo Algorithm

- Another dynamic scheduling technique
- Overcomes problems with scoreboards
 - Renaming of registers
 - Avoids WAW and WAR hazards

Dynamic Scheduling

- Dynamic scheduling implies:
 - Out-of-order execution
 - Out-of-order completion
- Example 1:

fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14

 - fsub.d is not dependent, issue before fadd.d

=> Scoreboarding?

Dynamic Scheduling

- Example 2:

fdiv.d f0,f2,f4

fmul.d f6,f0,f8

fadd.d f0,f10,f14

- fadd.d is not dependent, but the antidependence makes it impossible to issue earlier without register renaming
- **Scoreboarding ?**

Register Renaming

- Example 3:

fdiv.d f0,f2,f4

fadd.d f6,f0,f8

fsd f6,0(x1)

fsub.d f8,f10,f14

fmul.d f6,f10,f8

Suppongo di disporre di un registro temporaneo T e attribuire il nome "T" al registro f6 in queste due istruzioni. Faccendo questo si annulla l'antidipendenza così posso eseguire la scrittura in f6 dell'ultima riga prima che le due istruzioni rigiallo legano gli operandi, perché non abbiamo più una antidipendenza sul registro f6.

antidependence

antidependence

l'autodipendenza sul registro f8 :
chiamalo f8 con "S".

- name dependence with f6

Register Renaming

- Example 3:

fdiv.d f0,f2,f4

fadd.d **S**,f0,f8

fsd **S**,0(x1)

fsub.d **T**,f10,f14

fmul.d f6,f10,**T**

f6 è stato sostituito da "S"

Salviamo i risultati non nel register file
ma all'interno delle functional unit states
(che con Tomasulo si chiamerà diverso).

- Now only RAW hazards remain, which can be strictly ordered

Register Renaming

- Tomasulo's Approach
 - Tracks when operands are available
 - Introduces register renaming in hardware → register renaming per le WAR e WAW, che subentra quando si fa commit out of order.
 - Minimizes WAW and WAR hazards

- Register renaming is provided by reservation stations (RS) → equivalent delle functional unit status
 - Contains:
 - The instruction
 - Buffered operand values (when available)
 - Reservation station number of instruction providing the operand values

NON BASTA IDENTIFICARE UN'UNITÀ FUNZIONALE PER SAPERE QUALE È L'OPERAZIONE CHE PROVVEDERÀ AL VALORE DI RISULTATO, MA DOBBIANO IDENTIFICARE UN'ISTRUZIONE SPECIFICA ASSOCIAVA ALLA RESERVATION STATION ASSOCIATA A QUESTA UNITÀ FUNZIONALE.

tiene traccia della disponibilità degli operandi e se sono disponibili propaga

functional unit status

• contiene l'istruzione da eseguire e gli operandi (non il puntatore al register file (es. valore contenuto in R7)) ma contiene direttamente il valore del register file del momento in cui è stato fatto l'issue dell'istruzione. Per gli operandi che erano non disponibili nel register file quando si è fatto issue contiene un puntatore (quindi un tag) all'unità funzionale che deve produrre quel valore più specificatamente il puntatore

è alla reservation station dell'unità funzionale che produrrà quel valore. Questo perché adesso le istruzioni sono nella reservation s. dell'unità funzionale che produrrà quel valore. Ci sono più reserv.s. per unità funzionale.

2° MODIFICA: nel register file. Quando dico che il RF può contenere un valore valido o non valido bisogna riconoscere la validità o meno, cioè se non è valido bisogna sapere qual è la reservation station che produce quel risultato.

Nello scoreboard queste info era nel register result status, che contiene per ciascun registro se il valore del register file era valido o qual è l'unità funzionale che deve produrlo

In Tomm. il register file ha 2 campi, uno è "valore", che è un campo numerico se è valido. Se non è valido, affianca un altro campo, "campo tag" che contiene un identificativo attraverso cui sappiamo qual è la reservation station che deve produrre il risultato oppure se il campo valore è valido e in quel caso ci sarà uno zero.

Architettura delle CPU con scheduling dinamico

Scheduling Dinamico:

- Attivazione sequenziale: **IN-ORDER DISPATCHING**
 - Esecuzione fuori ordine: **OUT-OF-ORDER EXECUTION**
 - Completamento fuori ordine: **OUT-OF-ORDER COMPLETION**
- **Obiettivo: stallare la pipeline solo in caso di alee strutturali e di controllo.**
- **Soluzione:**
- Ogni Unità di Esecuzione dispone di una coda di ingresso ove le istruzioni ad essa destinate vengono mantenute fino al completamento dell'esecuzione
 - L'esecuzione procede in parallelo su tutte le unità di esecuzione. Vengono eseguite le istruzioni che dispongono degli operandi, “girando attorno” alle istruzioni stallate a causa di alea di dato; dunque l'esecuzione è **fuori ordine, data driven**. Questa soluzione, oggi in voga, fu già proposta da Tomasulo nel 1967 per la Floating Point Unit dei calcolatori IBM 360/91 (Algoritmo di Tomasulo)

↑ FETCH E ISSUE
(CID)

↓ meno due prime

$$N_{\text{stalli}} = N_{\text{stallistrutturali}} + N_{\text{stallidicontrollo}}$$

Ogni functional unit dispone di una code di ingresso dove si parcheggiano le istruzioni in attesa degli operandi. Nello scoreb. non c'è una code per le istruzioni in ciascuna unità funz., ma una code complessiva e disponibile da 1 posto per ciascuna functional unit. (functional unit status register)

Adesso ogni functional unit dispone di una code che può contenere più di 1 istruzione. Questo è realizzato dalle reservation station.

Algoritmo di Tomasulo

- Ogni unità di esecuzione multicycle viene equipaggiata con “stazioni di prenotazione” (reservation stations) che ospitano sia l’istruzione in esecuzione sia istruzioni pronte per essere eseguite e in attesa dell’unità di esecuzione, sia istruzioni “sospese” in attesa di uno o due operandi
- Nello stadio “ID” : viene decodificata l’istruzione e allochiamo l’istruz. a una determinata ref. station della specifica unità fu. Se ci sono reser. st. disponibili si può sempre fare issue. Stalliamo solo se le res. station di una specifica unità fu. sono piene.
 - l’istruzione viene decodificata e inviata a una appropriata “reservation station” se disponibile; altrimenti si stalla (alea strutturale, mancanza di RS).
 - Alla “reservation station” vengono inviati anche gli operandi (coppia TAG, valore); se l’operando è disponibile il TAG vale zero; altrimenti il TAG è l’identificatore della reservation station che fornirà l’operando mancante (caso dell’alea di dato)
- Il risultato generato dall’unità “EX”, insieme al TAG della RS che identifica l’operazione eseguita viene messo a disposizione del register file, di tutte le unità di “EX” che lo richiedono, e dello stadio “M” (“forwarding generalizzato” attraverso il Common Data Bus). → la propagazione del risultato alle varie unità fu. avviene con l’indirizzo dell’operando associato al tag che indica dell’u.f. che ha prodotto il risultato e il risultato è inviato a tutte le reservation station in attesa del risultato.
- le alee “raw” sono dunque risolte con il forwarding dei risultati a tutte le unità che hanno bisogno di operandi (reservation stations e memoria)

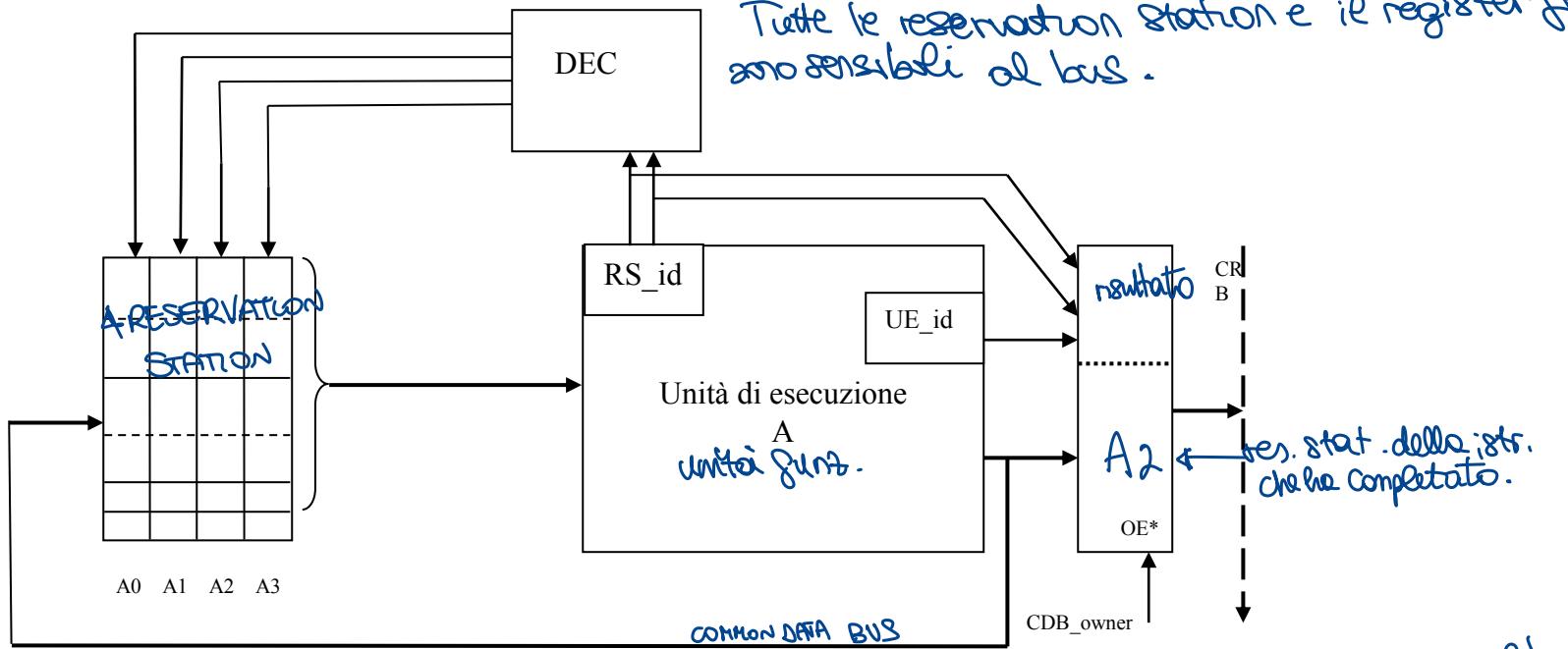
Venne propagata l’info di quale istruzione è terminata (identificativo delle res. st. a cui è associata l’istruzione, quindi il campo TAG, che consente di determinare univocamente quale istruz. ha terminato e il campo valore contiene il risultato associato all’operazione).

Mecanismo HW di broadcast che manda il risultato di un’operazione non solo al RF, ma anche a tutte le res. station in attesa di quel risultato.

l'ora sarà compito di res. stat e register file che possiedono quell'identificativo di trascrivere interamente il campo valore e aggiornare i propri campi all'interno del res. station del reg. file.

COMMON DATA BUS: c'è un bus comune dei risultati in cui tutte le functional unit interamente l'esecuzione scrivono nel bus il tag associato all'operazione quando terminano l'esecuzione appena terminata e il risultato.

Unità di esecuzione



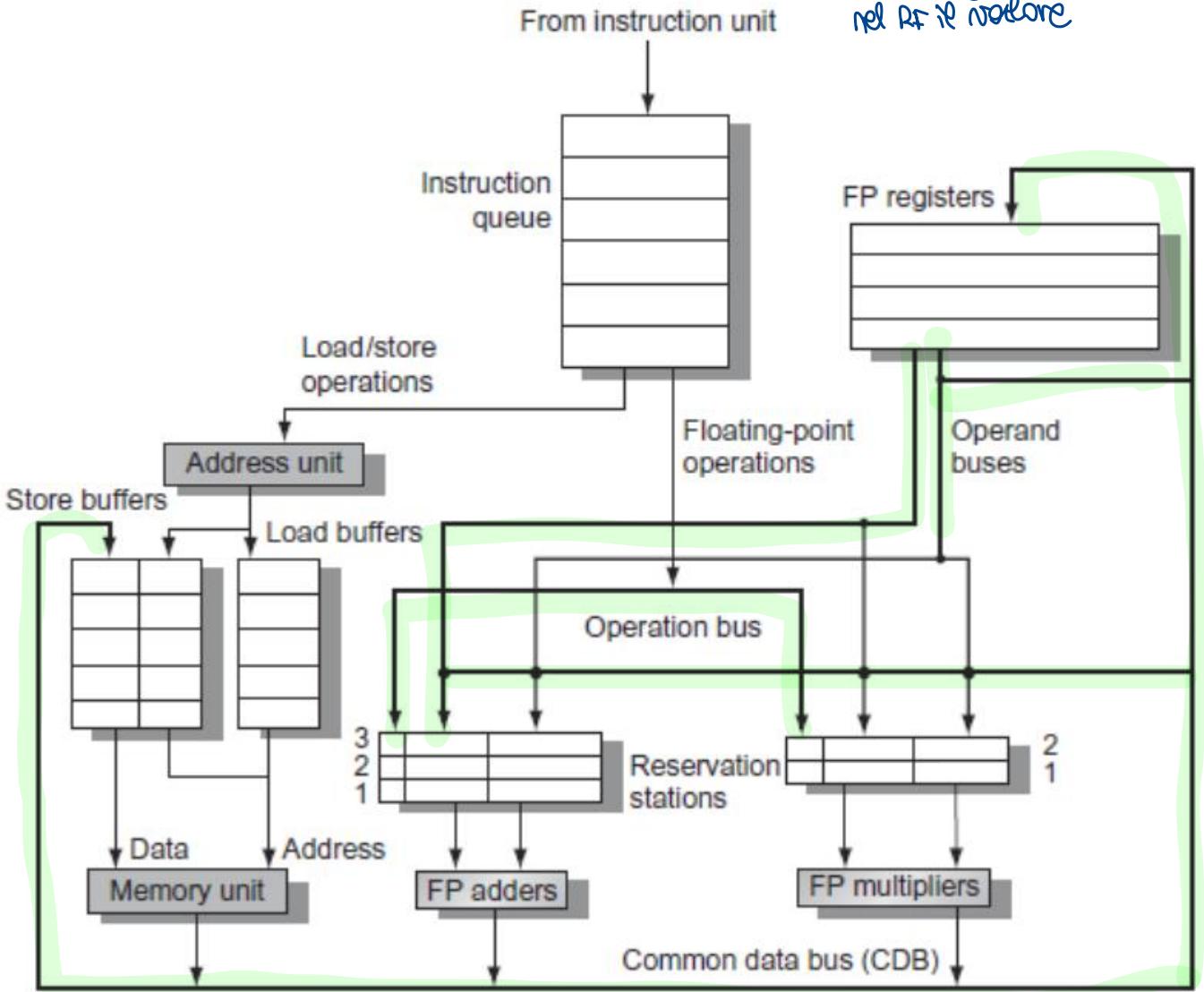
L'unità funzionale A ha 4 reservation station: A₀, A₁, A₂, A₃. Quando l'unità fu. ferma l'istruzione nella reservation station, l'informazione che ha terminato l'unità fu. A, che stava eseguendo l'istruzione nella reservation station dell'unità fu. con un determinato risultato.

Esempio di unità di esecuzione con quattro RS e registro di uscita

Questo viene comunicato sul common data bus e alle reser. stat. dell'unità fu. A, in modo che si scriva la res. stat. associata all'istruzione.

Tomasulo's Algorithm

In deade si copia il contenuto del register file nelle reservation station. Se l'operazione produce un registro destinazione, si mette nel RF il valore.



REGISTER FILE

Tag/Q

| | VALORE |
|-------|--------|
| f_1 | 0/RSS |
| f_2 | 5,5 |
| f_3 | 3,3 |
| | FADD-1 |
| | |

0: valore valido

RSS: indice della reservation station dell'unità funzionale es. FPA_0

(floating point add nello slot zero)

→ tag ≠ 0! se value non è quello che mi serve.

questi valori
i scrivo in issue.



| | | | | | | | |
|------------|---|------|--------|-----|---|-----|------------------|
| RSS.FADD-1 | 1 | fADD | 0 | 3,3 | 0 | 5,5 | |
| RSS.FADD-2 | 1 | - | - | - | - | - | saranno occupati |
| RSS.FADD-3 | 0 | fSUB | FADD-1 | XX | 0 | 7 | entry 3 libero |

↳ reservation station entry

$Q_j, V_j \}$ contengono il contenuto del
 $Q_k, V_k \}$ del register file per quell'istruzione.

Quando faccio issue di un'istruzione che finirà nella reservation station, se il reg. file contiene valori validi, alle decodifico i valori neiposso scritti, in questa tabella perché sono già disponibili.

fADD f₃, f₂, f₁



x permettere l'esecuzione di un'operazione da condizione e che i campi Q_j e Q_k siano a zero.
E che la functional unit non ha già in esecuzione un'altra istruzione che è nello res.st.

fSUB $\frac{RS1}{f_3}$, $\frac{RS2}{f_3, f_4}$



per la functional unit l'operando non viene dal registro 3 del register file, ma dal risultato che verrà prodotto dall' istruzione che si trova nelle reservation station 1 dell' unità FAD.

]
FETCH
ISSUE
EXECUTE
WRITE RESULT (o WRITE BACK)

Quando l'istruzione termina, al ciclo successivo viene comunicato sul common data bus l'informazione che è stato prodotto il risultato dell'informazione che è stato prodotto il risultato che ha come tag fadd-1 e come valore il valore numerico del risultato.

Res station e Register file hanno un ingresso che c'è sul common data bus che entra nei campi Q e dove c'è corrispondenza tra il tag sul data bus e un eventuale tag memorizzato nei campi Q, viene aggiornato il valore corrispondente.

Così implemento una logica di forwarding.

Così risolvo anche le altre WAW, il RF contiene il tag associato all'istruzione che produce l'ultimo dato.

Abbiamo fatto un RENAMING perché ad es si è sostituito al campo RF3 delle 1^a istruzione, la reservation station che produrrà quel risultato.: f3 → faddr-1

LOAD e STORE

faccio issue delle istruzioni di LOAD e LE STORE solo quando è noto l'indirizzo. Quindi se ho una dipendenza di dato per le istruzioni load e store, stallo in ISSUE.

Se ho noto l'indirizzo posso usare come campo di tag all'interno della reservation station quindi posso le strutture dati di load buffer e store buffer in cui vedo ad eseguire il tag associato all'indirizzo e il valore (in caso delle store) del dato letto dalla memoria.

→ Load e store buffer fanno la stessa cosa che fa la res.stat., ma per la load e store unit, però vengono riempite solo dopo aver fatto il calcolo dell'indirizzo.

Quindi il calcolo dell'indirizzo deve essere eseguito in ordine nell'ID.

Come 1^a coda deve calcolare l'indirizzo!!

Id X₁, X₂, 40

Indirizzo: contenuto di X₂ + 10

Id X₃, X₄, 20

= X₄ + 20
che è quello precedente. X₄ cui deve calcolare prima l'indirizzo e solo se ho una dipendenza di dato o nome.

se c'è una dipendenza di dato su X₄, posso stallare e mandare avanti l'istruzione prec, X₀ ma ho garantie che l'indirizzo che sarà risultato di X₄ + 20 sia + che quello precedente. X₄ cui deve calcolare prima l'indirizzo e solo se ho una dipendenza di dato o nome.

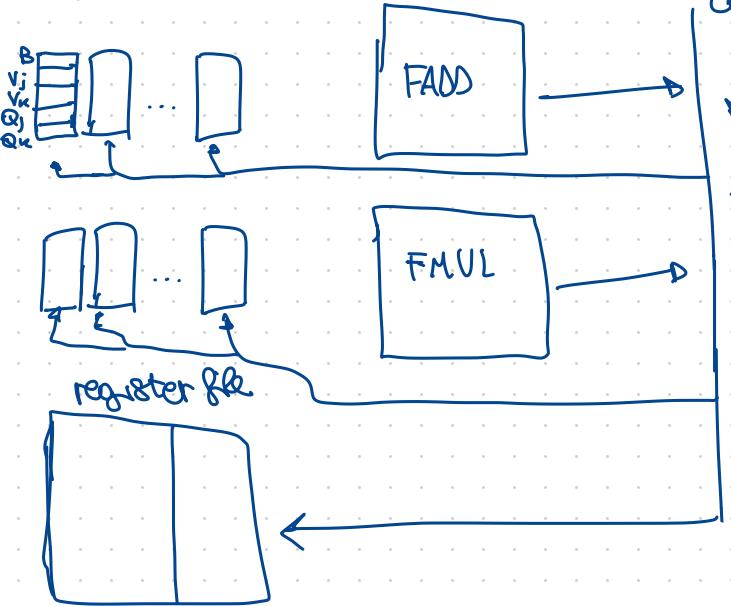
Tomasulo's Algorithm

- Three Steps:
 - Issue
 - Get next instruction from FIFO queue
 - If available RS, issue the instruction to the RS with operand values if available
 - If operand values not available, stall the instruction
 - Execute
 - When operand becomes available, store it in any reservation stations waiting for it
 - When all operands are ready, issue the instruction
 - Loads and store maintained in program order through effective address
 - No instruction allowed to initiate execution until all branches that proceed it in program order have completed
 - Write result
 - Write result on CDB into reservation stations and store buffers
 - (Stores must wait until address and value are received)

Se nel flusso di istruzioni ho un branch, perché non è risolto, le istruzioni che seguono dopo di cui ho fatto issue non valgono in esecuzione rimangono nelle reserv. st.

- Aree strutturali possibili nella logica di Tomasulo:**
- Reservation station tutta busy
 - Aree sul common data bus (se una unità su finito e' eseguito, i suoi risultati vengono scritti sul common data bus).
- Si può aumentare il parallelismo del c.d.b.

ESERCIZIO



COMMON DATA BUS → connesso anche alle singole resv. st.

+
di questo filo comunicano
il tag e il valore.

Il tag. contiene la functional unit (es. ADD)
e l'id delle resv. st. associate all'istruzione che
ha tr. tag.

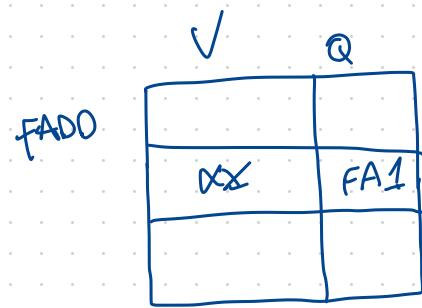
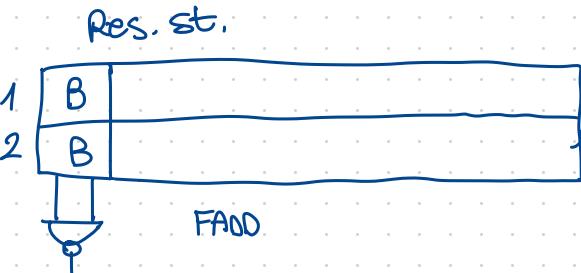
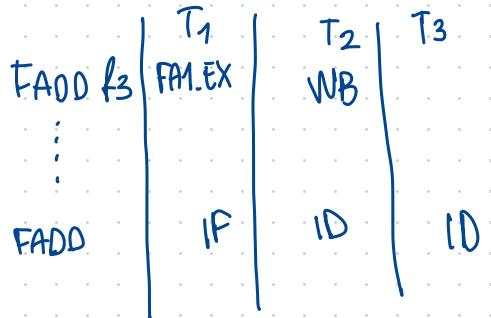
Il WB avviene nel primo semiperiodo del ciclo di ek.

Nel successivo ciclo di ck la res-statt. è libera.



Il controllo se la res.st. è occupata o no avviene nel 1° semiperiodo. Il WB avviene nel 2° semiperiodo; per cui se anche si libera in quel ciclo, la RS. risulta occupata perché la lettura del campo busy avviene nel primo semiperiodo.

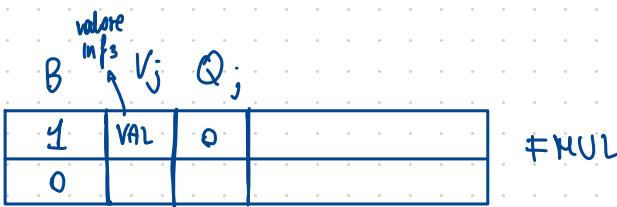
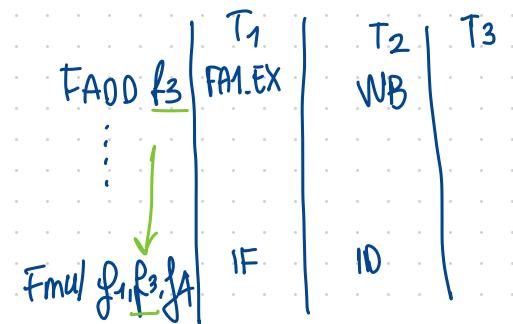
⇒ In questo caso si stalle e l'ID può essere fatto solo al ciclo successivo.



register file

✓ Q

| | |
|-------|---|
| | |
| value | 0 |
| | |



REGISTER FILE

**(accesso diretto in ID
e accesso per contenuto in WB)**

| Registro | TAG | VALORE |
|----------|-----|--------|
| R0 | | |
| R1 | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| R30 | | |
| R31 | | |

Struttura del register file

Ad ogni registro del register file sono associati 2 campi: **Qi** e **Vi**

- **Qi** contiene **l'identificatore del Load Buffer o della Reservation Station che produrrà il contenuto del registro [un TAG: (Unit_ID, RS_ID)]**
- **Vi** contiene il valore del registro se $Qi = 0$

Le coppie(Qi, Vi) su tutto il Register File costituiscono una tabella di coppie (nome, valore) realizzata con strutture completamente associative.

In ID, quando si deve leggere un operando sorgente Ri , si copia la coppia (Qi, Vi) negli omonimi campi (Q,V) della RS

In questo modo:

se $Qi = 0$, allora nel campo V della RS andrà il valore dell'operando pronto per essere utilizzato dall'unità di esecuzione

se $Qi \neq 0$, allora nel campo Q della RS andrà **il TAG (Unit_ID, RS_ID)** che identifica il Load Buffer o la Reservation Station che genererà il contenuto del registro (forwarding)

Qi (TAG) Vi(Contenuto del registro Ri)

| | | |
|--|--|------|
| | | R0 |
| | | |
| | | R31 |

Questo è il register file
La tabella contiene una riga per registro

Struttura delle RS

- Ad ogni unità che produce risultati (Load Buffer e Reservation Station) è associato un identificatore che viaggerà sul Result Bus insieme al risultato stesso **[TAG: (Unit_ID, RS_ID)]**
- Ogni Reservation Station ha sei campi:
 - BUSY: indica se la RS è libera o occupata
 - Q_j e Q_k : contengono l'identificatore del Load Buffer o della Reservation Station che produrrà l'operando sorgente (j : 1° operando, k : 2° operando)
 - V_j e V_k : contengono l'operando se $Q_j / Q_k = 0$
 - OP: operazione da eseguire (Es: FADD/FSUB...)

Busy OP Q_j (TAG) V_j (Operando) Q_k (TAG) V_k (Operando)

| | | | | | |
|--|--|--|--|--|--|
| | | | | | |
| | | | | | |
| | | | | | |

3 Reservation Stations, cioè stazioni di prenotazione (della unità di esecuzione)

Le coppie(Q_j, V_j) e (Q_k, V_k) su tutte le RS di un'unità di esecuzione costituiscono due tabelle di coppie (nome, valore) realizzate con strutture completamente associative.

Vincoli nella esecuzione di load e store

- Ordine nelle istruzioni di scrittura in memoria:
- Di solito si utilizza la politica di “strong write ordering”:
 - Le istruzioni di scrittura non sono mai eseguite fuori ordine tra loro
 - Una istruzione di scrittura non viene mai eseguita prima di una di lettura che precede la write nel codice (anche perché potrei non conoscere l’indirizzo di lettura)
- Le istruzioni di lettura possono essere eseguite fuori ordine tra loro
- Una istruzione di load può essere eseguita prima di una store se della store si conosce già l’indirizzo (altrimenti si rischierebbe di leggere un dato prima che questo sia scritto); nel Pentium le load non anticipavano le write, nel P6 questo vincolo può essere rimosso su certe aree di memoria

La politica di scrittura in ordine è necessaria sulla memoria condivisa dei sistemi multiprocessor

Sull’input output sia le letture che le scritture vanno assolutamente eseguite in ordine

Struttura dei Load Buffer

- Ad ogni Load Buffer sono associati 2 campi: BUSY, ADDRESS
- Il comportamento dei Load buffer è il seguente:
 - Il risultato generato dal Load Buffer insieme al tag che lo identifica, viene inviato al Common Data Bus, a disposizione del register file, di tutte le unità di “EX” che lo richiedono, e degli store buffers dello stadio “M” (“forwarding generalizzato”)
 - le alee “raw” sono dunque risolte con il forwarding del dato letto a tutte le unità che hanno bisogno di operandi

| Busy | Address | TAG di chi genererà l'address |
|------|---------|-------------------------------|
| | | |
| | | |
| | | |

Il tag del campo indirizzo consente alle load di essere eseguite fuori ordine: se non c'è l'indirizzo, la load si sospende nel load buffer; senza il tag, finchè l'indirizzo non è disponibile la pipeline stalla

Struttura degli store buffer

- Il comportamento degli store buffer dipende dalle politiche di scrittura; le operazioni di lettura scrittura su un dispositivo di i/o debbono essere ad esempio sempre eseguite in ordine.
- Ad ogni Store Buffer sono associati 4 campi:
BUSY, Qj, Vj, ADDRESS
 - Il comportamento degli Store Buffer si differenzia da quello delle RS perché le scritture vanno effettuate in ordine, per cui c'è una logica aggiuntiva ad hoc (in Intel si chiama "MOB - memory order buffer)

Example

| | | Instruction status | | | |
|-------------|-----------|--------------------|---------|--------------|---|
| Instruction | | Issue | Execute | Write result | |
| fld | f6,32(x2) | ✓ | ✓ | | ✓ |
| fld | f2,44(x3) | ✓ | ✓ | | |
| fmul.d | f0,f2,f4 | ✓ | | | |
| fsub.d | f8,f2,f6 | ✓ | | | |
| fdiv.d | f0,f0,f6 | ✓ | | | |
| fadd.d | f6,f8,f2 | ✓ | | | |

| Reservation stations | | | | | | | | |
|----------------------|------|------|----|--------------------|-------|-------|----------------|--|
| Name | Busy | Op | Vj | Vk | Qj | Qk | A | |
| Load1 | No | | | | | | | |
| Load2 | Yes | Load | | | | | 44 + Regs [x3] | |
| Add1 | Yes | SUB | | Mem[32 + Regs[x2]] | Load2 | | | |
| Add2 | Yes | ADD | | | Add1 | Load2 | | |
| Add3 | No | | | | | | | |
| Mult1 | Yes | MUL | | Regs[f4] | Load2 | | | |
| Mult2 | Yes | DIV | | Mem[32 + Regs[x2]] | Mult1 | | | |

| Register status | | | | | | | | | |
|-----------------|-------|-------|----|------|------|-------|-----|-----|-----|
| Field | f0 | f2 | f4 | f6 | f8 | f10 | f12 | ... | f30 |
| Qi | Mult1 | Load2 | | Add2 | Add1 | Mult2 | | | |

Lo scheduling dinamico e gli interrupt imprecisi

- Lo scheduling dinamico offre il fianco agli interrupt imprecisi
- Soluzione: effettuare le operazioni di WB se tutte le istruzioni precedenti sono state completate (es. con l'uso del ROB)

Esempio – passo passo

Ipotesi:

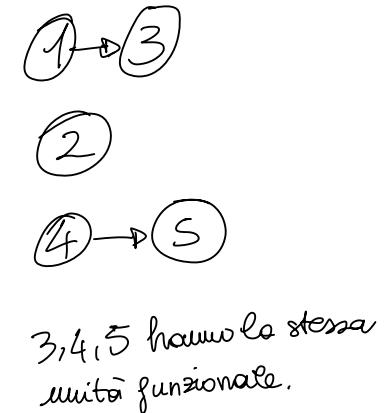
- ADD
- SUB
- LOGICHE

con Tommasulo si hanno solo 4 stadi:
IF, ID, EX, WB ma c'è MEM.

- tre unità funzionali (a, b e c)
- Ogni registro parte inizializzato con un valore uguale al suo indice (cioè a t_0 si ha $R_i = i$)
- Le istruzioni e le unità funzionali **assegnate** sono le seguenti:

al meglio: 11 cicli
Impiega sicuro 1 ciclo di IF + 1 di ID, 1 di WB + 8 cicli per l'esecuzione
Nel caso migliore impiega 11 cicli
Questo può richiedere solo dopo che è finita la 1.
AND
OR

| | | | |
|-----|-----|-----|-----|
| x5 | x6 | x7 | → a |
| x8 | x9 | x10 | → b |
| x11 | x5 | x12 | → c |
| x5 | x13 | x14 | → c |
| x15 | x5 | x16 | → c |



3 istruzioni usano l'unità C → l'unità C impiega 1 ciclo di ck per eseguire ciascuna istruzione 3×1 cicli in EX + 1 + 1 + 1 = 6 cicli. IF ID WB

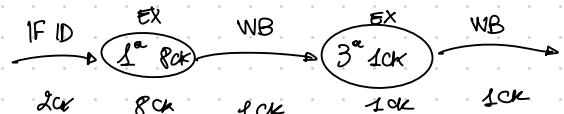
| | | | | |
|---|---|---|-----------------------|----------|
| 1 | 1 | a | $1 \times 8\text{ck}$ | $+1+1+1$ |
| 1 | 1 | b | $1 \times 8\text{ck}$ | $+1+1+1$ |

$$\left. \begin{array}{l} 3 \times C = 6\text{ck} \\ 1 \times a = 11\text{ck} \\ 1 \times b = 11\text{ck} \end{array} \right\} \text{con un ottica di occupazione delle unità funzionali.}$$

STIMA CONSIDERANDO LA DISPONIBILITÀ DELLE UNITÀ FUNZIONALI

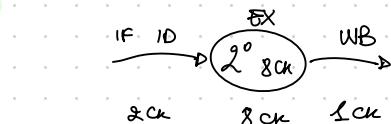
Se guardo alle dipendenze di dato, quindi alla combinazione delle istruzioni:

- So che XOR dipende da ADD, al termine del WB di ADD, XOR potrà eseguire e impiegherà 1 ciclo in EX

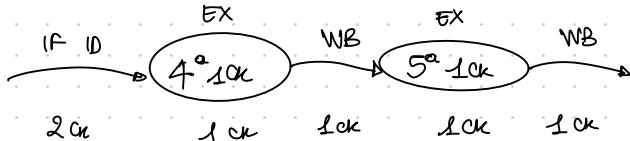


Questa coppia di istruzioni impiega 13 ck x eseguire.

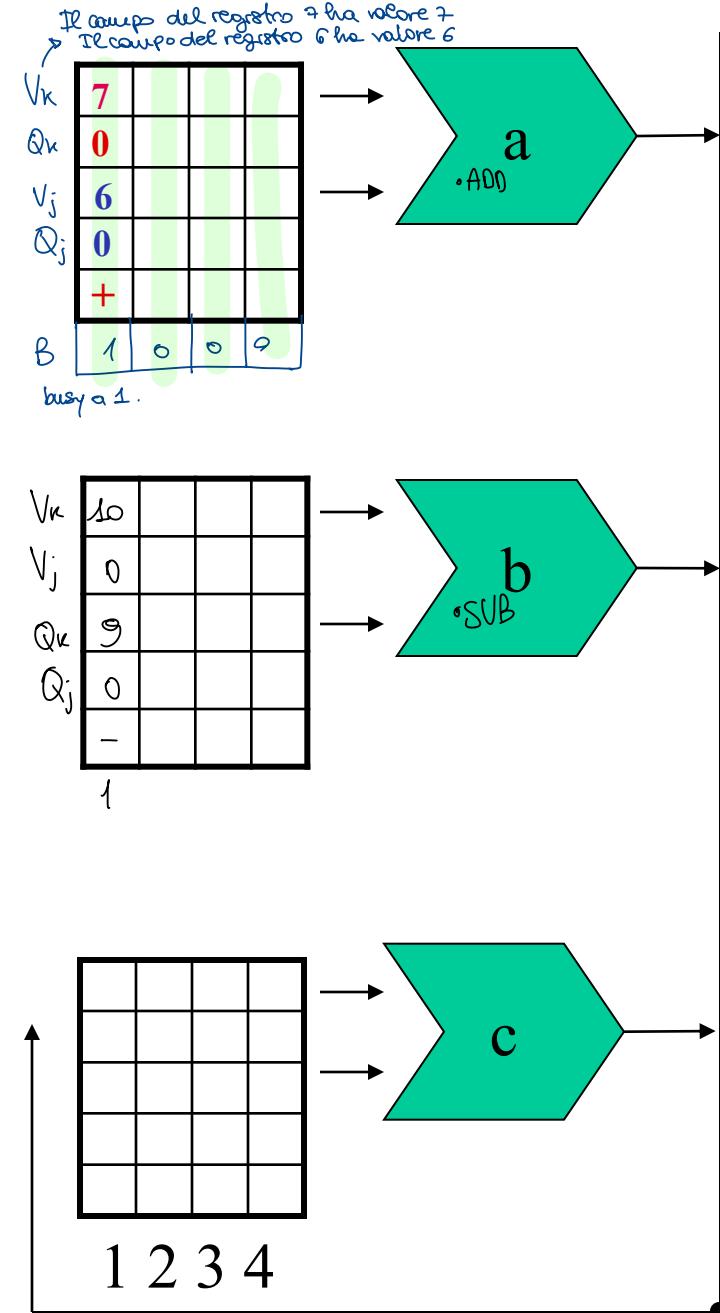
STIMA CONSIDERANDO LE DIPENDENZE DI DATO.



L'istruzione 2 impiega 11 cicli ck.



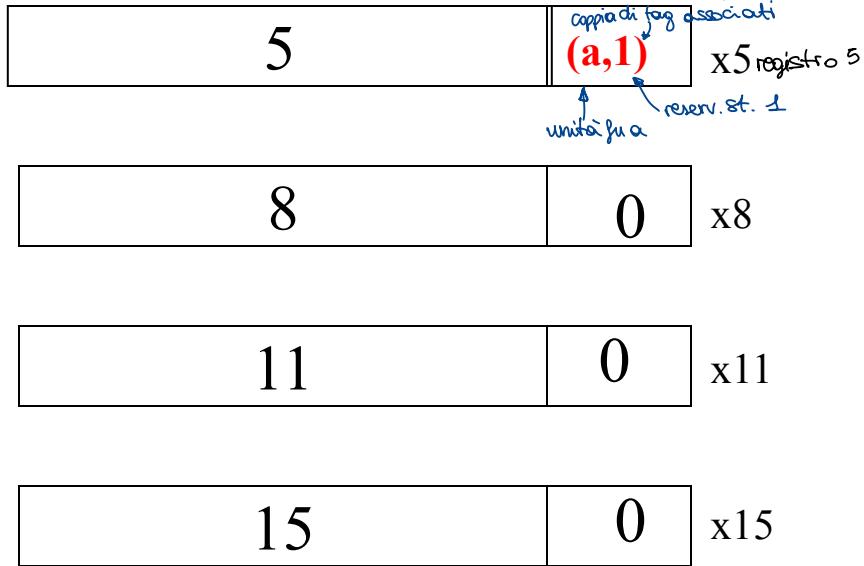
I istruzione 4 e 5 impiegano 6 ck.



Esempio

- ADD + x5 x6 x7 → ALU a
- SUB - x8 x9 x10 → ALU b
- XOR ⊕ x11 x5 x12 → ALU c
- AND ○ x5 x13 x14 → ALU c
- OR ○ x15 x5 x16 → ALU c

Identificare che il registro 5
dell'register file non contiene un valore
valido, ma è un valore interno
prodotto dall'esecuzione che si
trova nella reg. st. indicata
nel campo Q. → (a,1)

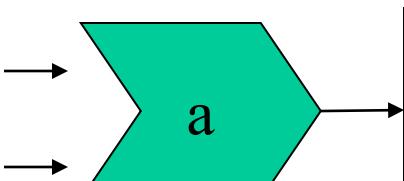


Register file

Attivazione di
ADD (issue)

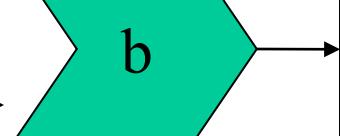
Logica di Tommaso

| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |



x6

| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |

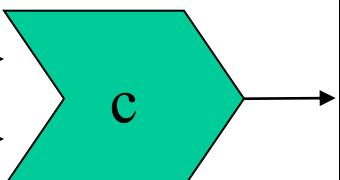


la sub può eseguire perché ha tutti i moduli che gli servono.

B 1

Quando faccio issue viene modificato il campo Q del register file. Infatti troveremo (b,1)

| | | | |
|--|--|--|--|
| | | | |
| | | | |
| | | | |
| | | | |



1 2 3 4

Esempio

- ADD + x5 x6 x7 → ALU a
- SUB - x8 x9 x10 → ALU b
- XOR ⊕ x11 x5 x12 → ALU c
- AND ○ x5 x13 x14 → ALU c
- OR ○ x15 x5 x16 → ALU c

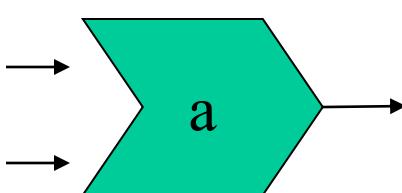


Register file

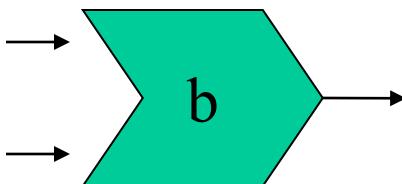
Attivazione di
SUB (issue)

Esempio

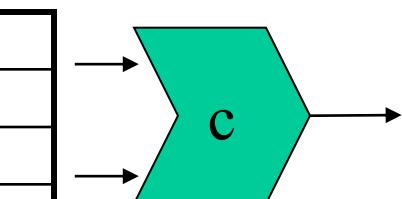
| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |



| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |



| | | | |
|-------|--|--|--|
| 12 | | | |
| 0 | | | |
| 5 | | | |
| (a,1) | | | |



1 2 3 4

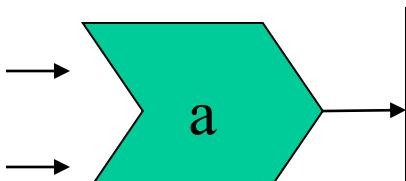
- ADD $+$ x5 x6 x7 → ALU a
- SUB $-$ x8 x9 x10 → ALU b
- XOR \oplus x11 x5 x12 → ALU c1
- AND \odot x5 x13 x14 → ALU c
- OR \cup x15 x5 x16 → ALU c



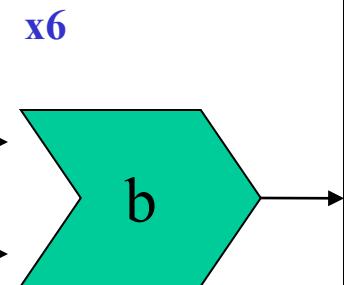
Register file

Attivazione di XOR (issue)
ALEA RAW risolta con forwarding
R5 non è coinvolto, non darà origine a WAR

| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |



| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |



| | | | |
|-------|----|--|--|
| 12 | 13 | | |
| 0 | 0 | | |
| 5 | 14 | | |
| (a,1) | 0 | | |
| ⊕ | ⊖ | | |

NON VA IN

1 2 3 4

Esempio

- ADD + x5 x6 x7 → ALU a
- SUB - x8 x9 x10 → ALU b
- XOR ⊕ x11 x5 x12 → ALU c
- AND ○ x5 x13 x14 → ALU c
- OR ⊖ x15 x5 x16 → ALU c

| | | |
|---|-------|----|
| 5 | (c,2) | x5 |
|---|-------|----|

| | | |
|---|-------|----|
| 8 | (b,1) | x8 |
|---|-------|----|

| | | |
|----|-------|-----|
| 11 | (c,1) | x11 |
|----|-------|-----|

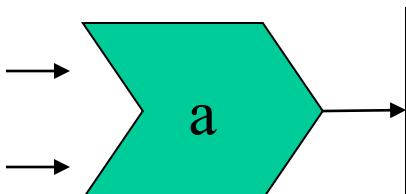
| | | |
|----|---|-----|
| 15 | 0 | x15 |
|----|---|-----|

Register file

Attivazione di
AND (issue)

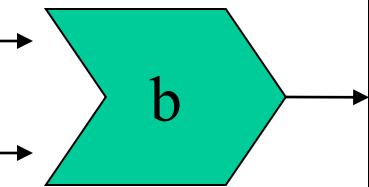
ESECUZIONE, RIMANE IN WAIT

| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |



R6

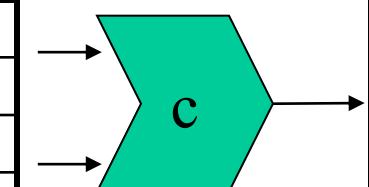
| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |



b

| | | | |
|-------|----|-------|--|
| 12 | 13 | 16 | |
| 0 | 0 | 0 | |
| 5 | 14 | 5 | |
| (a,1) | 0 | (c,2) | |
| ⊕ | ⊖ | ⊸ | |

1 2 3 4



c

Esempio

- ADD $+$ $x_5 \ x_6 \ x_7$ \rightarrow ALU a
- SUB $-$ $x_8 \ x_9 \ x_{10}$ \rightarrow ALU b
- XOR \oplus $x_{11} \ x_5 \ x_{12}$ \rightarrow ALU c
- AND \cap $x_5 \ x_{13} \ x_{14}$ \rightarrow ALU c
- OR \cup $x_{15} \ x_5 \ x_{16}$ \rightarrow ALU c



5

(c,2)

x5



8

(b,1)

x8



11

(c,1)

x11



15

(c,3)

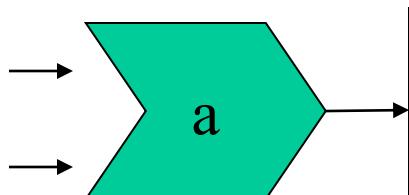
x15

Register file

Attivazione di
OR (issue)

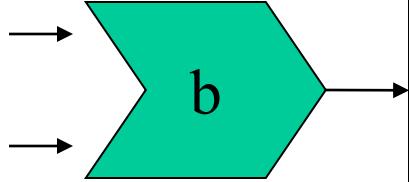
1 2 3 4

| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |



R6

| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |

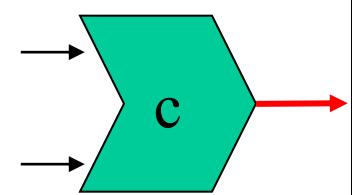


Esempio

- ADD $+$ $x_5 \ x_6 \ x_7$ \rightarrow ALU a
- SUB $-$ $x_8 \ x_9 \ x_{10}$ \rightarrow ALU b
- XOR \oplus $x_{11} \ x_5 \ x_{12}$ \rightarrow ALU c
- AND \cap $x_5 \ x_{13} \ x_{14}$ \rightarrow ALU c
- OR \cup $x_{15} \ x_5 \ x_{16}$ \rightarrow ALU c

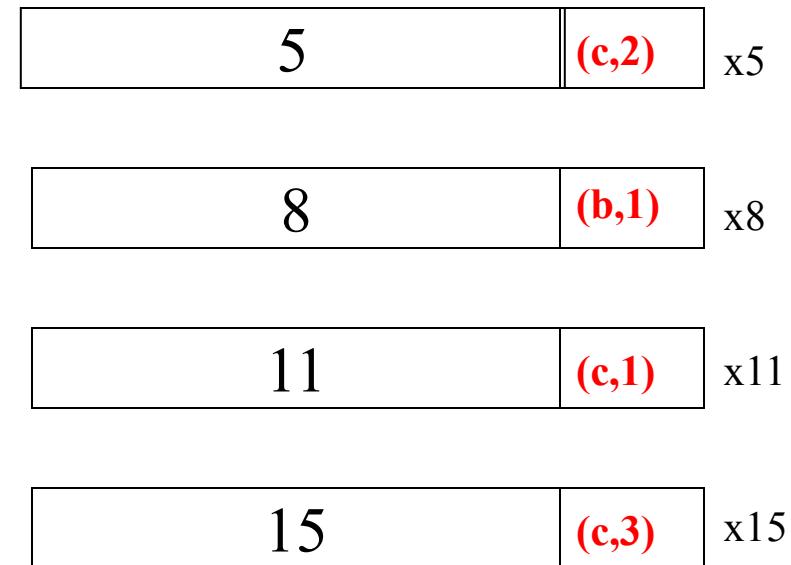
1 2 3 4

| | | | |
|----------|--------|--------|--|
| 12 | 13 | 16 | |
| 0 | 0 | 0 | |
| 5 | 14 | | |
| (a,1) | 0 | (c,2) | |
| \oplus | \cap | \cup | |
| 1 | 1 | 1 | |



busy

(c,2), V(c,2)

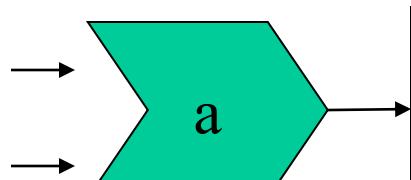


Register file

Completata
l'esecuzione, l'AND va in
WB dopodichè si libera
(c,2)

1 2 3 4

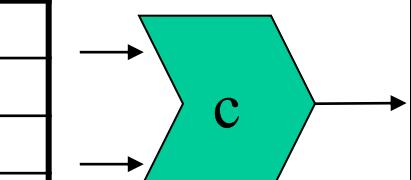
| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |
| 1 | | | |



busy

| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |
| 1 | | | |

busy



busy

Esempio

- ADD $+$ $x_5 \ x_6 \ x_7$ \rightarrow ALU a
- SUB $-$ $x_8 \ x_9 \ x_{10}$ \rightarrow ALU b
- XOR \oplus $x_{11} \ x_5 \ x_{12}$ \rightarrow ALU c
- AND \odot $x_5 \ x_{13} \ x_{14}$ \rightarrow ALU c
- OR \cup $x_{15} \ x_5 \ x_{16}$ \rightarrow ALU c

| | | |
|--------------|---|----|
| V(R13 . R14) | 0 | x5 |
|--------------|---|----|

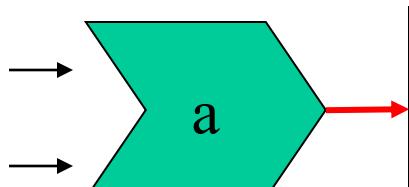
| | | |
|---|-------|----|
| 8 | (b,1) | x8 |
|---|-------|----|

| | | |
|----|-------|-----|
| 11 | (c,1) | x11 |
|----|-------|-----|

| | | |
|----|-------|-----|
| 15 | (c,3) | x15 |
|----|-------|-----|

Completata la fase WB dell' AND il bit busy di (a,1) viene posto a zero così il tag (a,1) torna disponibile. Inoltre il CRB è libero e potrebbe essere utilizzato da un altro risultato.

| | | | |
|---|--|--|--|
| 7 | | | |
| 0 | | | |
| 6 | | | |
| 0 | | | |
| + | | | |
| 1 | | | |



(a,1), V(a,1)

busy

| | | | |
|----|--|--|--|
| 10 | | | |
| 0 | | | |
| 9 | | | |
| 0 | | | |
| - | | | |
| 1 | | | |

busy

| | | | |
|-------|---|----|--|
| 12 | | 16 | |
| 0 | | 0 | |
| 5 | | V | |
| (a,1) | | 0 | |
| ⊕ | ⊖ | ⊸ | |
| 1 | 0 | 1 | |

busy

Esempio

- ADD + $x_5 \ x_6 \ x_7$ → ALU a
- SUB - $x_8 \ x_9 \ x_{10}$ → ALU b
- XOR \oplus $x_{11} \ x_5 \ x_{12}$ → ALU c
- AND \odot $x_5 \ x_{13} \ x_{14}$ → ALU c
- OR \cup $x_{15} \ x_5 \ x_{16}$ → ALU c



x5



x8



x11



x15

Register file

Completamento
della ADD

Alea WAW
implicitamente
risolta

1 2 3 4

Tommasulo vs. Scoreboarding

- Tommasulo vs Scoreboarding
 - 1. Control and buffers distributed with FUs vs. centralized in scoreboard (buffers called “reservation stations”)
 - 2. Registers in instructions replaced by pointers to reservation station buffer
 - 3. HW renaming of registers to avoid WAR and WAW hazards
 - 4. Common Data Bus broadcasts results to all FUs
 - 5. Load and Stores treated as FUs
 - 6. Memory disambiguation

Tommasulo vs. Scoreboarding

Advantages

1. **Broadcast on CDB more efficient** - operands available without register file read
2. **Avoids WAR hazards** by reading the operands in the instruction-issue order, instead of stalling the WB stage. To accomplish this an instruction reads an available operand before waiting for the other.
3. **Avoids WAW hazards** by renaming the registers (using the *id* of a reservation station rather than the register *id*)

Tommasulo cheat sheet

| Instruction state | Wait until | Action or bookkeeping |
|---|---|--|
| Issue FP operation | Station r empty | <pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r; </pre> |
| Load or store | Buffer r empty | <pre> if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes; </pre> |
| Load only | | RegisterStat[rt].Qi ← r; |
| Store only | | <pre> if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; </pre> |
| Execute FP operation | (RS[r].Qj = 0) and (RS[r].Qk = 0) | Compute result: operands are in Vj and Vk |
| Load/storestep 1 | RS[r].Qj = 0 & r is head of load-store queue | RS[r].A ← RS[r].Vj + RS[r].A; |
| Load step 2 | Load step 1 complete | Read from Mem[RS[r].A] |
| Write result FP operation or load | Execution complete at r & CDB available | <pre> ∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no; </pre> |
| Store | Execution complete at r & RS[r].Qk = 0 | <pre> Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no; </pre> |

Tommasulo steps #1

Issue:

- rd is the destination, rs and rt are the source register numbers, imm is the sign-extended immediate field, and r is the reservation station or buffer that the instruction is assigned to. RS is the reservation station data structure.
- The value returned by an FP unit or by the load unit is called result.
- RegisterStat is the register status data structure (not the register file, which is Regs[]).
- The destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued.
- If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands.

Tommasulo steps #2

Execution:

- The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued or when an instruction on which this instruction depends completes and does its write back.

Commit:

- When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose values of Qj or Qk are the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received.
- Thus the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store.
- Remember that, to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed.
- Because no concept of program order is maintained after the issue stage, this restriction is usually implemented by preventing any instruction from leaving the issue step if there is a pending branch already in the pipeline.

Load Store

Can be done safely out of order \Leftrightarrow if access different addresses.

- If they access the same address:

1. load before store in program order

=> interchanging them => WAR hazard!

2. Store before load in program order

=> interchanging them => RAW hazard!

3. Store before store in program order

=> interchanging them => WAW hazard!

- The processor has to check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load.
- A store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same data memory address
- This check requires the effective addresses to be computed.
 - Simple solution execute the effective address calculation in order.
 - If an active store or load in the store&load buffer has the same effective address wait to issue the instruction

Load Store

Can be done safely out of order \Leftrightarrow if access different addresses.

- If they access the same address:

1. load before store in program order

=> interchanging them => WAR hazard!

2. Store before load in program order

=> interchanging them => RAW hazard!

3. Store before store in program order

=> interchanging them => WAW hazard!

- The processor has to check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load.
- A store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same data memory address
- This check requires the effective addresses to be computed.
 - Simple solution execute the effective address calculation in order.
 - If an active store or load in the store&load buffer has the same effective address wait to issue the instruction