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

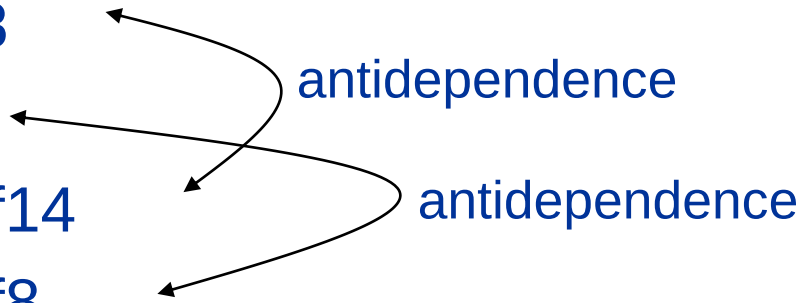
  fadd.d f0,f10,f14

  - 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
```



  - 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
```

- 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
    - Minimizes WAW and WAR hazards
- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values

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

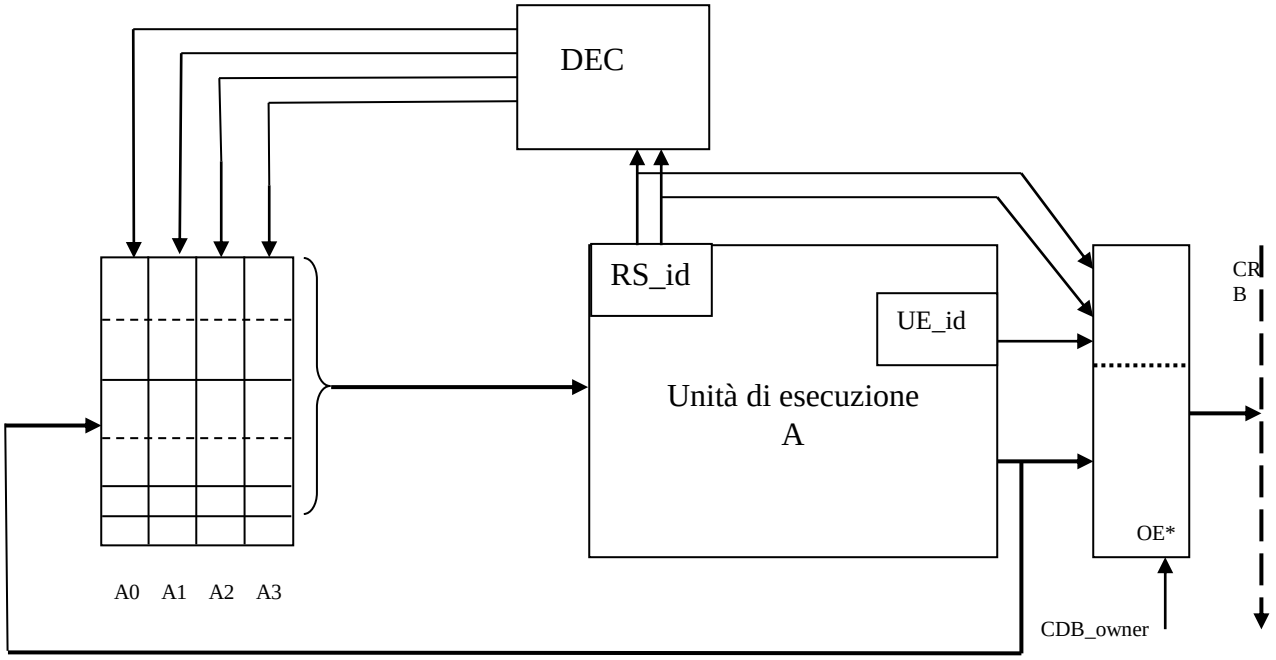
- **Obiettivo: stallare la pipeline solo in caso di alee strutturali e di controllo.**
- **Soluzione:**
  - Ogni Unità di Esecuzione dispone di una coda di ingresso ove le istruzioni ad essa destinate vengono mantenute fino al completamento dell'esecuzione
  - L'esecuzione procede in parallelo su tutte 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)

Nstalli = Nstallistrutturali + Nstallidicontrollo

Algoritmo di Tomasulo

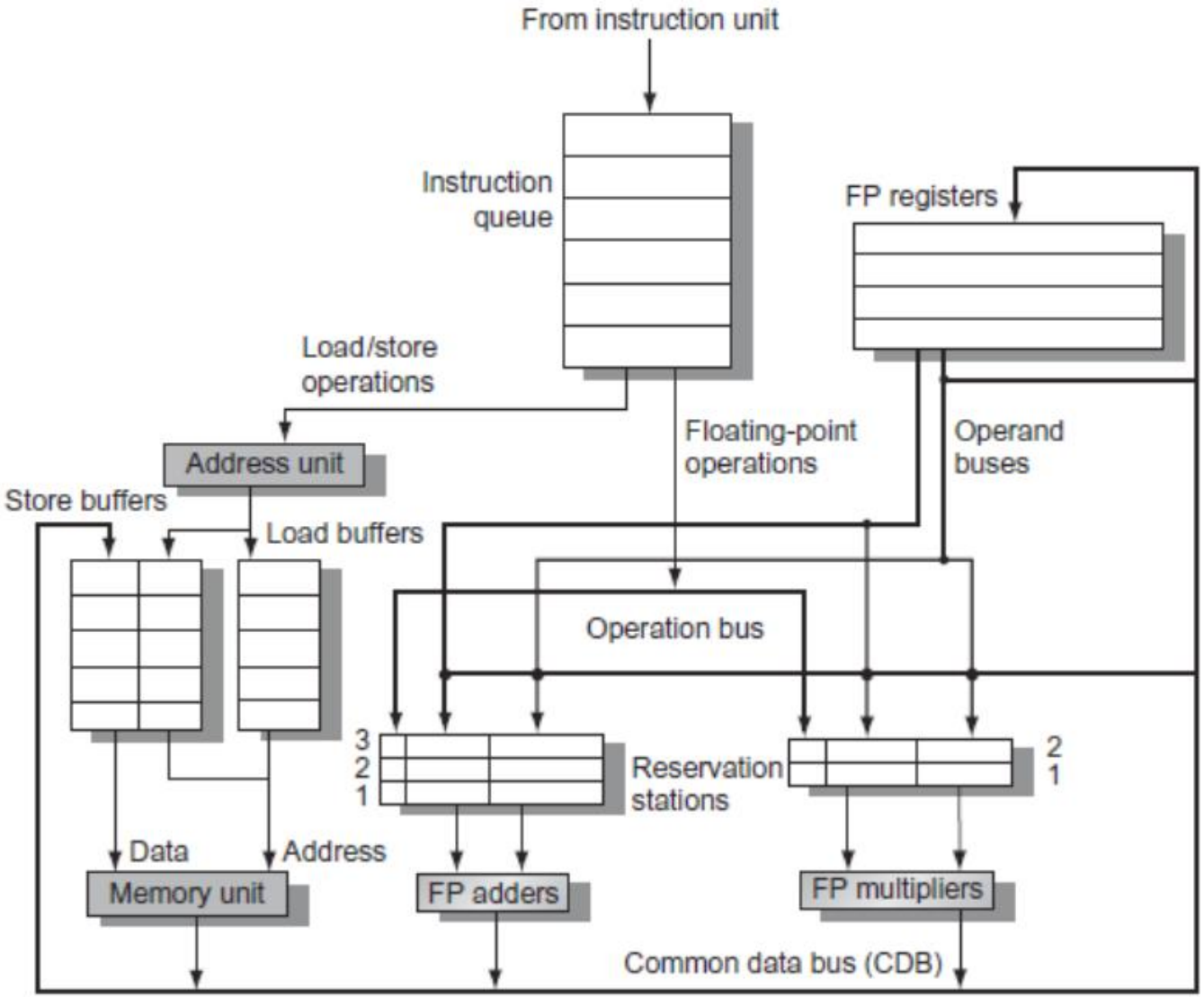
- Ogni unità di esecuzione multicycle viene equipaggiata con "stazioni di prenotazione" (reservation stations) che ospitano sia l'istruzione in esecuzione sia istruzioni pronte per essere eseguite e in attesa dell'unità di esecuzione, sia istruzioni "sospese" in attesa di uno o due operandi
- , Nello stadio "ID" :
  - 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*) .
- le alee "raw" sono dunque risolte con il forwarding dei risultati a tutte le unità che hanno bisogno di operandi (reservation stations e memoria)

Unità di esecuzione



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

Tomasulo's Algorithm



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)

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**

- Q_i contiene **l'identificatore del Load Buffer o della Reservation Station che produrrà il contenuto del registro [un TAG: (Unit_ID, RS_ID)]**
- V_i 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 ≠ 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)**

Q_i (TAG) V_i(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)
  - Vj e Vk: contengono l'operando se QJ / Qk = 0
  - OP: operazione da eseguire (Es: FADD/FSUB…)

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

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

Le coppie(Qj,Vj) e (Qk,Vk) 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

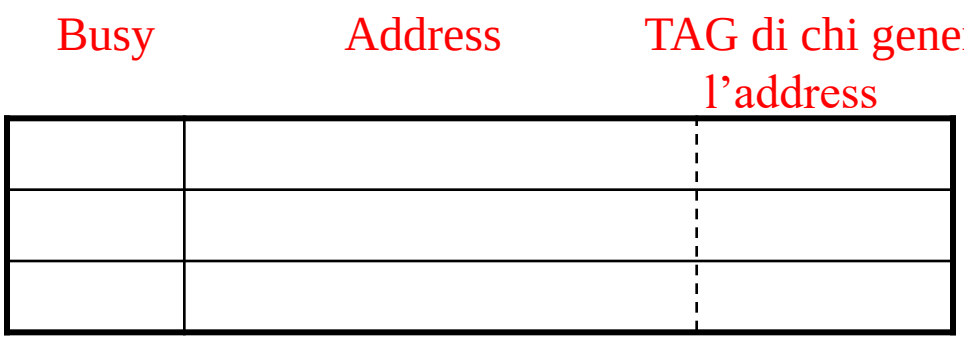
- Ordine nelle istruzioni di scrittura in memoria:
- Di solito si utilizza la politica di “strong write ordering”:
  - Le istruzioni di scrittura non sono mai eseguite fuori ordine tra loro
  - Una istruzione di scrittura non viene mai eseguita prima di una di lettura che precede la write nel codice (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



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 imprecis

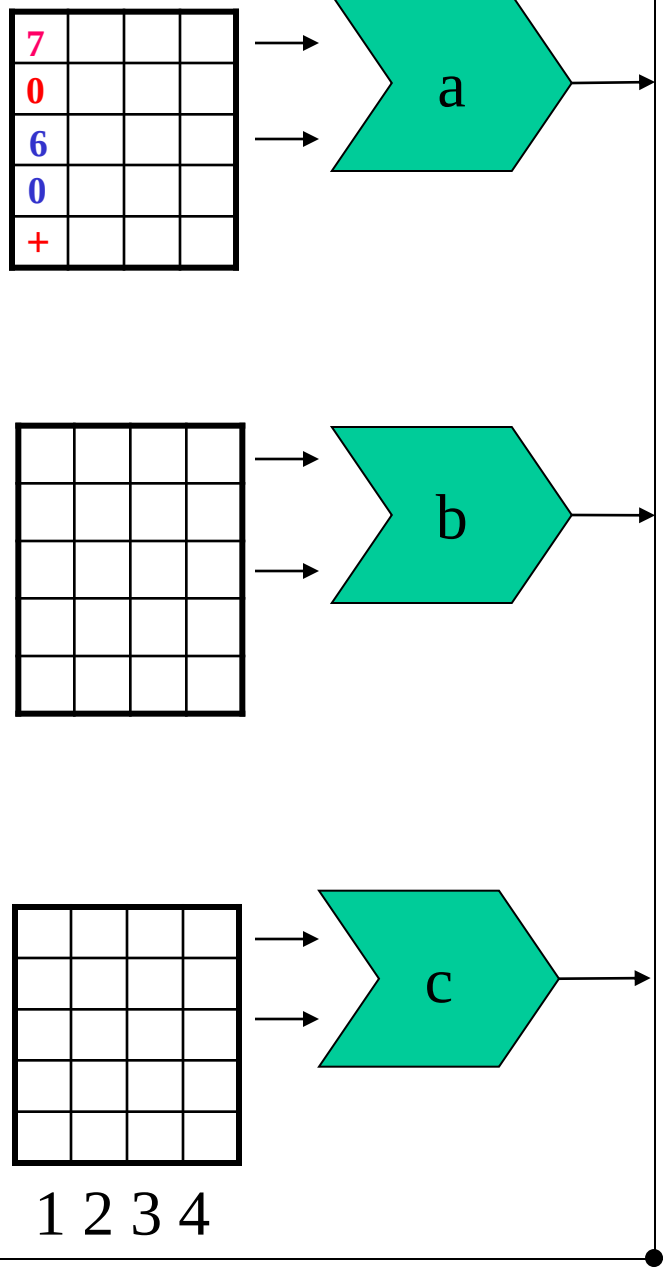
- Lo scheduling dinamico offre il fianco agli interrupt imprecisi
- Soluzione: effettuare le operazioni di WB se tutte le istruzioni precedenti sono state completate (es. con l'uso del ROB)

Esempio – passo passo

Ipotesi:

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

ADD	**x5** x6 x7	→ a
SUB	x8 x9 x10	→ b
XOR	x11 **x5** x12	→ c
AND	**x5** x13 x14	→ c
OR	x15 **x5** x16	→ c



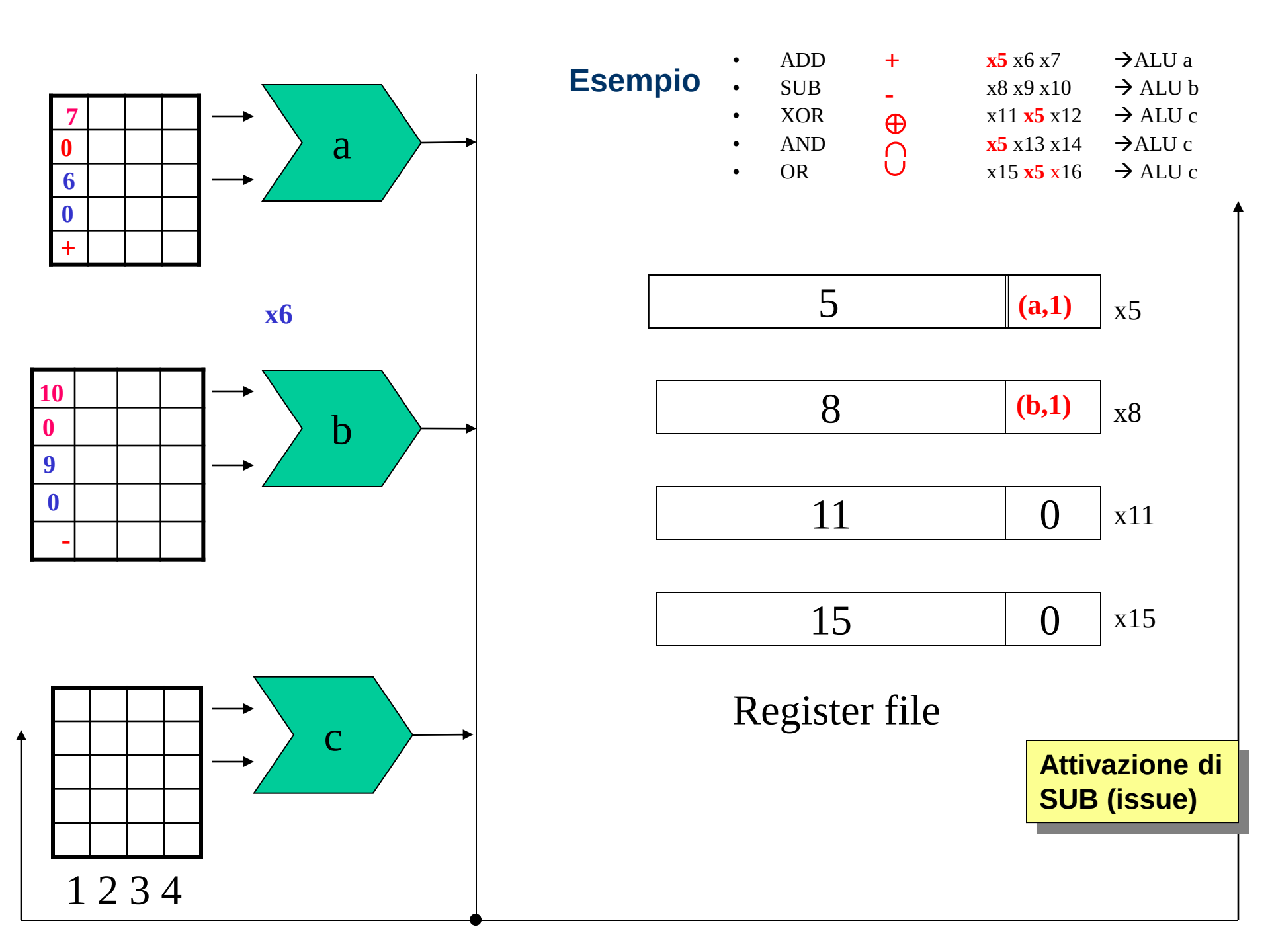
Esempio

• 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	**(a,1)**	x5
8	0	x8
11	0	x11
15	0	x15

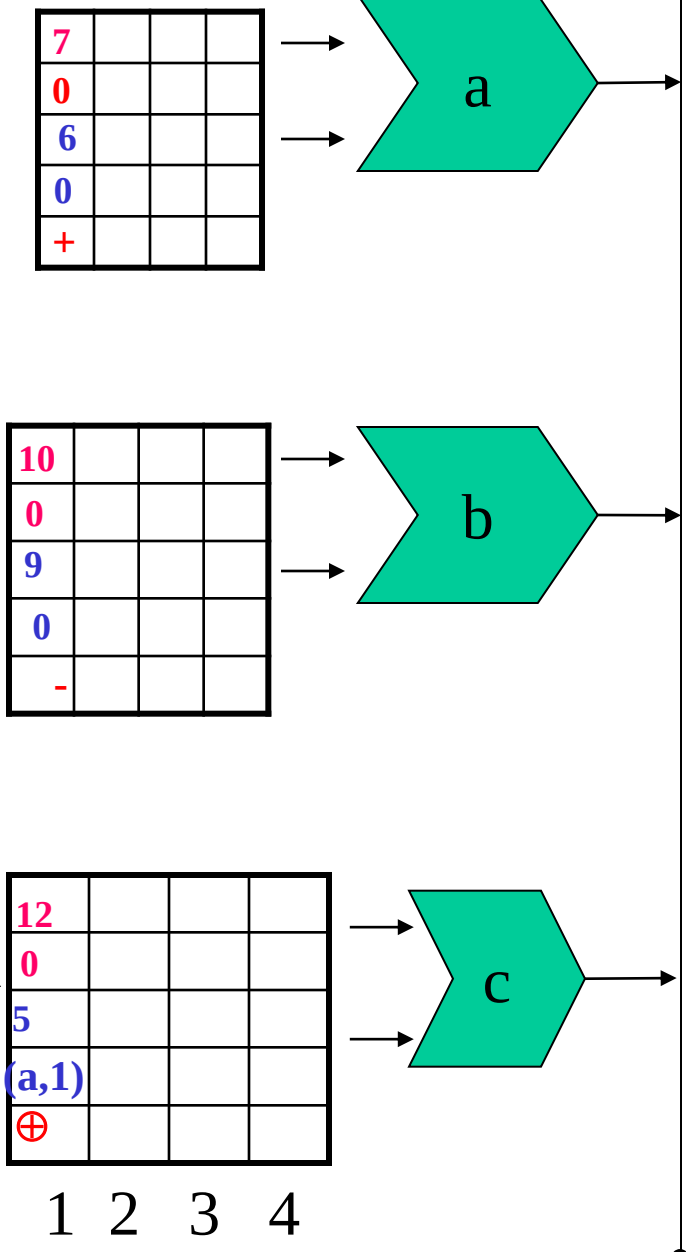
Register file

Attivazione di ADD (issue)


Esempio
ADD +
SUB -
XOR ⊕
AND ∩
OR ∪
x5 x6 x7 →ALU a
x8 x9 x10 → ALU b
x11 x5 x12 → ALU c
x5 x13 x14 →ALU c
x15 x5 x16 → ALU c
7
0
6
0
+
a
x6
10
0
9
0
-
b
c
1 2 3 4
5
(a,1)
x5
8
(b,1)
x8
11
0
x11
15
0
x15
Register file
Attivazione di SUB (issue)

Esempio

- ADD + **x5** x6 x7 → ALU a
- SUB - x8 x9 x10 → ALU b
- **XOR** ⊕ **x11 x5 x12** → **ALU c1**
- AND ∩ x5 x13 x14 → ALU c
- OR x15 **x5** x16 → ALU c



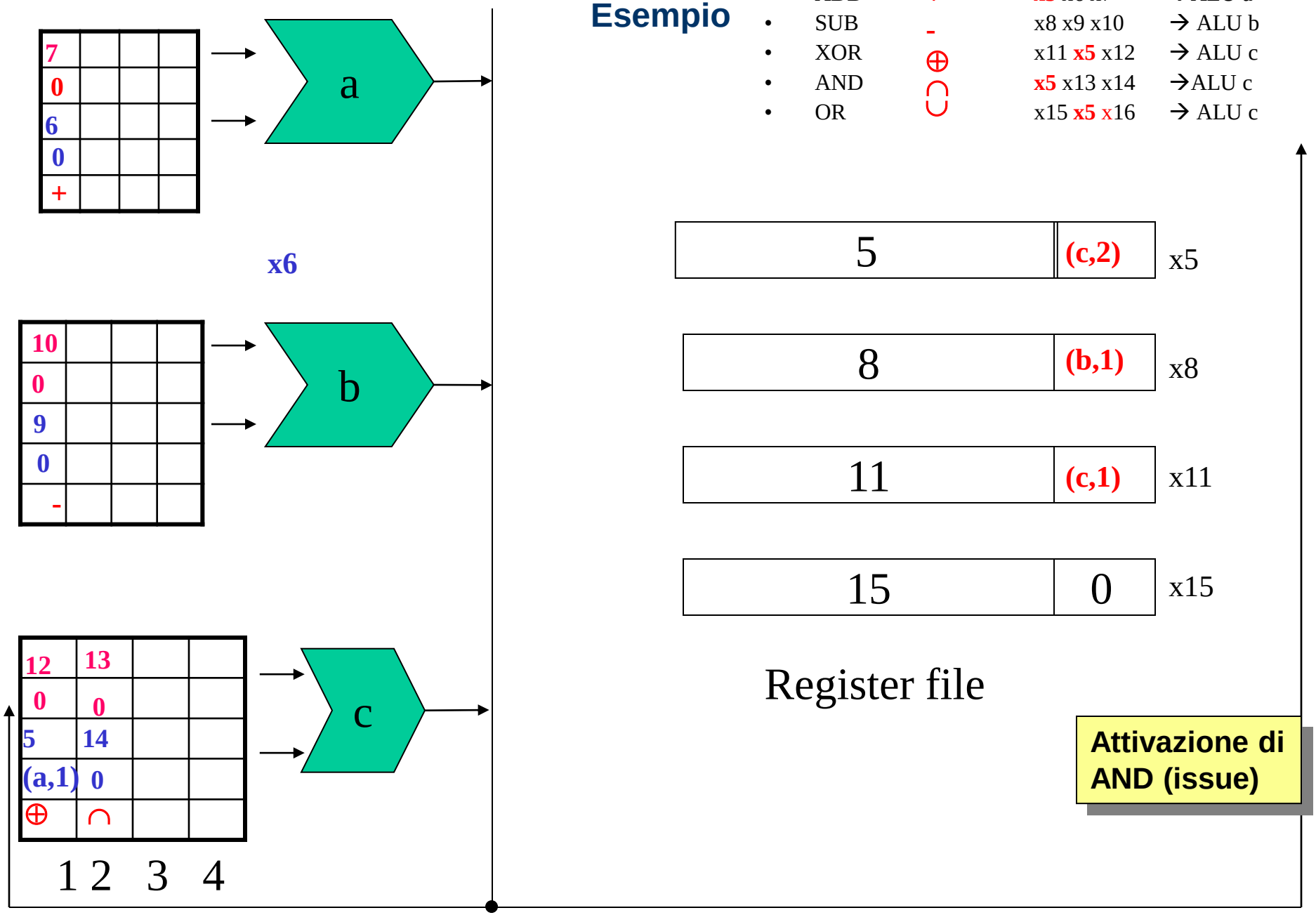
5	(a,1)	x5
8	(b,1)	x8
11	(c,1)	x11
15	0	x15

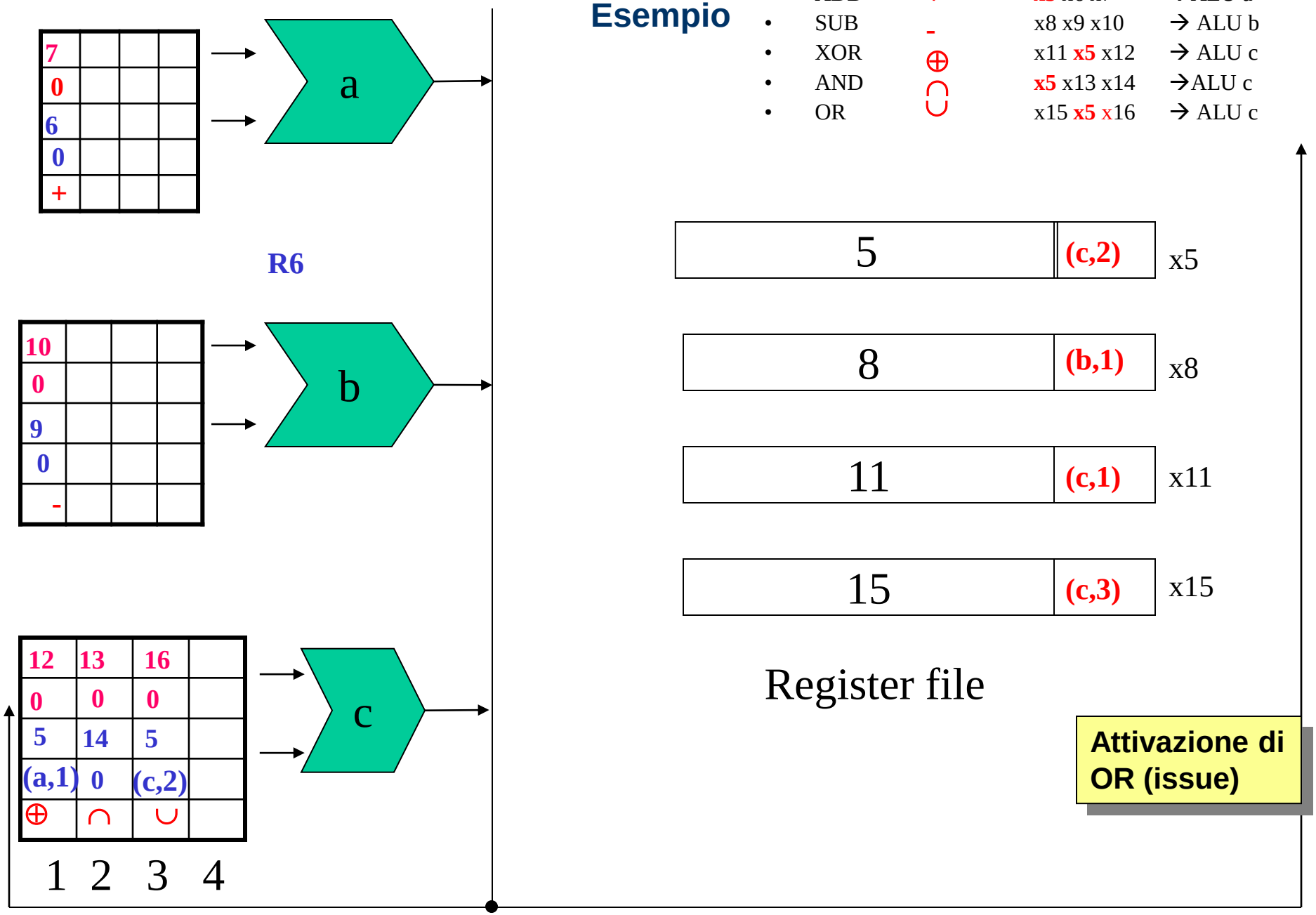
Register file

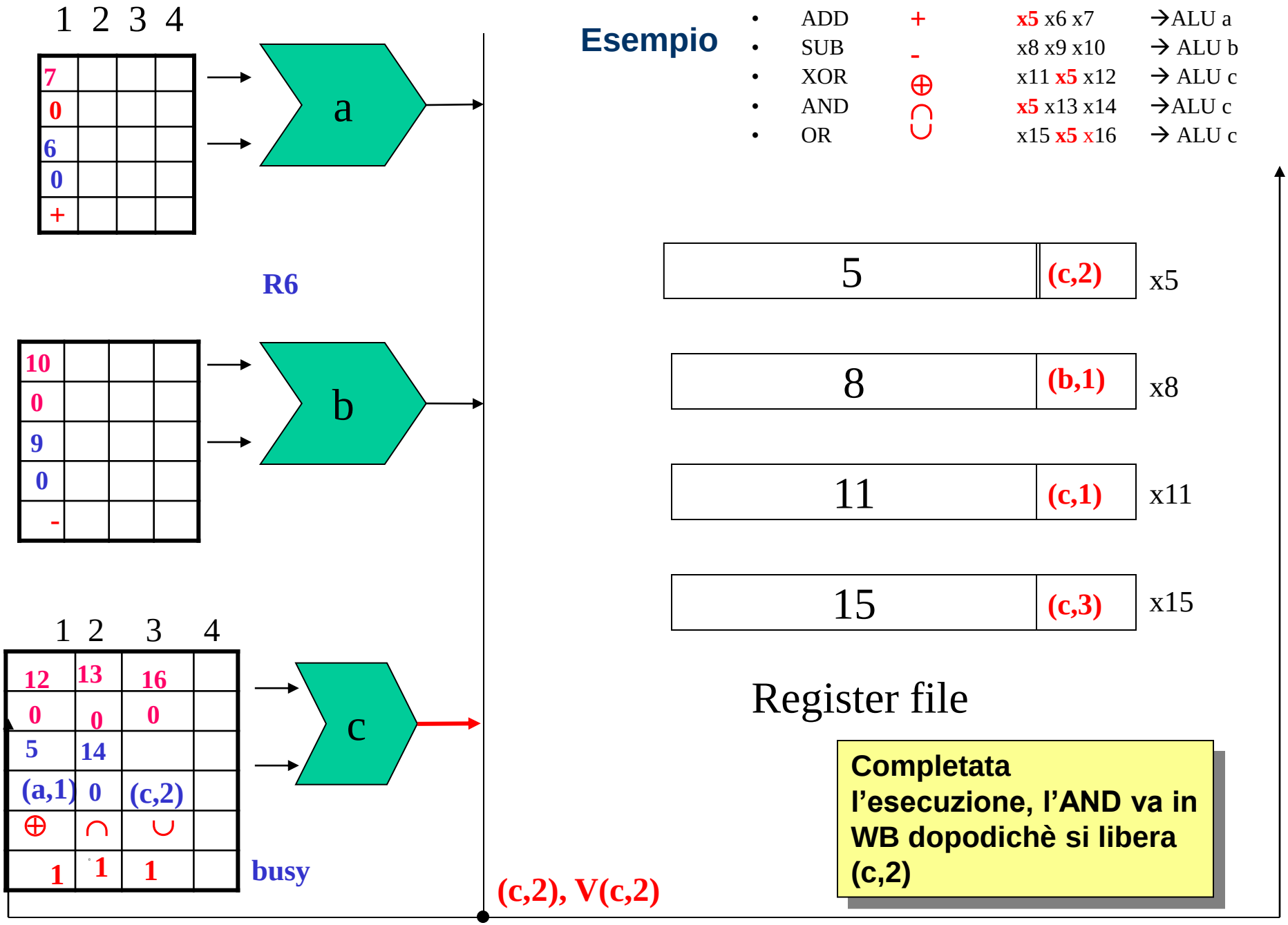
Attivazione di XOR (issue)

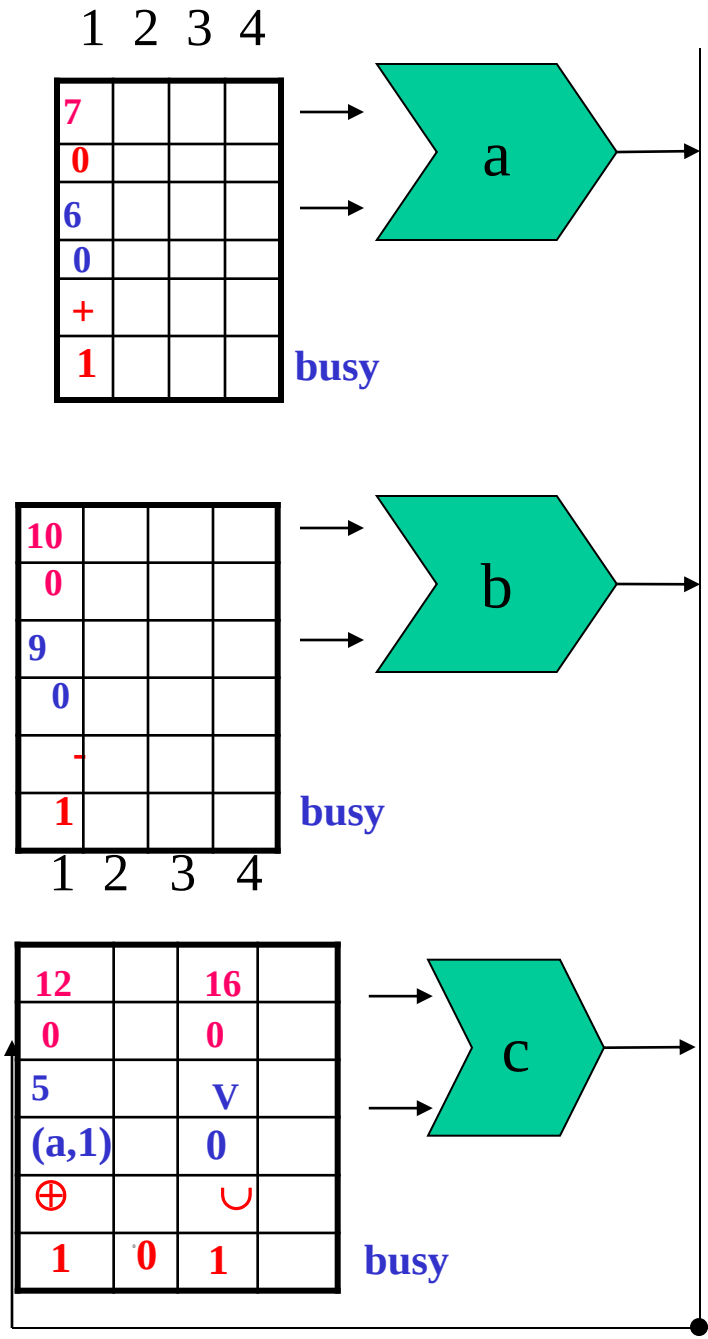
ALEA RAW risolta con forwarding

R5 non è coinvolto, non darà origine a WAR


Esempio
ADD +
SUB -
XOR ⊕
AND ∩
OR
x5 x6 x7 →ALU a
x8 x9 x10 → ALU b
x11 x5 x12 → ALU c
x5 x13 x14 →ALU c
x15 x5 x16 → ALU c
7
0
6
0
+
a
x6
10
0
9
0
-
b
12
13
0
0
5
14
(a,1)
0
⊕
∩
c
1 2 3 4
5
(c,2)
x5
8
(b,1)
x8
11
(c,1)
x11
15
0
x15
Register file
Attivazione di AND (issue)


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
7
0
6
0
+
a
R6
10
0
9
0
-
b
12 13 16
0 0 0
5 14 5
(a,1) 0 (c,2)
⊕ ∩ ∪
c
1 2 3 4
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
+
a
R6
10
0
9
0
-
b
1 2 3 4
12
0
5
(a,1)
⊕
1
13
0
14
0
∩
1
16
0
(c,2)
∪
1
c
busy
(c,2), V(c,2)
Esempio
ADD +
SUB -
XOR ⊕
AND ∩
OR ∪
x5 x6 x7 →ALU a
x8 x9 x10 → ALU b
x11 x5 x12 → ALU c
x5 x13 x14 →ALU c
x15 x5 x16 → ALU c
5 (c,2) x5
8 (b,1) x8
11 (c,1) x11
15 (c,3) x15
Register file
Completata l'esecuzione, l'AND va in WB dopodichè si libera (c,2)



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

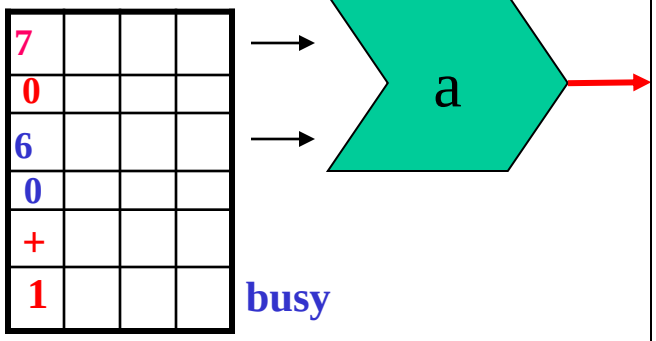
V(R13 . R14)	0	x5
8	(b,1)	x8
11	(c,1)	x11
15	(c,3)	x15

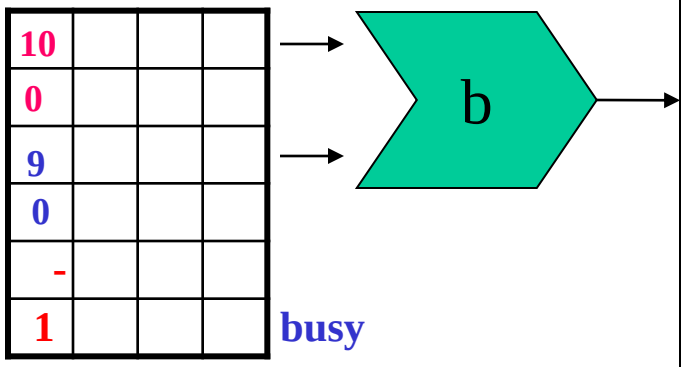
Completata la fase WB dell' AND il bit busy di (a,1) viene posto a zero così il tag (a,1) torna disponibile. Inoltre il CRB è libero e potrebbe essere utilizzato da un altro risultato.

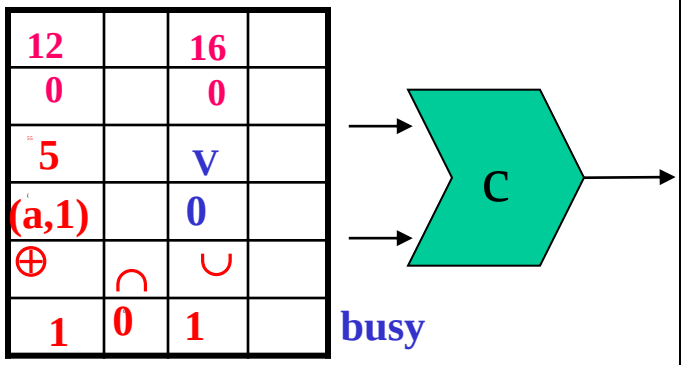
Esempio

- ADD +
- SUB -
- XOR ⊕
- AND ∩
- OR ∪

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







1 2 3 4

a

b

c

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

V(R13 . R14)	0	x5
	(b,1)	x8
	(c,1)	x11
	(c,3)	x15

Register file

Completamento della ADD

Alea WAW implicitamente risolta

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	`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;`
Load or store	Buffer r empty	`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;`
Load only		`RegisterStat[rt].Qi ← r;`
Store only		`if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
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	`∀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;`
Store	Execution complete at r & `RS[r].Qk = 0`	`Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;`

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 ⇔ 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