

## **– algoritmo di tommasulo**

## **– algoritmo di tommasulo**

# Scheduling dinamico con esecuzione speculativa

## Scheduling Dinamico (es. con algoritmo di TOMASULO):

**Obiettivo:** stallare la pipeline solo in caso di alee strutturali e di controllo.

- |                               |                         |
|-------------------------------|-------------------------|
| – Attivazione sequenziale:    | IN-ORDER DISPATCHING    |
| – Esecuzione fuori ordine:    | OUT-OF-ORDER EXECUTION  |
| – Completamento fuori ordine: | OUT-OF-ORDER COMPLETION |

## Scheduling Dinamico con esecuzione speculativa:

▪ **Obiettivo:** stallare la pipeline solo in caso di alee strutturali

▪ **Soluzione:** predizione, evitando azioni “irrevocabili” prima di essere certi che tale azione deve essere effettuata

▪ **Conseguenza:** si amplia la finestra su cui cercare il parallelismo a livello di istruzione

- |                               |                                       |
|-------------------------------|---------------------------------------|
| – Attivazione sequenziale:    | PREDICTION BASED IN-ORDER DISPATCHING |
| – Esecuzione fuori ordine     | OUT-OF-ORDER EXECUTION                |
| – Completamento fuori ordine: | OUT-OF-ORDER COMPLETION               |
| – Avanzamento in ordine:      | IN-ORDER COMMIT                       |

Se voglio intradurre un branch prediction vuol dire fare esecuzione fuori ordine.

fdiv  $f_3, f_2, f_1$

bne  $x_1, x_2$ , #LOOP

fadd  $f_4, f_5, f_6$

fmul . . . . .

LOOP fsub  $f_4, f_5, f_7$

suppongo che ci sia una dipendenza di dato su  $x_2$ , per cui impiegherò dei cicli prima di dare l'esito

ESECUZIONE OUT OF ORDER,  
COMMIT IN ORDER.

Voglio fare commit in order, cioè prima scrivo l'fdiv e poi le altre istruzioni e solo quando è noto l'esito del branch decido se eventuali istruzioni che hanno completato possono o meno fare commit.

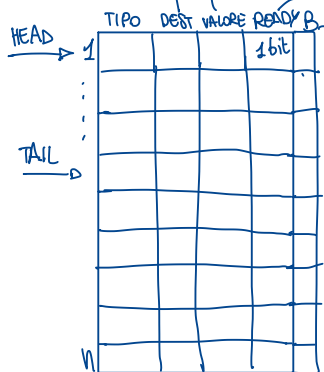
Ho bisogno di una struttura dati HW che mi consenta di immagazzinare temporaneamente i risultati delle operazioni in attesa di sapere se quelle istruzioni vanno eseguite o meno, cioè in attesa che esegua il branch e comunque dopo che tutte le istruzioni che venivano precedentemente nel codice abbiano terminato.

Es. la fadd scrive nel register file solo dopo che ci ha scritto fdiv. (anche se non ci sono dipendenze)

Ho bisogno di un reorder buffer per fare commit in ordine ed esecuzione fuori ordine.

# Soluzione

- È necessario separare la fase di esecuzione data driven dalla fase in cui si rendono impegnativi i risultati (COMMIT)
- Si utilizza uno schema simile a quello di Tomasulo ma con aggiunto il Reorder Buffer



dice qual è il registro in cui devo salvare il risultato.  
 dice qual è il valore che devo salvare nel registro  
 mi dice se l'istruzione è completa o no

mi dice se  
 la entry  
 è occupata  
 o no.

Ciascuna entry contiene determinati campi  
 che servono per immagazzinare potenziali  
 risultati.

Qui dentro metto istruzioni per le quali devo  
 aspettare per fare commit, perché se lo faccio modifi-  
 cherei lo stato della pipeline.

Memoria con due puntatori  
 con logica FIFO.

## ALGORITMO DI TOMM. CON REORDER BUFFER

I risultati nel register file li scriverà il reorder buffer, non li scriveranno più le reserv. st. perché adesso tengo congelati nel reorder buffer i risultati delle istruzioni finché non è stata fatta l'esecuzione in ordine delle istruz. prima. Quindi nel register file quando faccio l'issue vado a scrivere un identificativo dell'entry del reorder buffer che contiene l'istruzione che produrrà il risultato per quel registro (quindi un tag associato al reorder buffer).

Nel reorder buffer la coda punta alla entry del reorder buffer dell'istruzione di cui vado a fare decode. Se l'entry è busy faccio stall, non posso fare il decode, il r.buffer è pieno.

$$f_{div} \quad f_1, f_2, f_3$$
$$f_{mul} \quad f_5, f_1, f_4$$

## REORDER BUFFER

[illegible]

perché non  
ha committato,  
non ha finito.

## REGISTER FIVE

$r_1$		#3 -
$r_2$	2	0
$r_3$	3	0
$r_4$		
$r_5$		#4 -

Non e' più riferito  
alle res. st. ma alle  
entry del rear buffer.

sarà un puntatore a una entry del recorder buffer

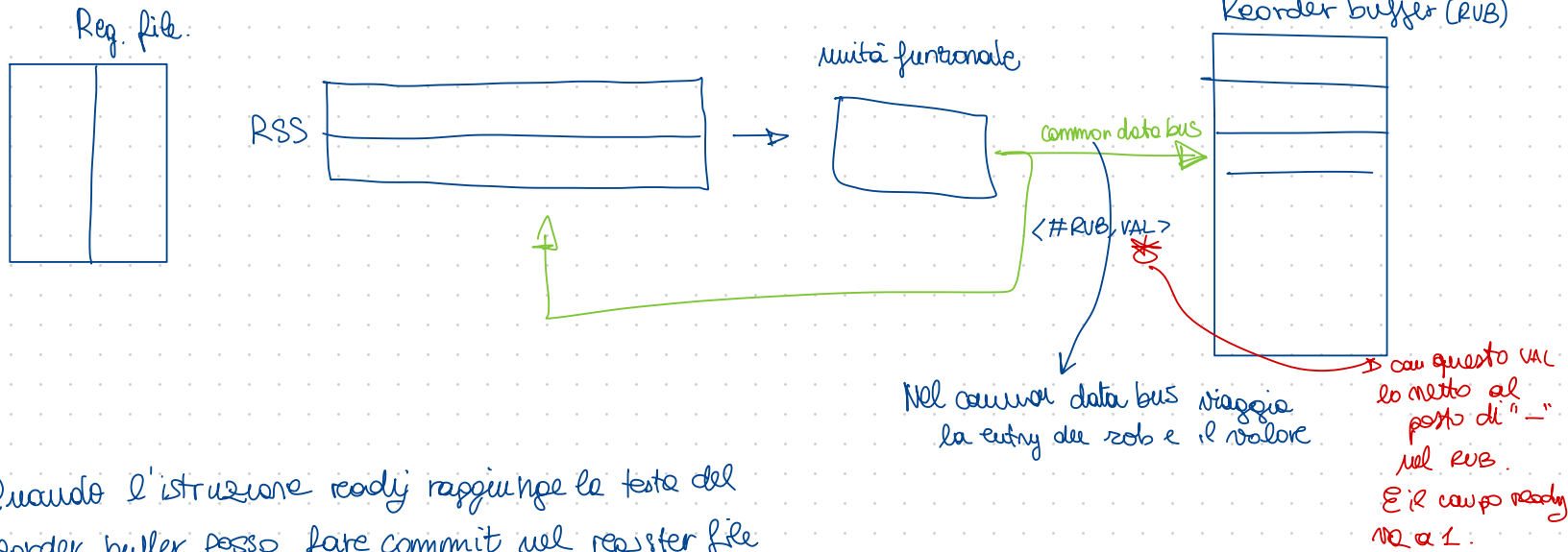
↑  
 $Q_j$   $V_j$   $Q_k$   $V_k$  → A quale entry del  $E_{\text{row}}$  è associato il risultato

FDIV	0	2	0	3	3	
FWL	#3	-	0	4	4	

Quando l'istruzione termina l'istruzione, il common data bus produce il valore associato a quell'istruzione, ma come tag che il tag delle entry del rob.

2. Il common data bus non deve scrivere nel register file ma nel reorder buffer alla res. st.

così posso fare forwarding dei risultati, quindi ricevere le RAW e non modificare il RF

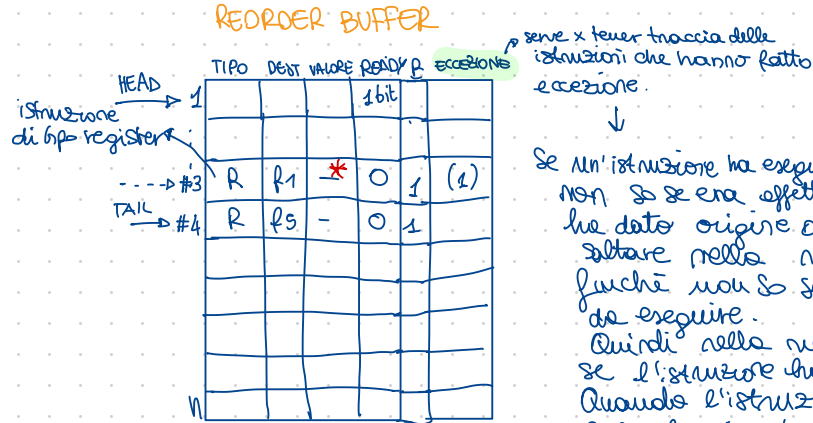


Quando l'istruzione ready raggiunge la testa del reorder buffer posso fare commit nel register file perché sono in ordine.



Quindi il reorder buffer tiene i dati prodotti dalle unità funzionali finché l'istruzione in testa non diventa ready. A quel punto si può fare commit nel register file. Quando faccio commit sposto la testa di una posizione.

Quando la branch arriva alla testa del reorder buffer e so l'esito della sua condizione posso sapere se tutte le istruzioni che ho eseguito in modo speculativo erano da eseguire o no. Se sono da eseguire continuo a committare in ordine, se non devo eseguire le istruzioni tra il branch e la label, cancello le rub e faccio il fetch dell'istruzione corretta.



Se un'istruzione ha eseguito speculativamente (per cui non so se era effettivamente da eseguire) ha dato origine a un'eccezione, non posso saltare nella routine di gestione dell'etichetta finché non so se l'istruzione era effettivamente da eseguire. Quindi nella nuova colonna tengo traccia di se l'istruzione ha generato o meno un'istruzione. Quando l'istruzione raggiunge la testa del RUB lancio l'eccezione.



# Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution

# Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

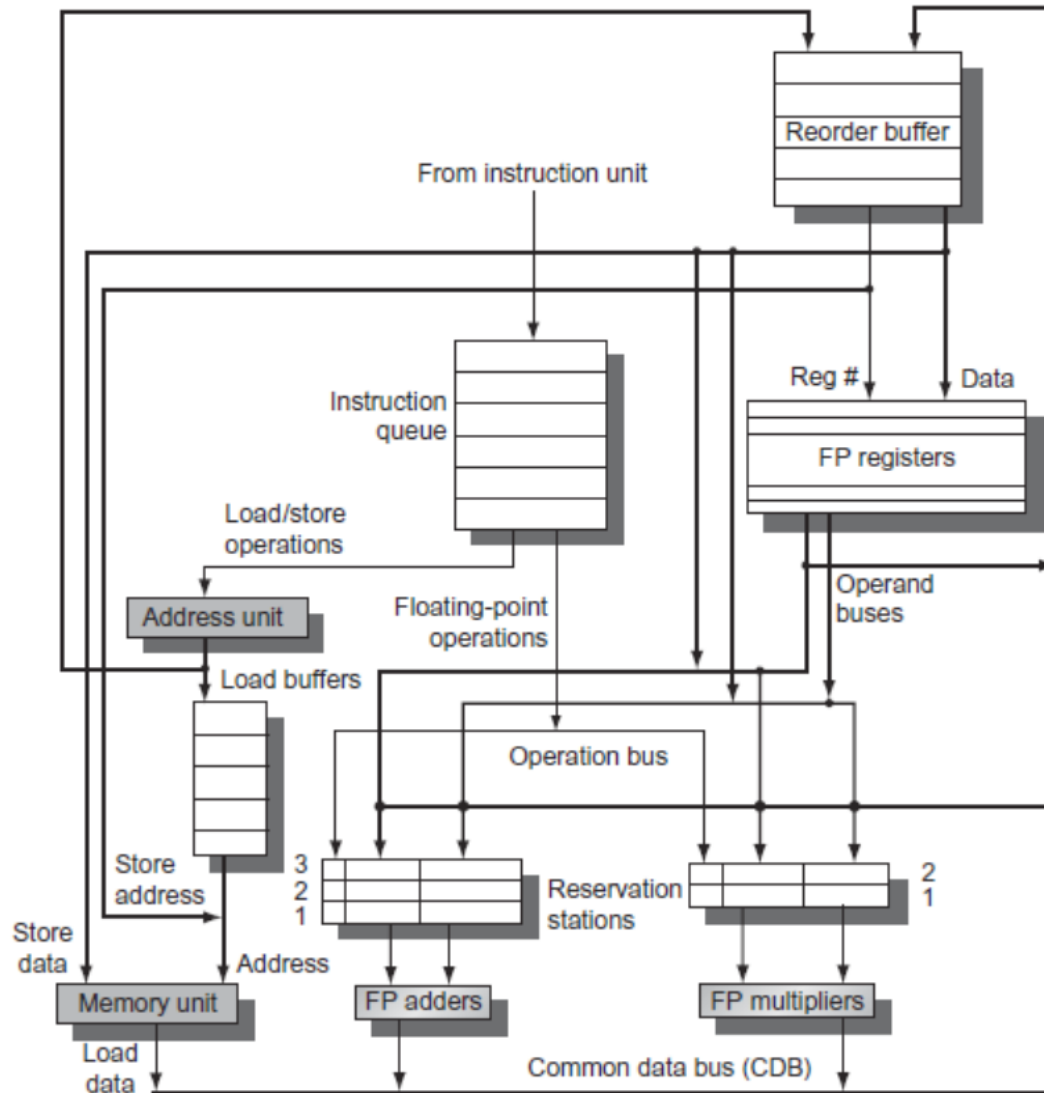
# Reorder Buffer

- Issue:
  - Allocate RS and ROB, read available operands
- Execute:
  - Begin execution when operand values are available
- Write result:
  - Write result and ROB tag on CDB
- Commit:
  - When a predicted branch reaches head of ROB, update register
  - When a mispredicted branch reaches head of ROB, discard all entries

# Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit

# Reorder Buffer



# Reorder Buffer

Status	Wait until	Action or bookkeeping
Issue all instructions		<pre> if (RegisterStat[rs].Busy) /* in-flight instr. writes rs */ { h ← RegisterStat[rs].Reorder;   if (ROB[h].Ready) /* Instr completed already */     { RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0; }     else { RS[r].Qj ← h; } /* wait for instruction */ } else { RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0; }; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no; </pre>
FP operations and stores	Reservation station (r) and ROB (b) both available	<pre> if (RegisterStat[rt].Busy) /* in-flight instr writes rt */ { h ← RegisterStat[rt].Reorder;   if (ROB[h].Ready) /* Instr completed already */     { RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0; }     else { RS[r].Qk ← h; } /* wait for instruction */ } else { RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0; }; </pre>
FP operations		<pre> RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd; </pre>
Loads		<pre> RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt; </pre>
Stores		<pre> RS[r].A ← imm; </pre>
Execute FP op	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj = 0) and there are no stores earlier in the queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj = 0) and store at queue head	ROB[h].Address ← RS[r].Vj + RS[r].A;
Write result all but store	Execution done at r and CDB available	<pre> b ← RS[r].Dest; RS[r].Busy ← no; ∀x (if (RS[x].Qj == b) { RS[x].Vj ← result; RS[x].Qj ← 0; }); ∀x (if (RS[x].Qk == b) { RS[x].Vk ← result; RS[x].Qk ← 0; }); ROB[b].Value ← result; ROB[b].Ready ← yes; </pre>
Store	Execution done at r and (RS[r].Qk = 0)	ROB[h].Value ← RS[r].Vk;
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready = yes	<pre> d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction == Branch) { if (branch is mispredicted)   { clear ROB[h], RegisterStat; fetch branch dest; }   else if (ROB[h].Instruction == Store)     { Mem[ROB[h].Destination] ← ROB[h].Value; }   else /* put the result in the register destination */     { Regs[d] ← ROB[h].Value; }; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder == h) { RegisterStat[d].Busy ← no; }; </pre>

# Reorder Buffer

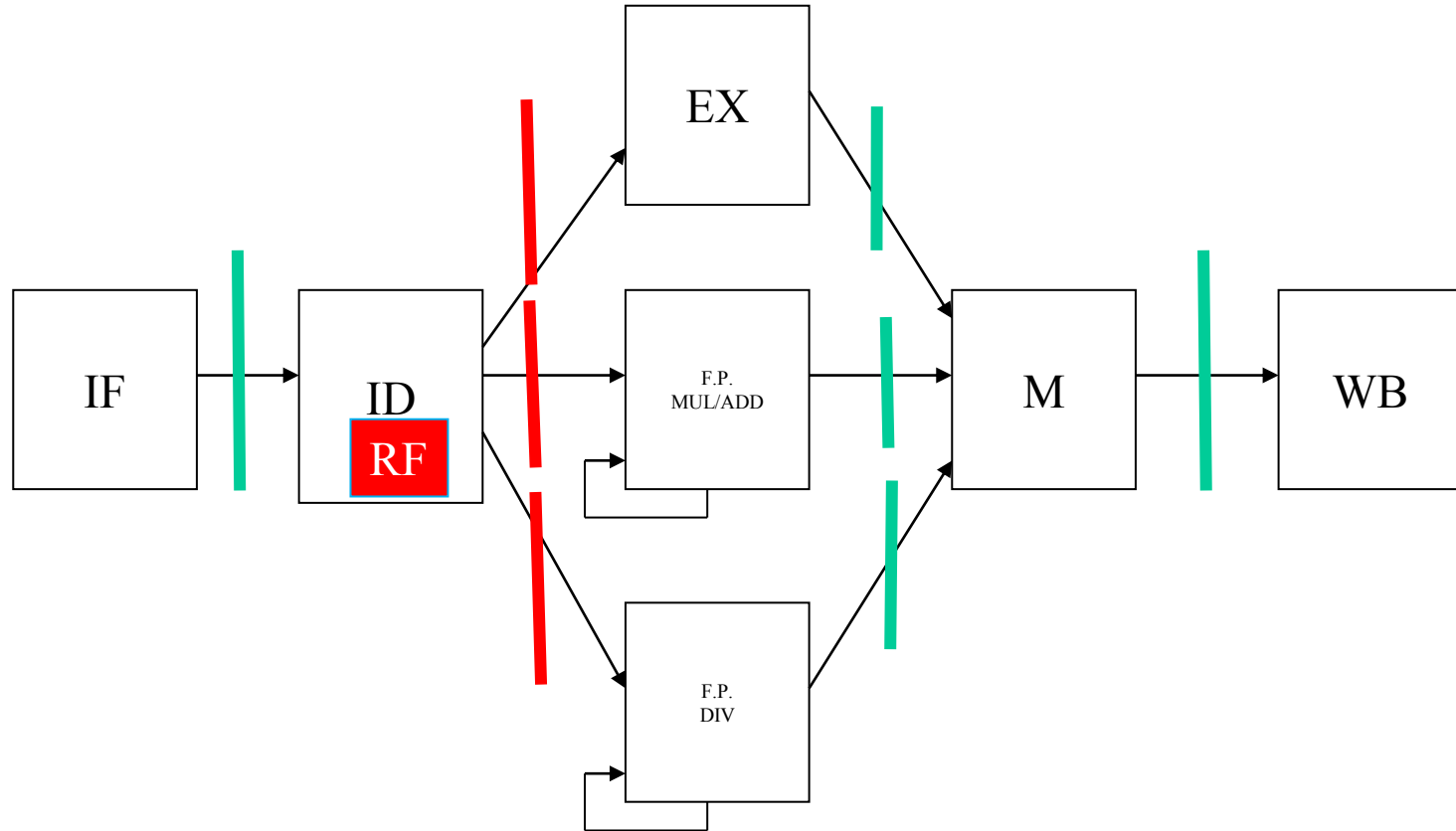
Reorder buffer						
Entry	Busy	Instruction		State	Destination	Value
1	No	f1d	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	f1d	f2,44(x3)	Commit	f2	Mem[44 + Regs[x3]]
3	Yes	fmul.d	f0,f2,f4	Write result	f0	#2 × Regs[f4]
4	Yes	fsub.d	f8,f2,f6	Write result	f8	#2 − #1
5	Yes	fdiv.d	f0,f0,f6	Execute	f0	
6	Yes	fadd.d	f6,f8,f2	Write result	f6	#4 + #2

Reservation stations								
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	fmul.d	Mem[44 + Regs[x3]]	Regs[f4]			#3	
Mult2	Yes	fdiv.d		Mem[32 + Regs[x2]]	#3		#5	

FP register status										
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	...	Yes	Yes

# Pipeline bloccante

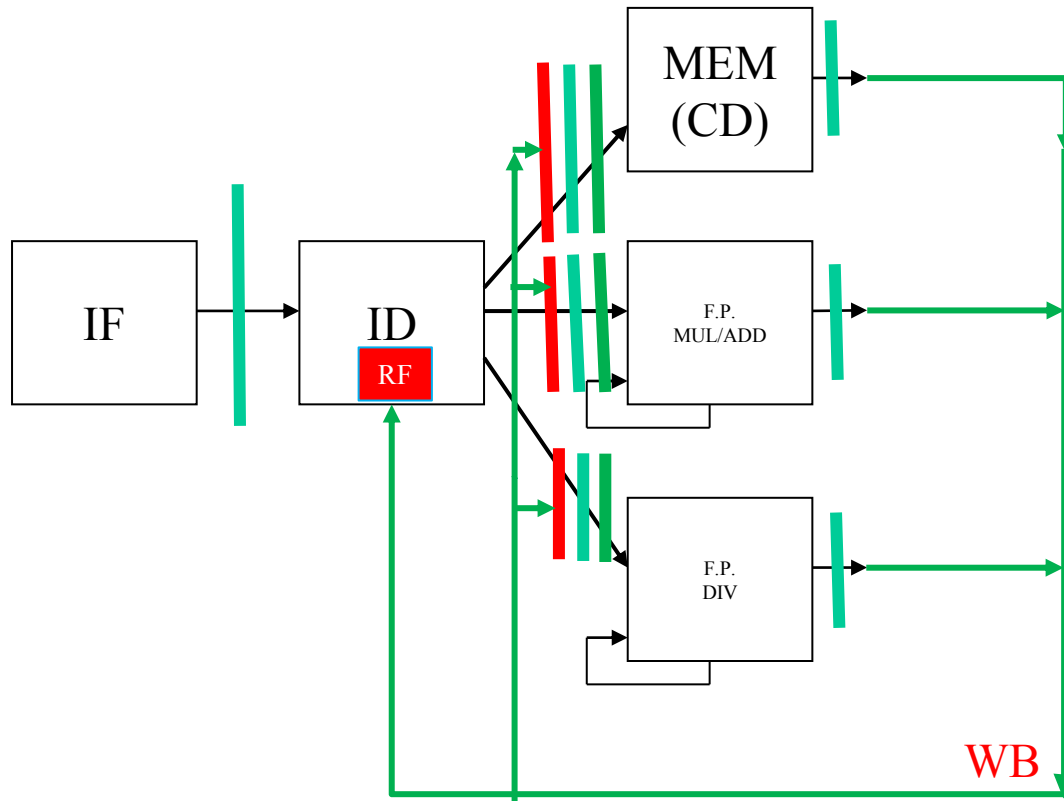
Stalla con alee RAW, di controllo e strutturali





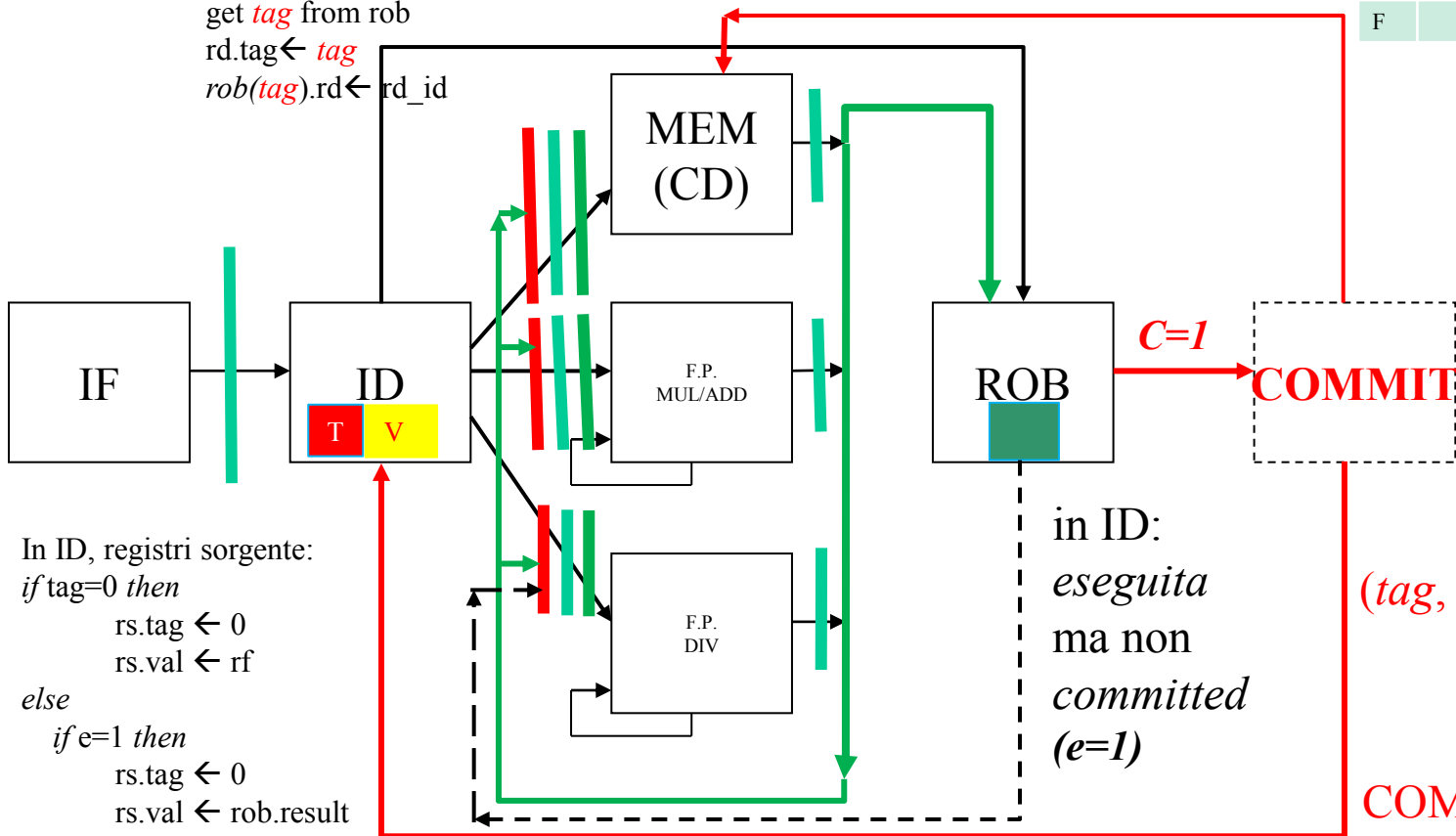
# Pipeline non bloccante senza rob

## Stalla con alee di controllo e strutturali



B/F	tag	rd	OP	valore	e	c
B	1	1	+	5	1	1
B	2		BR		0	0
B	3	1	-	-2	1	0
B	4	1	*	1	0	0
B	5	1	*	1	0	0
F						
F						

Reorder Buffer



(tag, result)

COMMIT

Quando c'è il ROB, si ha che: TAG = (Indice nel ROB)

In ID, registro destinazione

get tag from rob

rd.tag ← tag

rob(tag).rd ← rd\_id

In ID, registri sorgente:

if tag=0 then

rs.tag ← 0

rs.val ← rf

else

if e=1 then

rs.tag ← 0

rs.val ← rob.result

else (e=0)

rs.tag ← tag from rob

in ID:  
eseguita  
ma non  
committed  
(e=1)

# Il reorder buffer (ROB)

- Viene caricato in ordine dallo stadio ID contestualmente con il caricamento della Reservation Station
- Nell'intervallo tra il completamento dell'esecuzione e la fase di commit, il ROB e non il register file mette a disposizione i risultati
- Pertanto:
- Il TAG: (Unit\_ID, RS\_ID) diventa: TAG: (Indice nel ROB)
- Il risultato di una istruzione diventa “impegnativo”, cioè passa ad aggiornare lo stato del “sistema” se:
  - L'istruzione è stata eseguita
  - I risultati di tutte le istruzioni precedenti sono “impegnativi”

# Multiple Issue and Static Scheduling

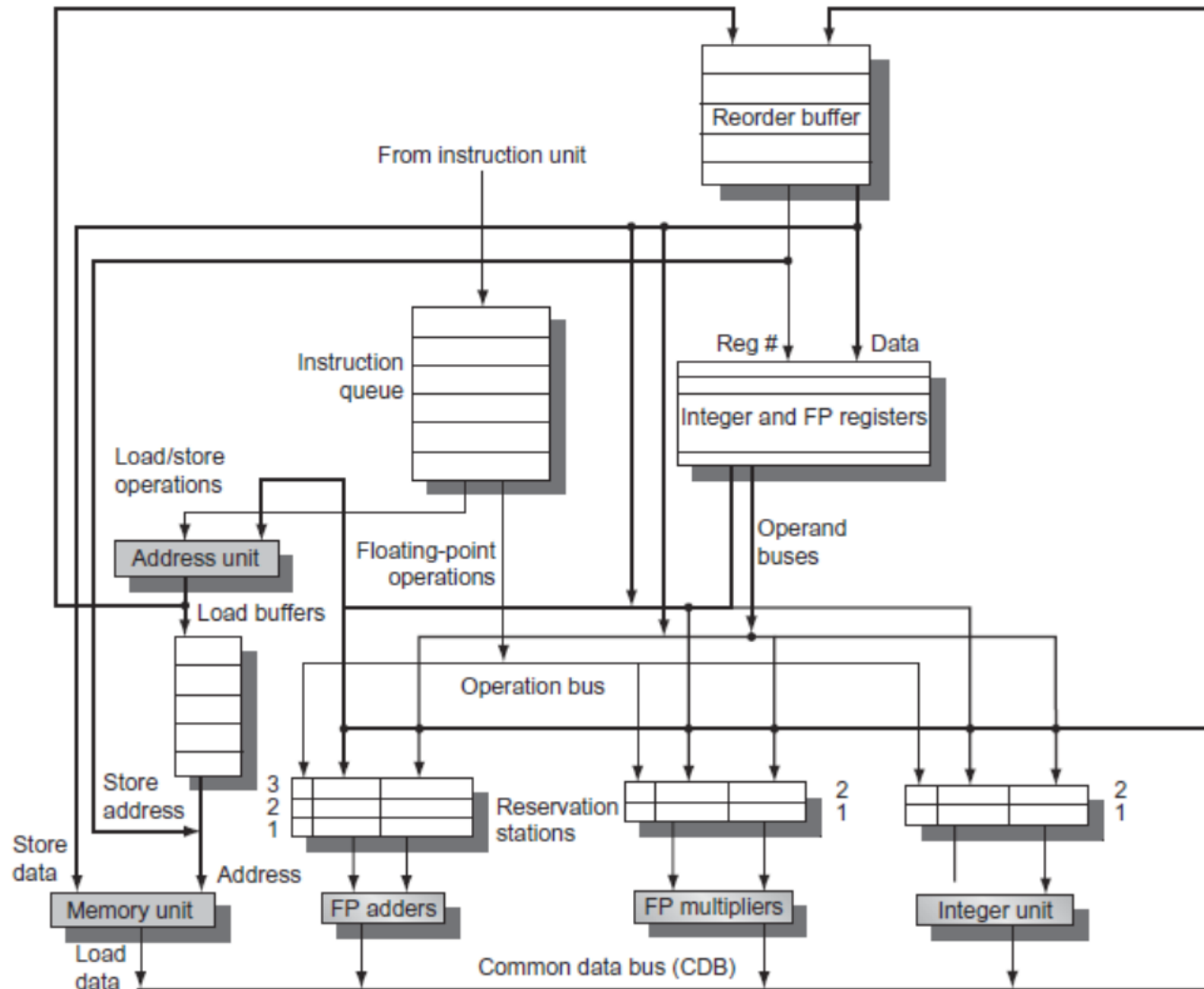
- To achieve  $CPI < 1$ , need to complete multiple instructions per clock
- Solutions:
  - Statically scheduled superscalar processors
  - Dynamically scheduled superscalar processors

fare fetch e decode di più di un'istruzione alla volta.

# Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
- Issue logic is the bottleneck in dynamically scheduled superscalars

# Overview of Design



# Multiple Issue

- Examine all the dependencies among the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Also need multiple completion/commit
- To simplify RS allocation:
  - Limit the number of instructions of a given class that can be issued in a “bundle”, i.e. on FP, one integer, one load, one store

# Example

Loop: ld x2,0(x1)	//x2=array element
addi x2,x2,1	//increment x2
sd x2,0(x1)	//store result
addi x1,x1,8	//increment pointer
bne x2,x3,Loop	//branch if not last



# Example (No Speculation)

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	ld x2,0(x1)	1	2	3	4	First issue
1	addi x2,x2,1	1	5		6	Wait for ld
1	sd x2,0(x1)	2	3	7		Wait for addi
1	addi x1,x1,8	2	3		4	Execute directly
1	bne x2,x3,Loop	3	7			Wait for addi
2	ld x2,0(x1)	4	8	9	10	Wait for bne
2	addi x2,x2,1	4	11		12	Wait for ld
2	sd x2,0(x1)	5	9	13		Wait for addi
2	addi x1,x1,8	5	8		9	Wait for bne
2	bne x2,x3,Loop	6	13			Wait for addi
3	ld x2,0(x1)	7	14	15	16	Wait for bne
3	addi x2,x2,1	7	17		18	Wait for ld
3	sd x2,0(x1)	8	15	19		Wait for addi
3	addi x1,x1,8	8	14		15	Wait for bne
3	bne x2,x3,Loop	9	19			Wait for addi

# Example (Multiple Issue with Speculation)

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	ld x2,0(x1)	1	2	3	4	5	First issue
1	addi x2,x2,1	1	5		6	7	Wait for ld
1	sd x2,0(x1)	2	3			7	Wait for addi
1	addi x1,x1,8	2	3		4	8	Commit in order
1	bne x2,x3,Loop	3	7			8	Wait for addi
2	ld x2,0(x1)	4	5	6	7	9	No execute delay
2	addi x2,x2,1	4	8		9	10	Wait for ld
2	sd x2,0(x1)	5	6			10	Wait for addi
2	addi x1,x1,8	5	6		7	11	Commit in order
2	bne x2,x3,Loop	6	10			11	Wait for addi
3	ld x2,0(x1)	7	8	9	10	12	Earliest possible
3	addi x2,x2,1	7	11		12	13	Wait for ld
3	sd x2,0(x1)	8	9			13	Wait for addi
3	addi x1,x1,8	8	9		10	14	Executes earlier
3	bne x2,x3,Loop	9	13			14	Wait for addi