Dynamic Scheduling Techniques

E. Sanchez

Politecnico di Torino
Dipartimento di Automatica e Informatica

INTRODUCTION

Dynamic scheduling **is a technique aimed at rearranging in hardware the instruction execution order to reduce the pipeline stalls while maintaining data flow and exception behavior.**

Advantages

Dynamic scheduling

- **Identifies some dependencies that are unknown at compile time (e.g., concerning memory cells)**
- **Simplifies the compiler job**
- **Allows the processor to tolerate unpredictable delays**
- **Allows to run the same code on different pipelined processors.**

Example

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

The pipeline stalls after the `fdiv.d` instruction, due to the data dependency between `fdiv.d` and `fadd.d`

The `fsub.d` instruction is also stalled, no matter the fact that it is not involved in any dependency with the previous instructions.

Example

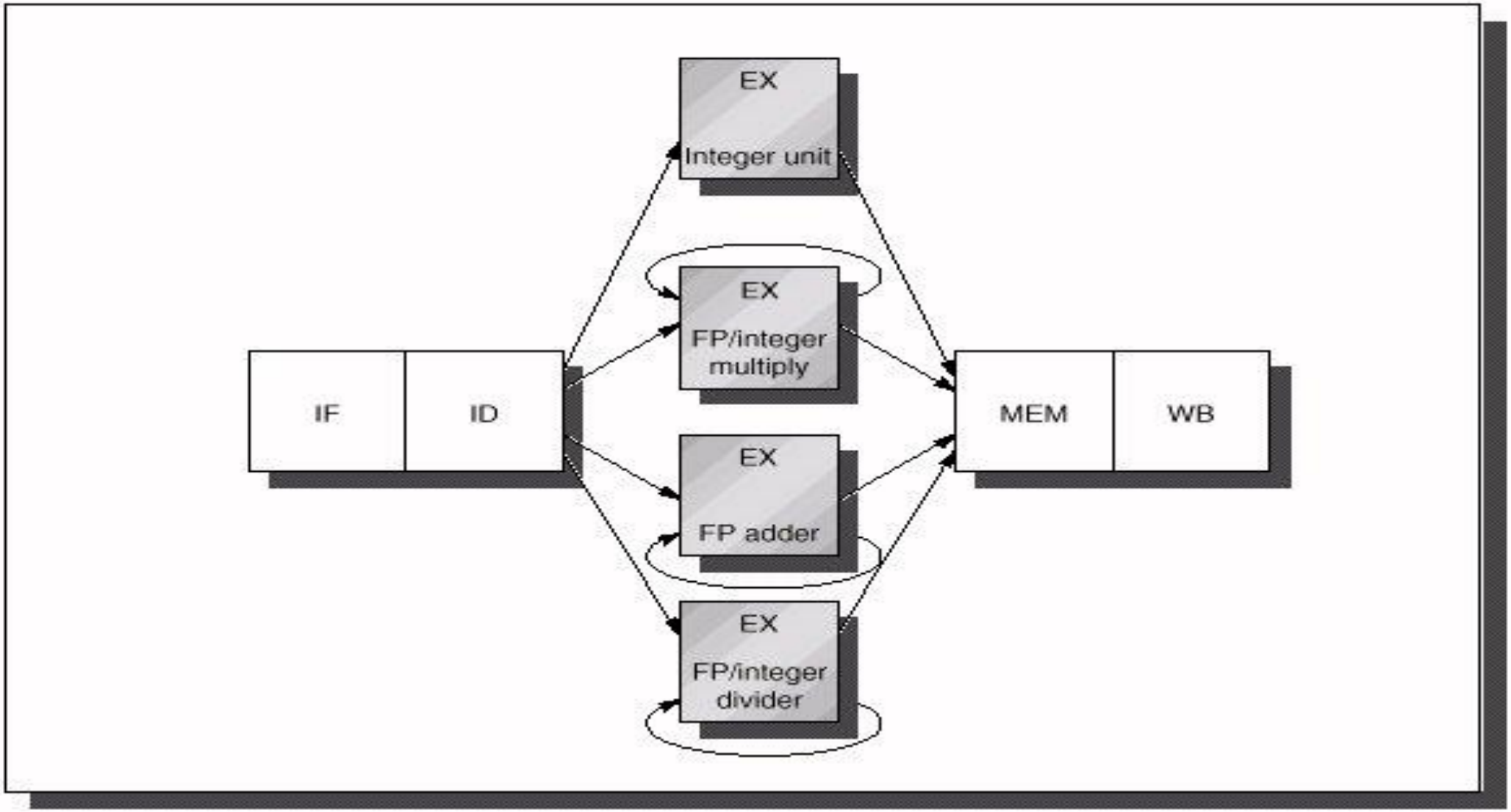
```
fdiv.d  f0, f2, f4
fadd.d  f10, f0, f8
fsub.d  f12, f8, f
```

The pipeline stalls after … due to the data dependency.

The `fsub.d` instruction … ter the fact that it is not involve… with the previous instructions.

We could improve performance by removing the constraint of *in-order instruction execution*.

FP pipelined processor



Out-of-order execution

It may introduce

- WAR hazards
- WAW hazards
- Imprecise exceptions.

WAR hazards

If out-of-order execution is allowed, WAR hazards may arise.

Example

```
fdiv.d  f0, f2, f4
fadd.d  f6, f0, f8
fsub.d  f8, f9, f14
```

If `fsub.d` is executed before `fadd.d`, the antidependency relation is violated, and the result is incorrect.

WAW hazards

If out-of-order execution is allowed, WAW hazards may also arise.

Example

```
fdiv.d  f0, f2, f4
fadd.d  f6, f0, f8
fsub.d  f8, f9, f14
fmul.d  f6, f10, f8
```

If `fmul.d` is executed before `fadd.d`, the output dependency relation is violated, and the result is incorrect.

Exceptions

If out-of-order execution (and completion) is allowed, precise exception handling is impossible.

This may mean that at the time when the exception is raised

- An instruction before that raising the exception still has to be completed, *or*
- An instruction after that raising the exception has been already completed.

In both cases resuming program execution after exception handling is difficult.

Some techniques will be introduced later to solve this problem.

Splitting the ID stage

To support out-of-order execution, the ID stage is split in two parts:

- ***issue*: decode instructions, check for structural hazards**
- ***read operands*: wait until no data hazards, then read operands.**

Splitting the ID stage (II)

The issue stage reads instructions from a register or a queue (written by the IF stage).

Instruction issue is performed in-order.

After the instructions are issued, they may wait for operands, and then (when operands are available) sent to the EX stage.

Instructions can be stalled or bypass each other while in the read operand stage.

Therefore, they can enter execution out-of-order.

EX stage

If the processor includes multiple functional units, multiple instructions may be in execution at the same time.

Instructions can also bypass each other while in the execution stage.

Therefore, they can leave execution out-of-order.

Hardware schemes for dynamic scheduling

- ***Scoreboarding*** **(adopted by CDC 6600)**
- ***Tomasulo's algorithm*** **(adopted by IBM 360/91 and then adopted in many other processors).**

	Instruction Status			
Instruction	Issue	Read operands	Exec complete	Write result
LD F6, 34(R2)	X	X	X	X
LD F2, 45(R3)	X	X	X	X
MULTD F0, F2, F4	X	X	X	X
SUBD F8, F6, F2	X	X	X	X
DIVD F10, F0, F6	X	X	X	
ADDD F6, F8, F2	X	X	X	X

	Function Unit Status								
Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

	Register Result Status							
	F0	F2	F4	F6	F8	F10	F12	F30
FU						Div		

THE TOMASULO APPROACH

Robert Tomasulo was the architect of the IBM 360/91 floating point unit: his ideas were first published in a paper of 1967.

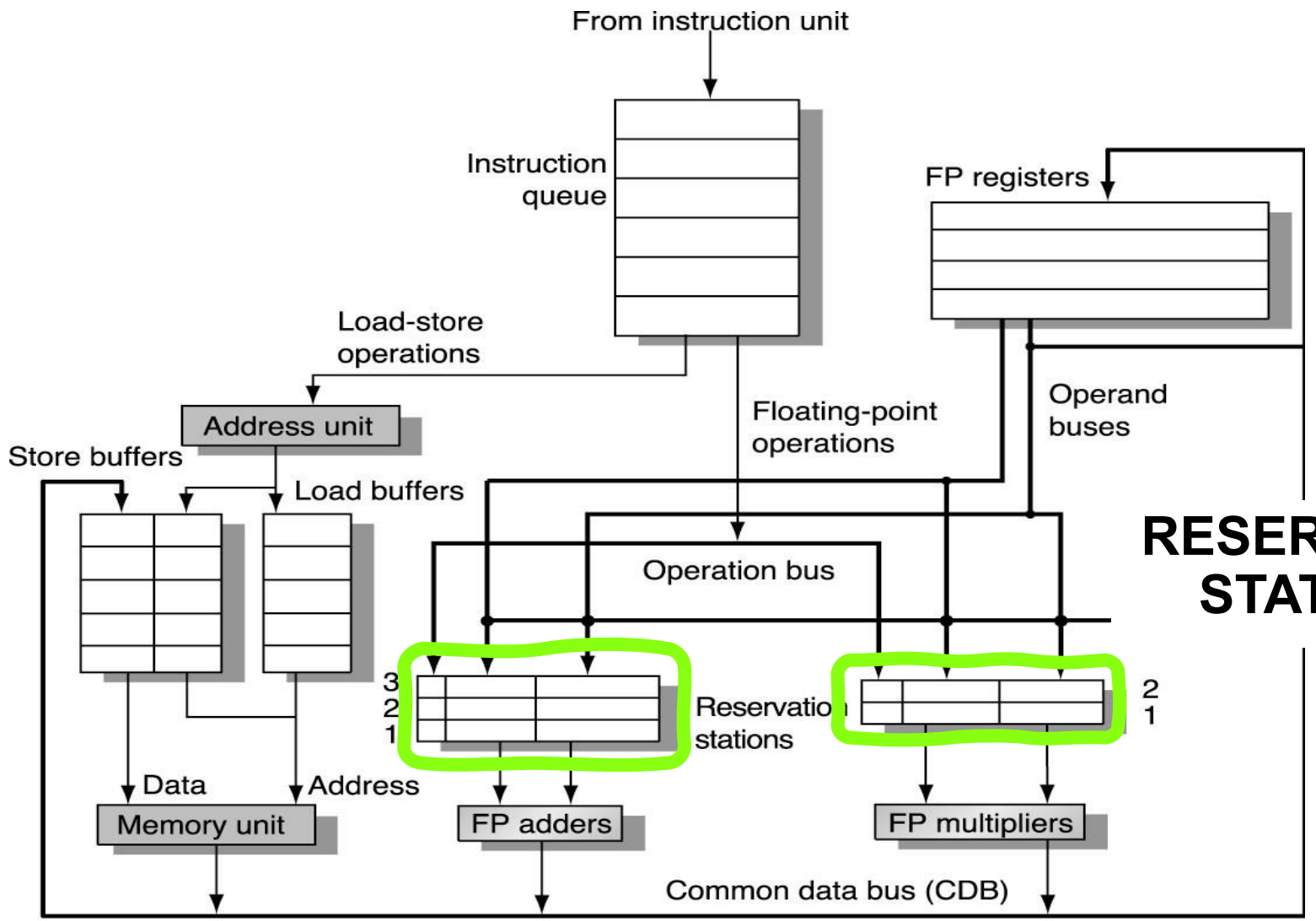
The same ideas were then re-used when the first superscalar processors were built.

The key ideas in the Tomasulo's algorithm are:

-Track the operands availability

-Introduce register renaming.

Tomasulo approach architecture



Reservation stations

They are the key novelty in Tomasulo's approach.

They have several functions:

- they buffer the operands of instructions waiting to execute; operands are stored in the station as soon as they are available
- they implement the issue logic
- they univocally identify an instruction in the pipeline: pending instructions designate the reservation station that will provide them with an input operand.

Register renaming

Each time an instruction is issued, the register specifiers for the pending operands are renamed to the names of the reservation stations in charge of computing them.

This implements a register renaming strategy.

WAW and WAR hazards are eliminated in this way.

Distributed hazard detection and execution control

The reservation stations related to each functional unit control when an instruction can begin execution at that unit.

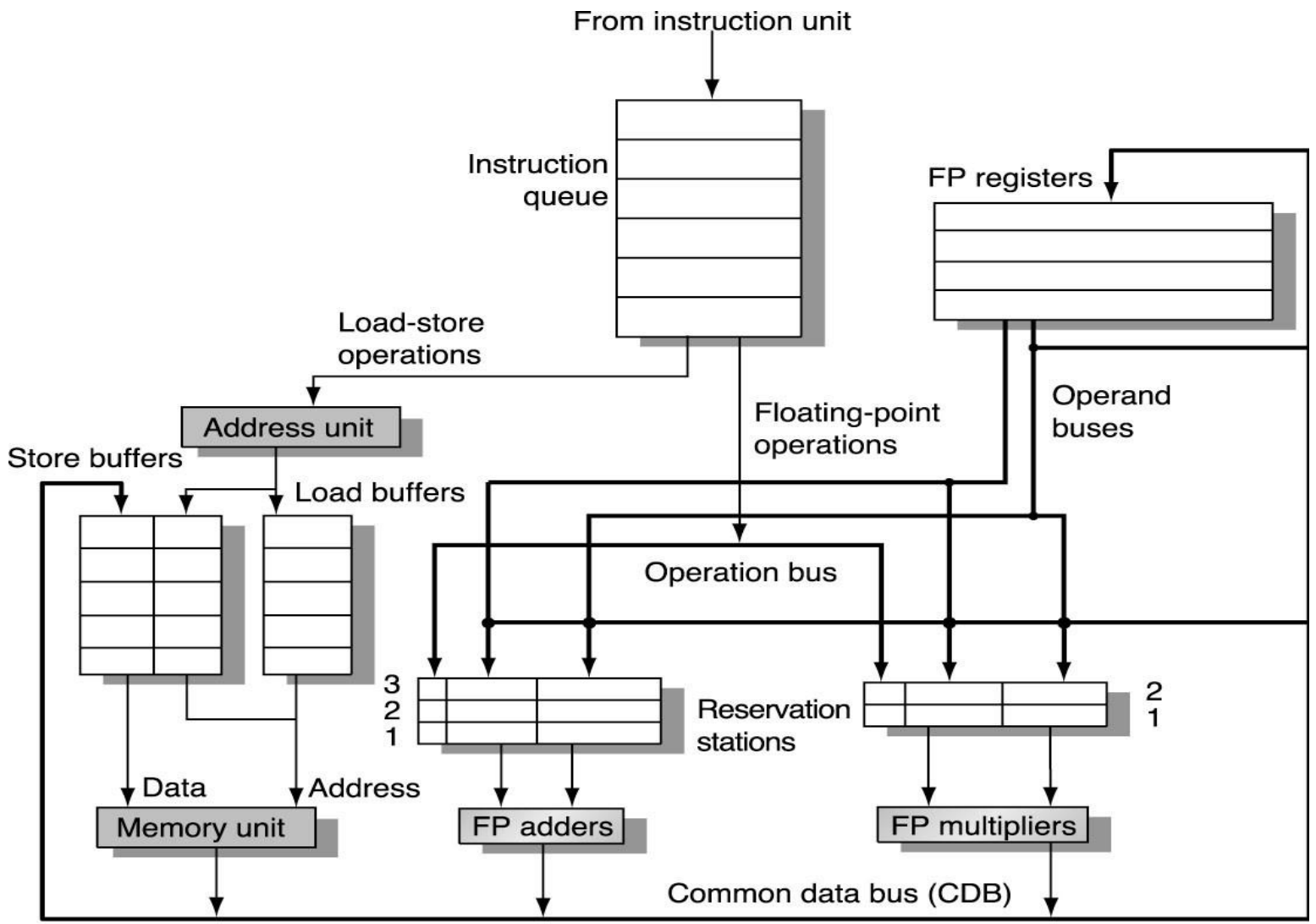
Common Data Bus (CDB)

Results are passed directly to other functional units, rather than going through the registers.

All results from the functional units and from memory are sent to the *Common Data Bus*, which

- **goes everywhere (except to the load buffer)**
- **allows all units waiting for an operand to load it simultaneously when it is available.**

Tomasulo approach architecture



Instruction execution steps

They are

- Issue
- Execute
- Write result.

They can have different length.

Issue

Get an instruction from the instruction queue (implementing a FIFO strategy)

- if no reservation station is available, a structural hazard occurs, and the instruction stalls until a reservation station becomes available
- if there is an empty reservation station, send the instruction to it
  - if the operands are available, they are sent to the reservation station
  - if not, the reservation stations of the functional units responsible for their generation are recorded.

Execution (normal instructions)

- When an operand appears on the CDB, it is read by the reservation unit
- As soon as all the operands for an instruction are available in the reservation unit, the instruction is executed.
- In this way, RAW hazards are avoided.

Execution (load and store instructions)

First step:

- **As soon as the base register is available, the effective address is computed and written in the *load/store buffer***

Second step:

- **Load instructions: execution takes place as soon as memory is available**
- **Store instructions:**
  - **wait for the operand that will be written in memory**
  - **execution takes place as soon as memory is available.**

Execution (branches)

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed.

Speculation may allow to improve this mechanism.

Write result

When the result of an instruction is available, it is immediately written on the CDB, and from there in the registers and functional units waiting for it.

Store instructions also write to memory in this step.

Instruction identifiers

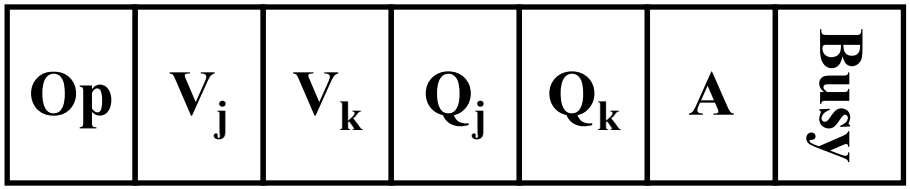
Every reservation station (i.e., instruction to be or being executed) is associated with an identifier.

This identifier also identifies the operand the instruction will produce.

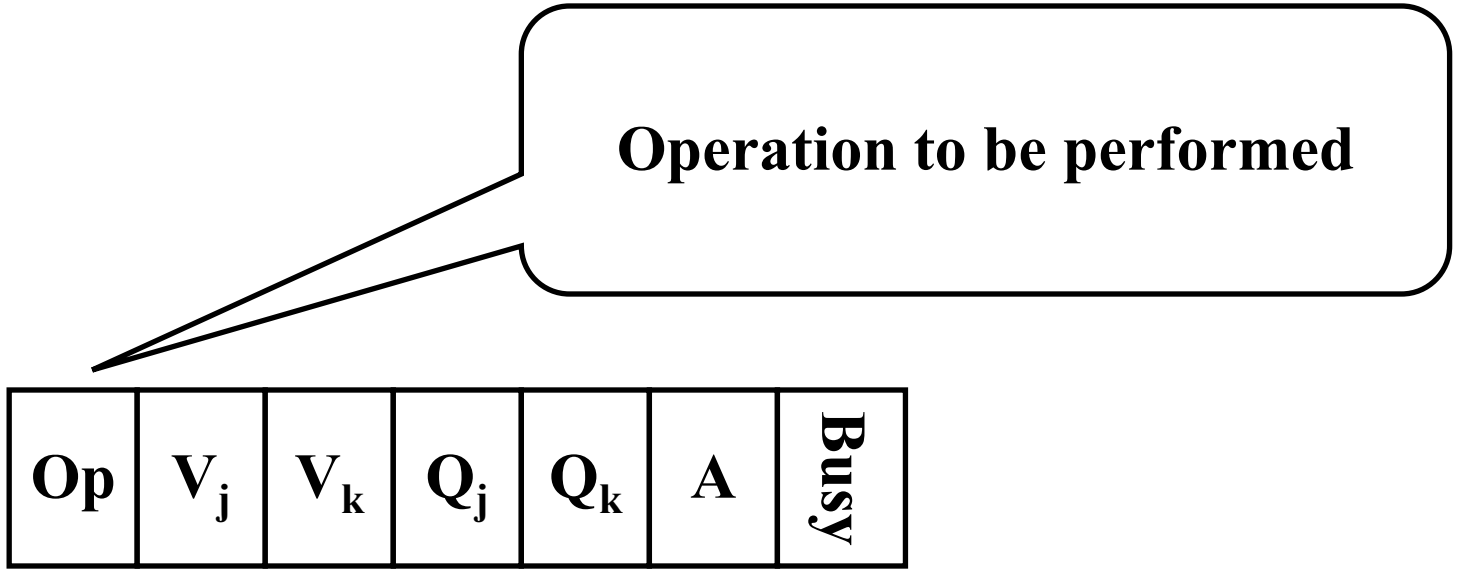
Instructions requiring that operand identify it using the identifier.

Identifiers play the role of names of virtual registers that can be used to implement register renaming.

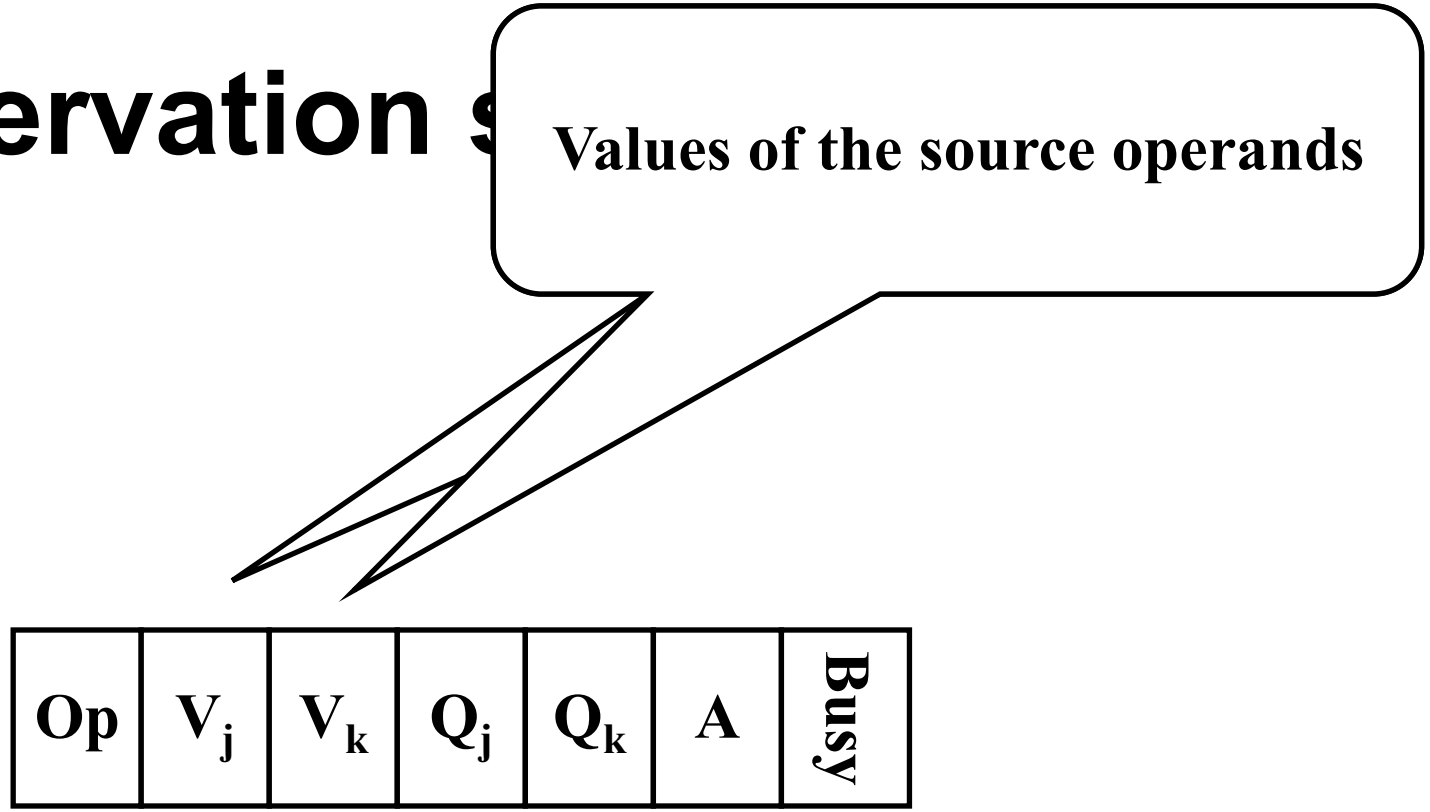
Reservation station fields



Reservation station fields

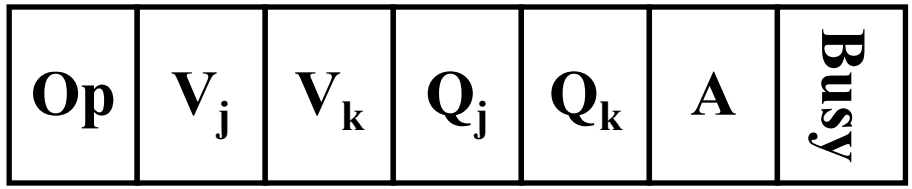


Reservation s



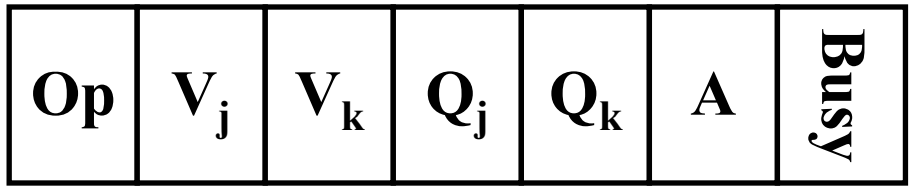
Reservation s

Reservation stations that will produce a source operand (if not available)

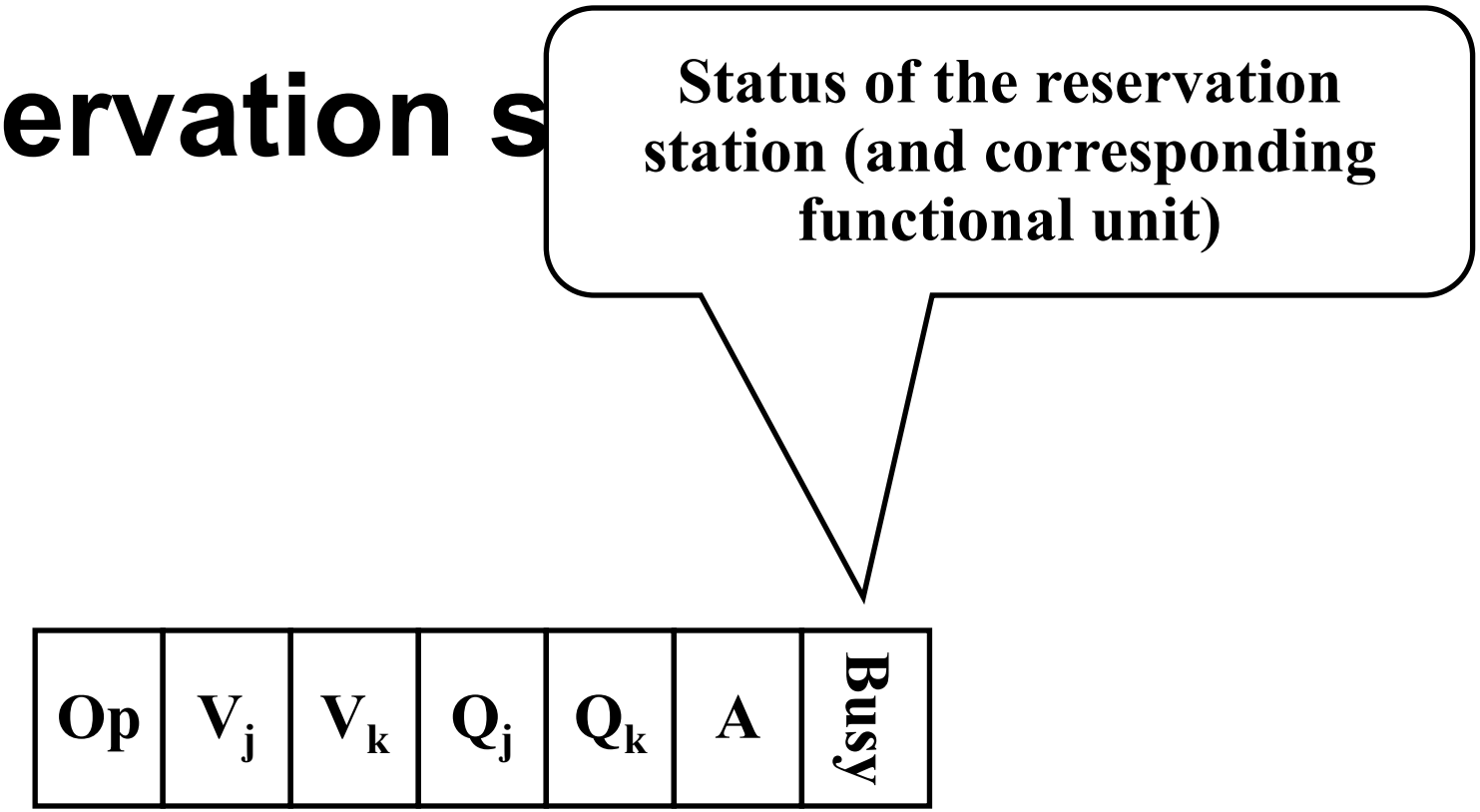


Reservation s

Used in the load/store buffer, only. Stores the immediate field first, and the effective address, then.



Reservation s



Register file

Each element in the register file contains one field Q_i.

Q_i contains the number of the reservation station that contains the instruction whose result should be stored in the register.

If Q_i is null, no currently active instruction is computing a result destined for this register.

Example

Let us consider the following fragment:

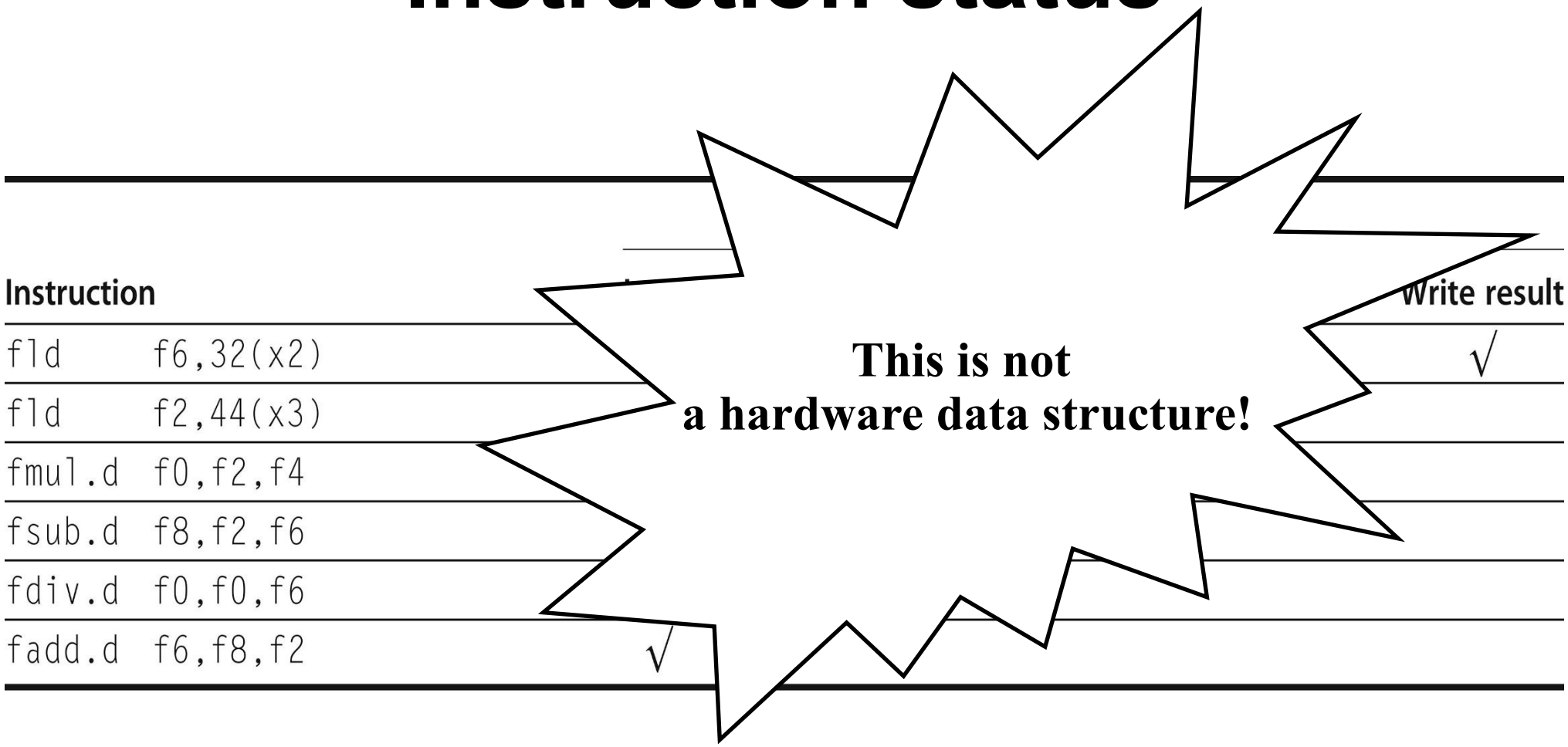
```
1   fld     f6, 34(x2)
2   fld     f2, 45(x3)
3   fmul.d  f0, f2, f4
4   fsub.d  f8, f2, f6
5   fdiv.d  f10, f0, f6
6   fadd.d  f6, f8, f2
```

We will analyze the content of the different data structures when the first `fld` has completed and written its result.

Instruction status

Instruction	Instruction status		
	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`	√		

Instruction status



Reservation stations

	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

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

Advantages

- The hazard detection logic is distributed
- Stalls for WAW and WAR hazards are eliminated.

Disadvantages

- High complexity hardware (including an associative buffer for each reservation station)
- The CDB can be a bottleneck.

Loop handling

Loop unrolling is not required by the Tomasulo architecture.

Loop instructions are naturally performed in parallel.

Example

Let consider the following piece of code:

```
Loop:   fld     f0, 0(x1)
        fmul.d  f4, f0, f2
        fsd     f4, 0(x1)
        addi    x1, x1, -8
        bne     x1,  x2,  loop;  branches  if
x1≠x2
```

We will consider the instant when all the instructions of the first two iterations of the code have been issued, but none of the floating point operations still completed.

We accept speculation and will assume that the hardware always predicts branches as taken.

Instruction status

Instruction		Instruction status			
		From iteration	Issue	Execute	Write result
`fld`	`f0,0(x1)`	1	√	√	
`fmul.d`	`f4,f0,f2`	1	√		
`fsd`	`f4,0(x1)`	1	√		
`fld`	`f0,0(x1)`	2	√	√	
`fmul.d`	`f4,f0,f2`	2	√		
`fsd`	`f4,0(x1)`	2	√		

Reservation stations

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					Regs[x1] + 0
Load2	Yes	Load					Regs[x1] – 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[f2]	Load1		
Mult2	Yes	MUL		Regs[f2]	Load2		
Store1	Yes	Store	Regs[x1]			Mult1	
Store2	Yes	Store	Regs[x1] – 8			Mult2	

Register status

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

Load and store order

Load and store instructions can be executed in any order, provided that they do not refer to the same address.

In this case, by exchanging their order

- if the load is before the store, a WAR hazard may happen
- if the store is before the load, a RAW hazard may happen.

Dynamic disambiguation

To avoid hazards coming from reordering load and store instructions, the processor must

- compute memory addresses in order
- perform some check for every load or store instruction already in the buffers.

Checks for a load instruction

Each time a load is ready to be issued:

- The store buffer is checked for store instructions acting on the same address
- In this case, the load is not sent to the load buffer until the store completes.

Checks for a store instruction

Each time a store is ready to be issued:

- The store and load buffers are checked for store instructions acting on the same address
- In this case, the store is not sent to the store buffer until all the previous load and store instructions complete.