Multiple-issue processors

E. Sanchez

Politecnico di Torino
Dipartimento di Automatica e Informatica

INTRODUCTION

The techniques for reducing the effects of data and control dependencies can be further exploited to obtain a CPI less than one: this requires issuing more than one instruction per clock cycle.

There are two kinds of processors able to do so:

- *superscalar* processors, either statically or dynamically scheduling instructions
- *Very Long Instruction Word* (VLIW) processors.

In both groups the architecture obviously includes multiple functional units.

MULTIPLE-ISSUE STATIC SCHEDULING

Implementing a superscalar processor able to possibly issue several instructions according to the static order defined by the compiler is rather easy and inexpensive.

Statically scheduled superscalar RISC-V version

Two instructions can be issued per clock cycle if:

- one is a load, store, branch, or integer ALU operation
- the other is any FP operation (but load and store, which belong to the first category).

Two instructions (64 bits) are fetched and decoded at every clock cycle.

The two instructions are aligned on a 64-bit boundary and constitute an *issue packet*.

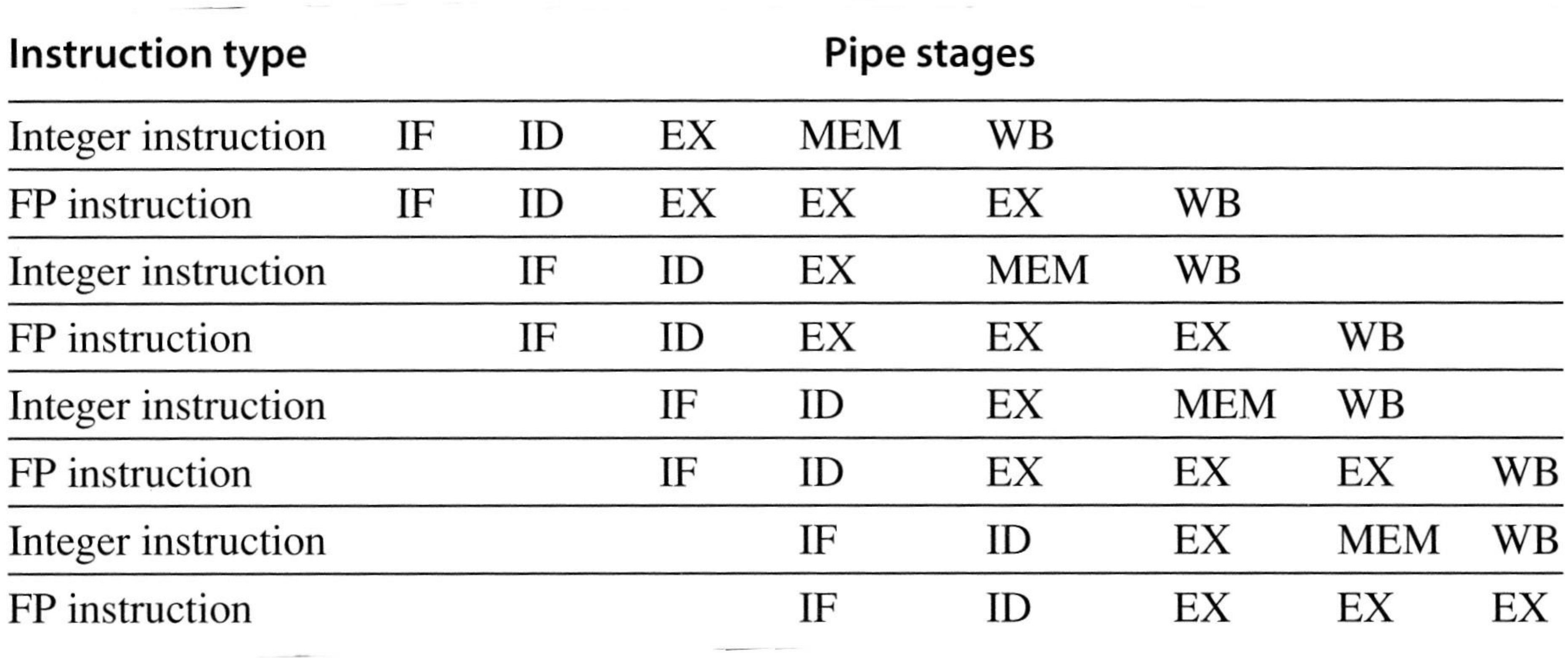
Instruction order within the issue packet

Old superscalar processors required a fixed structure for issue packets (e.g., that the integer instruction is always the first one).

Current processors normally do not have this limitation.

Ideal situation

FP instructions are all assumed to last for 3 clock cycles



Instruction type				Pipe stages				
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	WB	
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

Instruction fetching

At each clock cycle 2 instructions (64 bits) must be read from the instruction memory (i.e., from the cache).

If the two instructions belong to different cache blocks, several processors fetch one instruction, only.

The issue packet must only contain one branch instruction.

FP Units

In order to obtain a real benefit, the floating point units should be either pipelined, or multiple and independent.

FP Register contention

When the first instruction is a FP load, store, or move, there is a possible contention for a FP register port.

Possible solutions are:

- forcing the first instruction to be executed by itself
- giving the FP register file an additional port.

Possible RAW Hazard

When the first instruction is a FP load, store, or move, and the second reads its result, a RAW hazard is also possible.

In this case the second instruction must then be delayed by one clock cycle.

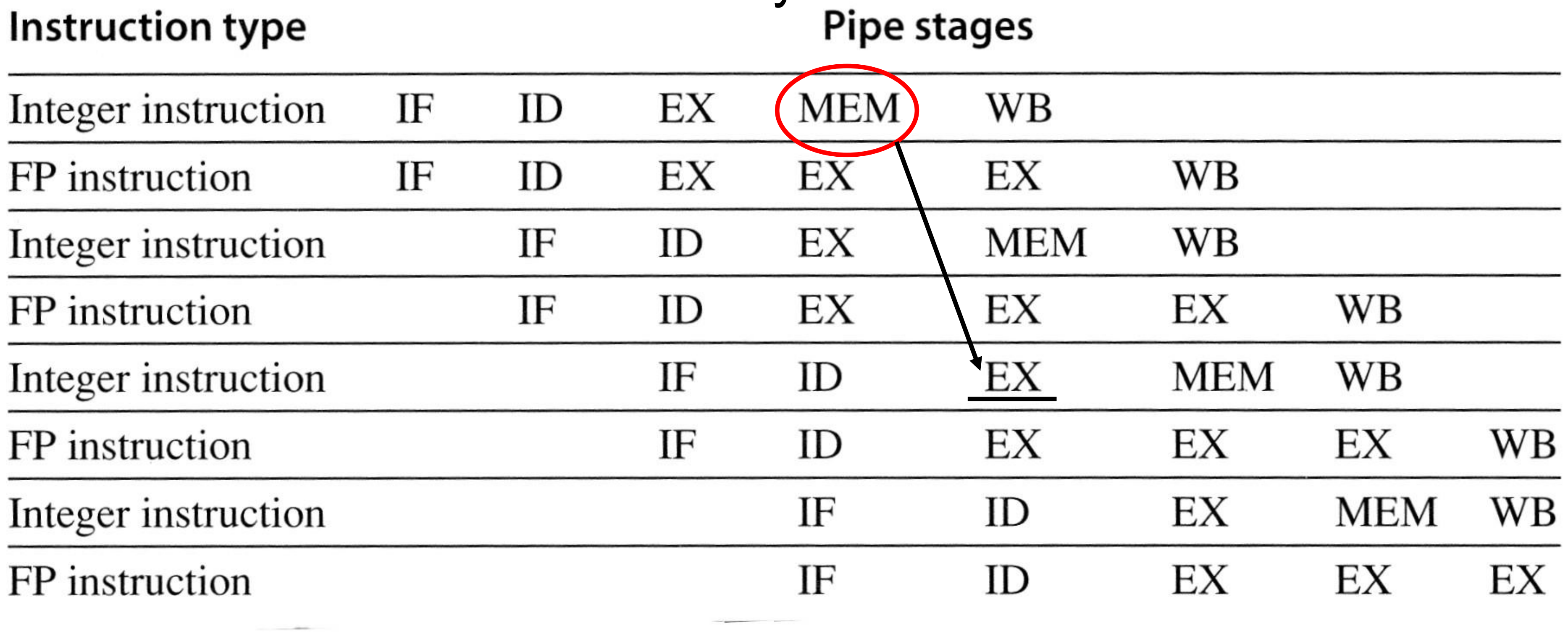
Data and Branch Delay

In the RISC-V pipeline, load has a latency of one clock cycle, which means that in the superscalar pipeline the result of a load instruction can not be used on the same clock cycle but on the next one.

Therefore, in the static RISC-V superscalar version the load delay slot (as well as the branch delay slot) becomes equal to *three* instructions.

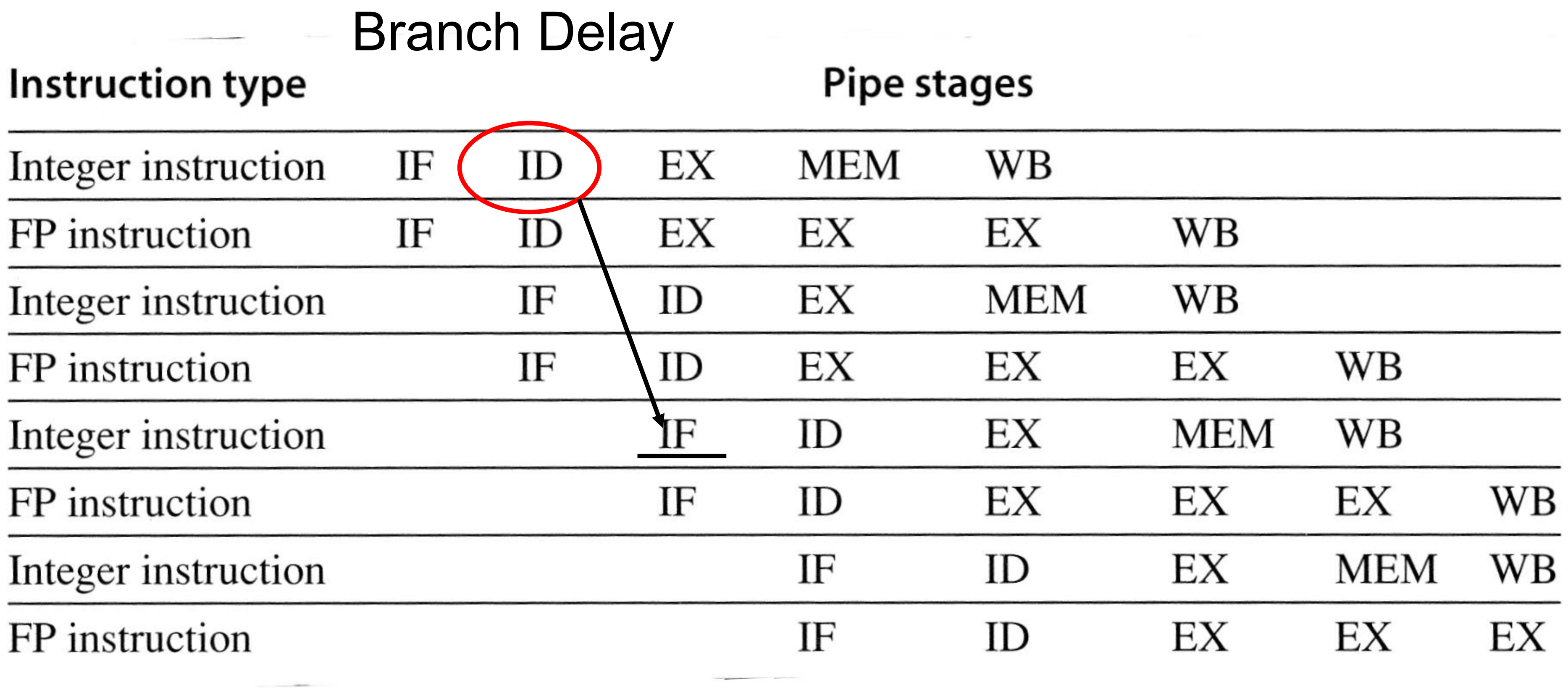
Data and Branch Delay

Load Delay



Instruction type	Pipe stages							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	WB	
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

Data and Branch Delay



Branch Delay

Instruction type	Pipe stages							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	WB	
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

Conclusions

Static multiple-issue scheduling is mainly adopted by processors for the embedded market, such as those by ARM, RISC-V and MIPS.

MULTIPLE-ISSUE DYNAMIC SCHEDULING

It can be obtained by adopting a scheme similar to the Tomasulo one.

To make the implementation easier, instructions are never issued to the ROB and Reservation Stations out-of-order.

CDB criticality

In some clock cycles, more than one instruction may be ready to write on the CDB, that can only service one instruction at a time.

For this reason, duplicating the CDB can provide higher performance, at the cost of some area overhead.

Example

Consider the following code

```
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
```

Let us suppose that up to two instructions can issue and commit per clock and consider the two cases:

1. Without speculation
2. With speculation.

Case 1

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`

Case 2

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`

Performance evaluation

In the first case the first 3 iterations require more than 19 clock cycles.

In the second one, they require 14 clock cycles.