Hardware-based Speculation

M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica

1

Introduction

Hardware-based speculation is a technique for reducing the effects of control dependences in a multiple-issue processor.

If a processor supports branch prediction with dynamic scheduling, it *fetches* and *issues* instructions, as if the branch prediction was always correct.

If a processor supports hardware-based speculation, it also *executes* them.

2

Hardware-based speculation

It combines three ideas:

- **Dynamic branch prediction**
- **Speculation**
- **Dynamic scheduling.**

In such a way, the processor implements *data flow execution*: operations execute as soon as their operands are available.

3

Examples

The following processors implement hardware-based speculation resorting to Tomasulo's architecture:

- **PowerPC 603/604/G3/G4**
- **MIPS R10000/R12000**
- **Intel Pentium II/III/4**
- **Alpha 21264**
- **AMD K5/K6/Athlon.**

4

Architecture

The basic Tomasulo's architecture is adopted, extending it to support speculation.

There are two different steps in instruction execution

- **the computation of results and their bypassing to other instructions**
- **the update of register file and memory, which is only performed when the instruction is no longer speculative (*instruction commit*); in this way, in-order commitment is implemented.**

ReOrder Buffer (ROB)

It is the data structure containing the instruction results while the instruction didn't commit yet.

It provides additional virtual registers and integrates the store buffer existing in the original Tomasulo's architecture.

ROB and register file

In Tomasulo's architecture, already computed results are read from the register file.

With speculation, data may be read
- **From the ROB, if the producing instruction didn't commit yet**
- **from the register file, otherwise.**

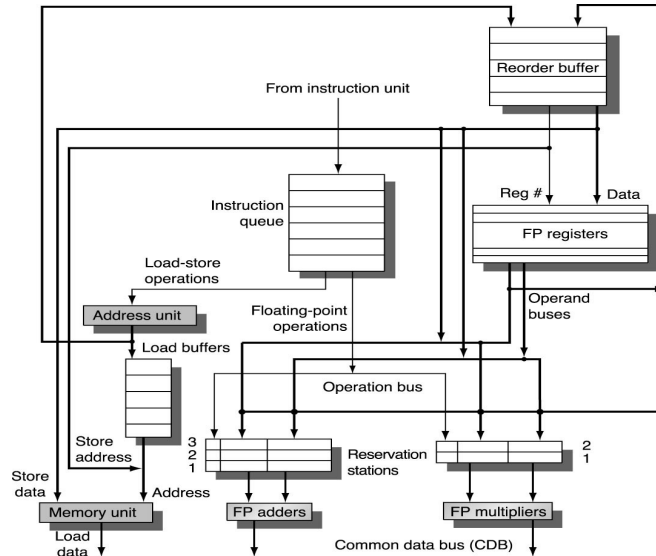# ROB fields

Each entry in the ROB has four fields:
- *Instruction type*: branch, store, or register
- *Destination*: register number, or memory address
- *Value*: contains the value when the instruction has completed but still didn't commit
- *Ready*: indicates whether the instruction completed its execution.

Architecture

9

Instruction Execution steps

- **Issue**
- **Execute**
- **Write result**
- **Commit**

Issue (or Dispatch) Step

- **An instruction is extracted from the instruction queue if there is**
    - **an empty reservation station *and***
    - **an empty slot in the reorder buffer.**
- **If this is not the case, the instruction issue is stalled.**
- **The operands for the instruction are sent to the reservation station, if they are in the register file or in the reorder buffer.**
- **The number of the reorder buffer entry for the instruction is sent to the reservation station to tag the instruction (and its results, when they will be written on the CDB).**

11

Execute Step

- **The instruction is executed as soon as all the required operands are available.**
- **This avoids any RAW hazard.**
- **Operands are possibly taken from the CDB as soon as another instruction produces them**
- **The length of this step varies depending on the instruction type (e.g., 2 for load instructions, 1 for integer instructions).**

12

Write Result Step

- **As soon as it is available, the result is put on the CDB (together with the tag identifying the instruction) and sent to the reorder buffer.**
- **Any reservation station waiting for the result reads it.**
- **The reservation station entry is marked as available.**

Commit (or Completion) Step

The reorder buffer is ordered according to instructions original order.

As soon as one instruction reached the head of the buffer

- if it is a mispredicted branch, the buffer is flushed, and the execution is restarted with the correct successor of the instruction
- otherwise, the result is written in the register or in memory (in case of a store)
- in both cases, the reorder buffer entry is marked as free.

The reorder buffer is implemented as a *circular buffer*.

WAW and WAR hazards

They can not arise, since memory updating occurs in-order (i.e., when a store reaches the top of the ROB).

RAW hazards through memory

They are prevented by

- not allowing a load to initiate its second step if any active ROB entry occupied by a store has a Destination field matching the A field of the load and
- enforcing the program order while computing the effective address of a load wrt all earlier store instructions.

Example

Let consider the following code

```
L.D     F6, 34(R2)
L.D     F2, 45(R3)
MUL.D   F0, F2, F4
SUB.D   F8, F6, F2
DIV.D   F10, F0, F6
ADD.D   F6, F8, F2
```

Assume the following latencies for the FP functional units:

- **add: 2 clock cycles**
- **multiply: 10 clock cycles**
- **divide: 40 clock cycles.**

The following slides report the content of the data structures when the `MUL.D` instruction is ready to be committed.

17

Situation

- **Only the two `L.D` instructions have been already committed**
- **The `SUB.D` and `ADD.D` instructions already completed, but still didn't commit, because they are waiting for the completion of `MUL.D`**
- **The `DIV.D` is being executed.**

18

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	No	MULT	Mem[45+Regs[R3]]	Regs[F4]			#3
Mult2	Yes	DIV		Mem[34+Regs[R2]]	#3		#5

19

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	No	L.D F6,34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	No	L.D F2,45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	Yes	MUL.D F0,F2,F4	Write result	F0	#2´Regs[F4]
4	Yes	SUB.D F8,F6,F2	Write result	F8	#1-#2
5	Yes	DIV.D F10,F0,F6	Execute	F10	
6	Yes	ADD.D F6,F8,F2	Write result	F6	#4+#2

20

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

Store instructions

They write to memory when they commit, only.

Therefore, their input operand is required when they commit, rather than in the Write Result stage.

This means that the ROB should have a further field, specifying where the input operand for each store instruction should come from.

Exception Handling

Exceptions are not executed as soon they are raised, but they are stored in reorder buffer.

When the instruction is committed, the possible exception is executed, and the following instructions flushed from the buffer.

If the instruction is flushed from the buffer, the exception is ignored.

Fully *precise exception* handling is thus supported.

Example

Consider the following code

```
Loop:   LD      R2, O(R1)
        DADDIU  R2, R2, #1
        SD      R2, O(R1)
        DADDIU  R1, R1, #4
        BNE     R2, R3, Loop
```

Let us suppose that up to two instructions can issue and commit per clock and consider the two cases:
- **Without speculation**
- **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	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	Execute directly
1	BNE	R2,R3,LOOP	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#4	5	8		9	Wait for BNE
2	BNE	R2,R3,LOOP	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#4	8	14		15	Wait for BNE
3	BNZ	R2,R3,LOOP	9	19			Wait for DADDIU

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	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#4	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#4	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#4	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

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.

Speculating expensive events

When a time-expensive event (e.g., second-level cache miss, TLB miss) occurs speculatively, some processors wait for its execution until the event is no more speculative.

On the other side, low-cost events (e.g., first-level cache miss) are normally executed speculatively.