UT 2. Pipelined Computers

Tema 2.4 Dynamic and speculative instruction scheduling

A. Doménech, J. Duato, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

Bibliography

John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Contents

Principles

Hardware increases ILP by reordering instructions at runtime:

- Independent instructions are simultaneously executed in the pipelined unit
- Dependent instructions are sequentially executed

Until now, when instruction *i* stalls, no following instruction *j* can proceed, even when *j* is *independent* from those under execution and the *operator* required by *j* is *idle*.

Example:

```
  DIV.D F0,F2,F4     IF ID DIV DIV ... DIV WB
i ADD.D F10,F0,F8       IF ID  ID  ... ID  A1  A2 ...
j MUL.D F12,F8,F14         IF  IF  ... IF  ID  M1 ...
```

Principles (cont.)

Key idea

Hardware must be able to issue instructions following the stalled one → instruction execution order is dynamically modified, thus avoiding stalled instructions to affect the following ones.

Advantages and drawbacks

Advantages:

- Simplifies the design of the compiler.
- Efficient resolution of dependencies that are unknown at compile time (like those generated between instructions involving data in memory, such as (`S.D ...,20(R1)` and `L.D .., 30(R2)` when R1=R2+10)...).
- Enables the efficient execution of any code, despite any existing optimization for another pipelined instruction unit → efficient binary compatibility

Drawbacks:

- *Hardware* becomes more complex.

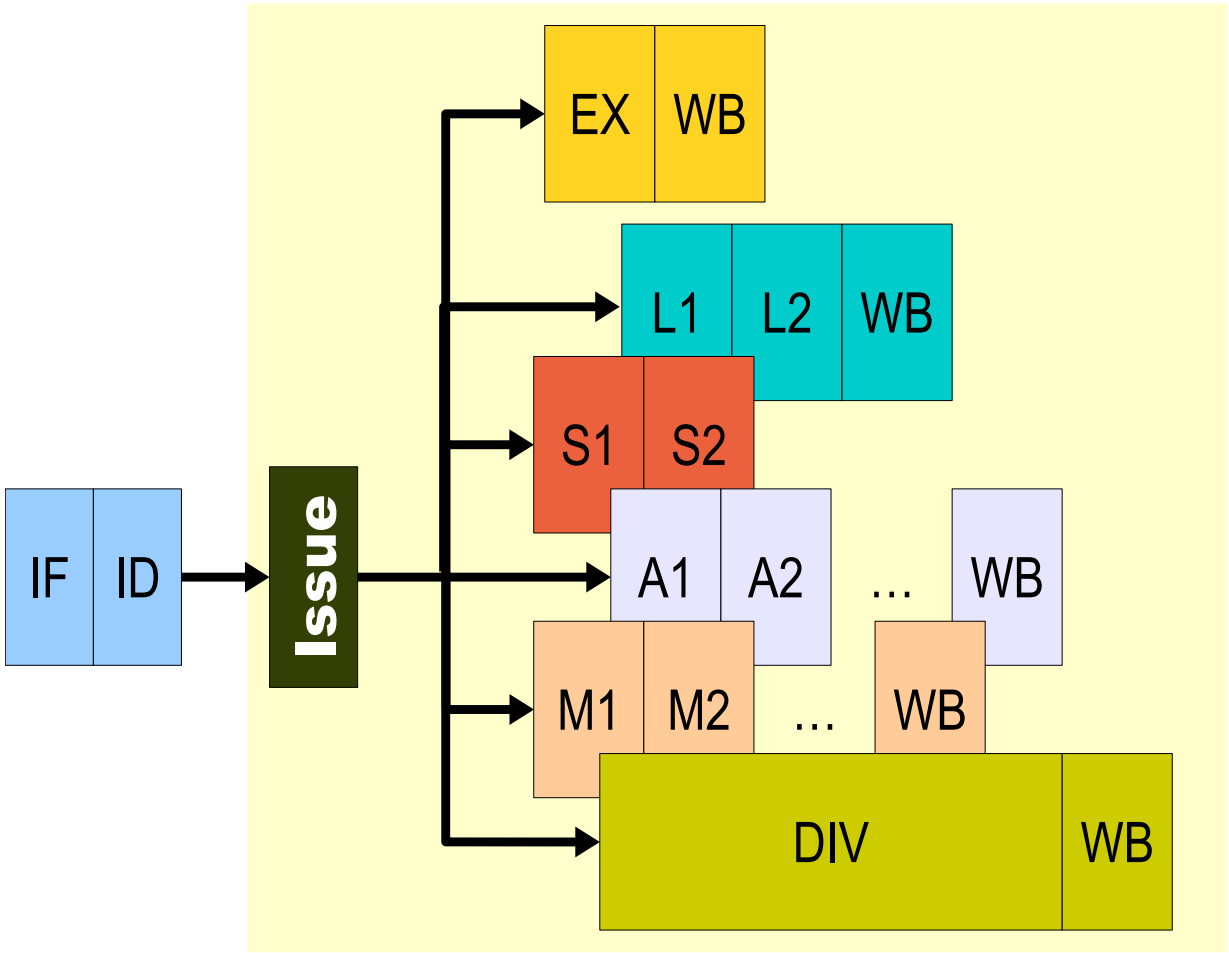
Contents

Goals

- Preventing instruction stalls at ID stage → CPI ≈ 1
- Simultaneous execution of independent instructions
- Correct detection and management of dependent instructions
- Allow out-of-order execution (independent instructions should pass stalled ones)

⇒ **Tomasulo's Algorithm**
(developped by Robert M. Tomasulo in 1967 for the IBM 360/91)

Modification of the pipelined instruction unit



Comments

1. *Issue* stage. When the ID stage decodes a multicycle instruction, the instruction is sent to the *Issue* stage
   - If the operator is available and all the instruction operands are ready, the instruction is issued
   - If the operator required by the instruction is not available, the instruction stalls
   - If any operand is not available (due to a data dependency with another instruction), the instruction is stalled
     - Where are dependent instructions stalled?
     - How are they resumed when the dependency disappears?

Comments (cont.)

2. For the sake of effectiveness, a load/store operator is incorporated to the design
   - Cache access time may be longer than one cycle
   - *Cache* misses only affect this operator, without stalling all the following instructions. In addition, when a *cache* miss occurs → longer access time
   - Dynamic instruction scheduling also applies to load and store instructions
   - Detection of dependencies in memory access instructions
     Examples:

```
S.D F2, 30(R2)
...
L.D F0, 20(R1)
```

     It has a data dependency when R1=R2+10

Where are dependent instructions stalled?

- At the *Issue* stage → this stops instruction decoding

```
DIV.D F0,F2,F4     IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8       IF ID I   I   ... I   A1  A2 ...
MUL.D F12,F8,F14         IF ID  ID  ... ID  I   M1 ...
```

  → This IS NOT dynamic instruction scheduling

- At the corresponding operator

```
DIV.D F0,F2,F4     IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8       IF ID I   A1  ... A1  A1  A2 ...
MUL.D F12,F8,F14         IF ID  I   M1  ...
```

  → OK

Where are dependent instructions stalled? (cont.)

What if the following instruction requests the *same* operator?

```
DIV.D F0,F2,F4     IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8       IF ID I   A1  ... A1  A1  A2 ...
ADD.D F12,F8,F14         IF ID  I   ... I   I   A1 ...
```

→ this IS NOT dynamic instruction scheduling

→ `ADD.D F10,F0,F8` takes the operator but it is not able to use it !

- At a data structure associated with the corresponding operator (**Reservation station**)

```
DIV.D F0,F2,F4  IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8    IF ID I   a1  ... a1  A1  A2 ...
ADD.D F12,F8,F14      IF ID  I   A1  ...
```

→ OK

A physical operator plus the reservation stations offers serveral *virtual operators*.

How are instructions resumed when the dependency disappears?

1. At the *Issue* stage:
   - The instruction is copied to a virtual operator, configuring the path to be followed by the instruction result, ”marking” the corresponding destination register.
   - If any of the operands is not available, the path to be followed by those operands from their current location to the corresponding virtual operator is configured.
2. When all the operands for an issued instruction have reached the corresponding virtual operator and the physical operator is free, the instruction is executed.
3. When the instruction ends its execution (*Writeback* stage), its result is forwarded to the configured instruction destination according to the path defined in 1.

Contents

Considerations

- Dynamic instruction scheduling requires keeping, during the execution of instructions, a representation of unsolved data dependencies.
- These dependencies relate virtual operations among them and with register files.
- Each time an instruction is issued (*Issue* stage), new dependencies are added to the graph.
- Each time an instruction traverses the *Writeback* stage, it broadcasts its result, thus solving some dependencies, which are then deleted from the graph.

An example

Instructions diagram with dynamic scheduling

	1	2	3	4	5	6	7	8	9	10	
`ADD.D F4,F2,F0`	IF	ID	I	A1	A2	A3	WB				
`MUL.D F6,F4,F0`		IF	ID	I	→	→	→	M1	M2	M3	...
`L.D F4,x`			IF	ID	I	L1	L2	WB			
`MUL.D F6,F6,F4`				IF	ID	I	→	→	→	...	

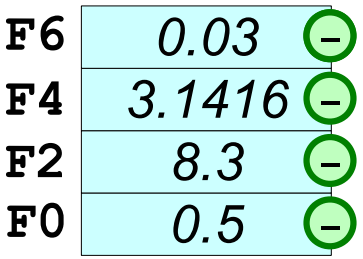
- Symbol → denotes that the instruction is waiting without blocking neither the decoding or issue of instructions.
- The instruction waits (during →) in a virtual operator.
- When an instruction reaches the WB stage, those instructions waiting for the instruction result continue their execution.

Example of dynamic instruction scheduling – Initial state

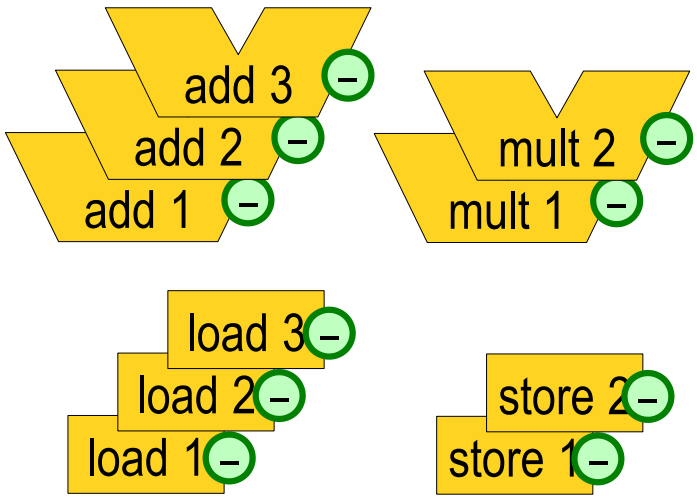
We have ...

- a set of identified instructions

```
1  ADD.D F4,F2,F0
2  MUL.D F6,F4,F0
3  L.D F4,x
4  MUL.D F6,F6,F4
```



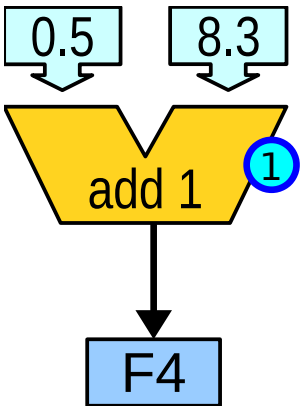
- a system of labels (1, 2, 3 and 4) to denote when operators and registers are related to an instruction and when they are free (–)
- a set of free registers with their initial value
- a set of free (virtual) operators



Example of dynamic instruction scheduling – 3rd cycle

		1	2	3	4	5	6	7	8	9	10
1	`ADD.D F4,F2,F0`	IF	ID	I							
2	`MUL.D F6,F4,F0`		IF	ID							
3	`L.D F4,x`			IF							
4	`MUL.D F6,F6,F4`										

Graph:



Reg. file:

F6	*0.03*	–
F4	*3.1416*	1
F2	*8.3*	–
F0	*0.5*	–

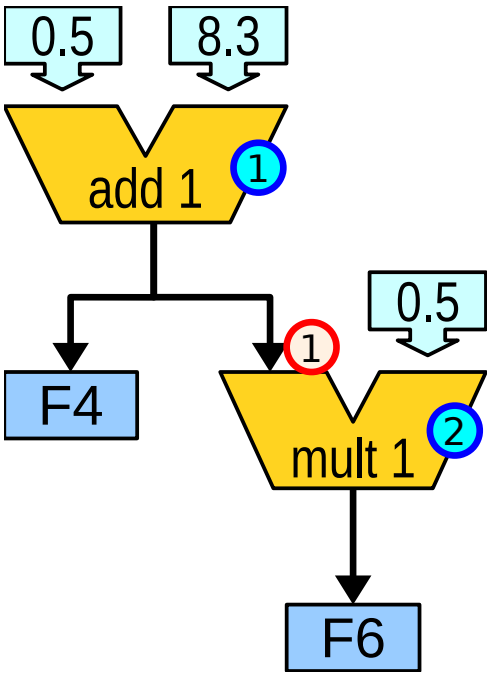
Labels

- Instruction 1 is issued and it takes an operator.
- Source registers F2 and F0 are free (–) and their content is sent to the operator.
- Destination register F4 becomes busy. Its content is no longer valid.
- It must be noted that the physical operator is free, so the instruction can start its execution.

Example of dynamic instruction scheduling – 4th cycle

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F2,F0`	IF	ID	I	A1						
②	`MUL.D F6,F4,F0`		IF	ID	I						
③	`L.D F4,x`			IF	ID						
④	`MUL.D F6,F6,F4`				IF						

Graph:



Reg. file:

F6	*0.03*	②
F4	*3.1416*	①
F2	*8.3*	–
F0	*0.5*	–

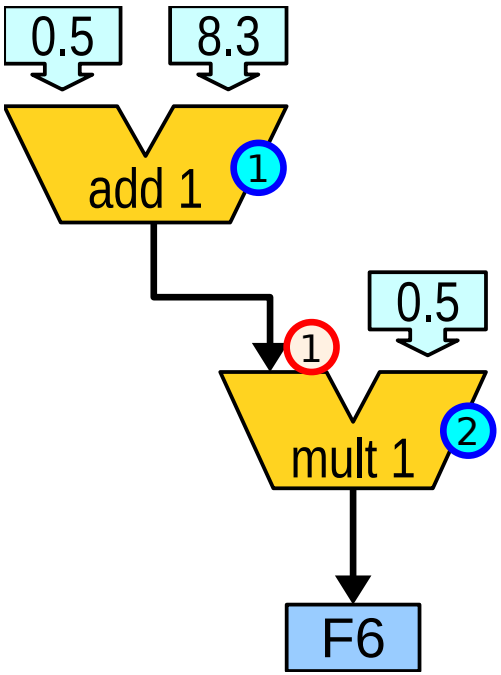
Dependence detection

- When ② is issued, a new operator is selected and, since source register F0 is free, its content can be used...
- ... but label ① in F4 denotes a dependence between ① and ②.
- The virtual operator assigned to ② will have to wait until the missing value is broadcast.

Example of dynamic instruction scheduling – 5th cycle

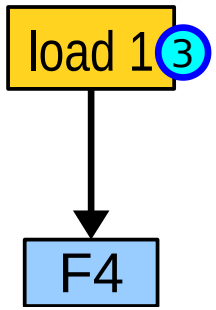
		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F2,F0`	IF	ID	I	A1	A2					
②	`MUL.D F6,F4,F0`		IF	ID	I	→					
③	`L.D F4,x`			IF	ID	I					
④	`MUL.D F6,F6,F4`				IF	ID					

Graph:



Reg. file:

F6	*0.03*	②
F4	*3.1416*	③
F2	*8.3*	–
F0	*0.5*	–



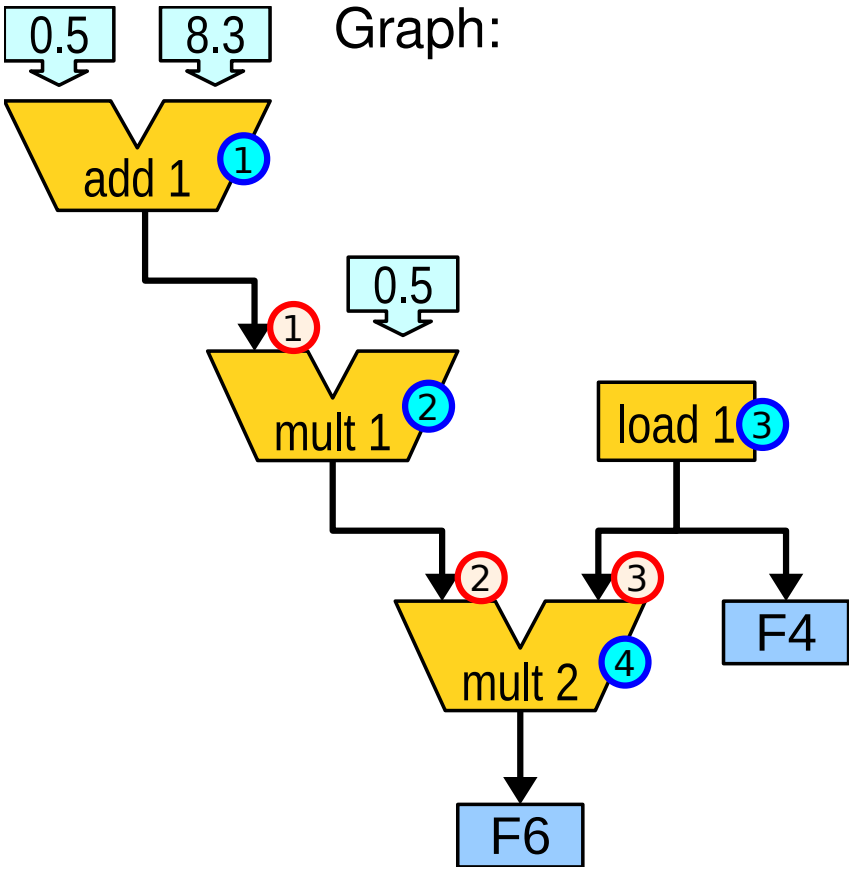
Label updating

- When ③ is issued, the label of F4 changes to ③.
- This does not affect the dependence that exists between ① and ②.
- Now we have two disjoint graphs

Example of dynamic instruction scheduling – 6th cycle

		1	2	3	4	5	6	7	8	9	10
1	`ADD.D F4,F2,F0`	IF	ID	I	A1	A2	A3				
2	`MUL.D F6,F4,F0`		IF	ID	I	→	→				
3	`L.D F4,x`			IF	ID	I	L1				
4	`MUL.D F6,F6,F4`				IF	ID	I				

Graph:



Reg. file:

F6	*0.03*	4
F4	*3.1416*	3
F2	*8.3*	–
F0	*0.5*	–

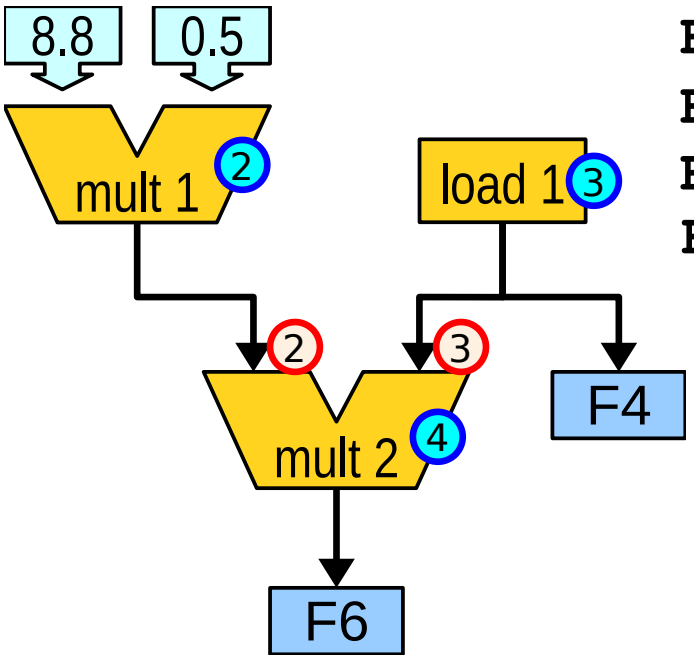
Label updating

- During the issue of 4, the label 3 of F4 enables proper connection of the new operator in the graph.

Example of dynamic instruction scheduling – 7th cycle

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F2,F0`	IF	ID	I	A1	A2	A3	**WB**			
②	`MUL.D F6,F4,F0`		IF	ID	I	→	→	→			
③	`L.D F4,x`			IF	ID	I	L1	L2			
④	`MUL.D F6,F6,F4`				IF	ID	I	→			

Graph:



Reg. file:

F6	*0.03*	4
F4	*3.1416*	3
F2	*8.3*	–
F0	*0.5*	–

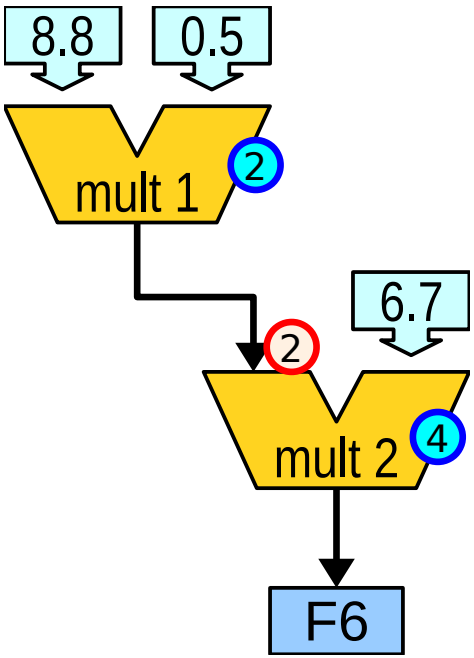
Result transmission

- When ① reaches stage WB, the result is broadcast to all dependent operators and the virtual operator `add 1` becomes free.
- Now, instructions waiting for this result are ready to start their execution.

Example of dynamic instruction scheduling – 8th cycle

		1	2	3	4	5	6	7	8	9	10
1	`ADD.D F4,F2,F0`	IF	ID	I	A1	A2	A3	WB			
2	`MUL.D F6,F4,F0`		IF	ID	I	→	→	→	M1		
3	`L.D F4,x`			IF	ID	I	L1	L2	**WB**		
4	`MUL.D F6,F6,F4`				IF	ID	I	→	→		

Graph:



Reg. file:

F6	*0.03*	4
F4	*6.7*	–
F2	*8.3*	–
F0	*0.5*	–

Register writing

- At the end of 3, F4 is updated and it becomes free since its label 3 is the same.
- Additionaly, the result is sent to 4.

Hazard resolution

Structural: They are solved by means of virtual operators. The execution only starts if the required physical operator is free.

Without dynamic instruction scheduling:

	1	2	3	4	5	6	7	8
DIV.D F6,F4,F2	IF	ID	D1	D2	D3	D4	D5	WB
DIV.D F12,F10,F8		IF	ID	*ID*	*ID*	*ID*	*ID*	D1
L.D F14,x			IF	*IF*	*IF*	*IF*	*IF*	ID

With dynamic instruction scheduling:

	1	2	3	4	5	6	7	8	9
DIV.D F6,F4,F2	IF	ID	I	D1	D2	D3	D4	D5	WB
DIV.D F12,F10,F8		IF	ID	I	–	–	–	–	D1
L.D F14,x			IF	ID	I	L1	L2	WB	

Hazard resolution (cont.)

RAW: They are solved by chaining those operators involved in the hazard.

Without dynamic instruction scheduling:

```
                    1   2   3   4   5   6
ADD.D F4,F2,F0      IF  ID  A1  A2  A3  WB
MUL.D F6,F4,F0          IF  ID  ID  ID  M1  M2
L.D F4,x                    IF  IF  IF  ID  L1
```

With dynamic instruction scheduling:

```
                    1   2   3   4   5   6   7   8
ADD.D F4,F2,F0      IF  ID  I   A1  A2  A3  WB
MUL.D F6,F4,F0          IF  ID  I   -   -   -   M1  M2
L.D F4,x                    IF  ID  I   L1  L2  WB
```

Hazard resolution (cont.)

WAW: The last issued instruction is the only one that effectively writes in the register involved in the hazard.

Without dynamic instruction scheduling:

```
                    1   2   3   4   5   6   7   8   9
MUL.D F2,F4,F0      IF  ID  M1  M2  M3  M4  M5  WB
ADD.D F2,F6,F0          IF  ID  ID  ID  A1  A2  A3  WB
DIV.D F4,F8,F10             IF  IF  IF  ID  D1  D2  D3
```

With dynamic instruction scheduling:

```
                    1   2   3   4   5   6   7   8   9
MUL.D F2,F4,F0      IF  ID  I   M1  M2  M3  M4  M5  WB
ADD.D F2,F6,F0          IF  ID  I   A1  A2  A3  WB
DIV.D F4,F8,F10             IF  ID  I   D1  D2  D3  D4
```

The dynamic dependence graph is built ensuring that only the last instruction updates F2.

Hazard resolution (cont.)

WAR: They are avoided by building the graph during the *Issue* stage.

Example:

```
                    1   2   3   4   5   6   7   8   9   10
MUL.D F2,F8,F0      IF  ID  I   M1  M2  M3  M4  M5  WB
MUL.D F6,F4,F2          IF  ID  I   -   -   -   -   -   M1
L.D F4,x                    IF  ID  I   L1  L2  WB
```

- Instructions execute the *Issue* stage in order
- Operands that are available are read at that time (F4 in the example),
- ... even when other operands have a data dependence with other previous instructions (F2 in the example).

Contents

Motivation

Branch prediction techniques:

- Try to determine the effective branch address as soon as possible in order to fetch instructions from such location.
- After executing a branch instruction, fetched instructions must be aborted if the condition was mispredicted.

Problem:

The branch can be predicted, but **the branch condition usually takes very long to be computed** → By the time the condition and the effective branch address are computed, instructions that follow the branch and have started their execution may have already completed execution. If so, they **cannot be cancelled**

Example:

```
loop: ...
      DIV.D F2,F0,F4 ; long operation
      C.GE.D F2,F12  ; ¿F2>=F12?
                     ; Result in FP status register (FPSR)
                     ; Waits until DIV.D ends
      BC1T  loop     ; If ((FPSR)=true) ... loop
                     ; Must wait for C.GE.D completion
      ...
Possible execution:
DIV.D F2,F0,F4       IF ID I  D1 D2 D3 D4 WB
C.GE.D F2,F12           IF ID I              A1 A2 WB
BC1T loop                  IF ID I                    EX
ADD.D F6,F4,F4                IF ID I  A1 A2 WB
```

→ By the time `BC1T loop` computes the condition, `ADD.D F6,F4,F4` has already finished.

Speculation

Technique that enables partial end even **complete** execution of instructions following a branch, before computing the branch condition. It takes into account that branch conditions may have been mispredicted, thus requiring execution rollback:

- Does not wait for the completion of branch instructions
- Bets on a given branch behavior (by using branch predictors)
- Executes instructions before knowing whether they must be executed or not → "speculative" execution
- If the prediction turns out to be incorrect, performed actions *should not have any effect* on the result of program execution
- But if the prediction is confirmed, performed actions *should be committed*

Key idea:

Speculative instructions should not alter program execution → the result of program execution with and without speculation must be the same:

- Data dependencies among instructions must be respected
- Speculative instructions should not modify neither registers nor "alive" memory locations.
- Speculative instructions should not generate new exceptions

until the branch prediction is confirmed, that is, until speculative instructions become "regular ones"

Contents

Basic idea

- At runtime, branch conditions (taken/not taken) are predicted, and subsequent instructions are “provisionally” fetched and executed
- . . . but the state of the machine remains unchanged (no writing on registers or memory locations, no exception generation) **until the branch condition is confirmed**

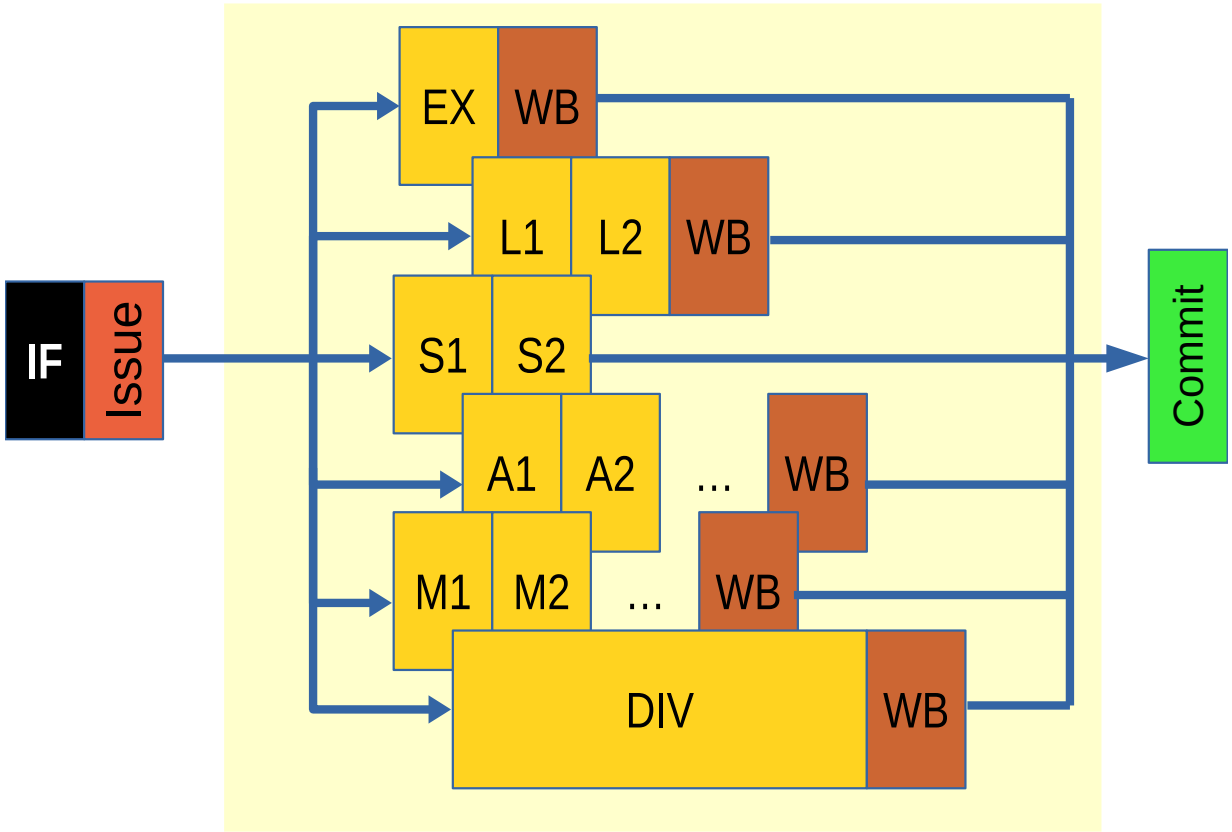
Implementation:

1. Dynamic branch prediction for selection of instructions to execute.
2. In-order instruction fetch (*IF*).
3. In-order instruction decode and issue (*ID+Issue* $\rightarrow$ *I*).
4. Out-of-order instruction execution (EX).
5. **In-order** instruction **completion**.

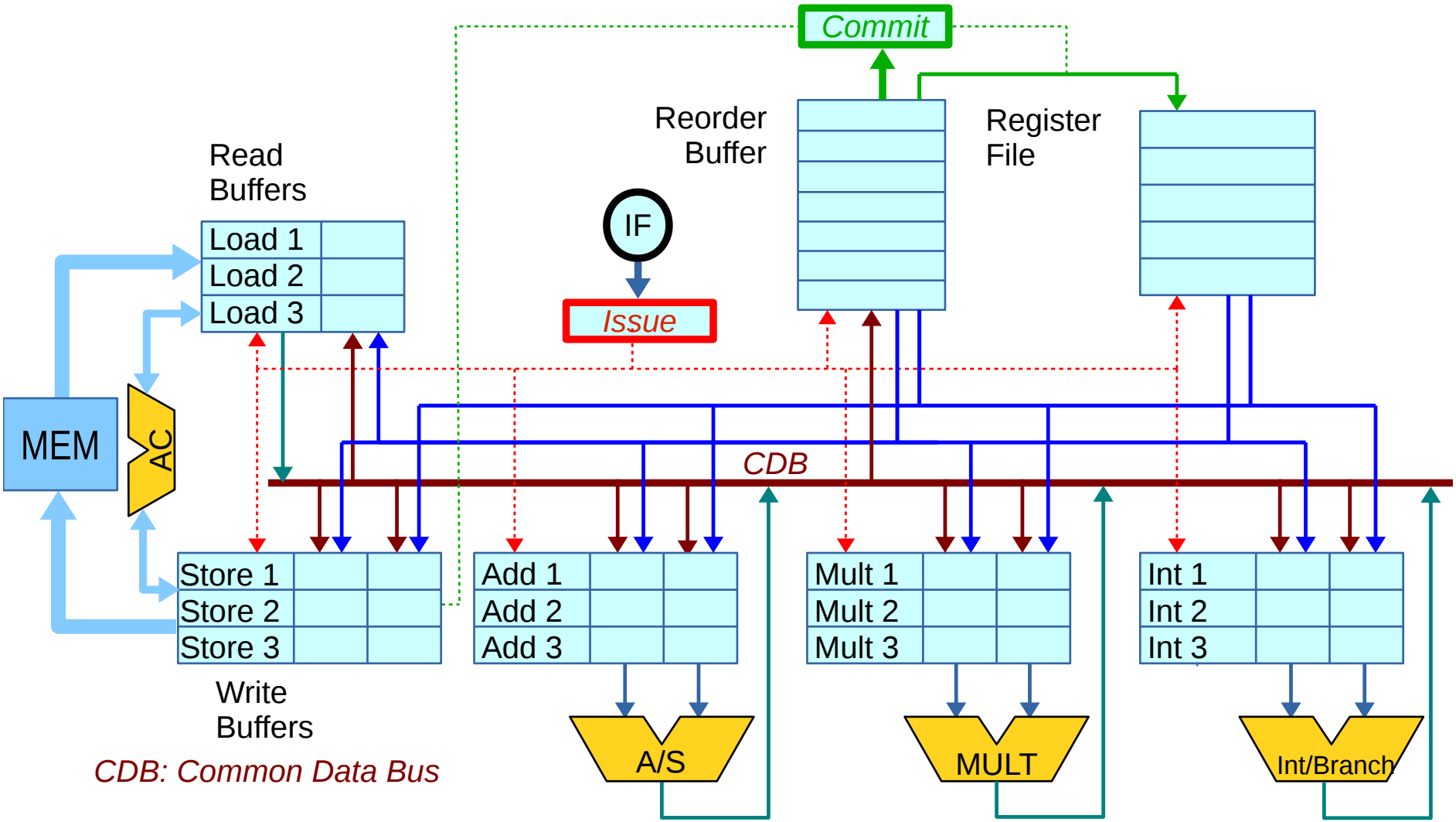
⇒ An additional stage is required for instruction completion: ***Commit* stage**

- In-order instruction arrival at this stage.
- Register/memory updates and exception management performed at this stage.
- When an instruction reaches the *Commit* stage, it is the oldest one in the processor pipeline:
  → all previous branches have been resolved and no previous instruction has provoked any exception.
  → no subsequent instruction has modified the state of the machine, and thus, they can be cancelled in case of need.
- Only those instructions that have been correctly speculated can reach the *Commit* stage.

Datapath



Datapath


Commit
Reorder Buffer
Register File
Read Buffers
IF
Issue
Load 1
Load 2
Load 3
MEM
AC
CDB
Store 1
Store 2
Store 3
Add 1
Add 2
Add 3
Mult 1
Mult 2
Mult 3
Int 1
Int 2
Int 3
Write Buffers
A/S
MULT
Int/Branch
CDB: Common Data Bus

Components of the datapath

General purpose registers

Operators Each operator has an associated data estructure with several entries ("reservation stations"). Each entry contains either an stalled or a running instruction.
Thus, each operator implements several "virtual operators".

Read and Write Buffers They contain data provided by / written to memory. They are managed by load/store units.

Common data bus fed by all units capable of generating results. It interconnects all the elements able to read data. Access to this bus is arbitrated when several units try to transfer data at the same time.

Reorder Buffer

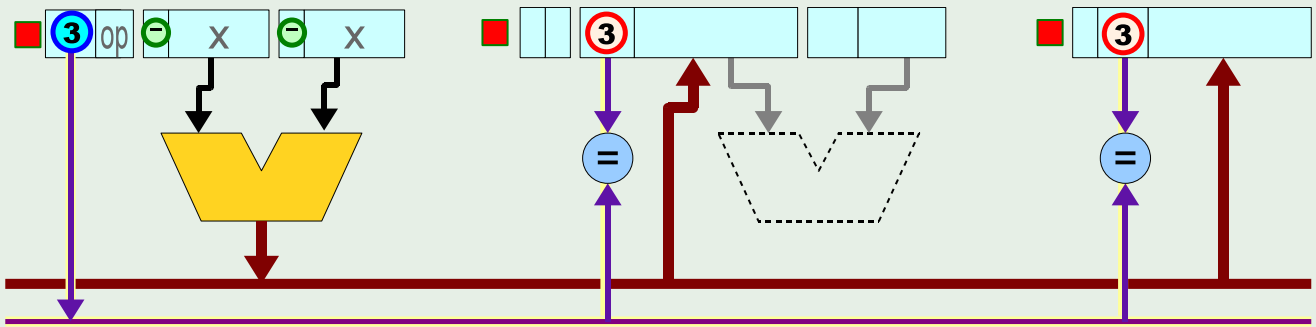
Transfers through the common data bus

- Elements writing results on the bus broadcast its value and a "code" identifying the instruction producing that value.
  1. Virtual operators
  2. Read buffers
- Elements reading data from the bus have a variable "mark" that identifies the instruction they are waiting for
  1. Reservation stations
  2. Write buffers
- Broadcasting a result: when an element places a data on the bus, it also writes its code. All the elements whose mark matches such code, read the data.

Transfers through the common data bus (cont.)

Example: Data transfer through the CDB

1. Preparation: Write sender code “3” in the mark field of targets.
2. Transfer:
   1. Place data and code (“3”) in the bus
   2. At each reservation station, if its mark matches the code, read data.



Reorder buffer (ROB)

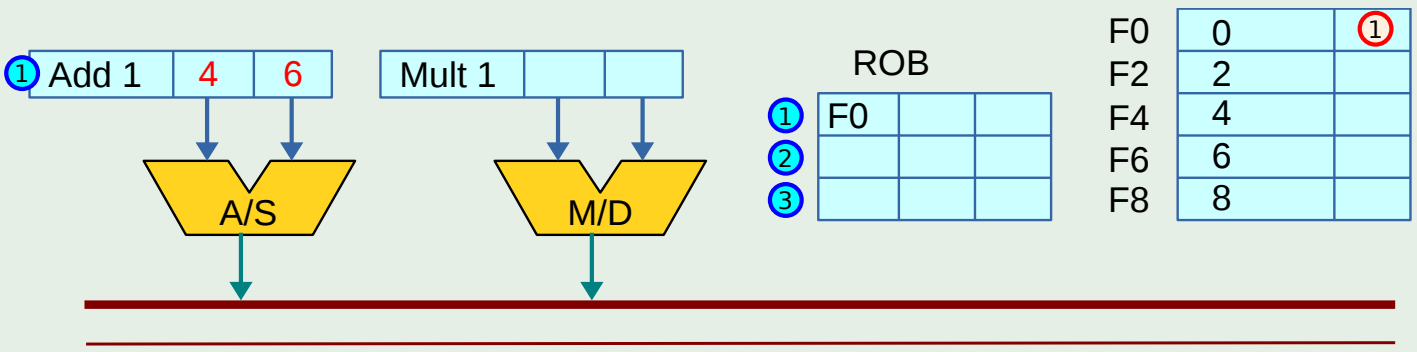
- When an instruction is issued, a ROB entry is allocated for it. The entry number is used as a mark to tag the required destinations (register file, etc)
- When an instruction reaches the WB stage, it broadcasts its result to the reservation stations but it writes its result into its ROB entry, not in the destination register.
  → if an instruction has a data dependency with a following one, the latter instruction will obtain its operands from the ROB, and not from registers.
- If an instruction provokes an exception, the event is annotated in its corresponding ROB entry.

Reorder buffer (ROB) (cont.)

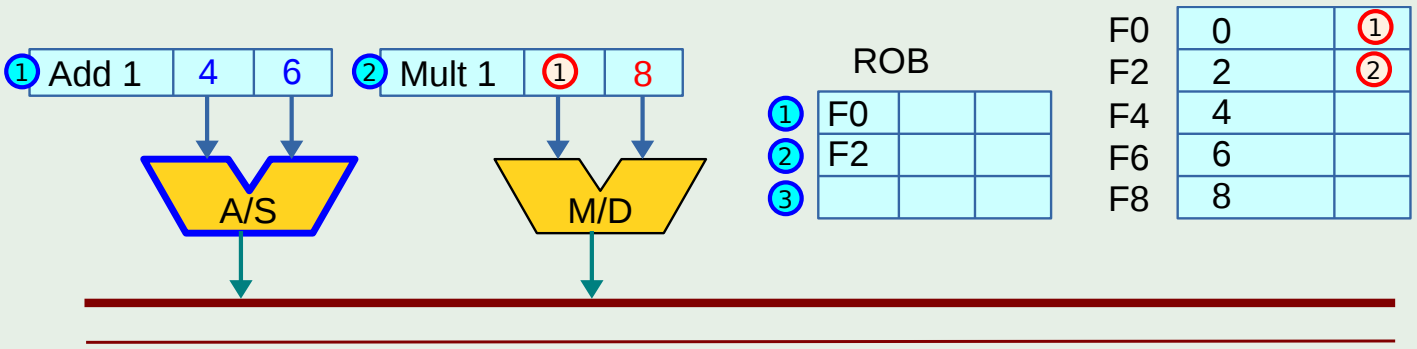
- When the oldest instruction in the *reorder buffer* reaches the *Commit* stage:
  - It is checked whether it has raised an exception. If this is the case, the corresponding handler is executed.
  - The instruction result is copied from its *reorder buffer* entry to the corresponding destination register or memory location.
  - The *reorder buffer* entry is released.
- If a branch is incorrectly predicted, when it reaches the *Commit* stage it flushes the *reorder buffer*.
  → speculative instructions that have been incorrectly fetched after the branch:
  - do not write into their destination register (they do not confirm their execution).
  - do not originate any exception.

Hardware speculation through an example:

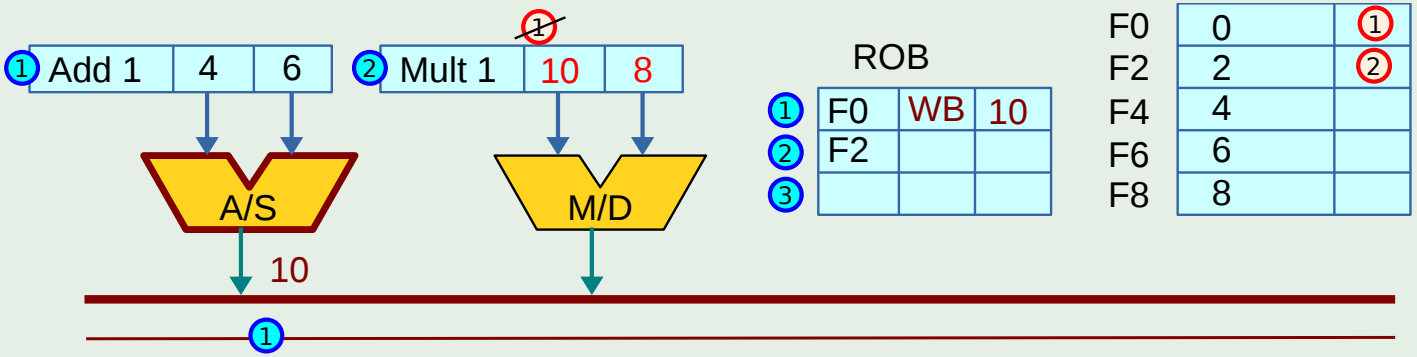
```
ADD.D  F0,F4,F6 -Issue
```



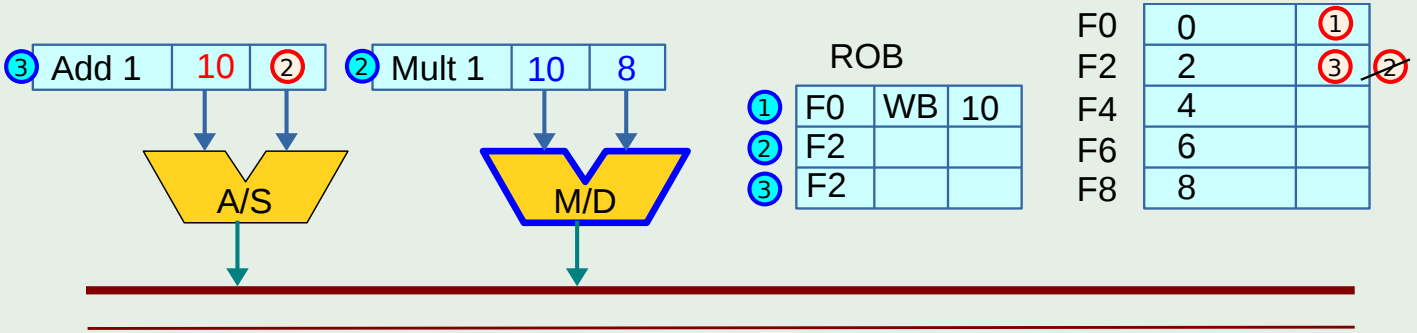

```
ADD.D  F0,F4,F6 -EX
MULT.D F2,F0,F8 -Issue
```



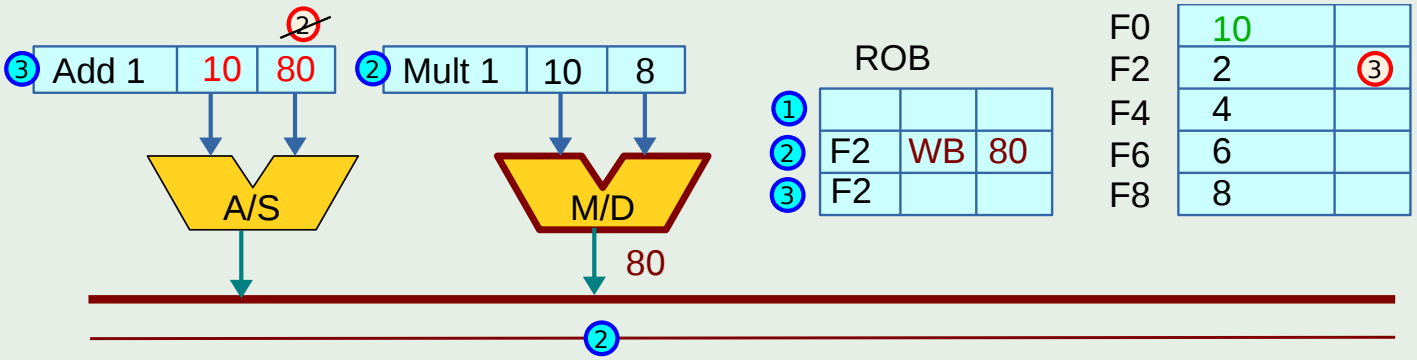
```
ADD.D  F0,F4,F6 -WB
MULT.D F2,F0,F8 -Ready for EX
```



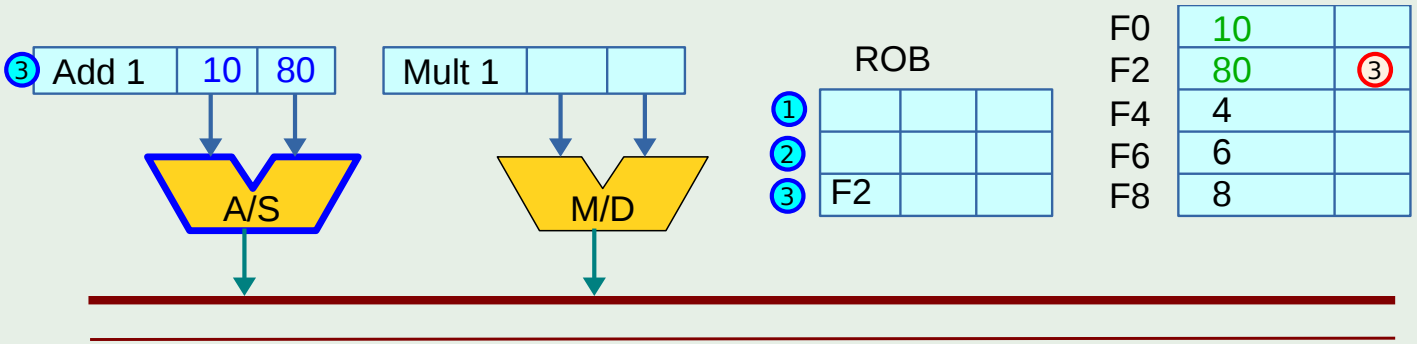

```
ADD.D  F0,F4,F6 -Waiting for Commit
MULT.D F2,F0,F8 -EX
ADD.D  F2,F0,F2 -Issue
```



```
ADD.D  F0,F4,F6 -Commit
MULT.D F2,F0,F8 -WB
ADD.D  F2,F0,F2 -Ready for EX
```




```
MULT.D F2,F0,F8 -Commit
ADD.D  F2,F0,F2 -EX
```



Tomasulo's algorithm with speculation: data structures

Registers (`Regs`):

- Value (`value`)
- *Reorder buffer* entry (`rob`)

Operators: Floating point, integer and branches, with several reservation stations (`RS`), each one including:

- Busy bit (`busy`)
- Operation to be executed (`op`)
- Operand 1 - Value (`Vj`)
- Operand 1 - Mark (`Qj`)
- Operand 2 - Value (`Vk`)
- Operand 2 - Mark (`Qk`)
- *Reorder buffer* entry (`rob`)
- Result (`result`)

Tomasulo's algorithm with speculation: data structures (cont.)

Read buffers (`RB`):

- Busy bit (`busy`)
- Memory address (with displacement addressing mode):
  - Base register - Value (`Vj`)
  - Base register - Mark (`Qj`)
  - Displacement (`disp`)
  - Computed address (`addr`) and its valid bit (`valid`)
- *Reorder buffer* entry (`rob`)
- Result (`result`)

Tomasulo's algorithm with speculation: data structures (cont.)

Write buffers (`WB`):

- Busy bit (`busy`)
- Operand - Value (`Vk`)
- Operand - Mark (`Qk`)
- Memory address (with displacement addressing mode):
  - Base register - Value (`Vj`)
  - Base register - Mark (`Qj`)
  - Displacement (`disp`)
  - Computed address (`addr`) and its valid bit (`valid`)
- Confirmation bit (`conf`)

Tomasulo's algorithm with speculation: data structures (cont.)

Reorder buffer (`ROB`):

- Busy bit (`busy`)
- Instruction PC (`PC`).
- Instruction (`instr`): Branch (no result), store (result goes to memory) or ALU/load (result goes to a register).
- Destination (`dest`): Register number, write buffer entry (store) or target address (branches).
- Value (`value`): Value to store or condition (`cond`) (branches). Exception identifier, when required.
- State (`state`): Indicates whether the instruction has reached the WB stage.
- Prediction (`pred`): Predicted condition (branches).

Tomasulo's algorithm with speculation

Instruction execution is performed in four stages:

Issue

```
Instruction (instr) decoding:
-ALU D, S1, S2
-LOAD D, displacement(S1)
-STORE S2, displacement(S1)
-BRANCH S1,address,prediction

// Is there any free operator?
// Is there any free ROB entry?
If {s:reservation station or buffer} is free and
   {b:ROB entry} is free, then

  // Reservation station or buffer
  RS[s].busy or RB[s].busy or WB[s].busy := ``1'';
  RS[s].op := {op: arithmetic operation};
  RS[s].rob or RB[s].rob := b; // ROB entry
```

Issue (cont.)

```
// Source operand 1: ALU, LOAD, STORE, BRANCH
If (Regs[S1].rob = null_mark) then // Read value
  RS[s].Vj or RB[s].Vj or WB[s].Vj := Regs[S1].value;
  RS[s].Qj or RB[s].Qj or WB[s].Qj := null_mark;
Else
  If ROB[Regs[S1].rob].state=WB then // Read ROB
    RS[s].Vj or RB[s].Vj or WB[s].Vj := ROB[Regs[S1].rob].value;
    RS[s].Qj or RB[s].Qj or WB[s].Qj := null_mark;
  Else // Annotate ROB entry
    RS[s].Qj or RB[s].Qj or WB[s].Qj := Regs[S1].rob;

// Source operand 2: ALU or STORE
If {instr is ALU or STORE}
  If (Regs[S2].rob = null_mark) then // Read value
    RS[s].Vk or WB[s].Vk := Regs[S2].value;
    RS[s].Qk or WB[s].Qk := null_mark;
  Else
    If ROB[Regs[S2].rob].state=WB then // Read ROB
      RS[s].Vk or WB[s].Vk := ROB[Regs[S2].rob].value;
      RS[s].Qk or WB[s].Qk := null_mark;
    Else // Annotate ROB entry
      RS[s].Qk or WB[s].Qk := Regs[S2].rob;
```

Issue (cont.)

```
// Displacement: LOAD and STORE
If {instr is LOAD or STORE}
  RB[s].disp or WB[s].disp := displacement;

// Reorder buffer
ROB[b].busy := ``1'';
ROB[b].instr := instr;
ROB[b].PC := PC;    // PC of the instruction at issue stage
If {instr is ALU or LOAD}
  ROB[b].dest := D;
If {instr is STORE}
  ROB[b].dest := s;
If {instr is BRANCH}
  ROB[b].dest := address; // Computed by Issue
  ROB[b].pred := prediction; // Provided by the predictor

// Destination register reservation: ALU and LOAD
If {instr is ALU or LOAD}
  Regs[D].rob := b; // ROB entry
```

EX (ALU)

```
If {there exist ready reservation stations} then
  x := Select_one()
  Operation:
    RS[x].result := RS[x].Vj op RS[x].Vk;
  If an exception occurs, record it
```

EX (Branches)

```
If {there exist ready reservation stations} then
  x := Select_one()
  Operation:
    RS[x].result := RS[x].Vj op 0;
```

AC, Address Calculation (LOAD and STORE)

```
If {there exist read or write buffers with a ready operand to compute
 the memory address} then
  x := Select_one()
  Compute address:
    If (LOAD) then RB[x].addr := RB[x].Vj + RB[x].disp;
    If (STORE) then WB[x].addr := WB[x].Vj + WB[x].disp;
  If an exception occurs, record it
```

MEM (LOAD)

```
If {there exist read buffers with address computed and disambiguated}
// Disambiguated: its address does not match the one for ongoing STORES

  x := Select_one()
  Memory access:
    RB[x].result := Mem[RB[x].addr];
```

WB

```
For {y: virtual operator (ALU, BRANCH) or read buffer (LOAD)} do
  Put {data := RS[y].result or RB[y].result} in common bus CDB
  Put {rb := RS[y].rob or RB[y].rob} in common bus CDB
  // Recorded information about exceptions is also transmitted

  For {x: virtual operator} do
    // Operand 1
    If RS[x].Qj=rb then       // Mark==#rb
      RS[x].Vj := data;       // Read data from bus
      RS[x].Qj := null_mark; // Delete mark

    // Operand 2
    If RS[x].Qk=rb then       // (Mark==#rb
      RS[x].Vk := data;       // Read data from bus
      RS[x].Qk := null_mark; // Delete mark

  For {x: read buffer} do
    // Operand 1
    If RB[x].Qj=rb then       // Mark==#rb
      RB[x].Vj := data;       // Read data from bus
      RB[x].Qj := null_mark; // Delete mark
```

WB (cont.)

```
For {x: write buffer} do
  // Operand 1
  If WB[x].Qj=rb then       // Mark==#rb
    WB[x].Vj := data;       // Read data from bus
    WB[x].Qj := null_mark; // Delete mark

  // Operand 2
  If WB[x].Qk=rb then       // (Mark==#rb
    WB[x].Vk := data;       // Read data from bus
    WB[x].Qk := null_mark; // Delete mark

RS[y].busy or RB[y].busy := ``0''; // Free reserv. station or buffer

// Copy to ROB
// In case of a branch, the data contains the condition
// In case of exception, information is recorded in the ROB
ROB[rb].value := data;

ROB[rb].state := WB;   // Ready for Commit
```

Commit

```
If {instruction at ROB head (h entry) has finished} then
    // An instruction has finished if:
    // - LOAD, ALU, BRANCH: Has completed the WB stage
    // - STORE: Has computed the memory address

    // Process pending exceptions, if any. Otherwise:

  If (ROB[h].instr=BRANCH) and (ROB[h].pred <> ROB[h].value) then
    // Incorrect prediction
    ROB[*].busy := ``0'';              // Delete entire ROB
    // Delete reservation stations, except for confirmed writes:
    RS,RB,WB[*].busy := ``0'';
    Regs[*].rob := null_mark;          // Free registers
    If (ROB[h].value) then // Fetch instructions from the right path:
      PC := ROB[h].dest;               // Branch is taken
    Else
      PC := ROB[h].PC+4;               // Branch is not taken
```

Commit

```
If (ROB[h].instr=STORE) then
  WB[ROB[h].dest].conf := ``1''; // Confirm write operation

If (ROB[h].instr=ALU) or (ROB[h].instr=LOAD) then
  Regs[ROB[h].dest].value := ROB[h].value; // Update register
  If (Regs[ROB[h].dest].rob = h) then
    // No subsequent instruction writes on this register
    Regs[ROB[h].dest].rob := null_mark;    // Free register

ROB[h].busy := ``0''; // Free ROB entry
```

MEM (STORE)

```
If {there are confirmed write buffers} then
  x := Select_one()
  Memory access:
    Mem[WB[x].addr] := WB[x].Vk ;
```

Comments:

- The ROB provides an space to store the instruction results → it **dynamically renames registers**.
- The ROB entry number allows chaining results between instructions with unsolved data dependencies.
- Reservation stations store information for instructions since they are issued (I) until they complete execution (WB).
- Reservation stations also **monitor** the common data bus looking for operands required by on-hold instructions.
- During the WB stage, reservation stations directly write their result or the generated exception information into the corresponding ROB entry.
  The ROB does not monitor the common bus (for a large ROB size, this monitoring would require too many comparators).

Comments: (cont.)

- During the *Commit* stage, the result in the ROB entry is copied to the destination register, even when any subsequent instruction has blocked the register (since it may end up not being executed).
- However, in that case the register is not released (since the ROB entry indicated in the register `rob` field is used to correctly chain dependent instructions).

Example 1

Data:

- Load unit=2 cycles, IR=$\frac{1}{2}$
- Add/sub=2 cycles, IR=1
- Mult/Div=7 cycles, IR=1
- Regs[F4] = 4.0; Regs[R1] = 8; Regs[R2] = 32;
  Mem[a+8] = *x*; Mem[b+32] = *y*;

Code:

```
l.d f1, a(r1)
l.d f2, b(r2)
mul.d f0,f2,f4
sub.d f3,f2,f1
div.d f5,f0,f1
add.d f0,f3,f2
```

i–t diagram

PC	Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	l.d f1,a(r1)	IF	I	AC	L1	L2	WB	C											
1	l.d f2,b(r2)		IF	I	AC	–	L1	L2	WB	C									
2	mul.d f0,f2,f4			IF	I	–	–	–	–	M1	M2	M3	M4	M5	M6	M7	WB	C	
3	sub.d f3,f2,f1				IF	I	–	–	–	A1	A2	WB	–	–	–	–	–	–	C
4	div.d f5,f0,f1					IF	I	–	–	–	–	–	–	–	–	–	–	M1	M2
5	add.d f0,f3,f2						IF	I	–	–	–	–	A1	A2	WB	–	–	–	–

Processor state

At the end of the 16th cycle:

ROB

	busy	instr	state	dest	value	pred	PC
0	No	*l.d f1,a(r1)*	*WB*	*F1*	*x*		*0*
1	No	*l.d f2,b(r2)*	*WB*	*F2*	*y*		*1*
2	Yes	mul.d f0,f2,f4	WB	F0	4*y		2
3	Yes	sub.d f3,f2,f1	WB	F3	y-x		3
4	Yes	div.d f5,f0,f1		F5			4
5	Yes	add.d f0,f3,f2	WB	F0	y-x+y		5

Processor state (cont.)

Reservation stations:

	busy	Op	Qj	Vj	Qk	Vk	rob	result
a1	No	-		*y*		*x*	*#3*	*y-x*
a2	No	+		*y-x*		*y*	*#5*	*y-x+y*
m1	No	*		*y*		*4.00*	*#2*	*4*y*
m2	Yes	/		4*y		x	#4	

Read/Write buffers:

	busy	Qj	Vj	disp	addr	rob	result
l1	No		*8*	*a*	*8+a*	*#0*	*x*
l2	No		*32*	*b*	*32+b*	*#1*	*y*

	busy	Qj	Vj	disp	addr	rob	Qk	Vk	confirm
s1	No								
s2	No								

Registers:

	F0	F1	F2	F3	F4	F5	F6	F7
rob	#5			#3		#4		
value		x	y		4.0			

	R0	R1	R2	R3	R4	R5	R6	R7
rob								
value	0	8	32					

Example 2

Data:

- Load/store=3 cycles, IR=$\frac{1}{3}$
- Mult/Div=3 cycles, IR=1
- Integer=1 cycle, IR=1
- Mem[V+72]= $x1$; Mem[V+64]= $x2$; ...
- Regs[F2]= 2.0; Regs[R1]= 72

Code:

```
loop: l.d f0,V(r1)
      mul.d f4,f0,f2
      s.d f4,V(r1)
      dsubi r1,r1,8
      bnez r1,loop
      trap 0
```

i–t diagram

PC	Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	l.d f0,V(r1)	IF	I	AC	L1	L2	L3	WB	C								
1	mul.d f4,f0,f2		IF	I	–	–	–	–	M1	M2	M3	WB	C				
2	s.d f4,V(r1)			IF	I	AC	–	–	–	–	–	–	–	C	L1	L2	L3
3	dsubi r1,r1,8				IF	I	E1	–	WB	–	–	–	–	–	C		
4	bnez r1,loop					IF	I	–	–	E1	WB	–	–	–	–	C	
0	l.d f0,V(r1)						IF	I	–	AC	L1	L2	L3	WB	–	–	C
1	mul.d f4,f0,f2							IF	I	–	–	–	–	–	M1	M2	M3
2	s.d f4,V(r1)								IF	I	AC	–	–	–	–	–	–
3	dsubi r1,r1,8									IF	I	E1	WB	–	–	–	–
4	bnez r1,loop										IF	I	–	E1	WB	–	–
0	l.d f0,V(r1)											IF	I	AC	–	–	–
1	mul.d f4,f0,f2												IF	I	–	–	–
2	s.d f4,V(r1)													IF	I	AC	–
3	dsubi r1,r1,8														IF	I	E1
4	bnez r1,loop															IF	I
0	l.d f0,V(r1)																IF

Note: In cycle 7 there is a structural hazard when accessing the CDB at the WB stage. It is solved by assigning higher priority to the oldest instruction.

Processor state (cycle 7)

ROB:

	busy	instr	state	dest	value	pred	PC
0	Yes	l.d f0,V(r1)	WB	F0	x1		0
1	Yes	mul.d f4,f0,f2		F4			1
2	Yes	s.d f4,V(r1)		s1			2
3	Yes	dsubi r1,r1,8		R1			3
4	Yes	bnez r1,loop		loop		taken	4
5	Yes	l.d f0,V(r1)		F0			0

Processor state (cycle 7) (cont.)

Reservation stations:

	busy	Op	Qj	Vj	Qk	Vk	rob	result
e1	Yes	-		72		8	#3	64
e2	Yes	B	#3			0	#4	
m1	Yes	*		x1		2	#1	
m2	No							

Read/Write buffers:

	busy	Qj	Vj	disp	addr	rob	result
l1	Yes	#3		V		#5	
l2	No						
l3	No						

	busy	Qj	Vj	disp	addr	rob	Qk	Vk	confirm
s1	Yes		72	V	V+72	#2	#1		No
s2	No								
s3	No								

Processor state (cycle 7) (cont.)

Registers:

	F0	F1	F2	F3	F4	F5	F6	F7
rob	#5				#1			
value			2.00					

	R0	R1	R2	R3	R4	R5	R6	R7
rob		#3						
value	0	72						

Processor state (cycle 15)

ROB:

	busy	instr	state	dest	value	pred	PC
0	No	*l.d f0,V(r1)*	*WB*	*F0*	*x1*		*0*
1	No	*mul.d f4,f0,f2*	*WB*	*F4*	*2*x1*		*1*
2	No	*s.d f4,V(r1)*		*s1*			*2*
3	No	*dsubi r1,r1,8*	*WB*	*R1*	*64*		*3*
4	No	*bnez r1,loop*	*WB*	*loop*	*taken*	*taken*	*4*
5	Yes	l.d f0,V(r1)	WB	F0	x2		0
6	Yes	mul.d f4,f0,f2		F4			1
7	Yes	s.d f4,V(r1)		s2			2
8	Yes	dsubi r1,r1,8	WB	R1	56		3
9	Yes	bnez r1,loop	WB	loop	taken	taken	4
10	Yes	l.d f0,V(r1)		F0			0
11	Yes	mul.d f4,f0,f2		F4			1
12	Yes	s.d f4,V(r1)		s3			2
13	Yes	dsubi r1,r1,8		R1			3
14	No						
15	No						

Processor state (cycle 15) (cont.)

Reservation stations:

	busy	Op	Qj	Vj	Qk	Vk	rob	result
e1	Yes	-		56		8	#13	
e2	No	*B*		*56*		*0*	*#9*	*taken*
m1	Yes	*	#10			2.0	#11	
m2	Yes	*		*x*2		2.0	#6	

Read/Write buffers:

	busy	Qj	Vj	disp	addr	rob	result
l1	No		*64*	*V*	*V+64*	*#5*	*x2*
l2	Yes		56	V	V+56	#10	
l3	No						

	busy	Qj	Vj	disp	addr	rob	Qk	Vk	confirm
s1	Yes		72	V	V+72	#2		2*x1	Yes
s2	Yes		64	V	V+64	#7	#6		No
s3	Yes		56	V	V+56	#12	#11		No

Processor state (cycle 15) (cont.)

Registers:

	F0	F1	F2	F3	F4	F5	F6	F7
rob	#10				#11			
value	x1		2.00		2*x1			

	R0	R1	R2	R3	R4	R5	R6	R7
rob		#13						
value	0	64						

Memory:

Addr	Data
..	...
V+56	x3
V+64	x2
V+72	x1

Example 2: The first branch is incorrectly predicted

PC	Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	l.d f0,V(r1)	IF	I	AC	L1	L2	L3	WB	C								
1	mul.d f4,f0,f2		IF	I	–	–	–	–	M1	M2	M3	WB	C				
2	s.d f4,V(r1)			IF	I	AC	–	–	–	–	–	–	–	C	L1	L2	L3
3	dsubi r1,r1,8				IF	I	E1	–	WB	–	–	–	–	–	C		
4	bnez r1,loop					IF	I	–	–	E1	WB	–	–	–	–	C	
5	trap 0						IF	I	–	–	–	–	–	–	–	x	
6	next							IF	I	–	–	–	–	–	–	x	
7	next								IF	I	–	–	–	–	–	x	
8	next									IF	I	–	–	–	–	x	
9	next										IF	I	–	–	–	x	
10	next											IF	I	–	–	x	
11	next												IF	I	–	x	
12	next													IF	I	x	
13	next														IF	X	
14	next															X	
0	l.d f0,0(r1)																IF

Processor state (cycle 15)

ROB:

	busy	instr	state	dest	value	pred	PC
0	No	*l.d f0,V(r1)*	*WB*	*F0*	*x1*		*0*
1	No	*mul.d f4,f0,f2*	*WB*	*F4*	*2*x1*		*1*
2	No	*s.d f4,V(r1)*		*s1*			*2*
3	No	*dsubi r1,r1,8*	*WB*	*R1*	*64*		*3*
4	No	*bnez r1,loop*	*WB*	*loop*	*taken*	*not taken*	*4*
5	No	*trap 0*	*WB*				*5*
6	No	*next*					*6*
7	No	*next*					*7*
8	No	*next*					*8*
9	No	*next*					*9*
10	No	*next*					*10*
11	No	*next*					*11*
12	No	*next*					*12*
13	No						
14	No						
15	No						

Processor state (cycle 15) (cont.)

Reservation stations:

	busy	Op	Qj	Vj	Qk	Vk	rob	result
e1	No	-		72		8	#3	64
e2	No	B		64		0	#4	taken
m1	No	*		x1		2.00	#1	2*x1
m2	No							

Read/Write buffers:

	busy	Qj	Vj	disp	addr	rob	result
l1	No		72	V	V+72	#0	x1
l2	No						
l3	No						

	busy	Qj	Vj	disp	addr	rob	Qk	Vk	confirm
s1	Yes		72	V	V+72	#2		2*x1	Yes
s2	No								
s3	No								

Processor state (cycle 15) (cont.)

Registers:

	F0	F1	F2	F3	F4	F5	F6	F7
rob								
value	x1		2.0		2*x1			

	R0	R1	R2	R3	R4	R5	R6	R7
rob								
value	0	64						

Memory:

Addr	Data
..	...
V+56	*x*3
V+64	*x*2
V+72	*x*1

Processor state (cycle 16)

Registers:

	F0	F1	F2	F3	F4	F5	F6	F7
rob								
value	x1		2.0		2*x1			

	R0	R1	R2	R3	R4	R5	R6	R7
rob								
value	0	64						

Memory:

Addr	Data
..	...
V+56	$x3$
V+64	$x2$
V+72	$2 * x1$