

How to improve (decrease) CPI

- Recall: $CPI = \text{Ideal CPI} + \text{CPI contributed by stalls}$
- Ideal CPI = 1 for single issue machine even with multiple execution units
- Ideal CPI will be less than 1 if we have several execution units and we can issue (and “commit”) multiple instructions in the same cycle, i.e., we take advantage of **Instruction Level Parallelism (ILP)**

Extending the simple pipeline

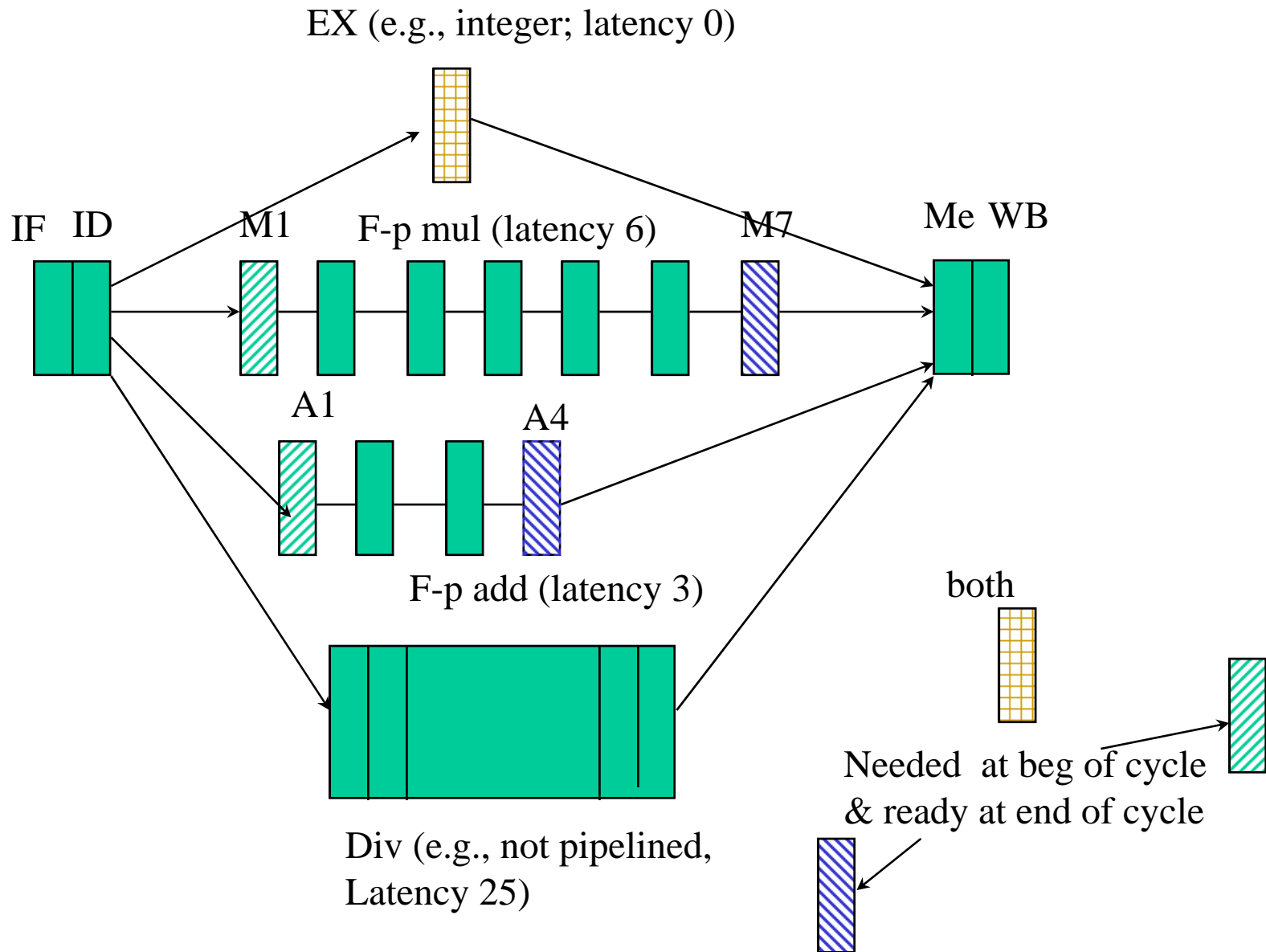
- Have multiple functional units for the EXE stage
- Increase the depth of the pipeline
 - Required because of clock speed
- Increase the width of the pipeline
 - Several instructions are fetched and decoded in the **front-end** of the pipe
 - Several instructions are **issued** to the functional units in the **back-end**
 - If m is the maximum number of instructions that can be issued in one cycle, we say that the processor is *m-wide*.

Superscalar Terminology

- Multiple issue processors are called **superscalars**
- If instructions are issued to the back-end in program order, we have **in-order processors**
 - In-order processors are **statically scheduled**, i.e., the scheduling is done at compile-time
- If instructions can be issued to the back-end in any order, we have **out-of-order (OOO) processors**
 - OOO processors are **dynamically scheduled** by the hardware

Extending simple pipeline to multiple pipes

- Single issue: in ID stage direct to one of several EX stages
- Common WB stage
- EX of various pipelines might take more than one cycle
- **Latency** of an EX unit = Number of cycles before its result can be forwarded = Number of stages – 1
- Not all EX need be pipelined
- IF EX is pipelined
 - A new instruction can be assigned to it every cycle (if no data dependency) or, maybe only after x cycles, with x depending on the function to be performed



Hazards in example multiple cycle pipeline

- **Structural:** Yes
 - Divide unit is not pipelined. In the example processor two Divides separated by less than 25 cycles will stall the pipe
 - Several writes might be “ready” at the same time and want to use WB stage at the same time (not possible if single write port)
- **RAW:** Yes
 - Essentially handled as in integer pipe (the dependent instruction is stalled at the beginning of its EX stage) but with higher frequency of stalls. Also more forwarding paths are needed.
- **WAW :** Yes (see later)
 - WAR no since read is in the ID stage
- **Out of order completion :** Yes (see later)

RAW: Example from the book (pg A-51)

```
F4 <- M      IF ID EX MeWB
F0 <- F4 * F6  IF ID st M1 M2 M3 M4 M5 M6 M7 Me WB
F2 <- F0 + F8  IF st ID st st st st st st st A1 A2 A3 A4 Me WB
M <- F2       IF st st st st st st st ID EX st st st Me WB
```

In **blue** data dependencies hazard

In **red** structural hazard

In **green** stall cycles

Note both the data dependency and structural hazard for the 4th instruction

Conflict in using the WB stage

- Several instructions might want to use the WB stage at the same time
 - E.g., A Multd issued at time t and an addd issued at time $t + 3$
- Solution 1: reserve the WB stage at ID stage (scheme already used in CRAY-1 built in 1976)
 - Keep track of WB stage usage in shift register
 - Reserve the right slot. If busy, stall for a cycle and repeat
 - Shift every clock cycle
- Solution 2: Stall before entering either Me or WB
 - Pro: easier detection than solution 1
 - Con: need to be able to trickle the stalls “backwards”.

Example on how to reserve the WB stage (Solution 1 in previous slide)

Time in ID stage	Operation	Shift register
t	multd	000 000 00 1
t + 1	int	00 1 000 0 1 0
t + 2	int	0 1 1 000 1 00
t + 3	addd	1 10 00 X 000

Note: multd and addd want WB at time t + 9. addd will be asked to stall one cycle

Instructions complete out of order (e.g., the two int terminate before the multd)

WAW Hazards

- Instruction i writes f-p register Fx at time t
Instruction $i + k$ writes f-p register Fx at time $t - m$
- But no instruction $i + 1, i + 2, i + k$ uses (reads) Fx (otherwise there would be a stall in in-order issue processors)
- Only requirement is that $i + k$'s result be stored
 - Note: this situation should be rare (useless instruction i)
- Solutions:
 - Squash i : difficult to know where it is in the pipe
 - At ID stage check that result register is not a result register in all subsequent stages of other units. If it is, stall appropriate number of cycles.

Out-of-order completion

- Instruction i finishes at time t
Instruction $i + k$ finishes at time $t - m$
 - No hazard etc. (see previous example on integer completing before multd)
- What happens if instruction i causes an exception at a time in $[t-m, t]$ and instruction $i + k$ writes in one of its own source operands (i.e., is **not restartable**)?
- We'll take care of that in OOO processors

Exception handling

- Solutions (cf. book pp A-54 – A-56 for more details)
 - Do nothing (imprecise exceptions; bad with virtual memory)
 - Have a precise (by use of testing instructions) and an imprecise mode; effectively restricts concurrency of f-p operations
 - Buffer results in a “history file” (or a “future file”) until previous (in order) instructions have completed; can be costly when there are large differences in latencies but a similar technique is used for OOO execution .
 - Restrict concurrency of f-p operations and on an exception “simulate in software” the instructions in between the faulting and the finished one.
 - Flag early those operations that might result in an exception and stall accordingly

Exploitation of Instruction Level Parallelism (ILP)

- Will **increase throughput** and decrease CPU execution time
- Will **increase structural hazards**
 - Cannot issue simultaneously 2 instructions to the same functional unit
- Makes **reduction in other stalls** even **more important**
 - A stall costs more than the loss of a single instruction issue
- Will make the **design more complex** mostly in OOO processors where:
 - **WAW and WAR hazards** can occur
 - **Out-of-order completion** can occur
 - Precise exception handling is more difficult

Where can we optimize exploitation of ILP?

- Speculative execution
 - Branch prediction (we have seen that already)
 - Bypassing Loads (memory reference speculation)
 - Predication (we'll see this technique with statically scheduled VLIW machines)
- Hardware (run-time) techniques
 - Forwarding (RAW; we have seen that)
 - Register renaming (WAW, WAR)

Data dependencies (RAW)

- Instruction (statement) S_j dependent on S_i if
$$O_i \cap I_j \neq \emptyset$$
 - Transitivity: Instruction j dependent on k and k dependent on i
- Dependence is a program property
- Hazards (RAW in this case) and their (partial) removals are a pipeline organization property
- Code scheduling goal
 - Maintain dependence and avoid hazard (pipeline is *exposed to the compiler*)

Name dependence

- **Anti dependence** $O_j \cap I_i \neq \emptyset$
 - Si: ...<- **R1**+ R2;; Sj: **R1** <- ...
 - At the instruction level, this is **WAR** hazard if instruction j finishes first
- **Output dependence** $O_i \cap O_j \neq \emptyset$
 - Si: **R1** <- ...;; Sj: **R1** <- ...
 - At the instruction level, this is a **WAW** hazard if instruction j finishes first
- In both cases, not really a dependence but a “naming” problem
 - **Register renaming** (compiler by register allocation, in hardware see later)

Static vs. dynamic scheduling

- Assumptions (for now):
 - 1 instruction issue / cycle (Same techniques will be used when we look at multiple issue)
 - Several pipelines with a common IF and ID
 - Ideal CPI still 1, but real CPI won't be 1 but will be closer to 1 than before
- **Static scheduling** (optimized by compiler)
 - When there is a stall (hazard) no further issue of instructions
 - Of course, the stall has to be enforced by the hardware
- **Dynamic scheduling** (enforced by hardware)
 - Instructions following the one that stalls can issue **if they do not produce structural hazards or dependencies**

Dynamic scheduling

- Implies possibility of:
 - Out of order issue (we say that an instruction is issued once it has passed the ID stage) and hence out of order execution
 - Out of order completion (also possible in static scheduling but less frequent)
 - Imprecise exceptions (will take care of it later)
- Example (different pipes for add/sub and divide)
 - $R1 = R2 / R3$ (long latency)
 - $R2 \rightarrow R1 + R5$ (stall, **no issue**, because of RAW on R1)
 - $R6 = R7 - R8$ (can be **issued, executed** and **completed** before the other 2)

How would static scheduling optimize for this example?

Issue and Dispatch

- Split the ID stage into:
 - *Issue* : decode instructions; check for structural hazards and maybe more hazards such as WAW depending on implementations. Stall if there are any. Instructions pass in this stage in order
 - *Dispatch*: wait until no data hazards then read operands. At the next cycle a functional unit, i.e. EX of a pipe, can start executing

- Example revisited.

$R1 = R2 / R3$ (long latency; in execution)

$R2 = R1 + R5$ (*issue* but *no dispatch* because of RAW on R1)

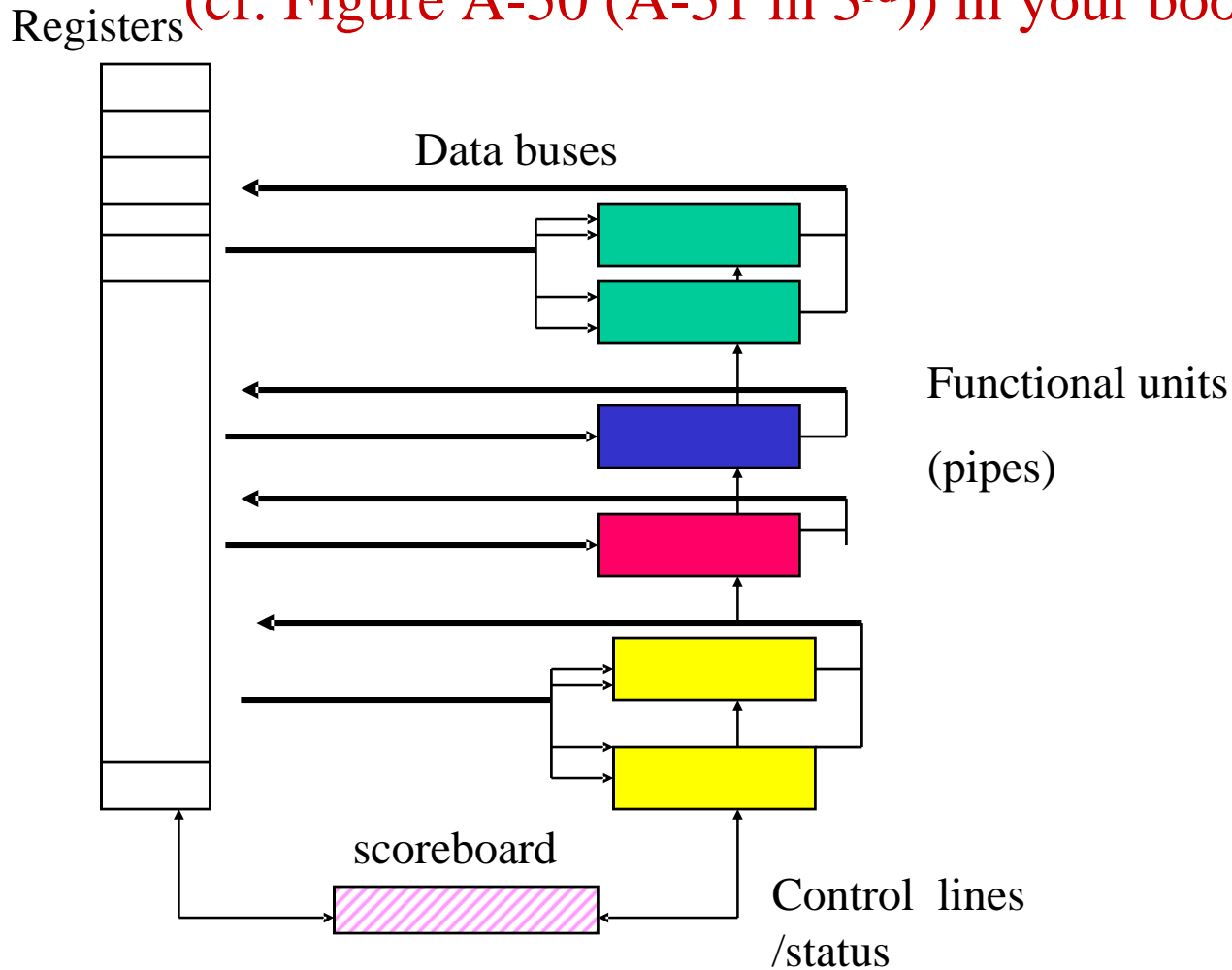
$R6 = R7 - R8$ (can be *issued, dispatched, executed* and *completed* before the other 2)

Implementations of dynamic scheduling

- In order to compute correct results, need to keep track of :
 - execution unit (free or busy)
 - register usage for read and write
 - completion etc.
- Two major techniques
 - **Scoreboard** (invented by Seymour Cray for the CDC 6600 in 1964)
 - **Tomasulo's algorithm** (used in the IBM 360/91 in 1967)

Scoreboarding -- The example machine

(cf. Figure A-50 (A-51 in 3rd) in your book)



Scoreboard basic idea

- The scoreboard keeps a record of **all data dependencies**
 - Keeps track of which registers are used as sources and destinations and which functional units use them
- The scoreboard keeps a record of **all pipe occupancies**
 - The original CDC 6600 was not pipelined but conceptually the scoreboard does not depend on pipelining
- The scoreboard decides if an **instruction can be issued**
 - Either the first time it sees it (no hazard) or, if not, at every cycle thereafter
- The scoreboard decides if an **instruction can store its result**
 - This is to prevent WAR hazards

An instruction goes through 5 steps

- We assume that the instruction has been successfully **fetched and decoded** (no I-cache miss)
- 1. **Issue**
 - The execution unit for that instruction type must be *free* (no structural hazard)
 - There should be **no WAW** hazard
 - If either of these conditions is false the instruction stalls. No further issue is allowed
 - There can be more fetches if there is an instruction fetch buffer (like there was in the CDC 6660)

Execution steps under scoreboard control

- 2. **Dispatch** (Read operands)
 - When the instruction is issued, the execution unit is reserved (becomes *busy*)
 - Operands are read in the execution unit when they are **both** ready (i.e., are not results of still executing instructions). **This prevents RAW hazards** (this conservative approach was taken because the CDC 6600 was not pipelined)
- 3. **Execution**
 - One or more cycles depending on functional unit latency
 - When execution completes, the unit notifies the scoreboard it's ready to write the result

Execution steps under scoreboard control (c'ed)

- 4. Write result
 - Before writing, check for WAR hazards. If one exists, the unit is stalled until all WAR hazards are cleared (note that an instruction in progress, i.e., whose operands have been read, won't cause a WAR)
- 5. Delay (you can forget about this one)
 - Because forwarding is not implemented, there should be one unit of delay between writing and reading the same register (this restriction seems artificial to me and is “historical”).
 - Similarly, it takes one unit of time between the release of a unit and its possible next occupancy


Optimizations and Simplifications


- There are opportunities for optimization such as:
 - Forwarding
 - Buffering for one copy of source operands in execution units (this allows reading of operands one at a time and minimizing the WAR hazards)
- We have assumed that there could be concurrent updates to (different) registers.
 - Can be solved (dynamically) by grouping execution units together and preventing concurrent writes in the same group or by having multiple write ports in the register file (expensive but common nowadays)

What is needed in the scoreboard (slightly redundant info)

- Status of each functional unit
 - Free or busy
 - Operation to be performed
 - The names of the result F_i and source F_j, F_k registers
 - Flags R_j, R_k indicating whether the source registers are ready
 - Names Q_j, Q_k of the units (if any) producing values for F_j, F_k
- Status of result registers
 - For each F_i the name of the unit (if any), say P_i that will produce its contents (redundant but easy to check)
- The instruction status
 - Been issued, dispatched, in execution, ready to write, finished?

Condition checking and scoreboard setting

- 
- Issue step
 - Unit free, say Ua and no WAW
 - Dispatch (Read operand) step
 - Rj and Rk must be yes (results ready)
 - Execution step
 - At end ask for writing permission (no WAR)
 - Write result
 - Check if Pi is an $Fj, Fk(Rj, Rk=no)$ in preceding instrs. If yes stall.

- 
- Issue step
 - Ua busy and record Fi, Fj, Fk
 - Record Qj, Qk and Rj, Rk
 - Record $Pi = Ua$
 - Dispatch (Read operand) step
 - Execution step
 - Write result
 - For subsequent instrs, if $Qj(Qk) = Ua$, set $Rj(Rk)$ to yes
 - Ua free and $Pi = 0$

Example

Load F6, 34(r2)

Load F2, 45(r3)

MulF F0, F2, F4

Sub F8, F6, F2

DivF F10, F0, F6

Add F6, F8, F2

Load f-p register F6

Load latency 1 cycle

Mult latency 10 cycles

Add/sub latency 2 cycles

Divide latency 40 cycles

—————> RAW

- - - -> WAR

Assume that the 2 Loads have been issued, the first one completed, the second ready to write. The next 3 instructions have been issued (but not dispatched).

Instruction	Issue	Dispatch	Executed	Result written
Load F6, 34(r2)	yes	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes	
Mul F0, F2, F4	yes			
Sub F8, F6, F2	yes			
Div F10, F0, F6	yes			

Add F6,F8,F2

Functional Unit status

No	Name	Busy	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Int	yes	F2	r3					
2	Mul	yes	F0	F2	F4	1		No	Y
3	Mul	no							
4	Add	yes	F8	F6	F2		1	Y	No
5	Div	yes	F10	F0	F6	2		No	Y

Register result status

F0 (2) F2 (1) F4 () F6() F8 (4) F10 (5) F12 ...

Instruction	Issue	Dispatch	Executed	Result written
Load F6, 34(r2)	yes	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes	yes
Mul F0, F2, F4	yes	yes		
Sub F8, F6, F2	yes	yes		
Div F10, F0, F6	yes			
Add F6,F8,F2				

1 cycle after 2nd load has written its result

Functional Unit status

No	Name	Busy	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Int	no							
2	Mul	yes	F0	F2	F4			Y	Y
3	Mul	no							
4	Add	yes	F8	F6	F2			Y	Y
5	Div	yes	F10	F0	F6	2		No	Y

Register result status

F0 (2)	F2 ()	F4 ()	F6 ()	F8 (4)	F10 (5)	F12 ...
--------	--------	--------	--------	--------	---------	---------

Instruction	Issue	Dispatch	Executed	Result written
Load F6, 34(r2)	yes	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes	yes
Mul F0, F2, F4	yes	yes	in progress	
Sub F8, F6, F2	yes	yes	yes	yes
Div F10, F0, F6	yes			
Add F6,F8,F2	yes	yes	yes	

Functional Unit status										
No	Name	Busy	Fi	Fj	Fk	Qj	Qk	Rj	Rk	
1	Int	no								6 cycles later; Mul in execution; Sub has completed; Div issues; Add waits for writing
2	Mul	yes	F0	F2	F4			Y	Y	
3	Mul	no								
4	Add	yes	F6	F8	F2			Y	Y	
5	Div	yes	F10	F0	F6	2		No	Y	

Register result status						
F0 (2)	F2 ()	F4 ()	F6(4)	F8 ()	F10 (5)	F12 ...

Instruction	Issue	Dispatch	Executed	Result written
Load F6, 34(r2)	yes	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes	yes
Mul F0, F2, F4	yes	yes	yes	yes
Sub F8, F6, F2	yes	yes	yes	yes
Div F10, F0, F6	yes	yes		
Add F6,F8,F2	yes	yes	yes	

Functional Unit status

No	Name	Busy	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Int	no							
2	Mul	no							
3	Mul	no							
4	Add	yes	F6	F8	F2			Y	Y
5	Div	yes	F10	F0	F6			Y	Y

4 cycles later (I think!)

Mul is finished; Div can dispatch; Add will write at next cycle

Register result status

F0 () F2 () F4 () F6(4) F8 () F10 (5) F12 ...

Instruction	Issue	Dispatch	Executed	Result written
Load F6, 34(r2)	yes	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes	yes
Mul F0, F2, F4	yes	yes	yes	yes
Sub F8, F6, F2	yes	yes	yes	yes
Div F10, F0, F6	yes	yes		
Add F6,F8,F2	yes	yes	yes	yes

Functional Unit status

No	Name	Busy	Fi	Fj	Fk	Qj	Qk	Rj	Rk	1 cycle later. Only Div is not finished
1	Int	no								
2	Mul	no								
3	Mul	no								
4	Add	no								
5	Div	yes	F10	F0	F6			Y	Y	

Register result status

F0 ()	F2 ()	F4 ()	F6 ()	F8 ()	F10 (5)	F12 ...
--------	--------	--------	--------	--------	---------	---------

Tomasulo's algorithm

- “Weaknesses” in scoreboard:
 - Centralized control
 - No forwarding (more RAW than needed)
 - No buffering
- Tomasulo's algorithm as implemented first in IBM 360/91
 - Control decentralized at each functional unit
 - Forwarding
 - Concept and implementation of *renaming registers* that eliminates WAR and WAW hazards

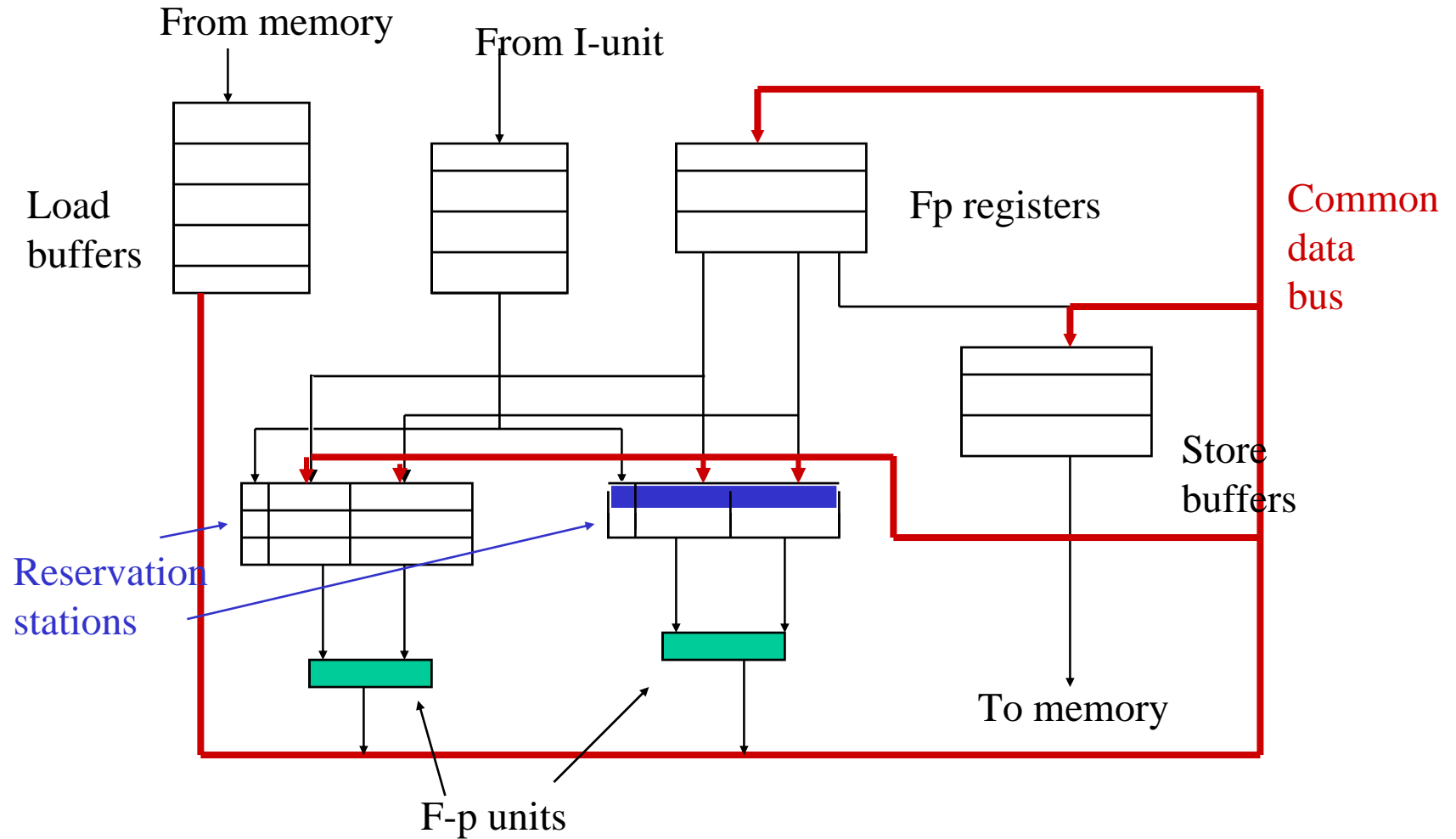
Improving on Dispatch with Reservation Stations

- With each functional unit, have a set of buffers or *reservation stations*
 - Keep operands and function to perform
 - Operands can be **values** or **names** of units that will produce the value (**register renaming**) with appropriate flags
- Not both operands have to be “ready” at the same time
- When both operands have values, functional unit can execute on that pair of operands
- When a functional unit computes a result, it **broadcasts** its name and the value.

Tomasulo's solution to resolve hazards

- **Structural hazards**
 - No free reservation station (stall at issue time). No further issue
- **RAW dependency** (detected in each functional unit -- decentralized)
 - The instruction with the dependency is issued (put in a reservation station) but not dispatched (stalled). Subsequent instructions can be issued, dispatched, executed and completed.
- **No WAR and WAW hazards**
 - Because of register renaming through reservation stations
- **Forwarding**
 - Done at end of execution by use of a common (broadcast) data bus

Example machine (cf. Figure 2.9 (3.2 3rd))



An instruction goes through 3 steps

- Assume the instruction has been fetched
- 1. Issue, dispatch, and read operands
 - Check for structural hazard (no free reservation station or no free load-store buffer for a memory operation). If there is structural hazard, stall until it is not present any longer
 - Reserve the next reservation station
 - Read source operands
 - If they have values, put the values in the reservation station
 - If they have names, store their names in the reservation station
 - Rename result register with the name of the reservation station in the functional unit that will compute it

An instruction goes through 3 steps (c'ed)

- 2. **Execute**
 - If any of the source operands is not ready (i.e., the reservation station holds at least one name), monitor the bus for broadcast
 - When both operands have values, execute
- 3. **Write result**
 - Broadcast name of the unit and value computed. Any reservation station/result register with that name grabs the value
- Note two more sources of **structural hazard** due to **contention**:
 - Two reservation stations in the same functional unit are ready to execute in the same cycle: choose the “first” one
 - Two functional units want to broadcast at the same time. Priority is encoded in the hardware

Implementation

- All registers (except load buffers) contain a pair {value,tag}
- The tag (or name) can be:
 - Zero (or a special pattern) meaning that the value is indeed a value
 - The name of a load buffer
 - The name of a reservation station within a functional unit
- A reservation station consists of :
 - The operation to be performed
 - 2 pairs (value,tag) (V_j, Q_j) (V_k, Q_k)
 - A flag indicating whether the accompanying f-u is busy or not

Instruction	Issue	Execute	Write result
Load F6, 34(r2)	yes	yes	yes
Load F2, 45(r3)	yes	yes	
Mul F0, F2, F4	yes		
Sub F8, F6, F2	yes		
Div F10, F0, F6	yes		
Add F6,F8,F2	yes		

Initial: waiting for F2 to be loaded from memory

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	yes	Sub	(Load1)		0	Load2
Add2	yes	Add			Add1	Load2
Add3	no					
Mul1	yes	Mul		(F4)	Load2	0
Mul2	yes	Div		(Load1)	Mul1	0

(x) Means a value: contents of x

Qj = 0 means Vj has a value

Register status

F0 (Mul1) F2 (Load2) F4 () F6(Add2) F8 (Add1) F10 (Mul2) F12...

Instruction	Issue	Execute	Write result
Load F6, 34(r2)	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes
Mul F0, F2, F4	yes	yes	
Sub F8, F6, F2	yes	yes	
Div F10, F0, F6	yes		
Add F6,F8,F2	yes		

Cycle after 2nd
load has written
its result

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	yes	Sub	(Load1)	(Load2)	0	0
Add2	yes	Add		(Load2)	Add1	0
Add3	no					
Mul1	yes	Mul	(Load2)	(F4)	0	0
Mul2	yes	Div		(Load1)	Mul1	0

Register status

F0 (Mul1) F2 () F4 () F6(Add2) F8 (Add1) F10 (Mul2) F12...

Instruction	Issue	Execute	Write result
Load F6, 34(r2)	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes
Mul F0, F2, F4	yes	yes	
Sub F8, F6, F2	yes	yes	yes
Div F10, F0, F6	yes		
Add F6,F8,F2	yes		

Cycle after sub has written its result

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	no					
Add2	yes	Add	(Add1)	(Load2)	0	0
Add3	no					
Mul1	yes	Mul	(Load2)	(F4)	0	0
Mul2	yes	Div		(Load1)	Mul1	0

Register status

F0 (Mul1) F2 () F4 () F6(Add2) F8 () F10 (Mul2) F12...

Instruction	Issue	Execute	Write result
Load F6, 34(r2)	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes
Mul F0, F2, F4	yes	yes	
Sub F8, F6, F2	yes	yes	yes
Div F10, F0, F6	yes		
Add F6,F8,F2	yes	yes	yes

Cycle after add has written its result

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	no					
Add2	no					
Add3	no					
Mul1	yes	Mul	(Load2)	(F4)	0	0
Mul2	yes	Div		(Load1)	Mul1	0

Register status

F0 (Mul1) F2 () F4 () F6() F8 () F10 (Mul2) F12...

Instruction	Issue	Execute	Write result
Load F6, 34(r2)	yes	yes	yes
Load F2, 45(r3)	yes	yes	yes
Mul F0, F2, F4	yes	yes	yes
Sub F8, F6, F2	yes	yes	yes
Div F10, F0, F6	yes	yes	
Add F6,F8,F2	yes	yes	yes

Cycle after mul
has written its
result

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	no					
Add2	no					
Add3	no					
Mul1	no					
Mul2	yes	Div (Mul1)	(Load1)	0	0	0

Register status

F0 () F2 () F4 () F6() F8 () F10 (Mul2) F12...

Other checks/possibilities

- In the example in the book there is no load/store dependencies but since they can happen
 - Load/store buffers must keep the addresses of the operands
 - On load, check if there is a corresponding address in store buffers. If so, get the value/tag from there (load/store buffers have tags)
 - Better yet, have load/store functional units (still needs checking)
- The Tomasulo engine was intended only for f-p operations. We need to generalize to include
 - Handling branches, exceptions etc
 - In-order completion
 - More general register renaming mechanisms
 - Multiple instruction issues

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.