Static vs. dynamic scheduling

- Assumptions (for now):
  - 1 instruction issue / cycle
    - Ideal CPI still 1, but real CPI will be closer to 1
    - Same techniques will be used when we look at multiple issue
  - Several pipelines with a common IF and ID
- 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 = R1 + R5 (stall, no issue, because of RAW on R1)
R6 = R7 - R8 (can be issued, executed and completed before the other 2)

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.

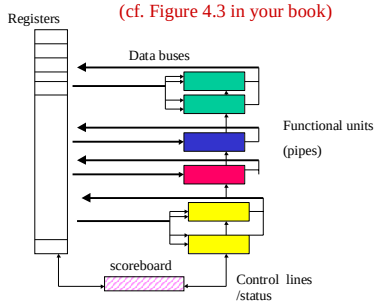
R1 = R2/ R3 (long latency; in execution)
R2 = R1 + R5 (issue but no dispatch because of RAW on R1)
R6 = R7 - R8 (can be issued, executed and completed before the other 2)

Implementations of dynamic scheduling

- In order to compute correct results, need to keep track of :
  - execution pipes
  - 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 4.3 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
- 1. Issue
  - The execution unit 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

What is needed in the scoreboard

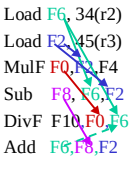
- Status of each functional unit
  - Free or busy
  - Operation to be performed
  - The names of the result *Fi* and source *Fj, Fk* registers
  - Flags *Rj, Rk* indicating whether the source registers are ready
  - Names *Qj,Qk* of the units (if any) producing values for *Fj, Fk*
- Status of result registers
  - For each *Fi* the name of the unit (if any), say *Pi* 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 free
- 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 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							
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

6 cycles later; Mul in execution; Sub has completed;Div issues; Add waits for writing

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	Int	no							
2	Mul	no							
3	Mul	no							
4	Add	no							
5	Div	yes	F10	F0	F6			Y	Y

1 cycle later. Only Div is not finished

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)
- 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

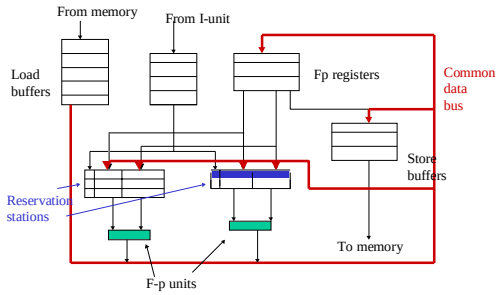
Reservation stations

- 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 stalled (but others might be issued)
- 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 4.8)



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 one, 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 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 a name), monitor the bus for broadcast of a result
  - 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:
  - 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) *(Vj,Qj) (Vk,Qk)*
  - 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)			Load2
Add2	yes	Add			Add1	Load2
Add3	no					
Mul1	yes	Mul		(F4)	Load2	
Mul2	yes	Div		(Load1)	Mul1	

(x) Means a value: contents of x

Register status

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

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			Cycle after 2nd load has written its result

Reservation Stations

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

Register status

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

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	Cycle after sub has written its result
Div F10, F0, F6	yes			
Add F6,F8,F2	yes			

Reservation Stations

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

Register status

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

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	Cycle after add has written its result
Div F10, F0, F6	yes			
Add F6,F8,F2	yes	yes	yes	

Reservation Stations

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

Register status

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

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	Cycle after mul has written its result
Div F10, F0, F6	yes	yes		
Add F6,F8,F2	yes	yes	yes	

Reservation Stations

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

Register status

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

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 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