Review

- Last Class:
  - Static parallelism (VLIW)
- Today's class:
  - Dynamic parallelism
- Announcement and reminder
  - First reading assignment will be posted on canvas today.
  - The logistics of reading report will be posted. Please read it carefully before writing your report.

(94)

Review: Multiple Instruction Issue Possibilities

- Fetch and issue **more than one** instruction in a cycle

1. **Statically-scheduled (in-order)**
   - **Very Long Instruction Word (VLIW)** e.g., TransMeta (4-wide)
     - Compiler figures out what can be done in parallel, so the hardware can be dumb and low power
     - Compiler must group parallel instr's, requires new binaries
   - **SuperScalar** e.g., Pentium (2-wide), ARM CortexA8 (2-wide)
     - Hardware figures out what can be done in parallel
     - Executes unmodified sequential programs
   - **Explicitly Parallel Instruction Computing (EPIC)** e.g., Intel Itanium (6-wide)
     - A compromise: compiler does some, hardware does the rest
2. **Dynamically-scheduled (out-of-order) SuperScalar**
   - Hardware dynamically determines what can be done in parallel (can extract much more ILP with OOO processing)
   - E.g., Intel Pentium Pro/II/III (3-wide), Core i7 (4 cores, 4-wide, SMT2), IBM Power5 (5-wide), Power8 (12 cores, 8-wide, SMT8)

(95)

Why OOO?

- Consider the following instruction sequence:

  I1
  I2
  I3

- If we do not employ OOO execution and I2 is stalled (if, for example, it depends on I1), I3 will be stalled as well
- An instruction is stalled because of an *irrelevant instruction*
  - A consequence of in-order execution
- Solution: Let I3 get scheduled and execute while I2 is waiting – out-of-order execution
  - Improves utilization and performance

(96)

96

Why OOO?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2          LD    R3 ← R1 (0)
ADD   R3 ← R3, R1          ADD   R3 ← R3, R1
ADD   R9 ← R6, R7          ADD   R9 ← R6, R7
IMUL  R5 ← R6, R8          IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5          ADD   R7 ← R3, R5
```

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed
- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture
  - Can compiler resolve the issue?

(97)

97

Out-of-order Execution (Dynamic Scheduling)

- Idea: Move the dependent instructions out of the way of independent ones
  - Rest areas for dependent instructions: Reservation Stations
- Monitor the source “values” of each instruction in the resting area
- When all source “values” of an instruction are available, “fire” (i.e. issue) the instruction
  - Instructions dispatched in dataflow (not control-flow) order
- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation

(98)

Advantages of Dynamic Scheduling

- Allows code that was compiled with one pipeline in mind to run efficiently on a *different* pipeline
  - Eliminates need to have multiple binaries and recompile
- Enables handling some cases when dependences are unknown at compile time
  - E.g., may involve memory reference or data-dependent branch
- Allows the processor to tolerate unpredictable delays
  - E.g., cache misses

(99)

Dynamic OOO Datapaths

- Scoreboarding – CDC 6600 (Thornton) first publication in 1964
  - Named after CDC 6600 scoreboard
  - Used **centralized** hazard detection logic (scoreboard) to support OOO execution. Instr's were stalled when their FU was busy, for RAW dependencies, *and* for WAW and WAR dependencies
- Tomasulo – IBM 360/91 (Tomasulo) first publication in 1967
  - Used **distributed** hazard detection logic (reservation stations feeding each FU) to support OOO execution with register renaming that eliminated WAW and WAR dependencies; distributed results from FUs to reservation stations on a Common Data Bus (potential bottleneck)
  - Writes results to register file and memory when instr's completes – possibly out-of-order – so could *not* support precise interrupts or speculative execution (e.g., branch speculation)

(100)

More Recent Dynamic OOO Datapaths

- HPS – (Hwu, Patt, Shebanow) first publication in 1985
  - Used a register alias table and distributed node alias tables that fed each FUs (essentially reservation stations) to support OOO execution with register renaming; distributed results from FUs to reservation stations on multiple distribution buses (one per FU)
  - Supported precise interrupts and speculative execution with a checkpoint repair mechanism
- RUU – (Sohi) first publication in 1987
  - Uses a centralized Register Update Unit (RUU) that 1) receives new instr's from decode, 2) renames registers, 3) monitors the (single) result bus to resolve dependencies, 4) determines when instr's are ready to issue (send for execution), and 5) holds completed instr's until they can commit
  - Supports precise interrupts and speculative execution via in-order commit out of the RUU
    - For precise interrupts and branch speculation, need to do commit in-order, so need additional resources to keep track of results that have been written, but not yet committed

(101)

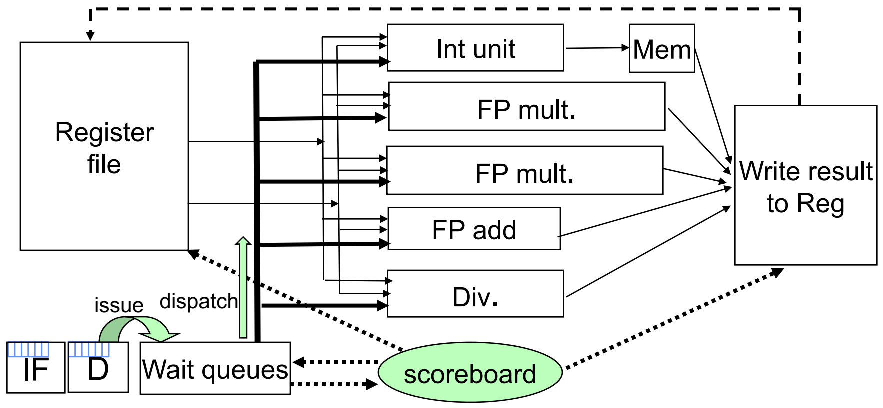
Dynamically scheduled pipelines (§3.4 - 3.5)

Using Scoreboards (see Appendix C.7):

- Dates to the first supercomputer, the CDC 6600 in 1963
- Split the ID stage into
  - Issue - decode and check for structural hazards,
  - Read operands → wait until no data hazards, then read operands.
- Instructions wait in a queue and may move to the EX stage (dispatched) out of order.

(102)

A scoreboard architecture



- The scoreboard is responsible for instruction issue and execution, including hazard detection. It is also controlling the writing of the results.
- The "scoreboard" consists of 3 tables to keep track of execution progress and the associated intelligence to determine when to dispatch instructions
- One entry (buffer) in the "wait queue" is associated with each functional unit.

(103)

Scoreboard information (3 tables)

- **Instruction status**:
  - issued, read operands and started execution (dispatched), completed execution or wrote result,
- **Functional unit status** (assuming non-pipelined units)
  - busy/not busy
  - operation (if unit can perform more than one operation)
  - destination register - F_i
  - source registers (containing source operands) - F_j and F_k
  - the unit producing the source operands (if stall to avoid RAW hazards) - Q_j and Q_k
  - flags to indicate that source operands are ready -- R_j and R_k
- **Register result status**:
  - indicates the functional unit that contains an instruction which will write into each register (if any)

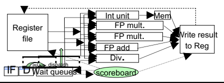
(104)

Four stages of scoreboard control

- **Issue only if no structural, WAR or WAW hazards.**
  - Issue (and reserve the functional unit) if the functional unit is free and
    - » No issued or dispatched instruction (in state “issued” or “dispatched”) will write to the destination register (to avoid WAW)
    - » No issued instruction (in state “issued”) will read from the destination register (to avoid WAR)
  - otherwise, stall, and block subsequent instructions
  - the fetch unit stalls when the queue between the fetch and the issue stages is full (may be only one buffer).
- **Read operands only if no RAW hazard.**
  - If a RAW hazard is detected, wait until the operands are ready,
  - When the operands are ready, read the registers and move to the execution stage,
  - Note that instructions may proceed to the EX stage out-of-order.
- **Execution.**
  - When execution terminates, notify the score board.
- **Write result to register file**

(105)

Example:



Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result	
fld	F6, 34(R2)	X	X	X	X	done
fld	F2, 45(R3)	X	X	X		
fmul.d	F0, F2, F4	X				
fsub,d	F8, F6, F2	X				
fdiv.d	F10, F0, F12	X				
fadd.d	F6, F8, F2					Not in

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Int-m	Yes	Load	F2	R3				Yes	
Mult1	Yes	Mult	F0	F2	F4	Int.		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Int.	Yes	No
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1	Int.			Add	Div			

🟩 Redundant information

(106)

Example: when “fld F2, 45(R3)” is writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result	
fld	F6, 34(R2)	X	X	X	X	done
fld	F2, 45(R3)	X	X	X	X	
fmul.d	F0, F2, F4	X				
fsub,d	F8, F6, F2	X				
fdiv.d	F10, F0, F12	X				
fadd.d	F6, F8, F2					Not in

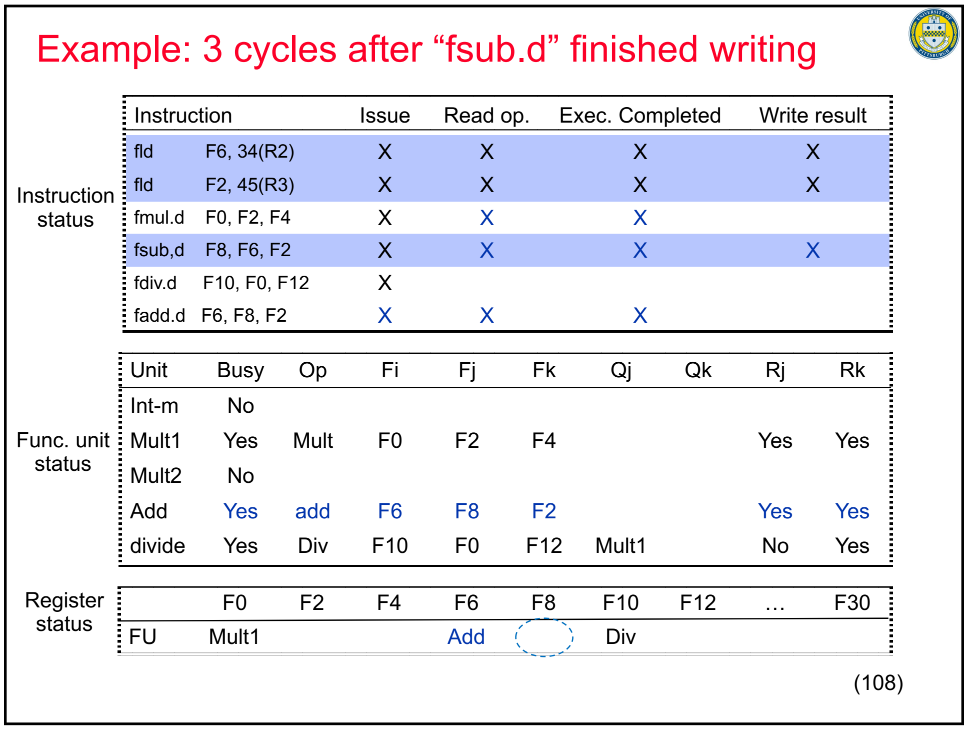
Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Int-m	Yes	Load	F2	R3				Yes	
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2			Yes	Yes
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1				Add	Div			

(107)



Example: 3 cycles after “fsub.d” finished writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result
fld	F6, 34(R2)	X	X	X	X
fld	F2, 45(R3)	X	X	X	X
fmul.d	F0, F2, F4	X	X	X	
fsub,d	F8, F6, F2	X	X	X	X
fdiv.d	F10, F0, F12	X			
fadd.d	F6, F8, F2	X	X	X	

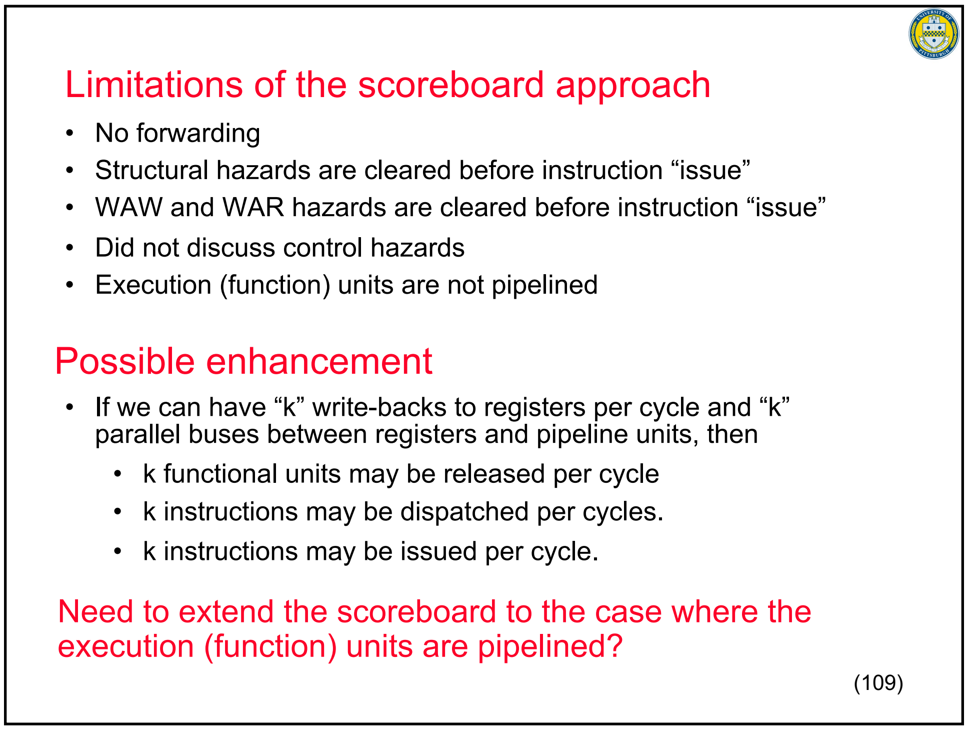
Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Int-m	No								
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	add	F6	F8	F2			Yes	Yes
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1			Add		Div			

(108)



Limitations of the scoreboard approach

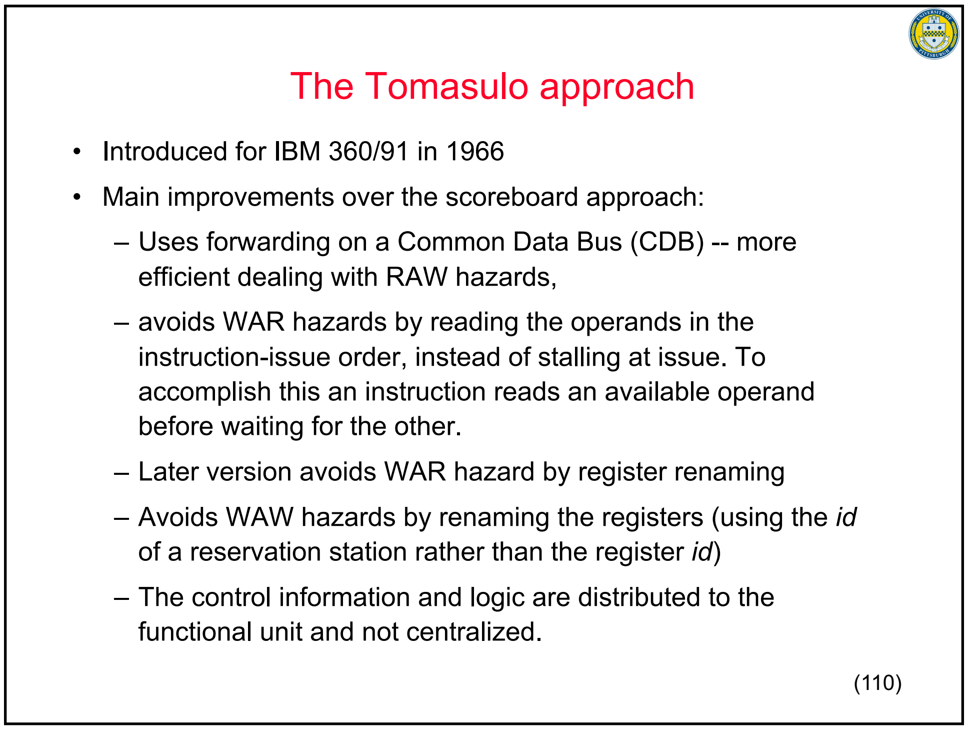
- No forwarding
- Structural hazards are cleared before instruction “issue”
- WAW and WAR hazards are cleared before instruction “issue”
- Did not discuss control hazards
- Execution (function) units are not pipelined

Possible enhancement

- If we can have “k” write-backs to registers per cycle and “k” parallel buses between registers and pipeline units, then
  - k functional units may be released per cycle
  - k instructions may be dispatched per cycles.
  - k instructions may be issued per cycle.

Need to extend the scoreboard to the case where the execution (function) units are pipelined?

(109)

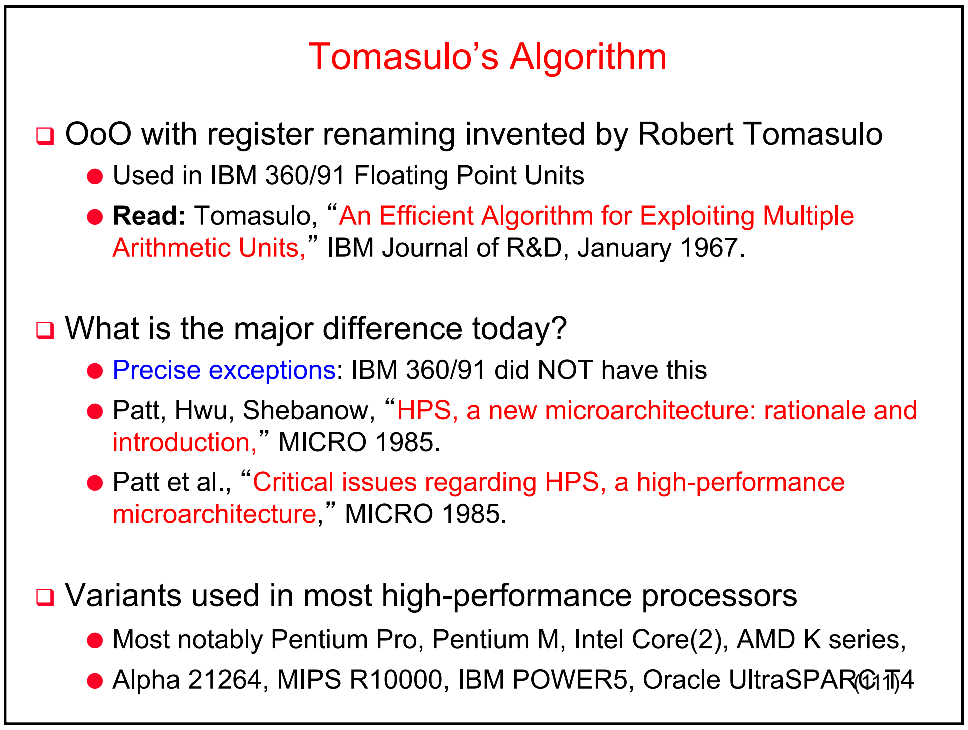


The Tomasulo approach

- Introduced for IBM 360/91 in 1966
- Main improvements over the scoreboard approach:
  - Uses forwarding on a Common Data Bus (CDB) -- more efficient dealing with RAW hazards,
  - avoids WAR hazards by reading the operands in the instruction-issue order, instead of stalling at issue. To accomplish this an instruction reads an available operand before waiting for the other.
  - Later version avoids WAR hazard by register renaming
  - Avoids WAW hazards by renaming the registers (using the *id* of a reservation station rather than the register *id*)
  - The control information and logic are distributed to the functional unit and not centralized.

(110)

110



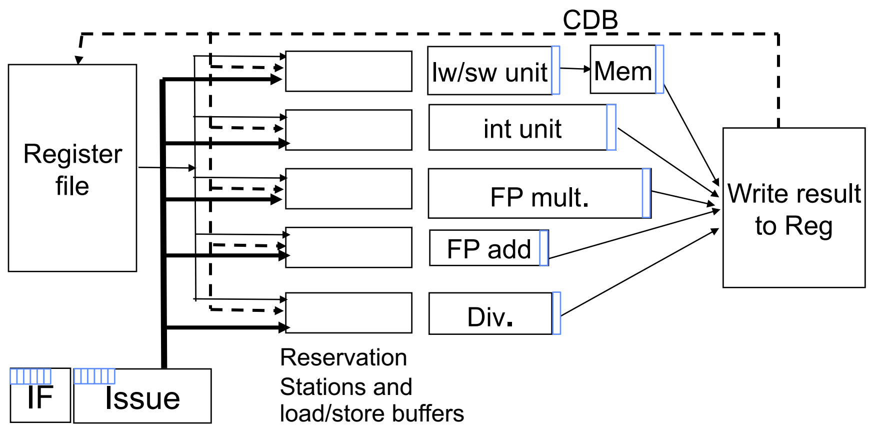
Tomasulo's Algorithm

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - **Read:** Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, January 1967.
- What is the major difference today?
  - Precise exceptions: IBM 360/91 did NOT have this
  - Patt, Hwu, Shebanow, "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.
  - Patt et al., "Critical issues regarding HPS, a high-performance microarchitecture," MICRO 1985.
- Variants used in most high-performance processors
  - Most notably Pentium Pro, Pentium M, Intel Core(2), AMD K series,
  - Alpha 21264, MIPS R10000, IBM POWER5, Oracle UltraSPARC T4

(111)

111

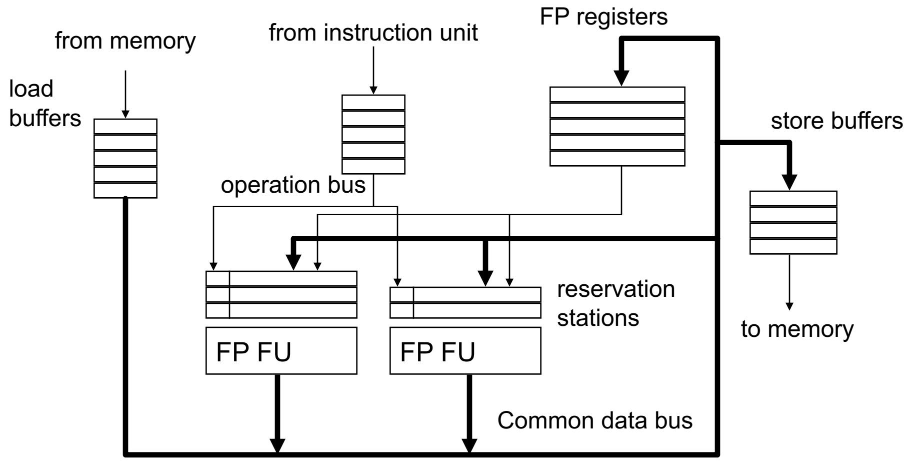
The architecture for the Tomasulo scheme



- Each reservation station has an *id* and is used by one instruction during the lifetime of this instruction.
- Each unit has one or more reservation stations
- Reservation stations play the role of temporary registers (renaming)

(112)

Tomasulo's Machine: IBM 360/91



(113)

Book keeping in Tomasulo's algorithm

- **Instruction status**:
  - issued, executing or writing result,
- **Reservation stations (functional units) status**:
  - busy/not busy
  - operation (if unit can perform more than one operation)
  - source operands (data values) - V_j and V_k
  - the *reservation stations* producing the source operands (if stall to avoid RAW hazards) - Q_j and Q_k
  - Address field, *A*, for load/store buffers (store effective address)
- **Register result status**:
  - indicates the *reservation station* that contains an instruction which will write into each register (if any)

Note that the first two tables can be combined

(114)

114

Three stages of control

- **Issue**
  - If a reservation station is available for the needed functional unit
    - » read ready operands
    - » for operands that are not ready, rename the register to the reservation station that will produce it,
  - Reservation stations for load/store instructions are called load/store buffers.
- **Execution.**
  - Monitor the CDB for the operand that is not ready,
  - When both operands are available, execute.
  - If more than one station per unit, only one station can start execution.
  - Do not start execution before previous branches have completed.
- **Write result.**
  - Write to CDB (and to registers) -- may be a structural hazards if only one CDB bus.
  - Make the reservation station (the functional unit) available.

(115)

115

Load and store instructions:

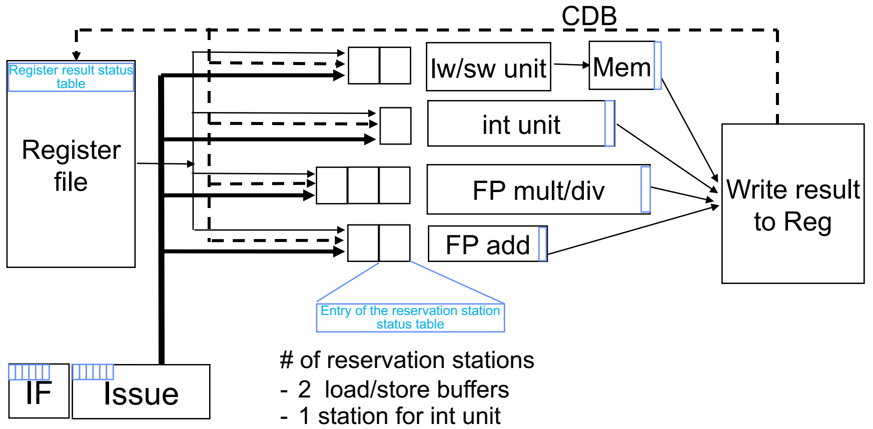
- Uses load/store buffers, and each buffer is like a reservation station.
- Address calculation (put result in buffer), then memory operation
- The result of a load is put on the CDB
- Stores are executed in program order (loads in any order)
- Performs memory disambiguation between store and load buffers,

Remarks:

- May have more reservation stations than registers (a large virtual register space)
- The original Tomasulo algorithm was introduced before caches were incorporated into commercial processors
- If more than one issued instruction writes into a register, only the last one does the actual write (no WAW hazards).

(116)

Example



# of reservation stations
- 2 load/store buffers
- 1 station for int unit
- 3 station for FP mult/div unit
- 2 station for FP add unit

(117)

Instruction		Issue	Execute	Write result
fld	F6, 34(R2)	X	X	
fld	F2, 45(R3)	X	X	
fmul.d	F0, F2, F4	X		
fsub.d	F8, F2, F6	X		
fdiv.d	F10, F0, F12	X		
fadd.d	F6, F8, F2	X		

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Y	Load					34+Reg[R2]
Load2	Y	Load					45+Reg[R3]
Add1	Y	Sub			Load2	Load1	
Add2	Y	Add			Add1	Load2	
Add3	no						
Mult1	Y	Mul		Reg[F4]	Load2		
Mult2	Y	Div		Reg[F12]	Mult1		
Int	no						

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1	load2		Add2	Add1	Mult2			

(118)

118

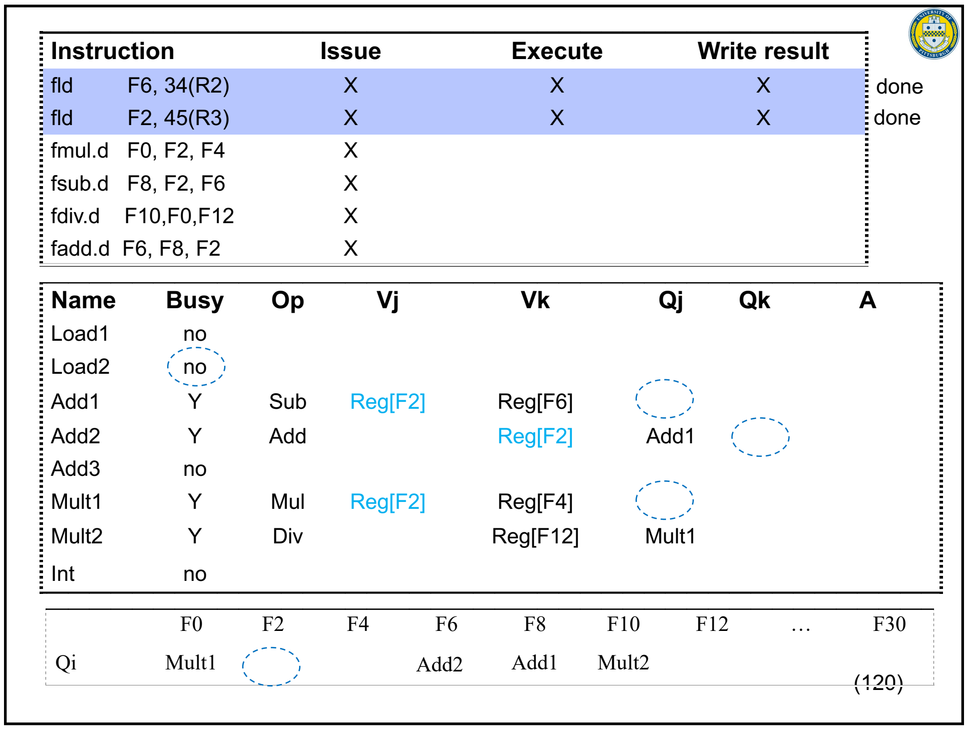
Instruction		Issue	Execute	Write result	
fld	F6, 34(R2)	X	X	X	done
fld	F2, 45(R3)	X	X		
fmul.d	F0, F2, F4	X			
fsub.d	F8, F2, F6	X			
fdiv.d	F10, F0, F12	X			
fadd.d	F6, F8, F2	X			

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	Y	Load					45+Reg[R3]
Add1	Y	Sub		Reg[F6]	Load2		
Add2	Y	Add			Add1	Load2	
Add3	no						
Mult1	Y	Mul		Reg[F4]	Load2		
Mult2	Y	Div		Reg[F12]	Mult1		
Int	no						

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1	load2		Add2	Add1	Mult2			

(119)

119

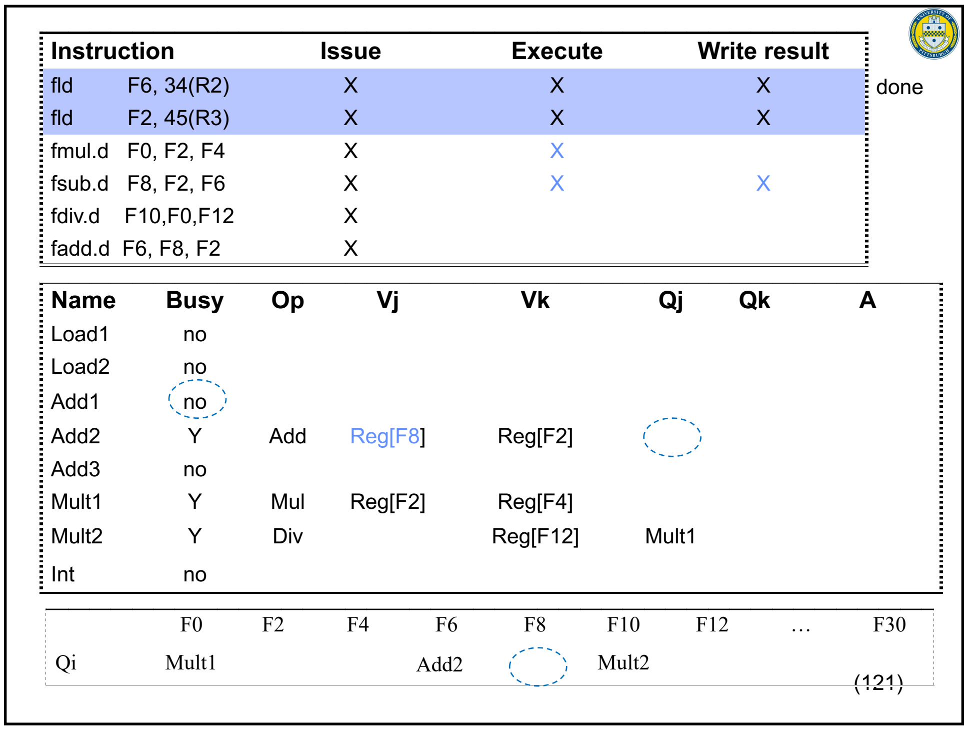


Instruction	Issue	Execute	Write result	
fld F6, 34(R2)	X	X	X	done
fld F2, 45(R3)	X	X	X	done
fmul.d F0, F2, F4	X			
fsub.d F8, F2, F6	X			
fdiv.d F10,F0,F12	X			
fadd.d F6, F8, F2	X			

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	Y	Sub	Reg[F2]	Reg[F6]			
Add2	Y	Add		Reg[F2]	Add1		
Add3	no						
Mult1	Y	Mul	Reg[F2]	Reg[F4]			
Mult2	Y	Div		Reg[F12]	Mult1		
Int	no						

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1			Add2	Add1	Mult2			

(120)



Instruction	Issue	Execute	Write result	
fld F6, 34(R2)	X	X	X	done
fld F2, 45(R3)	X	X	X	
fmul.d F0, F2, F4	X	X		
fsub.d F8, F2, F6	X	X	X	
fdiv.d F10,F0,F12	X			
fadd.d F6, F8, F2	X			

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	Y	Add	Reg[F8]	Reg[F2]			
Add3	no						
Mult1	Y	Mul	Reg[F2]	Reg[F4]			
Mult2	Y	Div		Reg[F12]	Mult1		
Int	no						

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1			Add2		Mult2			

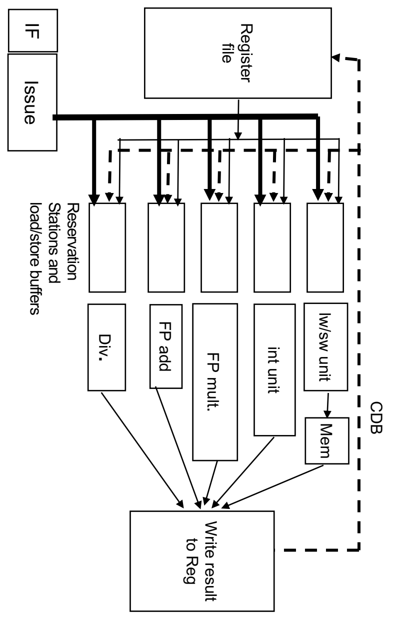
(121)

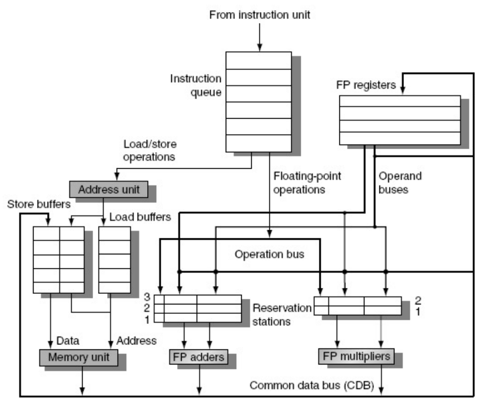
Instruction		Issue	Execute	Write result	
fld	F6, 34(R2)	X	X	X	done
fld	F2, 45(R3)	X	X	X	done
fmul.d	F0, F2, F4	X	X		
fsub.d	F8, F2, F6	X	X	X	done
fdiv.d	F10,F0,F12	X			
fadd.d	F6, F8, F2	X	X	X	

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	Y	Mul	Reg[F2]	Reg[F4]			
Mult2	Y	Div		Reg[F12]	Mult1		
Int	no						

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1					Mult2			

Figure 3.10


IF
Issue
Register file
Reservation Stations and load/store buffers
Div.
FP add
FP mult.
int unit
lw/sw unit
Mem
CDB
Write result to Reg


From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)