Review

- Last Class:
  - Scoreboard OOO execution
  - Tomasulo OOO execution
- Today's class:
  - Tomasulo speculative OOO execution
- Announcement and reminder
  - HW1 dues tonight 11:59pm. No late submission

(124)

Another example

Instruction		Issue	Execute	Write result
fld	F0, 0(R1)	X	X	
fmul.d	F4, F0, F2	X		
fsd	F4, 0(R1)	X		
fld	F0, 8(R1)			
fmul.d	F4, F0, F2			
fsd	F4, 8(R1)			

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	y	ld					Reg[R1]+0
Load2	no						
store1	y	sd	Reg[R1]			Mult1	
store2	no						
Add	no						
Mult1	y	Mult		Reg[F2]	Load1		
Mult2	no						

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Load1		Mult 1						

(125)

Another example

Instruction		Issue	Execute	Write result
fld	F0, 0(R1)	X	X	
fmul.d	F4, F0, F2	X		
fsd	F4, 0(R1)	X		
fld	F0, 8(R1)	X	X	
fmul.d	F4, F0, F2	X		
fsd	F4, 8(R1)	X		

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	y	ld					Reg[R1]+0
Load2	y	ld					Reg[R1]+8
store1	y	sd				Mult1	Reg[R1]+0
store2	y	sd	Reg[R1]			Mult2	
Add	no						
Mult1	Y	Mul		Reg[F2]	Load1		
Mult2	Y	Mul		Reg[F2]	Load2		

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Load2		Mult 2						

(126)

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

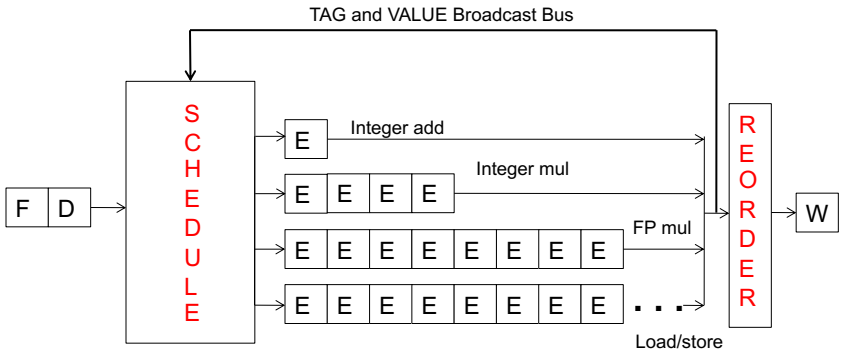
(127)

Hardware-based Speculation (§3.6)

- The goal is to allow instructions after a branch to start execution before the outcome of the branch is confirmed.
- There should be *no* consequences (including exceptions) if it is determined that the instruction should not execute.
  - Use dynamic branch prediction and use OOO execution
  - Use a Reorder Buffer (ROB) to reorder instructions after execution
  - Commit results to registers and memory in-order
  - All un-committed results can be flushed if a branch is miss-predicted
- Can free the reservation station when an instruction is in the reorder buffer.
- For each register, R, the register status table keeps the ROB number reserved by the instruction which will write into R (instead of the RES station number).

(133)

Two Humps in the Pipeline

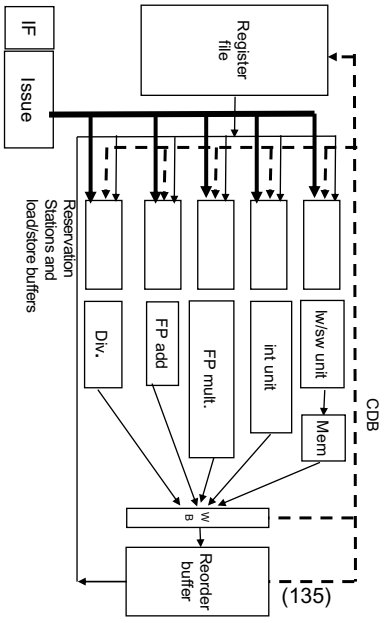


- **Hump 1:** Reservation stations (scheduling window)
- **Hump 2:** Reordering (reorder buffer)

(134)

Reorder buffers (ROB)

- 3 fields: instruction, destination, value
- When issuing a new instruction, read a register value from the ROB if the status table indicates that the source instruction is in a ROB.
- Hence, ROBs supply operands between execution complete & commit => more virtual registers.
- ROBs form a circular queue ordered in the "issue order".
- Once an instruction reaches the head of the ROB queue, it commits the results into register or memory.
- Hence, it's easy to undo speculated instructions on miss-predicted branches or on exceptions
- Should flush the pipe as soon as you discover a miss-predictions – all earlier instructions should commit



(135)

Steps of Speculative Tomasulo Algorithm

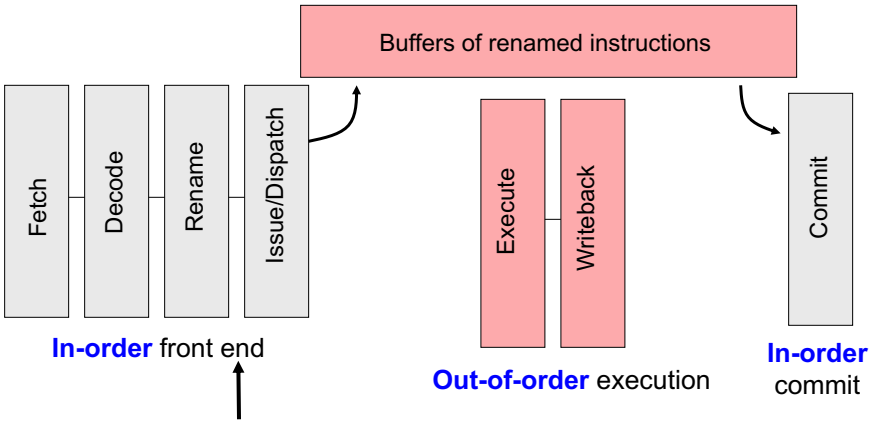
Combining branch prediction with dynamic scheduling

- **Issue** (sometimes called Dispatch)
  - If a RES station and a ROB are free, issue the instruction to the RES station after reading ready registers and renaming non-ready registers
- **Execution** (sometimes called issue)
  - When both operands are ready, then execute; if not ready, watch CDB for result; when both in reservation station, execute (checks RAW)
- **Write result** (WB)
  - Write on CDB to all awaiting RES stations & send the instruction to the ROB; mark reservation station available.
- **Commit** (sometimes called graduation)
  - When instruction is at head of ROB, update registers (or memory) with result and free ROB. A miss-predicted branch flushes all non-committed instructions.

(136)

Out-of-Order Pipeline Progress

- Have completed Fetch, Decode, Rename (in program order) and are ready to Dispatch



Instr's now have unique register names, so can now put into OOO execution structures

(137)

Example:

- Assume that *fmul.d* just finished WB and is entering the ROB
- *fsub.d* and *fadd.d* are already in the ROB
- *fdiv.d* is in RES station waiting for the result of *fmul.d*

Instruction		Issued	Execute	In ROB	committed	
fld	F6, 34(R2)	X	X	X	X	done
fld	F2, 45(R3)	X	X	X	X	done
fmul.d	F0, F2, F4	X	X			
fsub.d	F8, F2, F6	X	X	X		
fdiv.d	F10, F0, F6	X				
fadd.d	F6, F8, F2	X	X	X		

(138)

Reservation stations status

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest.	A
Load1	no							
Load2	no							
Add1	Y	sub	Reg[F2]	Reg[F6]			ROB4	
Add2	Y	add	Reg[F8]	Reg[F2]			ROB6	
Add3	no							
Mult1	Y	Mul	Reg[F2]	Reg[F4]			ROB3	
Mult2	Y	Div		Reg[F6]	ROB3		ROB5	

ROB status

Name	Busy	Instruction		State	Dest.	value
ROB1	no	fld	F6, 34(R2)	Commit	F6	xxx
ROB2	no	fld	F2, 45(R3)	Commit	F2	xxx
ROB3	yes	fmul.d	F0, F2, F4	Execute	F0	xxx
ROB4	yes	fsub.d	F8, F2, F6	Wrote result	F8	xxx
ROB5	yes	fdiv.d	F10, F0, F6	Issued/executing	F10	
ROB6	yes	fadd.d	F6, F8, F2	Wrote result	F6	xxx

Register status

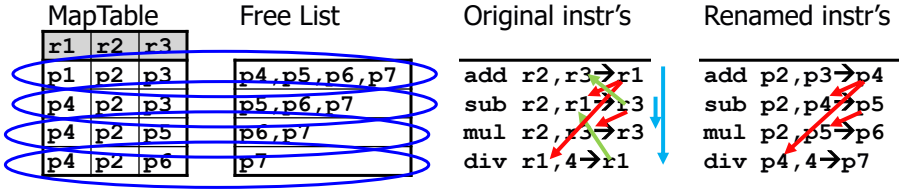
	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	ROB3			ROB6	ROB4	ROB5			

Note: reservation stations Add1 and Add2 may be released after the instructions finish the write back.

(139)

Register renaming

- To eliminate (WAW, WAR) register conflicts/hazards
- "Architected" vs "Physical" registers – level of indirection
  - Architected (ISA) register names: `r1,r2,r3`
  - Physical register locations: `p1,p2,p3,p4,p5,p6,p7`
  - Original mapping: `r1`→`p1`, `r2`→`p2`, `r3`→`p3`, `p4`–`p7` are "available"



- Renaming – conceptually write each register once
  - + Removes **false** dependences (WAW and WAR)
  - + Leaves **true** dependences (RAW) intact!
- When to *reuse* a physical register? After overwriting instr commits.

(140)

Register Renaming

- Common variation of speculative design
- Reorder buffer keeps instruction information but not the result
- Extend register file with extra *renaming registers* to hold speculative results
- Rename register allocated at issue;
- Renaming registers are physical registers.
- Operands read either from register file (real or speculative) or via Common Data Bus
- Advantage: operands are always from single source (extended register file)

Name	Busy	Instruction		State	Dest.	R Reg.
ROB1	no	fld	F6, 34(R2)	Commit	F6	1
ROB2	no	fld	F2, 45(R3)	Commit	F2	13
ROB3	yes	fmul.d	F0, F2, F4	Write result	F0	15
ROB4	yes	fsub.d	F8, F2, F6	Write result	F8	2
ROB5	yes	fdiv.d	F10, F0, F6	Issued/executing	F10	
ROB6	yes	fadd.d	F6, F8, F2	Write result	F6	14

ROB status

Renaming Registers

(141)

Multiple issue processors (§3.7)

- Issue more than one instruction per clock cycle
  - VLIW processors (static scheduling by the compiler)
  - Superscalar processors
    - » Statically scheduled (in-order execution)
    - » Dynamically scheduled (out-of-order execution)
- results in CPI < 1 (Instructions per clock, IPC > 1)
- The fetch unit gets an *issue packet* which contains multiple instructions
- The issue unit issues 1-8 instructions every cycle (usually, the issue unit is itself pipelined)
  - independent instructions
  - multiple resources should be available
  - branch prediction should be accurate
- Leads to multiple instruction completion per cycle

(142)

Dynamic scheduling (§3.8)

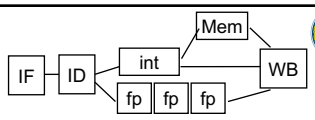
- Extends Tomasulo's algorithm to issue two (or more) instructions simultaneously to reservation stations.
- Either issue an instruction every half clock cycle, or double the logic to handle two instructions at once.
- Use the same logic. Some restrictions may be used to simplify hardware.
  - Example: issue only one FP and one int. operation every clock cycle. This reduces the load on the register files.
  - Do not issue dependent instructions in the same cycle
- To deal with control hazards without speculation (no ROBs): instructions following a branch can be issued but cannot start execution before the branch is resolved.
- Will look at the execution of

```
L1:  fld     F0, 0(R1)
     fadd.d  F4, F0, F2
     fsd     F4, 0(R1)
     addi    R1, R1, -8
     bne     R1, R2, L1
```

(149)

149

Dual issue with one int. and one FP add unit (not pipelined). Latency = 1, 2 and 3 for int. operations, loads and FP adds (one CDB).



Iteration	Instruction		Issued at	Executes	Mem access	Write CDB	comments
1	fld	F0, 0(R1)	1	2	3	4	First issue
1	fadd.d	F4, F0, F2	1	5		8	Wait for fld
1	fsd	F4, 0(R1)	2	3	9		Wait for fadd.d
1	addi	R1, R1, -8	2	4		5	Wait for ALU
1	bne	R1, R2, L1	3	6			wait for addi
2	fld	F0, 0(R1)	4	7	8	9	Wait for bne
2	fadd.d	F4, F0, F2	4	10		13	Wait for fld
2	fsd	F4, 0(R1)	5	8	14		Wait for fadd.d
2	addi	R1, R1, -8	5	9		10	Wait for ALU
2	bne	R1, R2, L1	6	11			wait for addi
3	fld	F0, 0(R1)	7	12	13	14	Wait for bne
3	fadd.d	F4, F0, F2	7	15		18	Wait for fld
3	fsd	F4, 0(R1)	8	13	19		Wait for fadd.d
3	addi	R1, R1, -8	8	14		15	Wait for ALU
3	bne	R1, R2, L1	9	16			wait for addi

- No speculation: Instruction after *bne* cannot be issued in same cycle since the branch is not yet predicted (orange arrows)
- *fld, fsd, addi* and *bne* use the same integer Execute unit (see blue circles)
- Assume as many reservation stations as needed

(150)

150