Reorder Buffer: register renaming and in-order completion

- Use of a reorder buffer
  - Reorder buffer = circular queue with head and tail pointers
- At issue (renaming time), an instruction is assigned an entry at the tail of the reorder buffer (ROB) which becomes the name of (or a pointer to) the result register.
  - Recall that instructions are issued in program order, thus the ROB stores instructions in program order
- At end of functional-unit computation, value is put in the instruction reorder buffer's position
- When the instruction reaches the head of the buffer, its value is stored in the logical or physical (other reorder buffer entry) register.
- Need of a mapping table between logical registers and ROB entries

Example

Before:		after	
	add r3,r3,4		add rob6,r3,4
	add r4,r7,r3		add rob7,r7,rob6
	add r3, r2, r7		add rob8,r2,r7

Assume reorder buffer is initially at position 6 and has more than 8 slots

The *mapping table* indicates the correspondence between ROB entries and logical registers

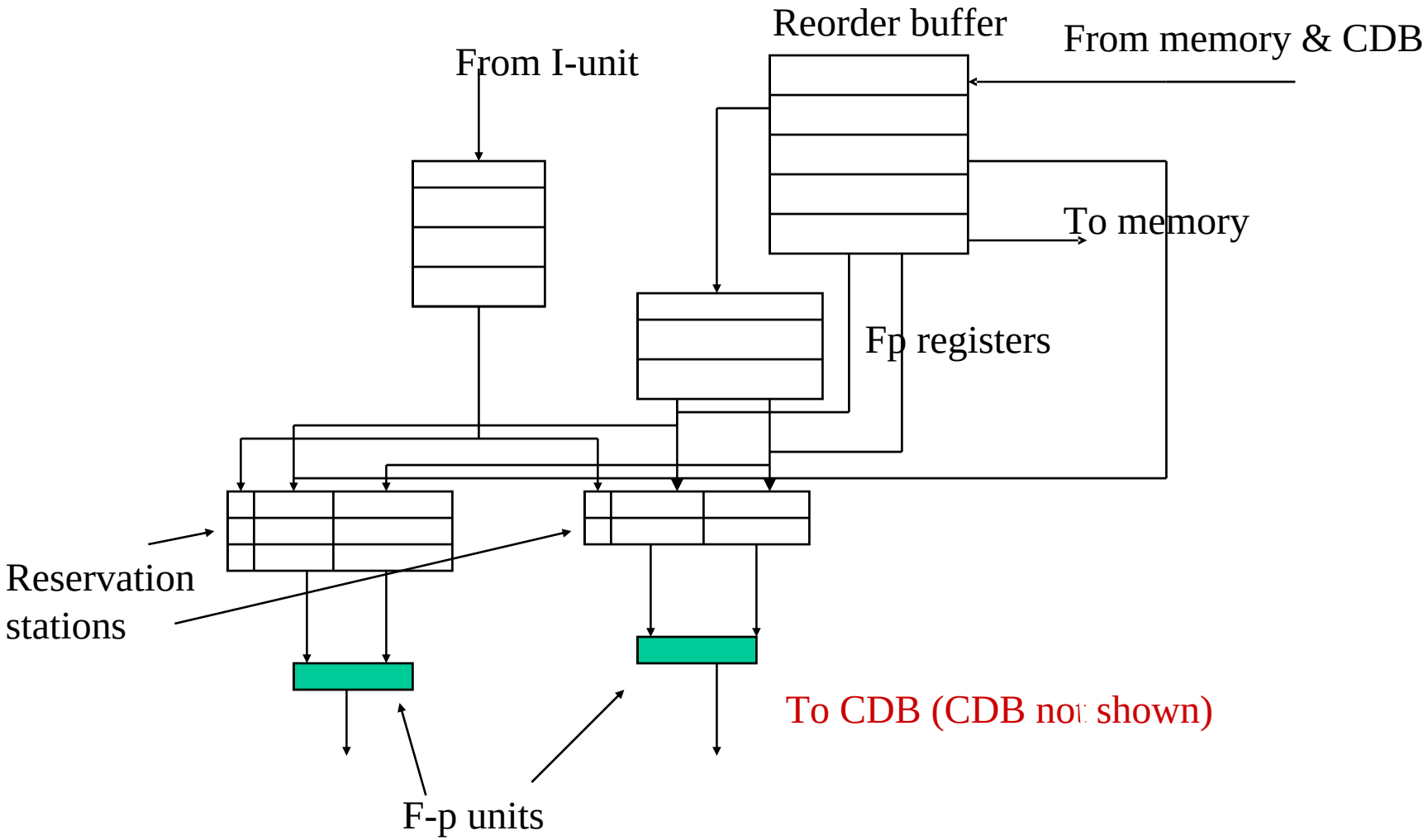
Data dependencies with register renaming

- Register renaming does not get rid of RAW dependencies
  - Still need for forwarding or for indicating whether a register has received its value
- Register renaming gets rid of WAW and WAR dependencies
- The reorder buffer, as its name implies, can be used for in-order completion

More on reorder buffer

- Tomasulo's scheme can be extended with the possibility of completing instructions in order
- Reorder buffer entry contains (this is not the only possible solution)
  - Type of instruction (branch, store, ALU, or load)
  - Destination (none, memory address, register including other ROB entry)
  - Value and its presence/absence
- Reservation station tags and "true" register tags are now ids of entries in the reorder buffer

Example machine revisited (Fig 2.14 (3.29))



Need for 4 stages

- In Tomasulo's solution 3 stages: issue, execute, write
- Now 4 stages: issue, execute, write, commit
- Dispatch and Issue
  - Check for structural hazards (reservation stations busy, reorder buffer full). If one exists, stall the instruction and those following
  - If dispatch possible, send source operand values to reservation station if the values are available in either the registers or the reorder buffer. Otherwise send tag.
  - Allocate an entry in the reorder buffer (rename result register) and send its number to the reservation station (to be used as a tag on CDB)
  - When both operands are ready, issue to functional unit

Need for 4 stages (c'ed)

- Execute
- Write
  - Broadcast on common data bus the value and the tag (reorder buffer number). Reservation stations, if any match the tag, and reorder buffer (always) grab the value.
- Commit
  - When instr. at head of the reorder buffer has its result in the buffer it stores it in the real register (for ALU) or memory (for store). The reorder buffer entry (and/or physical register) is freed.

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes		
3	Mul F0, F2, F4	yes			
4	Sub F8, F6, F2	yes			
5	Div F10, F0, F6	yes			
6	Add F6,F8,F2	yes			

Reservation Stations

Initial

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	yes	Sub	(#1)			(#2)
Add2	yes	Add			(#4)	(#2)
Add3	no					
Mul1	yes	Mul		(F4)	(#2)	
Mul2	yes	Div		(#1)	(#3)	

Register status

F0	F2	F4	F6	F8	F10	F12...
(#3)	(#2)	( )	(#6)	(#4)	(#5)	

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes	yes	yes
3	Mul F0, F2, F4	yes	yes		
4	Sub F8, F6, F2	yes	yes		
5	Div F10, F0, F6	yes			
6	Add F6,F8,F2	yes			

Cycle after 2nd load has committed

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	no					
Add2	yes	Add		(#2)	(#4)	
Add3	no					
Mul1	yes	Mul	(#2)	(F4)		
Mul2	yes	Div		(#1)	(#3)	

Register status

F0	F2	F4	F6	F8	F10	F12...
(#3)	( )	( )	(#6)	(#4)	(#5)	

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes	yes	yes
3	Mul F0, F2, F4	yes	yes		
4	Sub F8, F6, F2	yes	yes	yes	
5	Div F10, F0, F6	yes			
6	Add F6,F8,F2	yes	yes		

Cycle after sub has written its result in reorder buffer but can't commit yet

Reservation Stations

Name	Busy	Fm	Vj	Vk	Qj	Qk
Add 1	no					
Add2	yes	Add	(#2)	(#4)		
Add3	no					
Mul1	yes	Mul	(#2)	(F4)		
Mul2	yes	Div		(#1)	(#3)	

Still waiting for #3 to commit

Register status

F0	F2	F4	F6	F8	F10	F12...
(#3)	( )	( )	(#6)	(#4)	(#5)	

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes	yes	yes
3	Mul F0, F2, F4	yes	yes		
4	Sub F8, F6, F2	yes	yes	yes	
5	Div F10, F0, F6	yes			
6	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	(#2)	(F4)		
Mul2	yes	Div		(#1)	(#3)	

Cycle after add has written its result in reorder buffer but cannot commit

Still waiting for #3 to commit

Register status

F0	F2	F4	F6	F8	F10	F12...
(#3)	( )	( )	(#6)	(#4)	(#5)	

Still waiting for #3 to commit

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes	yes	yes
3	Mul F0, F2, F4	yes	yes	yes	yes
4	Sub F8, F6, F2	yes	yes	yes	
5	Div F10, F0, F6	yes	yes		
6	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	(#3)	(#1)		

Cycle after mul has written its result and committed

Still waiting: only 1 commit per cycle

Register status

F0 ()	F2( )	F4 ( )	F6(#6)	F8 (#4)	F10 (#5)	F12...

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes	yes	yes
3	Mul F0, F2, F4	yes	yes	yes	yes
4	Sub F8, F6, F2	yes	yes	yes	yes
5	Div F10, F0, F6	yes	yes		
6	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	(#3)	(#1)		

Now #4 can commit

Register status

F0 () F2( ) F4 ( ) F6(#6) F8 () F10 (#5) F12...

Reorder buffer

Entry #	Instruction	Issue	Execute	Write result	Commit
1	Load F6, 34(r2)	yes	yes	yes	yes
2	Load F2, 45(r3)	yes	yes	yes	yes
3	Mul F0, F2, F4	yes	yes	yes	yes
4	Sub F8, F6, F2	yes	yes	yes	yes
5	Div F10, F0, F6	yes	yes		
6	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		(#3)		(#1)

The next "interesting event is completion of div; then commit of #5, then commit of #6

Register status

F0 () F2( ) F4 ( ) F6(#6) F8 () F10 (#5) F12...

Still waiting for #4, #5 to commit

Register renaming – Physical Register file

- Use a *physical* register file (as an alternative to reservation station or reorder buffer) larger than the ISA *logical* one
- When instruction is decoded
  - Give a new name to result register from free list. The register is *renamed*
  - The mapping table is updated
  - Give source operands their physical names (from mapping table)

Register renaming –File of physical registers

- Extra set of registers organized as a free list
- At decode:
  - Rename the result register (get from free list; update mapping table). If none available, we have a structural hazard
  - Note that several physical registers can be mapped to the same logical register (corresponding to instructions at different times; avoids WAW hazards)
- When a physical register has been read for the last time, return it to the free list
  - Have a counter associated with each physical register (+ when a source logical register is renamed to physical register; - when instruction uses physical register as operand; release when counter is 0)
  - Simpler to wait till logical register has been assigned a new name by a later instruction and that later instruction has been *committed*

Example

Before: add r3,r3,4
add r4,r7,r3
add r3, r2, r7

after add r37,r3,4
add r38,r7,r37
add r39,r2,r7

Free list r37,r38,r39 ….
r2, r3, r4, r7 not renamed yet

At this point r3 is remapped from r37 to r39
When r39 commits, r37 will be returned to the free list

Conceptual execution on a processor which exploits ILP

- Instruction fetch and branch prediction
  - Corresponds to IF in simple pipeline
  - Complicated by multiple issue (see in a couple of slides)
- Instruction decode, dependence check, dispatch, issue
  - Corresponds (many variations) to ID
  - Although instructions are issued (i.e., assigned to functional units), they might not execute right away (cf. reservation stations)
  - It is at this point that one distinguishes between in-order and out-of-order superscalars
- Instruction execution
  - Corresponds to EX and/or MEM (with various latencies)
- Instruction commit (for OOO only)
  - Corresponds to WB but more complex because of speculation and out-of-order completion

Multiple Issue Alternatives

- Superscalar (hardware detects conflicts)
  - Statically scheduled (in order dispatch and hence execution; cf. (DEC)Alpha 21164, Sun processor in Niagara, IBM Cell Synergetic Processor)
  - Dynamically scheduled (in order issue, out of order dispatch and execution; cf. MIPS 10000, IBM Power 4 and 5, Intel Pentium P6 microarchitecture, AMD K5 et al, (DEC)Alpha 21264, Sun UltraSparc etc.)
- VLIW – EPIC (Explicitly Parallel Instruction Computing)
  - Compiler generates “bundles “ of instructions that can be executed concurrently (cf. Intel Itanium, lot of DSP’s)

Multiple Issue for Static/Dynamic Scheduling

- Issue in order
  - Otherwise bookkeeping is complex (the old "data flow" machines could issue any ready instruction in the whole program; see also new "grid" machines such as WaveScalar and Trip)
  - Check for structural hazards; if any stall
- Dispatch for static scheduling
  - Check for data dependencies; stall adequately
  - Can take forwarding into account
- Dispatch for dynamic scheduling
  - Dispatch out of order (reservation stations, instruction window)
  - Rename registers
  - Requires possibility of dispatching concurrently dependent instructions (otherwise little benefit over static scheduling)

Impact of Multiple Issue on IF

- IF: Need to fetch more than 1 instruction at a time
  - Simpler if instructions are of fixed length
  - In fact need to fetch as many instructions as the issue stage can handle in one cycle
  - Simpler if restricted not to overlap I-cache lines
  - But with branch prediction, this is not realistic hence introduction of (instruction) fetch buffers and trace caches
  - Always attempt to keep at least as many instructions in the fetch buffer as can be issued in the next cycle (BTB's help for that)
  - For example, have an 8 wide instruction buffer for a machine that can issue 4 instructions per cycle

Stalls at the IF Stage

- Instruction cache miss
- Instruction buffer is full
  - Most likely there are stalls in the stages downstream
- Branch misprediction
- Instructions are stored in several I-cache lines
  - In one cycle one I-cache line can be brought into fetch buffer
  - A basic block might start in the middle (or end) of an I-cache line
  - Requires several cache lines to fill the buffer
  - The ID (issue-dispatch) stage will stall if not enough instructions in the fetch buffer

Sample of Old and Current Micros

- Two instruction issue: Alpha 21064, Sparc 2, Pentium, Cyrix
- Three instruction issue: Pentium P6 (but 5 uops from IF/ID to EX; Pentium 4 and AMD K7 have 4 uops, Intel Core has 6 uops)
- Four instruction issue: Alpha 21164, Alpha 21264, IBM Power4 and Power5 (but somewhat restricted), Sun UltraSparc, HP PA-8000, MIPS R10000
- Many papers written in mid-90's predicted 16-way issue by 2000. We are still at 4 in 2007!

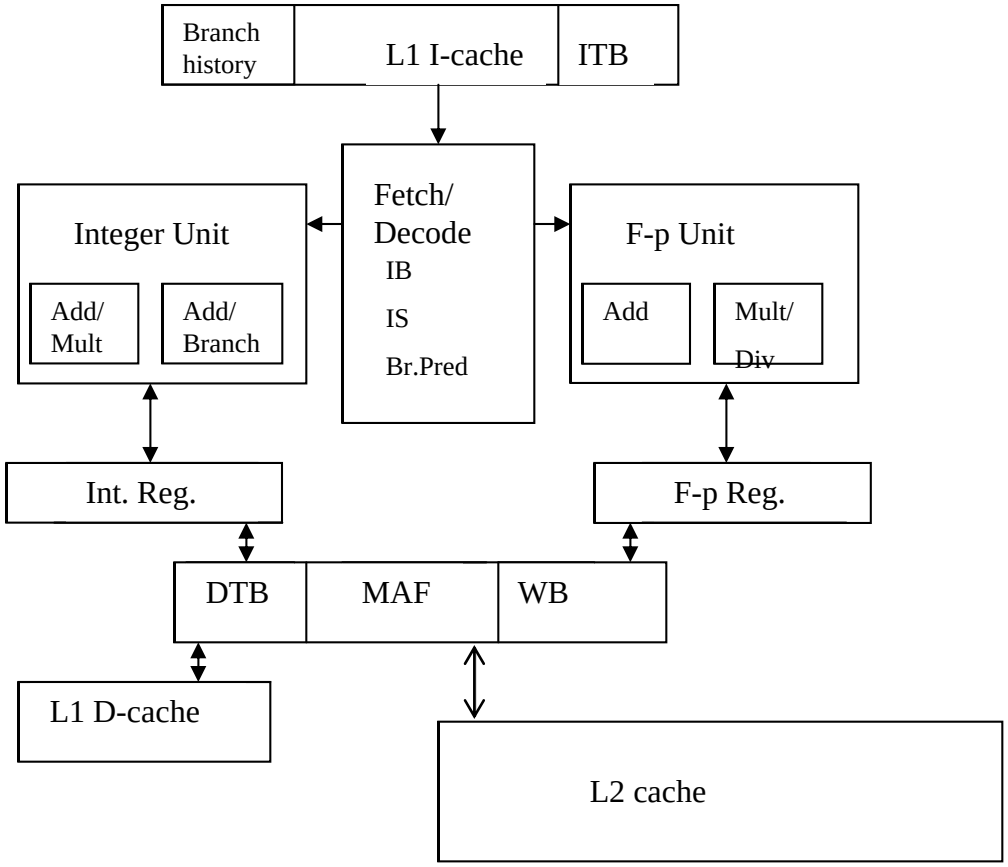
The Decode Stage (simple case: dual issue and static scheduling)

- ID = Dispatch + Issue
  - Some authors would call this “Issue + Dispatch”!
- Look for conflicts between the (say) 2 instructions
  - If one integer op. and one f-p op., only check for structural hazard, i.e. the two instructions need the same f-u (easy to check with opcodes)
  - RAW dependencies resolved as in single pipelines
  - Note that the load delay (assume 1 cycle) can now delay up to 3 instructions, i.e., 3 issue slots are lost

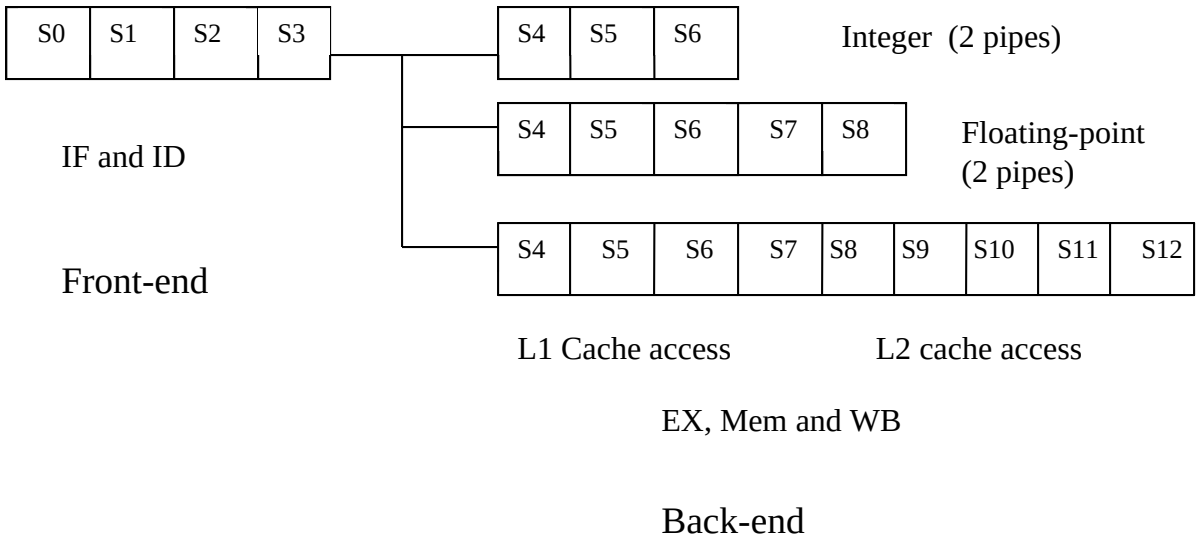
Decode in Simple Multiple Issue Case

- If instructions i and i+1 are fetched together and:
  - Instruction i stalls, instruction i+1 will stall
  - Instruction i is dispatched but instruction i+1 stalls (e.g., because of structural hazard = need the same f-u), instruction i+2 will not advance to the issue stage. It will have to wait till both i and i+1 have been dispatched

Alpha 21164 (@1995) 4-wide



Pipeline.



Front-end 4 stages; Back-end from 3 to 9

Alpha 21164 – Front-end

- IF S0: Access I-cache
  - Prefetcher fetches 4 instructions (16 bytes) at a time in one of two instruction buffers (IB). Each instruction has been predecoded (5 bits)
- IF-S1 : Branch Prediction
  - Prefetcher contains branch prediction logic tested at this stage: 4 entry return stack; 2 bit/instruction in the I-cache + static prediction BTFNT
- ID-S2: Slotting
  - Initial decode yields 0, 1, 2, 3 or 4 instruction potential issue; align instructions depending on the functional unit there are headed for.
- ID-S3.
  - Check for issue: WAW and WAR (my guess) so that all instructions after S3 can execute successfully w/o stalls

Alpha 21164 Restrictions in front-end

- In integer programs, only 2 arithmetic instructions can pass from S2 to S3 (structural hazards)
  - This percolates back ….
- In S0, only instructions in the same cache line can be fetched in a given cycle
  - Too bad if you branch in the middle of a cache line…
- Target branch address computed in S1
  - So if predict taken, you have one “bubble”. Good chance it will be amortized by other effects downstream
- S3 uses the equivalent of a (simplified) scoreboard

Alpha 21164 - Back-end

- Load latency : 2 cycles
  - If instruction *i* is a load issued (leave S3) at time *t* and inst. *i+1* depends on it: real bubble since inst *i+1* will leave S3 at time *t+2*
  - (If instead of inst *i+1* it were inst i+2 that were dependent, could we still have a real bubble?)
- Scoreboard does not know if cache hit or miss
  - Speculates hit (why?) If wrong, known at S5, instructions already in the back-end not dependent on the load can proceed (scoreboard knows that). Others are aborted
- On branch mispredict (and precise) exceptions
  - Known at S5. All inst. in program order after the branch are aborted
  - (how can we enforce precise exceptions on the integer and memory pipelines?)
- Other possible structural hazards due to store buffers etc. (see later)
- What happens on a D-TLB miss?

Dynamic Scheduling: Reservation stations, register renaming and reorder buffer

- Decode means:
  - Dispatch to either
    - A centralized instruction window common to all functional units (Pentium Pro, Pentium III and Pentium 4)
    - Reservation stations associated with functional units (MIPS 10000, AMD K5-7, IBM Power4 and Power5)
  - Rename registers (either via ROB or physical file)
    - Note the difficulty when renaming in the same cycle

      R1 <- R2 + R3; R4 <- R1 + R5
  - Set up entry at tail of reorder buffer (if supported by architecture)
  - Issue operands, when ready, to functional unit

Stalls in Decode (issue/dispatch) Stage

- If there are decentralized reservation stations, there can be several instructions ready to be dispatched in same cycle to same functional unit
  - Possibility of not enough reservation stations
- If there is a centralized instruction window, there might not be enough bus/ports to forward values to the execution units that need them in the same cycle
- Both instances are instances of structural hazards
  - Conflicts are resolved via a *scheduling* algorithm
  - Try and define *critical* instructions

The Execute Stage

- Use of forwarding
  - Use of broadcast bus or cross-bar or other interconnection network
- We'll talk at length about memory operations (load-store) in subsequent lecture and when we study memory hierarchies

The Commit Step (in-order completion)

- Recall: need of a mechanism (reorder buffer) to:
  - "Complete" instructions in order. This commits the instruction. Since multiple issue machine, should be able to commit (retire) several instructions per cycle
  - Know when an instruction has completed non-speculatively,i.e., what to do with branches
  - Know whether the result of an instruction is correct, i.e., what to do with exceptions

Impact on Branch Prediction and Completion

- When a conditional branch is decoded:
  - Save the current physical-logical mapping
  - Predict and proceed
- When branch is ready to commit (head of buffer)
  - If prediction correct, discard the saved mapping
  - If prediction incorrect
    - Flush all instructions following mispredicted branch in reorder buffer
    - Restore the mapping as it was before the branch as per the saved map
- Note that there have been proposals to execute both sides of a branch using register shadows
  - limited to one extra set of registers

Exceptions

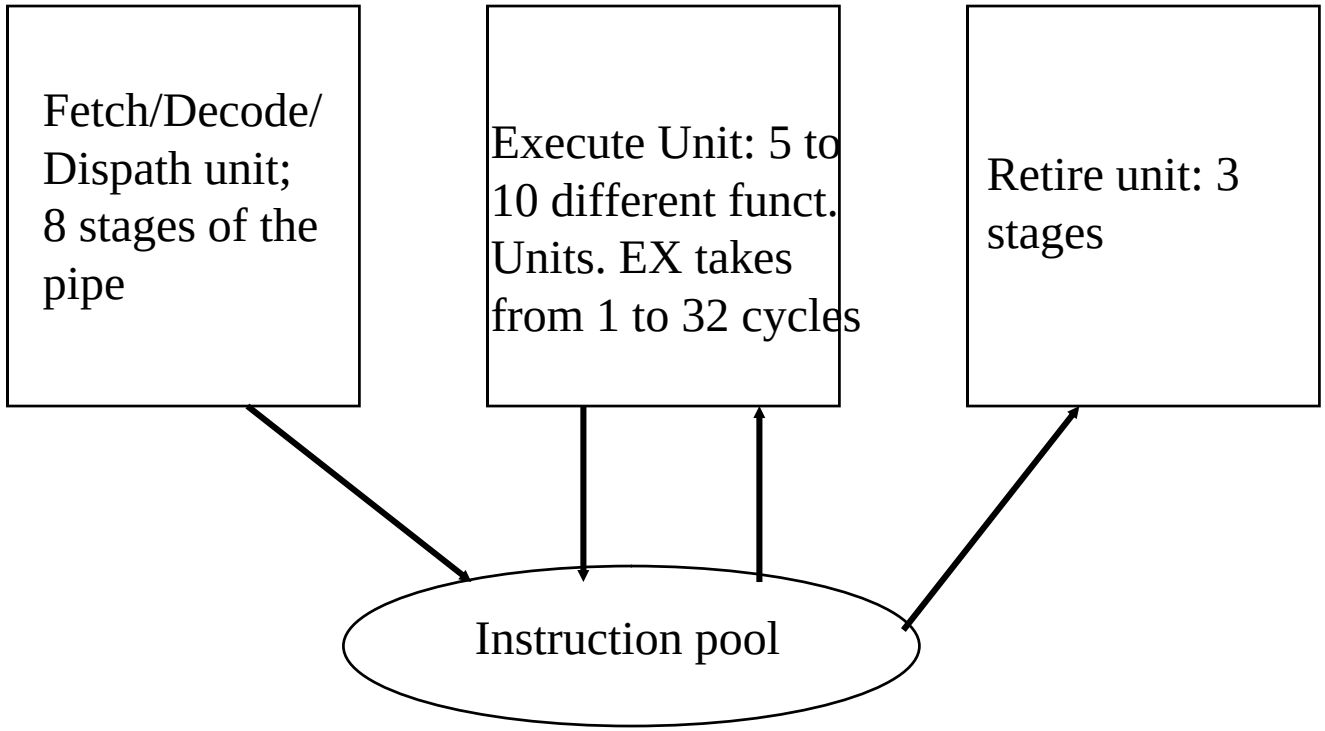
- Instructions carry their exception status
- When instruction is ready to commit
  - No exception: proceed normally
  - Exception
    - Flush (as in mispredicted branch)
    - Restore mapping (more difficult than with branches because the mapping is not saved at every instruction; this method can also be used for branches)

Summary: OOO flow of instructions

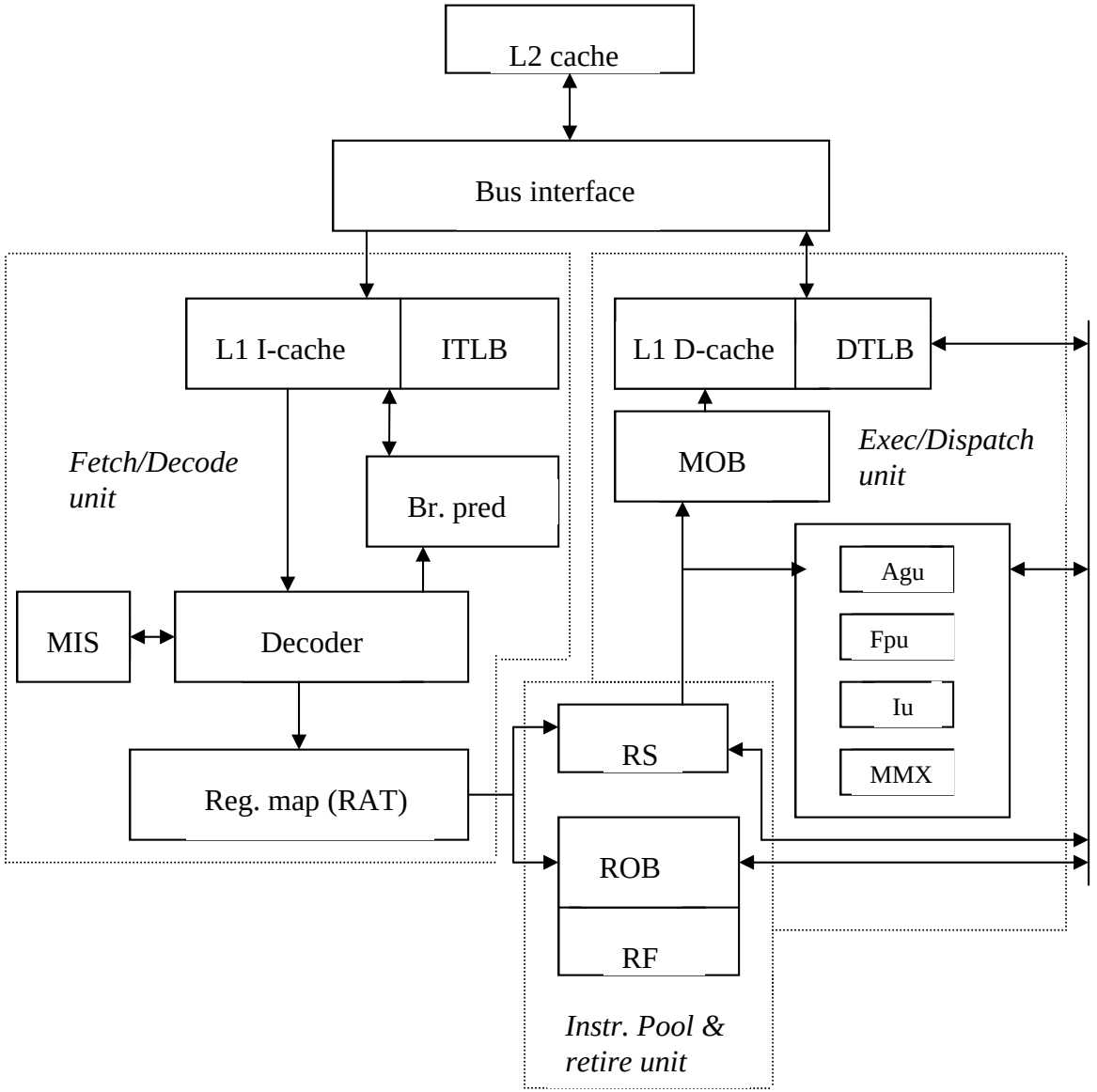
	Step	Resources read	Resources written or utilized
Front-end	Fetch	PC Branch Predictor I-cache	PC Instruction Buffer
	Decode-rename	Instruction Buffer Register map	Decode Buffer Register map ROB
	Dispatch	Decode Buffer Register map Register file (logical and physical)	Reservation stations ROB
Back-end	Issue	Reservation stations	Functional units D-cache
	Execute	Functional Units D-cache	Reservation stations ROB Physical register file Branch Predictor Store Buffer etc…
	Commit	ROB Physical register file Store buffer	ROB Logical register file Register map D-cache

Pentium Family (slightly more details in H&P Sec 2.10 (3.10 in 3rd))

- Fetch-Decode unit
  - Transforms up to 3 instructions at a time into micro-operations (uops) and stores them in a global reservation table (instruction window). Does register renaming (RAT = register alias table)
- Dispatch (aka issue)-execution unit
  - Issues uops to functional units that execute them and temporarily store the results
    - Depending on the implementation from 3 to 6 uops can be issued concurrently
- Retire unit
  - Commits the instructions in order (up to 3 commits/cycle)



The 3 units of the Pentium P6 are “independent” and communicate through the instruction pool


L2 cache
Bus interface
L1 I-cache
ITLB
L1 D-cache
DTLB
Fetch/Decode unit
Br. pred
MOB
Exec/Dispatch unit
Agu
Fpu
Iu
MMX
MIS
Decoder
RS
Reg. map (RAT)
ROB
RF
Instr. Pool & retire unit

A Few More Details: Front-end

- Instruction Fetch (not in Pentium 4)
  - 4 (mini) stages for IF
    1. Access BTB-BPB combination (what if a miss?). If hit and predicted taken, a bubble is generated
    2. Initiates I-cache access at address given by BTB (what if a miss?)
    3. Continues I-cache access
    4. Completes I-cache access and transfer 16 bytes in Decode buffer
- Instruction Decode
  - 3 (mini) stages

    1 and 2. Find end of first 3 instructions and break then down in µops
    - only one branch decoded
    - Some CISC instructions require the “leftmost” decoder (MIS)

    3. Detect branches; can correct some situations (undetected unconditional branch for example)

Front-end (ctd)

- Register renaming
- Enter μops in reservation stations and ROB

Back-end

- µops can get executed when
  - Operands are available
  - The Execution Unit for that µop is available
  - A result bus will be available at completion
  - No more “important” µop should be executed
  - So it takes two cycle (pipe stages) to do all that. Then:
- µops are executed
  - We’ll see about load-store later
- Commit (aka retire)
  - All µops from the same instruction should be retired together (done by marking beg. And end of instructions when put in the ROB)

Limits to Hardware-based ILP

- Inherent lack of parallelism in programs
  - Partial remedy: loop unrolling and other compiler optimizations
  - Branch prediction to allow earlier issue and dispatch
- Complexity in hardware
  - Needs large bandwidth for instruction fetch (might need to fetch from more than one I-cache line in one cycle)
  - Requires large register bandwidth (multiported register files)
  - Forwarding/broadcast requires “long wires” (long wires are slow) as soon as there are many units.

Limits to Hardware-based ILP (c'ed)

- Difficulties specific to the implementation
  - More possibilities of structural hazards (need to encode some priorities in case of conflict in resource allocations)
  - Parallel search in reservation stations, reorder buffer etc.
  - Additional state savings for branches (mappings), more complex updating of BPT's and BTB's.
  - Keeping precise exceptions is more complex