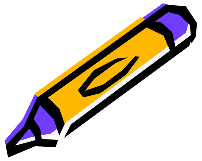
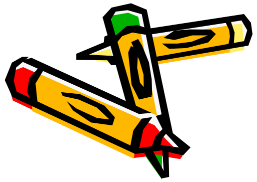
5008: Computer Architecture

Chapter 2 - Instruction-Level Parallelism and Its Exploitation



Review from Last Lecture

- Instruction Level Parallelism
  - Leverage implicit parallelism for performance:
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
  - Works when can't know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another



Review from Last Lecture

- Reservations stations: *renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation

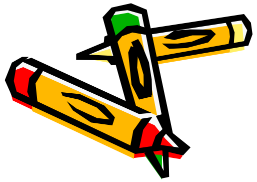
Outline

- Review
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Advanced Techniques for Instruction Delivery and Speculation
- Summary

Greater ILP ?

- Essentially a data flow execution model: Operations execute as soon as their operands are available

- Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  - Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
  - Dynamic scheduling ⇒ only fetches and issues instructions



Speculation to greater ILP

3 components of HW-based speculation:

1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow execution of instructions before control dependences are resolved

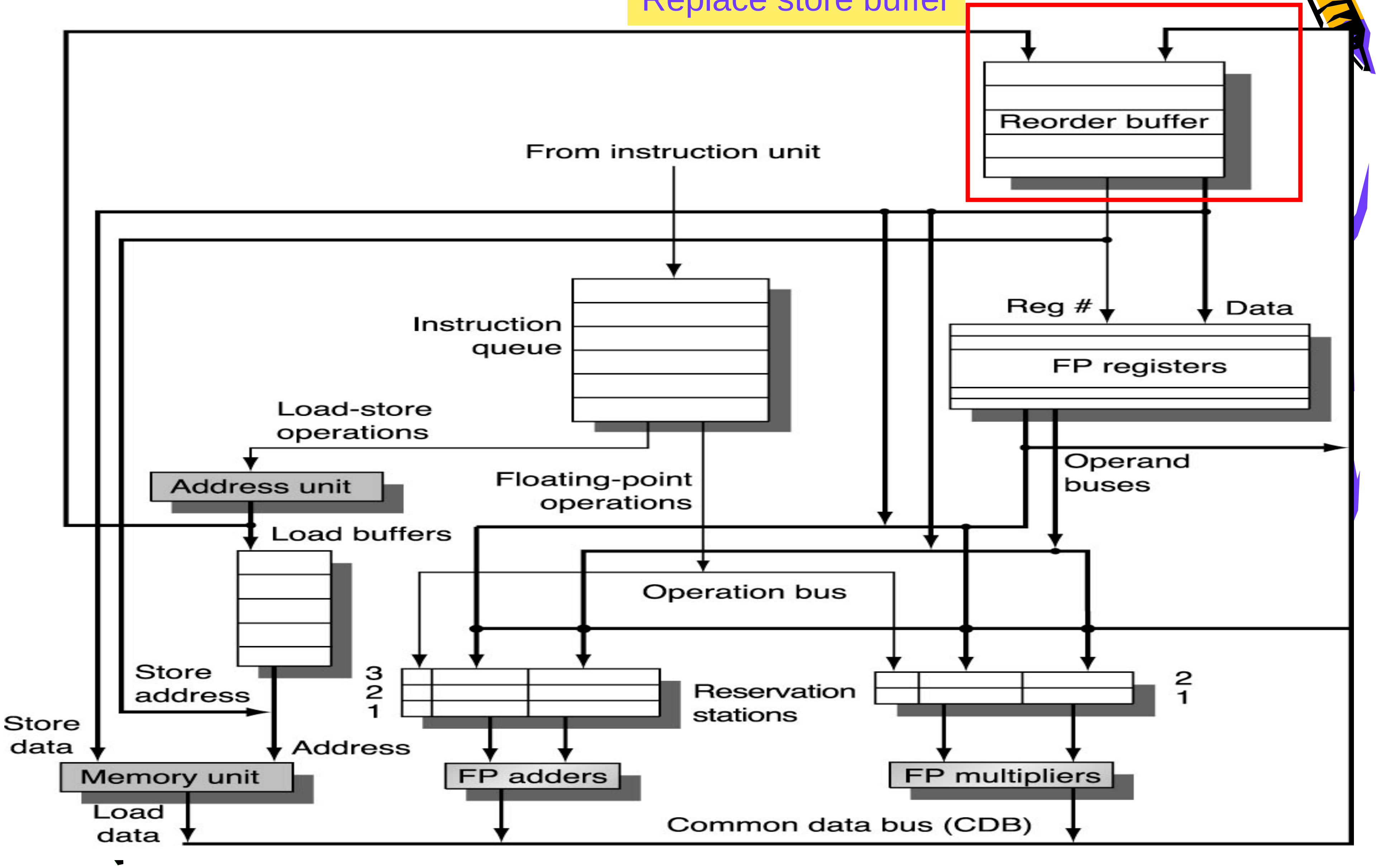
   + ability to undo effects of incorrectly speculated sequence
3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called instruction commit
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This reorder buffer (ROB) is also used to pass results among instructions that may be speculated

The Speculative MIPS

Replace store buffer

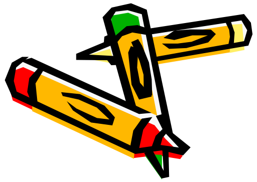


Observations

- For an execution result, separate
  - data forwarding (thru RS) path
  - write-back (thru ROB) path
- Data forwarding path
  - still use RS to buffer operands
  - provide speculative register reads
  - provide out-of-order completion
- Register write-back path
  - use ROB to buffer results
  - when it's committed, update RF (in order)

Reorder Buffer (ROB)

- Additional registers, just like reservation stations
  - ROB is a source of operands
  - It holds the results of instruction that have finished execution but not committed
  - Use ROB number instead of RS to indicate the source of operands when execution completes (but not committed)
  - It also uses to pass results among instructions that may be speculated
  - Each (pending) instruction occupies an ROB entry before being committed
  - Instructions in ROB are committed in order
    - Once instruction commits, the result is put into register
  - In case of misprediction, the corresponding ROB entry will be flushed



Reorder Buffer Entry

Each entry in the ROB contains four fields:

1. Instruction type
   - a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
2. Destination
   - Register number (for loads and ALU operations) or memory address (for stores)
     where the instruction result should be written
3. Value
   - Value of instruction result until the instruction commits
4. Ready
   - Indicates that instruction has completed execution, and the value is ready

4 Steps in Speculative Execution

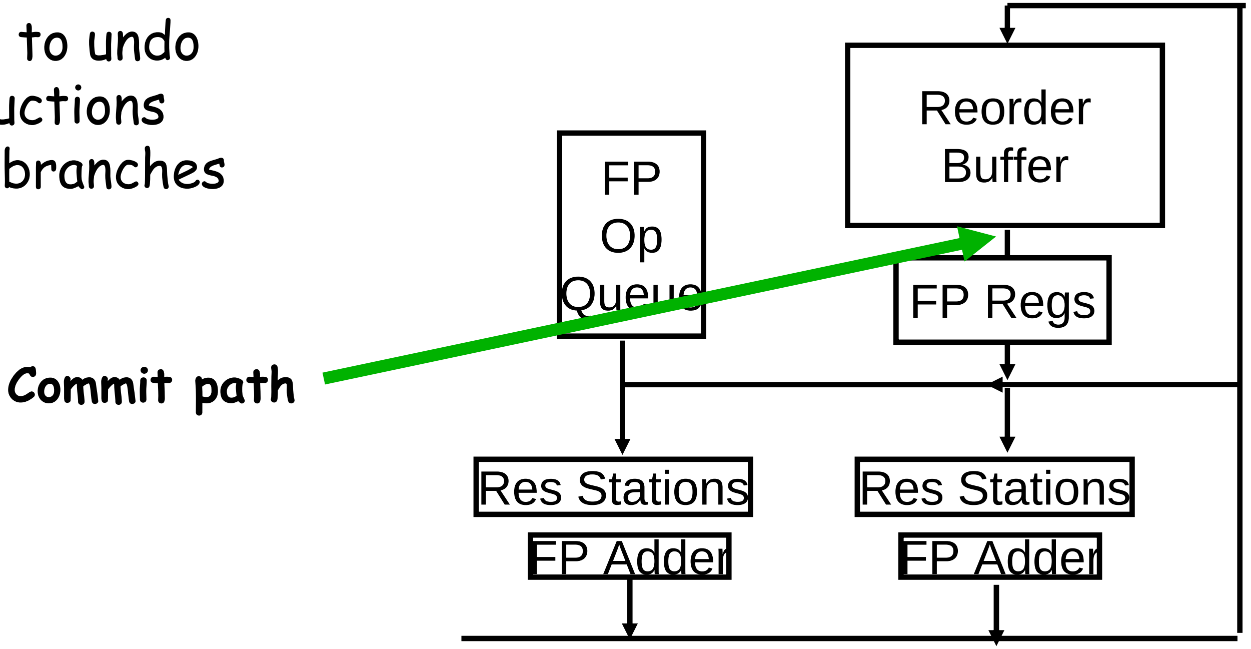
1. Issue (or dispatch)
   - Get instruction from the instruction queue
   - In-order issue if available RS AND ROB slot; otherwise, stall
   - Send operands to RS if they are in register or ROB
   - Update the control entries to indicate the buffers are in use
   - The ROB no. allocated for the result is sent to RS, so that the number can be used to tag the result when it is placed on CDB
2. Execute (or issue)
   - If not all operands are ready, monitor CDB and wait for it to be computed (check for RAW hazards)
   - When all operands are there, execution happens
3. Write Result
   - Result posted to ROB via the CDB
   - Waiting reservation stations can grab it as well

4 Steps in Speculative Execution

4. Commit (or graduate) – instruction reaches the ROB head
   - Only the head of ROB is allowed committing
   - Normal commit – when instruction reaches the ROB head and its result is present in the buffer
     - Update the register and remove the instruction from ROB
   - Store – Update memory and remove the instruction from ROB
   - Branch with incorrect prediction – wrong speculation
     - Flush ROB and the related FP OP queue (RS)
     - Restart at the correct successor of the branch
     - Remove the instruction from ROB
   - Branch with correct prediction – finish the branch
     - Remove the instruction from ROB

Reorder Buffer Operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

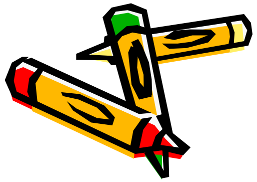


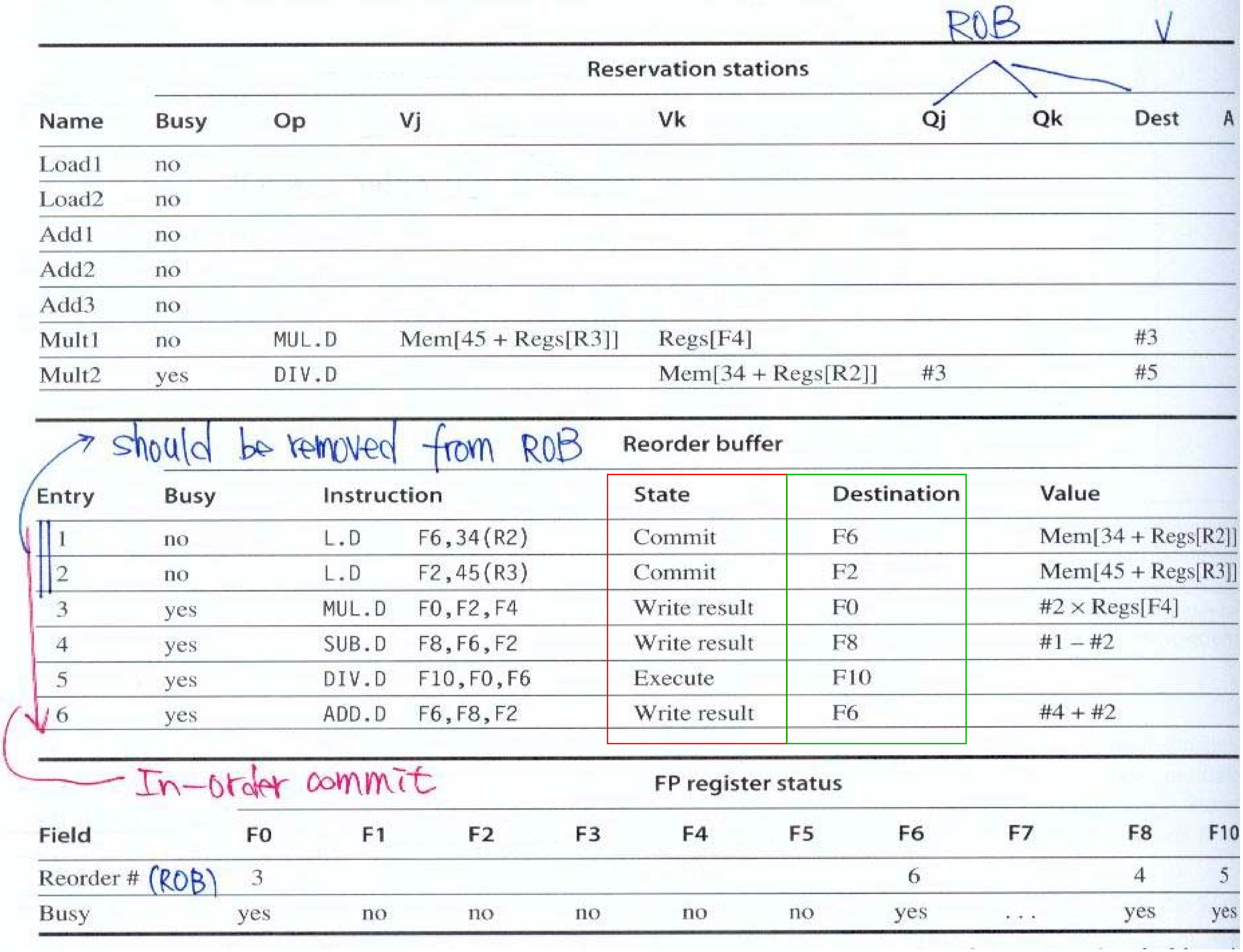
Example

- The same example as Tomasulo without speculation.
  - L.D F6, 34(R2)
  - L.D F2, 45(R3)
  - MUL.D F0, F2, F4
  - SUB.D F8, F6, F2
  - DIV.D F10, F0, F6
  - ADD.D F6, F8, F2

Assume
FP ADD: 2 cycles
MUL: 10 cycles
DIV: 40 cycles

- Modified status tables
  - Qj and Qk fields, and register status fields use ROB (instead of RS)
  - Add Dest field to RS (ROB to put the operation result)
- Show the status tables when MUL.D is ready to go to commit
  - At this time, only two L.D instructions have been committed





ROB V

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34 + Regs[R2]]	#3		#5	

should be removed from ROB

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	no	L.D	F6,34(R2)	Commit	F6	Mem[34 + Regs[R2]]
2	no	L.D	F2,45(R3)	Commit	F2	Mem[45 + Regs[R3]]
3	yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D	F8,F6,F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10,F0,F6	Execute	F10	
6	yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

In-order commit

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder # (ROB)	3						6		4	5
Busy	yes	no	no	no	no	no	yes	. . .	yes	yes

Comparisons

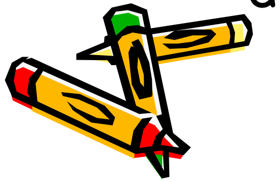
- Consider the case if MUL.D causes an interrupt…
- Tomasulo without speculation
  - SUB.D and ADD.D have completed (clock cycle 16, slide 04-108)
- Tomasulo with speculation
  - No instruction after the earliest uncompleted instruction (MUL.D) is allowed to complete
  - In-order commit
- Implication – ROB with in-order instruction commit provides precise exceptions
  - Precise exceptions – exceptions are handled in the instruction order

Tomasulo's + ROB

- Performance is more sensitive to branch-prediction
  - Impact of a mis-prediction will be higher
- Precise exception
  - Handled by not recognizing the exception until it is ready to commit
  - If a speculation instruction raises an exception, the exception is recorded in ROB
    - Mis-prediction branch → exception are flushed as well
    - If the instruction reaches the ROB head → take the exception
- Tomasulo's + ROB provides precise exceptions !!!

Remark: Avoiding Memory Hazards

- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
- RAW hazards through memory are maintained by two restrictions:
  1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
  2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
- these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

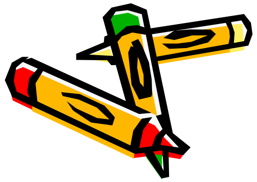


Remark: Exceptions and Interrupts

- IBM 360/91 invented "imprecise interrupts"
  - Computer stopped at this PC; its likely close to this address
  - Not so popular with programmers
  - Also, what about Virtual Memory? (Not in IBM 360)
- Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as need to do with precise exceptions
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
  - If a speculated instruction raises an exception, the exception is recorded in the ROB
  - This is why reorder buffers in all new processors

Outline

- Review
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Advanced Techniques for Instruction Delivery and Speculation
- Summary



Getting CPI below 1

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  1. statically-scheduled superscalar processors,
  2. dynamically-scheduled superscalar processors, and
  3. VLIW (very long instruction word) processors
- 2 types of superscalar processors issue varying numbers of instructions per clock
  - use in-order execution if they are statically scheduled, or
  - out-of-order execution if they are dynamically scheduled
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)

Multiple Issue with Speculation

- To maintain throughput of greater than one instructions per cycle, we must handle multiple instruction commits per clock
- Extend Tomasulo speculation algorithm to multiple-issue scheme
  - 2 challenges
    - Instruction issue
    - Monitor CDB for instruction completion
  - In addition,
    - How to handle multiple instruction commits per clock cycle?

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Basic VLIW

- A VLIW uses multiple, independent functional units
- A VLIW packages multiple independent operations into one very long instruction
  - The burden for choosing and packaging independent operations falls on compiler
  - HW in a superscalar makes these issue decisions is unnecessary
- VLIW depends on enough parallelism for keeping FUs busy
  - Loop unrolling and then code scheduling
  - Compiler may need to do local scheduling and global scheduling
- Here we consider a VLIW processor might have instructions that contain 5 operations, including 1 integer (or branch), 2 FP, and 2 memory references
  - Depend on the available FUs and frequency of operation

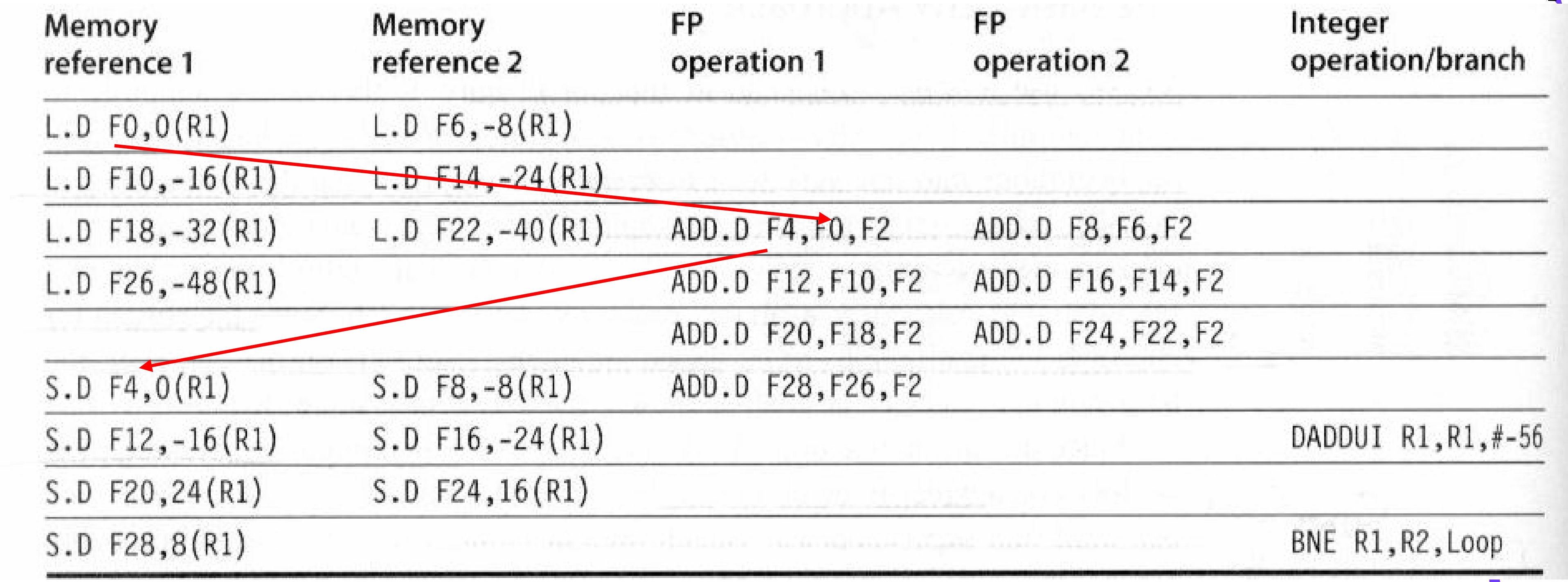
Recall: Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop: L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    0(R1),F4
10      S.D    -8(R1),F8
11      S.D    -16(R1),F12
12      DSUBUI R1,R1,#32
13      BNEZ   R1,LOOP
14      S.D    8(R1),F16    ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW



Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.29 clocks per iteration

23 ops in 9 clock, average 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW

VLIW Problems – Technical

- Increase in code size
  - Ambitious loop unrolling
  - Whenever instructions are not full, the unused FUs translate to waste bits in the instruction encoding
    - An instruction may need to be left completely empty if no operation can be scheduled
    - Clever encoding or compress/decompress

VLIW Problems – Logistical

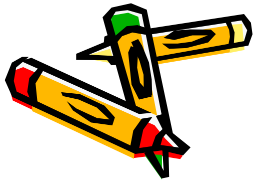
- Operated in lock-step; no hazard detection HW
  - Early VLIW – all FUs must be kept synchronized
    - A stall in any FU pipeline may cause the entire processor to stall
    - Compiler might prediction function units, but caches hard to predict
  - Recent VLIW – FUs operate more independently
    - Compiler is used to avoid hazards at issue time
    - Hardware checks allow for unsynchronized execution once instructions are issued.
    - Hardware complexity

VLIW Problems – Logistical

- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code
  - Need migration between successive implementations, or between implementations ➔ recompliation
  - Solution ➔Object-code translation or emulation

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA-64: instruction set architecture
- 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies
  (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags)
  => 40% fewer mispredictions?
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3



Advantages of Superscalar over VLIW

- Old codes still run
  - Like those tools you have that came as binaries
  - HW detects whether the instruction pair is a legal dual issue pair
    - If not they are run sequentially
- Little impact on code density
  - Don't need to fill all of the can't issue here slots with NOP's
- Compiler issues are very similar
  - Still need to do instruction scheduling anyway
  - Dynamic issue hardware is there so the compiler does not have to be too conservative

Multiple Issue with Speculation

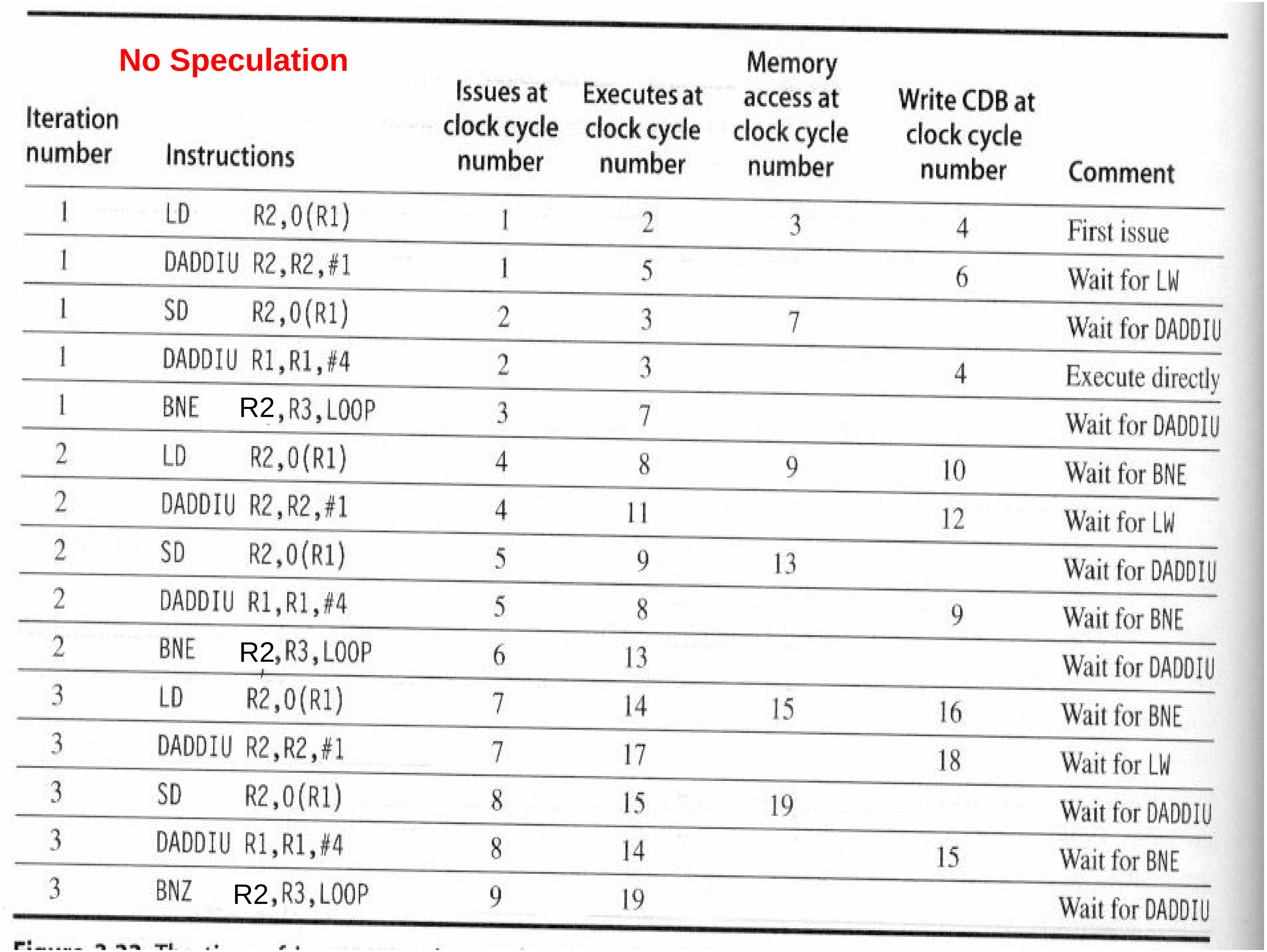
- Extend Tomasulo speculation algorithm to multiple-issue scheme
  - 2 challenges
    - Instruction issue
    - Monitor CDB for instruction completion
  - In addition,
    - How to handle multiple instruction commits per clock cycle?

Example

-
```
Loop:   LD       R2, 0(R1)
        DADDIU   R2, R2, #1
        SD       R2, 0(R1)
        DADDIU   R1, R1, #4
        BNE      R2, R3, LOOP
```

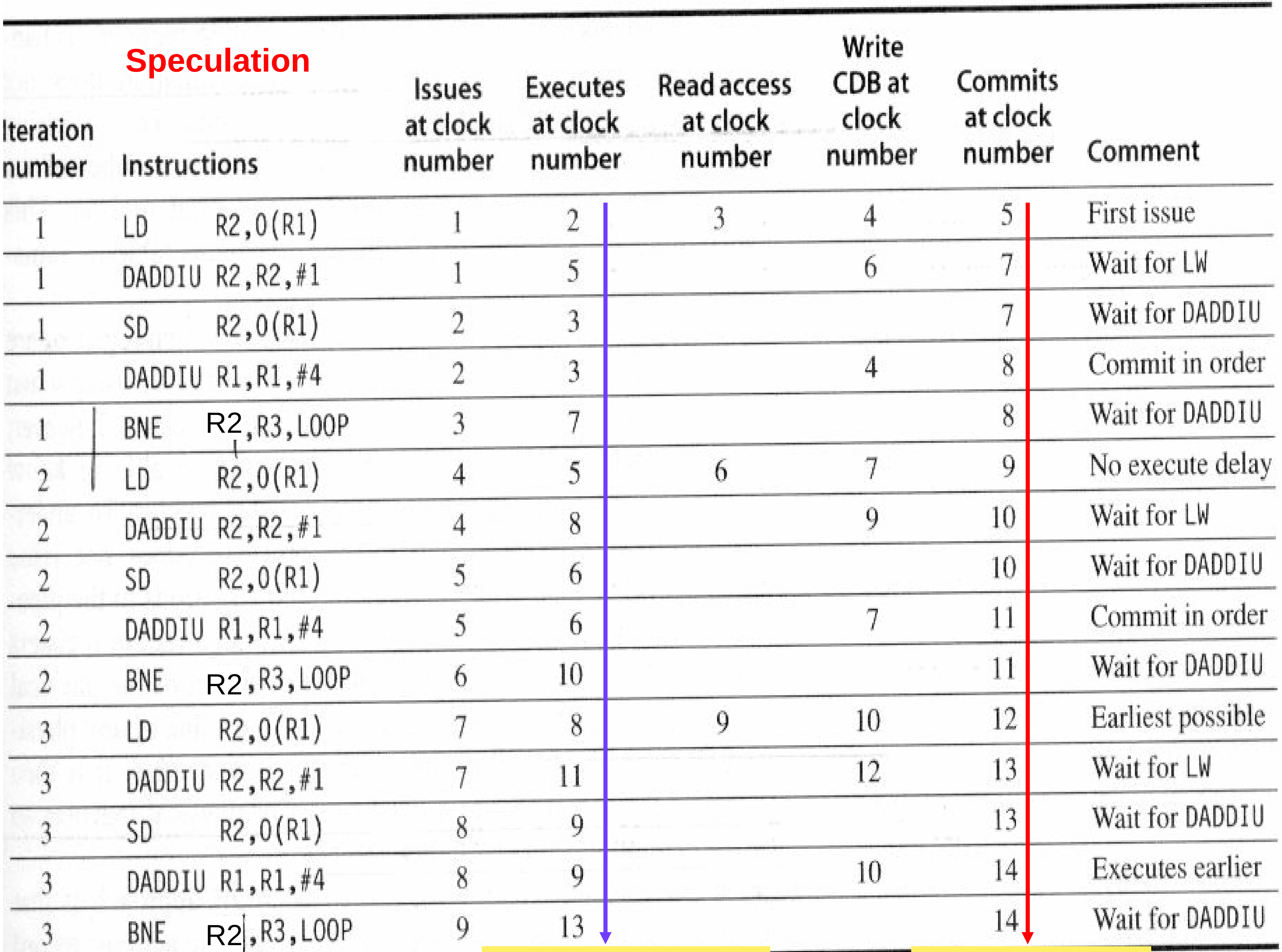
- Assume separate integer FUs:
  - for effective address calculation,
  - ALU operations, and
  - branch condition evaluation
- Assume up to 2 instructions of any type can commit per clock

No Speculation



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Speculation



Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Out-of-order executing

In-order committing

Comparisons

- Without speculation (Tomasulo only)
  - L.D following BNE cannot start execution earlier → wait until branch outcome is determined
  - Completion rate is falling behind the issue rate rapidly, stall when a few more iterations are issued
- With speculation
  - L.D following BNE can start execution early because it is speculative
  - More complex HW is required
  - Completion rate is almost equal to issue rate

Outline

- Review
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Advanced Techniques for Instruction Delivery and Speculation
- Summary

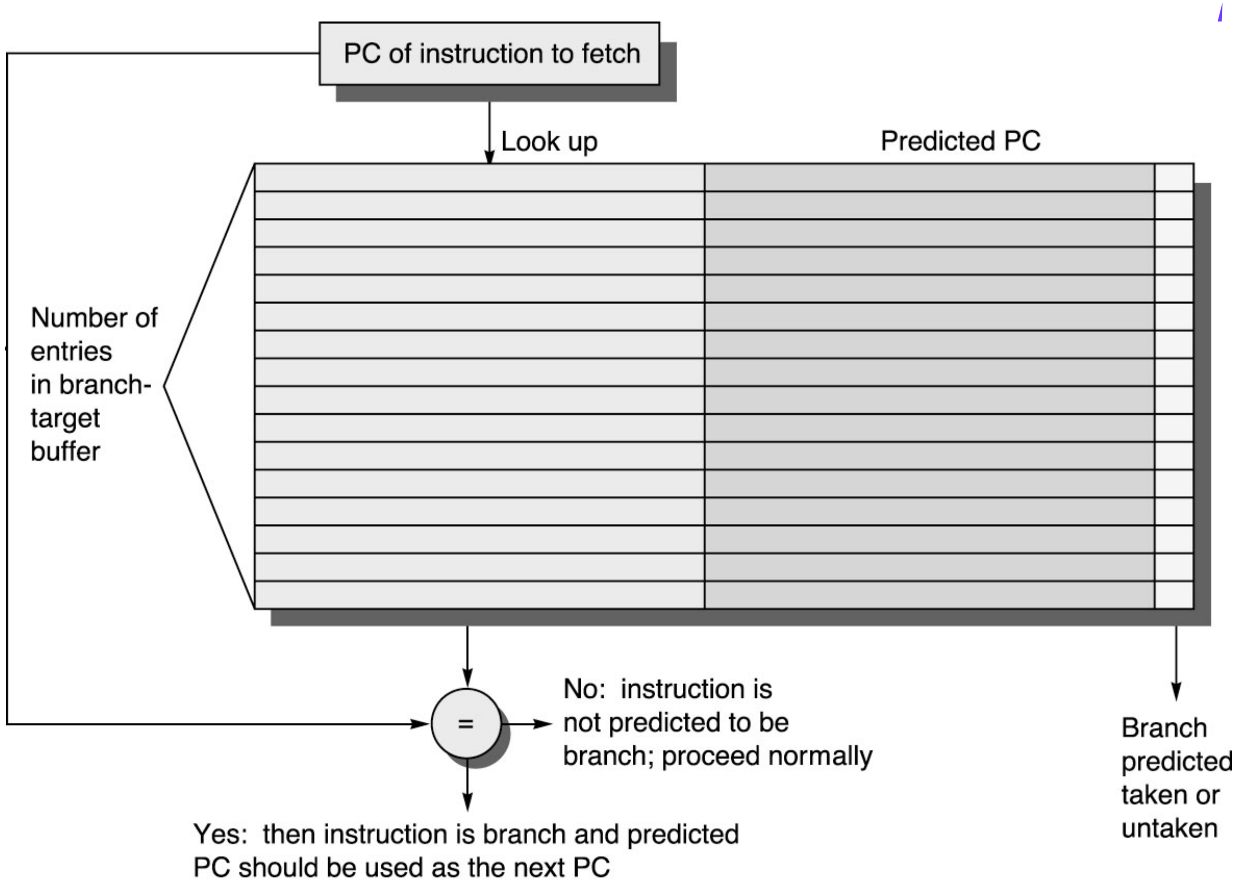
High-Performance Instruction Delivery

- For a multiple issue processor, predicting branches well is not enough
- Deliver a high-bandwidth instruction stream is necessary (e.g., 4~8 instructions/cycle)
  - Increasing instruction fetch bandwidth
  - Speculation (branch, value prediction)

Increasing Instruction Fetch Bandwidth

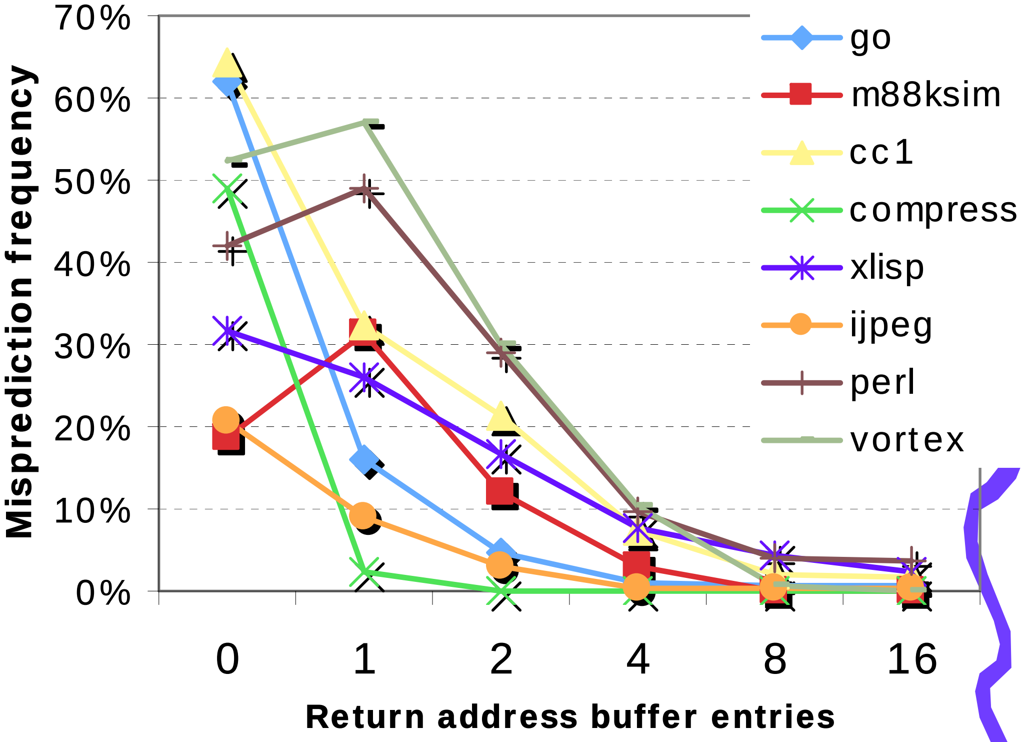
- Predicts next instruct address, sends it out *before* decoding instruction
- PC of branch sent to BTB
- When match is found, Predicted PC is returned
- If branch predicted taken, instruction fetch continues at Predicted PC

Branch Target Buffer (BTB)


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

IF BW: Return Address Predictor

- Small buffer of return addresses acts as a stack
- Caches most recent return addresses
- Call ⇒ Push a return address on stack
- Return ⇒ Pop an address off stack & predict as new PC



More Instruction Fetch Bandwidth

- Integrated branch prediction: branch predictor is part of instruction fetch unit and is constantly predicting branches
- Instruction prefetch: Instruction fetch units prefetch to deliver multiple instructions per clock, integrating it with branch prediction
- Instruction memory access and buffering: Fetching multiple instructions per cycle:
  - May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks)
  - Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed

Speculation: Register Renaming vs. ROB

- Alternative to ROB is a larger physical set of registers combined with register renaming
  - Extended registers replace function of both ROB and reservation stations
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits
- Most Out-of-Order processors today use extended registers with renaming

Value Prediction

- Attempts to predict value produced by instruction
  - E.g., Loads a value that changes infrequently
- Value prediction is useful only if it significantly increases ILP
  - Focus of research has been on loads; so-so results, no processor uses value prediction
- Related topic is *address aliasing prediction*
  - RAW for load and store or WAW for 2 stores
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
  - Has been used by a few processors

Remarks

- Interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Conservative in ideas, just faster clock and bigger
- Processors of last 5 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units $\Rightarrow$ performance 8 to 16X
- Peak vs. delivered performance gap increasing

In Conclusion …

- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with *precise exceptions* provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  - Important enabling factor for out-of-order execution