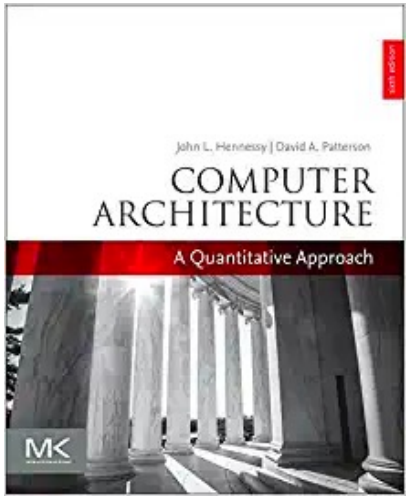


Chapter 3 (and Appendix C)

Instruction-Level Parallelism and Its Exploitation

Part 3: Advanced Scheduling

"Who's first?"
"America."
"Who's second?"
"Sir, there is no second."

-Dialog between two observers of the sailing race later named "The America's Cup" and run every few years -- the inspiration for John Cocke's naming of the IBM research processor as "America." This processor was the precursor to the RS/6000 series and the first superscalar microprocessor.

Acknowledgements

- Thanks to many sources for slide material

© 1990 Morgan Kaufmann Publishers, © 2001-present Elsevier
Computer Architecture: A Quantitative Approach by J. Hennessy & D. Patterson

© 1994 Morgan Kaufmann Publishers, © 2001-present Elsevier
Computer Organization and Design by D. Patterson & J. Hennessy

© 2002 K. Asinovic & Arvind, MIT

© 2002 J. Kubiatowicz, University of California at Berkeley

© 2006, © 2010 No Starch Press for Inside the Machine by J. Stokes

© 2007 W.-M. Hwu & D. Kirk, University of Illinois & NVIDIA

© 2007-2010 J. Owens, University of California at Davis

© 2010 CRC Press for Introduction to Concurrency in Programming Languages by M. Sottile, T. Mattson, and C. Rasmussen

© 2017, IBM POWER9 Processor Architecture by Sadasivam et al., IBM

© 2016, © 2019 POWER9 Processor User's Manual, IBM

© The OpenPOWER Foundation

© 2022, W. Feng, Virginia Tech

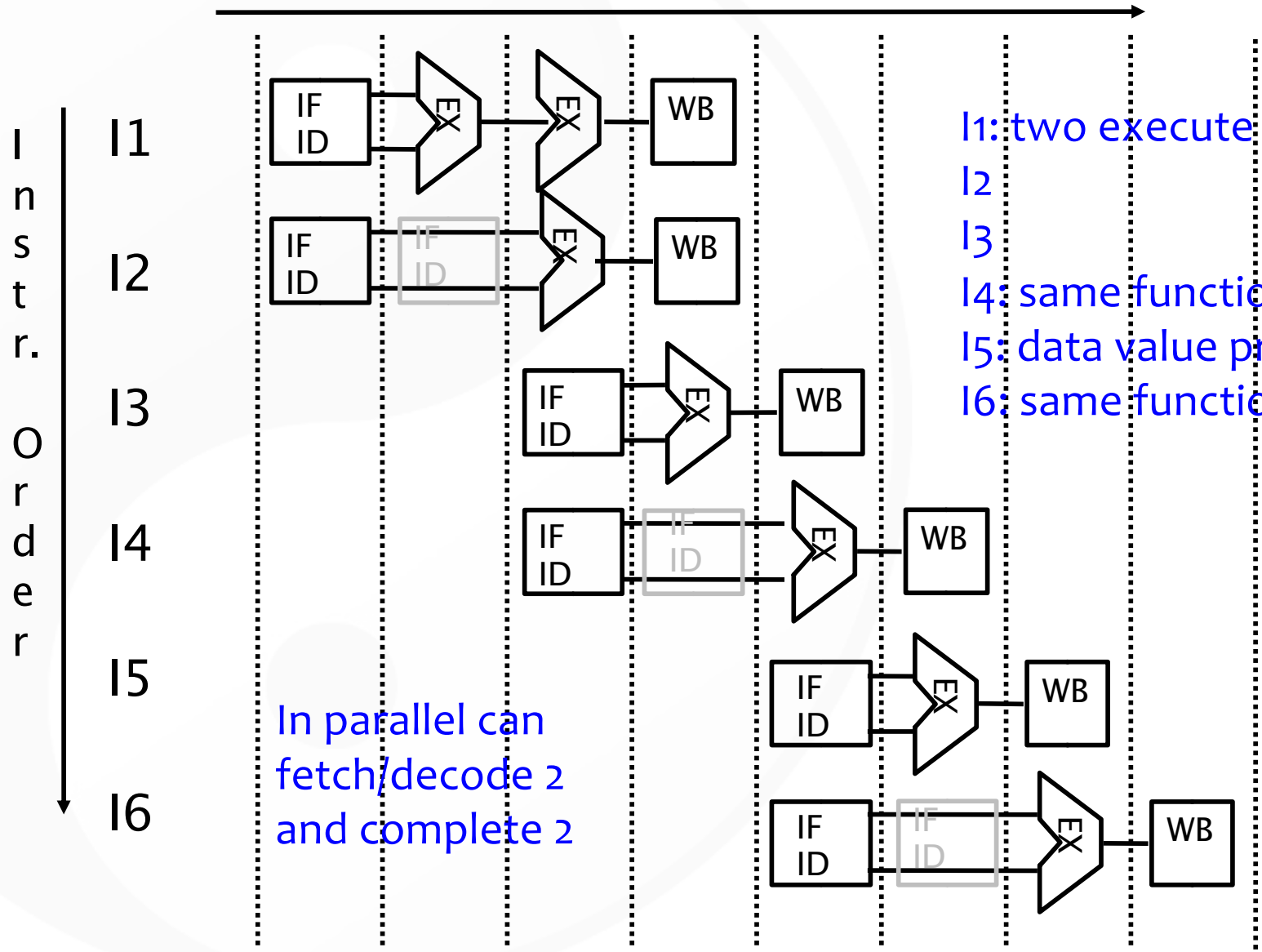
Instruction Issue and Completion

- Instruction issue—initiate execution
  - Instruction lookahead capability—fetch, decode and issue instructions beyond the current instruction
- Instruction completion—complete execution
  - Processor lookahead capability—complete issued instructions beyond the current instruction
- Instruction commit—write back results to the RegFile or D$ (i.e., change the machine state)
- Scenarios
  - In-order issue w/ in-order completion
  - In-order issue w/ out-of-order completion
  - Out-of-order issue w/ out-of-order completion & in-order commit
  - Out-of-order issue w/ out-of-order completion

In-Order Issue, In-Order Completion (IOI-IOC)

- Simplest policy is to issue instructions in exact program order and to complete them in the same order they were fetched (i.e., in program order)
- Example
  - Assume pipelined processor that can ***fetch & decode 2 instructions per cycle***, has ***3 functional units*** (single-cycle adder, a single-cycle shifter, and a two-cycle multiplier), and can ***complete (and write back) two results per cycle***
  - Instruction sequence:
    I1 – needs two execute cycles (a multiply)
    I2
    I3
    I4 – needs the same function unit as I3
    I5 – needs data value produced by I4
    I6 – needs the same function unit as I5

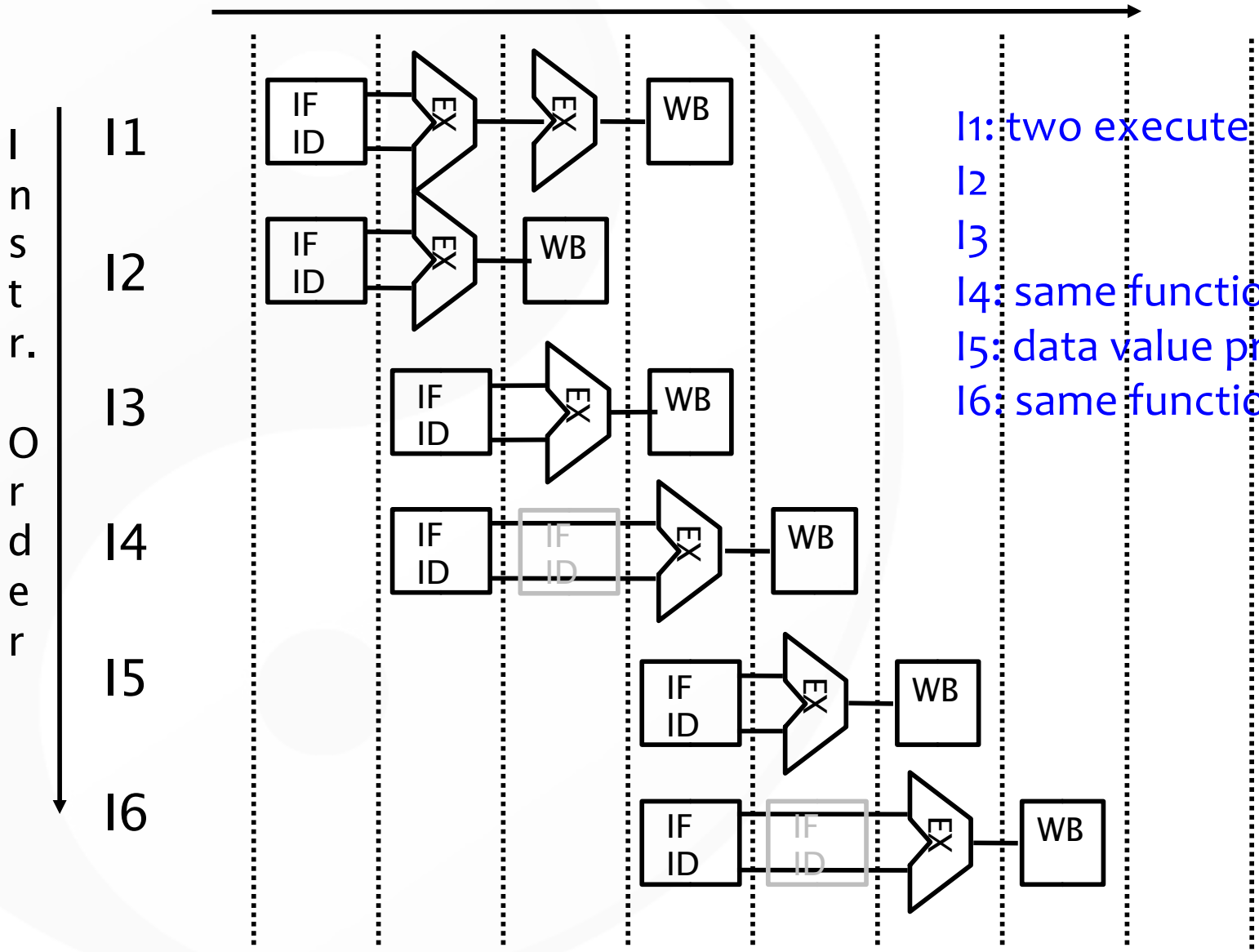
Example: IOI-IOC



In-Order Issue, Out-of-Order Completion

- With out-of-order completion, a later instruction may complete before a previous instruction
  - Out-of-order completion is used in single-issue pipelined processors to improve the performance of program that has long-latency operations such as divide
- When using out-of-order completion, instruction issue is stalled when there is a resource conflict (e.g., for a functional unit) or when the instructions ready to issue need a result that has not yet been computed

Example: IOI-OOC



I1: two execute cycles
I2
I3
I4: same function unit as I3
I5: data value produced by I4
I6: same function unit as I5

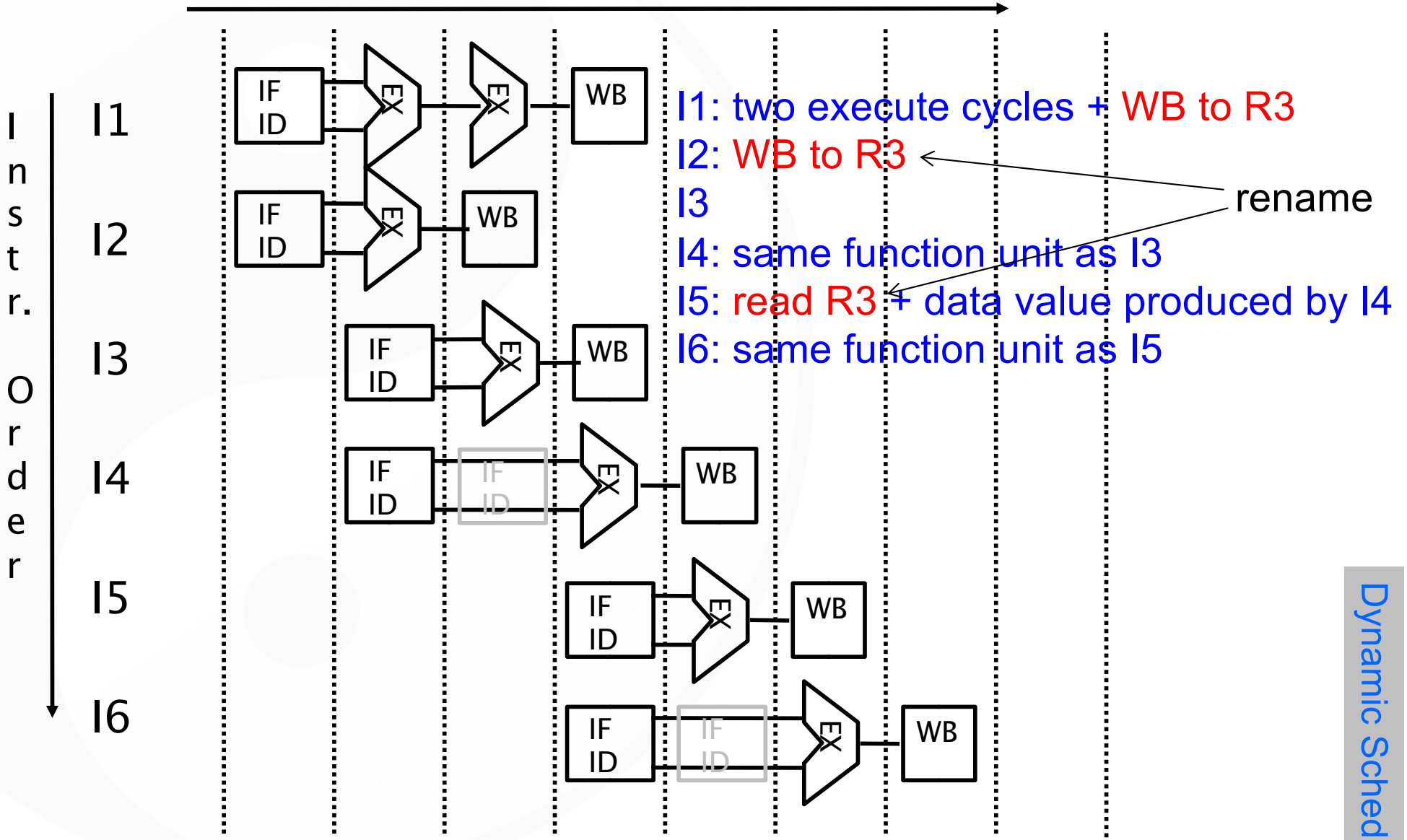
Handling Output Dependencies

- One more situation that stalls instruction issue w/ IOI-OOC

  I1 – writes to R3
  I2 – writes to R3
  I5 – reads R3

- If the I1 write occurs after the I2 write, then I5 reads an incorrect value for R3
- I2 has an output dependency on I1—write after write
- The issuing of I2 would have to be stalled if its result might later be overwritten by a previous instruction (i.e., I1) that takes longer to complete—the stall happens before instruction issue
- While IOI-OOC yields higher performance, it requires more dependency checking hardware ***(both write-after-read and write-after-write hazards)***

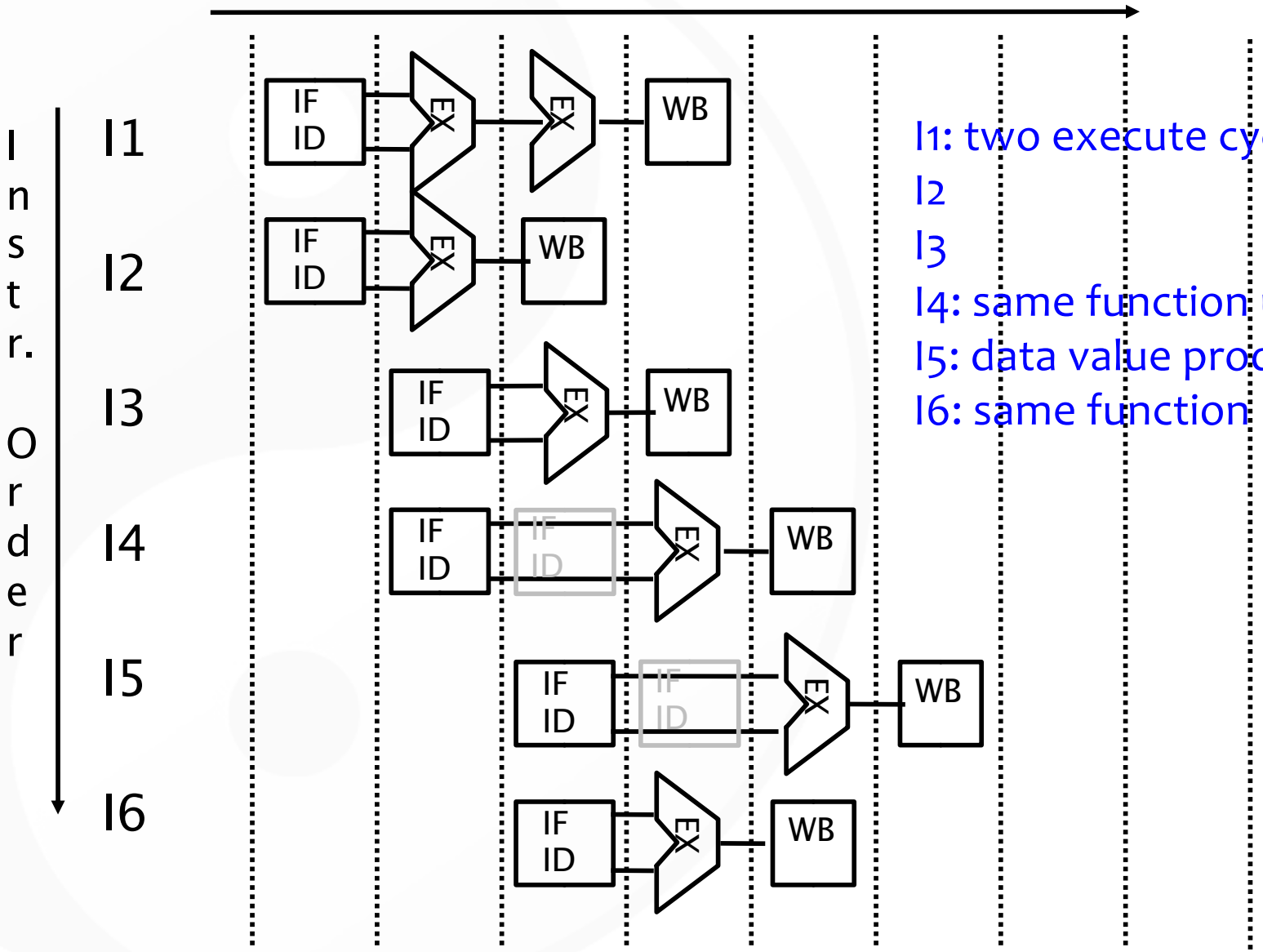
Example: IOI-OOC



Out-of-Order Issue with Out-of-Order Completion (OOI-OOC)

- With *in-order issue*, CPU stops ID when decoded instruction has resource conflict or a data dependency on an issued but uncompleted instruction
  - CPU is unable to look beyond conflicted instruction even though more downstream instructions might have no conflicts and thus be issuable
- Fetch & decode instructions beyond the conflicted one ("instruction window": Tetris), store them in an instruction buffer (if room), and flag those instructions in buffer that do *NOT* have resource conflicts or data dependencies
- Flagged instructions are then issued from the buffer w/o regard to their program order → re-order buffer (ROB)

OOI-OOC Example



I1: two execute cycles
I2
I3
I4: same function unit as I3
I5: data value produced by I4
I6: same function unit as I5

Overview: Dynamic Scheduling

- Definition: Rearrange order of instructions to reduce stalls while maintaining data flow
- Implications
  - Out-of-order execution
  - Out-of-order completion

Creates the possibility for WAR and WAW hazards

Tomasulo's Algorithm

- Advantages
  - Compiler does NOT need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantages
  - Substantial increase in hardware complexity
  - Complicates exception-handling

Recap: Dependencies

- Each of the three data dependencies …
  - True data dependencies (read after write)
  - Antidependencies (write after read)
  - Output dependencies (write after write)

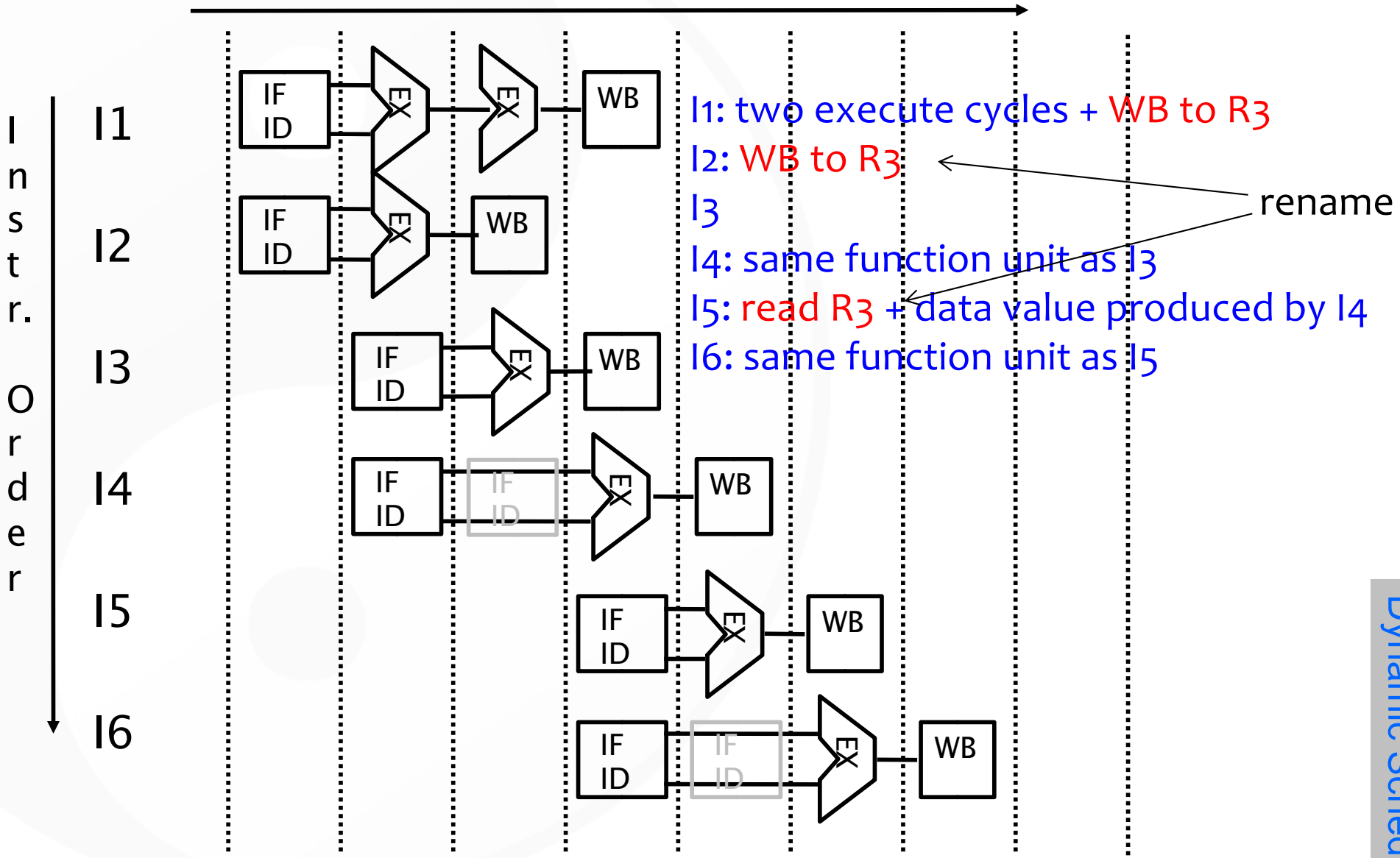
  … manifests itself through the use of registers (or other storage locations) } storage conflicts

- True dependencies represent the flow of data and information through a program
- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations

Recap: Pipeline Scheduling

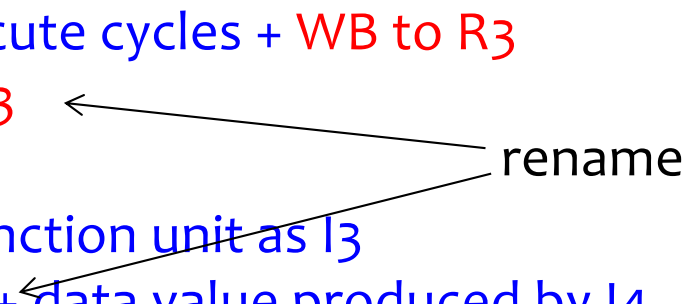
- Statically Scheduled Pipeline
  - Fetches an instruction and issues it unless there is a data dependence that cannot be hidden with forwarding.
  - *Implication: In-order issue; in-order completion*
- Dynamically Scheduled Pipeline
  - Rearranges the order of instructions to reduce stalls while maintaining data flow
  - *Implication: In-/out-of-order issue; out-of-order completion*
- Observation
  - Commonality: Both must maintain data flow for correctness.
  - Difference: Other than static vs. dynamic …
    - Static: Minimize stalls by separating dependent instructions
    - Dynamic: Minimize stalls by re-arranging instructions based on dynamic presence of dependences

Example: IOI-OOC



Dynamic Scheduling with Tomasulo's Algorithm

- Tracks when operands are available (minimize RAW hazards)
- Introduces register renaming in hardware
  - Addresses WAW & WAR hazards
- Extends to handle speculation
  - Technique used to reduce effect of control dependences



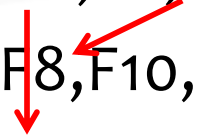
- IF precedes ID and fetches into an IR or equivalent (e.g., a queue of pending instructions → re-order buffer).
  ID has two sub-stages:
  - Issue: Decode instruction + *check for structural hazards*
  - Read Operands: *Wait until no data hazards*

Delineate between when instruction *begins exect'n* & when it *completes execut'n*

Another Example: IOI-OOC Introduces WAR & WAW

- Can dynamically schedule with IOI-OOC
  - OOC introduces possibility of WAR and WAW hazards, which are NOT issues in a statically scheduled pipeline.

```
DIV.D F0, F2, F4
ADD.D F6,F0,F8
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```



Antidependence between ADD.D and SUB.D
- If pipeline tries SUB.D before ADD.D, antidependence violated & yields WAR hazard.

Output dependence between ADD.D and MUL.D
- If pipeline tries MUL.D before ADD.D, output dependence violated & yields WAW hazard.

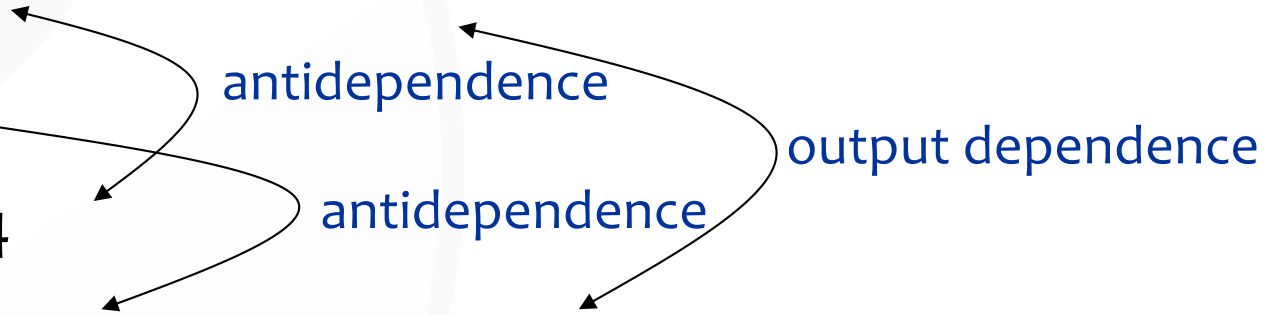
- How to avoid the above? Register renaming
  - Eliminates WAR & WAW hazards by renaming all destination registers, including those with a *pending* read or write for an earlier instruction, so that the out-of-order write does *not* affect any instructions that depend on an earlier value of an operand.

Register Renaming

1. Identify all the data hazards for an IOC pipeline.
2. Identify all the data hazards for an OOC pipeline.

- Example

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8
S.D     F6,0(R1)
SUB.D   F8,F10,F14
MUL.D   F6,F10,F8
```



Two antidependences (WAR) and an output dependence (WAW)

Note: The above code snippet uses the MIPS ISA. The textbook uses the RISC V ISA.

Register Renaming

Example

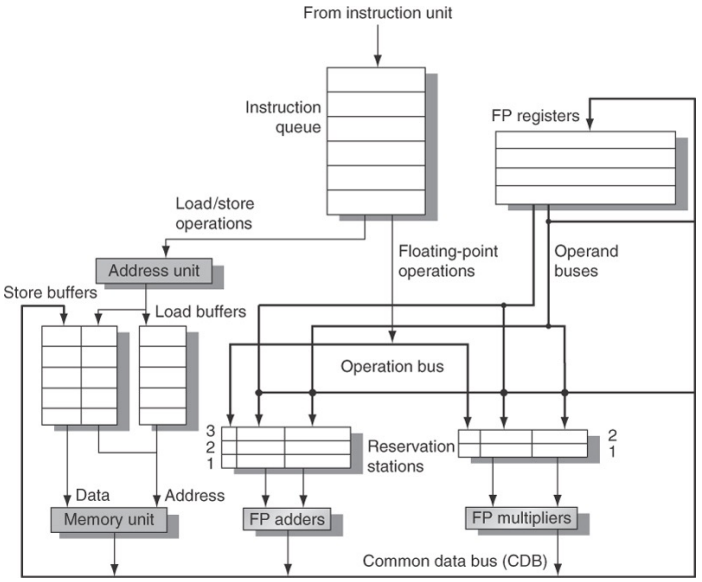
```
DIV.D   F0,F2,F4
ADD.D   S,F0,F8
S.D     S,0(R1)
SUB.D   T,F10,F14
MUL.D   F6,F10,T
```

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8
S.D     F6,0(R1)
SUB.D   F8,F10,F14
MUL.D   F6,F10,F8
```

- Now only RAW hazards remain, which can be strictly ordered
- Any subsequent uses of F8 must be replaced by the register T

Note: The above code snippet uses the MIPS ISA. The textbook uses the RISC V ISA.

Register Renaming

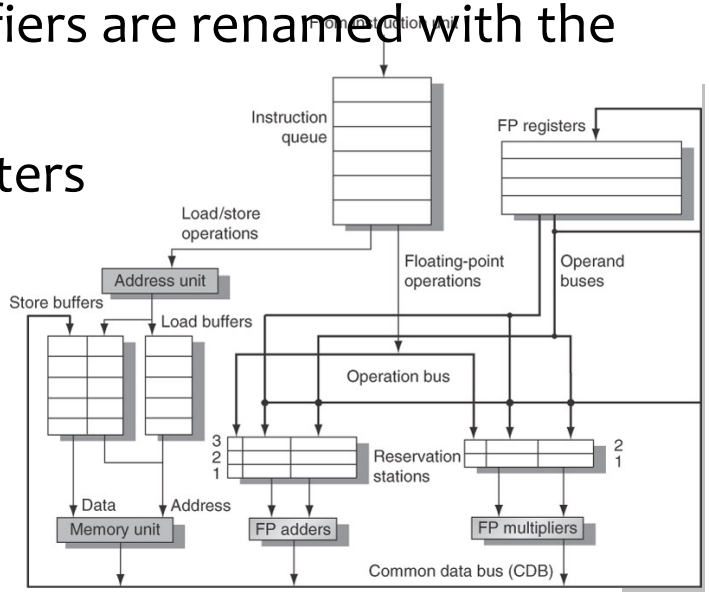


- Tomasulo's Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware via reservation stations (RS)
    - Minimizes WAW and WAR hazards (by renaming destination regs)

- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values

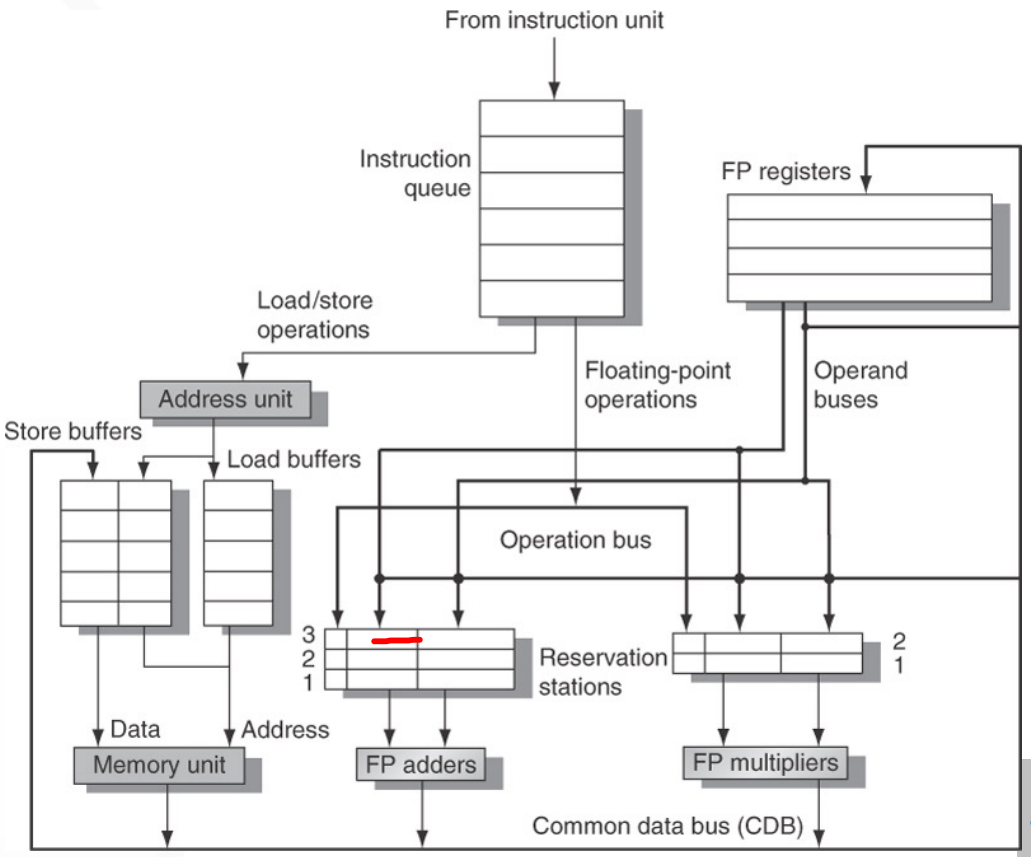
Register Renaming

- Register renaming provided by reservation stations (RS)
  - RS fetches and buffers an operand as soon as it becomes available, *eliminating the need to get the operand from a register*
  - Pending instructions designate the RS to which they will send their output
    - Result values broadcast on a result bus called common data bus (CDB)
  - Only the last output updates the register file
- Observations
  - As instructions are issued, the register specifiers are renamed with the reservation station
  - May be more reservation stations than registers
  - Load and store buffers
    - Contain data and addresses, act like reservation stations



High-Level Architecture for Tomasulo's Algorithm

- Register renaming via reservation stations (RS)
- Load and store buffers
  - Contain data & addresses, act like reservation stations
- Advantages
  - Distribution of hazard detection logic
  - Elimination of stalls for WAW and WAR hazards by renaming all destination registers



No execution tables shown.

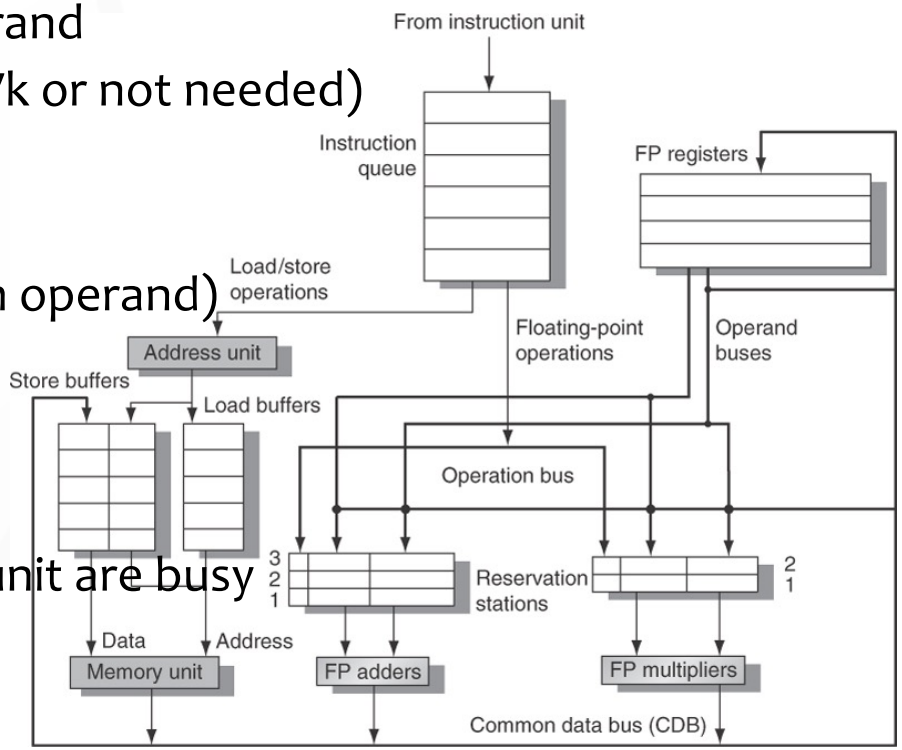
Tomasulo's Algorithm at High Level

- Three steps that an instruction goes through …
  - Issue (or Dispatch), where register renaming occurs
    - Get next instruction from FIFO queue
    - If available RS, issue instruction to RS w/ operand values, if available. If operand values not available, stall instruction in RS and *keep track of the functional units that will produce operand values.*
    - If no available RS, structural hazard → stall until RS or buffer free …
  - Execute
    - When operand available (via CDB), store it in any RS waiting for it
    - When all operands ready (i.e., no RAW hazard), issue instruction
    - Loads & store maintained in program order via effective address calc.
    - No instruction allowed to initiate execution until all branches that precede it in program order have completed (to preserve exception behavior)
  - Write result
    - Write result on CDB into registers and RS (including store buffers)
      - (Stores must wait until address *and* value are received)

High-Level Architecture for Tomasulo's Algorithm

Each RS has seven fields:

- Op
  - Op to do on src operands
- Qj, Qk
  - RS that produce corresponding src operand
  - (Zero value means src operand in Vj or Vk or not needed)
- Vj, Vk
  - Value of src operands
  - (Only one V field or Q field valid for each operand)
- A
  - Holds info for memory addr calc
- Busy
  - Indicates RS and associated functional unit are busy



Example: Tomasulo's in Action

		Instruction status		
Instruction		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Assume the following execute latencies:

- L.D 1 cycle
- ADD.D 2 cycles
- MUL.D 6 cycles
- DIV.D 12 cycles

What do the status tables look like when MUL.D is ready to write its result?

Let's find out!

Note: The above code snippet uses the MIPS ISA. The textbook uses the RISC V ISA.

Tomasulo's w/ Straightline Code

Instruction		Instruction status: Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2							
Add1							
Add2							
Add3							
Mult1							
Mult2							

Reservation stations

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi									

Register status

Assume the following execute latencies:

- L.D 1 cycle
- ADD.D 2 cycles
- MUL.D 6 cycles
- DIV.D 12 cycles

What do the status tables look like when MUL.D is ready to write its result?

Note: The above code snippet uses the MIPS ISA. The textbook uses the RISC V ISA.

Tomasulo's with Loop-Based Code

RISC-V Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

Assumption

- Predict branches taken.
  Implication: RS allow multiple executions of the loop to run simultaneously, an advantage gained WITHOUT changing the code. Loop is unrolled dynamically by hardware using the RS obtained by "register renaming."
  (Ignore integer ALU operation for now as branch is predicted as taken.)

Tomasulo's with Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

Observations

- A load (`fld`) and a store (`fsd`) can be safely done *out of order* iff they access different locations, but what if they access the same?
  - Load is before store in program order & interchanging them → WAR hazard
  - Store is before load in program order & interchanging them → RAW hazard
  - Also, interchanging two stores → WAW hazard

How to Detect Hazards on Loads & Stores?

Tomasulo's with Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

How to Detect Hazards on Loads & Stores?

- Loads
  - Compute the data memory address associated with any earlier memory operation → pragmatically check the "A" field of all active (store) buffers
  - If load address matches address of any active entry in store buffer, load is *not* sent to load buffer until conflicting store completes (i.e., respect RAW hazard)
- Stores
  - Similar to loads except that CPU must check for conflicts on both the load buffers AND store buffers (since conflicting stores cannot be reordered with respect to either a load or a store, i.e., WAR or WAW).

Tomasulo's with Loop-Based Code

		Instruction status			
Instruction		From iteration	Issue	Execute	Write result
fld	f0,0(x1)	1	√	√	
fmul.d	f4,f0,f2	1	√		
fsd	f4,0(x1)	1	√		
fld	f0,0(x1)	2	√	√	
fmul.d	f4,f0,f2	2	√		
fsd	f4,0(x1)	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					Regs[x1] + 0
Load2	Yes	Load					Regs[x1] – 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[f2]	Load1		
Mult2	Yes	MUL		Regs[f2]	Load2		
Store1	Yes	Store	Regs[x1]			Mult1	
Store2	Yes	Store	Regs[x1] – 8			Mult2	

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

VLIW approach later

Dynamic Scheduling

Adoption of Tomasulo's Algorithm

- Proposed in 1967! 😳
- Not widely adopted until … the 1990s!
  - Intro of caches created unpredictable delays → need for *dynamic scheduling*
    - Out-of-order execution (and completion) allows CPU to continue executing instructions while waiting the completion of a cache miss, thus hiding all or part of the cache-miss penalty.
      - What kind of advanced cache optimization is this?
  - CPUs became more aggressive about their issue capability while CPU designers concerned with the performance of difficult-to-schedule code (e.g., non-numeric code → graphs) → need for *automated register renaming, dynamic scheduling, and speculation* (coming up next)
  - Can achieve higher performance without the compiler needing to target code to a specific pipeline structure (i.e., architecture)

Recap: Dynamic Scheduling

- Dynamic scheduling implies
  - Out-of-order execution
  - Out-of-order completion
- Creates the possibility for WAR and WAW hazards
- Tomasulo's Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results *if prediction was correct*

- Instruction commit
  - Allow an instruction to update the register file when instruction is no longer speculative

- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - i.e., updating state or taking an execution

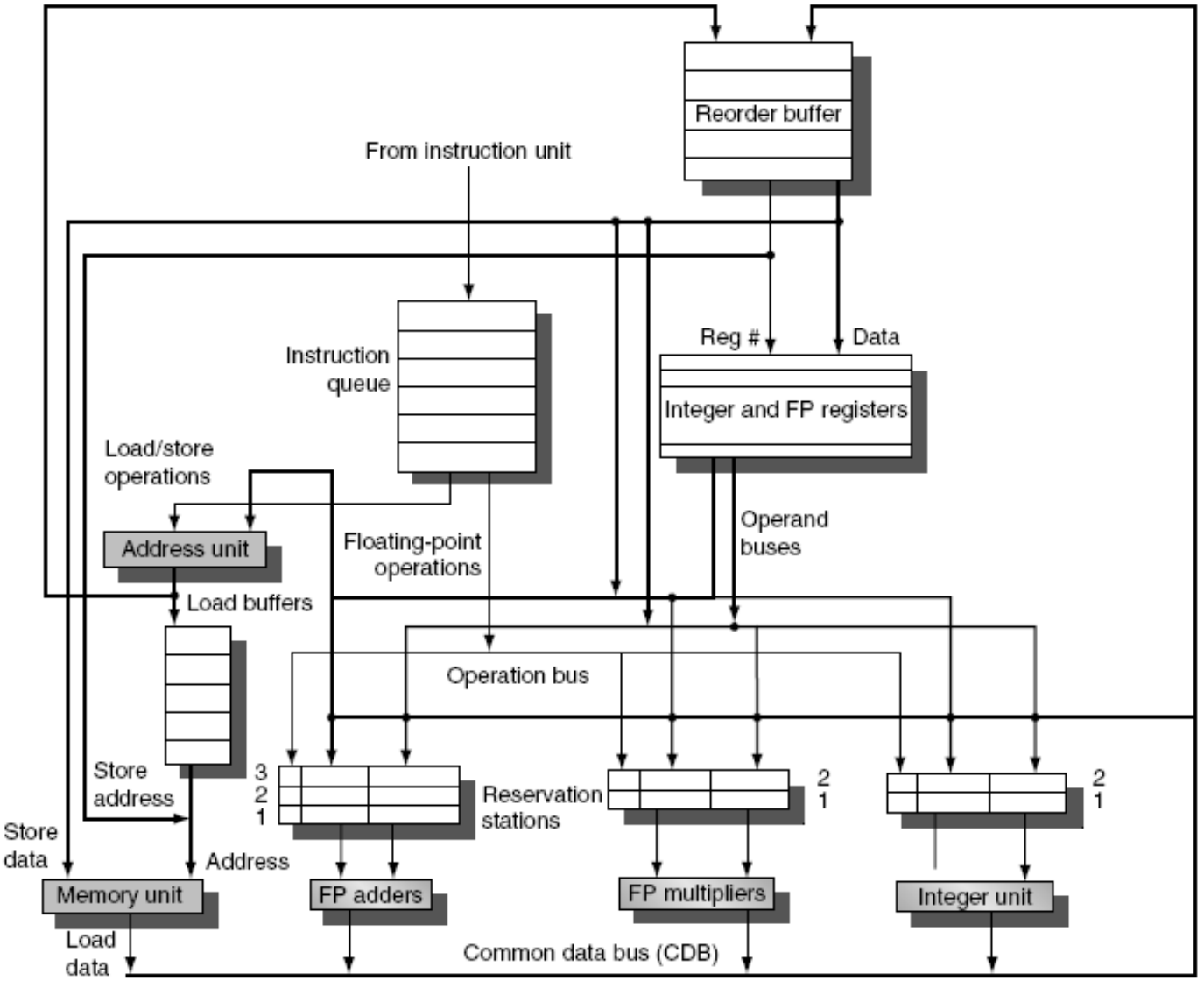
Hardware-Based Speculation

- Dynamic Scheduling vs. (Hardware-Based) Speculation
- Dynamic Scheduling
  - Fetch, decode, and issue instructions
- Speculation
  - Fetch, decode, issue, and *execute* instructions
- Key Ideas Behind Hardware-Based Speculation
  1. Dynamic branch prediction
     - Choose which instructions to execute
  2. Speculation
     - Execute instructions before control dependences are resolved
  3. Dynamic scheduling
     - Deal with the scheduling of different combinations of basic blocks

Re-order Buffer (ROB)

- Re-order buffer
  - Holds the result of instruction between completion and commit
- Four fields
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations
  - Operand source is now reorder buffer instead of functional unit

Tomasulo's with Reorder Buffer



Reorder Buffer

- Issue
  - Allocate RS and ROB; read available operands
- Execute
  - Begin execution when operand values are available
- Write result
  - Write result and ROB tag on CDB
- Commit
  - When ROB reaches head of ROB, update register
  - When a mispredicted branch reaches head of ROB, discard all entries

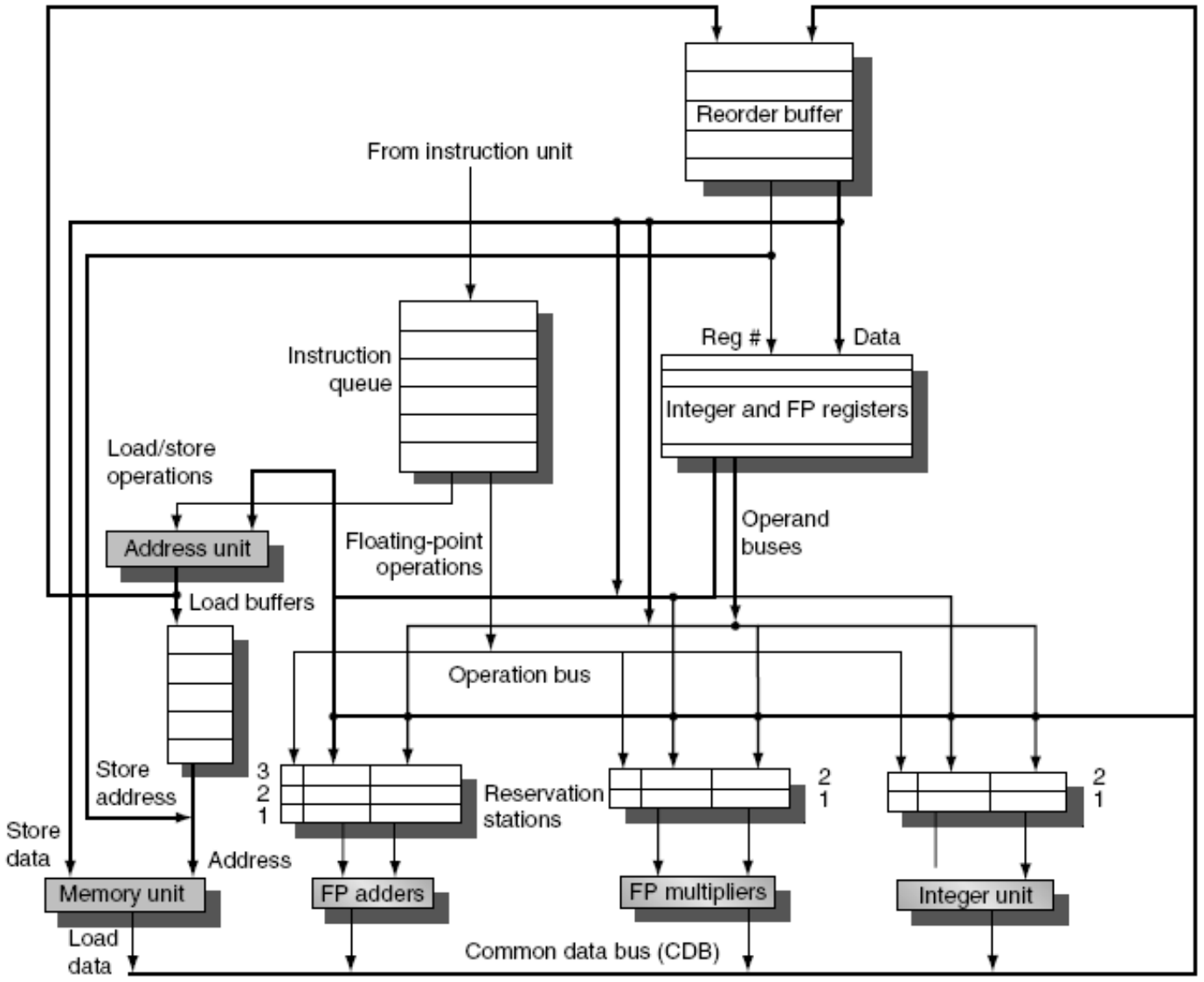
Re-order Buffer (ROB)

- Register values and memory values are not written *until* an instruction commits

- On misprediction
  - Speculated entries in reorder buffer (ROB) are cleared

- Exceptions
  - Not recognized until it is ready to commit

Tomasulo's with Reorder Buffer



ROB w/ Previous Straightline Code

(no loop)

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	L.D	F6,32(R2)	Commit	F6	Mem[32 + Regs[R2]]
2	No	L.D	F2,44(R3)	Commit	F2	Mem[44 + Regs[R3]]
3	Yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[f4]
4	Yes	SUB.D	F8,F2,F6	Write result	F8	#2 – #1
5	Yes	DIV.D	F10,F0,F6	Execute	F10*	
6	Yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL	Mem[44 + Regs[R3]]	Regs[f4]			#3	
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	#3		#5	

FP register status

Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

ROB with Loop-Based Code

MIPS Loop-Based Code

```
Loop:  L.D    F0,0(R1)
       MUL.D  F4,F0,F2
       S.D    F4,0(R1)
       ADDI   R1,R1,#8
       BNE    R1,R2,Loop    // branches if R1 != R2
```

RISC-V Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

ROB with Loop-Based Code

RISC-V Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

Assumption

- Predict branches taken.
  Implication: RS allow multiple executions of the loop to run simultaneously, an advantage gained WITHOUT changing the code. Loop is unrolled dynamically by hardware using the RS obtained by "register renaming." (Ignore integer ALU operation for now as branch is predicted as taken.)

ROB with Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

Observations

- A load (`fld`) and a store (`fsd`) can be safely done *out of order* iff they access different locations, but what if they access the same?
  - Load is before store in program order & interchanging them → WAR hazard
  - Store is before load in program order & interchanging them → RAW hazard
  - Also, interchanging two stores → WAW hazard

How to Detect Hazards on Loads & Stores?

ROB with Loop-Based Code

```
Loop:  fld    f0,0(x1)
       fmul.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,8
       bne    x1,x2,Loop    // branches if x1 != x2
```

How to Detect Hazards on Loads & Stores?

- Loads
  - Compute the data memory address associated with any earlier memory operation → pragmatically check the "A" field of all active (store) buffers
  - If load address matches address of any active entry in store buffer, load is *not* sent to load buffer until conflicting store completes (i.e., respect RAW hazard)
- Stores
  - Similar to loads except that CPU must check for conflicts on both the load buffers AND store buffers (since conflicting stores cannot be reordered with respect to either a load or a store, i.e., WAR or WAW).

ROB with Loop-Based Code

Instruction		Instruction status: From iteration	Issue	Execute	Write result
fld	f0,0(x1)	1	√	√	
fmul.d	f4,f0,f2	1	√		
fsd	f4,0(x1)	1	√		
fld	f0,0(x1)	2	√	√	
fmul.d	f4,f0,f2	2	√		
fsd	f4,0(x1)	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					Regs[x1] + 0
Load2	Yes	Load					Regs[x1] – 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[f2]	Load1		
Mult2	Yes	MUL		Regs[f2]	Load2		
Store1	Yes	Store	Regs[x1]			Mult1	
Store2	Yes	Store	Regs[x1] – 8			Mult2	

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

Tomasulo's Algorithm

VLIW approach later

Adoption of Tomasulo's

- Proposed in 1967! 😳
- Not widely adopted until … the 1990s!
  - Intro of caches created unpredictable delays → need for *dynamic scheduling*
    - Out-of-order execution (and completion) allows CPU to continue executing instructions while waiting the completion of a cache miss, thus hiding all or part of the cache-miss penalty.
      - What kind of advanced cache optimization is this?
  - CPUs became more aggressive about their issue capability while CPU designers concerned with the performance of difficult-to-schedule code (e.g., non-numeric code → graphs) → need for *automated register renaming, dynamic scheduling, and speculation* (coming up next)
  - Can achieve higher performance without the compiler needing to target code to a specific pipeline structure (i.e., architecture)

Recall:

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results *if prediction was correct*

- Instruction commit
  - Allow an instruction to update the register file when instruction is no longer speculative

- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - i.e., updating state or taking an execution

Recall:

Hardware-Based Speculation

- Dynamic Scheduling vs. (Hardware-Based) Speculation
- Dynamic Scheduling
  - Fetch, decode, and issue instructions
- Speculation
  - Fetch, decode, issue, and *execute* instructions
- Key Ideas Behind Hardware-Based Speculation
  1. Dynamic branch prediction
     - Choose which instructions to execute
  2. Speculation
     - Execute instructions before control dependences are resolved
  3. Dynamic scheduling
     - Deal with the scheduling of different combinations of basic blocks

Dynamic, Multiple Issue, and Speculation

- Modern Microarchitectures
  - Dynamic scheduling + multiple issue + speculation
- Approaches
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches
- Issue logic is the bottleneck in dynamically-scheduled superscalar processors

Multiple Issue

- Limit the number of instructions of a given class that can be issued in a "bundle" (to simplify RS allocation)
  - Example: one FP, one integer, one load, one store
- Examine all the dependencies between the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Need multiple completion/commit

Recap

- What is instruction-level parallelism (ILP)?
  - A measure of how many of the operations in a computer program can be performed simultaneously.
  - Pipelining → universal technique in 1985
    - Overlaps execution of instructions → exploits ILP
- How do processors extract ILP to get faster?
  - More parallelism (or more work per pipeline stage): fewer clocks/instruction [more instructions/cycle]
    - Get WIDER
  - Deeper pipeline stages: fewer gates/clock
    - Get DEEPER
  - Transistors get faster (Moore's Law): fewer ps/gate
    - Get FASTER
- What do processors do to extract ILP?

Get WIDER: Multiple Issue

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions
  - Dynamically scheduled superscalar processors ✔
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors ✔

Get WIDER: Machine Parallelism

- Two approaches for machine parallelism, where the responsibility of resolving hazards is …
  - Hardware-based dynamic approaches
    - Dynamic-issue superscalar
    - Typically used in server and desktop processors
    - Not used as extensively in PMP processors
  - Software-based static approaches (i.e., compiler)
    - Static "VLIW: Very Long Instruction Word"
    - Not as successful outside of scientific applications

Multiple-Issue Processor Styles

- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
    - Example: IBM Power 2, Pentium Pro/2/3/4, Core, MIPS R10K, HP PA 8500

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
    - Example: Intel Itanium and Itanium 2 for the IA-64 ISA—EPIC (Explicit Parallel Instruction Computer)

Multiple-Issue Processor Styles

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Multiple-Issue Processor Styles

- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
    - Example: IBM Power 2, Pentium Pro/2/3/4, Core, MIPS R10K, HP PA 8500

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
    - Example: Intel Itanium and Itanium 2 for the IA-64 ISA—EPIC (Explicit Parallel Instruction Computer)

VLIW Beginnings

- VLIW: Very Long Instruction Word

[4] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. 10th Symp. Comput. Architecture*, IEEE, June 1983, pp. 140–150.

- Led to a startup (MultiFlow) whose computers worked, but which went out of business ... the ideas remain influential.

History of VLIW Processors

- Started with (horizontal) microprogramming
  - Very-wide microinstructions used to directly generate control signals in single-issue processors (e.g., IBM 360 series)
- VLIW for multi-issue processors first appeared in the Multiflow and Cydrome (in the early 1980's)
- Commercial VLIW processors from past decade
  - Intel i860 RISC (dual mode: scalar and VLIW)
  - Intel I-64 (EPIC: Itanium and Itanium 2)
  - Transmeta Crusoe (the chip that powered Green Destiny)
  - Lucent/Motorola StarCore, ADI TigerSHARC, Infineon (Siemens) Carmel

Overview: VLIW Processors

- Package multiple operations into one instruction
- Example: VLIW processor
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

VLIW Processors
(Static Multiple-Issue Machines)

- Static multiple-issue processors (aka VLIW) use the compiler to decide which instructions to issue and execute simultaneously
  - Issue packet or "bundle"
    - The set of instructions that are bundled together and issued in one clock cycle—think of it as *one large instruction with multiple operations*
    - The mix of instructions in the packet (bundle) is usually restricted—a single "instruction" with several predefined fields
      - e.g., VLIW: one integer (or branch), two FP, two memory
  - Performance
    - Must be enough parallelism in code to fill the available slots
    - The compiler does static branch prediction and code scheduling to reduce (ctrl) or eliminate (data) hazards

VLIW Processors

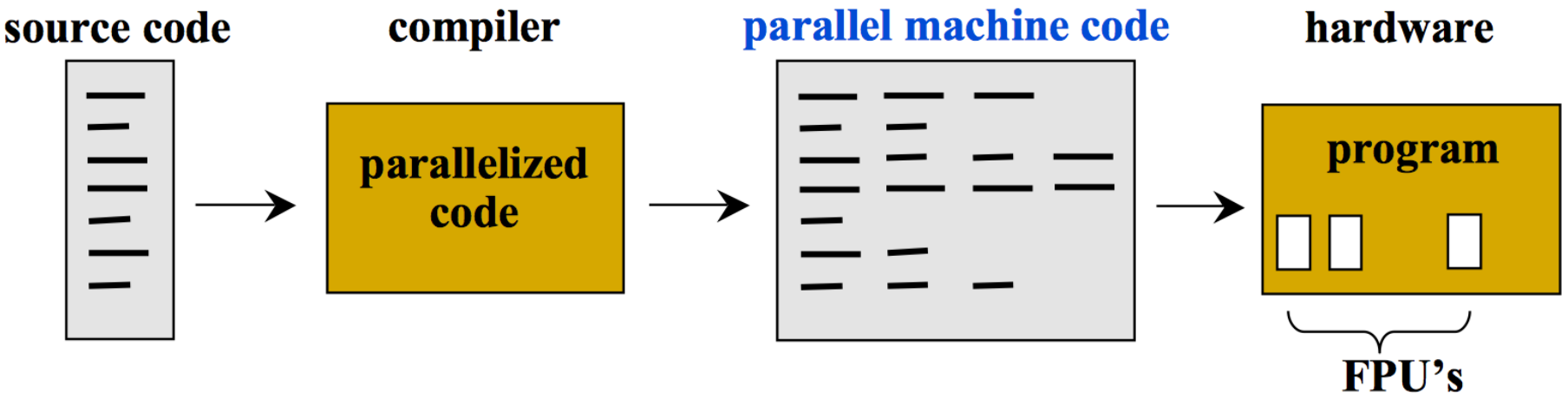
- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instructions per bundle & add more FUs
- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

RISC-V Code Snippet

```
Loop: fld     f0,0(x1)
      fadd.d  f4,f0,f2
      fsd     f4,0(x1)
      addi    x1,x1,-8
      bne     x1,x2,Loop
```

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

VLIW in Illustration



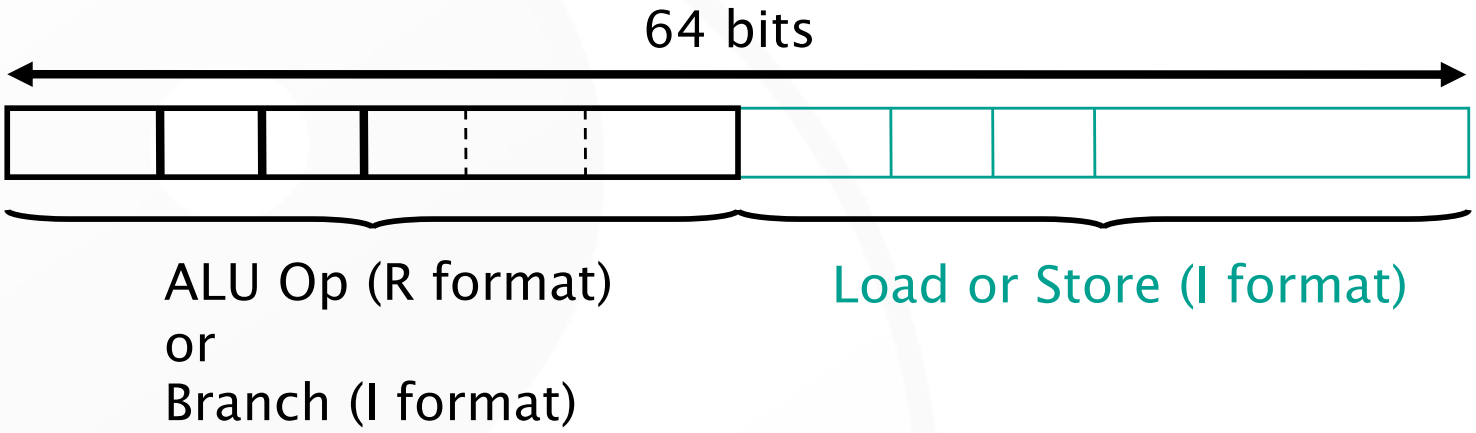
Characteristics

- Hardware does *not* need to re-analyze code to detect dependences
- Hardware does *not* perform OoO execution
- Each instruction controls multiple functional units
- Each instruction is explicitly parallel

Static Multiple-Issue Machines (VLIW)

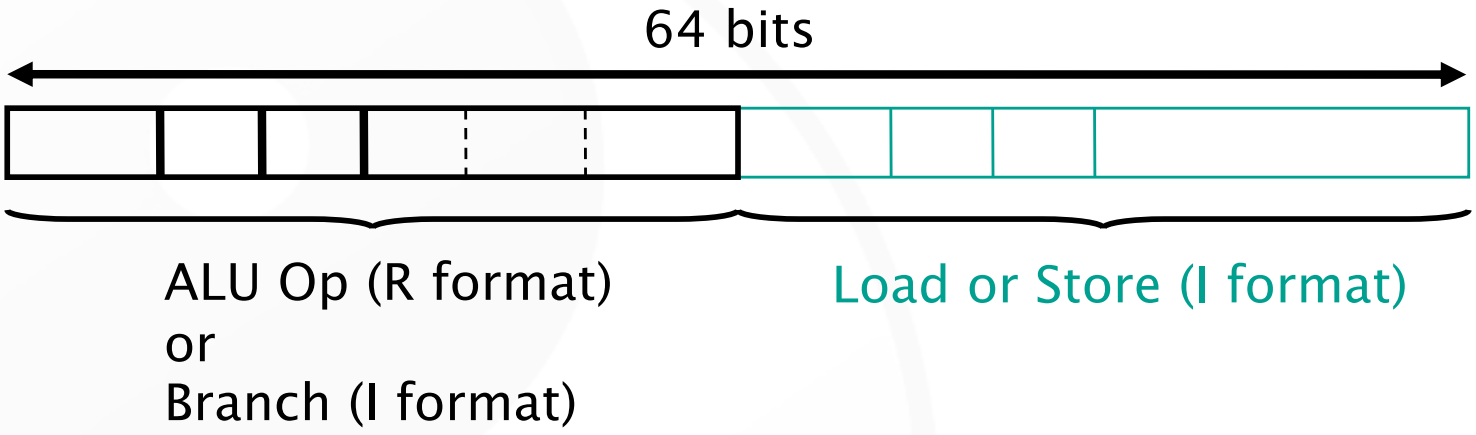
- Profile of VLIWs
  - Multiple functional units (like SS processors)
  - Multi-ported register files (again like SS processors)
  - Wide program bus
- Example of VLIW Processor
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Key
  - Must be enough parallelism in code to fill the available slots

An Example: VLIW MIPS



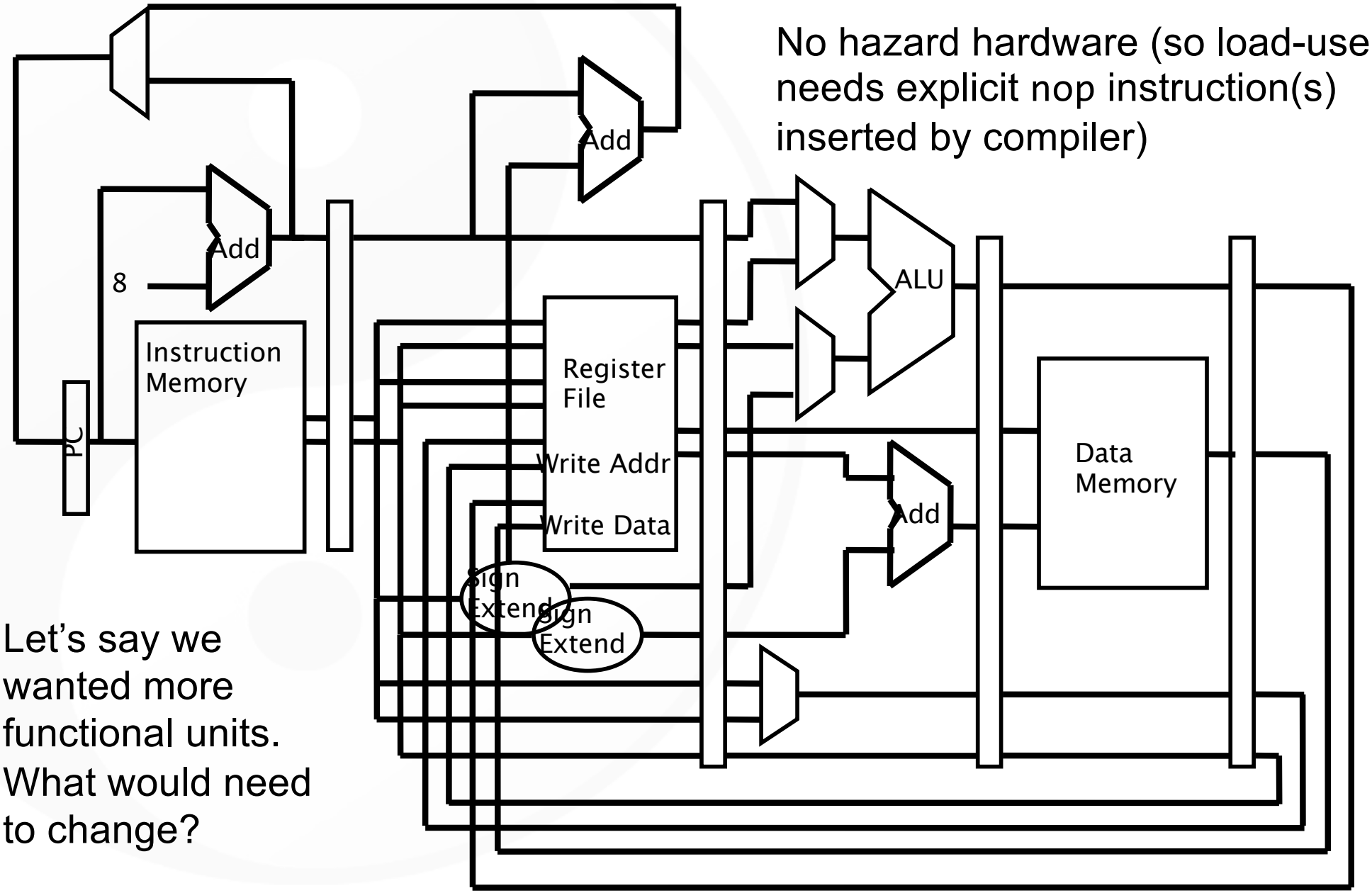
- Consider a 2-issue MIPS with a 2-instr bundle
- Instructions are always fetched, decoded, and issued in pairs
  - What happens if one instr slot is *not* used?
  - What does the register file have to support?
  - What other hardware must we add?

An Example: VLIW MIPS



- Consider a 2-issue MIPS with a 2-instr bundle
- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair cannot be used, it is replaced with a "no op" (nop)
  - Need 4 read ports and 2 write ports and a separate memory address adder

A MIPS VLIW (2-issue) Datapath



No hazard hardware (so load-use needs explicit nop instruction(s) inserted by compiler)

Let's say we wanted more functional units. What would need to change?

Code Scheduling Example

```
i = n;
do {
  a[i] += const;
} while (--i != 0);
```

- Consider the following loop code:

```
lp: lw     $t0,0($s1)   # $t0=array element
    addu   $t0,$t0,$s2  # add scalar in $s2
    sw     $t0,0($s1)   # store result
    addi   $s1,$s1,-4   # decrement pointer
    bne    $s1,$0,lp    # branch if $s1 != 0
```

- Must "schedule" the instructions to avoid pipeline stalls
  - Instructions in one bundle must be independent
  - Must separate load-use instructions from their loads by one cycle
  - Notice that the first two instructions have a load-use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:			1
			2
			3
			4
			5

- How many clock cycles?
- How many instructions?
- CPI? Best case?
- IPC? Best case?

Loop Unrolling

- Loop unrolling—multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP
- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load-use hazards
- During unrolling, the compiler applies register renaming to eliminate all data dependencies that are *not* true dependencies

Unrolled Code Example

```
lp: lw   $t0,0($s1)   # $t0=array element
    lw   $t1,-4($s1)  # $t1=array element
    lw   $t2,-8($s1)  # $t2=array element
    lw   $t3,-12($s1) # $t3=array element
    addu $t0,$t0,$s2  # add scalar in $s2
    addu $t1,$t1,$s2  # add scalar in $s2
    addu $t2,$t2,$s2  # add scalar in $s2
    addu $t3,$t3,$s2  # add scalar in $s2
    sw   $t0,0($s1)   # store result
    sw   $t1,-4($s1)  # store result
    sw   $t2,-8($s1)  # store result
    sw   $t3,-12($s1) # store result
    addi $s1,$s1,-16  # decrement pointer
    bne  $s1,$0,lp    # branch if $s1 != 0
```

Scheduled Code (Unrolled)

	ALU or branch	Data transfer	CC
lp:			1
			2
			3
			4
			5
			6
			7
			8

What does N=14 assembly look like?

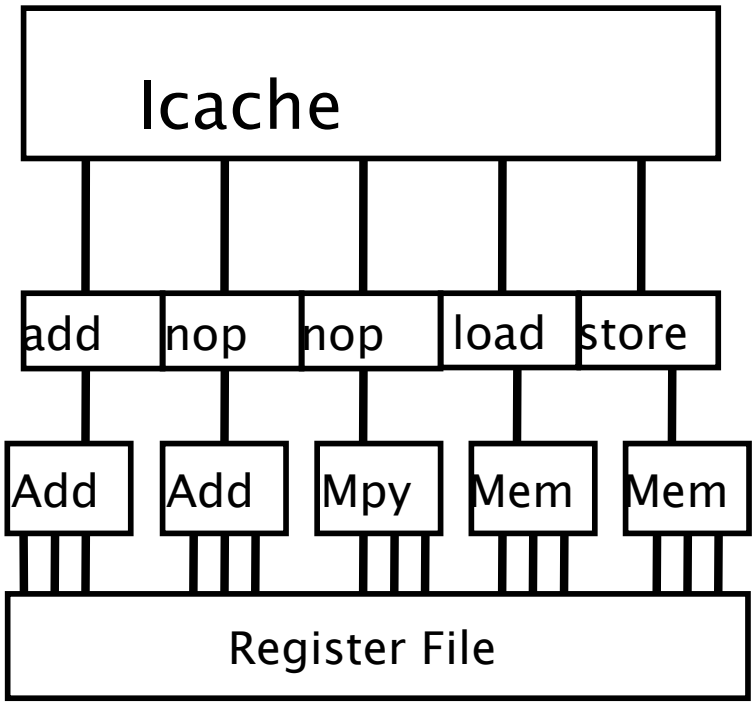
- Two instructions from a scientific benchmark (Linpack) for a MultiFlow CPU with 14 operations per instruction.

```
instr   cl0     ialu0e    st.64          sb1.r0,r2,17#144
        cl0     ialu1e    cgt.s32        li1bb.r4,r34,6#31
        cl0     falu0e    add.f64        lsb.r4,r8,r0
        cl0     falu1e    add.f64        lsb.r6,r40,r32
        cl0     ialu0l    dld.64         fb1.r4,r2,17#208
        cl1     ialu0e    dld.64         fb1.r34,r1,17#216
        cl1     ialu1e    cgt.s32        li1bb.r3,r32,zero
        cl1     falu0e    add.f64        lsb.r4,r8,r6
        cl1     falu1e    add.f64        lsb.r6,r40,r38
        cl1     ialu0l    st.64          sb1.r2,r1,17#152
        cl1     ialu1l    add.u32        lib.r32,r36,6#32
        cl1     br        true and r3    L23?3
        cl0     br        false or r4    L24?3;

instr   cl0     ialu0e    dld.64         fb0.r0,r2,17#224
        cl0     ialu1e    cgt.s32        li1bb.r3,r34,6#30
        cl0     falu0e    mpy.f64        lfb.r10,r2,r10
        cl0     falu1e    mpy.f64        lfb.r42,r34,r42
        cl0     ialu0l    st.64          sb0.r4,r2,17#160
        cl1     ialu0e    dld.64         fb0.r32,r1,17#232
        cl1     ialu1e    cgt.s32        li1bb.r4,r35,6#29
        cl1     falu0e    mpy.f64        lfb.r10,r0,r10
        cl1     falu1e    mpy.f64        lfb.r42,r32,r42
        cl1     ialu0l    st.64          sb0.r6,r1,17#168
        cl1     ialu1l    bor.32         ib0.r32,zero,r32
        cl1     br        false or r4    L25?3
        cl0     br        true and r3    L26?3;
```

Defining Attributes of VLIW

- Compiler
  1. MultiOp: instruction containing multiple independent operations
  2. Specified number of resources of specified types
  3. Exposed, architectural latencies



VLIW instruction = 5 independent operations

Compiler Support for VLIW Processors

1. The compiler packs groups of independent instructions into the bundle
   - Because branch prediction is not perfect, done by code re-ordering (trace scheduling)
2. The compiler uses loop unrolling to expose more ILP
3. The compiler uses register renaming to solve name dependencies and ensures no-load use hazards occur

Compiler Support for VLIW Processors

- While superscalars use dynamic prediction, VLIWs primarily depend on the compiler for extracting ILP
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
    - We'll cover this in a future lecture as well
- The compiler predicts memory bank references to help minimize memory bank conflicts

VLIW Advantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instructions per VLIW bundle and add more FUs

VLIW Disadvantages

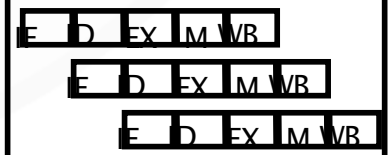
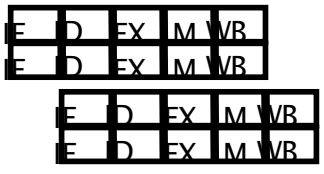
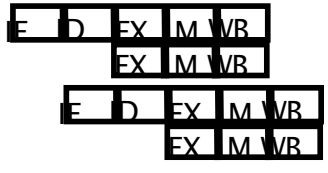
- Programmer/compiler complexity and longer compilation times
  - Deep pipelines and long latencies can be confusing (making peak performance elusive)
- Lock-step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
- Object (binary) code incompatibility
- Needs lots of program memory bandwidth
- Code bloat
  - "No ops" are a waste of program memory space
  - Loop unrolling to expose more ILP uses more program memory space

Review:

Multiple-Issue Processor Styles

- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8 in 2005) are being made dynamically (at run time by the hardware)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500 IBM
- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128-bit "bundles" containing 3 instructions each 41 bits + 5 bit template field (specifies which FU each instr needs)
    - Five functional units (IntALU, MMedia, DMem, FPALU, Branch)
    - Extensive support for speculation and predication

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special	many GP	GP and rename	many, many GP
Memory reference	embedded in many instrs	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware dependency resolution	(compiler) code scheduling
Instruction flow				

Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium