

# Chapter 4D

## The Processor

# Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage  $\Rightarrow$  shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages  $\Rightarrow$  multiple pipelines
    - Start multiple instructions per clock cycle
    - $CPI < 1$ , so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak  $CPI = 0.25$ , peak  $IPC = 4$
    - But dependencies substantially reduce this.

# Multiple Issue

- **Static** multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- **Dynamic** multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

# Speculation

- An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions
- Example
  - Whether a branch is taken or not
  - Whether a store that proceeds a load does not refer to the same address.
- **Must** include two methods
  - A method to check if the guess was right
  - A method to unroll or back out the effects of the instructions that were executed speculatively.

# Speculation

- Can be done in the compiler or by the hardware.
- Moving an instruction across a branch or a load across a store.
- Recovery mechanism in software
  - Insert an additional routine to check the accuracy of speculation
  - Provide a fix-up routine to use when the speculation is incorrect
- Recovery mechanism in hardware
  - Buffers the speculative results until it knows they are no longer speculative.
  - If the speculation is correct, allow them to be written into the registers or memory
  - If the speculation is incorrect, flushes the buffers and re-executes the correct instruction sequence.

# Multi-issue Taxonomy

Common Name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscaler (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/III
Superscaler (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	IBM Power2
Sperscaler (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium III/4 MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW/LIW	static	software	static	no hazards between issue packets	Trimedia, i860
EPIC	mostly static	mostly software	mostly static	explicit dependencies marked by compiler	Itanium (IA-64 is one implementation)

# Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

# Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

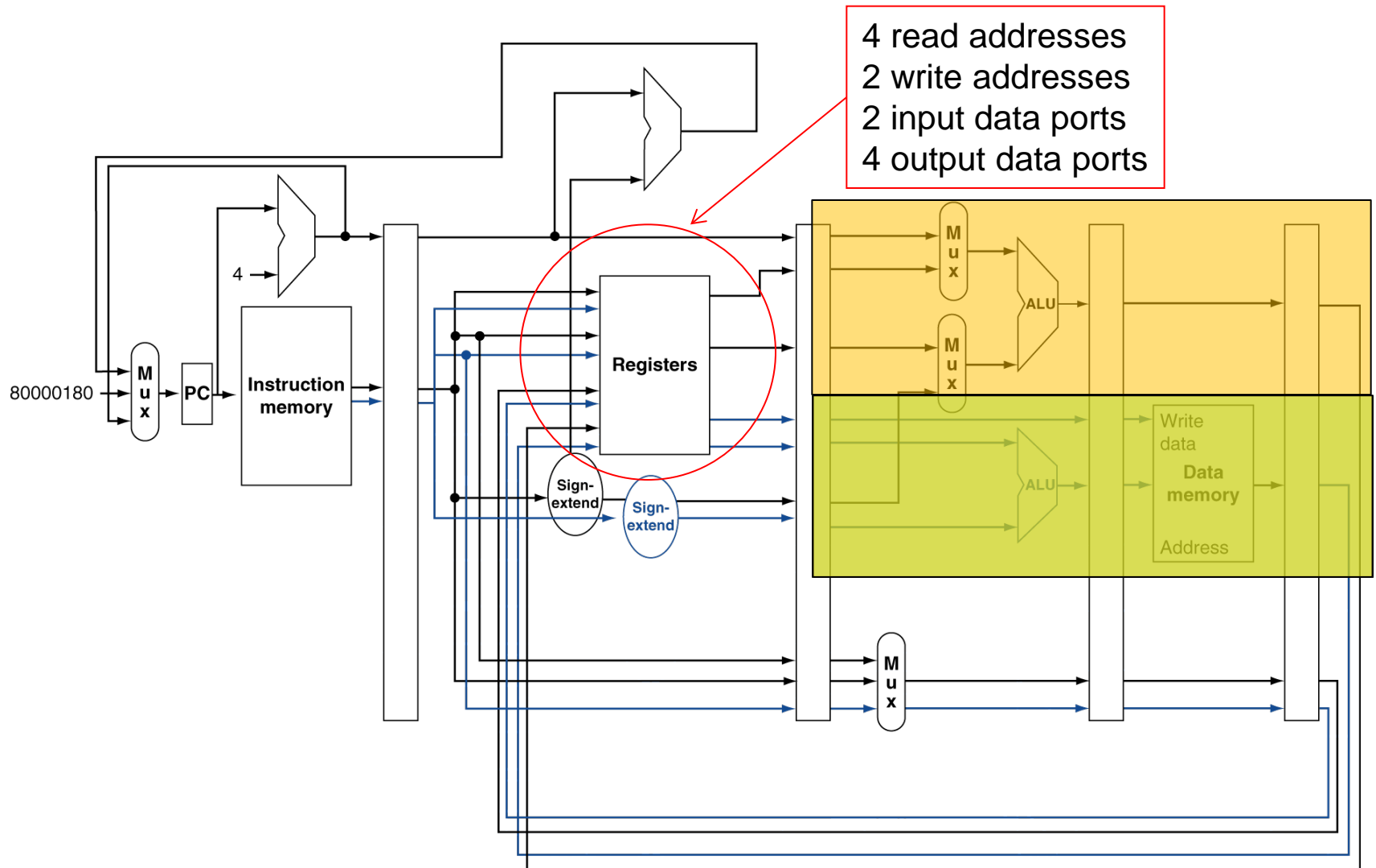


# MIPS with Static Dual Issue

- **Two-issue** packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
<b>n</b>	ALU/branch	IF	ID	EX	MEM	WB		
<b>n + 4</b>	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

# MIPS with Static Dual Issue



# Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Avoided stalls with forwarding in single issue
  - Now can't use ALU result in load/store in same packet
    - add  $\$t0$ ,  $\$s0$ ,  $\$s1$   
load  $\$s2$ ,  $0(\$t0)$
    - Split into two packets: effectively a stall
- Load-use hazard
  - Still one-cycle use latency, which means the next two instructions cannot use the load result
- Even for ALU instruction
  - One-instruction use latency, which means its result cannot be used in the paired load or store
- More aggressive scheduling required

# Scheduling Example

- Schedule this for dual-issue MIPS

```

Loop: 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, $zero, Loop # branch $s1!=0
  
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

# Loop Unrolling

- A technique to get more performance from loops that access arrays.
- Multiple copies of the loop body are scheduled together.
  - Reduces loop-control overhead
- Use different registers per replication (**register renaming**) to avoid loop-carried anti-dependencies (name dependencies)
  - Store followed by a load of the same register
  - **Avoid WAR and WAW hazards**

# Loop Unrolling Example

- Unrolling 4 times
  - Name dependence or anti dependence

Loop:	lw \$t0, 0(\$s1)	# \$t0=array[0]
	addu \$t0, \$t0, \$s2	# add scalar in \$s2
	sw \$t0, 0(\$s1)	# store result
	lw \$t0, 4(\$s1)	# \$t0=array[1]
	addu \$t0, \$t0, \$s2	# add scalar in \$s2
	sw \$t0, 4(\$s1)	# store result
	lw \$t0, 8(\$s1)	# \$t0=array[2]
	addu \$t0, \$t0, \$s2	# add scalar in \$s2
	sw \$t0, 8(\$s1)	# store result
	lw \$t0, 12(\$s1)	# \$t0=array[3]
	addu \$t0, \$t0, \$s2	# add scalar in \$s2
	sw \$t0, 12(\$s1)	# store result
	addi \$s1, \$s1, -16	# decrement pointer
	bne \$s1, \$zero, Loop	# branch \$s1!=0

# Loop Unrolling Example

- Unrolling 4 times

- Register renaming:  $t0 \rightarrow t1, t2, t3$

```
Loop: lw    $t0, 0($s1)      # $t0=array[0]
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)     # store result
      lw    $t1, 4($s1)     # $t1=array[1]
      addu  $t1, $t1, $s2    # add scalar in $s2
      sw    $t1, 4($s1)     # store result
      lw    $t2, 8($s1)     # $t2=array[2]
      addu  $t2, $t2, $s2    # add scalar in $s2
      sw    $t2, 8($s1)     # store result
      lw    $t3, 12($s1)    # $t3=array[3]
      addu  $t3, $t3, $s2    # add scalar in $s2
      sw    $t3, 12($s1)    # store result
      addi  $s1, $s1, -16    # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

# Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)	1
	nop	lw \$t1, 12(\$s1)	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1)	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1)	5
	addu \$t3, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

- $IPC = 14/8 = 1.75$ 
  - Closer to 2, but at cost of registers and code size



# Statically Scheduled Superscaler

Common Name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscaler (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/III

- Instructions are issued in order:
- All hazards checked dynamically at issue time
- Variable number of instructions issued per clock cycle: one , two, or more
- Require the compiler techniques to be efficient.

# Superscaler vs VLIW

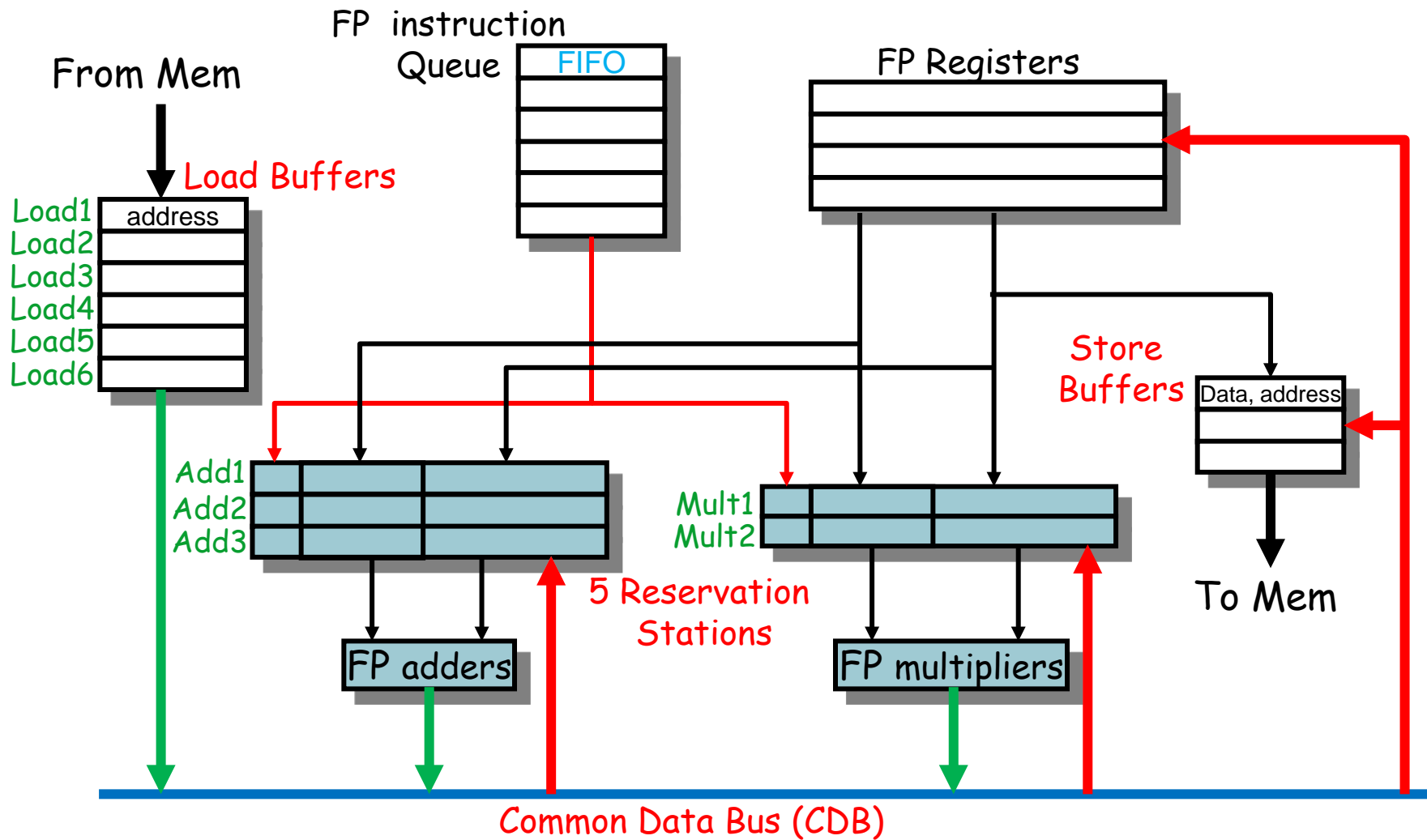
- superscaler
  - The code is guaranteed by the hardware to execute correctly, independently of the issue rate or pipeline structure of the processor
- VLIW
  - In **some** VLIW designs, recompilation is required
  - In other VLIW designs, the code would run in different implementations, but often so poorly as to make compilation effectively required.

# Dynamically Scheduled Superscaler

Common Name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscaler (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	IBM Power2

- Dynamic scheduling does not restrict the types of instructions that can be issued on a single clock cycle.
- Think of it as **Tomasulo's algorithm** extended to support multiple-issue
- Allows N instructions to be issued whenever reservation stations are available.
- Branch prediction is used for fetch and issue (but not execute)

# Tomasulo Organization



# Tomasulo Algorithm

- Control & buffers for each function unit (FU)
  - FU buffers called a “**reservation station**” (RS);
  - A RS for a FU
  - Instructions in RS have pending operands
- Registers in instructions replaced by values or **pointers to reservation stations or load buffers if pending**;
  - called **register renaming** (avoids WAR, WAW hazards)
  - 4-bit tag: 5 reservation stations and 6 load buffers
- Results to FU from RS, **not through registers**, over **Common Data Bus (CDB)** that broadcasts results to all FUs
- **Load and Stores treated as FUs** with RSs as well
- Integer instructions can go across branches, allowing FP ops beyond a basic block in FP queue

# Reservation Station Components

**Op**: Operation to perform in the unit (e.g., + or –)

**Qj, Qk**: Reservation station IDs producing source registers (value to be written)

- Note: No ready flags as in Scoreboard;  $Q_j, Q_k=0 \Rightarrow$  ready
- Store buffers only have  $Q_i$  for RS producing result

**Vj, Vk**: Value of Source operands

- Store buffers has V field, result to be stored
- Either V or Q file is valid for each operand

**A**: used to hold an address for a load or store

**Busy**: Indicates reservation station or FU is busy

**Register result status**—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

# Tomasulo Algorithm

- 1. Issue**—(dispatch) get instruction from FP Op Queue  
If an RS entry is free (no structural hazard), issues an instruction & reads its operands if available. (register renaming: avoid WAR and WAW hazards)
  - 2. Execution**—(issue) operate on operands (EX)  
When both operands ready, then execute; if not ready, watch Common Data Bus, waiting for result (avoid RAW hazards)
  - 3. Write result**—finish execution (WB)  
Write on Common Data Bus to all awaiting RS units, FP registers, and store buffers; mark its reservation station available
- **Normal data bus**: data + destination (“go to” bus)
  - **Common data bus**: data + source (“come from” bus)
    - 64 bits of data + 4-bit tag of Functional Unit source address
    - Does the broadcast
    - Write if tags matches (expected FU produces result)

# Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- **Not all stalls are predicable**
  - e.g., cache misses
- Can't always schedule around branches
  - **When branch outcome is dynamically determined**
- Different implementations of an ISA have different latencies and hazards



# Dynamic Pipeline Scheduling

- Allow the CPU to
  - execute instructions **out of order** to avoid stalls
  - But commit result to registers **in order**

- Example

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

- Can start **sub** while **addu** is waiting for **lw**, which might take many clock cycles (**when lw causes a data cache miss**)
- Dynamic scheduling allows such hazards to be avoided either fully or partially

# Dynamic Scheduling: Issue



- Simple pipeline had only one stage to check both structural and data hazards: Instruction Decode (ID), also called **Instruction Issue**
- Dynamic scheduling HW splits the ID pipe stage into 2 stages:
  - *Issue:*
    - Decode instructions, check for structural hazards
  - *Read operands:*
    - Wait until no data hazards, then read operands

# HW Schemes to Schedule Instructions

- Key idea: Allow **instructions** behind stall to proceed

DIVD    **F0** , F2 , F4

ADDD    F10 , **F0** , F8

SUBD    F12 , F8 , F14

- Enables **out-of-order execution** and allows **out-of-order completion** (e.g., schedule **SUBD** before **slow DIVD**)
  - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (**in-order issue**)
- Will distinguish
  - when an instruction *begins execution* from
  - when it *completes execution*;
  - between the two times, the instruction is *in execution*

# Advantages of Dynamic Scheduling

- **Dynamic scheduling** - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
- It handles cases in which dependences were unknown at compile time
  - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
- It allows code compiled for one pipeline to run efficiently on a different pipeline
- It simplifies the compiler
- **Hardware speculation**, a technique with significant performance advantages, builds on dynamic scheduling

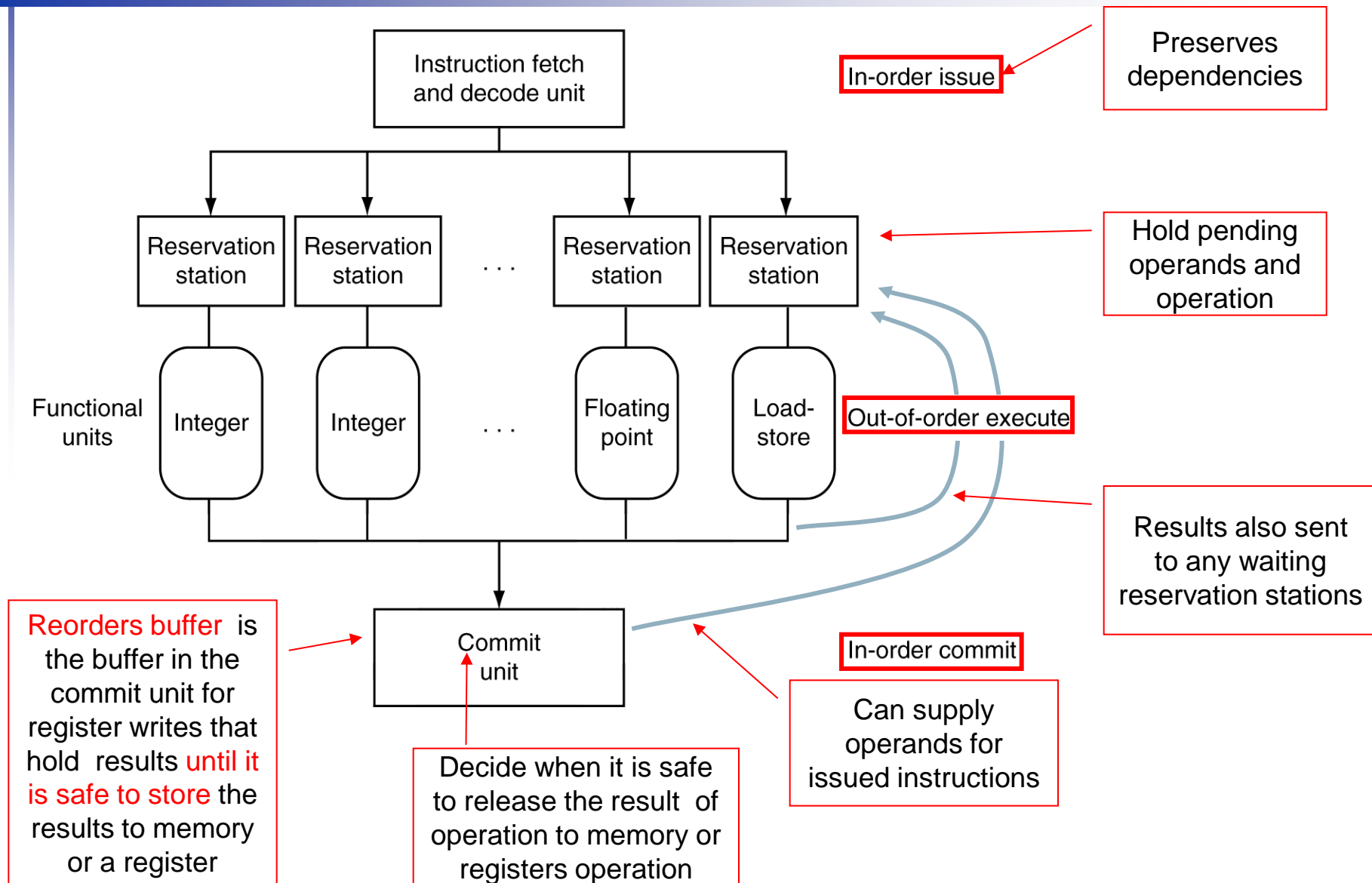
# Speculative Superscaler

Common Name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscaler (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium III/4 MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III

## Hardware-based Speculation

- **Dynamic branch prediction** to choose which instructions to execute
- **Speculation** to allow execution of instructions before control dependencies are resolved
- **Dynamic scheduling** to deal with scheduling of different combinations of **basic** blocks

# Dynamically Scheduled CPU



# Speculation


- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- **Speculate on branch** outcome: move
  - Move an instruction across an branch
  - Roll back if path taken is different
- **Speculate on load**
  - Move a load across a store
  - Roll back if the location is updated by the store

# Memory Data Dependences

- Besides branches, long memory latencies are one of the biggest performance challenges today.
- To **preserve sequential** (in-order) state in the **data caches** and **external memory** (so that recovery from exceptions is possible) **stores are performed in order**. This takes care of antidependences (**WAR**) and output dependences (**WAW**) to memory locations.
- However, **loads can be issued out of order** with respect to stores if the out-of-order loads check for data dependences with respect to previous, pending stores.


WAW

store X  
:  
store X




WAR

load X  
:  
store X



RAW

store X  
:  
load X





# Memory Dependencies

```
st r1, (r2)
ld r3, (r4)
```

When can we execute the load?

Does the load move across the store?

# Memory Data Dependences

## ■ Memory Aliasing:

- Two memory references involving the same memory location (collision of two memory addresses).

## ■ Memory Disambiguation:

- Determining whether two memory references will alias or not (whether there is a dependence or not).

## ■ Memory Dependency Detection:

- Must **compute effective addresses** of both memory references
- Effective addresses can depend on run-time data and other instructions

# Conservative OOO Load Execution

```
st r1, (r2)
```

```
ld r3, (r4)
```

- Split execution of store instruction into two phases:\ul>  - address calculation and
  - data write
- (load bypassing) Can execute load before store, **if addresses known and  $r4 \neq r2$**
- Each load address compared with addresses of **all previous uncommitted stores**
  - *can use partial conservative check i.e., bottom 12 bits of address*
- Don't execute load **if any previous store address not known**
  - *MIPS R10K, 16 entry address queue*

# Address Speculation

```
st r1, (r2)
ld r3, (r4)
```

- Assume that  $r4 \neq r2$
- Execute load before store address known
- Need to hold all **completed but uncommitted** load/store addresses in program order
- If subsequently find  $r4 == r2$ , squash load and *all* following instructions
  - ⇒ **Large penalty** for inaccurate address speculation

# Memory Dependence Prediction

```
st r1, (r2)
ld r3, (r4)
```

- Guess that  $r4 \neq r2$  and execute load before store
- If later find  $r4 == r2$ , squash load and all following instructions, but **mark load instruction** as *store-wait* for future executions
- **Subsequent** executions of the same load instruction will wait for all previous stores to complete
- **Periodically** clear *store-wait* bits

# Compiler/Hardware Speculation

- SW: Compiler can reorder instructions
  - Also include a “fix-up” routine to recover from incorrect guess
- HW: can look ahead for instructions to execute
  - Buffer speculative results until it knows the speculation is correct
  - Allow the buffer contents to be written into the register or memory if the speculation is correct,
  - Flush the buffers on incorrect speculation

# Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
  - A speculated instruction should not cause an exception: **unnecessary negative performance effects**
- For static speculation (compiler)
  - Can add ISA support for deferring exceptions
- For dynamic speculation (hardware)
  - Can **buffer exceptions** until instruction is no more speculative

# HW Support for Compiler Speculation

- Three capabilities are required for speculation
  - (compiler) Ability to find instructions that can be speculatively moved.
  - (Hw) Ability to ignore exceptions in speculated instructions
  - (Hw) Ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts.
- Two types of exceptions
  - Termination: memory protection violation: **ignore**
  - Resumption: page faults: **acceptable**



# For Preserving Exception Behavior

- There are four methods

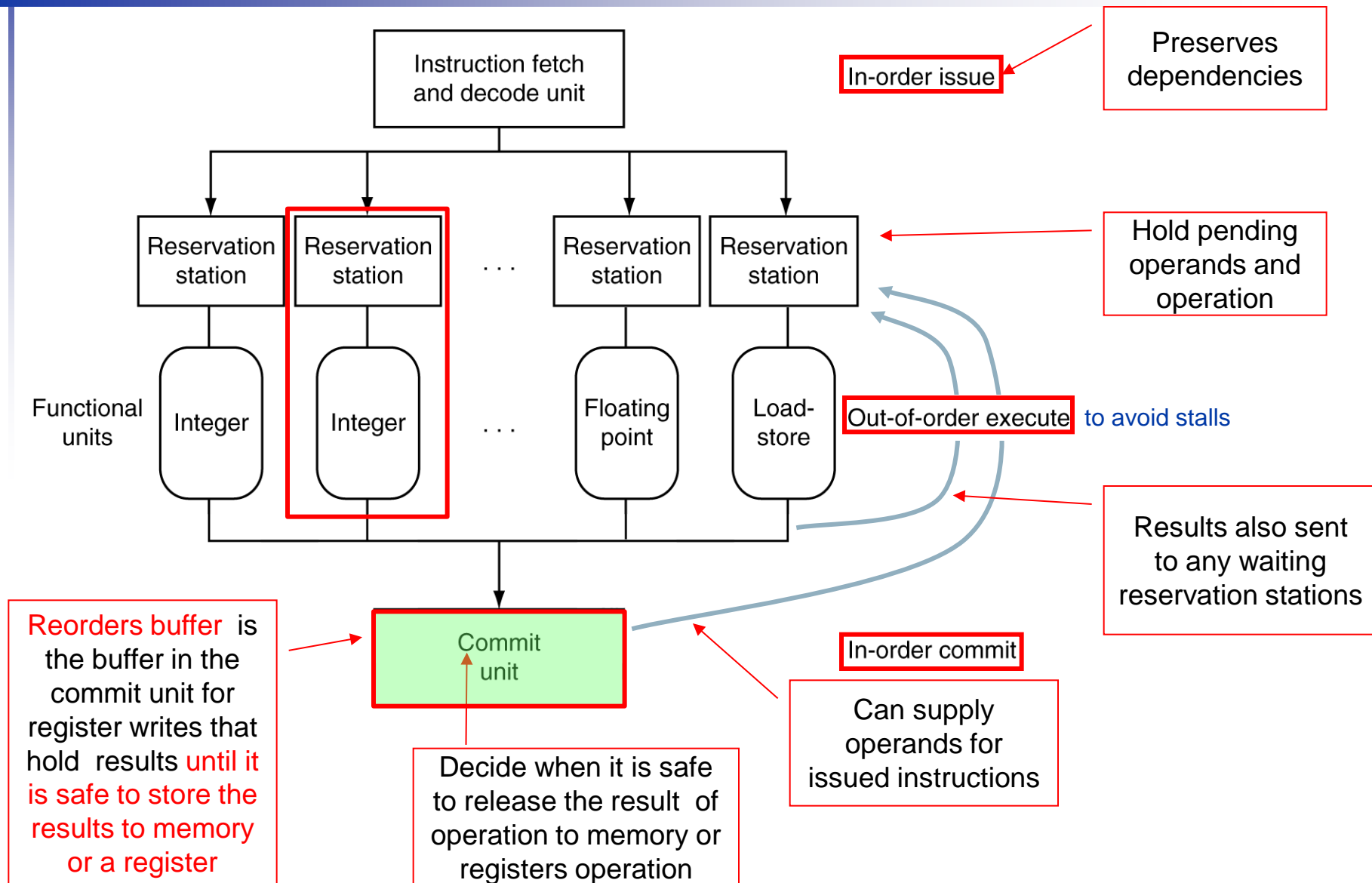
1. The hardware and operating system cooperatively **ignore (terminating) exceptions** for speculative instructions. As we will see later, this approach preserves exception behavior only for correct programs, but not for incorrect ones. This approach may be viewed as unacceptable for some programs, but it has been used, under program control, as a “fast mode” in several processors.

2. Speculative instructions that **never raise (terminating) exceptions** are used, and checks are introduced to determine when a (terminating) exception should occur.

3. A set of status bits, called *poison bits*, are attached to the result registers written by speculated instructions when the instructions cause (terminating) exceptions. The poison bits **cause a fault** when a **normal instruction attempts to use the register**.

4. A mechanism is provided to indicate that an instruction is speculative, and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

# Dynamically Scheduled CPU



# Adding Speculation to Tomasulo

- Must separate **execution** from allowing instruction to finish or “**commit**”
- This additional step is called **instruction commit**; it occurs
  - whenever the branch prediction is confirmed **for the branch immediately before a block** of speculated instructions.
- When an instruction is no longer speculative, allow it to update the register file or memory.
- Requires an 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**.

# Reorder Buffer (ROB)

- In Tomasulo's algorithm, **once an instruction writes its result**, any subsequently issued instructions will find result in the register file
- With speculation, **the register file is not updated until the instruction commits**
  - when know for sure that the instruction should have executed
- The ROB **supplies** operands in the interval between **end of instruction execution** and **instruction commit**
  - ROB is a source of operands for instructions, just as **reservation stations** (RS) provide operands in Tomasulo's algorithm
  - ROB extends architecture registers as the reservation stations did

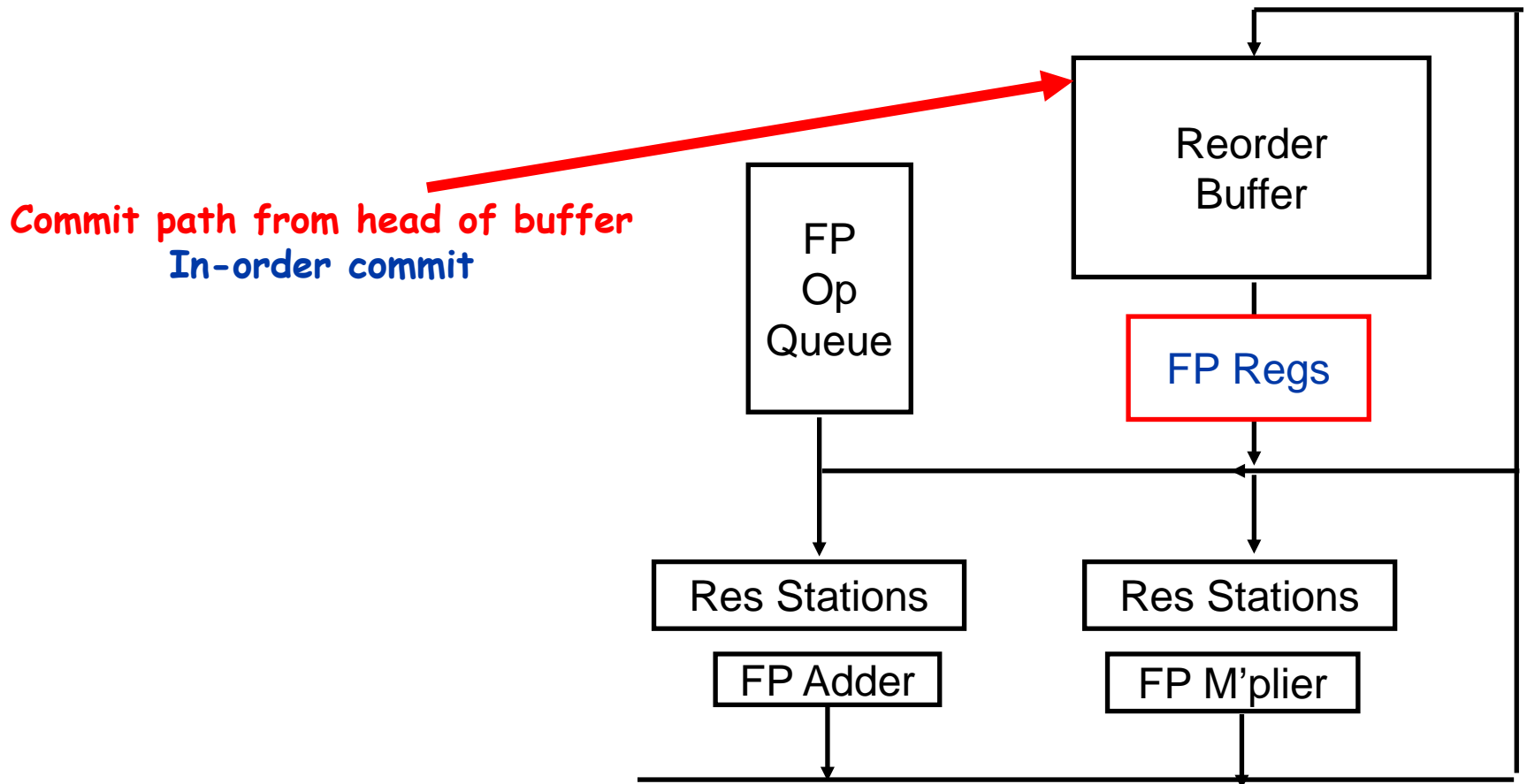
# Reorder Buffer Entry Fields

- Each entry in the ROB contains four fields:
  1. **Instruction type**
    - A branch (has yet no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, each of which has a register destination for writeback)
  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 being held until the instruction commits
  4. **Ready**
    - Indicates that instruction has completed execution, and the value is ready once the instruction commits

# 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  $\Rightarrow$  more registers like RSs (reservation stations)
  - Tag results with ROB buffer number instead of reservation station number
- Instructions **commit**  $\Rightarrow$  values **at head of ROB** placed in registers
- **As a result**, **easy to undo** speculated instructions on mispredicted branches or on **exceptions**

# Reorder Buffer Operation



# Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue  
If reservation station free (no structural hazard),  
control issues instr & sends operands (renames registers).
2. **Execution**—operate on operands (EX)  
When both operands ready then execute;  
if not ready, watch Common Data Bus for result
3. **Write result**—finish execution (WB)  
Write on Common Data Bus to all awaiting units;  
mark reservation station available



# Speculative Tomasulo Algorithm

- 1. Issue** (dispatch) —get instruction from FP Op Queue  
If reservation station **and** **reorder buffer slot** free, issue instr & send operands **& reorder buffer index for destination**
- 2. Execution** (issue)—operate on operands (EX)  
Checks for RAW hazards; when both operands ready then execute; if not ready, watch Common Data Bus for result; when both in reservation station, execute.
- 3. Write result**—finish execution (WB)  
Write on Common Data Bus to all awaiting RSs **& reorder buffer**; mark reservation station available.
- 4. Commit** (graduation) —**update register with reorder result**  
**When instr. at head of reorder buffer & result are present and the branch before it has been confirmed, update register with result (or store to memory) and remove instr from reorder buffer.**  
**Mispredicted branch flushes reorder buffer above (executed after) the branch**

# Does Multiple Issue Work?

## The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

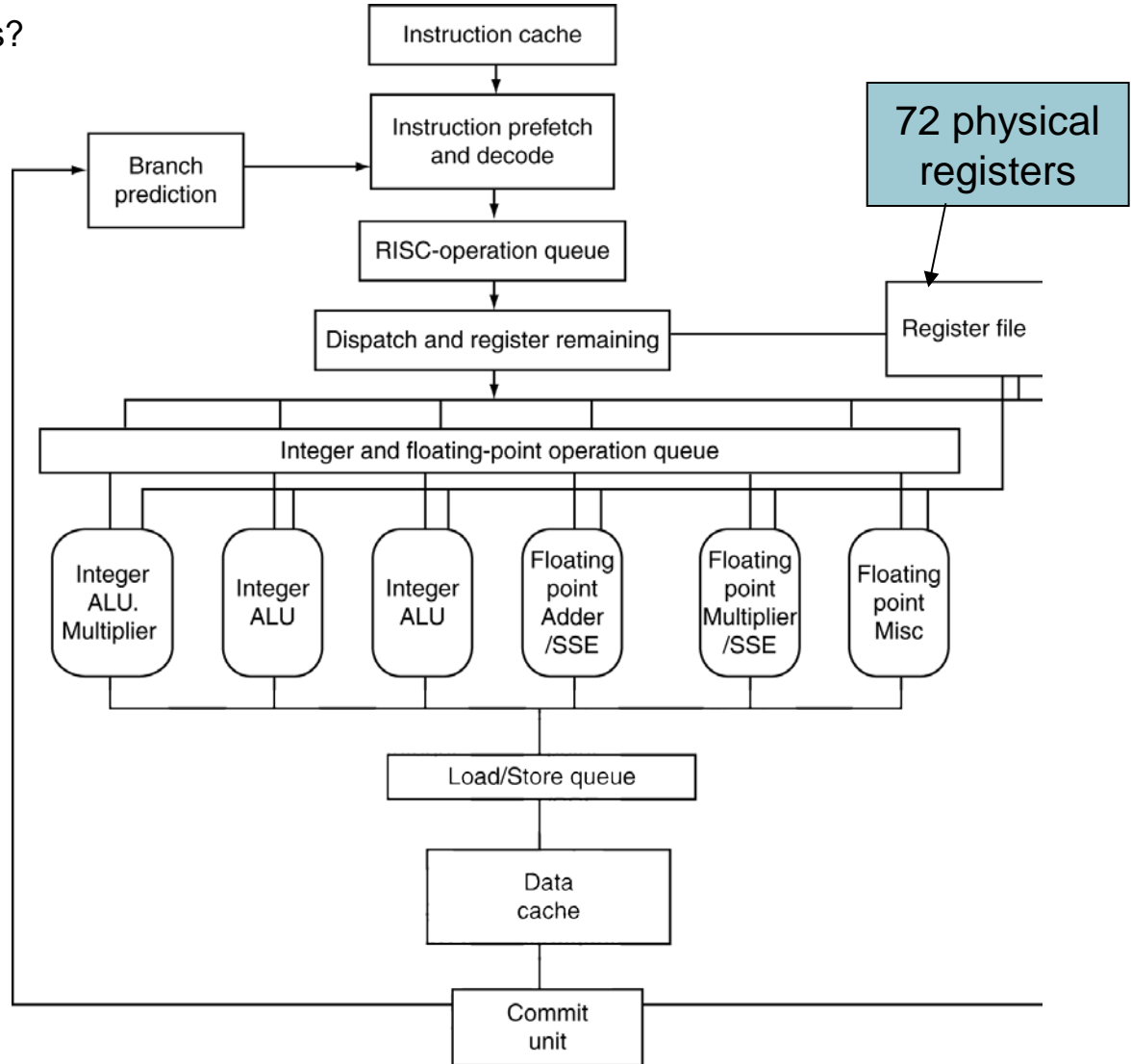
# Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

# The Opteron X4 Microarchitecture

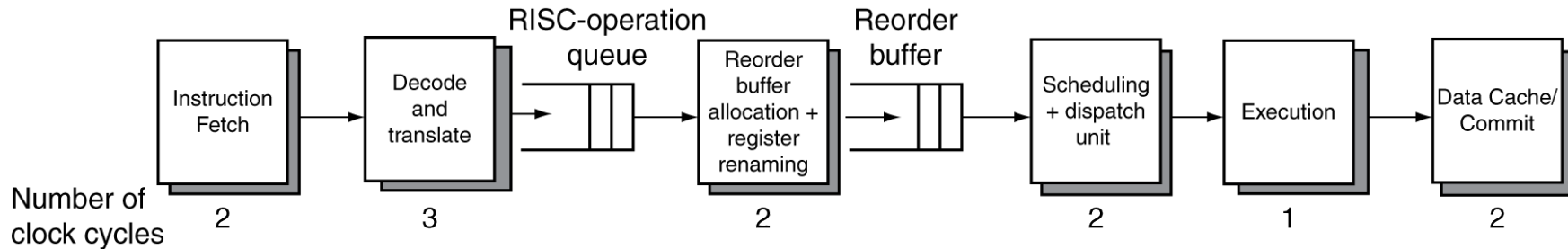
What is architectural registers?  
16 Visible register in ISA



Register renaming requires to the processor to maintain a map between the architectural registers and the physical registers, indicating that which physical register is the Most current copy of an architectural register

# The Opteron X4 Pipeline Flow

- For integer operations



- FP is 5 stages longer
- Up to 106 RISC-ops in progress
- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays

# Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

# Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

# Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall



# Summary

- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
  - Works when cannot know dependences at compile time
  - Can hide L1 cache misses
  - Code for one pipelined machine runs well on another

# Homework: chapter 4

- Due before starting the midterm on Oct. 27.
- Exercise 4.14
- Exercise 4.22
- Exercise 4.29
- Exercise 4.35