Computer Architecture and Operating Systems
Lecture 10: Exceptions. Multiple Issue.

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

2

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at $0000\ 0000\ 1C09\ 0000_{hex}$

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Undefined opcode $00\ 0100\ 0000_{two}$
  - Hardware malfunction: $01\ 1000\ 0000_{two}$
  - …:   …
- Instructions either
  - Deal with the interrupt, or
 - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - Use SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …

Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage
  `add x1, x2, x1`
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware
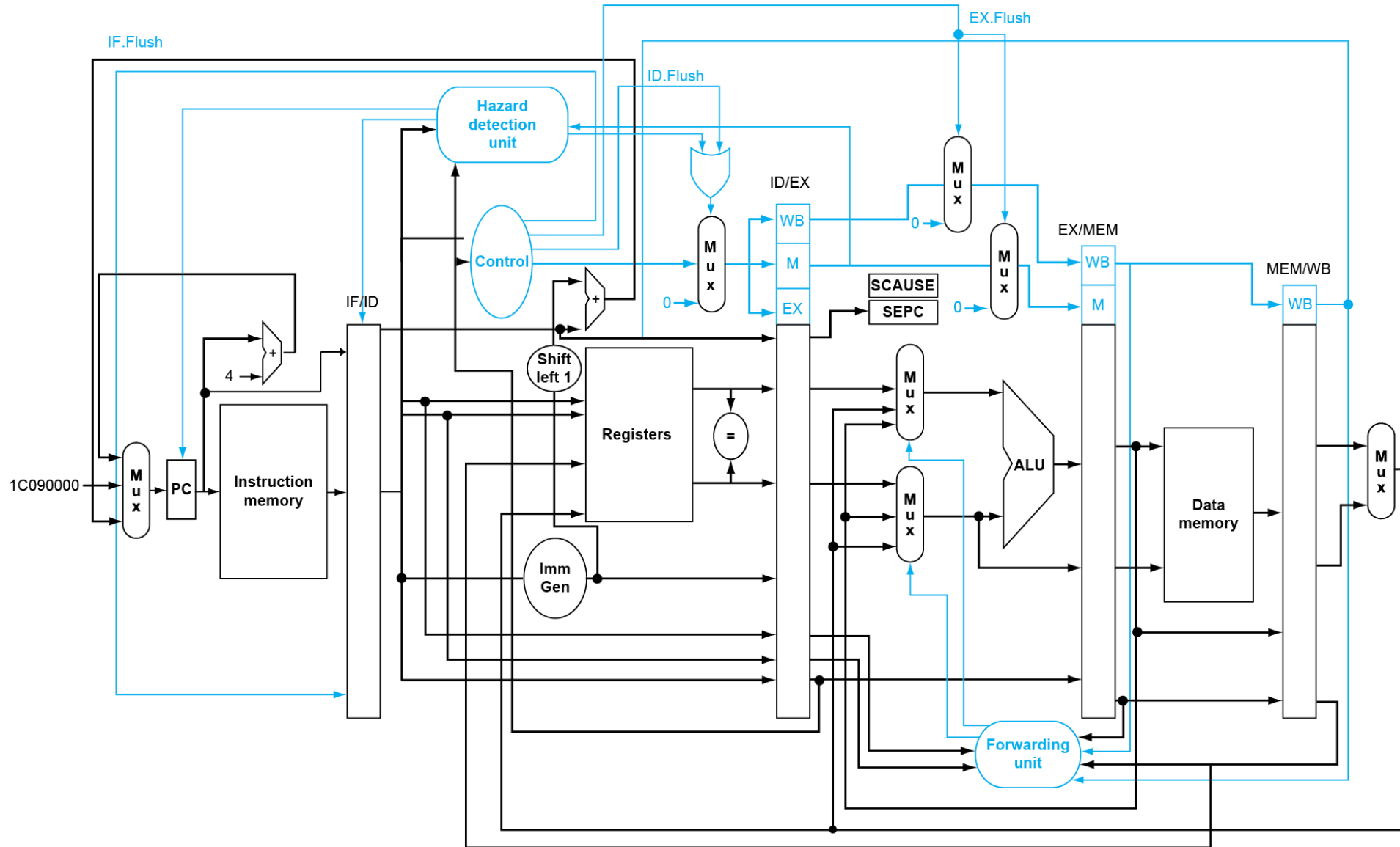
Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch

- PC saved in SEPC register
  - Identifies causing instruction

Exception Example

- Exception on **add** in

```
40    sub   x11, x2, x4
44    and   x12, x2, x5
48    orr   x13, x2, x6
4c    add   x1,  x2, x1
50    sub   x15, x6, x7
54    ld    x16, 100(x7)
…
```
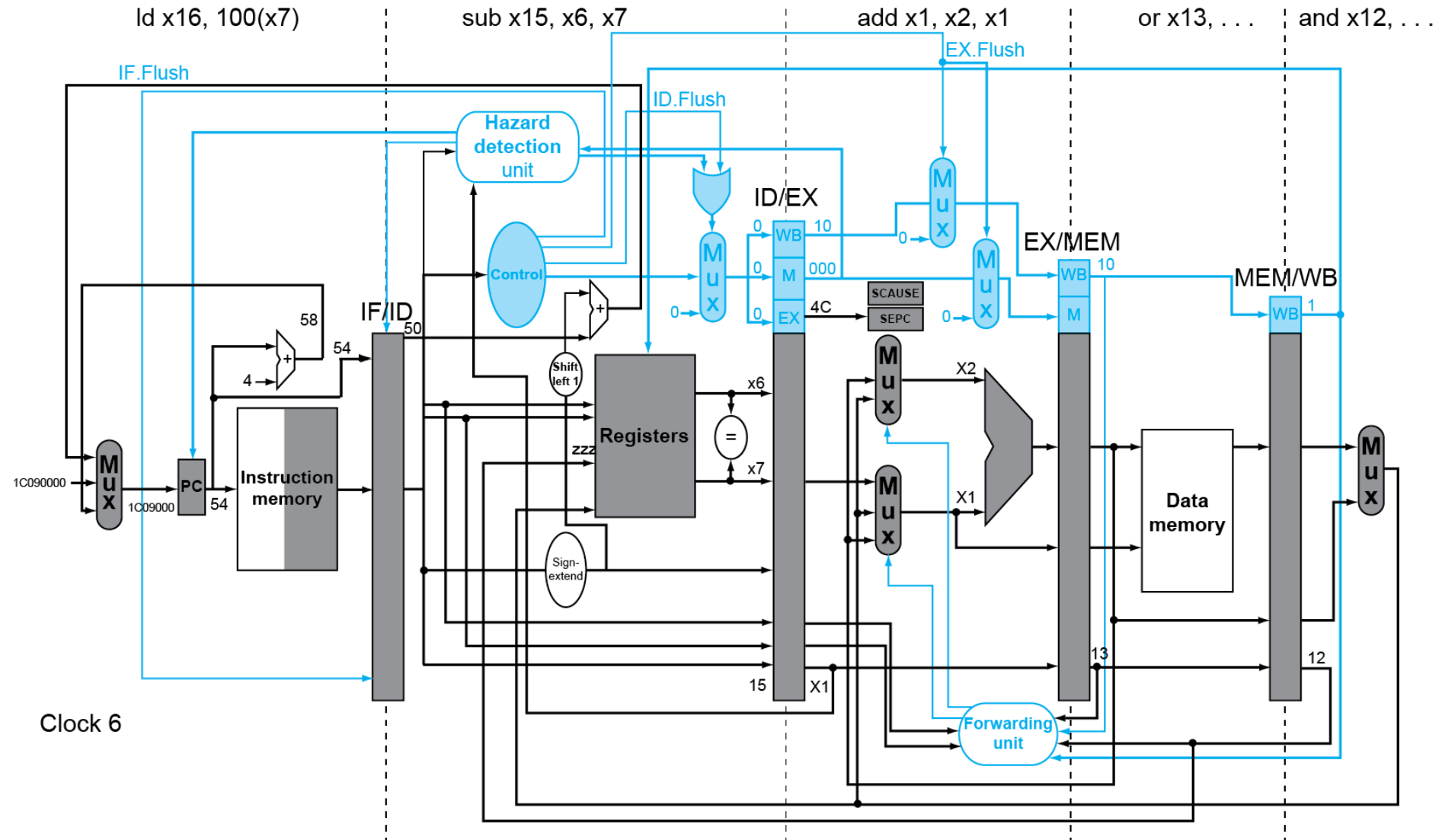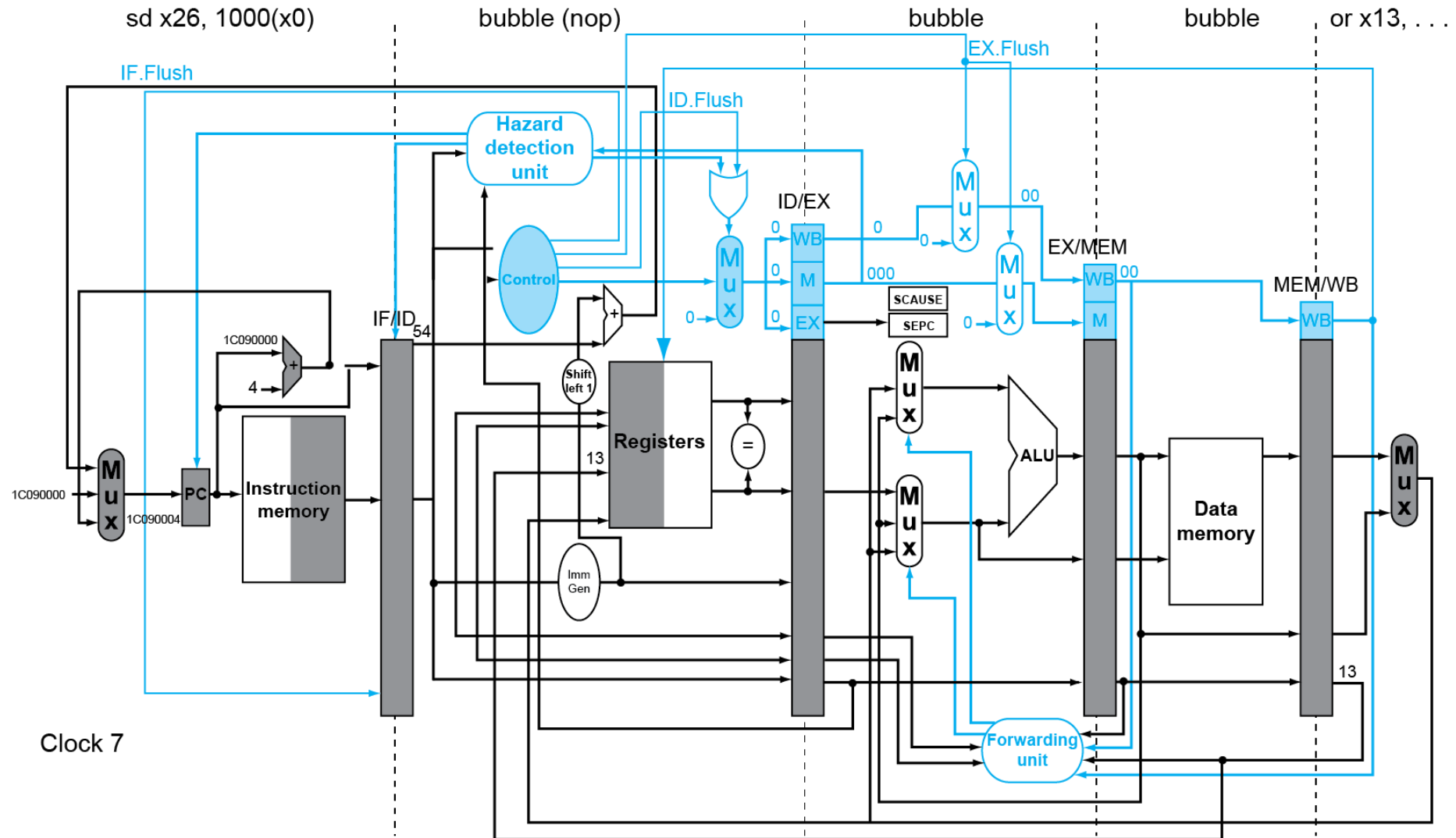
- Handler

```
1C090000 sd x26, 1000(x10)
1c090004 sd x27, 1008(x10)
…
```

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once

- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions

- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require "manual" completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

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 reduce this in practice

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

- "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
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions

  - e.g., move load before branch

  - Can include "fix-up" instructions to recover from incorrect guess

- Hardware can look ahead for instructions to execute

  - Buffer results until it determines they are actually needed

  - Flush buffers on incorrect speculation

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
  - $\Rightarrow$ 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

RISC-V 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						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	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

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
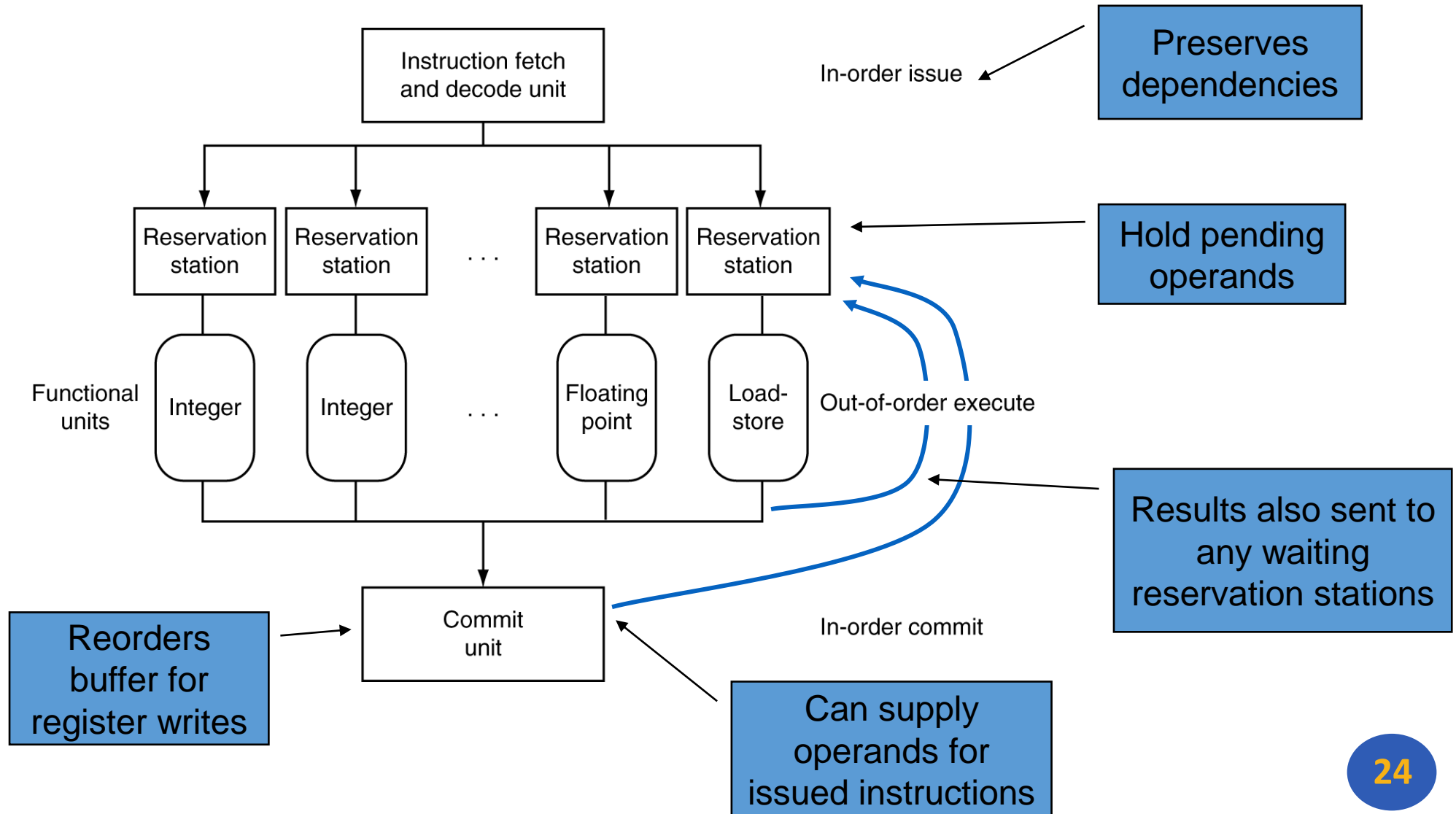   - But commit result to registers in order
- Example
```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```
    - Can start sub while add is waiting for ld

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
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Dynamically Scheduled CPU

Instruction fetch and decode unit

In-order issue

Preserves dependencies

Reservation station — Reservation station — . . . — Reservation station — Reservation station

Hold pending operands

Functional units — Integer — Integer — . . . — Floating point — Load-store

Out-of-order execute

Results also sent to any waiting reservation stations

Commit unit

Reorders buffer for register writes

In-order commit

Can supply operands for issued instructions

24

Does Multiple Issue Work?

- 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

Conclusion

- 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

Any Questions?

```
                .text
        __start:    addi t1, zero, 0x18
                    addi t2, zero, 0x21
    cycle:          beq t1, t2, done
                    slt t0, t1, t2
                    bne t0, zero, if_less
                    nop
                    sub t1, t1, t2
                    j cycle
                    nop
    if_less:        sub t2, t2, t1
                    j cycle
    done:           add t3, t1, zero
```