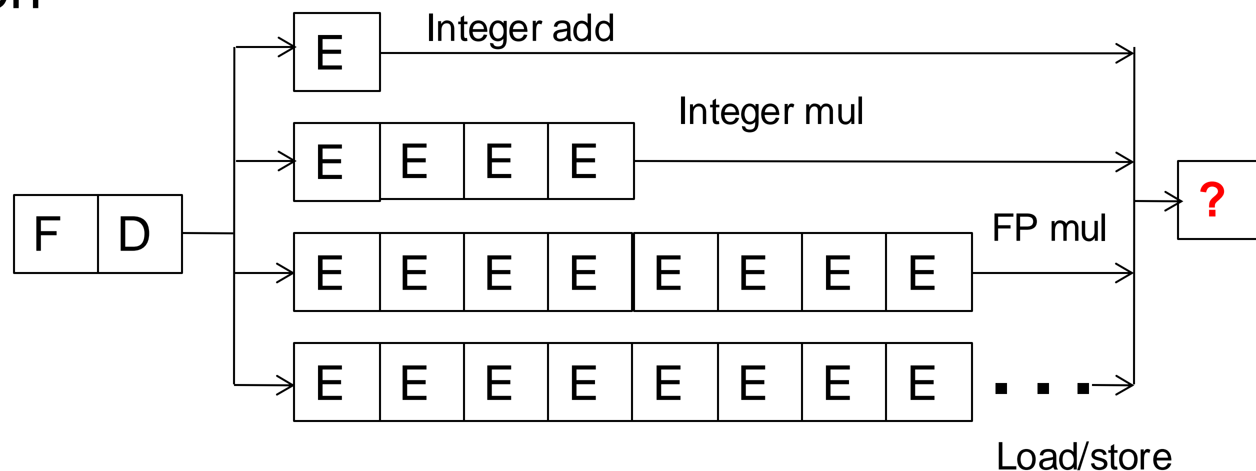
Pipelining and Precise Exceptions: Preserving Sequential Semantics

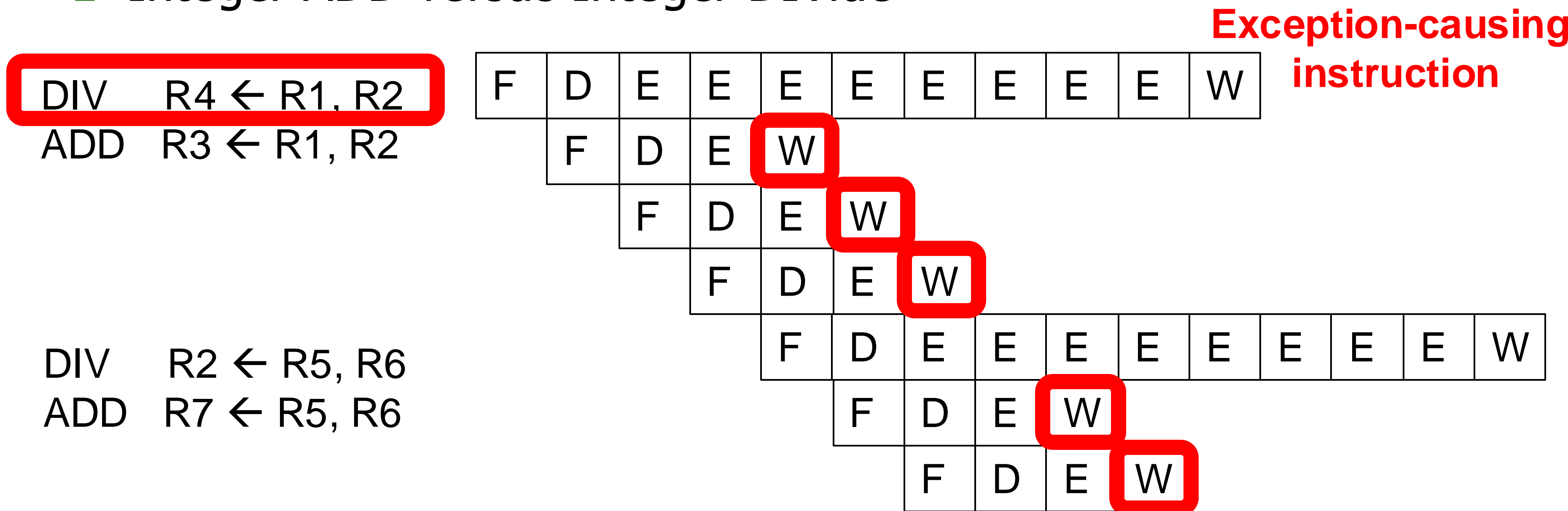
Multi-Cycle Execution

- Not all instructions take the same amount of time in the "execute stage" of the pipeline
- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution



Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus Integer DIVide



  - What is wrong with this picture in a Von Neumann architecture?
    - Sequential semantics of the ISA NOT preserved!
    - What if DIV incurs an exception? (e.g., DIV by zero)

Exceptions and Interrupts

- "Unplanned" changes or interruptions in program execution

- Due to **internal** problems in execution of the program
  → Exceptions

- Due to **external** events that need to be handled by the processor
  → Interrupts

- Both exceptions and interrupts require
  - stopping of the current program
  - saving the architectural state
  - handling the exception/interrupt → switch to handler
  - (if possible and makes sense) returning back to program execution

Exceptions and Interrupts: Examples

- Exception examples
  - Divide by zero
  - Overflow
  - Undefined opcode
  - General protection (or access protection)
  - Page fault
  - …

- Interrupt examples
  - I/O device needing service (e.g., keyboard input, video input)
  - (Periodic) system timer expiration
  - Power failure
  - Machine check
  - …

Exceptions vs. Interrupts

- Cause
  - Exceptions: internal to the running thread
  - Interrupts: external to the running thread

- When to Handle
  - Exceptions: when detected (and known to be non-speculative)
  - Interrupts: when convenient
    - Except for very high priority ones
      - Power failure
      - Machine check (error)

- Priority: process (exception), depends (interrupt)

- Handling Context: process (exception), system (interrupt)

Precise Exceptions/Interrupts

- The architectural state should be consistent (precise) when the exception/interrupt is ready to be handled

1. All previous instructions should be completely retired

2. No later instruction should be retired

Retire = commit = finish execution and update arch. state

DIV R4 ← R1, R2

ADD R3 ← R1, R2

DIV R2 ← R5, R6

ADD R7 ← R5, R6

Precise state (clean separation of sequential instructions)

Checking for and Handling Exceptions in Pipelining

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic
  - Ensures architectural state is precise (register file, PC, memory)
  - Flushes all younger instructions in the pipeline
  - Saves PC and registers (as specified by the ISA)
  - Redirects the fetch engine to the appropriate exception handling routine

Aside: From the x86-64 ISA Manual

6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT *n* instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor's interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. Dataflow

- Aids software debugging

- Enables (easy) recovery from exceptions

- Enables (easily) restartable processes

- Enables traps into software (e.g., software implemented opcodes)

Ensuring Precise Exceptions

- Easy to do in single-cycle and multi-cycle machines

- Single-cycle
  - Instruction boundary == Cycle boundary
  - An instruction is guaranteed to be finished in one cycle
    → no possibility of violating sequential execution semantics

- Multi-cycle
  - Add special states in the control FSM that lead to the exception or interrupt handlers
  - Switch to the handler only at a precise state
    → before fetching the next instruction

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture

Precise Exceptions in Multi-Cycle Datapath

EPC register: Holds the exception causing PC

Cause register: Holds the cause of the exception

Exception Handler starts at address 0x80000180

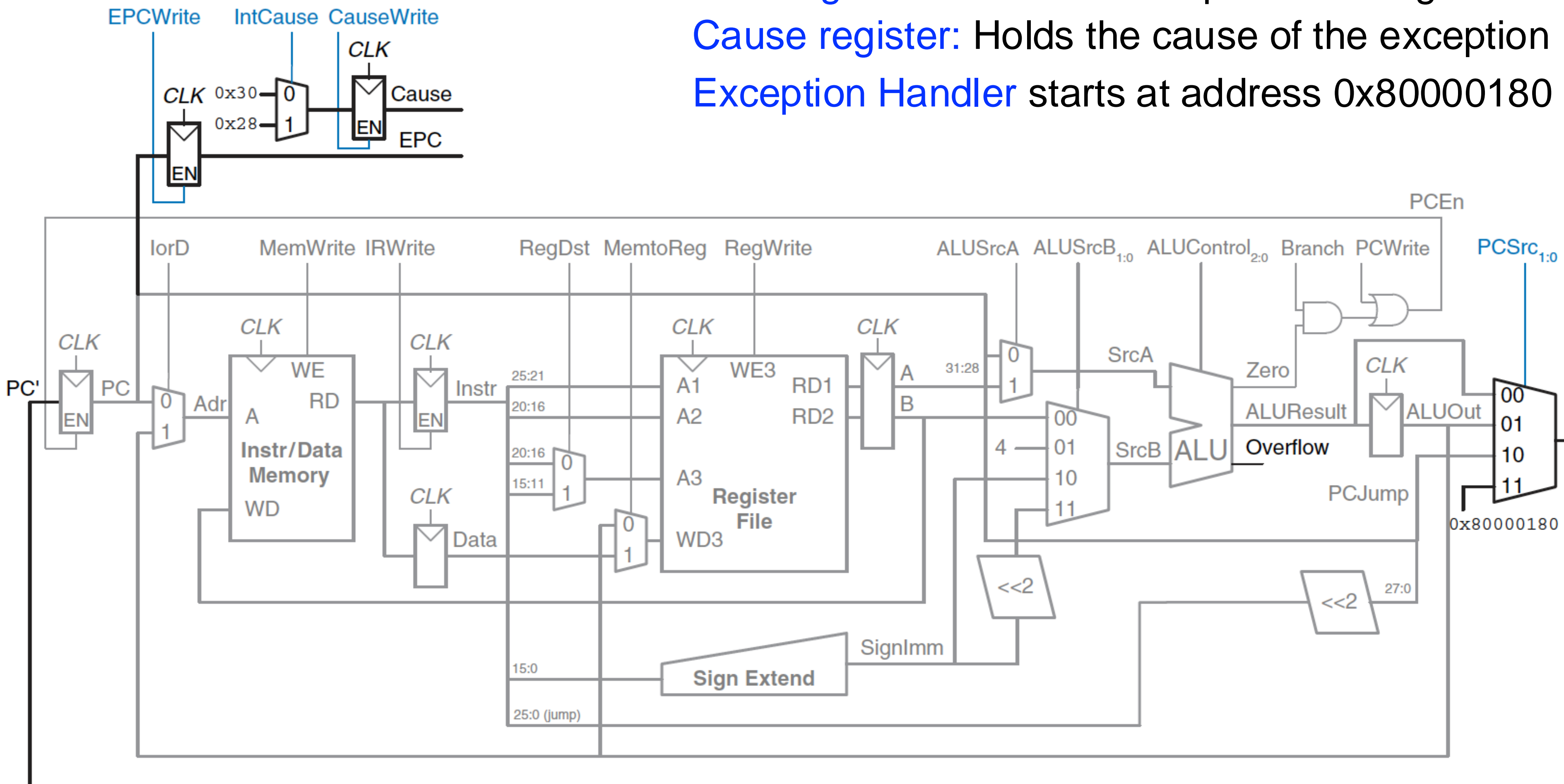


Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture

Precise Exceptions in Multi-Cycle FSM

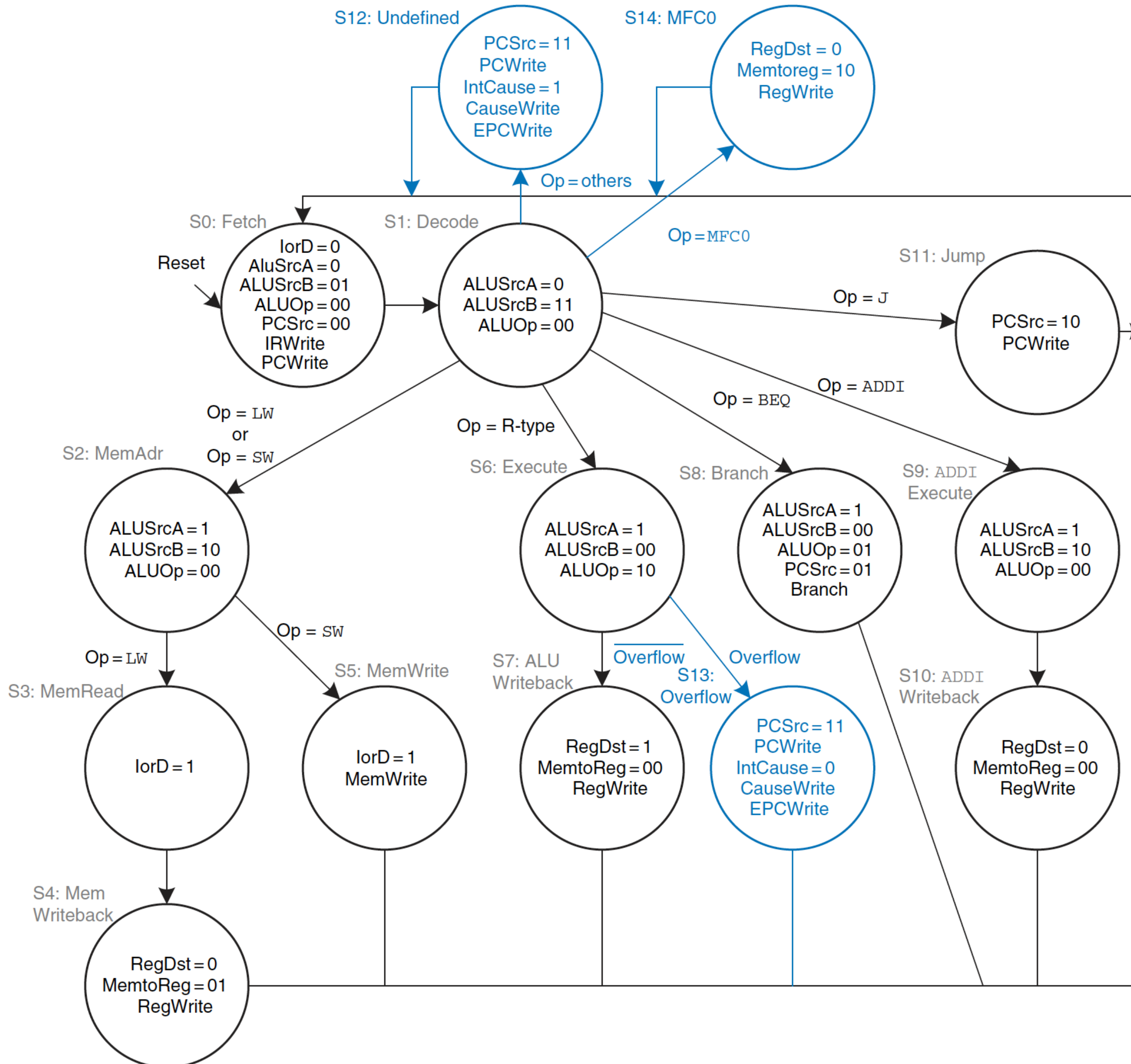


Figure 7.64 **Controller supporting exceptions and mfc0**

- Supports
  - Overflow
  - Undefined instruction
- mfc0 instruction is used to copy the exception cause into a general-purpose register

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture

Precise Exceptions in Multi-Cycle Datapath

In summary, an exception causes the processor to jump to the exception handler. The exception handler saves registers on the stack, then uses `mfc0` to look at the cause and respond accordingly. When the handler is finished, it restores the registers from the stack, copies the return address from `EPC` to `$k0` using `mfc0`, and returns using `jr $k0`.

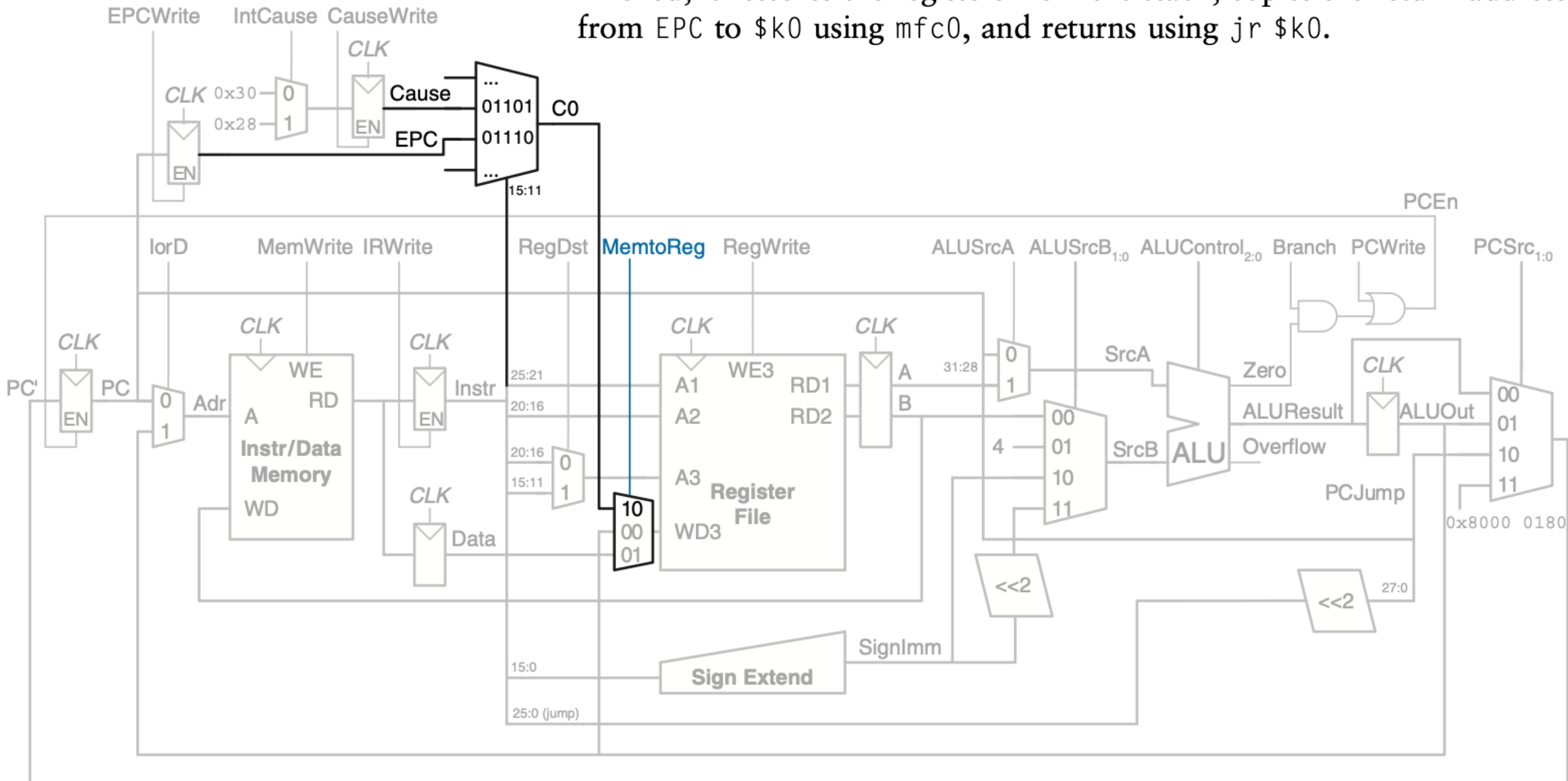
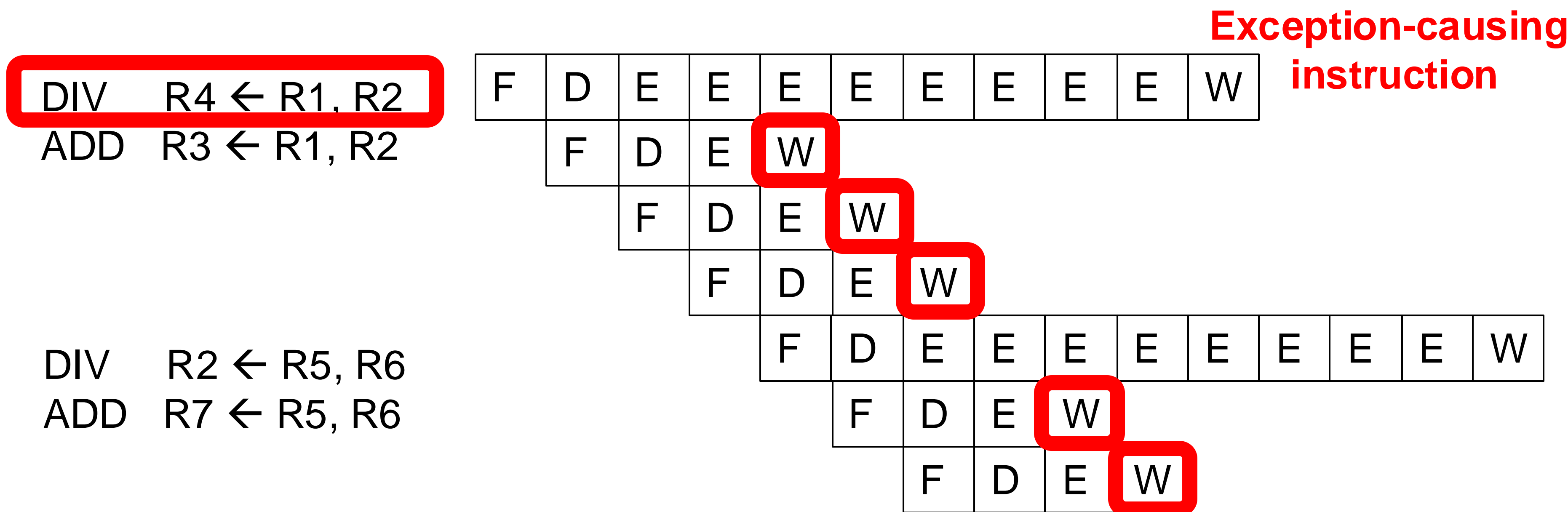


Figure 7.63 **Datapath supporting** `mfc0`

Multi-Cycle Execute: More Complications

- Instructions can take different number of cycles in EXECUTE stage → **This complicates exception/interrupt handling**

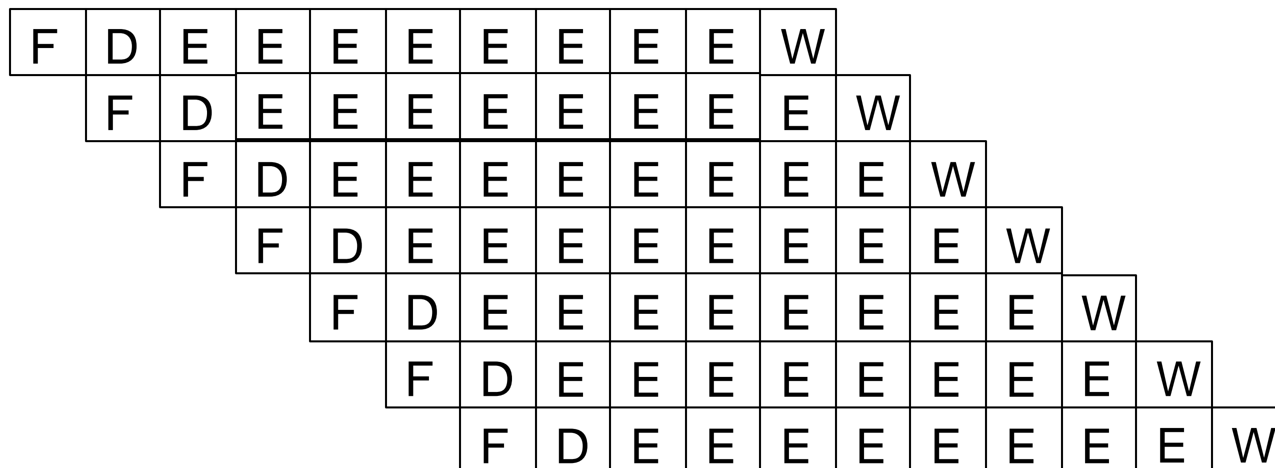


- What is wrong with this picture in a Von Neumann architecture?
  - Sequential semantics of the ISA NOT preserved!
  - What if DIV incurs an exception? (e.g., DIV by zero)

Ensuring Precise Exceptions in Pipelining

- Idea: Make each operation take the same amount of time

DIV R3 ← R1, R2
ADD R4 ← R1, R2



- Downside
  - Worst-case instruction latency determines all instructions' latency
    - What about memory operations?
    - Each functional unit takes worst-case number of cycles?

Solutions: Supporting Precise Exceptions

- How do we support precise exceptions in the presence of instructions completing out of program order?

- Reorder buffer

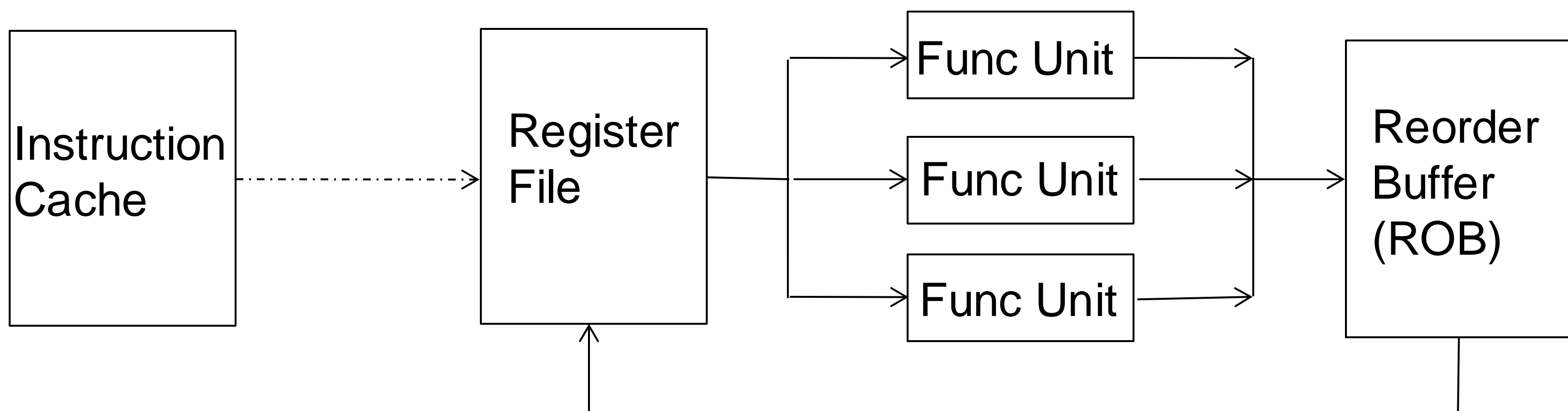
- History buffer
- Future register file
- Checkpointing

We will not cover these
See suggested lecture video from Spring 2015
Also see backup slides

- Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 and ISCA 1985.

Solution I: Reorder Buffer (ROB)

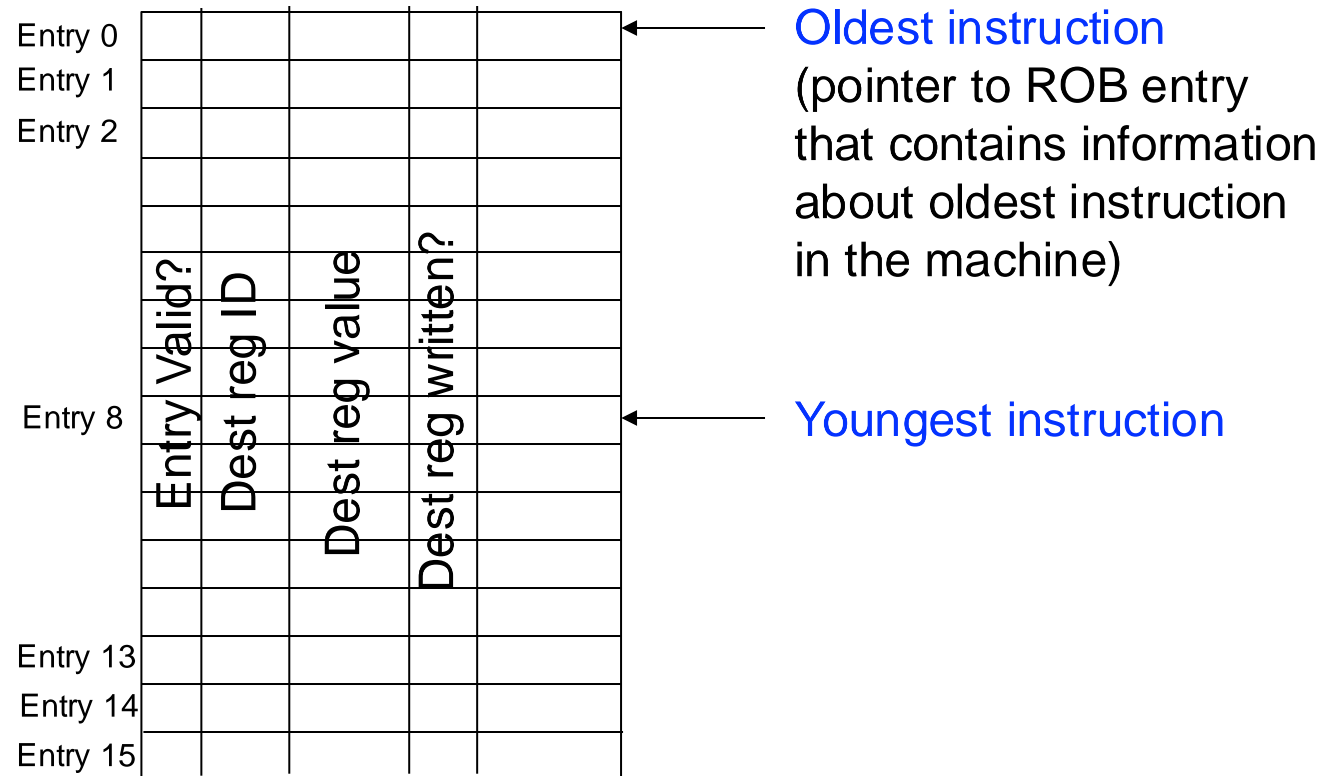
- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is **decoded**, it reserves the next-sequential entry in a special buffer called the Reorder Buffer (ROB)
- When instruction **completes**, it writes result into ROB entry
- When instruction **oldest in ROB** and it has completed without exceptions, its result moved to reg. file or memory



ROB is implemented as a circular queue in hardware

Reorder Buffer

- A hardware structure that keeps information about all instructions that are **decoded** but **not yet retired**/committed



ROB is implemented as a circular queue in hardware

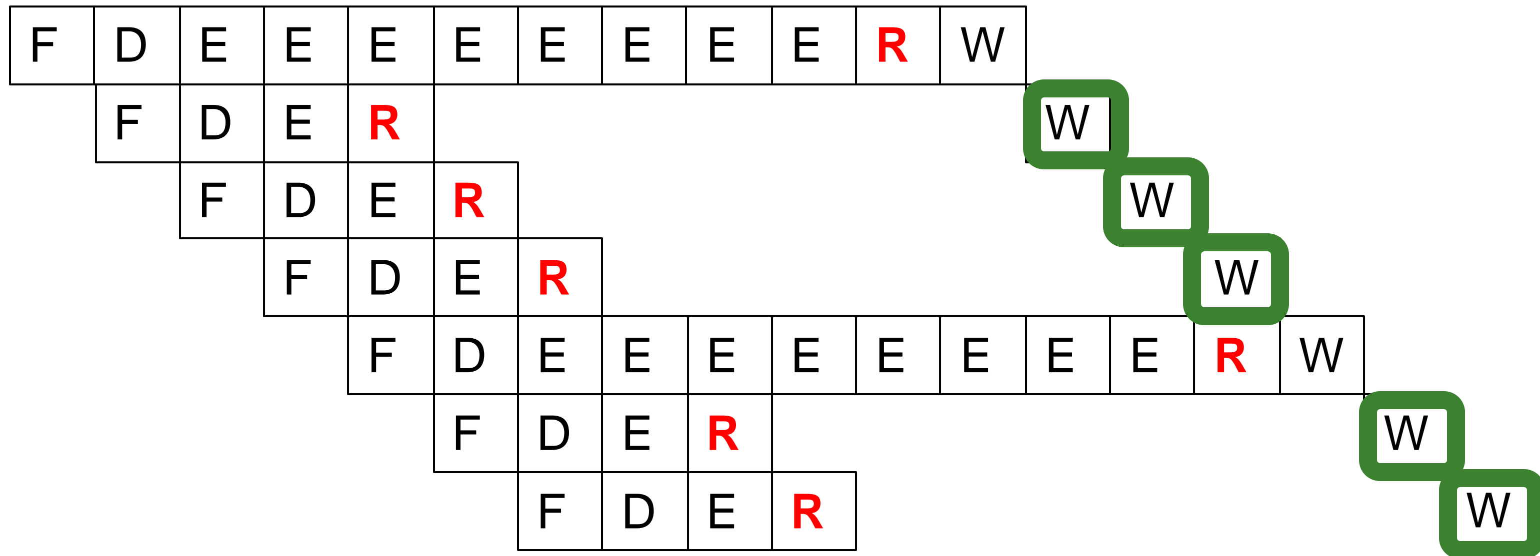
What's in a ROB Entry?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exception?

- Everything required to:
  - correctly reorder instructions back into the program order
  - update the architectural state with the instruction's result(s), if instruction can retire without any issues
  - handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction
- Need valid bits to keep track of readiness of the result(s) and find out if the instruction has completed execution

Reorder Buffer: Independent Operations

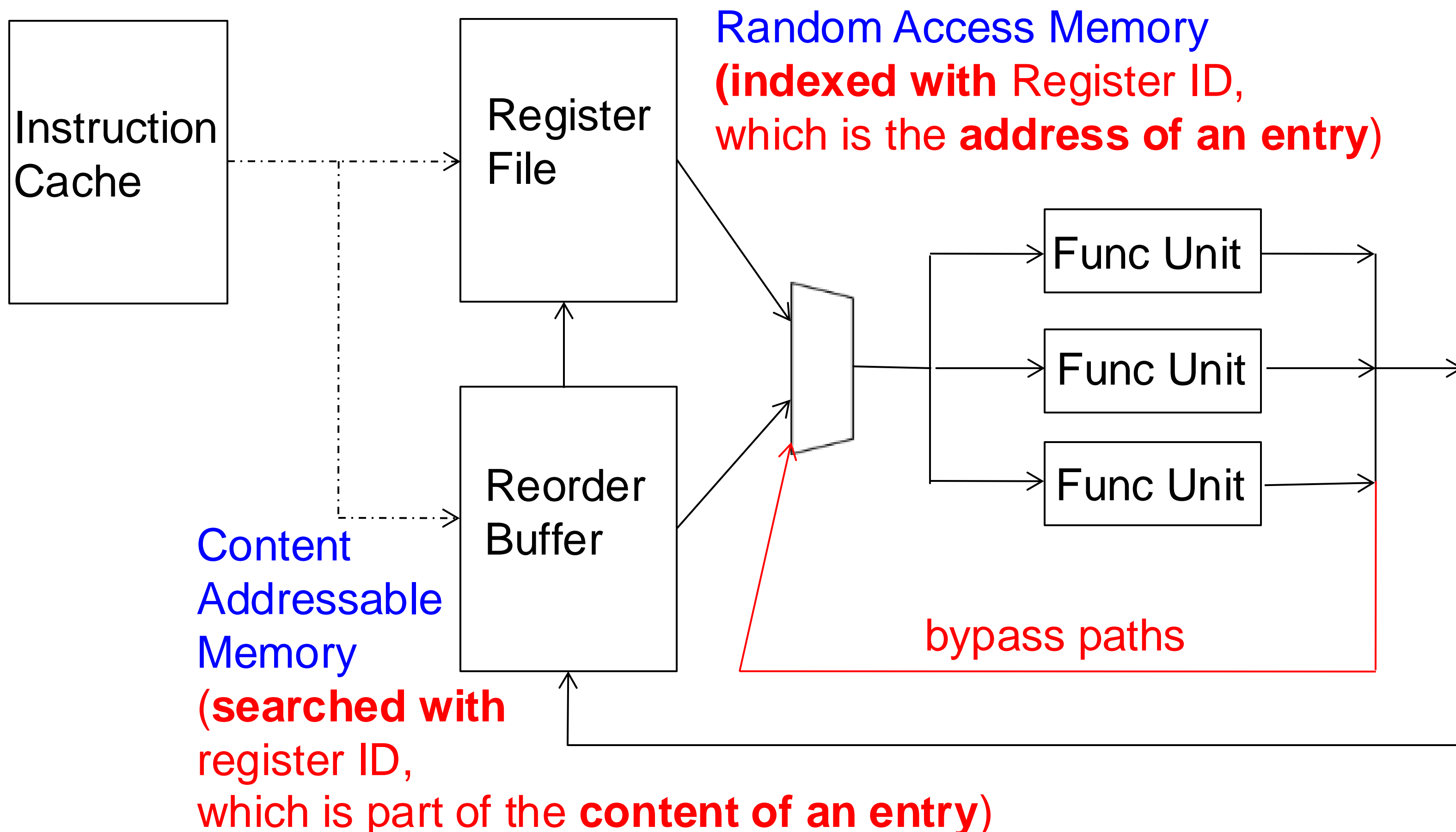
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- What if a later instruction needs a value in the reorder buffer?
  - One option: stall the operation → stall the pipeline
  - Better: Read the value from the reorder buffer. How?

Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



Reorder Buffer Example

Register File (RF)

	Value Valid?	Value
R0		
R1		
R2		
R3		
R4		
R5		
R6		
R7		

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0					← Oldest instruction
Entry 1					
Entry 2					
Entry 8					← Youngest instruction
Entry 13					
Entry 14					
Entry 15					

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3

MUL R3, R4 → R11

ADD R5, R6 → R3

ADD R3, R8 → R12

Simplifying Reorder Buffer Access

- Idea: Use indirection

- Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- Access reorder buffer next

- Now, reorder buffer does not need to be content addressable

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value	Tag (pointer to ROB entry)
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

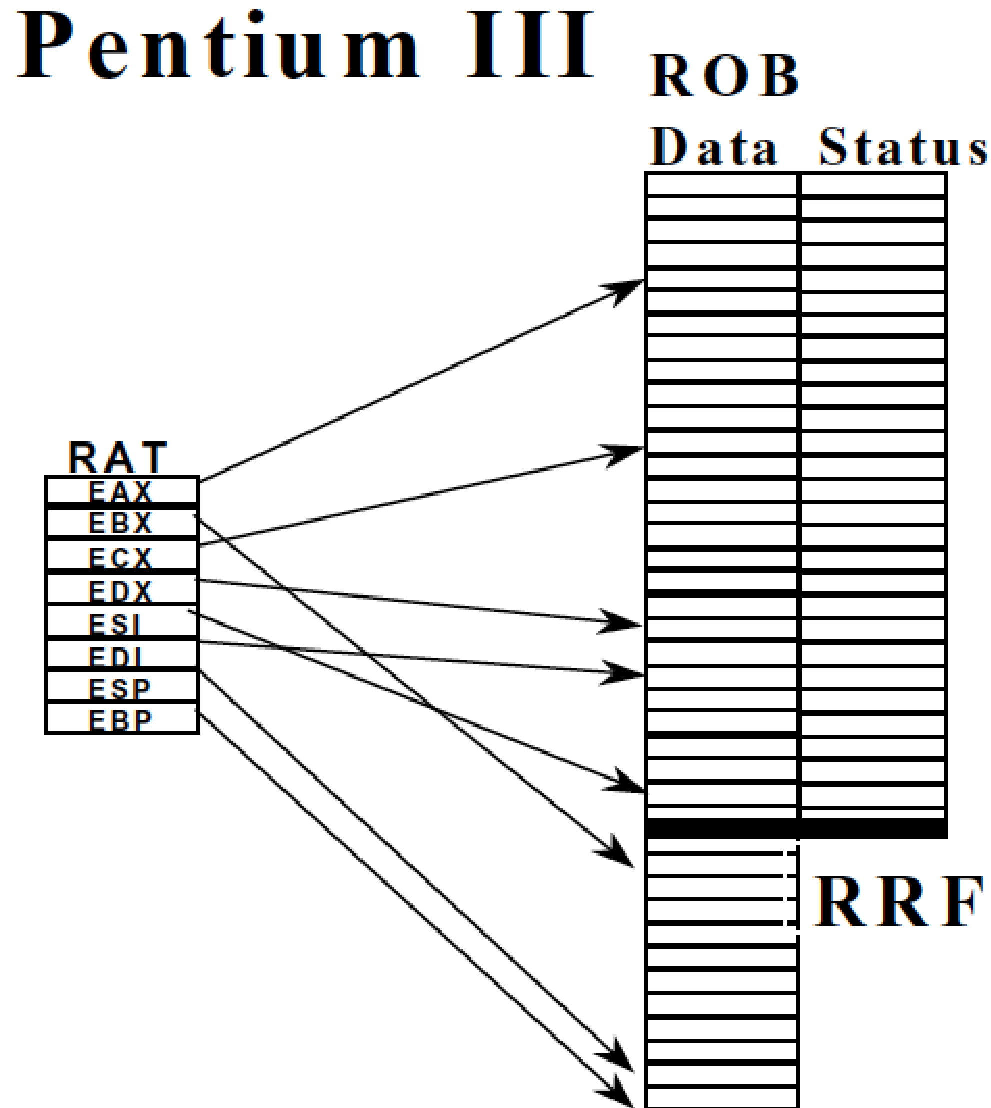
Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					

Oldest instruction → Entry 0

Youngest instruction → Entry 8

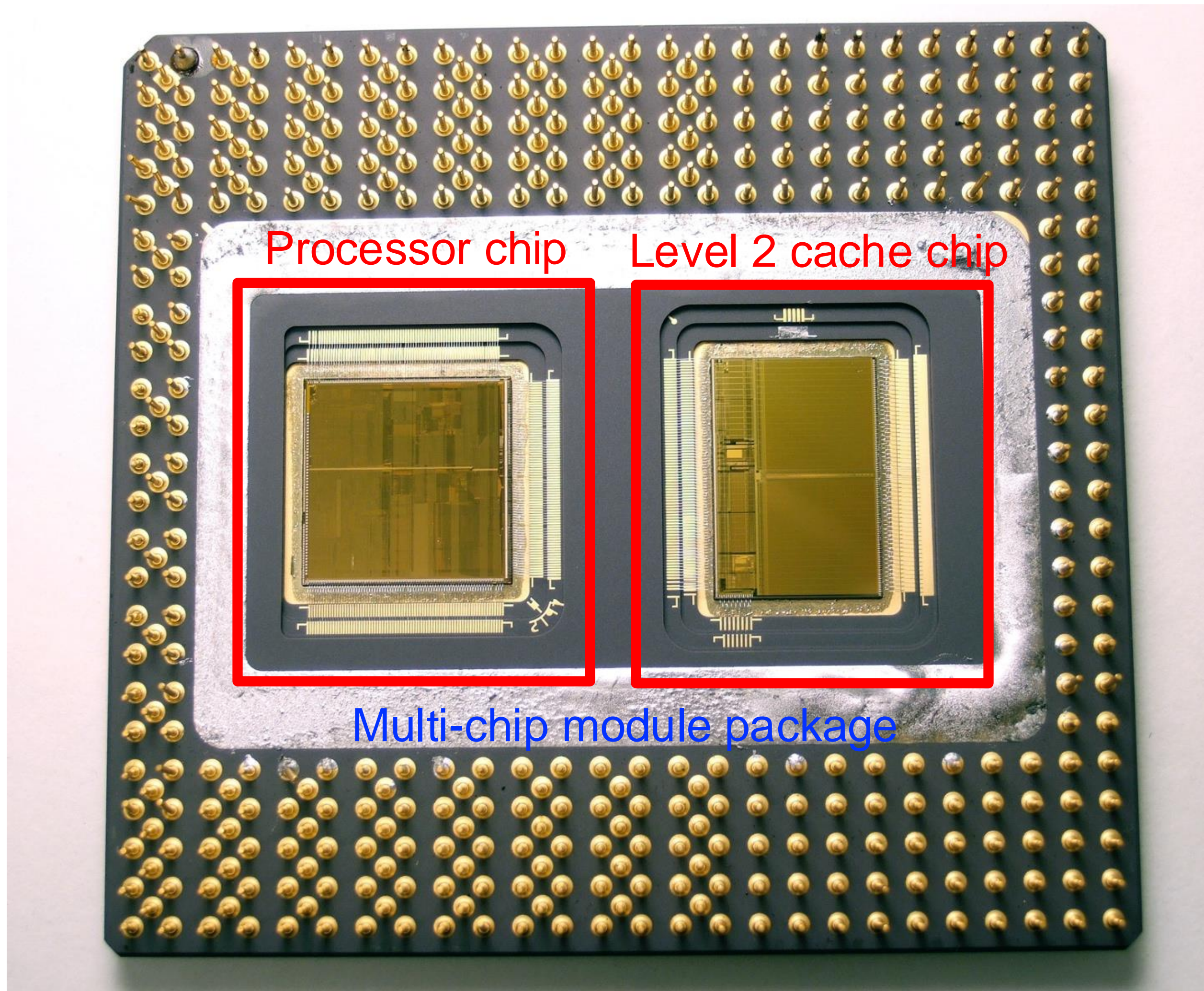
Reorder Buffer in Intel Pentium III/Pro



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

A Register Alias Table (RAT) points to where each register's current value is (or will be)

Intel Pentium Pro (1995)



By Moshen - http://en.wikipedia.org/wiki/Image:Pentiumpro_moshen.jpg, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=2262471

Important: Register Renaming with a Reorder Buffer

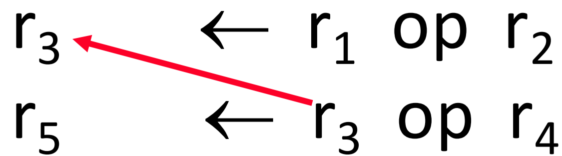
- Output and anti dependences are **not true dependences**
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID's (i.e. names) in the ISA**

- The register ID is **renamed** to the reorder buffer entry that will hold the register's value
  - Register ID → ROB entry ID
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register

- This eliminates anti and output dependences
  - Gives the illusion that there are a large number of registers

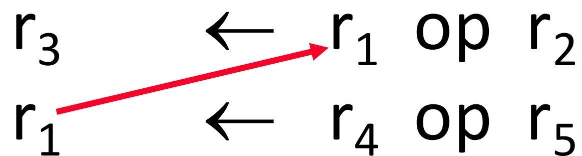
Recall: Data Dependence Types

Flow dependence



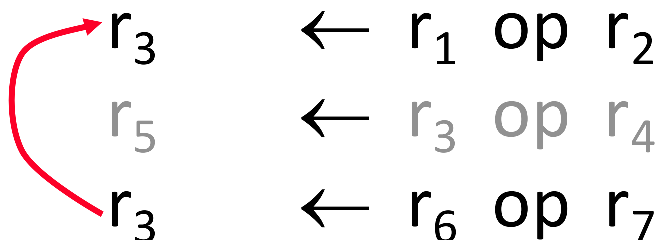
Read-after-Write
(RAW)

Anti dependence



Write-after-Read
(WAR)

Output dependence



Write-after-Write
(WAW)

Register Renaming Example (On Your Own)

- Assume
  - Register file has a pointer to the reorder buffer entry that contains or will contain the value, if the register is not valid
  - Reorder buffer works as described before

- Where is the latest definition of R3 for each instruction below in sequential order?

  LD R0(0) → R3
  LD R3, R1 → R10
  MUL R1, R2 → R3
  MUL R3, R4 → R11
  ADD R5, R6 → R3
  ADD R3, R8 → R12

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value	Tag (pointer to ROB entry)
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

LD R0(0) → R3
LD R3, R1 → R10
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer (ROB)

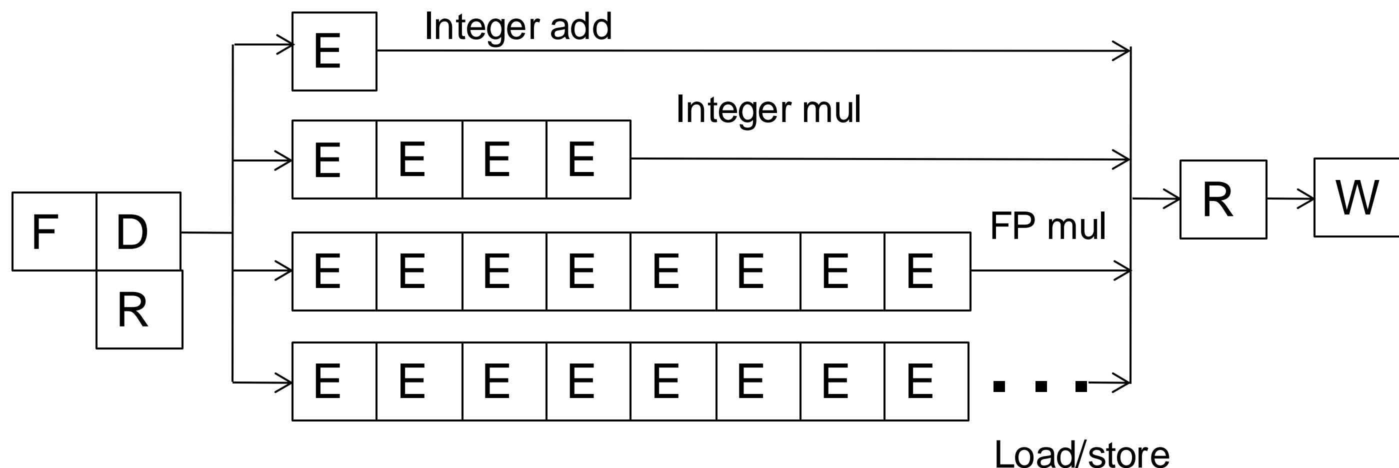
	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					

Oldest instruction (→ Entry 0)

Youngest instruction (→ Entry 8)

In-Order Pipeline with Reorder Buffer

- Decode (D): Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to reorder buffer
- Retirement/Commit (W): Check oldest instruction for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- In-order dispatch/execution, out-of-order completion, in-order retirement



ROB is implemented as a circular queue in hardware

Reorder Buffer Tradeoffs

- Advantages
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependences

- Disadvantages
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection → increased latency and complexity

- Other solutions aim to eliminate the disadvantages
  - History buffer
  - Future file
  - Checkpointing

We will not cover these
See suggested lecture video from Spring 2015
Also see backup slides