15-740/18-740 Computer Architecture Lecture 5: Precise Exceptions

Prof. Onur Mutlu
Carnegie Mellon University

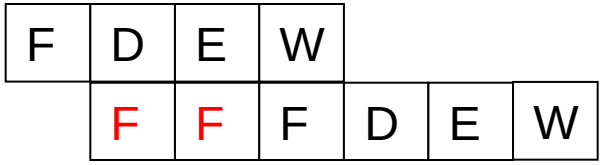
Last Time …

- Performance Metrics
- Amdahl's Law
- Single-cycle, multi-cycle machines
- Pipelining
- Stalls
- Dependencies

Issues in Pipelining: Increased CPI

- Control dependency stall: what to fetch next

BEQ R1, R2, TARGET

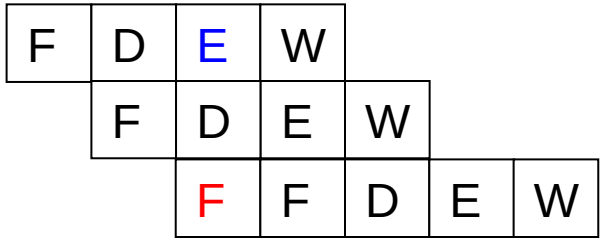


  - Solution: predict which instruction comes next
    - What if prediction is wrong?
  - Another solution: hardware-based fine-grained multithreading
    - Can **tolerate** both data and control dependencies
    - **Read:** James Thornton, "Parallel operation in the Control Data 6600," AFIPS 1964.
    - **Read:** Burton Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Issues in Pipelining: Increased CPI

- Resource Contention Stall
  - What if two concurrent operations need the same resource?

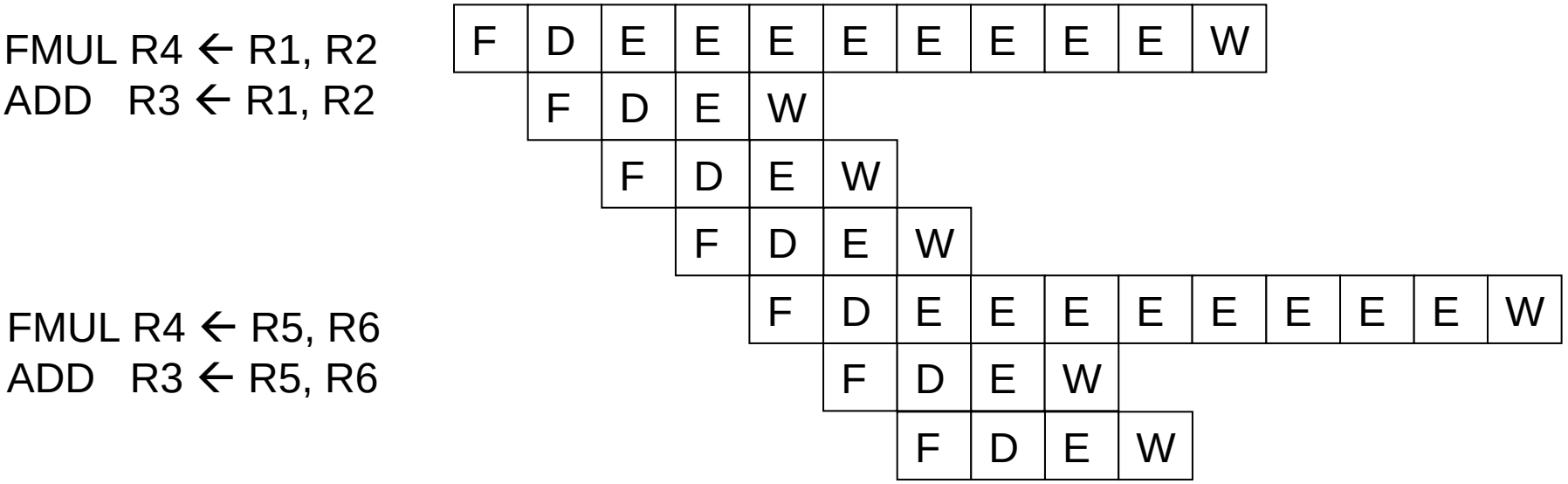
```
LD    R1 ← R2(4)
ADD   R2 ← R1, R5
ADD   R6 ← R3, R4
```



  - Examples:
    - *Instruction fetch* and *data fetch* both need memory. Solution?
    - *Register read* and *register write* both need the register file
    - A *store instruction* and a *load instruction* both need to access memory. Solution?

Issues in Pipelining: Multi-Cycle Execute

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



  - What is wrong with this picture?
    - What if FMUL incurs an exception?
    - Sequential semantics of the ISA NOT preserved!

Handling Exceptions in Pipelining

- Exceptions versus 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
- Priority: process (exception), depends (interrupt)
- Handling Context: process (exception), system (interrupt)

Precise Exceptions/Interrupts

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

1. All previous instructions should be completely retired.

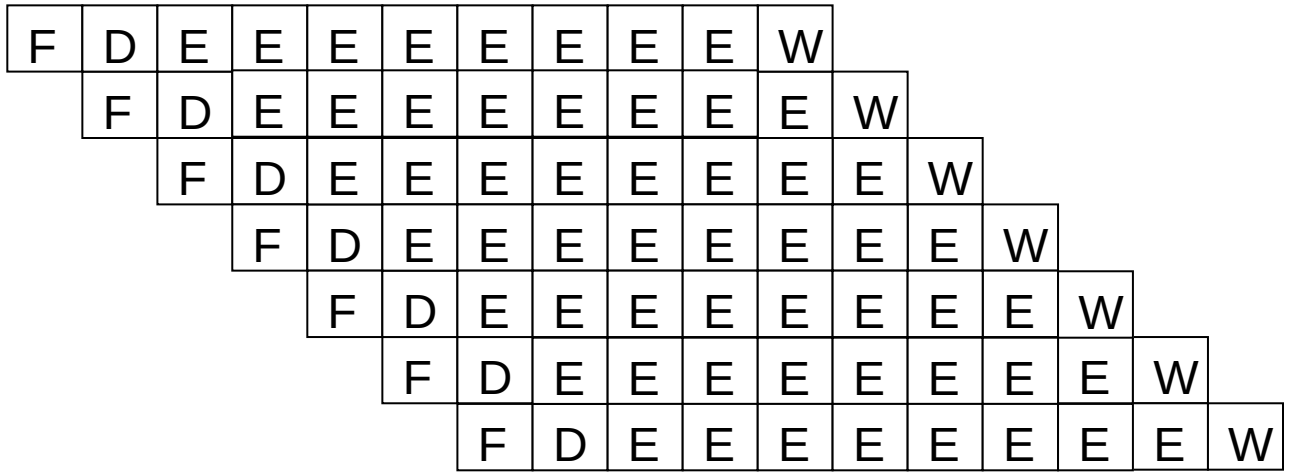
2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state

Ensuring Precise Exceptions in Pipelining

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

FMUL R3 ← R1, R2
ADD R4 ← R1, R2



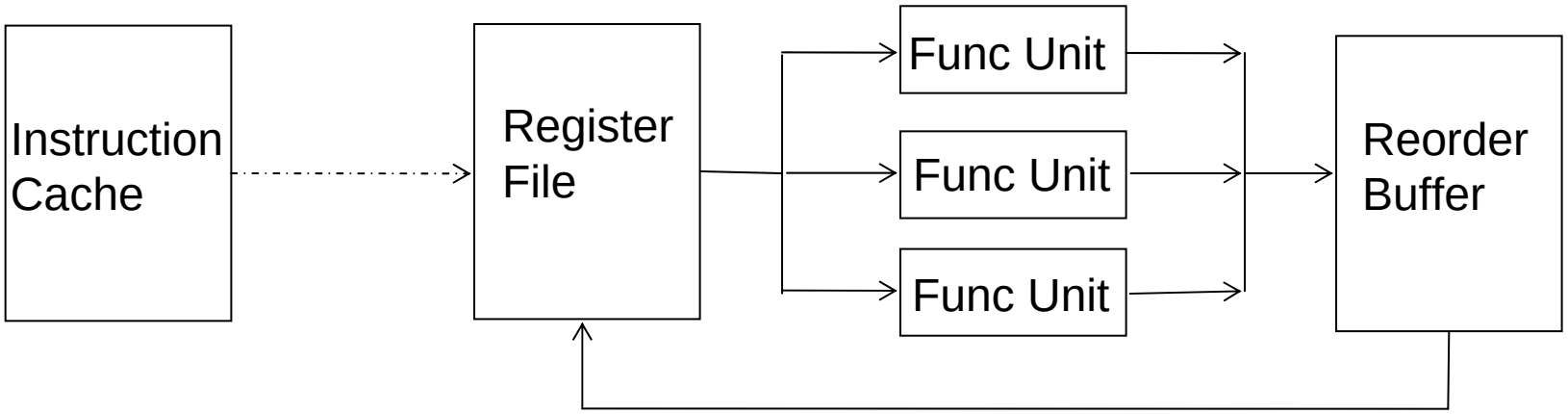
- Downside
  - What about memory operations?
  - Each functional unit takes 500 cycles?

Solutions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing
- Reading
  - Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors" IEEE Trans on Computers 1988 and ISCA 1985.
  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

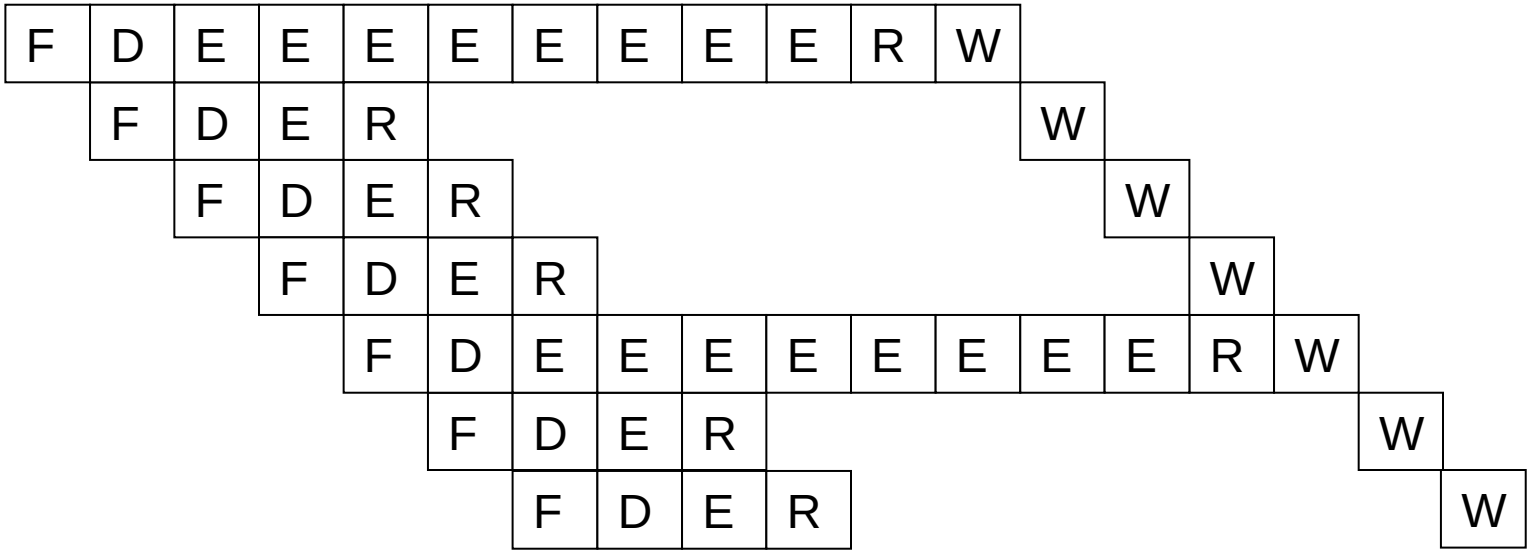
Solution I: Reorder Buffer (ROB)

- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves an entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB, its result moved to reg. file or memory



Reorder Buffer: Independent Operations

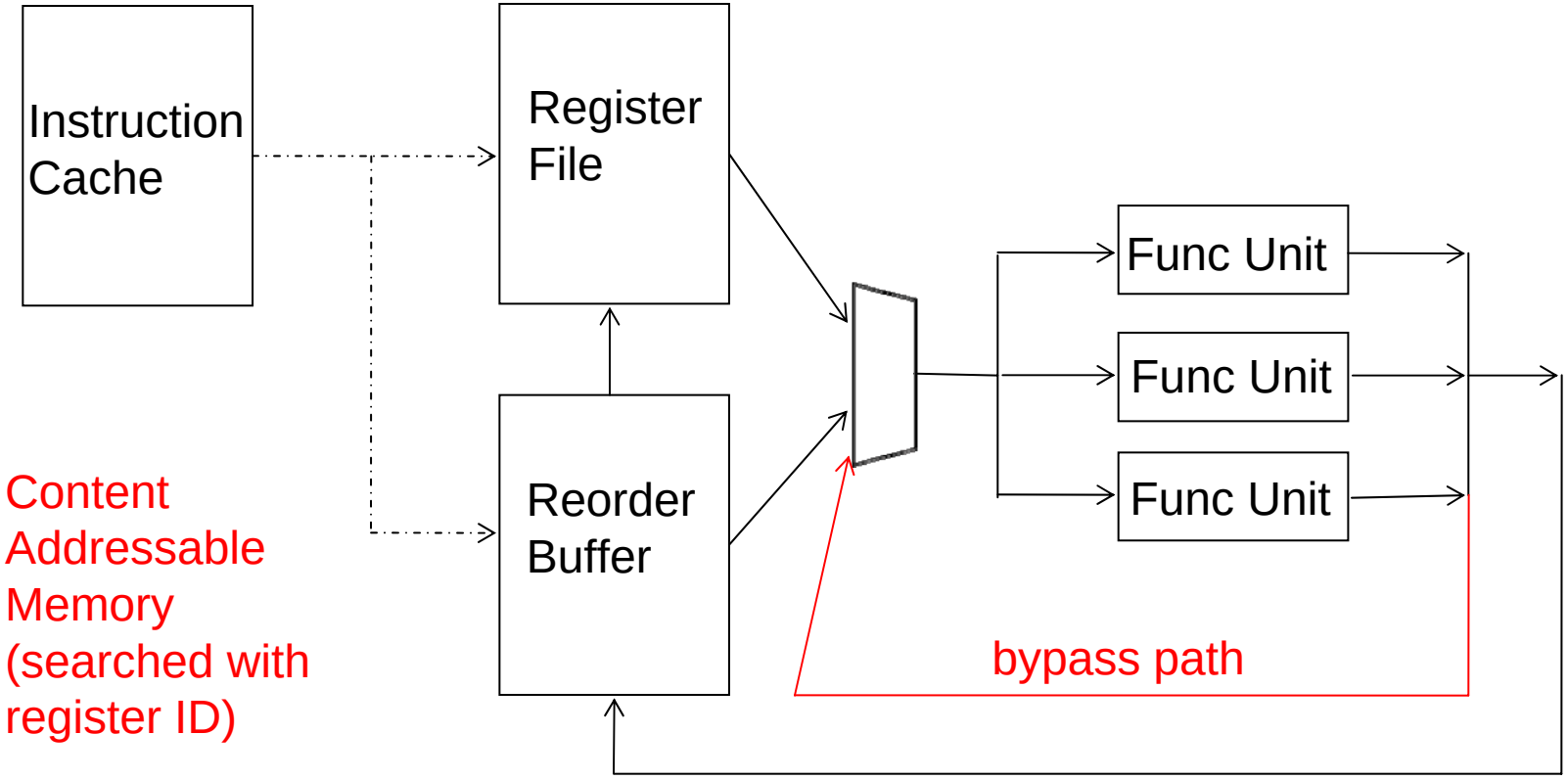
- Results first written to ROB, then to register file at commit time



- What if a later operation needs a value in the reorder buffer?
  - Read reorder buffer in parallel with the register file. How?

Reorder Buffer: How to Access?

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



Simplifying Reorder Buffer Access

- Idea: Use indirection
- Access register file first
  - 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
- Access reorder buffer next

- What is in a reorder buffer entry?

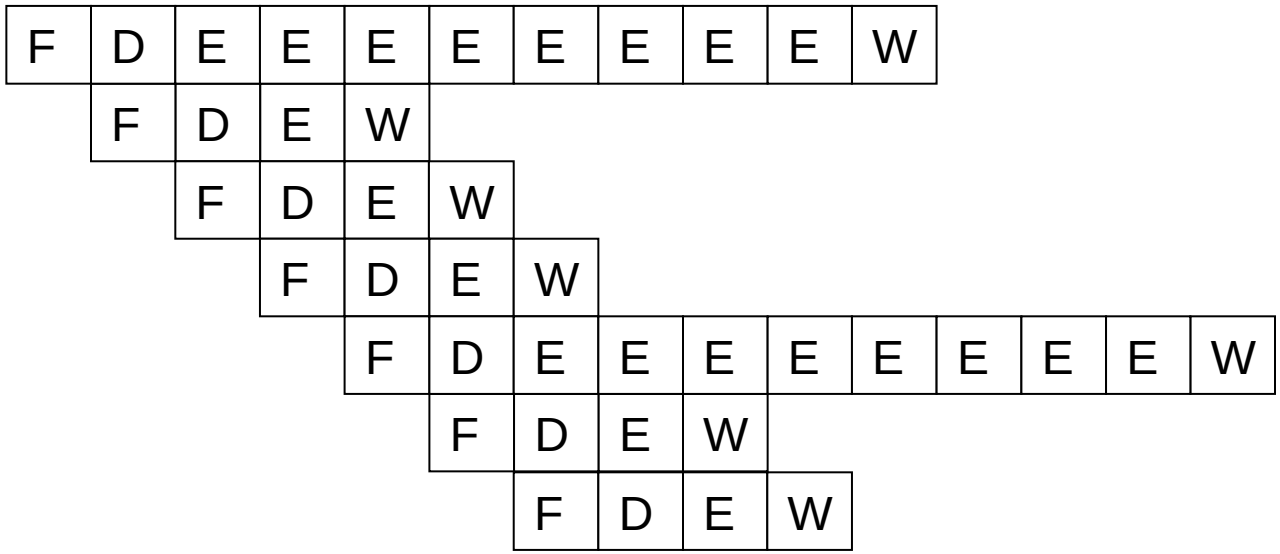
V	DestRegID	DestRegVal	StoreAddr	StoreData	BranchTarget	PC/IP	Control/valid bits

  - Can it be simplified further?

What is Wrong with This Picture?

FMUL R4 ← R1, R2
ADD R3 ← R1, R2

FMUL R2 ← R5, R6
ADD R4 ← R5, R6



- What is R4's value at the end?
  - The first FMUL's result
  - Output dependency not respected

Register Renaming with a Reorder Buffer

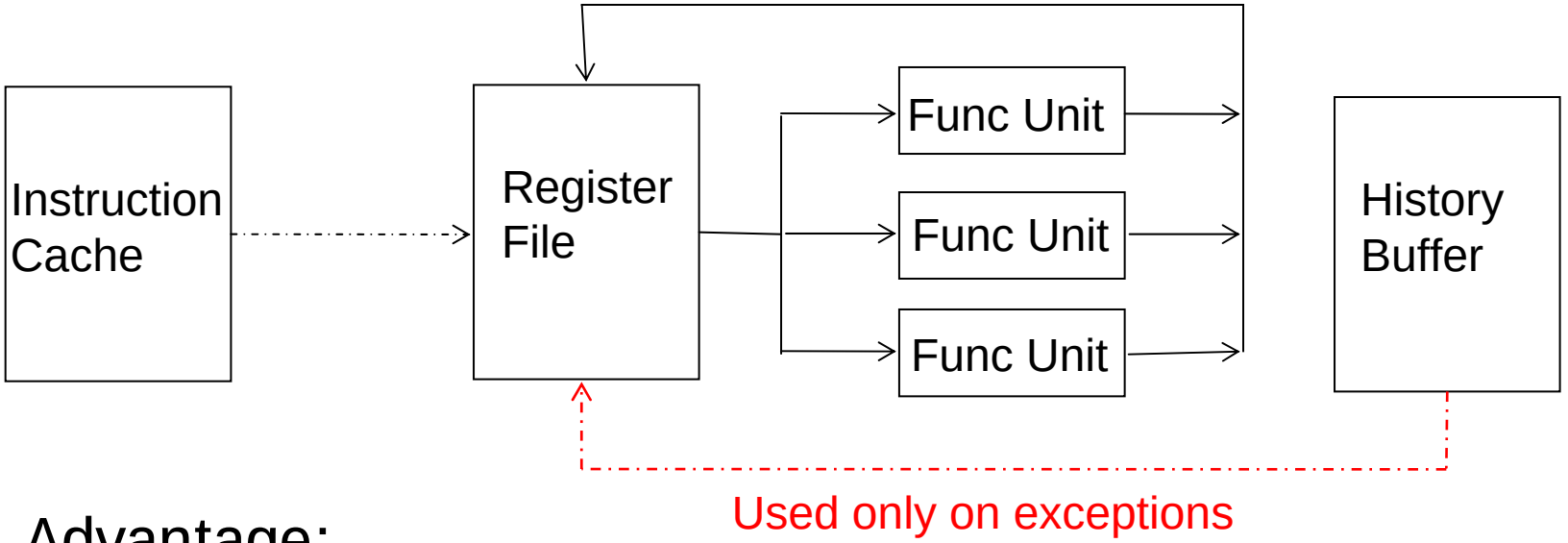
- Output and anti dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID's (i.e. names) in the ISA**
- The register ID is renamed to the reorder buffer entry that will hold the register's value
  - 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- dependencies
  - Gives the illusion that there are a large number of registers

Solution II: History Buffer (HB)

- Idea: Update architectural state when instruction completes, but UNDO UPDATES when an exception occurs

- When instruction is decoded, it reserves an HB entry
- When the instruction completes, it stores the old value of its destination in the HB
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head

History Buffer



- Advantage:
  - Register file contains up-to-date values. History buffer access not on critical path
- Disadvantage:
  - Need to read the old value of the destination
  - What about stores?

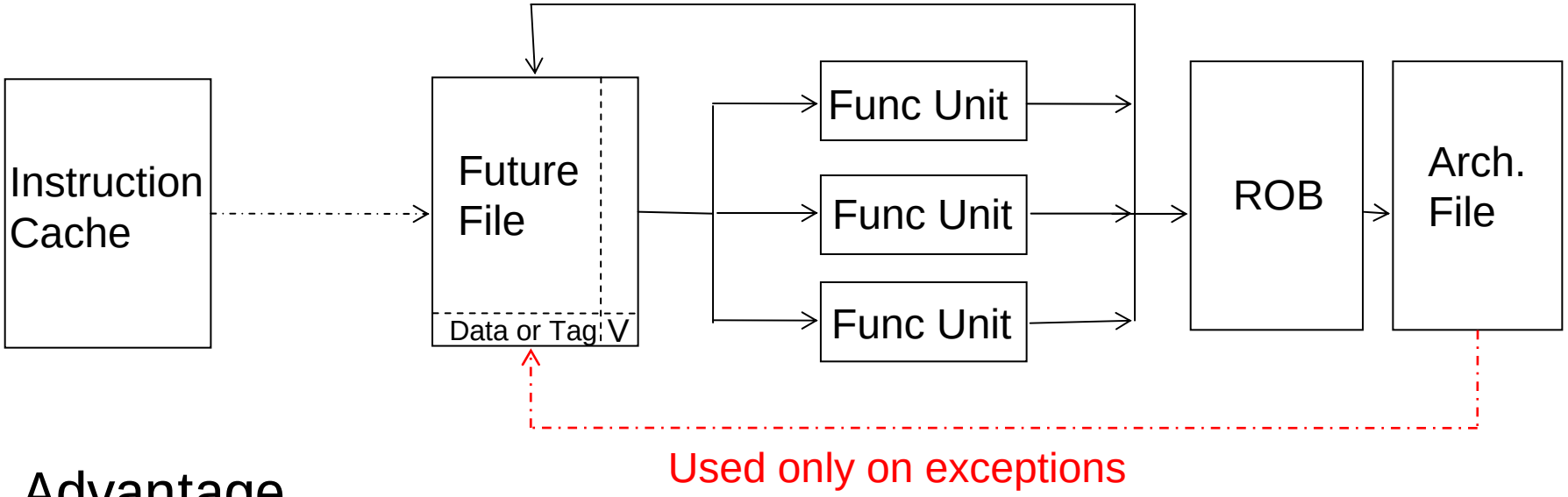
Solution III: Future File (FF)

- Idea: Keep two register files:
  - Arch reg file: Updated in program order for precise exceptions
  - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)

- Future file is used for fast access to latest register values

- Architectural file is used for recovery on exceptions

Future File



- Advantage
  - No sequential scanning of history buffer: Upon exception, simply copy arch file to future file
  - No need for extra read of destination value
- Disadvantage
  - Multiple register files + reorder buffer

Checkpointing

- Idea: Periodically checkpoint the register file state. When exception/interrupt occurs, go back to the most recent checkpoint and re-execute instructions one by one to re-generate exception.
- State guaranteed to be precise only at checkpoints.
- Advantage:
  - Allows for aggressive execution between checkpoints
  - Per-instruction reorder buffer is not needed
- Disadvantage:
  - Interrupt latency depends on distance from checkpoint
  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

Summary: Precise Exceptions in Pipelining

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

Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an “exception”
  - Except it is not visible to software

- What about branch misprediction recovery?
  - Similar to exception handling except can be initiated before the branch is the oldest instruction
  - All three state recovery methods can be used

- Difference between exceptions and branch mispredictions?
  - Branch mispredictions more common: need fast recovery

Pipelining Issues: Stores

- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. Why?
  - One idea: Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - Store/write buffer: Similar to reorder buffer, but used only for store instructions
    - Program-order list of un-committed store operations
    - When store is decoded: Allocate a store buffer entry
    - When store address and data become available: Record in store buffer entry
    - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data