ECSE 425 Lecture 9: Exceptions; Multi-cycle Operations

H&P Appendix A

Last Time

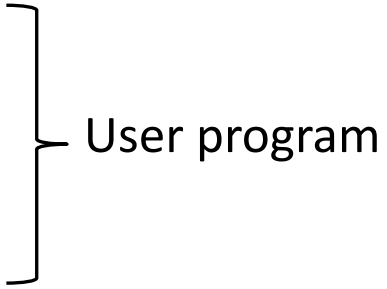
- Branch Hazards
- Branch Delay Mitigation
- Implementing Pipeline Control

Today

- Maintaining Precise Exceptions
- Implementing Multi-cycle Operations
  - E.g., floating point instructions
- Looking ahead: Instruction-level Parallelism!
  - Read Chapter 2!

What is an exception?

- *Exceptional* situations sometimes disrupt a program
- Major causes:
  - I/O device request
  - Invoking OS service from user program
  - Tracing instruction execution
  - Breakpoints
  - Integer arithmetic overflow
  - FP arithmetic anomaly
  - Page fault (not in memory, related to virtual memory, Chap 5)
  - Misaligned memory access (if alignment is required)
  - Memory protection violation
  - Undefined/unimplemented instructions
  - Hardware malfunctions of various kinds
  - Power failure



Characteristics of Exceptions

- Synchronous vs. asynchronous
  - always occur at the same place and data, or at random
- User requested vs. coerced
  - pre-programmed by the user, or caused by external event
- Maskable vs. nonmaskable
  - Hardware can be programmed to ignore, or not
- Within vs. between instructions
  - Within: prevents current instruction from completing
- Resume vs. terminate
  - CPU can restart where it was interrupted after service, or not

Exception Classification

I/O	Async.	Coerced	Nonmaskable	Between	Resume
OS Invoke	Sync.	Requested	Nonmaskable	Between	Resume
Trace Instr. ex.	Sync.	Requested	Maskable	Between	Resume
Breakpoint	Sync.	Coerced	Maskable	Between	Resume
Int. Overflow	Sync.	Coerced	Maskable	Within	Resume
FP Exception	Sync.	Coerced	Maskable	Within	Resume
Page Fault	Sync.	Coerced	Nonmaskable	Within	Resume
Misalign Mem. Acc.	Sync.	Coerced	Maskable	Within	Resume
Mem. Protection	Sync.	Coerced	Nonmaskable	Within	Resume
Undef. Instruction	Sync.	Coerced	Nonmaskable	Within	Terminate
Hardware Fault	Async.	Coerced	Nonmaskable	Within	Terminate
Power Failure	Async.	Coerced	Nonmaskable	Within	Terminate

Why Exceptions are Challenging

- Hardest case: within instruction and resumed (e.g.: page fault)
- Hardware must
  - Safely shut down the pipeline
  - Call exception handling routines
  - Restart the pipeline in the same state as when it was interrupted

Saving Pipeline State

- Insert a trap instruction into the pipeline
  - Cause the CPU to switch to an exception handling routine
  - Turn off all the writes for the faulting instruction and those that follow but not for those that precede
    - Preceding instructions can complete as if nothing happened
    - Those after (a) must not modify state, (b) must be re-executed
- Exception handling routine then takes control
  - Save the PC of faulty instruction
- Delayed branches complicate things
  - Need to save the PC for each delay slot

Exceptions and Floating Point Operations

- Floating point operations (more on this later)
  - can take unpredictable number of clock cycles
  - often complete out of order
- "Imprecise" exception mode
  - Boosts performance when using floating point ops
  - Can be >10x faster by allowing more inst. overlapping
- Precise exceptions are often required
  - For integer pipeline
  - For accommodating virtual memory

Simultaneous Exceptions

IF	Page Fault, misalignment, memory protection
ID	Undefined instruction
EX	Arithmetic exception (e.g. overflow)
MEM	Page Fault, misalignment, memory protection
WB	None

- Multiple exceptions can occur in the same cycle
- Example

  LD IF ID EX **MEM** WB

  DADD IF ID **EX** MEM WB
- A page fault in MEM, an arithmetic exception in EX
  - Deal with page fault first, then restart, deal with EX later

Out-of-order Exceptions

- Exceptions can be raised out of program order
- Example

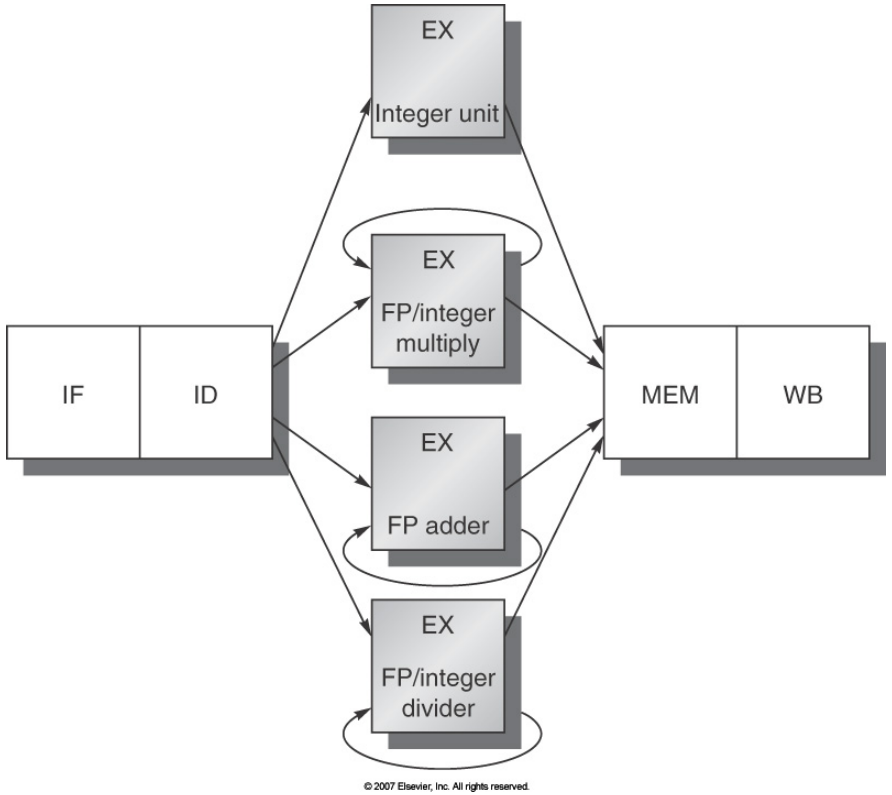
  LD IF ID EX **MEM** WB
  DADD **IF** ID EX MEM WB
- A data page fault in the MEM stage is raised after a an instruction page fault in the IF stage of a later instruction
  - Exceptions raised by LD must be serviced first!
- Track exceptions in a status vector associated with each instruction in the pipe
  - Check the status vector when the instruction enters WB
  - Exceptions can be processed in program order

ISA Complications for Exception Handling

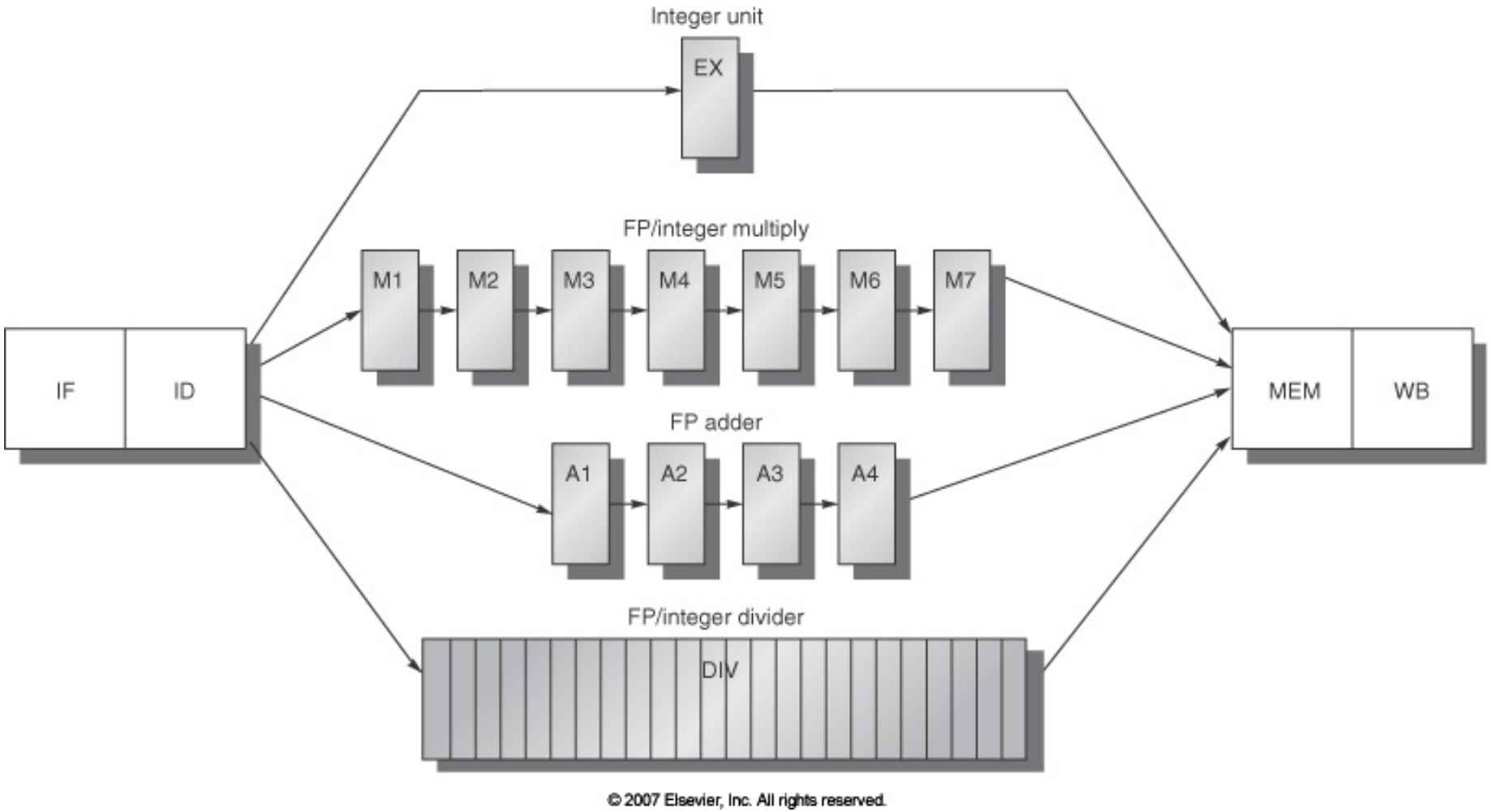
- Precise exceptions are straightforward in MIPS
  - Each instruction has only one result
  - This result is committed the end of instruction execution
- Some ISAs can change state in the middle of an instruction's execution
  - E.g., IA-32 auto-increment addressing mode
- Others make many changes to CPU state
  - E.g., IA-32 string copy
- Maintaining precise exceptions requires that these changes be reversible

MIPS and Multicycle Operations

- FP Operations
  - Require many more cycles than integer operations
  - May require a variable number of cycles
- The solution: divide the data path
  - Integer
  - FP/Int multiplier
  - FP adder
  - FP/Int divider



MIPS Pipeline with Multicycle Operations



Multicycle Operations in MIPS

- Initiation interval
  - Min cycles between two operations of a given type
- Latency
  - number of cycles after entering EX required to produce a result

Functional Unit	Latency	Initiation Interval
Integer ALU (EX)	0	1
Data Memory (integer and FP)	1	1
FP add	3	1
FP and int. multiply	6	1
FP and int. divide	23	24

Multicycle Operations and Hazards

- Structural hazards
  - Divide unit is not pipelined
  - Two instructions could compete to write one register
- New kinds of data hazards
  - One instruction (ADD) fetched after another (DIV) could write the result register before the earlier one
- More and longer stalls
- More difficult exception handling

Classification of data hazards

- Read After Write (RAW)
  - The most common type
  - A consumer tries to read before the producer writes
- Write After Write (WAW)
  - A later producer tries to write before an earlier producer
- Write After Read (WAR)
  - A later producer writes before an earlier consumer reads
- Note that Read After Read (RAR) is not a hazard

Hazards and Forwarding

- Examples of multicycle pipeline hazards
  - Yellow marks: RAW,
  - Red marks: structural hazards
- Even with full bypassing, the pipeline is stalled for every instruction but the first

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	ME	W	B											
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	ME	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	ME	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	ME

Hazards and Forwarding, Cont'd

- Three instructions compete for the register file.
- Instructions must be stalled so program order is preserved
  - Delaying ADD.D rather than L.D would create a WAW hazard

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	ME	WB					
…		IF	ID	EX	ME	WB										
…			IF	ID	EX	ME	WB									
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	ME	WB					
…					IF	ID	EX	ME	WB							
…						IF	ID	EX	ME	WB						
L.D F2,0(r2)							IF	ID	EX	ME	WB					

Hazard Detection

- *Assuming that all hazard detection is done in ID*
- Three hazard checks must be performed
  - Structural hazards: delay until the required unit is free
  - RAW hazards: delay until the source registers are not listed as pending destinations.
  - WAW hazards: delay until instructions with same destination clear
- More forwarding in longer pipeline
  - From any of Ex/MEM, A4/MEM, M7/MEM, D/MEM, MEM/WB to the input of the ALU
  - Bigger multiplexers

WAW Hazard

- Compilers never generate two writes without an intervening read
  - How is a WAW hazard possible?
- Branches:

```
            BNEZ R1,foo
            DIV.D F0,F2,F4     // moved from fall
            …                  // through into delay slot
            …
      foo:  L.D    F0,something
```

- If the predicted untaken and the branch is taken, L.D will reach WB before DIV.D can complete

Maintaining Precise Exceptions

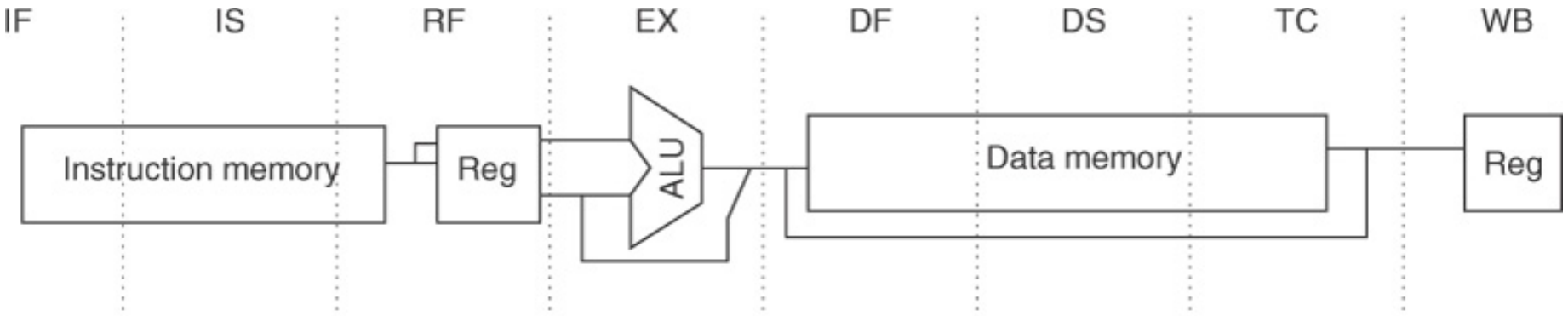
- There are further complications with exceptions
- Example:

```
DIVD.D   F0,F2,F4
ADD.D    F10,F10,F8
SUB.D    F12,F12,F14
```

- SUB.D could cause an imprecise exception
  - DIVD.D is not yet completed
  - ADD.D has completed (out-of-order)
- In this case the ADD.D has destroyed an operand, so the exception cannot be precise (can't restore F10)
- Solutions: imprecise, buffer, software backup, hybrid

Superpipelining

- Subdividing the clock cycles even further
- Example, the MIPS R4000 superpipeline
  - ICache and DCache access are pipelined
  - New accesses can start with each cycle
- Everything we have seen generalizes
  - More hazards, longer delays, more difficult exceptions
  - Hopefully higher throughput



Summary

- Exceptions
  - Stop normal execution to handle exception scenarios
- Precise Exceptions and Pipelines
  - Stop pipeline, save state
  - Handle the exception
  - Restore state and re-execute offending instruction
- Challenges
  - Multiple exceptions
  - Out-of-order exceptions
- Multi-cycle operations
  - New hazards
  - More difficult exceptions

Next Time

- Instruction-level Parallelism
  - Read Chapter 2!