Exceptions and the Memory Hierarchy

CSCI 237: Computer Organization
26th Lecture, Friday, November 7, 2025

Kelly Shaw

Administrative Details

- Lab #5 checkpoint due Tuesday at 11pm
- Read CSAPP 6.2-6.4
- Apply to be a TA by TODAY!

Last Time: The Y86 Pipelined Datapath

- Construction of a pipelined datapath for Y86
  - Control hazards
    - Branch prediction
  - Exceptions

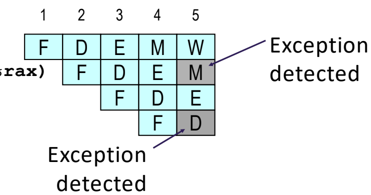
Today: Exceptions and Storage

- Construction of a pipelined datapath for Y86
  - Exceptions
- Storage technologies and trends (Ch 6.1)
  - Memory technologies
- The memory hierarchy (Ch 6.3)
- Locality of reference (Ch 6.2)

Exceptions in Pipeline Processor #1

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```

```
0x000: irmovq $100,%rax
0x00a: rmmovq %rax,0x10000(%rax)
0x014: nop
0x015: .byte 0xFF
```

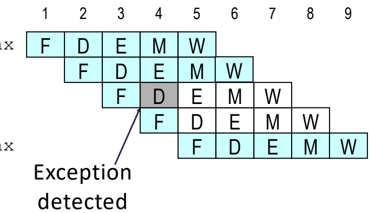


- Desired Behavior
  - `rmmovq` should cause exception
  - Following instructions should have no effect on processor state

Exceptions in Pipeline Processor #2

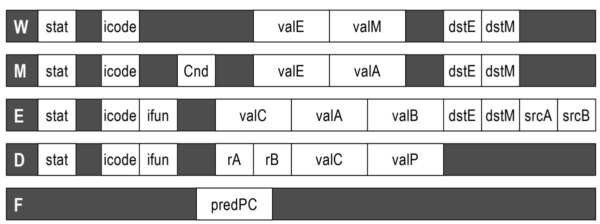
```
0x000:    xorq %rax,%rax    # Set condition codes
0x002:    jne t             # Not taken
0x00b:    irmovq $1,%rax
0x015:    irmovq $2,%rdx
0x01f:    halt
0x020: t: .byte 0xFF        # Target
```

```
0x000:    xorq %rax,%rax
0x002:    jne t
0x020: t: .byte 0xFF
0x???: (I'm lost!)
0x00b:    irmovq $1,%rax
```



- Desired Behavior
  - No exception should occur

Maintaining Exception Ordering

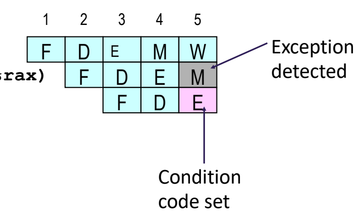


- Add status field to pipeline registers
  - Pass through value from last stage or set if exception detected in stage
  - Values in Y86-64: "AOK", "ADR" (when bad fetch address), "HLT" (halt instruction) or "INS" (illegal instruction)
- Exception triggered only when instruction hits write back

Side Effects in Pipeline Processor

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax)  # invalid address
addq %rax,%rax             # Sets condition codes
```

```
0x000: irmovq $100,%rax
0x00a: rmmovq %rax,0x1000(%rax)
0x014: addq %rax,%rax
```



- Desired Behavior
  - `rmmovq` should cause exception
  - No following instruction should have any effect

Avoiding Side Effects

- Presence of Exception Should Disable State Update
  - Invalid instructions are converted to pipeline bubbles
    - Except have stat indicating exception status
  - Data memory will not write to invalid address
  - Prevent invalid update of condition codes
    - Detect exception in memory stage
    - Disable condition code setting in execute
    - Must happen in same clock cycle
  - Handling exception in final stages
    - When detect exception in memory stage
      - Start injecting bubbles into memory stage on next cycle
    - When detect exception in write-back stage
      - Stall excepting instruction

Rest of Real-Life Exception Handling

- Call Exception Handler
  - Push PC onto stack
    - Either PC of faulting instruction or of next instruction
    - Usually pass through pipeline along with exception status
  - Jump to handler address
    - Usually fixed address
    - Defined as part of ISA

Practice on Your Own

- When we talk about the performance of a pipelined datapath, we talk about CPI, or Cycles Per Instruction, which is the number of cycles it takes to complete each instruction on average. Ideally, CPI=1. Suppose that 20% of our instructions were control instructions and that we mispredicted the target of these control instructions 10% of the time, causing a two wasted cycles (stall cycles) on each misprediction. Assume all other instructions have a CPI of 1. What would be the CPI for this system?

Practice In Class

```
1.    irmovq    $1, %rsi
2.    mrmovq    (%rdi), %r8
3.    irmovq    $4, %r9
4.    subq      %r8, %r9
5.    jne       L1        # taken
6.    addq      %rsi, %rax
7.  L1: rmmovq  %r8, 8(%rdi)
8.    addq      %r8, %rax
```

- Pipeline: Fetch, Decode, **Ex1, Ex2, Mem1, Mem2**, Writeback
- Data forwarding used between Mem2 and Ex1 and between Ex2 and Ex1
  - Also between Mem1 and Ex1
- Control instructions don't determine the next instruction until end of Ex2
- The pipeline uses predict-not-taken branch prediction scheme

Practice in Class

```
1.    irmovq    $1, %rsi
2.    mrmovq   (%rdi), %r8
3.    irmovq    $4, %r9
4.    subq      %r8, %r9
5.    jne        L1         # taken
6.    addq      %rsi, %rax
7.  L1: rmmovq  %r8, 8(%rdi)
8.    addq      %r8, %rax
```

- Figure out RAW dependences that require stalling
- Figure out controls due to control.

14

14

Practice in Class

1.) irmovq $1, %rsi — F D E1 E2 M1 M2 W
2.) mrmovq (%rdi), %r8 — F D E1 E2 M1 M2 W
3.) irmovq $4, %r9 — F D E1 E2 M1 M2 W
4.) subq %r8, %r9 — F D E1 E1 E1 E2 M W
5.) jne L1 — F D D D E1 E2 M W
6.) addq %rsi, %rax — F F F D E1 X X X
7) rmmovq %r8, 8(%rdi) — F D X X X X
6) addq %r8, %rax — F X X X X X
7) rmmovq %r8, 8(%rdi) — F D E1 E2 M1 M2 W
8) addq %r8, %rax — F D E1 E2 M1 M2 W

r8 forwarding

r9

mispredict of jne

I noted in class verbally that there was a forwarding path between M1 and E1

15

15

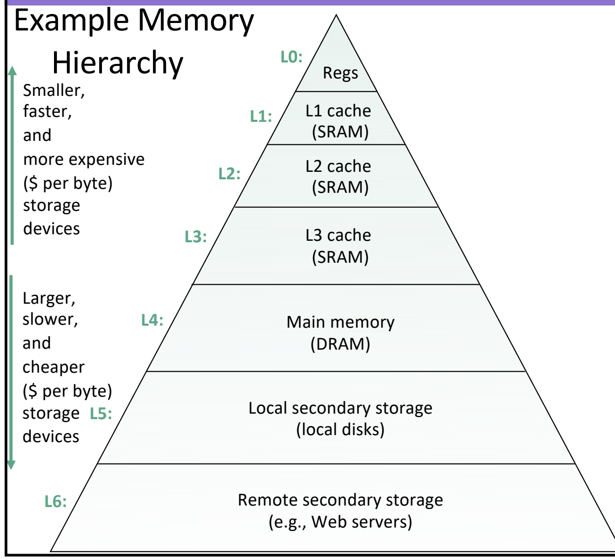
Today: Exceptions and Storage

- Construction of a pipelined datapath for Y86
  - Exceptions
- Storage technologies and trends (Ch 6.1)
  - Memory technologies
- The memory hierarchy (Ch 6.3)
- Locality of reference (Ch 6.2)

17

17

Example Memory Hierarchy



18

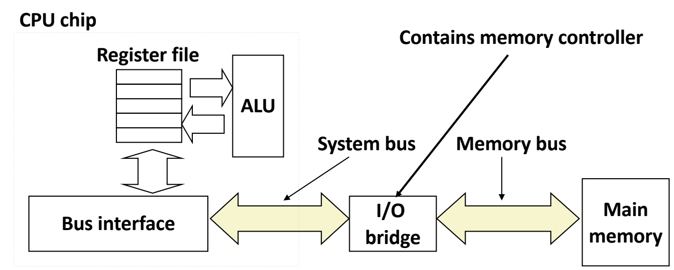
18

Nonvolatile Memories

- DRAM and SRAM are **volatile** memories
  - Lose information if powered off
- Nonvolatile memories retain value even if powered off
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased w/ UV light
  - Electrically eraseable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasings
- Uses for Nonvolatile Memories
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,…)
  - Disk caches

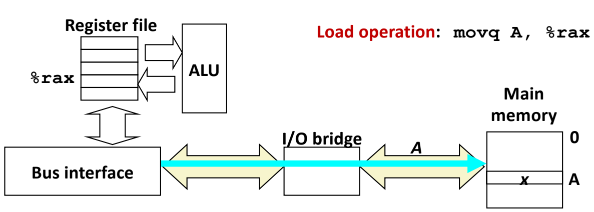
Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



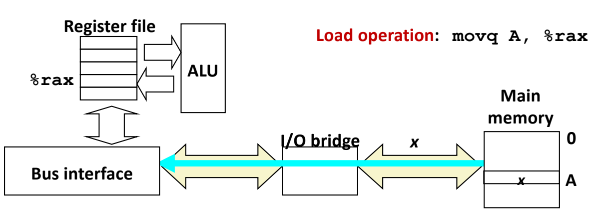
Memory Read Transaction (1)

- CPU places memory address A on the memory bus.



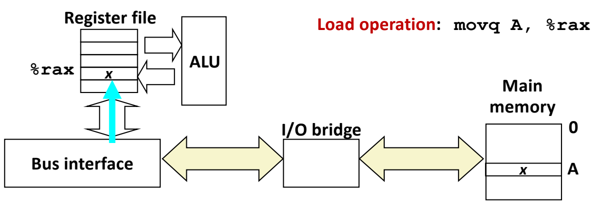
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



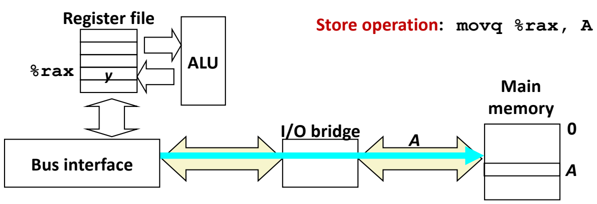
Memory Read Transaction (3)

- CPU reads word x from the bus and copies it into register %rax.



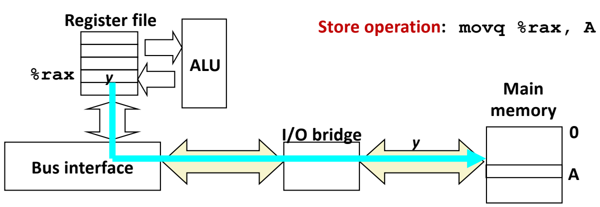
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



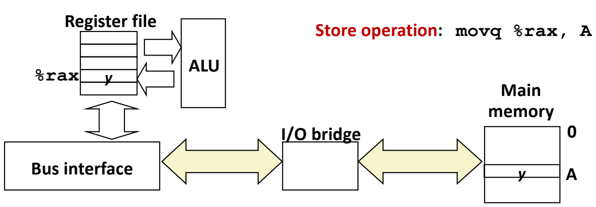
Memory Write Transaction (2)

- CPU places data word y on the bus.



Memory Write Transaction (3)

- Main memory reads data word y from the bus and stores it at address A.



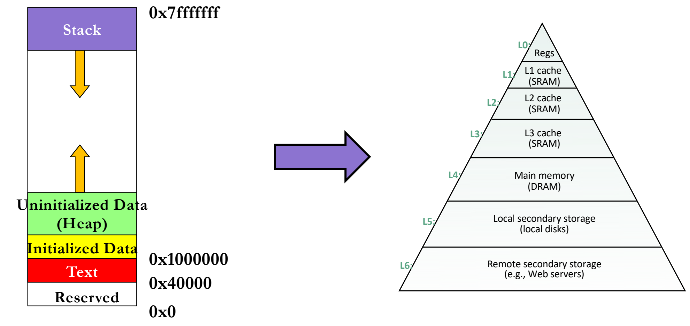
Today: Exceptions and Storage

- Construction of a pipelined datapath for Y86
  - Exceptions
- Storage technologies and trends (Ch 6.1)
  - Memory technologies
- The memory hierarchy (Ch 6.3)
- Locality of reference (Ch 6.2)

27

27

Mapping Program Addresses into Memory Hierarchy

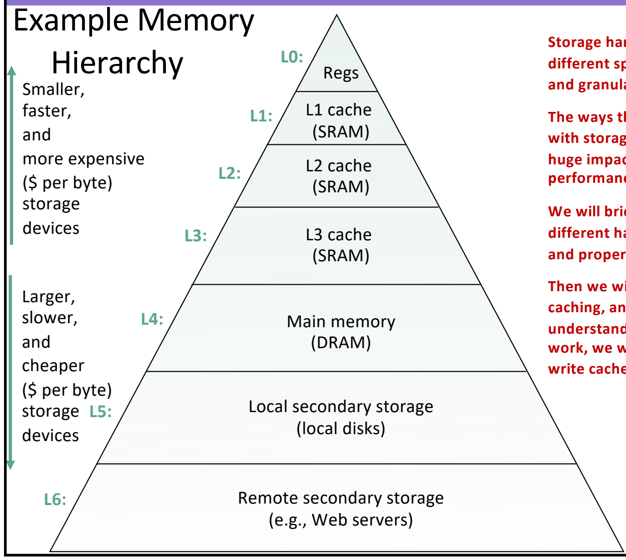


Virtual address space **Memory hierarchy**

28

28

Example Memory Hierarchy



Storage hardware has different speeds, costs ($), and granularities of access.

The ways that we interact with storage hardware has a huge impact on program performance.

We will briefly discuss the different hardware types and properties.

Then we will discuss caching, and once we understand how caches work, we will discuss how to write cache-efficient code.

29

29