Multi-Cycle Exceptions

Exceptions

What are they?  
What do we do about them?

Introduction to pipelining

Why pipelining?  
Why is it difficult?  
How can we do it efficiently?  
Examples

Today’s Menu

Exceptions and Interrupts

Exceptions are “exceptional events” that disrupt the normal flow of a program

Terminology varies between different machines

Examples of Interrupts

User hitting the keyboard
Disk drive asking for attention
Arrival of a network packet

Examples of Exceptions

Divide by zero
Overflow
Page fault

Handling Exceptions and Interrupts

When do we jump to an exception?

Upon detection, invoke the OS to “service the event”

Right when it occurs?  
What about in the middle of executing a multi-cycle instruction  
Difficulty to abort the middle of an instruction
Processor checks for event at the end of every instruction
Processor provides EPC & Cause registers to inform OS of cause

EPC - Exception Program Counter
  Holds PC that the OS should jump to when resuming execution

Cause Register
  Holds bit-encoded cause of the exception

Exceptions and Interrupts

Exception Flow

When an exception (or interrupt) occurs, control is transferred to the OS

When the OS is done, it jumps back to the user program (if it can)

User Process

Event

Exception

Exception processing by exception handler

Exception return (optional)

Operating System

Why This Is Very Messy

You have many instructions in flight

In one of these instructions, a “bad thing” happens, e.g., divide-by-zero

What do we have to do?

We have to deal with this event, since normal program execution is probably now incorrect

But, we have a bunch of instructions in flight

Many of them, but maybe not all of them, need to get killed

Don’t want to kill stuff that is actually correct, and waste that work.

When do we kill them?

NOW – die die die…?

Wait till exception-causing instruction finishes?

Wait till the pipeline empties?

Very very very messy part of real machine design.
Review of Multicycle vs. Single Cycle

- Single cycle implementations have to consider the worst case delay through the datapath to come-up with the cycle time.
- Multicycle implementations have the advantage of using a different number of cycles for executing each instruction.
- In general, the multicycle machine is better than the single cycle machine, but the actual execution time strongly depends on the workload.
- The most widely used machine implementation is neither single cycle, nor multicycle – it’s the pipelined implementation. (Next lecture)

Complete Single-cycle Datapath

Cost of the Single Cycle Architecture

<table>
<thead>
<tr>
<th>Instr Class 1</th>
<th>Instr Class 2</th>
<th>Instr Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our Cycle Time (longest instruction)

Most of the time is wasted!

Multi-cycle Solution

Idea: Let the FASTEST instruction determine clock period

<table>
<thead>
<tr>
<th>Instr Class 1</th>
<th>Instr Class 2</th>
<th>Instr Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Takes 4 cycles
Takes 2 cycles

Less Wasted Time

Multi-cycle Reality

- We are going to go further than allowing the fastest instruction to determine rate
- We are going to break EVERY instruction up into phases
Pipelining

- Multicycle ➔ Pipelining
- Let's build cars

Can we go faster?

Pipelining:

- Production assembly lines
- Henry Ford, Model T, 1908
- Two ways to build a car:
  - Each step takes 1 hour

Non-pipelined: 1 car/4 hours
Pipelining

- Can we go faster?
- Pipelining:
  - Production assembly lines
  - Henry Ford, Model T, 1908
- Two ways to build a car:
  - Each step takes 1 hour

Non-pipelined: 1 car/4 hours
pipelined: 1 car/hour

Can we go faster?
- Pipelining:
  - Production assembly lines
  - Henry Ford, Model T, 1908

Two ways to build a car:
  - Each step takes 1 hour

Analogy: Gasoline Transportation

- Trucking gas from depot to gas station
  - Get the barrels
  - Load them into the truck
  - Drive to the gas station
  - Unload the gas
  - Return for more oil
- Let’s do the math
  - Each truck can carry 5 barrels
  - Can load a truck with 5 barrels in 1 hour
  - Takes each truck 1 day to drive to and from gas station
  - Q: How many barrels per week are delivered?
  - Q: What if I had more trucks?

Looks a Lot Like a Multicycle Processor

- What are the steps
  - Fetch an instruction (Get the barrels)
  - Decode the instruction (Load them into the truck)
  - ALU OP (Drive to the gas station)
  - Memory Access (Unload the gas)
  - Write-back (Return for more oil)

Business 201

- Roll the barrels down the road
  - Big fire hazard - probably will not meet OSHA standards

US Occupational Safety and Health Administration

Trucking vs. Pipelines

- Trucks
  - Each truck can carry 5 barrels
  - Can load a truck with 5 barrels in 1 hour
  - Takes 1 day to drive to and from gas station
  - Lots of TIME when loading area, gas station, and pieces of the road are unused
  - Unless you have lots of trucks
- Pipelines
  - Can accept 1 barrel every hour
  - Resources (loading area, gas station, pipelines) are always in use
  - As long as you can keep your pipeline full (e.g., have enough barrels)

Business 201

- Build a pipeline
  - Will meet OSHA standards
  - Might make the environmentalists angry
  - Now let’s do the math
    - Pipeline can accept 1 barrel every hour
    - Q: How many barrels get delivered to the gas station per day?
    - Q: How many barrels are “in-flight” at any moment?
**Big Idea: Pipeline Concurrency**

This computation is "too long"

Latches, called "Pipeline registers" break up computation into stages

Pipelined version, 5 pipe stages

~20 ns

**Big Idea: It’s Faster**

I can "launch" a new computation every 100ns in this structure

Pipelined version, 5 pipe stages:

I can launch a new computation every 20ns in pipelined structure

~20 ns

**Pipelining: Implementation Issues**

- What prevents us from just doing a zillion pipe stages?
  - Some computations just won't divide into any finer (shorter in time) logical implementations
  - Ultimately, often comes down to circuit design issues

- 5 stages: OK

- ~20 ns

- 50 stages: nope, sorry

- ~2 ns

**Pipelining: Implementation Issues**

- What prevents us from just doing a zillion pipe stages?
  - Those latches are NOT free, they take up area, and there is a real delay to go THROUGH the latch itself
  - In modern, deep pipeline (10-20 stages), this is a real effect
  - Typically see logic depths in one pipe stage of 10-20 gates

- At these speeds, and with this few levels of logic, latch delay is important

**Remember the ARM big.LITTLE Idea?**

**How Many Pipeline Stages?**

- E.g., Intel
  - Pentium 4: over 20 stages
  - More than 120 instructions in flight
  - High clock frequency (>3GHz)
  - High IPC (Instructions per Cycle)

- Too many stages:
  - Lots of complications
  - Should take care of possible dependencies among in-flight instructions
  - Control logic is huge
  - Too little work per stage, too high a branch miss-prediction penalty → bad performance
### Performance of Pipelined Systems

- **Unpipelined**
  - Throughput: 1 per 5 cycles
  - Time required: 5 cycles

- **Pipelined**
  - Throughput: 1 per cycle
  - Pipeline stage ideal speedup only if we can keep the pipeline full
  - Time: sequential Pipeline Depth

- **Ideally,** Speedup\(_{\text{pipeline}}\) = Time\(_{\text{sequential Pipeline Depth}}\)

### MIPS Pipeline Stages

- **Stage 1:** Instruction Fetch (IF)
- **Stage 2:** Instruction Decode (ID)
- **Stage 3:** Execute (EX)
- **Stage 4:** Memory Access (MEM)
- **Stage 5:** Write Back (to register file) (WB)

### 5-stage Version of MIPS Datapath

- **STAGE 1** (Instruction Fetch)
- **STAGE 2** (Decode)
- **STAGE 3** (ALU Execute)
- **STAGE 4** (Memory Access)
- **STAGE 5** (Writeback)

### Complete 5 Stage Pipeline (Drawn Smaller)

- **IF/ID**
- **ID/EX**
- **EX/MEM**
- **MEM/WB**

### Flow of Instructions Through Pipeline

- **Program Execution**
  - LW R1, 100(R0)
  - LW R2, 200(R0)
  - LW R3, 300(R0)

- **Time**
  - Clock Cycle 1
  - Clock Cycle 2
  - Clock Cycle 3
  - Clock Cycle 4
  - Clock Cycle 5
  - Clock Cycle 6
  - Clock Cycle 7

- **In cycle 4 we have 3 instructions “in-flight”:**
  - Inst 1 is accessing the memory (DM)
  - Inst 2 is using the ALU (EX)
  - Inst 3 is accessing the register file (ID)

### Stage 1 - IF (Instruction Fetch)

- **Instruction Fetch**
- **Instruction Memory (RAM)**
- **IF/ID**
- **IF/ID**
- **IF/ID**
- **IF/ID**
- **IF/ID**
- **IF/ID**
Stage 2 - ID (Instruction Decode)

Stage 3 - EX (Execution)

Stage 4 - MEM (Memory)

Stage 5 - WB (Write Back)

New Complications

- **The good news**
  - Multiple instructions are running at the same time, thru the datapath
  - This works because each stage of pipeline is isolated by latches
  - So, in the best of all possible worlds, N stage pipe has N instructions flowing thru it, speedup is close to N.

- **The bad news**
  - Instructions interfere with each other
  - Common name for these: **conflicts**

- **Why?**
  - Different instructions "in flight" thru data path at same time
  - Different instructions might want to use the same piece of hardware in the datapath at the same time (i.e., in same clock cycle)
  - These conflicts — contention for an over-used resource — are the source of endless grief in pipeline design

Good News: >1 Instruction “In Flight” in Pipe
Instruction Interference in a Pipe

- In its most basic form, it's about contention for a resource
  - 2 instructions want to "use" a piece of hardware in the pipe
  - There's only one of those in the pipe, maybe it can't "service" the requirements of more than one instruction at a time

Sometimes, You Can Redesign the Resource

- In this particular case...
  - The problem is one instruction reads register file
  - Solution: allow write-then-read in one clock cycle ("double pump")

But..This Case Still Screws Up

- Basic structure
  - An instruction in flight wants to use a data value that's not "done" yet
  - "Done" means "it's been computed" and "it's located where I would normally expect to go look in the pipe hardware to find it"

Another Conflict: Data Hazards

- Basic cause
  - You are used to assuming a purely sequential model of instruction execution
  - Instruction N finishes before instruction N+k, for k > 1
  - Nope, sorry -- not true any more in a pipeline
  - There are dependencies now between "nearby" instructions ("near" in sequential order of fetch from memory)

- Consequence
  - Data hazards -- instructions want data values that are not done yet, or in the right place yet
This Data Hazard, Revisited

In this particular case...
- R10 value is not computed or returned to register file when later instruction wants to use it as an input.

Double pumping reg file doesn’t help here; later instruction needs R10 2 clock cycles before it’s been computed & stored back. Oops...

Oops...

Coping with Data Hazards

- What do you do?
  - Sometimes the dumb-sounding answer is right

- Hypothesis:
  - It is BAD when certain instructions overlap in time in certain patterns in our 5 stage MIPS pipeline

- Proposed solution
  - Don’t let them overlap like this…?
  - Right - that is one solution

- Mechanics
  - Don’t let the instruction flow thru the pipe
  - In particular, don’t let it WRITE any bits anywhere in the pipe hardware that represents REAL CPU state (e.g., register file, memory)
  - Name for this operation: PIPELINE STALL

Coping with Data Hazards: Example

- Program Execution
  - Time
    - Clock Cycle 1: ADD R10, R11, R12
    - Clock Cycle 2: ADD R12, R10, R11
    - Clock Cycle 3: ADD R11, R10, R12

Solution 1: Stall

- Program Execution
  - Time
    - Clock Cycle 1: ADD R10, R11, R12
    - Clock Cycle 2: ADD R12, R10, R11
    - Clock Cycle 3: ADD R11, R10, R12

Mechanically: How Do We Stall?

- Add extra hardware to detect stall situations
  - Watches the instruction field bits
  - Looks for “read versus write” conflicts in particular pipe stages
  - Basically, a bunch of careful “case logic”

- Add extra hardware to push bubbles thru pipe
  - Actually, relatively easy
  - Can just let the instruction you want to stall GO FORWARD thru the pipe...
  - ...but, TURN OFF the bits that allow any results to get written into the machine state
  - So, the instruction “executes” (it does the work), but doesn’t “save”

“If an instruction executes in the middle of forest, but no registers are around to save the results...did it really execute?” (No.)

Recall the Registers Between Pipeline Stages
Recall What an Instruction Looks Like

- add R8, R17, R18
  - is stored in binary format as
    
    | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
    |----|----|----|----|----|----|----|----|----|----|----|----|
    | 00000000 | 110010 | 001100 | 000000 |
  - MIPS lays out instructions into
    
    - op: operation of the instruction
    - Rs: first register source operand
    - rt: second register source operand
    - rd: register destination operand
    - shamt: shift amount
    - funct: function (select type of operation)

We gotta watch these reg op fields

Example

```
sub R2, R1, R3
Rd = R2
Rs = R1
Rt = R3
```

```
and R12, R2, R5
Rd = R12
Rs = R2
Rt = R5
```

```
or R13, R6, R2
Rd = R13
Rs = R6
Rt = R2
```

```
add R14, R2, R2
Rd = R14
Rs = R2
Rt = R2
```

```
sw R15, 100(R2)
Rd = R15
Rs = R2
Rt = XX
```

Example

```
sub R2, R1, R3
Rd = R2
Rs = R1
Rt = R3
```

```
and R12, R2, R5
Rd = R12
Rs = R2
Rt = R5
```

```
or R13, R6, R2
Rd = R13
Rs = R6
Rt = R2
```

```
add R14, R2, R2
Rd = R14
Rs = R2
Rt = R2
```

```
sw R15, 100(R2)
Rd = R15
Rs = R2
Rt = XX
```

SUB-AND Hazard
- EX/MEM. RegisterRd == ID/EX. RegisterRs
  
SUB-OR Hazard
- MEM/WB. RegisterRd == ID/EX. RegisterRt

Interactions (real or not) can be tricky
- Example: do instruction #1 (sub) and #4 (add) interact, conflict?
  
No Dependence Between #1 and #4

In this case, double pumped reg file makes it ok...

How Else Could We Stall the Pipeline?

- Compiler can insert nops
Or, The Hardware Can Simulate NOPS

Next lecture

- How to fix the pipeline to avoid (most) dependency problems ...