Design of Pipelined MIPS Processor

Sept. 24 & 26, 1997

Topics

- Instruction processing
- Principles of pipelining
- Inserting pipe registers
- Data Hazards
- Control Hazards
- Exceptions
MIPS architecture subset

R-type instructions (add, sub, and, or, slt): \( rd \leftarrow rs \text{ funct } rt \)

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-11</td>
<td>10-6</td>
</tr>
</tbody>
</table>

RI-type instructions (addiu): \( rt \leftarrow rs \text{ funct } I16 \)

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>I16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
</tr>
</tbody>
</table>

Load: \( rt \leftarrow \text{Mem}[rs + I16] \)  
Store: \( \text{Mem}[rs + I16] \leftarrow rt \)

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>I16</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 or 43</td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
</tr>
</tbody>
</table>

Branch equal: \( PC \leftarrow (rs == rt) ? PC + 4 + I16 <<2 : PC + 4 \)

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>I16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
</tr>
</tbody>
</table>
Datapath

**IF**
- instruction fetch

**ID**
- instruction decode/register fetch

**EX**
- execute/address calculation

**MEM**
- memory access

**WB**
- write back

[Diagram of datapath with various components and flows labeled accordingly.]
R-type instructions

IF: Instruction fetch
- IR <-- IMemory[PC]
- PC <-- PC + 4

ID: Instruction decode/register fetch
- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

Ex: Execute
- ALUOutput <-- A op B

MEM: Memory
- nop

WB: Write back
- Register[IR[15:11]] <-- ALUOutput
Load instruction

IF: Instruction fetch
  • IR <-- IMemory[PC]
  • PC <-- PC + 4

ID: Instruction decode/register fetch
  • A <-- Register[IR[25:21]]
  • B <-- Register[IR[20:16]]

Ex: Execute
  • ALUOutput <-- A + SignExtend(IR[15:0])

MEM: Memory
  • Mem-Data <-- DMemory[ALUOutput]

WB: Write back
  • Register[IR[20:16]] <-- Mem-Data
Store instruction

**IF: Instruction fetch**
- IR <-- IMemory[PC]
- PC <-- PC + 4

**ID: Instruction decode/register fetch**
- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

**Ex: Execute**
- ALUOutput <-- A + SignExtend(IR[15:0])

**MEM: Memory**
- DMemory[ALUOutput] <-- B

**WB: Write back**
- nop
Branch on equal

**IF: Instruction fetch**
- IR <-- IMemory[PC]
- PC <-- PC + 4

**ID: Instruction decode/register fetch**
- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

**Ex: Execute**
- Target <-- PC + SignExtend(IR[15:0]) << 2
- Z <-- (A - B == 0)

**MEM: Memory**
- If (Z) PC <-- target

**WB: Write back**
- nop

*Is this a delayed branch?*
Pipelining Basics

Unpipelined System

- One operation must complete before next can begin
- Operations spaced 33ns apart

Delay = 33ns
Throughput = 30MHz

Time

Op1 → Op2 → Op3 → …
3 Stage Pipelining

- Space operations 13ns apart
- 3 operations occur simultaneously

Delay = 39ns
Throughput = 77MHz
Limitation: Nonuniform Pipelining

- Throughput limited by slowest stage
- Must attempt to balance stages

Delay = 39ns
Throughput = 55MHz
Limitation: Deep Pipelines

- Diminishing returns as add more pipeline stages
- Register delays become limiting factor
  - Increased latency
  - Small throughput gains

Delay = 48ns, Throughput = 128MHz
Limitation: Sequential Dependencies

• Op4 gets result from Op1!
• *Pipeline Hazard*

![Diagram showing sequential dependencies](image)

- Time
- Op1
- Op2
- Op3
- Op4
- Clock
- Comb. Logic
- REG
- Comb. Logic
- REG
- Comb. Logic
- REG
- Comb. Logic
- REG

CS 740 F'97
Pipelined datapath

Diagram showing the pipeline stages: IF/ID, ID/EX, EX/MEM, MEM/WB. The diagram includes components such as PC, MUX, Instruction mem, Registers, ALU, and MUX with control signals. Instructions and data flow through these stages, with specific paths labeled for data and control signals.
Pipeline Structure

Notes
- Each stage consists of operate logic connecting pipe registers
- WB logic merged into ID
- Additional paths required for forwarding
Pipe Register

**Operation**
- Current State stays constant while Next State being updated
- Update involves transferring Next State to Current
Pipeline Stage

Operation

- Computes next state based on current
  - From/to one or more pipe registers
- May have embedded memory elements
  - Low level timing signals control their operation during clock cycle
  - Writes based on current pipe register state
  - Reads supply values for Next state
MIPS Simulator

Features
• Based on MIPS subset
  – Code generated by dis
  – Hexadecimal instruction code
• Executable available
  – HOME/public/sim/solve_tk

Demo Programs
• HOME/public/sim/solve_tk/demos
Reg-Reg Instructions

R-type instructions (add, sub, and, or, slt): rd <-- rs funct rt

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31-26</td>
<td>25-21</td>
<td>20-16</td>
<td>15-11</td>
<td>10-6</td>
</tr>
</tbody>
</table>

Diagram of the pipeline stages and data flow for an R-type instruction.
Simulator ALU Example

IF
- Fetch instruction

ID
- Fetch operands

EX
- Compute ALU result

MEM
- Nothing

WB
- Store result in destination reg.

0x0: 24020003  li  r2,3
0x4: 24030004  li  r3,4
0x8: 00000000  nop
0xc: 00000000  nop
0x10:00432021  addu  r4,r2,r3  # --> 7
0x14:00000000  nop
0x18:0000000d  break  0
0x1c:00000000  nop

.demo1.O

.demo1.s

.set noreorder
.addiu $2, $0, 3
.addiu $3, $0, 4
.nop
.nop
.addu $4, $2, $3
.nop
.break 0
.set reorder

CS 740 F'97
Simulator Store/Load Examples

**IF**
- Fetch instruction

**ID**
- Get addr reg
- Store: Get data

**EX**
- Compute EA

**MEM**
- Load: Read
- Store: Write

**WB**
- Load: Update reg.

```assembly
demo2.o
0x0: 24020003 li r2,3 # Store/Load -->3
0x4: 24030004 li r3,4 # --> 4
0x8: 00000000 nop
0xc: 00000000 nop
0x10: ac430005 sw r3,5(r2) # 4 at 8
0x14: 00000000 nop
0x18: 00000000 nop
0x1c: 8c640004 lw r4,4(r3) # --> 4
0x20: 00000000 nop
0x24: 00000000d break 0
0x28: 00000000 nop
0x2c: 00000000 nop
```
Simulator Branch Examples

**IF**
- Fetch instruction

**ID**
- Fetch operands

**EX**
- Subtract operands
  - test if 0
- Compute target

**MEM**
- Taken: Update PC to target

**WB**
- Nothing

---

**demo3.O**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0:</td>
<td>li r2,3</td>
<td></td>
</tr>
<tr>
<td>0x4:</td>
<td>li r3,4</td>
<td></td>
</tr>
<tr>
<td>0x8:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0xc:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x10:</td>
<td>beq r2,r3,0x34 # Don't take</td>
<td></td>
</tr>
<tr>
<td>0x14:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x18:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x1c:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x20:</td>
<td>bne r2,r3,0x34 # Take</td>
<td></td>
</tr>
<tr>
<td>0x24:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x28:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x2c:</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>0x30:</td>
<td>addu r4,r2,r3 # skip</td>
<td></td>
</tr>
<tr>
<td>0x34:</td>
<td>addu r4,r3,r3 # Target</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Hazards in MIPS Pipeline

Problem

- Registers read in ID, and written in WB
- Must resolve conflict between instructions competing for register array
  - Generally do write back in first half of cycle, read in second
- But what about intervening instructions?
- E.g., suppose initially $2$ is zero:

```
$2
$3
$4
$5
$6
```

```
addiu $2, $0, 63
addiu $3, $2, 0
addiu $4, $2, 0
addiu $5, $2, 0
addiu $6, $2, 0
```
Simulator Data Hazard Example

Operation
- Read in ID
- Write in WB
- Write-before-read register file

```
0x0: 2402003f li r2,63
0x4: 00401821 move r3,r2 # --> 0x3F?
0x8: 00402021 move r4,r2 # --> 0x3f?
0xc: 00402821 move r5,r2 # --> 0x3f?
0x10:00403021 move r6,r2 # --> 0x3f?
0x14:00000000 nop
0x18:0000000d break 0
0x1c:00000000 nop
demo4.O
```
Handling Hazards by Stalling

Idea

- Delay instruction until hazard eliminated
- Put “bubble” into pipeline
  - Dynamically generated NOP

Pipe Register Operation

- Normally transfer next state to current
- “Stall” indicates that current state should not be changed
- “Bubble” indicates that current state should be set to 0
  - Stage logic designed so that 0 is like NOP
  - [Other conventions possible]
Stall Control

Stall Logic

- Determines which stages to stall or bubble on next update
Observations on Stalling

Good

• Relatively simple hardware
• Only penalizes performance when hazard exists

Bad

• As if placed NOP’s in code
  – Except that does not waste instruction memory

Reality

• Some problems can only be dealt with by stalling
• E.g., instruction cache miss
  – Stall PC, bubble IF/ID
• Data cache miss
  – Stall PC, IF/ID, ED/EX, EX/MEM, bubble MEM/WB
Forwarding (Bypassing)

Observation
- ALU data generated at end of EX
  - Steps through pipe until WB
- ALU data consumed at beginning of EX

Idea
- Expedite passing of previous instruction result to ALU
- By adding extra data pathways and control
Forwarding for ALU-ALU Hazard

Program:

0x18: \( \text{add } $2, $1, $2 \)

0x1c: \( \text{add } $1, $1, $2 \)
Some Hazards with Loads & Stores

<table>
<thead>
<tr>
<th></th>
<th>Data Generated by Load</th>
<th>Data Generated by Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load-ALU</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $1, 8($2)</td>
<td></td>
<td>sw $1, 8($2)</td>
</tr>
<tr>
<td>add $2, $1, $2</td>
<td></td>
<td>lw $3, 8($2)</td>
</tr>
<tr>
<td><strong>Load-Store Data</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $1, 8($2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $1, 12($2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load-Store (or Load) Addr.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $1, 8($2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $1, 12($1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis of Data Transfers

Data Sources
  • Available after EX
    – ALU Result Reg-Reg Result
  • Available after MEM
    – Read Data Load result
    – ALU Data Reg-Reg Result passing through MEM stage

Data Destinations
  • ALU A input Need in EX
    – Reg-Reg Operand
    – Load/Store Base
  • ALU B input Need in EX
    – Reg-Reg Operand
  • Write Data Need in MEM
    – Store Data
Complete Bypassing
Simulator Data Hazard Examples

- demo5.O

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Register(s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>2402003f li</td>
<td>r2</td>
<td>63 → 0x3F</td>
</tr>
<tr>
<td>0x4</td>
<td>00401821 move</td>
<td>r3, r2</td>
<td>EX-EX 0x3F</td>
</tr>
<tr>
<td>0x8</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xc</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>2402000f li</td>
<td>r2</td>
<td>15 → 0xF</td>
</tr>
<tr>
<td>0x14</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>00401821 move</td>
<td>r3, r2</td>
<td>MEM-EX 0xF</td>
</tr>
<tr>
<td>0x1c</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>2402000c li</td>
<td>r2</td>
<td>12 → 0xC</td>
</tr>
<tr>
<td>0x24</td>
<td>24020010 li</td>
<td>r2</td>
<td>16 → 0x10</td>
</tr>
<tr>
<td>0x28</td>
<td>ac430000 sw</td>
<td>r3, 0(r2)</td>
<td>EX-EX 0xF at 0x10</td>
</tr>
<tr>
<td>0x2c</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td>8c440000 lw</td>
<td>r4, 0(r2)</td>
<td>0x0F</td>
</tr>
<tr>
<td>0x34</td>
<td>00822821 addu</td>
<td>r5, r4, r2</td>
<td>Stall 0x1F</td>
</tr>
<tr>
<td>0x38</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3c</td>
<td>00000000d break</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x44</td>
<td>00000000 nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Impact of Forwarding

Single Remaining Hazard Class
  • Load followed by ALU operation
    – Including address calculation

MIPS I Architecture
  • Programmer may not put instruction after load that uses loaded register
    – Not even allowed to assume it will be old value
    – “Load delay slot”
  • Can force with .set noreorder

MIPS II Architecture
  • No restriction on programs
  • Stall following instruction one cycle
  • Then pick up with MEM/EX bypass

Load-ALU

```
lw $1, 8($2)
add $2, $1, $2
```

Load-Store (or Load) Addr.

```
lw $1, 8($2)
lw $4, 12($1)
```
Methodology for characterizing and Enumerating Data Hazards

The space of data hazards (from a program-centric point of view) can be characterized by 3 independent axes:

5 possible write regs (axis 1):
   R.rd, RI.rt, LW.rt, JAL.$31, JALR.rd

10 possible read regs (axis 2):
   R.rs, R.rt, RI.rs, RI.rs, SW.rs, SW.rt
   Bx.rs, Bx.rt, JR.rs, JALR.rs

A dependent read can be a distance of either 1 or 2 from the corresponding write (axis 3):

\[
\text{distance 2 hazard:} \quad \text{addiu } \$2, \$0, 63 \quad \text{distance 1 hazard:} \quad \text{addu } \$3, \$0, \$2
\]
\[
R.rd/R.rs/2 \quad \text{addu } \$4, \$2, \$0 \\
R.rd/R.rt/1
\]
## Enumerating data hazards

### distance = 1

<table>
<thead>
<tr>
<th>writes</th>
<th>R.rs</th>
<th>R.rt</th>
<th>RI.rs</th>
<th>LW.rs</th>
<th>SW.rs</th>
<th>SW.rt</th>
<th>Bx.rs</th>
<th>Bx.rt</th>
<th>JR.rs</th>
<th>JALR.rs</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI.rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW.rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL.$31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>JALR.rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### distance = 2

<table>
<thead>
<tr>
<th>writes</th>
<th>R.rs</th>
<th>R.rt</th>
<th>RI.rs</th>
<th>LW.rs</th>
<th>SW.rs</th>
<th>SW.rt</th>
<th>Bx.rs</th>
<th>Bx.rt</th>
<th>JR.rs</th>
<th>JALR.rs</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI.rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW.rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAL.$31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR.rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simulator Microtest Example

Tests for single failure mode
- ALU Rd --> ALUI Rs, dist 1

- Hits break 0 when error
- Jumps to break 1 when OK
  - Grep for ERROR or break 0

```
0x0: 24020010    li   r2,16
0x4: 00000000    nop
0x8: 00000000    nop
0xc: 00022821    addu  r5,r0,r2
0x10: 00a00821   move  r1,r5
0x14: 00000000    nop
0x18: 00000000    nop
0x1c: 24040010    li   r4,16
0x20: 00000000    nop
0x24: 00000000    nop
0x28: 1024000b    beq  r1,r4,0x58
0x2c: 00000000    nop
0x30: 00000000    nop
0x34: 00000000    nop
0x38: 00000000    nop
0x3c: 00000000    nop
0x40: 0000000d    break 0
0x44: 00000000    nop
```

```
0x48: 00000000    nop
0x4c: 00000000    nop
0x50: 00000000    nop
0x54: 00000000    nop
0x58: 001000d     break 1
0x5c: 00000000    nop
0x60: 00000000    nop
0x64: 00000000    nop
0x68: 00000000    nop
0x6c: 00000000    nop
```
Pipelined datapath

What happens with a branch?
Control Hazards in MIPS Pipeline

Problem

- Instruction fetched in IF, branch condition set in MEM
- When does branch take effect?
- E.g.: assume initially that all registers = 0

```
beq $0, $0, target
addiu $2, $0, 63
addiu $3, $0, 63
addiu $4, $0, 63
addiu $5, $0, 63
target: addiu $6, $0, 63
```
Branch Example

Assume
- All registers initially 0

Desired Behavior
- Take branch at 0x00
- Execute delay slot 0x04
- Execute target 0x18
  - PC + 4 + I16<<2
  - PC = 0x00
  - I16 = 5

Branch Code (demo8.O)

```
0x00: BEQ $0, $0, 5
0x04: li $2, 63  # Slot
0x08: li $3, 63  # Xtra1
0x0c: li $4, 63  # Xtra2
0x10: li $5, 63  # Xtra3
0x14: nop
0x18: li $6, 63  # Target
```
Stall Until Resolve Branch

- Detect when branch in stages ID or EX
- Stop fetching until resolve
- Delay slot instruction just before target
Branch Not Taken Example

Assume
• All registers initially 0

Desired Behavior
• Execute entire sequence

Effect of Stalling
• Wastes 2 cycles waiting for branch decision

<table>
<thead>
<tr>
<th>Branch Code (demo9.O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00: <strong>BNE</strong> $0, $0, 5</td>
</tr>
<tr>
<td>0x04: li $2, 63 # Slot</td>
</tr>
<tr>
<td>0x08: li $3, 63 # Xtra1</td>
</tr>
<tr>
<td>0x0c: li $4, 63 # Xtra2</td>
</tr>
<tr>
<td>0x10: li $5, 63 # Xtra3</td>
</tr>
<tr>
<td>0x14: nop</td>
</tr>
<tr>
<td>0x18: li $6, 63 # Target</td>
</tr>
</tbody>
</table>
Fetch & Cancel When Taken

- Instruction does not cause any updates until MEM or WB stages
- Instruction can be “cancelled” from pipe up through EX stage
  - Replace with bubble

Strategy
- Continue fetching under assumption that branch not taken
- If decide to take branch, cancel undesired ones
Branch Prediction Analysis

Our Scheme Implements “Predict Not Taken”

• But 67% of branches are taken
• Would give \( \text{CPI} \geq 1.0 + 0.16 \times 0.67 \times 2.0 = 1.22 \)

Alternative Schemes

• Predict taken
  – Would be hard to squeeze into our pipeline
    » Can’t compute target until ID
  – In principle, would give \( \text{CPI} \geq 1.11 \)

• Backwards taken, forwards not taken
  – Predict based on sign of I16
  – Exploits fact that loops usually closed with backward branches
How Does Real MIPS Handle Branches?

Avoids any Branch Penalty!
- Do not treat PC as pipe register
  - Loaded at beginning of IF
- Compute branch target & branch condition in ID
  - Can fetch at target in following cycle
Exceptions

An exception is a transfer of control to the OS in response to some event (i.e. change in processor state)
Internal (CPU) exceptions

Internal exceptions occur as a result of events generated by executing instructions.

Execution of a SYSCALL instruction.
  • allows a program to ask for OS services (e.g., timer updates)

Execution of a BREAK instruction
  • used by debuggers

Errors during instruction execution
  • arithmetic overflow, address error, parity error, undefined instruction

Events that require OS intervention
  • virtual memory page fault
External (I/O) exceptions

External exceptions occur as a result of events generated by devices external to the processor.

I/O interrupts
- hitting ^C at the keyboard
- arrival of a packet
- arrival of a disk sector

Hard reset interrupt
- hitting the reset button

Soft reset interrupt
- hitting ctl-alt-delete on a PC
Exception handling (hardware tasks)

Recognize event(s)

Associate one event with one instruction.
- external event: pick any instruction
- multiple internal events: typically choose the earliest instruction.
- multiple external events: prioritize
- multiple internal and external events: prioritize

Create Clean Break in Instruction Stream
- Complete all instructions before excepting instruction
- Abort excepting and all following instructions
Exception handling (hardware tasks)

Set status registers

- **EPC register**: exception program counter
  - external exception: address of instruction about to be executed.
  - internal exception (not in the delay slot): address of instruction causing the exception
  - internal exception (in delay slot): address of preceding branch or jump
- **Cause register**
  - records the event that caused the exception
- **Others**
  - 15 other registers! Which get set depends on CPU and exception type

Disable interrupts and switch to kernel mode
Jump to common exception handler location
Exception handling (software tasks)

Deal with event

(Optionally) Resume execution
  • using special \texttt{rfe} (return from exception) instruction
  • similar to \texttt{jr} instruction, but restores processor to user mode as a side effect.

Where to resume execution?
  • usually re-execute the instruction causing exception
  • unless instruction was in branch delay slot, in which case re-execute the branch instruction immediately preceding the delay slot
    – EPC points here anyhow
  • Also, what about \texttt{syscall} or \texttt{break}?
Example: Integer overflow (EX)

user code
sub $11, $2, $4
and $12, $2, $5
or $13, $2, $6
add $1, $2, $1
slt $15, $6, $7
lw $16, 50($7)

handler code
sw $25, 1000($0)
...

Overflow detected here
flush these instructions
start handler code
the or instruction completes

Overflow

overflow

and
or
add
slt
lw
sw

Contents
Exception Handling In pMIPS Simulator

Relevant Pipeline State

• Address of instruction in pipe stage (SPC)
• Exception condition (EXC)
  – Set in stage when problem encountered
    » IF for fetch problems, EX for instr. problems, MEM for data probs.
  – Triggers special action once hits WB
MIPS Exception Examples

• In directory $\text{HOME/public/sim/exc}$

<table>
<thead>
<tr>
<th>Illegal Instruction (exc1.O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0: <code>srl r3, r2, 4</code> # Unimplemented</td>
</tr>
<tr>
<td>0x4: <code>li r2, 4</code> # Should cancel</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Illegal Instruction followed by store (exc2.O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0: <code>li r2, 15</code> # --&gt; 0xf</td>
</tr>
<tr>
<td>0x4: <code>srl r3, r2, 4</code> # Unimplemented</td>
</tr>
<tr>
<td>0x8: <code>sw r2, 4(r0)</code> # Should cancel</td>
</tr>
</tbody>
</table>
More MIPS Exception Examples

Good instruction in delay slot (exc3.O)

```
0x0: li   r3, 3
0x4: jr   r3    # Bad address
0x8: li   r3, 5  # Should execute
```

Bad instruction in delay slot (exc4.O)

```
0x0: li   r3, 3
0x4: jr   r3    # Bad address
0x8: sw   r3, 0(r3)  # Bad address
```
Final MIPS Exception Example

Avoiding false alarms (exc5.O)

0x0: beq r2,r2,0x10
0x4: nop
0x8: srl r2,r4,4 # Should cancel
0xc: nop
0x10: li r2,1    # --> 1
0x14: break 0
Implementation Features

Correct

- Detects excepting instruction (exc4.O)
  - Furthest one down pipeline = Earliest one in program order
- Completes all preceding instructions (exc3.O)
- Usually aborts excepting instruction & beyond (exc1.O)
- Prioritizes exception conditions
  - Earliest stage where instruction ran into problems
- Avoids false alarms (exc5.O)
  - Problematic instructions that get canceled anyhow

Shortcomings

- Store following excepting instruction (exc2.O)
- EPC ≠ SPC for delay slot instruction