Appendix A

Pipelining: Basic and Intermediate Concepts

Overview

- Basics of Pipelining
- Pipeline Hazards
- Pipeline Implementation
- Pipelining + Exceptions
- Pipeline to handle Multicycle Operations

Unpipelined Execution of 3 LD Instructions

- Assumed are the following delays: Memory access = 2 nsec, ALU operation = 2 nsec, Register file access = 1 nsec
- Assuming 2 nsec clock cycle time (i.e. 500 MHz clock), every ld instruction needs 4 clock cycles (i.e. 8 nsec) to execute.
- The total time to execute this sequence is 12 clock cycles (i.e. 24 nsec). CPI = 12 cycles/3 instructions = 4 cycles / instruction

Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads
**Key Definitions**

Pipelining is a key implementation technique used to build fast processors. It allows the execution of multiple instructions to overlap in time.

A pipeline within a processor is similar to a car assembly line. Each assembly station is called a pipe stage or a pipe segment.

The throughput of an instruction pipeline is the measure of how often an instruction exits the pipeline.

**Pipeline Stages**

We can divide the execution of an instruction into the following 5 “classic” stages:

- **IF**: Instruction Fetch
- **ID**: Instruction Decode, register fetch
- **EX**: Execution
- **MEM**: Memory Access
- **WB**: Register write Back

**Pipeline Throughput and Latency**

Consider the pipeline above with the indicated delays. We want to know what is the pipeline throughput and the pipeline latency.

Pipeline throughput: instructions completed per second.

Pipeline latency: how long does it take to execute a single instruction in the pipeline.

**Pipeline Throughput and Latency**

Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction.

We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

**Pipelining Lessons**

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
Other Definitions

- Pipe stage or pipe segment
  - A decomposable unit of the fetch-decode-execute paradigm
- Pipeline depth
  - Number of stages in a pipeline
- Machine cycle
  - Clock cycle time
- Latch
  - Per phase/stage local information storage unit

Design Issues

- Balance the length of each pipeline stage

\[
\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}
\]

- Problems
  - Usually, stages are not balanced
  - Pipelining overhead
  - Hazards (conflicts)
- Performance (throughput \rightarrow CPU performance equation)
  - Decrease of the CPI
  - Decrease of cycle time

Basic Pipeline

<table>
<thead>
<tr>
<th>Instr #</th>
<th>Clock number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>IF</td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
</tr>
</tbody>
</table>

Pipelined Datapath with Resources

Physics of Clock Skew

- Basically caused because the clock edge reaches different parts of the chip at different times
  - Capacitance-charge-discharge rates
    - All wires, leads, transistors, etc. have capacitance
    - Longer wire, larger capacitance
    - Repeaters used to drive current, handle fan-out problem
  - C is inversely proportional to rate-of-change of V
    - Time to charge/discharge adds to delay
    - Dominant problem in old integration densities.
    - For a fixed C, rate-of-change of V is proportional to I
      - Problem with this approach is power requirements go up
      - Power dissipation becomes a problem.
  - Speed-of-light propagation delays
    - Dominates current integration densities as nowadays capacitances are much lower.
    - But nowadays clock rates are much faster (even small delays will consume a large part of the clock cycle)
- Current day research \rightarrow asynchronous chip designs
Performance Issues

- Unpipelined processor
  - 1.0 nsec clock cycle
  - 4 cycles for ALU and branches
  - 5 cycles for memory
  - Frequencies
  - ALU (40%), Branch (20%), and Memory (40%)
- Clock skew and setup adds 0.2ns overhead
- Speedup with pipelining?

Computing Pipeline Speedup

\[
\text{Speedup} = \frac{\text{average instruction time unpipelined}}{\text{average instruction time pipelined}}
\]

\[
\text{CPI pipelined} = \text{Ideal CPI} + \frac{\text{Pipeline stall clock cycles per instr}}{\text{Clock Cycle unpipelined}}
\]

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \frac{\text{Pipeline stall CPI}}{\text{Clock Cycle pipelined}}}
\]

Remember that average instruction time = CPI*Clock Cycle. And ideal CPI for pipelined machine is 1.

Pipeline Hazards

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline

Structural Hazards

- Overlapped execution of instructions:
  - Pipelining of functional units
  - Duplication of resources
- Structural Hazard
  - When the pipeline cannot accommodate some combination of instructions
- Consequences
  - Stall
  - Increase of CPI from its ideal value (1)

Structural Hazard with 1 port per Memory

Pipelining of Functional Units

Fully pipelined

Partially pipelined

Not pipelined
To pipeline or Not to pipeline

• Elements to consider
  – Effects of pipelining and duplicating units
  – Increased costs
  – Higher latency (pipeline register overhead)
  – Frequency of structural hazard

• Example: unpipelined FP multiply unit in MIPS
  – Latency: 5 cycles
  – Impact on midisp2 program?
    • Frequency of FP instructions: 14%
    • Depends on the distribution of FP multiplies
      • Best case: uniform distribution
      • Worst case: clustered, back-to-back multiplies

Example: Dual-port vs. Single-port

• Machine A: Dual ported memory
• Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
• Ideal CPI = 1 for both
• Loads are 40% of instructions executed
  
  \[
  \text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}
  \]

  \[
  \text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \frac{\text{clock}_{\text{unpipe}}}{(\text{clock}_{\text{unpipe}} / 1.05)}
  \]

  \[
  = \frac{\text{Pipeline Depth}}{1.4} \times 1.05
  \]

  \[
  = 0.75 \times \text{Pipeline Depth}
  \]

  \[
  \frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
  \]

• Machine A is 1.33 times faster

Data Hazards

Three Generic Data Hazards

Instr$_i$ followed by Instr$_j$

• Read After Write (RAW)
  Instr$_j$ tries to read operand before Instr$_i$ writes it

Three Generic Data Hazards (Cont’d)

Instr$_i$ followed by Instr$_j$

• Write After Read (WAR)
  Instr$_j$ tries to write operand before Instr$_i$ reads it
  – Gets wrong operand

• Can’t happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and
  – Reads are always in stage 2, and
  – Writes are always in stage 5

Three Generic Data Hazards (Cont’d)

Instr$_i$ followed by Instr$_j$

• Write After Write (WAW)
  Instr$_j$ tries to write operand before Instr$_i$ writes it
  – Leaves wrong result (Instr$_i$ not Instr$_j$)

• Can’t happen in MIPS 5 stage pipeline because:
  – All instructions take 5 stages, and
  – Writes are always in stage 5

• Will see WAR and WAW in later more complicated pipes
Examples in more complicated pipelines

- **WAW - write after write**
  
  \[
  \begin{align*}
    \text{LW } R1, 0(R2) & \quad \text{IF ID EX M1 M2 WB} \\
    \text{ADD } R1, R2, R3 & \quad \text{IF ID EX WB}
  \end{align*}
  \]

- **WAR - write after read**
  
  \[
  \begin{align*}
    \text{SW } 0(R1), R2 & \quad \text{IF ID EX M1 M2 WB} \\
    \text{ADD } R2, R3, R4 & \quad \text{IF ID EX WB}
  \end{align*}
  \]

  This is a problem if
  Register writes are during The first half of the cycle
  And reads during the Second half

Avoiding Data Hazard with Forwarding

Forwarding of Operands by Stores

Stalls in spite of Forwarding

Software Scheduling to Avoid Load Hazards

Try producing fast code for

\[
\begin{align*}
  a &= b + c; \\
  d &= e - f;
\end{align*}
\]

assuming a, b, c, d, e, and f in memory.

Slow code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb,b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Rc,c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD Ra,Rb,Re</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Re,e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB Rd,Re,Rf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW d,Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fast code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb,b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Rc,c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD Ra,Rb,Re</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Re,e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB Rd,Re,Rf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW d,Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effect of Software Scheduling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb,b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Rc,c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD Ra,Rb,Re</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW Re,e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB Rd,Re,Rf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW d,Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compiler Scheduling

- Eliminates load interlocks
- Demands more registers
- Simple scheduling
  - Basic block (sequential segment of code)
  - Good for simple pipelines
  - Percentage of loads that result in a stall
    - FP: 13%
    - Int: 25%

Control (Branch) Hazards

- Stall the pipeline until we reach MEM
  - Easy, but expensive
  - Three cycles for every branch
- To reduce the branch delay
  - Find out branch is taken or not taken ASAP
  - Compute the branch target ASAP

Impact of Branch Stall on Pipeline Speedup

- If CPI = 1, 30% branch,

Reduction of Branch Penalties

Static, compile-time, branch prediction schemes
1. Stall the pipeline
   - Simple in hardware and software
2. Treat every branch as not taken
   - Continue execution as if branch were normal instruction
   - If branch is taken, turn the fetched instruction into a no-op
3. Treat every branch as taken
   - Useless in MIPS .... Why?
4. Delayed branch
   - Sequential successors (in delay slots) are executed anyway
   - No branches in the delay slots

Delayed Branch

#4: Delayed Branch
- Define branch to take place AFTER a following instruction

    branch instruction
    sequential successor
    ........
    sequential successor
    branch target if taken

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Predict-not-taken Scheme

Compiler organizes code so that the most frequent path is the not-taken one
Canceling Branch Instructions

Canceling branch includes the predicted direction
• Incorrect prediction => delay-slot instruction becomes no-op
• Helps the compiler to fill branch delay slots (no requirements for b and c)
• Behavior of a predicted-taken canceling branch

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Talk branch includes the predicted direction
<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Canceling branch includes the predicted direction
• Incorrect prediction => delay-slot instruction becomes no-op
• Helps the compiler to fill branch delay slots (no requirements for b and c)
• Behavior of a predicted-taken canceling branch

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Talk branch includes the predicted direction
<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Branch Slot Requirements

Strategy | Requirements | Improves performance
---|-------------|-------------------
a) From before | Branch must not depend on delayed instruction | Always
b) From target | Must be OK to execute delayed instruction if branch is not taken | When branch is taken
c) From fall through | Must be OK to execute delayed instruction if branch is taken | When branch is not taken

Limitations in delayed-branch scheduling
Restrictions on instructions that are scheduled
Ability to predict branches at compile time

Branch Behavior in Programs

<table>
<thead>
<tr>
<th></th>
<th>Integer</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward conditional branches</td>
<td>13%</td>
<td>7%</td>
</tr>
<tr>
<td>Backward conditional branches</td>
<td>3%</td>
<td>2%</td>
</tr>
<tr>
<td>Unconditional branches</td>
<td>4%</td>
<td>1%</td>
</tr>
<tr>
<td>Branches taken</td>
<td>62%</td>
<td>70%</td>
</tr>
</tbody>
</table>

Pipeline speedup = (1 + Branch frequency × Branch penalty)
Branch Penalty for predict taken = 1
Branch Penalty for predict not taken = probability of branches taken
Branch Penalty for delayed branches is function of how often delay Slot is usefully filled (not cancelled), always guaranteed to be as Good or better than the other approaches.
Static Branch Prediction for scheduling to avoid data hazards
- Correct predictions
  - Reduce branch hazard penalty
  - Help the scheduling of data hazards:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>If branch is almost never taken</th>
<th>If branch is almost always taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, 0(R2)</td>
<td>SUB R1, R1, R3</td>
<td>BEQZ R1, L</td>
</tr>
<tr>
<td>SUB R1, R1, R3</td>
<td>OR R4, R5, R6</td>
<td>ADD R10, R4, R3</td>
</tr>
<tr>
<td>ADD R7, R8, R9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Prediction methods
  - Examination of program behavior (benchmarks)
  - Use of profile information from previous runs

How is Pipelining Implemented?
MIPS Instruction Formats

I
\[
\begin{array}{cccccc}
\text{opcode} & \text{rs} & \text{rd} & \text{immediate} \\
\hline
0 & 5 & 6 & 10 & 11 & 15 & 16 & 31 \\
\end{array}
\]

R
\[
\begin{array}{ccccccc}
\text{opcode} & \text{rs} & \text{rs2} & \text{rd} & \text{Shamt/function} \\
\hline
0 & 5 & 6 & 10 & 11 & 15 & 16 & 20 & 21 & 31 \\
\end{array}
\]

J
\[
\begin{array}{cccccc}
\text{opcode} & \text{address} \\
\hline
0 & 5 & 6 & 31 \\
\end{array}
\]

1st and 2nd Instruction cycles
- Instruction fetch (IF)
  - IR ← Mem[PC];
  - NPC ← PC + 4

- Instruction decode & register fetch (ID)
  - A ← Regs[IR_{10}];
  - B ← Regs[IR_{11,15}];
  - Imm ← (IR_{16} & # IR_{16...31})

3rd Instruction cycle
- Execution & effective address (EX)
  - Memory reference
    - ALUOutput ← A + Imm
  - Register - Register ALU instruction
    - ALUOutput ← A func B
  - Register - Immediate ALU instruction
    - ALUOutput ← A op Imm
  - Branch
    - ALUOutput ← NPC + Imm; Cond ← (A op 0)

4th Instruction cycle
- Memory access & branch completion (MEM)
  - Memory reference
    - PC ← NPC
    - LMD ← Mem[ALUOutput] (load)
    - Mem[ALUOutput] ← B (store)
  - Branch
    - if (cond) PC ← ALUOutput; else PC ← NPC

5th Instruction cycle
- Write-back (WB)
  - Register - register ALU instruction
    - Regs[IR_{16,20}] ← ALUOutput
  - Register - immediate ALU instruction
    - Regs[IR_{11,13}] ← ALUOutput
  - Load instruction
    - Regs[IR_{11,13}] ← LMD
5 Stages of MIPS Datapath

Events on Every Pipe Stage

Implementing the Control for the MIPS Pipeline

Pipeline Interlocks

Load Interlock Implementation
Forwarding Implementation

• Source: ALU or MEM output
• Destination: ALU, MEM or Zero? input(s)
• Compare (forwarding to ALU input):

  • Important
    – Please refer to Fig. A.22 in slide #63

Forwarding Implementation - All Possible Forwarding

Handling Branch Hazards

Revised Pipeline Structure

Exceptions

• I/O device request
• Operating system call
• Tracing instruction execution
• Breakpoint
• Integer overflow
• FP arithmetic anomaly
• Page fault
• Misaligned memory access
• Memory protection violation
• Undefined instruction
• Hardware malfunctions
• Power failure
Exception Categories

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. nonmaskable
- Within vs. between instructions
- Resume vs. terminate
- Most difficult
  - Occur in the middle of the instruction
  - Must be able to restart
  - Requires intervention of another program (OS)

Overview of Exceptions

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Synchronous User Requested</th>
<th>User Maskable vs. Nonmaskable</th>
<th>Between-Instruction</th>
<th>Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td>_trap</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Resume</td>
</tr>
<tr>
<td>trap</td>
<td>Synchronous</td>
<td>User request</td>
<td>Maskable</td>
<td>Resume</td>
</tr>
<tr>
<td>trap</td>
<td>Synchronous</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>trap</td>
<td>Synchronous</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Trap</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Resume</td>
</tr>
<tr>
<td>Trap</td>
<td>Synchronous</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Trap</td>
<td>Synchronous</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>Synchronous</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>Synchronous</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Memory protection violation</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Resume</td>
</tr>
<tr>
<td>Memory protection violation</td>
<td>Synchronous</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Memory protection violation</td>
<td>Synchronous</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Hardware faults</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Resume</td>
</tr>
<tr>
<td>Hardware faults</td>
<td>Asynchronous</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Hardware faults</td>
<td>Asynchronous</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
</tbody>
</table>

Exception Handling

Stopping and Restarting Execution

- TRAP, RFE(return-from-exception) instructions
- IAR register saves the PC of faulting instruction
- Safely save the state of the pipeline
  - Force a TRAP on the next IF
  - Until the TRAP is taken, turn off all writes for the faulting instruction and the following ones.
  - Exception-handling routine saves the PC of the faulting instruction
- For delayed branches we need to save more PCs
- Precise Exceptions

Exceptions in MIPS

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault, misaligned memory access, memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault, misaligned memory access, memory-protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

Exception Handling in MIPS

LW

ADD
ISA and Exceptions

- Instructions before complete, instructions after do not, exceptions handled in order → Precise Exceptions

- Precise exceptions are simple in MIPS
  - Only one result per instruction
  - Result is written at the end of execution
- Problems
  - Instructions change machine state in the middle of the execution
  - Autoincrement addressing modes
  - Multicycle operations
- Many machines have two modes
  - Imprecise (efficient)
  - Precise (relatively inefficient)

Handling Multicycle Operations

Precise Exceptions

- Precise exceptions are simple in MIPS
  - Only one result per instruction
  - Result is written at the end of execution
- Problems
  - Instructions change machine state in the middle of the execution
  - Autoincrement addressing modes
  - Multicycle operations
- Many machines have two modes
  - Imprecise (efficient)
  - Precise (relatively inefficient)

Latencies and Initiation Intervals

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Latency</th>
<th>Initiation Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data Memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP adder</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP/int multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP/int divider</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>MULTD</td>
<td>IF ID</td>
<td>M1 M2 M3 M4 M5 M6 M7 Mem WB</td>
</tr>
<tr>
<td>ADDD</td>
<td>IF ID</td>
<td>A1 A2 A3 A4 Mem WB</td>
</tr>
<tr>
<td>LD</td>
<td>IF ID</td>
<td>EX Mem WB</td>
</tr>
<tr>
<td>SD</td>
<td>IF ID</td>
<td>EX Mem WB</td>
</tr>
</tbody>
</table>

Hazards in FP pipelines

- Structural hazards in DIV unit
- Structural hazards in WB
- WAW hazards are possible (WAR not possible)
- Out-of-order completion
  - Exception handling issues
- More frequent RAW hazards
  - Longer pipelines

Hazards in FP pipelines (Cont’d)
## Hazard Detection Logic at ID

- **Check for Structural Hazards**
  - Divide unit/make sure register write port is available when needed

- **Check for RAW hazard**
  - Check source registers against destination registers in pipeline latches of instructions that are ahead in the pipeline. Similar to I-pipeline

- **Check for WAW hazard**
  - Determine if any instruction in A1-A4, M1-M7 has same register destination as this instruction.