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 2nsec 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: how often an instruction is completed.
\[
\text{Instr throughput} = \frac{\text{instr}}{\max[5\text{ns}, 4\text{ns}, 5\text{ns}, 10\text{ns}, 4\text{ns}]}
\]
\[
= \frac{\text{instr}}{10\text{ns}} \quad \text{(ignoring pipeline register overhead)}
\]

Pipeline latency: how long does it take to execute an instruction in the pipeline.
\[
L = \text{lat(IF)} + \text{lat(ID)} + \text{lat(EX)} + \text{lat(MEM)} + \text{lat(WB)}
\]
\[
= 5\text{ns} + 4\text{ns} + 5\text{ns} + 10\text{ns} + 4\text{ns} = 28\text{ns}
\]

Is this right?
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 → 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 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>$i$</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>$i+1$</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>$i+2$</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>$i+3$</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>$i+4$</td>
<td>IF ID EX MEM WB</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 problems
    - 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 → 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

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

\[
\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{IDEAL CPI} + \text{Pipeline stall per instr}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}
\]

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

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 can not 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 `mdljdip2` 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

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}} = \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \frac{\text{clock}_{\text{unpipe}}}{(\text{clock}_{\text{unpipe}} / 1.05)} = (\text{Pipeline Depth}/1.4) \times 1.05 = 0.75 \times \text{Pipeline Depth}
\]

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

- Machine A is 1.33 times faster
Three Generic Data Hazards

Instrᵢ followed by Instrⱼ

- **Read After Write (RAW)**
  Instrⱼ tries to read operand before Instrᵢ writes it
Three Generic Data Hazards (Cont’d)
Instr_1 followed by Instr_2

• **Write After Read (WAR)**
  Instr_2 tries to write operand *before* Instr_1 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_1 followed by Instr_2

• **Write After Write (WAW)**
  Instr_2 tries to write operand *before* Instr_1 writes it
  – Leaves wrong result (Instr_1 not Instr_2)

• 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
  
  \[
  \text{LW R1, 0(R2)} \quad \text{IF ID EX M1 M2 WB} \\
  \text{ADD R1, R2, R3} \quad \text{IF ID EX WB} \\
  \]

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

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

\[ a = b + c; \]

\[ d = e - f; \]

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

Slow code:

**LW** \( Rb,b \)

**LW** \( Rc,c \)

**ADD** \( Ra,Rb,Rc \)

**SW** \( a,Ra \)

**LW** \( Re,e \)

**SW** \( d,Rd \)

Fast code:

**LW** \( Rb,b \)

**LW** \( Rc,c \)

**LW** \( Re,e \)

**ADD** \( Ra,Rb,Rc \)

**SW** \( a,Ra \)

**LW** \( Re,e \)

**LW** \( Rf,f \)

**SW** \( a,Ra \)

**SW** \( d,Rd \)

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><strong>LW</strong> ( Rb,b )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LW</strong> ( Rc,c )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ADD</strong> ( Ra,Rb,Rc )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SW</strong> ( a,Ra )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LW</strong> ( Re,e )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SW</strong> ( Rd,Re,Rf )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SW</strong> ( 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

<table>
<thead>
<tr>
<th>Branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch successor</td>
<td>IF stall stall</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch successor+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor+2</td>
<td>IF ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch successor+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Branch successor+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

\[
\begin{align*}
\text{branch instruction} & \\
\text{sequential successor}_1 & \\
\text{sequential successor}_2 & \\
\ddots & \\
\text{sequential successor}_n & \\
\text{branch target if taken} & \\
\end{align*}
\]

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

Predict-not-taken Scheme

\begin{tabular}{|c|c|c|c|c|c|}
\hline
Untaken Branch & IF & ID & EX & MEM & WB \\
\hline
Instruction i+1 & IF & ID & EX & MEM & WB \\
Instruction i+1 & IF & ID & EX & MEM & WB \\
Instruction i+2 & IF & ID & EX & MEM & WB \\
Instruction i+3 & IF & ID & EX & MEM & WB \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
Taken Branch & IF & ID & EX & MEM & WB \\
\hline
Instruction i+1 & IF & stall & stall & stall & \text{(clear the IF/ID register)} \\
Branch target & IF & ID & EX & MEM & WB \\
Branch target+1 & IF & ID & EX & MEM & WB \\
Branch target+2 & IF & ID & EX & MEM & WB \\
\hline
\end{tabular}

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

```
Untaken Branch
Instruction i+1: IF stall stall stall stall (clear the IF/ID register)
Instruction i+2: IF ID EX MEM WB
Instruction i+3: IF ID EX MEM WB
Instruction i+4: IF ID EX MEM WB

Taken Branch
Instruction i+1: IF ID EX MEM WB
Branch target: IF ID EX MEM WB
Branch target i+1: IF ID EX MEM WB
Branch target i+2: IF ID EX MEM WB
```

Delayed Branch

• Where to get instructions to fill branch delay slot?
  – Before branch instruction
  – From the target address: only valuable when branch taken
  – From fall through: only valuable when branch not taken
• Compiler effectiveness for single branch delay slot:
  – Fills about 60% of branch delay slots
  – About 80% of instructions executed in branch delay slots useful in computation
  – About 50% (60% x 80%) of slots usefully filled
• Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)
Optimizations of the Branch Slot

Branch Slot Requirements

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Requirements</th>
<th>Improves performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) From before</td>
<td>Branch must not depend on delayed instruction</td>
<td>Always</td>
</tr>
<tr>
<td>b) From target</td>
<td>Must be OK to execute delayed instruction if branch is not taken</td>
<td>When branch is taken</td>
</tr>
<tr>
<td>c) From fall through</td>
<td>Must be OK to execute delayed instruction if branch is taken</td>
<td>When branch is not taken</td>
</tr>
</tbody>
</table>

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

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 = \( \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}} \)

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:

  ```
  LW R1, 0(R2)
  SUB R1, R1, R3
  BEQZ R1, L
  OR R4, R5, R6
  ADD R10, R4, R3
  L: ADD R7, R8, R9
  ```

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

How is Pipelining Implemented?

MIPS Instruction Formats

<table>
<thead>
<tr>
<th>I</th>
<th>opcode</th>
<th>rs₁</th>
<th>rd</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 5 6 10 11 15 16 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs₁</th>
<th>rs₂</th>
<th>rd</th>
<th>Shamt/function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 5 6 10 11 15 16 20 21 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 5 6 31</td>
<td></td>
</tr>
</tbody>
</table>

Fixed-field decoding
1st and 2nd Instruction cycles

• Instruction fetch (IF)
  IR ← Mem[PC];
  NPC ← PC + 4

• Instruction decode & register fetch (ID)
  A ← Regs[IR6..10];
  B ← Regs[IR11..15];
  Imm ← ((IR16)$_{16}$ # # 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[IR16..20] ← ALUOutput
  - Register - immediate ALU instruction
    - Regs[IR11..15] ← ALUOutput
  - Load instruction
    - Regs[IR11..15] ← LMD
5 Stages of MIPS Datapath

5 Stages of MIPS Datapath with Registers
Events on Every Pipe Stage

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF/ID, IR ← Mem[PC]; IF/ID, IR, PC ← (IF ((EX/MEM.opcode == branch) &amp; EX/MEM.cond) ? EX/MEM, ALUOutput) else PC[4:0]);</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A ← Regs[1/ID, IR[16-9]]; ID/EX.R ← Regs[1/ID, IR[15-9]]; ID/EX.NPC ← IF/ID, IR; ID/EX.IR ← IF/ID, IR; ID/EX.irm ← sign-extend(IF/ID, IR[15-12]);</td>
</tr>
</tbody>
</table>

**Implementing the Control for the MIPS Pipeline**

- **LD R1, 45 (R2)**
  - DADD R5, R6, R7
  - DSUB R8, R6, R7
  - OR R9, R6, R7
- **LD R1, 45 (R2)**
  - DADD R5, R1, R7
  - DSUB R8, R6, R7
  - OR R9, R6, R7
- **LD R1, 45 (R2)**
  - DADD R5, R6, R7
  - DSUB R8, R1, R7
  - OR R9, R6, R7
- **LD R1, 45 (R2)**
  - DADD R5, R6, R7
  - DSUB R8, R6, R7
  - OR R9, R1, R7

58
Pipeline Interlocks

Load Interlock Implementation

- RAW load interlock detection during ID
  - Load instruction in EX
  - Instruction that needs the load data in ID
- Logic to detect load interlock

<table>
<thead>
<tr>
<th>ID/EX.IR_{0..5}</th>
<th>IF/ID.IR_{0..5}</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load r-r ALU</td>
<td>ID/EX.IR_{[RT]} == IF/ID.IR_{[RS]}</td>
<td></td>
</tr>
<tr>
<td>Load r-r ALU</td>
<td>ID/EX.IR_{[RT]} == IF/ID.IR_{[RT]}</td>
<td></td>
</tr>
<tr>
<td>Load, Store, r-i ALU, branch</td>
<td>ID/EX.IR_{[RT]} == IF/ID.IR_{[RS]}</td>
<td></td>
</tr>
</tbody>
</table>

- Action (insert the pipeline stall)
  - ID/EX.IR_{0..5} = 0 (no-op)
  - Re-circulate contents of IF/ID
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 (Cont’d)
Forwarding Implementation - All Possible Forwarding

<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison of equal then forwarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register, ALU</td>
<td>ID/EX</td>
<td>ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM[M][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>Register, ALU</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM[M][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Register, ALU</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Top ALU input</td>
<td>MEM/WB[B][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Register, ALU</td>
<td>ID/EX</td>
<td>Bottom ALU input</td>
<td>MEM/WB[B][e] = ID/EX[I][r]</td>
<td></td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Top ALU input</td>
<td>EX/MEM[M][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB[B][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Top ALU input</td>
<td>MEM/WB[B][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB[B][e] = ID/EX[I][r]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register, ALU</td>
<td>Top ALU input</td>
<td>MEM/WB[B][e] = ID/EX[I][r]</td>
</tr>
</tbody>
</table>

Handling Branch Hazards
Revised Pipeline Structure

<table>
<thead>
<tr>
<th>Pipe stage</th>
<th>Branch instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>( \text{IF/ID.1R} \leftarrow \text{Mem}[\text{PC}] ); \ ( \text{IF/ID.NPC, PC} \leftarrow (\text{IP} \text{IF/ID.1R, op}) );</td>
</tr>
<tr>
<td>ID</td>
<td>( \text{IF/ID.1R} \leftarrow \text{IF/ID.1R} + \text{IF/ID.1R} );</td>
</tr>
<tr>
<td>EX</td>
<td>( \text{IF/ID.1R} \leftarrow \text{IF/ID.1R} + \text{IF/ID.1R} );</td>
</tr>
<tr>
<td>MEM</td>
<td>( \text{IF/ID.1R} \leftarrow \text{IF/ID.1R} + \text{IF/ID.1R} );</td>
</tr>
<tr>
<td>WB</td>
<td>( \text{IF/ID.1R} \leftarrow \text{IF/ID.1R} + \text{IF/ID.1R} );</td>
</tr>
</tbody>
</table>

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 vs. asynchronous</th>
<th>User request vs. coerced</th>
<th>User maskable vs. nonmaskable</th>
<th>Within vs. between instructions</th>
<th>Resume vs terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Invoke operating system</td>
<td>Synchronous</td>
<td>User request</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Tracing instruction execution</td>
<td>Synchronous</td>
<td>User request</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Synchronous</td>
<td>User request</td>
<td>User maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Integer arithmetic overflow</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>User maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Floating-point arithmetc overflow or underflow</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>User maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned memory accesses</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>User maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Memory protection violations</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Using undefined instructions</td>
<td>Synchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Asynchronous</td>
<td>Coerced</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>
### Exception Handling

![Exception Handling Diagram]

### 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**: IF ID EX M WB
- **ADD**: IF ID EX M WB
- **LW**: IF ID EX M WB
- **ADD**: IF ID EX M WB

**Exception Status Vector**
- Check exceptions here
ISA and Exceptions

- Instructions before complete, instructions after do not, exceptions handled in order \textit{Precise Exceptions}

- \textbf{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
Handling Multicycle Operations
(Cont’d)

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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MULTD</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>Mem</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

```
LD F4, 0(R2)  IF  ID  EX  Mem  WB
MULTD F0, F4, F6  IF  ID  stall  M1  M2  M3  M4  M5  M6  M7  Mem  WB
ADD F2, F0, F8  IF  stall  ID  stall  stall  stall  stall  stall  stall  A1  A2  A3  A4  Mem  WB
```

Hazards in FP pipelines (Cont’d)

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT  F0, F4, F6</td>
<td>IF  ID  M1  M2  M3  M4  M5  M6  M7  MEM  WB</td>
</tr>
<tr>
<td></td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td></td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>ADD  F2, F4, F6</td>
<td>IF  ID  A1  A2  A3  A4  MEM  WB</td>
</tr>
<tr>
<td></td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td></td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>LD  F2, 0(R2)</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
</tbody>
</table>
```
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.

Handling Hazards – Working Examples