Pipe hazards include:

- **Structural hazards**
- **Data hazards**
- **Control hazards**

Pipelining Lessons:

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Use different resources to execute multiple tasks simultaneously
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

Classic Five-Stage Pipeline:

- **Ifetch**: Instruction Fetch
  – Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Perform calculation for instruction
- **Mem**: Access the data from the Data Memory
- **Wr**: Write the data back to the register file
Conventional Pipelined Execution Representation

<table>
<thead>
<tr>
<th>Time</th>
<th>IFetch</th>
<th>Dc</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program Flow

- Assume each stage is one cycle
- \# of cycles = \# of instructions + (\# of stages – 1)
  - Assuming full utilization of pipeline

Pipelining is not quite that easy!

- 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: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
Resolving Structural Hazards

- **Definition**: attempt to use same hardware for two different things at the same time
- **Solution 1**: Wait
  - must detect the hazard
  - must have mechanism to stall
- **Solution 2**: Throw more hardware at the problem

Data Hazards

- **Read After Write (RAW)**
  - Instr$_j$ tries to read operand before Instr$_i$ writes it
  - I: add r1,r2,r3
  - J: sub r4,r1,r3
  - Caused by a “dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- **Write After Read (WAR)**
  - Instr$_j$ writes operand before Instr$_i$ reads it
  - I: sub r4,r1,r3
  - J: add r1,r2,r3
  - K: mul r6,r1,r7
  - Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.
  - 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

- **Write After Write (WAW)**
  
  Instr$_i$ writes operand before Instr$_j$ writes it.
  
  $I$: sub $r1, r4, r3$
  
  $J$: add $r1, r2, r3$
  
  $K$: mul $r6, r1, r7$

- Called an “output dependence” by compiler writers
  
  This also results from the reuse of name “r1”.

- 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 more complicated pipes

---

Data Hazards

- **Read After Write (RAW)**
  - True data dependence (most common type)
  - Must preserve program order to ensure correct execution

- **Write After Read (WAR)**
  - Output dependence
  - Only present in pipelines that
    - write in more than one pipe stage
    - allow instructions to proceed after previous instructions stalls

- **Write After Write (WAW)**
  - Anti-dependence
  - Can occur when instructions are reordered

---

Datapath of Mic-3 Microarchitecture

- Add latches on A bus, B bus, and C bus

- Benefits
  - Speed up the clock cycle because maximum delay is shorter
  - Use all parts of datapath during every cycle

---

Implementation of SWAP Instruction

<table>
<thead>
<tr>
<th>Label</th>
<th>Operations</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw/ap1</td>
<td>MAR ← SP – 1; rd</td>
<td>Read 2nd word from stack; set MAR to SP</td>
</tr>
<tr>
<td>sw/ap2</td>
<td>MAR ← SP</td>
<td>Prepare to write new 2nd word</td>
</tr>
<tr>
<td>sw/ap3</td>
<td>H ← MDR, cr</td>
<td>Store new TOS; write 2nd word to stack</td>
</tr>
<tr>
<td>sw/ap3</td>
<td>MDR ← TOE</td>
<td>Copy old TOS to MDR</td>
</tr>
<tr>
<td>sw/ap4</td>
<td>MAR ← SP – 1; cr</td>
<td>Write old TOS to 2nd place on stack</td>
</tr>
<tr>
<td>sw/ap5</td>
<td>TOE ← n, goto (MBR1)</td>
<td>Load TOE</td>
</tr>
</tbody>
</table>

---

Figure 4.12. The Micc2 code for SWAP

Figure 4.13. The implementations of SWAP on the Micc3
Control Hazards

- The IFU pipelines with sequential instruction stream in method area
- A branch may indicate another control flow

```java
ILOAD i
BIPUSH 3
IF_ICMPEQ L1
ILOAD j
BIPUSH 1
ISUB
ISTORE j
GOTO L2
```

L1: BIPUSH 0
ISTORE k

L2: ...

Branches

- Unconditional
  - Must decide where to fetch before knowing what instruction it just got
  - Can insert a “delay slot” after unconditional branch
    - Try to fill it with useful work, but typically a NOP
- Conditional
  - Must test the condition before knowing what instruction to fetch
  - Can stall the pipeline until the branch is resolved
    - Defeats the purpose of pipelining!

Control Hazard on Branches

=> Three Stage Stall

```
10: beq r1,r3,36
14: and r2,r3,r5
18: or r6,r1,r7
22: add r8,r1,r9
36: xor r10,r1,r11
```

Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
    - 47% MIPS branches not taken on average

#3: Predict Branch Taken
  - Must fetch instructions from the target address
  - 53% MIPS branches taken on average
Branch Prediction

- **A simple method**
  - Assume backward branching is taken
  - Assume forward branching is not taken

- **If prediction is correct, then the pipeline is OK**

- **If prediction is incorrect, then must have a way to roll back the CPU state to before mispredicted branch**
  - Costly in H/W

Dynamic Branch Prediction

- **Branch History Table (BHT)**
  - CPU maintains a history of branch locations and their previous behavior

- **Branch Target Address Cache (BTAC)**
  - Stores the target location for branches predicted taken

- **Use 1 bit to store history**
  - Stores what the branch did the last time
  - With loops, you will always mispredict entering and exiting

- **Use 2 bits to store history**
  - Maintains a strong and weak prediction state
  - Must mispredict twice to change the prediction

Dynamic Branch Prediction

- **Dynamic prediction**
  - CPU maintains a history of branch locations and their previous behavior
  - Algorithms include always taking backward conditional branches or predicting based on what the branch did the last time

- **Static prediction**
  - Compiler uses branch instructions that specify the branch’s normal outcome (Taken or Not Taken)
  - Program is simulated to profile the branch behavior

---

Adapted from John Kubiatowicz’s CS 252 lecture notes. Copyright © 2003 UCB.
2-Bit Branch Prediction FSM

Strongly predicts taken

Weakly predicts taken

Weakly predicts not taken

Strongly predicts not taken

Static Branch Prediction

• Compiler uses branch instructions that specify the branch’s normal outcome
  – Based upon the code to be compiled
  – Taken or Not Taken

• Program is simulated to profile the branch behavior
  – Branch statistics are given to the compiler

Summary

• Pipelines must deal with structural, data, and control hazards

• There is no such thing as a “Read After Read” (RAR) data hazard

• Branch prediction is an important part of scalar execution