What is Pipelining?

- Overlapping execution to produce faster results
  - Washing and drying dishes
  - Washing and drying laundry
  - Automobile assembly line
  - Chipotle, Quiznos, etc
- Pipelining in computer architecture
  - Multiple instructions are overlapped in execution
  - Exploits parallelism
  - Not visible to programmer
- Each stage is a pipeline "cycle"
  - Each stage happens simultaneously so results are produced only as fast as the longest pipeline cycle
  - Determines clock cycle time

Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts

A "Typical" RISC ISA (Load/Store)

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero)
- ALU instructions
  - 3-address, reg-reg arithmetic instruction
  - 2-address, reg-im arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch
Example: MIPS (- MIPS)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
<th>Opx</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21120</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

Register-Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
<th>Opx</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21120</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Immediate</th>
<th>Opx</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21120</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Opx</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21120</td>
<td>16</td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Opx</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>25</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Datapath vs Control (FSM+D)

- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the datapath
  - Based on desired function and signals

Approaching an ISA

- **Instruction Set Architecture**
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- **Meaning of each instruction is described by RTL on architected registers and memory**
- **Given technology constraints assemble adequate datapath**
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MBR, ...)
  - Interconnect to move information among regs and FUs
- **Implement controller (Finite State Machine (FSM))**

Outline

- **MIPS – An ISA for Pipelining**
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
5 Steps of MIPS Datapath

Instruction Fetch

Instr. Decode

Execute Addr. Calc

Memory Access

Write Back

5 Steps of MIPS Datapath

Instruction Fetch

Instr. Decode

Execute Addr. Calc

Memory Access

Write Back

Inst. Set Processor Controller

If jmp(A, R1)
PC <= IR[pc]
A <= IR[rs]
B <= IR[rt]
rslt <= op
Reg[IR[rd]] <= WB
WB <= rslt

If bop(A, B)
PC <= IR[jaddr]
A <= IR[rd]
B <= IR[rt]
rslt <= op
Reg[IR[rd]] <= WB
WB <= rslt

If br
PC <= PC + 4

If jr
PC <= IR[rd]

If jsr
IR <= mem[PC];
PC <= PC + 4

If load
A <= Reg[IR[rs]];
B <= Reg[IR[rt]];
WB <= Mem[IR[rd]];
Reg[IR[rd]] <= WB

If store
A <= Reg[IR[rs]];
B <= Reg[IR[rt]];

Visualizing Pipelining

Figure A.2, Page A-8

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7

Time (clock cycles)

Instr. Order
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).

One Memory Port/Structural Hazards
(Similar to Figure A.5, Page A-15)

**Speed Up Equation for Pipelining**

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

For simple RISC pipeline, CPI = 1:
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- 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
  
  \[
  \text{SpeedUp}_A = \frac{\text{Pipeline Depth}(1 + 0)}{\text{Pipeline Depth}} \times \left( \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}} \right) = \text{Pipeline Depth}
  \]
  
  \[
  \text{SpeedUp}_B = \frac{\text{Pipeline Depth}(1 + 0.4 \times 1)}{\text{Pipeline Depth}(1.4)} \times 1.05 = 0.75 \times \text{Pipeline Depth}
  \]

- Machine A is 1.33 times faster

Data Hazard on R1

- Read After Write (RAW): Instruction J tries to read operand before Instruction I writes it
  
  - I: \( \text{add } r1,r2,r3 \)
  - J: \( \text{sub } r4,r1,r3 \)

- Caused by a "dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- Write After Read (WAR): Instruction I writes operand before Instruction J reads it
  
  - I: \( \text{sub } r4,r1,r3 \)
  - J: \( \text{add } r1,r2,r3 \)
  - K: \( \text{mul } r6,r1,r7 \)

- 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\textsubscript{J} writes operand *before* Instr\textsubscript{I} 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.

---

**Forwarding to Avoid Data Hazard**

![Diagram](Figure A.7, Page A-19)

**Forwarding to Avoid LW-SW Data Hazard**

![Diagram](Figure A.8, Page A-20)

**HW Change for Forwarding**

![Diagram](Figure A.23, Page A-37)
Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \) in memory.

\[
\begin{align*}
\text{Slow code:} & \quad \text{Fast code:} \\
\text{LW} & \quad \text{LW} \\
Rb, b & \quad Rb, b \\
Rc, c & \quad Rc, c \\
\text{ADD} & \quad \text{ADD} \\
Ra, Rb, Rc & \quad Ra, Rb, Rc \\
\text{LW} & \quad \text{LW} \\
Re, e & \quad Lf, f \\
\text{SW} & \quad \text{SW} \\
a, Ra & \quad a, Ra \\
\text{SUB} & \quad \text{SUB} \\
Rd, Re, Rf & \quad Rd, Re, Rf \\
\text{SW} & \quad \text{SW} \\
d, Rd & \quad d, Rd
\end{align*}
\]

Compiler optimizes for performance. Hardware checks for safety.

---

Outline
- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion
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

What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath

Four 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
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
- 53% MIPS branches taken on average
- But haven’t calculated branch target address in MIPS
- MIPS still incurs 1 cycle branch penalty
- Other machines: branch target known before outcome
Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER a following instruction
  - branch instruction
  - sequential successor
  - sequential successor
  - sequential successor
- Branch delay of length $n$
  - 1 slot delay allows proper decision and branch target address in 3 stage pipeline
  - MIPS uses this

Scheduling Branch Delay Slots (Fig A.14)
A. From before branch
  - add $1, $2, $3$
  - if $2>0$ then
    - delay slot
    - becomes
    - add $1, $2, $3$
    - if $2>0$ then
  - becomes
  - add $1, $2, $3$

B. From branch target
  - sub $4, $5, $6$
  - if $1=0$ then
    - delay slot
    - becomes
    - add $1, $2, $3$
    - if $1=0$ then
  - becomes
  - sub $4, $5, $6$

C. From fall through
  - add $1, $2, $3$

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Delayed Branch

- 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: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pipeline speedup</th>
<th>Pipeline depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stalled pipeline</td>
<td>1.00</td>
<td>1 + Branch frequency x Branch penalty</td>
</tr>
</tbody>
</table>

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<table>
<thead>
<tr>
<th>Branch penalty</th>
<th>Speedup v. unip.</th>
<th>Speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3.10</td>
<td>1.00</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1.20</td>
<td>1.33</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1.14</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>1.10</td>
<td>1.45</td>
</tr>
</tbody>
</table>