What is Pipelining?

- Overlapping execution to produce faster results
  - Washing and drying dishes
  - Washing and drying laundry
  - Automobile assembly line
  - Chipotle, Quiznos, etc
- Speeds up production
  - Master employees
  - Eliminates "jack of all trades, master of none" syndrome
- 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)

- Invented to be easy to pipeline
- 32-bit fixed format instruction (3 formats)
  - Fixed length, easy to decode
- 31+1 32-bit GPR (General Purpose Registers) (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

A “Typical” RISC ISA (Load/Store)

- Invented to be easy to pipeline
- 32-bit fixed format instruction (3 formats)
  - Fixed length, easy to decode
- 31+1 32-bit GPR (General Purpose Registers) (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

Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>shamt</td>
<td>Opx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Datapath vs Control (FSM+D)

Datapath: Storage, Functional units (Fus), interconnect sufficient to perform the desired functions
- Inputs are Control Points
- Outputs are signals
Controller: State machine to orchestrate operation on the data path
- Based on desired function and signals

Approaching an ISA – How to Pipeline

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

Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
5 Steps of MIPS Datapath
Figure A.2, Page A-8

No pipelining here, just steps. 1 cycle does it all.

Inst. Set Processor Controller

Visualizing Pipelining
Figure A.2, Page A-8
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 One Port Structural Hazards**

Figure A.4, Page A-14

**Time (clock cycles)**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ins1</td>
<td>Ins2</td>
<td>Ins3</td>
<td>Ins4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- How do you “bubble” the pipe?

---

**One Memory Port/Structural Hazards**

(Similar to Figure A.5, Page A-15)

**Time (clock cycles)**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ins1</td>
<td>Ins2</td>
<td>Ins3</td>
<td>Ins4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Performance of Pipeline with Stalls**
  - **Ideal CPI speedup is simply the pipeline depth**
    - Assumes no stalls, perfect execution
  - **But pipelining causes stalls and changes the clock cycle time**
    
      \[
      \text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{\text{CPI unpipelined} \times \text{Clock cycle time unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle time pipelined}}
      \]
  - **Ideal CPI is 1**

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} = 1 + \text{Pipeline stall clock cycles per instruction}
\]

\[
\text{CPI unpipelined} = \text{Ideal CPI} \times \text{Pipeline Depth} = \text{Pipeline Depth}
\]
**Performance of Pipelines with Stalls**

- Let's ignore cycle time overhead for pipelining and assume all stages are balanced, thus cycle times for each are equal.

\[
\text{Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}}
\]

- Assuming no pipeline stalls, speedup is equal to pipeline depth.
- But pipelining changes the clock cycle time too....

**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

\[
\begin{align*}
\text{Avg inst time}_A &= \text{CPI} \times \text{Clock cycle time} = \text{Clock cycle time} \\
\text{Avg inst time}_B &= \text{CPI} \times \text{Clock cycle time} = (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}}{1.05} \\
&= 1.3 \times \text{Clock cycle time}
\end{align*}
\]

Machine A is 1.3 times faster

**Performance of Pipelines with Stalls**

- Pipelining reduced clock cycle time (increases frequency) – less work to do in each stage.
- CPI unpipelined is 1

\[
\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{\text{CPI unpipelined} \times \text{Clock cycle time unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle time pipelined}}
\]

- If all pipeline stages are balanced:

\[
\frac{\text{Clock cycle pipelined}}{\text{Clock cycle unpipelined}} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}}
\]

\[
\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
\]
Data Hazard on R1

Figure A.6, Page A-17

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID/RF</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
</tbody>
</table>

Forwarding to Avoid Data Hazard

Figure A.7, Page A-19

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID/RF</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td>xfer</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
<td>ra</td>
</tr>
</tbody>
</table>

Three Generic Data Hazards

- Read After Write (RAW)
  Instr_j tries to read operand before Instr_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)
  Instr_j writes operand _before_ Instr_i 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\(_J\) writes operand *before* Instr\(_I\) writes it.
  
  \[
  \begin{align*}
  \text{I: } & \text{sub } r1, r4, r3 \\
  \text{J: } & \text{add } r1, r2, r3 \\
  \text{K: } & \text{mul } r6, r1, r7
  \end{align*}
  \]

- 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 LW-SW Data Hazard

Figure A.8, Page A-20

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
</table>

- `add r1, r2, r3`
- `lw r4, 0(r1)`
- `sw r4, 12(r1)`
- `or r8, r6, r9`
- `xor r10, r9, r11`

Data Hazard Even with Forwarding

Figure A.9, Page A-21

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
<th>Load use dependency</th>
</tr>
</thead>
</table>

- `lw r1, 0(r2)`
- `sub r4, r1, r6`
- `and r6, r1, r7`
- `or r8, r1, r9`
Data Hazard Even with Forwarding
(Similar to Figure A.10, Page A-21)

Time (clock cycles)

How is this detected?

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

Software Scheduling to Avoid Load Hazards

• Try to produce faster code for:
  \[ a = b + c \]
  \[ d = e - f \]
assuming a, b, c, d, e, and f in memory

Controller optimizes for performance. Hardware checks for safety.

9/13/10

Control Hazard on Branches
Three Stage Stall

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

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

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

Pipelined MIPS Datapath
Figure A.24, page A-38

Interplay of instruction set design and cycle time.
### Scheduling Branch Delay Slots

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1,$2,$3 if $2=0$ then delay slot</td>
<td>sub $4,$5,$6 add $1,$2,$3 if $1=0$ then delay slot</td>
<td>add $1,$2,$3 if $1=0$ then delay slot sub $4,$5,$6</td>
</tr>
<tr>
<td>becomes</td>
<td></td>
<td>becomes</td>
</tr>
<tr>
<td>if $2=0$ then add $1,$2,$3</td>
<td></td>
<td>add $1,$2,$3 if $1=0$ then sub $4,$5,$6</td>
</tr>
</tbody>
</table>

**Notes:**
- A is the best choice, fills delay slot & reduces instruction count (IC)
- B and C incorporate branch prediction, essentially, and instructions must be squashed (aborted) if incorrect
- In B, may need to copy sub if it can be reached by other execution paths

### 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

Pipeline speedup = \[
\frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}
\]

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

<table>
<thead>
<tr>
<th>Scheduling Branch</th>
<th>CPI</th>
<th>speedup v. scheme penalty</th>
<th>unpipelined</th>
<th>stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3.1</td>
<td>0.5</td>
<td>4.5</td>
<td>1.45</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1.2</td>
<td>0.5</td>
<td>4.5</td>
<td>1.45</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>4.4</td>
<td>1.0</td>
<td>4.5</td>
<td>1.45</td>
</tr>
</tbody>
</table>