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 personal
  - 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
- 32 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**

<table>
<thead>
<tr>
<th>Register-Register</th>
<th>Register-Immediate</th>
<th>Branch</th>
<th>Jump / Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 26 25 2120 16 15 1110 6 5 0</td>
<td>31 26 25 2120 16 15 0</td>
<td>31 26 25 2120 16 15 0</td>
<td>31 26 25 0</td>
</tr>
<tr>
<td>Op Rs1 Rs2 Rd shamt Opx</td>
<td>Op Rs1 Rd immediate</td>
<td>Op Rs1 Rs2/Opx immediate</td>
<td>Op target</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 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
- **Examine needed components and FUs**
- **Map instructions to RTL statements on architected registers and memory**
- **Assemble adequate datapath**
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (e.g. MAR, MBR, ...)
  - 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**

**Instruction Fetch**
- Next PC
- Memory Access
- Write Back

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

**Instruction Decode**
- Reg. Fetch
- Instr. Decode
- Execute
- Addr. Calc

**Execute**
- ALU
- Data Memory

**Memory Access**
- Memory
- Reg File
- MUX

**Write Back**
- WB Data
- Next PC

IF/ID
- IF
- ID
- EX/MEM

ID/EX
- ID
- EX
- MEM/WB

Pipeline registers

IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR\_rs];
B <= Reg[IR\_rt]
rslt <= A \ op \ IRop \ B
WB <= rslt
Reg[IR\_rd] <= WB

Inst. Set Processor Controller

**Figure A.3, Page A-9**

IF/ID
- IF
- ID
- EX/MEM

ID/EX
- ID
- EX
- MEM/WB

Pipeline registers

IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR\_rs];
B <= Reg[IR\_rt]
rslt <= A \ op \ IRop \ B
WB <= rslt
Reg[IR\_rd] <= WB

**Visualizing Pipelining**

**Figure A.2, Page A-8**

**Instruction Fetch**
- Next PC
- Memory Access
- Write Back

**Instruction Decode**
- Reg. Fetch
- Instr. Decode
- Execute
- Addr. Calc

**Execute**
- ALU
- Data Memory

**Memory Access**
- Memory
- Reg File
- MUX

**Write Back**
- WB Data
- Next PC

WS <= r
 RS <= IR
 LD <= Mem[r]

**Shows BW vs. Latency**

**Time (clock cycles)**

Cycle 1
- I fetch
- Reg
- DMem
- Reg

Cycle 2
- I fetch
- Reg
- DMem
- Reg

Cycle 3
- I fetch
- Reg
- DMem
- Reg

Cycle 4
- I fetch
- Reg
- DMem
- Reg

Cycle 5
- I fetch
- Reg
- DMem
- Reg

Cycle 6
- I fetch
- Reg
- DMem
- Reg

Cycle 7
- I fetch
- Reg
- DMem
- Reg

Cycle 8
- I fetch
- Reg
- DMem
- Reg

Cycle 9
- I fetch
- Reg
- DMem
- Reg
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**

(Figure A.4, Page A-14)

![Time (clock cycles) Diagram](image)

**Speed Up Equation for Pipelining**

\[
\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}
\]

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

For simple RISC pipeline, CPI = 1:

\[
\text{Speedup} = \frac{\text{Pipeline depth} \times \text{Cycle Time}_{\text{unpipelined}}}{1 + \text{Pipeline stall CPI} \times \text{Cycle Time}_{\text{pipelined}}}
\]
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
  \[
  \text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}} \\
  = \text{Pipeline Depth}
  \]
  \[
  \text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05} \\
  = \frac{\text{Pipeline Depth}(1/1.4) \times 1.05}{0.75 \times \text{Pipeline Depth}} \\
  = \frac{0.75 \times \text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
  \]
• Machine A is 1.33 times faster

Three Generic Data Hazards

• Read After Write (RAW)
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

\[
\begin{align*}
\text{I:} & \quad \text{add } r1, r2, r3 \\
\text{J:} & \quad \text{sub } r4, r1, r3
\end{align*}
\]
• 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\textsubscript{J} writes operand before Instr\textsubscript{I} reads it

\[
\begin{align*}
\text{I:} & \quad \text{sub } r4, r1, r3 \\
\text{J:} & \quad \text{add } r1, r2, r3 \\
\text{K:} & \quad \text{mul } r6, r1, r7
\end{align*}
\]
• 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.

  \[\text{I: sub } r1, r4, r3\]
  \[\text{J: add } r1, r2, r3\]
  \[\text{K: mul } r6, r1, r7\]

- 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

**Figure A.7, Page A-19**

**Time (clock cycles)**

\[\text{Instr. Order} \]
\[\text{add } r1, r2, r3\]
\[\text{sub } r4, r1, r3\]
\[\text{and } r6, r1, r7\]
\[\text{or } r8, r1, r9\]
\[\text{xor } r10, r1, r11\]

---

HW Change for Forwarding

**Figure A.23, Page A-37**

What circuit detects and resolves this hazard?

---

Forwarding to Avoid LW-SW Data Hazard

**Figure A.8, Page A-20**

**Time (clock cycles)**

\[\text{Instr. Order} \]
\[\text{add } r1, r2, r3\]
\[\text{lw } r4, 0(r1)\]
\[\text{sw } r4, 12(r1)\]
\[\text{or } r8, r6, r9\]
\[\text{xor } r10, r9, r11\]
Data Hazard Even with Forwarding

Figure A.9, Page A-21

Time (clock cycles)

Instr. Order

I

lw r1, 0(r2)

sub r4, r1, r6

and r6, r1, r7

or r8, r1, r9

Load use dependency

Data Hazard Even with Forwarding

(Similar to Figure A.10, Page A-21)

Time (clock cycles)

Instr. Order

lw r1, 0(r2)

sub r4, r1, r6

and r6, r1, r7

or r8, r1, r9

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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fast code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb, b</td>
<td>LW Rb, b</td>
</tr>
<tr>
<td>LW Rc, c</td>
<td>LW Rc, c</td>
</tr>
<tr>
<td>ADD Ra, Rb, Rc</td>
<td>LW Re, e</td>
</tr>
<tr>
<td>SW a, Ra</td>
<td>ADD Ra, Rb, Rc</td>
</tr>
<tr>
<td>LW Re, e</td>
<td>LW Rf, f</td>
</tr>
<tr>
<td>LW Rf, f</td>
<td>SW a, Ra</td>
</tr>
<tr>
<td>SUB Rd, Re, Rf</td>
<td>SUB Rd, Re, Rf</td>
</tr>
<tr>
<td>SW d, Rd</td>
<td>SW d, Rd</td>
</tr>
</tbody>
</table>

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"?

---

**Pipelined MIPS Datapath**

Figure A.24, page A-38

- Interplay of instruction set design and cycle time.

---

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

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>Scheme</th>
<th>Stalls</th>
<th>Pipeline depth</th>
<th>Branch CPI speedup</th>
<th>Speedup vs. unipipelined stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpipelined</td>
<td>31.603</td>
<td>11.0</td>
<td>1.0</td>
<td>31.603</td>
</tr>
<tr>
<td>Stall taken</td>
<td>11.204</td>
<td>11.410</td>
<td>1.36</td>
<td>11.204</td>
</tr>
<tr>
<td>Stall not taken</td>
<td>11.144</td>
<td>11.410</td>
<td>1.35</td>
<td>11.144</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.51</td>
<td>104.514</td>
<td>1.49</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Scheduling Branch Delay Slots (Fig A.14)

A. From before branch
```
add $1,$2,$3
if $2=0 then
  delay slot
```

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

C. From fall through
```
add $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

- 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