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)

Register-Register

| 31 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Op | Rs1| Rs2| Rd |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Register-Immediate

| 31 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Op | Rs1| Rd | immediate |

Branch

| 31 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Op | Rs1| Rd | immediate |

Jump / Call

| 31 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Op |    |    | target |

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

- 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 (e.g. 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**
- Next PC

**Instr. Decode Reg. Fetch**
- Memory Access
- Write Back

**Execute Addr. Calc**
- Memory

**Write Back**
- ALU

### Inst. Set Processor Controller

- **IR** <= mem[PC];
- **PC** <= PC + 4
- **A** <= Reg[IR<sub>rs</sub>];
- **B** <= Reg[IR<sub>rt</sub>];
- r <= A op<sub>IRop</sub> B
- **Reg[IR<sub>rd</sub>]** <= WB
- **WB** <= r

**Visualizing Pipelining**

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

**Time (clock cycles)**

**Cycle 1**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 2**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 3**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 4**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 5**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 6**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 7**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Instruction Order**

**Cycle 4**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 5**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 6**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg

**Cycle 7**
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- Reg
- DMem
- Reg
- Ifetch
- 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

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

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 1</td>
</tr>
<tr>
<td>Instr 1</td>
</tr>
</tbody>
</table>

- **Load**
- **Instr 1**
- **Instr 2**
- **Stall**
- **Instr 3**

---

### Performance of Pipelines 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**

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

- **Ideal CPI is 1**

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

  \[
  \text{CPI unpipelined} = \text{Ideal CPI} \times \text{pipeline depth}
  \]
Performance of Pipelines with Stalls

• Lets 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….

Performance of Pipelines with Stalls

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

\[
\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}} = \frac{1}{1 + \frac{\text{Pipeline stall cycles per instruction}}{\text{Clock cycle time unpipelined}}} \times \frac{\text{Clock cycle time unpipelined}}{\text{Clock cycle time pipelined}}
\]

• If all pipeline stages are balanced:

\[
\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}} \\
\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
\]

Performance of Pipelines with Stalls

Example: Dual-port vs. Single-port

• Machine A: Dual ported memory (“Harvard Architecture”)
• Machine B: Single ported memory and the clock rate is 1.05 times faster
• Ideal CPI = 1 for both
• Loads are 40% of instructions executed

Average instruction time_A = CPI \times \text{Clock cycle time} = \text{Clock cycle time}
Average instruction time_B = 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}

• Machine A is 1.3 times faster
Data Hazard on R1
Figure A.6, Page A-17

Time (clock cycles)

\[ \text{Instr. Order} \]

\[ \begin{align*}
\text{add} & \ r_1, r_2, r_3 \\
\text{sub} & \ r_4, r_1, r_3 \\
\text{and} & \ r_6, r_1, r_7 \\
o r & \ r_8, r_1, r_9 \\
xor & \ r_{10}, r_1, r_{11}
\end{align*} \]

Forwarding to Avoid Data Hazard
Figure A.7, Page A-19

Time (clock cycles)

\[ \text{Instr. Order} \]

\[ \begin{align*}
\text{add} & \ r_1, r_2, r_3 \\
\text{sub} & \ r_4, r_1, r_3 \\
\text{and} & \ r_6, r_1, r_7 \\
o r & \ r_8, r_1, r_9 \\
xor & \ r_{10}, r_1, r_{11}
\end{align*} \]

Three Generic Data Hazards

- **Read After Write (RAW)**
  \( \text{Instr}_J \) tries to read operand before \( \text{Instr}_I \) writes it

  \[ \begin{align*}
  \text{I: } & \ \text{add} \ r_1, r_2, r_3 \\
  \text{J: } & \ \text{sub} \ r_4, r_1, r_3
  \end{align*} \]

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

Three Generic Data Hazards

- **Write After Read (WAR)**
  \( \text{Instr}_J \) writes operand before \( \text{Instr}_I \) reads it

  \[ \begin{align*}
  \text{I: } & \ \text{sub} \ r_4, r_1, r_3 \\
  \text{J: } & \ \text{add} \ r_1, r_2, r_3 \\
  \text{K: } & \ \text{mul} \ r_6, r_1, r_7
  \end{align*} \]

  Called an “anti-dependence” by compiler writers. This results from reuse of the name “\( r_1 \)”.

- 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 \textit{before} Instr\textsubscript{i} writes it.

- I: sub \texttt{r1}, \texttt{r4}, \texttt{r3}
- J: add \texttt{r1}, \texttt{r2}, \texttt{r3}
- K: mul \texttt{r6}, \texttt{r1}, \texttt{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

HW Change for Forwarding

[Figure A.23, Page A-37]

Forwarding to Avoid LW-SW Data Hazard

[Figure A.8, Page A-20]

Data Hazard Even with Forwarding

[Figure A.9, Page A-21]
Data Hazard Even with Forwarding
(Similar to Figure A.10, Page A-21)

Time (clock cycles)

lw r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or r8,r1,r9

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 producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \(a, b, c, d, e, \) and \(f\) in memory.

Slow code:

Fast code:

Slow code:

Fast code:

Compiler optimizes for performance. Hardware checks for safety.

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

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
Scheduling Branch Delay Slots (Fig A.14)

- 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