Instruction Execution - Pipelines

♦ Execute billions of instructions, so **throughput** is what matters
♦ What is desirable in instruction sets for pipelining?
  - Variable length instructions vs. all instructions same length?
  - Memory operands part of any operation vs. memory operands only in loads or stores?
  - Register operand in various places in instruction format vs. registers located in same place?
♦ Conclusion: RISC is easier to pipeline
"MIPS" - A "Typical" RISC

- 32-bit fixed length instruction (3 formats)
- Memory access only via load/store instructions
- 32 32-bit GPR (R0 contains zero)
- 32 32-bit FPR - 16 64-bit double-precision
  - DP uses a pair
- 3-address, reg-reg arithmetic instruction; registers in same place in instruction format
- Single address mode for load/store: base + displacement
- Simple branch conditions; addressing modes: PC relative and register indirect
- Delayed branch

some versions of SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, DSP processors

Data Formats and Memory Addresses

Data formats:
- Bytes, Half words, words and double words

- Byte addressing
  - Big Endian vs. Little Endian

- Word alignment
  - Byte addressable memory
  - A word address can begin only at 0, 4, 8, ...
MIPS Instruction Set Architecture

- **Instruction Categories**
  - Load/Store
  - Computational (Fixed-point etc)
  - Floating-Point
  - Jump and Branch
  - Special

3 Instruction Formats: all 32 bits wide

- | OP | rs | rt | rd | sa | funct |
- |-----|-----|-----|-----|-----|--------|
- | OP  | rs  | rd  | immediate |
- | OP  | jump target |

MIPS Instruction Formats

- **ALU**
  - Opcode rs rt rd 0 func
  - rd ← (rs) func (rt)

- **ALUi**
  - Opcode rs rt immediate
  - rt ← (rs) op immediate

- **Mem**
  - Opcode rs rt displacement
  - M[(rs) + displacement]

- | Opcode | rs | rt | offset |
- |--------|----|----|--------|
- | BEQZ, BNEZ |
- | JR, JALR |
- | J, JAL |
**Instruction Execution**

Execution of a MIPS instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)

**Control Signals**

**instr fetch:**
- MA ← PC
- A ← PC
- PC ← A + 4
- IR ← Memory

**ALU:**
- A ← Reg[rs]
- B ← Reg[rt]
- Reg[rd] ← func(A,B)

**ALUi:**
- A ← Reg[rs]
- B ← Imm  \(\text{sign extension} \ldots\)
- Reg[rt] ← Opcode(A,B)

**Alternative: Microinstructions**
**Control Signals (Microinstructions) - cont’d**

- **LW:**
  - A ← Reg[rs]
  - B ← Imm
  - MA ← A + B
  - Reg[rt] ← Memory

- **beqz:**
  - A ← Reg[rs]
  - If zero?(A) then go to bz-taken
  - instruction fetch

- **bz-taken:**
  - A ← PC
  - B ← Imm << 2
  - PC ← A + B

- **J:**
  - A ← PC
  - B ← IR
  - PC ← JumpTarg(A,B)

\[ \text{JumpTarg}(A,B) = \{A[31:28],B[25:0],00\} \]

**Microarchitecture: Implementation of an ISA**

[Diagram showing microarchitecture with controller, status lines, control signals, data path, and bus connections.]
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg=yes; ldMA = yes
B ← Reg[rt] means RegSel = rt; enReg=yes; ldB = yes

Can this be pipelined?

Execution Cycle - pipeline stages

1. Obtain instruction from program storage
2. Determine required actions and instruction size
3. Locate and obtain operand data
4. Compute result value or status
5. Deposit results in storage
6. Determine successor instruction
5 Steps of MIPS Datapath w/o pipelining

5 Steps of MIPS Datapath w/pipelining

Instruction fields in each pipeline stage
Visualizing Pipelining

Time (clock cycles)

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

I-Fetch ALU DMem Reg

I-Fetch ALU DMem Reg

I-Fetch ALU DMem Reg

I-Fetch ALU DMem Reg

I-Fetch ALU DMem Reg

I-Fetch ALU DMem Reg

I-Fetch ALU DMem Reg

Visualizing Pipelining – 2nd way

I-Fetch (IF) Decode, Reg. Fetch (ID) Execute (EX) Memory (MA) Write - Back (WB)

Time (clock cycles)

| time  | I0 | I1 | I2 | I3 | I4 | I5 | I6 | I7 | ...
|------|----|----|----|----|----|----|----|----|------
| IF   | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9   |
| ID   | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9   |
| EX   | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9   |
| MA   | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9   |
| WB   | I1 | I2 | I3 | I4 | I5 | I6 | I7 | I8 | I9   |
Calculating CPI - Example

Bus-based machine

<table>
<thead>
<tr>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 cycles</td>
<td>5 cycles</td>
<td>10 cycles</td>
</tr>
</tbody>
</table>

Unpipelined n-stage machine

<table>
<thead>
<tr>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 instructions, 22 cycles, CPI=7.33</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipelined machine

<table>
<thead>
<tr>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 instructions, 3 cycles, CPI=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time = \( \frac{\text{Program}}{\text{Instructions}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}} \)

Pipelined Datapath

\[ t_C > \max \{ t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW} \} \quad ( = t_{DM} \text{ probably}) \]
Technology Assumptions

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

Thus, the following timing assumption is made

\[ t_{IM} \approx t_{RF} \approx t_{ALU} = t_{DM} \approx t_{RW} \]

MIPS pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF/ID.IR ← Mem[PC]; if/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) &amp; FX/MEM.Aluoutput) else [PC+4]);</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A ← #Reg[IF/ID.IR[rs]]; ID/EX.A ← Regs[1x/ID.IR[rt]]; ID/EX.NPC ← IF/ID.IR; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[Immmediate field]);</td>
</tr>
<tr>
<td>EX</td>
<td>EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUoutput ← ID/EX.IR; ID/EX.A func 10/EX.B; EX/MEM.ALUoutput ← 10/EX.IR; ID/EX.IR ← ID/EX.IRR;</td>
</tr>
<tr>
<td>ALU</td>
<td>ALU Instruction Load or store instruction Branch Instruction</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUoutput ← EX/MEM.ALUoutput; MEM/WB.IR ← (EX/MEM.IR); MEM/WB.LMD ← Mem[EX/MEM.ALUoutput]; MEM[EX/MEM.ALUoutput] ← FX/MEM.B;</td>
</tr>
<tr>
<td>WB</td>
<td>Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUoutput; Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUoutput; For load only: Regs[MEM/WB.IR[rd]] ← Mem[MEM/WB.IR[rd]]; MEM/WB.LMD ← Mem[MEM/WB.IR[rd]]; Mem[MEM/WB.IR[rd]] ← FX/MEM.B;</td>
</tr>
</tbody>
</table>
Instruction pipeline speedup

The pipeline "forces" all instructions to go through all five stages

- \( P4 = \% \) of instructions requiring 4 cycles (e.g., ALU)
- \( P3 = \% \) of instructions requiring 3 cycles (e.g., Branch)
- \( P5 = \% \) of instructions requiring 5 cycles (e.g., Load) = ?

\[
CPI_{\text{unpipelined}} = P4 \times 4 + P3 \times 3 + P5 \times 5
\]

\( C4 \), \( CPI_{\text{unpipelined}} = 0.5 \times 4 + 0.2 \times 3 + 0.3 \times 5 = 4.1 \)

\[\begin{align*}
CPI_{\text{pipelined}} &= 1 \text{ (ideally)} \\
\text{Speedup} &= \frac{CPI_{\text{unpipelined}}}{CPI_{\text{pipelined}}} \times \frac{T_{\text{unpipelined}}}{T_{\text{pipelined}}} < 4.1 \times 5 \text{ (ideal speedup)}
\end{align*}\]

ExTime = (# of instr.) * CPI * T

Instruction pipelines are not ideal

- Instructions interact with each other in pipeline
- **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: An instruction in the pipeline may need a resource being used by a previous instruction in the pipeline (e.g., address calculation for one instruction using the same adder used for addition in another instruction)
  - **Data hazards**: Instruction depends on (data) result of prior instruction still in the pipeline:
    \[
    A \leftarrow B + C \\
    D \leftarrow A \times B
    \]
  - **Control hazards**: Branches and jumps
  - **Interrupts/exceptions**
- **Issues**:
  - How to detect?
  - How to minimize the penalty?
Structural Hazards - one Memory Port

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><strong>Instr 1</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instr 2</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instr 3</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instr 4</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One Memory Port/Structural Hazards

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><strong>Instr 1</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instr 2</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stall</strong></td>
<td>Bubble</td>
<td>Bubble</td>
<td>Bubble</td>
<td>Bubble</td>
<td>Bubble</td>
<td></td>
</tr>
<tr>
<td><strong>Instr 3</strong></td>
<td>Fetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Resolving Structural Hazards

- Structural hazards occur when two instructions need the same hardware resource at the same time.
  - Can resolve in hardware by stalling newer instruction till older instruction finishes with resource.
- A structural hazard can always be avoided by adding more hardware to the design.
  - E.g., if two instructions both need a port to memory at the same time, could avoid hazard by adding a second port to memory.
- Our 5-stage pipe has no structural hazards by design.

Data Hazards

I1: \( r_3 \leftarrow r_2 + 10 \)  
I2: \( r_4 \leftarrow r_3 + 17 \)  

\( r_3 \) is stale