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

### Registers

<table>
<thead>
<tr>
<th>R0 - R31</th>
<th>PC</th>
<th>IR</th>
</tr>
</thead>
</table>

#### 3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rd</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td>jump target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

MIPS Instruction Formats

- **ALU**
  - 6 5 5 5 5 6
  - rs rt rd 0 func
  - rd ← (rs) func (rt)

- **ALU_i**
  - 6 5 5 16
  - opcode rs rt 0 immediate

- **Mem**
  - 6 5 5 16
  - opcode rs rt displacement
  - M[(rs) + displacement]

- 6 5 5 16
  - opcode rs offset
  - BEQZ, BNEZ

- 6 5 16
  - opcode rs
  - JR, JALR

- 6 26
  - opcode offset
  - 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 (Microinstructions)**

**instr fetch:**

- \( MA \leftarrow PC \)
- \( A \leftarrow PC \)
- \( IR \leftarrow \text{Memory} \)
- \( PC \leftarrow A + 4 \)

**ALU:**

- \( A \leftarrow \text{Reg}[rs] \)
- \( B \leftarrow \text{Reg}[rt] \)
- \( \text{Reg}[rd] \leftarrow \text{func}(A,B) \)

**ALUi:**

- \( A \leftarrow \text{Reg}[rs] \)
- \( B \leftarrow \text{Imm} \)  \( \text{sign extension} \ldots \)
- \( \text{Reg}[rt] \leftarrow \text{Opcode}(A,B) \)
Control Signals (Microinstructions) – cont’d

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

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

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

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

Microarchitecture: Implementation of an ISA

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

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg=yes; IdMA=yes
B ← Reg[rt] means RegSel = rt; enReg=yes; IdB=yes

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
   Determine successor instruction
5 Steps of MIPS Datapath w/o pipelining

- Instruction Fetch
- Instr. Decode
- Reg. Fetch
- Execute
- Addr. Calc
- Memory Access
- Write Back

- Local decoded instruction fields for each phase / pipeline stage
Visualizing Pipelining

Time (clock cycles)

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

I-Fetch | Reg | DMem | Reg | I-Fetch | Reg | DMem | Reg | I-Fetch | Reg | DMem | Reg

Reg | Reg | ALU | DMem | Reg | Reg | ALU | DMem | Reg | Reg | ALU | DMem

Visualizing Pipelining – 2nd way

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

Time (clock cycles)

Resources

<table>
<thead>
<tr>
<th>time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>I₁</td>
<td>I₁</td>
<td>I₁</td>
<td>I₁</td>
<td>I₁</td>
</tr>
<tr>
<td>t1</td>
<td>I₂</td>
<td>I₂</td>
<td>I₂</td>
<td>I₂</td>
<td>I₂</td>
</tr>
<tr>
<td>t2</td>
<td>I₃</td>
<td>I₃</td>
<td>I₃</td>
<td>I₃</td>
<td>I₃</td>
</tr>
<tr>
<td>t3</td>
<td>I₄</td>
<td>I₄</td>
<td>I₄</td>
<td>I₄</td>
<td>I₄</td>
</tr>
<tr>
<td>t4</td>
<td>I₅</td>
<td>I₅</td>
<td>I₅</td>
<td>I₅</td>
<td>I₅</td>
</tr>
<tr>
<td>t5</td>
<td>I₆</td>
<td>I₆</td>
<td>I₆</td>
<td>I₆</td>
<td>I₆</td>
</tr>
<tr>
<td>t6</td>
<td>I₇</td>
<td>I₇</td>
<td>I₇</td>
<td>I₇</td>
<td>I₇</td>
</tr>
<tr>
<td>t7</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

ECE568/Koren Part.2.15
Adapted from UCB and other sources
Copyright UCB & Morgan Kaufmann
Calculating CPI - Example

Microcoded machine
7 cycles 5 cycles 10 cycles
Inst 1 Inst 2 Inst 3

Unpipelined machine
3 instructions, 22 cycles, CPI=7.33

Pipelined machine
3 instructions, 3 cycles, CPI=1

Pipelined Datapath

\[ t_c > \max \{ t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW} \} = t_{DM} \text{ probably} \]
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 reasonable

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any Instruction</th>
</tr>
</thead>
</table>
| IF    | IF/ID.IR ← Mem[PC];  
IF/ID.NPC,PC ← (if ((EX/MEM.opcode -- branch) & FX/MEM. 
ALUoutput) else [PC+4]); |
| ID    | ID/EX.A ← Regs[IF/ID.1R[rt]];  
ID/EX.B ← Regs[IF/ID.1R[rt]];  
ID/EX.NPC ← IF/ID.NPC;  
ID/EX.IR ← IF/ID.IR;  
ID/EX.Imm ← sign-extend[IF/ID.1R[immediate field]]; |
| ALU   | ALU Instruction | Load or store instruction | Branch instruction |
| EX    | EX/MEM.IR ← ID/EX.IR;  
EX/MEM.ALUOutput ←  
ID/EX.A func ID/EX.B;  
or  
EX/MEM.ALUOutput ←  
ID/EX.A op ID/EX.Imm; |  
ex/MEM.IR ← ID/EX.IR  
ex/MEM.ALUOutput ←  
ID/EX.A + ID/EX.Imm; |  
ex/MEM.ALUOutput ←  
ID/EX.IR +  
(ID/EX.Imm << ?);  
ex/MEM.B ← ID/EX.B; |  
ex/MEM.cond ←  
(ID/EX.A == 0); |
| MTM   | 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];  
or  
Mem[EX/MEM.ALUOutput] ←  
FX/MEM.B; |
| WB    | Regs[MEM/WB.IR[rd]] ←  
MEM/WB.ALUOutput;  
or  
Regs[MEM/WB.IR[rt]] ←  
MEM/WB.ALUOutput; | For load only:  
Regs[MEM/WB.IR[rd]] ←  
MEM/WB.LMD;  
or  
Mem[EX/MEM.ALUOutput] ←  
FX/MEM.B; |
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) = ?

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

e.g., \( \text{CPI}_{\text{unpipelined}} = 0.5 \times 4 + 0.2 \times 3 + 0.3 \times 5 = 4.1 \)

\[\text{CPI}_{\text{pipelined}} = 1 \text{ (ideally)}\]

\[
\text{ExTime} = (\# \text{ of instr.}) \times \text{CPI} \times T
\]

\[\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{T_{\text{unpipelined}}}{T_{\text{pipelined}}} < 4.1 < 5 \text{ (ideal speedup)}\]

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., operand address calculation using the same fixed-point adder used for addition)
  - 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)

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

Load
Instr 1
Instr 2
Instr 3
Instr 4

One Memory Port/Structural Hazards

Time (clock cycles)

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

Load
Instr 1
Instr 2
Instr 3

Stall

Bubble

Bubble

Bubble

Bubble

Bubble
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 instructions until the older instruction finishes with the resource.
- A structural hazard can always be avoided by adding more hardware to the design.
  - For example, if two instructions both need a port to memory at the same time, adding a second port could avoid the hazard.
- Our 5-stage pipeline has no structural hazards by design.
  - Thanks to the MIPS ISA, which was designed for pipelining.

Data Hazards

I1: r0 ← r0 + 10
I2: r4 ← r1 + 17

...
Data Hazard on R1

Time (clock cycles)

IF ID/RF EX MEM WB

Instr. Order

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11

Pipeline control design
(avoid structural hazards)

“Effective control for pipelined computers,” by Davidson et al. (available on the web page)

Reservation Table: Task A

<table>
<thead>
<tr>
<th></th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unit s0 is the busiest
3 out of 6 time units
throughput ≤ 2 instr./6 cycles
1/6 ≤ throughput ≤ 1/3
How can we find the real throughput?

Define: Pipeline Collision Vector (PCV)

PCV = C0 C1 ... C(k-1); k = # of cycles (t0, ..., t{k-1})
Ci=1 if a 2nd task initiated i cycles after the 1st will result in a collision; Ci=0 otherwise.
Reservation Table for a multiplier


Greedy control