Quick MIPS ISA overview
5 stage pipelining
Structural and Data Hazards
Forwarding
Branch Schemes
Exceptions and Interrupts
Conclusion

A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg 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>31 26 25 21 12 16 15 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register-Immediate</th>
<th>31 26 25 21 12 16 15 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch</th>
<th>31 26 25 21 12 16 15 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
</tr>
</tbody>
</table>

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

Datapath vs Control

- Datapath: Storage, FU, interconnect sufficient to perform 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 register transfer level (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
  - Map each instruction to sequence of RTLs
  - Collate sequences into symbolic controller state transition diagram (STD)
  - Lower symbolic STD to control points
  - Implement controller

Outline

- Quick MIPS ISA overview
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion
5 Steps of MIPS Datapath

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

**Instruction Fetch**
- Next PC
- IR <= mem[PC]; PC <= PC + 4
- Next SEQ PC

**Instr. Decode**
- Execute
- Memory Access
- Write Back

**Reg. Fetch**
- IR <= mem[PC]; PC <= PC + 4
- Reg[IR] <= Reg[IR]

**Addr. Calc**
- A <= Reg[IR]
- B <= Reg[IR]
- rslt <= A op IRop B

**ALU**
- A <= Reg[IR]
- B <= Reg[IR]
- rslt <= A op IRop B

**Data Memory**
- Memory Fetch
- Memory Read
- Memory Write

**Inst. Set Processor Controller**

**Watch**
- JSR
- JR
- br
- jmp
- opFetch-DCD
- ST

**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).

**Visualizing Pipelining**

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

**Time (clock cycles)**
- Cycle 1: IR <= mem[PC]; PC <= PC + 4
- Cycle 2: Reg[IR] <= Reg[IR]
- Cycle 3: A <= Reg[IR]; B <= Reg[IR]
- Cycle 4: rslt <= A op IRop B
- Cycle 5: WB <= rslt
- Cycle 6: Reg[IR] <= WB
- Cycle 7: Next PC

**Pipelining is not quite that easy!**

- Data stationary control
  - local decode for each instruction phase / pipeline stage
### One Memory Port/Structural Hazards

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

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
<th>Instr Order</th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 1</td>
<td></td>
<td>I</td>
<td>D</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 2</td>
<td></td>
<td>I</td>
<td>D</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 3</td>
<td></td>
<td>I</td>
<td>D</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 4</td>
<td></td>
<td>I</td>
<td>D</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How do you “bubble” the pipe?

Can be represented in this fashion

### Speed Up Equation for Pipelining

\[ CPI_{\text{piped}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst} \]

\[ \text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \]

\[ \text{Cycle Time}_{\text{unpiped}} / \text{Cycle Time}_{\text{piped}} \]

For simple RISC pipeline, CPI = 1:

\[ \text{Speedup} = \frac{\text{Pipeline depth}}{\text{Cycle Time}_{\text{unpiped}} / \text{Cycle Time}_{\text{piped}}} \]

### 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
  - SpeedUpA = Pipeline Depth / 1.00
  - SpeedUpB = Pipeline Depth / 0.75 x Pipeline Depth
  - SpeedUpB = 0.75 x Pipeline Depth
- Machine A is 1.33 times faster

### Data Hazard on R1

(Similar to Figure A.6, Page A-17)

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
<th>Instr Order</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Three Generic Data Hazards

- **Read After Write (RAW)**
  - Instr$_I$ tries to read operand before Instr$_J$ writes it.
  - Example: $I$: add $r1$, $r2$, $r3$
  - $J$: sub $r4$, $r1$, $r3$
  - Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

- **Write After Read (WAR)**
  - Instr$_J$ writes operand before Instr$_I$ reads it.
  - Called an “anti-dependence” by compiler writers.
  - Example: $I$: sub $r4$, $r1$, $r3$
  - $J$: add $r1$, $r2$, $r3$
  - 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

- **Write After Write (WAW)**
  - Instr$_J$ writes operand before Instr$_I$ writes it.
  - Called an “output dependence” by compiler writers.
  - Example: $I$: sub $r4$, $r1$, $r3$
  - $J$: add $r1$, $r2$, $r3$
  - 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

- Time (clock cycles)
- Instructions: $I$: add $r1$, $r2$, $r3$  $J$: sub $r4$, $r1$, $r3$  $K$: mul $r6$, $r1$, $r7$
- Forwarding diagram shows data flow and resolution of hazards.

Forwarding to Avoid LW–SW Data Hazard

- Time (clock cycles)
- Instructions: $I$: add $r1$, $r2$, $r3$  $J$: lw $r4$, (0($r1$))  $K$: sw $r4$, 12($r1$)  $L$: xor $r10$, $r9$, $r11$
- Forwarding diagram highlights data movement and resolution steps.

What circuit detects and resolves this hazard?

- HW Change for Forwarding: Diagram shows hardware modifications for forwarding, with data and control signals highlighted.
How is this detected?

Software Scheduling to Avoid Load Hazards

Review

Quantify and summarize performance
- Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, 1 point fail danger
- 252 Administrivia
- 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”?

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

#### #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 downsides:
  - As processors 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

#### Scheduling Branch Delay Slots (Fig A.14)
- **A:** From before branch
  - `add $1,$2,$3 if $2=0 then delay slot` becomes `add $1,$2,$3 if $2=0 then add $1,$2,$3

- **B:** From branch target
  - `add $1,$2,$3 if $1=0 then delay slot` becomes `add $1,$2,$3 if $1=0 then add $1,$2,$3

- **C:** From fall through
  - `add $1,$2,$3 if $1=0 then delay slot` becomes `add $1,$2,$3 if $1=0 then add $1,$2,$3

- **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

#### Examples
- **LD** `R1, 45(R2)`
- **DADD** `R5, R6, R7`
- **DSUB** `R8, R6, R7`
- **OR** `R9, R6, R7`

- **Situation:**
  - No dependence

- **Situation:**
  - Dependence requiring stall

- **Situation:**
  - Dependence overcome by forwarding

- **Situation:**
  - Dependence with accesses in order
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken.

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Branch CPI speedup v.</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.60</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.20</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
</tr>
</tbody>
</table>

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

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

### Problems with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution.
  - Examples: divide by zero, undefined opcode
- **Interrupt**: Hardware signal to switch the processor to a new instruction stream.
  - Example: a sound card interrupts when it needs more audio output samples (an audio 'click' happens if it is left waiting).
- **Problem**: It must appear that the exception or interrupt must appear between 2 instructions \( I_i \) and \( I_{i+1} \).
  - The effect of all instructions up to and including \( I_i \) is totally complete.
  - No effect of any instruction after \( I_i \) can take place.
  - The interrupt (exception) handler either aborts program or restarts at instruction \( I_{i+1} \).

\[ \text{Speed Up} \leq \text{Pipeline Depth} \]

\[ \text{Speed Up} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_\text{unpipelined}}{\text{Cycle Time}_\text{pipelined}} \]

### And In Conclusion: Pipelining

- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:
  
- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity