Lecture 5: Review of MIPS 5-stage Pipeline

Slides adapted and revised from UC Berkeley CS252, Fall 2006

Reading: Textbook (5th edition) Appendix C
Appendix A in 4th edition

Outline

• MIPS – An ISA for Pipelining
• 5 stage pipelining
• Structural and Data Hazards
• Forwarding
• Branch Schemes
• Exceptions and Interrupts
• Conclusion

Example: MIPS Assembly Code

```assembly
; C code:
; c = (a < b) ? (a - b) : (b - a);
; a = MEM(4), b = MEM(8), c = MEM(12)
; R8 = a, R9 = b, R10 = tmp, R11 = c
lw $8, 4($0)    ; load a to reg 8
lw $9, 8($0)    ; load b to reg 9
slt $10, $9, $8 ; set reg 10 if b < a
beq $10, $0, +2 ; no, skip next two
sub $11, $9, $8 ; c = b - a
beq $0, $0, +1 ; skip next
sub $11, $8, $9 ; c = a - b
sw $9, 12($0) ; store c
```

MIPS Instruction Format (32-bit)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register-Register</th>
<th>Register-Immediate</th>
<th>Branch</th>
<th>Jump / Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>Opx</td>
</tr>
<tr>
<td>31 26 25 22 20 16 15 11 10 6 5 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>immediate</td>
</tr>
<tr>
<td>31 26 25 22 20 16 15 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>immediate</td>
</tr>
<tr>
<td>31 26 25 22 20 16 15 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 26 25 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 indirect addressing
• Simple branch conditions
• Delayed branch

Example: MIPS Binary Code

```
100011_00000_01000_000000000000000100   // lw $8, 4($0)
100011_00000_01001_000000000000000100   // lw $9, 8($0)
000000_01010_01000_000000000000000100   // slt $10, $9, $8
000101_01010_00000_000000000000010010   // beq $10, $0, +2
000000_01011_01000_000000000000000100   // sub $11, $9, $8
000101_00000_00000_000000000000000001   // beq $0, $0, +1
000000_01000_01001_01011_00000_100010   // sub $11, $8, $9
101011_00000_01011_000000000000011000   // sw $9, 12($0)
```

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Datapath vs Control

- **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**
  - Architectured 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**

5 Steps of MIPS Datapath

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode</th>
<th>Fetch</th>
<th>Acc</th>
<th>Write</th>
<th>Memory Access</th>
<th>Write Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR &lt;= mem[PC];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC &lt;= PC + 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A &lt;= Reg[IRr];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B &lt;= Reg[IRr];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rslt &lt;= A opIRop B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg[IRrd] &lt;= WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB &lt;= rslt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Steps of MIPS Datapath

![Diagram of the 5 steps of MIPS Datapath](image)
5 Steps of MIPS Datapath

- Instruction Fetch
- Instr. Decode
- Execute
- Memory Access
- Write Back

- Data stationary control
  - local decode for each instruction phase / pipeline stage

One Memory Port/Structural Hazards

- Time (clock cycles)
  - Cycle 1
  - Cycle 2
  - Cycle 3
  - Cycle 4
  - Cycle 5
  - Cycle 6
  - Cycle 7

- Inst. Order
  - Instr 1
  - Instr 2
  - Instr 3
  - Instr 4

Visualizing Pipelining

- Time (clock cycles)
  - Cycle 1
  - Cycle 2
  - Cycle 3
  - Cycle 4
  - Cycle 5
  - Cycle 6
  - Cycle 7

- Inst. Order
  - Instr 1
  - Instr 2
  - Instr 3
  - Instr 4

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
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Speed Up Equation for Pipelining

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

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{average}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{average}}}{\text{Cycle Time}_{\text{pipelined}}}$$
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 period
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}/1.05}
\]

\[
= \frac{0.75 \times \text{Pipeline Depth}}{1.05}\times 1.05
\]

\[
= \frac{0.75 \times \text{Pipeline Depth}}{1.05}
\]

\[
= \frac{\text{Pipeline Depth}}{1.4}
\]

\[
= 0.75 \times \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_A / \text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
\]

- Machine A is 1.33 times faster

Three Generic Data Hazards

- Write After Read (WAR)
  Instr \(_2\) writes operand before Instr \(_1\) reads it
  
  \[
  \text{I}: \text{sub } r4, r1, r3
  \]
  
  \[
  \text{J}: \text{add } r1, r2, r3
  \]
  
  \[
  \text{K}: \text{mul } r6, r1, r7
  \]
  
  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

Data Hazard on R1

<table>
<thead>
<tr>
<th>Time (clock cycles)</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>
</tr>
<tr>
<td>sub r4, r1, r3</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>
</tr>
<tr>
<td>or r8, r1, r9</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>
</tr>
</tbody>
</table>

Three Generic Data Hazards

- Write After Write (WAW)
  Instr \(_2\) writes operand before Instr \(_1\) writes it.
  
  \[
  \text{I}: \text{sub } r1, r4, r3
  \]
  
  \[
  \text{J}: \text{add } r1, r2, r3
  \]
  
  \[
  \text{K}: \text{mul } r6, r1, r7
  \]
  
  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

Three Generic Data Hazards

- Read After Write (RAW)
  Instr \(_2\) tries to read operand before Instr \(_1\) writes it
  
  \[
  \text{I}: \text{add } r1, r2, r3
  \]
  
  \[
  \text{J}: \text{sub } r4, r1, r3
  \]
  
  Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

Forwarding to Avoid Data Hazard

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
<th>I</th>
<th>S</th>
<th>E</th>
<th>M</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</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>
</tr>
<tr>
<td>and r6, r1, r7</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>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What circuit detects and resolves this hazard?

Forwarding to Avoid LW-SW Data Hazard

Data Hazard Even with Forwarding

Control Hazard on Branches Three Stage Stall

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:
- \[ \text{LW Rb}, b \]
- \[ \text{LW Rc}, c \]
- \[ \text{ADD Ra,Rb,Rc} \]
- \[ \text{SW a,Ra} \]
- \[ \text{LW Re}, e \]
- \[ \text{LW Rf}, f \]
- \[ \text{SW d,Rd} \]
- \[ \text{SW d,Rd} \]

Fast code:
- \[ \text{LW Rb}, b \]
- \[ \text{LW Rc}, c \]
- \[ \text{LW Re}, e \]
- \[ \text{ADD Ra,Rb,Rc} \]
- \[ \text{SW a,Ra} \]
- \[ \text{LW Rf}, f \]
- \[ \text{SW a,Ra} \]
- \[ \text{SW d,Rd} \]
- \[ \text{SW d,Rd} \]

Compiler optimizes for performance. Hardware checks for safety.

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

Scheduling Branch Delay Slots (Fig A.14)

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

B. From branch target
   add $4, $5, $6
   if $1=0 then
   delay slot
   becomes
   add $4, $5, $6
   if $1=0 then
   add $4, $5, $6

C. From fall through
   add $8, $9, $10
   if $1=0 then
   delay slot
   becomes
   add $8, $9, $10

- 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

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

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

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

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>Speedup vs. unpipelined</th>
<th>Speedup vs. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.60</td>
<td>3.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
<td>1.33</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
<td>4.41</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
<td>1.45</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
<td>1.45</td>
</tr>
</tbody>
</table>

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 complete
  - No effect of any instruction after \(i_{i+1}\) can take place
- The interrupt (exception) handler either aborts program or restarts at instruction \(i_{i+1}\)

Precise Exceptions in Static Pipelines

Key observation: architected state only change in memory and register write stages.

And In Conclusion: Control and Pipelining

- Just overlap tasks; easy if tasks are independent
- Speed Up \(\leq\) Pipeline Depth; if ideal CPI is 1, then:
  \[
  \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
  \]
- 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
- Next time: Read Appendix C, record bugs online!

Pipelines and Cache

For in-order pipelines: Cache miss?
1. Stall the pipeline
2. Move data from memory to cache
3. Un-stall the pipeline

Pipelines and Cache