CSE502 – Computer Architecture

Lecture 03
Instruction Pipelining (Appendix A and C)

Dr. Ilchul Yoon (icyoon@sunykorea.ac.kr)

Slides adapted from:
Larry Wittie, SBU & John Kubiatowicz, EECS, UC, Berkeley
Outline

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

(From Patterson)
A "Typical" RISC ISA

- 32-bit fixed format instruction (only 3 formats: R,I,J)
- 32 32-bit GPR (R0 contains zero, DP takes pair)
- 3-address, register-register arithmetic instruction
- Single address mode for load/store
  - Simply, base + displacement
  - No memory indirection (since it needs another memory access)
- Simple branch conditions (e.g., single-bit: 0 or not?)
- (Delayed branch - ineffective in deep pipelines, so no longer used)

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

(From Patterson)
Example: MIPS

Register-Register – R Format – Arithmetic operations

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate – I Format – All immediate arithmetic ops

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch – I Format – Moderate relative distance conditional branches

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call – J Format – Long distance jumps

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Offset added to PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(From Patterson)
Data path vs. Control

- **Data path**: Storage, Functional Units, Interconnections 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

(From Patterson)
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 (register transfer language) on architected registers and memory
- Given technology constraints, assemble adequate data path
  - Architected storage mapped to actual storage
  - Function Units (FUs) to do all the required operations
  - Possible additional storage (eg. Internal registers: MAR, MDR, IR, … {Memory Address Register, Memory Data Register, Instruction Register})
  - Interconnect to move information among registers and function units
- Map each instruction to a sequence of RTL operations
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

(From Patterson)
5 Steps of a (pre-pipelined) MIPS Data Path
Figure C.21

Stages: 1
Instruction Fetch

Stages: 2
Instr. Decode
Reg. Fetch

Stages: 3
Execute
Addr. Calc

Stages: 4
Memory Access

Stages: 5
Write Back

IR \leftarrow \text{mem}[PC];

PC \leftarrow PC + 4

\text{Reg[IR}_{rd}] \leftarrow (\text{Reg[IR}_{rs}] \cdot \text{op}_{IR_{op}} \text{Reg[IR}_{rt}] ) \# \text{op is done in stages } 2-5

(From Patterson)
5 Steps of a MIPS Data Path (with pipeline latches)

Figure C.22

Stages:

1. Instruction Fetch
2. Instr. Decode
3. Execute
4. Memory Access
5. Write Back

IR ← mem[PC]; #1
PC ← PC + 4
A ← Reg[IR<sub>rs</sub>]; #2
B ← Reg[IR<sub>rt</sub>]
rs1t ← A op<sub>IRop</sub> B #3
WB ← rs1t #4
Reg[IR<sub>rd</sub>] ← WB #5

(From Patterson)
Instruction Set Processor Controller

IR \leftarrow \text{mem}[\text{PC}];
\text{PC} \leftarrow \text{PC} + 4

Ifetch

A \leftarrow \text{Reg}[\text{IR}_{rs}];
B \leftarrow \text{Reg}[\text{IR}_{rt}]

opFetch-DeCoDe

if bop(A, B)
\text{PC} \leftarrow \text{PC} + \text{IR}_{im}

branch

JAL

JR

jmp

PC \leftarrow \text{IR}_{jaddr}

RR

r \leftarrow A \circ \text{op}_{IRop} B

Reg[IR_{rd}] \leftarrow \text{WB}

JR

If

W B \leftarrow r

Loc

ST

LD

r \leftarrow A + \text{IR}_{im}

Load

Reg[IR_{rd}] \leftarrow \text{WB}

Mem

Reg[IR_{rd}] \leftarrow \text{WB}

Save

Reg[IR_{rd}] \leftarrow \text{WB}

(From Patterson)
5 Steps of a MIPS Data Path (with pipeline latches)

Figure C.22

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

(From Patterson)
Visualizing Pipelining
Figure C.3

(From Patterson)
Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**
    - Hardware cannot support this combination of instructions
    - e.g., having a single person to fold and put clothes away at same time
  - **Data hazards**
    - Instruction depends on result of prior instruction still in the pipeline
    - e.g., having a missing sock in a later wash; cannot put away
  - **Control hazards**
    - Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

(From Patterson)
One Memory-Port / Structural Hazards

Figure C.4

Time (clock cycles)

Instr. Order

Instr 4

Instr 3

Instr 2

Instr 1

Load

Cycle 1

Cycle 2

Cycle 3

Cycle 4

Cycle 5

Cycle 6

Cycle 7

Cycle 8

Ifetch

Reg

DMem

Reg

Ifetch

Reg

DMem

Reg

Ifetch

Reg

DMem

Reg

Ifetch

Reg

DMem

Reg

(From Patterson)
One Memory-Port / Structural Hazards
Similar to Figure C.5

In
str
Order

Instr 1
Instr 2
Instr 3
Load

Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8

How do you “bubble” the pipe?
(From Patterson)
Code Speed Up Equation for Pipelining

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

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

For simple RISC pipeline, $\textbf{Ideal CPI} = 1$,

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Assume stages are perfectly balanced, Ignore cycle time overhead. Then,

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}}$$

(From Patterson)
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
- Assume loads are 20% of instructions executed

\[
\text{Speedup}_A = \frac{\text{Pipeline depth}}{1 + 0} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]
\[
\text{Speedup}_B = \frac{\text{Pipeline depth}}{1 + 0.2 \times 1} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\left(\frac{\text{Cycle Time}_{\text{pipelined}}}{1.05}\right)}
\]

\[
\frac{\text{Speedup}_A}{\text{Speedup}_B} = \frac{1.2}{1.05} = 1.14
\]

**Machine A is 1.14 times faster!!!**

(From Patterson)
Data Hazard on Register R1 (If No Forwarding)

Figure C.6

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>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td>DMem</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>add r1, r2, r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td></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>
<td></td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>

No forwarding needed since write reg. in 1st half cycle, read reg. in 2nd half cycle.

(From Patterson)
Three Generic Data Hazards (1/3)

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

  \[
  \begin{align*}
  I: & \quad \text{add } r1, r2, r3 \\
  J: & \quad \text{sub } r4, r1, r3
  \end{align*}
  \]

- **Caused by a “(True) Dependence”** (in compiler nomenclature). This hazard results from an actual need for communicating a new data value.

(From Patterson)
Three Generic Data Hazards (2/3)

- Write After Read (WAR)
  Instr\textsubscript{J} writes operand before Instr\textsubscript{I} reads it

  I: sub r4, r1, r3
  J: add r1, r2, r3
  K: mul r6, r1, r7

- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

- Cannot happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Register reads are always in stage 2, and
  - Register writes are always in stage 5

(From Patterson)
Three Generic Data Hazards (3/3)

- **Write After Write (WAW)**
  Instr\(_J\) writes operand before Instr\(_I\) writes it.

  I: sub r1, r4, r3  
  J: add r1, r2, r3  
  K: mul r6, r1, r7

- Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”.

- Cannot happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Register writes are always in stage 5

- Will see WAR and WAW in more complicated pipes

(From Patterson)
Forwarding to Avoid Data Hazard

Figure C.7

Forwarding of ALU outputs needed as ALU inputs 1 & 2 cycles later.

Forwarding of LW MEM outputs to SW MEM or ALU inputs 1 or 2 cycles later.

No forwarding needed since write reg. in 1st half cycle, read reg. in 2nd half cycle.

Instr. | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8
---|---|---|---|---|---|---|---|---
add r1, r2, r3 | Ifetch | Reg | DMem | Reg | | | |
sub r4, r1, r3 | Ifetch | Reg | DMem | Reg | | | |
and r6, r1, r7 | Ifetch | Reg | DMem | Reg | | | |
or r8, r1, r9 | Ifetch | Reg | DMem | Reg | | | |
xor r10, r1, r11 | Ifetch | Reg | DMem | Reg | | | |

(From Patterson)
Figure C.27

HW Datapath Changes (in red) for Forwarding

To forward ALU output 1 cycle to ALU inputs
(From LW Data Memory)

To forward MEM 1 cycle to SW MEM input

(From ALU)

To forward ALU, MEM 2 cycles to ALU

(From Patterson)
Forwarding Avoids ALU-ALU & LW-SW Data Hazards

Figure C.8

**Time (clock cycles)**

- **Cycle 1**: add r1, r2, r3
- **Cycle 2**: lw r4, 0(r1)
- **Cycle 3**: sw r4, 12(r1)
- **Cycle 4**: or r8, r6, r9
- **Cycle 5**: xor r10, r9, r11

(From Patterson)
**LW-ALU Data Hazard Even with Forwarding**

*Figure C.9*

---

**Time (clock cycles)**

- **Cycle 1**: `lw r1, 0(r2)`
- **Cycle 2**: `sub r4, r1, r6`
- **Cycle 3**: `and r6, r1, r7`
- **Cycle 4**: `or r8, r1, r9`

*Figure Notes:*
- No forwarding needed since write reg. in 1st half cycle, read reg. in 2nd half cycle.

*(From Patterson)*
Data Hazard Even with Forwarding
(Similar to Figure C.10)

No forwarding needed since write reg. in 1st half cycle, read reg. in 2nd half cycle.

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

(From Patterson)
Software Scheduling to Avoid Load Hazards

- Try producing fast code with no stalls for

  \[
  a = b + c; \\
  d = e - f;
  \]

  assuming \(a, b, c, d, e,\) and \(f\) are in memory.

  **Slow code:**

  - `LW Rb, b`
  - `LW Rc, c`
  - `ADD Ra, Rb, Rc`
  - `LW Rf, f`
  - `SUB Rd, Re, Rf`

  **Fast code (no stalls):**

  - `LW Rb, b`
  - `LW Rc, c`
  - `ADD Ra, Rb, Rc`
  - `LW Rf, f`
  - `SUB Rd, Re, Rf`
  - `SW d, Rd`

  Compiler optimizes for performance. Hardware checks for safety. Important technique!!

  (From Patterson)