CS425
Computer Systems Architecture

Fall 2018

Pipelining
Previous Lecture

• Measurements and metrics:
  – Performance, Cost, Dependability, Power

• Guidelines and principles in the design of computers
Outline

• Processor review
• Hazards
  – Structural
  – Data
  – Control
• Performance
• Exceptions
Clock Cycle

- Old days: 10 levels of gates
- Today: determined by numerous time-of-flight issues + gate delays
  - clock propagation, wire lengths, drivers
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
“Typical” RISC ISA

• 32-bit fixed format instruction (3 formats)
• 32 32-bit GPR (R0 contains zero)
• 3-address, reg-reg arithmetic instruction
• Single address mode for load/store: base + displacement
  − no indirection
• Simple branch conditions
• Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: 32bit MIPS

Register-Register

```
31  26  25  21  20  16  15  11  10  6  5  0
Op  Rs1  Rs2  Rd     Opx
```

Register-Immediate

```
31  26  25  21  20  16  15  0
Op  Rs1  Rd     immediate
```

Branch

```
31  26  25  21  20  16  15  0
Op  Rs1  Rs2/Opx immediate
```

Jump / Call

```
31  25
Op     target
```

Example:
- lw $2, 100($5)
- add $4, $5, $6
- beq $3, $4, label
Example Execution Steps

- **Instruction Fetch**: Obtain instruction from program storage.
- **Instruction Decode**: Determine required actions and instruction size.
- **Operand Fetch**: Locate and obtain operand data.
- **Execute**: Compute result value or status.
- **Result Store**: Deposit results in storage for later use.
- **Next Instruction**: Determine successor instruction.

5-stage execution is a bit different (see next slides)...
Pipelining: Latency vs Throughput

Pipelining doesn’t help **latency** of single task, it helps **throughput** of entire workload.
5-stage Instruction Execution - Datapath

IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR<sub>rs</sub>];
B <= Reg[IR<sub>rt</sub>]
rslt <= A op<sub>IRop</sub> B
WB <= rslt
Reg[IR<sub>rd</sub>] <= WB
Visualizing Pipelining
5-stage Instruction Execution - Control

Pipeline Registers: IR, A, B, r, WB
Limits in Pipelining

• Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  
  - **Structural hazards:** Resource conflicts, 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).

**In order:** when an instruction is stalled, all instructions issued *later* than the stalled instruction are also stalled.
Example of Structural Hazard

<table>
<thead>
<tr>
<th>Time (clock cycles)</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>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>Instr 1</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>Instr 2</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>Instr 3</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>Instr 4</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
<td>Reg</td>
</tr>
</tbody>
</table>

CS425 - Vassilis Papaefstathiou
Example of Structural Hazard

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>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
<td>Reg</td>
</tr>
<tr>
<td>Instr 1</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
<tr>
<td>Instr 2</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Reg</td>
<td>Ifetch</td>
</tr>
</tbody>
</table>

Stall

Instr 3

How do you “bubble” this pipe (if instr1 = load)?
Example of Structural Hazard

How do you “bubble” this pipe (if instr1 = load)?

CS425 - Vassilis Papaefstathiou
Speed Up Equation of Pipelining

\[
\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\
= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
\]

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\
= 1 + \text{Pipeline stall clock cycles per instruction}
\]

For simple RISC pipeline, Ideal CPI = 1:

\[
\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\
= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}
\]
Example: Dual-port vs Single-port

- Machine A: Dual read ported memory ("Harvard Architecture")
- Machine B: Single read ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Suppose that Loads/Stores are 40% of instructions executed

\[
\text{Average instruction time}_B = \text{CPI}_B \times \text{Clock cycle time}_B = (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_A}{1.05} = 1.3 \times \text{Clock cycle time}_A
\]

- Machine A is 1.33 times faster (CPUtime = IC x Aver instr time)

Why would a designer allow structural hazards?
Data Hazard

Time (clock cycles)

Instr. Order

add \( r1, r2, r3 \)

sub \( r4, r1, r3 \)

and \( r6, r1, r7 \)

or \( r8, r1, r9 \)

xor \( r10, r1, r11 \)
Read After Write

- **Read After Write (RAW)**
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

  \[
  \text{I: add } r1, r2, r3
  \]
  \[
  \text{J: sub } r4, r1, r3
  \]

- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.
Write After Read

- **Write After Read (WAR)**
  Instr\(_J\) writes operand *before* Instr\(_I\) reads it

\[
\begin{align*}
\text{I:} & \quad \text{sub} \ r4, r1, r3 \\
\text{J:} & \quad \text{add} \ r1, r2, r3 \\
\text{K:} & \quad \text{mul} \ r6, r1, r7
\end{align*}
\]

- 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 in order stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5
Write After Write

- **Write After Write (WAW)**
  \( \text{Instr}_j \) writes operand **before** \( \text{Instr}_i \) 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 “\( r1 \)”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 in order stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in more complicated pipelines
Forwarding to avoid data hazards

Time (clock cycles)

Instr. Order

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

No Stall!

Ignore what you read from Register File
What circuit detects and resolves this hazard?
Why we need forwarding lines for both inputs of the ALU?
Forwarding to Avoid LW-SW Data Hazard

### Instruction Order

- **add** \( r1, r2, r3 \)
- **lw** \( r4, 0(r1) \)
- **sw** \( r4, 12(r1) \)
- **or** \( r8, r6, r9 \)
- **xor** \( r10, r9, r11 \)
Data Hazard Even with Forwarding

\[ \text{lw } r1, 0(r2) \]

\[ \text{sub } r4, r1, r6 \]

\[ \text{and } r6, r1, r7 \]

\[ \text{or } r8, r1, r9 \]
Data Hazard Even with Forwarding

1. **lw** `r1`, 0(`r2`)
2. **sub** `r4`, `r1`, `r6`
3. **and** `r6`, `r1`, `r7`
4. **or** `r8`, `r1`, `r9`
Try producing fast code for
\[
a = b + c; \\
d = e - f;
\]
assuming \( a, b, c, d, e, \) and \( f \) in memory.

**Slow code:**

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

**Fast code:**

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
LW    Re,e
LW    Rf,f
SW    a,Ra
```

---

**Software Scheduling to Avoid Load Hazards**
Control Hazard on Branches: Three Stage Stall

10: beq r1,r3,26
14: and r2,r3,r5
18: or r6,r1,r7
22: add r8,r1,r9
26: 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
Pipelined MIPS Datapath

- Interplay of instruction set design and cycle time.

CS425 - Vassilis Papaefstathiou 31
Control Hazard on Branches: One 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
Four Branch Hazard Alternatives

• #1: Stall until branch direction is clear (simplicity)
• #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
    o MIPS still incurs 1 cycle branch penalty
    o Other machines: branch target known before outcome
  - What happens on not-taken branches?
# Four Branch Hazard Alternatives

## #4: Delayed Branch

- Define branch to take place **AFTER** a following instruction branch instruction
  - sequential successor$_1$
  - sequential successor$_2$
  - ........
  - sequential successor$_n$

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

A. From before branch

\[ \text{add } \$1,\$2,\$3 \]

if \( \$2=0 \) then

\[ \text{delay slot} \]

becomes

if \( \$2=0 \) then

\[ \text{add } \$1,\$2,\$3 \]

B. From branch target

\[ \text{sub } \$4,\$5,\$6 \]

\[ \text{add } \$1,\$2,\$3 \]

if \( \$1=0 \) then

\[ \text{delay slot} \]

becomes

\[ \text{add } \$1,\$2,\$3 \]

if \( \$1=0 \) then

\[ \text{sub } \$4,\$5,\$6 \]

\[ \text{sub } \$4,\$5,\$6 \]

\[ \text{sub } \$4,\$5,\$6 \]

C. From fall through

\[ \text{add } \$1,\$2,\$3 \]

if \( \$1=0 \) then

\[ \text{delay slot} \]

\[ \text{OR } \$7,\$8,\$9 \]

\[ \text{sub } \$4,\$5,\$6 \]

becomes

\[ \text{add } \$1,\$2,\$3 \]

if \( \$1=0 \) then

\[ \text{OR } \$7,\$8,\$9 \]

\[ \text{sub } \$4,\$5,\$6 \]

\[ \text{sub } \$4,\$5,\$6 \]

• 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/C, must be okay to execute sub/\text{OR} when branch is untaken/taken
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
Example: Evaluating Branch Alternatives

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

Deep pipeline in this example:
- 2 cycles for address (2 stalls)
- 1 more cycle to evaluate condition

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Penalty unconditional</th>
<th>Penalty untaken</th>
<th>Penalty taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush pipeline</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Unconditional branches</th>
<th>Untaken conditional branches</th>
<th>Taken conditional branches</th>
<th>All branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency of event</td>
<td>4%</td>
<td>6%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>Flush pipeline</td>
<td>0.08</td>
<td>0.18</td>
<td>0.30</td>
<td>0.56</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>0.08</td>
<td>0.18</td>
<td>0.20</td>
<td>0.46</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>0.08</td>
<td>0.00</td>
<td>0.30</td>
<td>0.38</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 (precise interrupt?): It must appear that the exception or interrupt happens between 2 instructions \((i \text{ and } i+1)\)
  - The effect of all instructions up to and including \(i\) is totaling complete
  - No effect of any instruction after \(i\) can take place

• The interrupt (exception) handler either aborts program or restarts at instruction \(i+1\)
Precise Exceptions in Static Pipelines

- **Key observation**: architectural state changes **only** in memory and register write stages.
Summary: Pipelining

• Next time: Read Appendix A
• Control via State Machines and Microprogramming
• 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