ENGN1640: Design of Computing Systems
Topic 05: Pipeline Processor Design

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2014
Pipelining analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

Four loads:
- Speedup = \( \frac{16}{7} = 2.3 \)

Non-stop:
- Ideal Speedup = \( \frac{4n}{n + 3} \approx 4 \)
  = number of stages
MIPS stages

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register
MIPS datapath pipeline stages

- Need registers between stages
- To hold information produced in previous cycle
Pipeline datapath abstraction

- Form showing resource usage

```
Program execution order
(in instructions)

lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```
Multi-cycle datapath pipeline diagram

- Traditional form

Program execution order (in instructions)

- lw $10, 20($1)
- sub $11, $2, $3
- add $12, $3, $4
- lw $13, 24($1)
- add $14, $5, $6

Diagram showing the execution order and time in clock cycles (CC 1 to CC 9) for each instruction.
Tracing \texttt{lw} in its journey: 1\textsuperscript{st} cycle
Tracing lw in its journey: 2\textsuperscript{nd} cycle
Tracing lw in its journey: 3rd cycle
Tracing lw in its journey: 4th cycle
Tracing $lw$ in its journey: $5^{th}$ cycle
Corrected pipeline datapath for lw
Pipeline state in 5th cycle

lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
Pipeline performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
Single-cycle versus pipeline performance

**Single-cycle (T_c = 800ps)**

- Program execution order (in instructions)
- Instruction fetch, Reg, ALU, Data access, Reg
- Time: 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800
- lw $1, 100($0)
- lw $2, 200($0)
- lw $3, 300($0)
- Time: 800 ps

**Pipelined (T_c = 200ps)**

- Program execution order (in instructions)
- Instruction fetch, Reg, ALU, Data access, Reg
- Time: 200, 400, 600, 800, 1000, 1200, 1400
- lw $1, 100($0)
- lw $2, 200($0)
- lw $3, 300($0)
- Time: 200 ps
Pipeline speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}

- Ideal speedup (n instructions and s stages)
  \[ \frac{sn}{n + s - 1} \]

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
  - Branches will also reduce the speedup
  - Added pipeline registers reduce the speedup
Pipeline datapath summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $14, $5, $6</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>lw $13, 24 ($1)</td>
<td>Instruction decode</td>
</tr>
<tr>
<td>add $12, $3, $4</td>
<td>Execution</td>
</tr>
<tr>
<td>sub $11, $2, $3</td>
<td>Memory</td>
</tr>
<tr>
<td>lw $10, 20($1)</td>
<td>Write-back</td>
</tr>
</tbody>
</table>

- How many cycles it takes to finish code?
Reminder of single-cycle control
Control signals
Modifications to pipeline control

- Control signals derived from instruction
  - Same as in single-cycle implementation
  - Control delayed to proper pipeline stage
Pipelined datapath + control
Example: Cycle 1

\text{lw} \quad 10, 20(1)
\text{sub} \quad 11, 2, 3
\text{and} \quad 12, 4, 5
\text{or} \quad 13, 6, 7
\text{add} \quad 14, 8, 9
Cycle 2

lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Cycle 3

```
lw     $10, 20($1)
sub    $11, $2, $3
and    $12, $4, $5
or     $13, $6, $7
add    $14, $8, $9
```
Cycle 4

lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Cycle 5

```
lw    $10, 20($1)
sub   $11, $2, $3
and   $12, $4, $5
or    $13, $6, $7
add   $14, $8, $9
```
Cycle 6

```
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or   $13, $6, $7
add  $14, $8, $9
```
Cycle 7

lw    $10, 20($1)
sub   $11, $2, $3
and   $12, $4, $5
or    $13, $6, $7
add   $14, $8, $9
lw    $10, 20($1)
sub   $11, $2, $3
and   $12, $4, $5
or    $13, $6, $7
add   $14, $8, $9
Cycle 9

```!nasm
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or   $13, $6, $7
add  $14, $8, $9
```
Pipelining hazards

1. Situations that prevent starting the next instruction in the next cycle
   - Structural hazards
     - A required resource is busy
   - Data hazard
     - Need to wait for previous instruction to complete its data read/write
   - Control hazard
     - Deciding on control action depends on previous instruction
1. Structural hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
2. Data Hazards: compute-use

add $s0, $s2, $s3

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5
Data Hazard: load-use

Program execution order (in instructions):

lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
Handling data hazards

A. Compile-time techniques
B. Forward data at run time
C. Stall the processor at run time
A. Data hazard elimination using compile-time techniques (code rescheduling)

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = D + F;
- Compiler must be aware of pipeline structure

C code:
```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
lw $t5, 16($t0)
add $t5, $t4, $t5
sw $t5, 20($t0)
```

HAZARD in SW

Reordered code:
```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
lw $t5, 12($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t4, $t5
sw $t5, 20($t0)
```
A. Data hazard elimination using compile-time techniques (nop)

- Insert enough nops until result is ready (wastes cycles)

```assembly
add $s0, $s2, $s3

nop

nop

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5
```
B. Data hazard elimination using data forwarding/bypassing during runtime

- Don’t wait for result to be stored in a register → forward the results whenever the results
- Requires extra connections in the datapath
Dependencies and forwarding

Program execution order (in instructions)

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Circuitry for forwarding

b. With forwarding
Close up: one more MUX for immediates
Forward unit design: When to Forward?

- **Three conditions must be met:**
  1. EX/MEM.RegWrite and/or MEM/WB.RegWrite are true
  2. Destination register(s) are equal to the source registers of the next 1 - 2 instructions. That is,
     - EX/MEM.RegisterRd == ID/EX.RegisterRs
     - EX/MEM.RegisterRd == ID/EX.RegisterRt
     - MEM/WB.RegisterRd == ID/EX.RegisterRs
     - MEM/WB.RegisterRd == ID/EX.RegisterRt
  3. Destination register in EX/MEM and/or MEM/WB is not $0$. That is,
     - EX/MEM.RegisterRd ≠ 0
     - MEM/WB.RegisterRd ≠ 0
Double data hazard

- Consider the sequence:
  - add $1, \$1, \$2$
  - add $\$1, \$1, \$3$
  - add $\$1, \$1, \$4$

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Give priority to EX results. That is, only fwd from MEM if EX hazard condition isn’t true
Forwarding illustration

\[
\begin{align*}
\text{sub} & \quad \$2, \$1, \$3 \\
\text{and} & \quad \$4, \$2, \$5 \\
\text{or} & \quad \$4, \$4, \$2 \\
\text{add} & \quad \$9, \$4, \$2
\end{align*}
\]
3\textsuperscript{rd} cycle

or $4, 4, 2$
and $4, 2, 5$

sub $2, 1, 3$
and $4, 2, 5$

or $4, 4, 2$
add $9, 4, 2$
4th cycle

add $9, $4, $2

and $4, $4, $2

sub $4, $2, $5

sub $2, ...

before<1>

sub $2, $1, $3

and $4, $2, $5

or $4, $4, $2

add $9, $4, $2
5th cycle

```
sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2
```

after<1>

add $9, $4, $2

or $4, $4, $2

and $4, ...

sub $2, ...

Clock 5
6th cycle

after<2>  |  after<1>  |  add $9, $4, $2  |  or $4,...  |  and $4,...

Clock 6
Pipelining hazards

- Structural hazards
  
2. Data hazard

3. Control hazard

- Compile-time techniques
- Forward data at run time
C. Stall the processor at run time
Data hazards from lw

Time (in clock cycles) | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9
---|---|---|---|---|---|---|---|---|---

Program execution order (in instructions)

lw $2, 20($1)

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

slt $1, $6, $7
Forwarding is not going to eliminate all hazards
C. Data hazard elimination by stalling

Program execution order (in instructions)

- lw $2, 20($1)
- and becomes nop
- and $4, $2, $5
- or $8, $2, $6
- add $9, $4, $2

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9  CC 10

Clock cycle wasted — necessary for correctness

Stall inserted here
Another example

lw $s0, 40($0)

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5

lw $s0, 40($0)

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5

Stall

Trouble!
When to stall? How to stall?

**When?**

- ID/EX.MemRead is true
- ID/EX.RegisterRt = IF/ID.RegisterRs
  or ID/EX.RegisterRt = IF/ID.RegisterRt

**How to insert a bubble?**

- Do not update PC or IF/ID (instruction in ID stage is decoded again, instruction in IF stage is fetched again)
- Force control values in ID/EX register to 0 ➞ 1-cycle stall allows MEM to read data for `lw` before being used
Pipeline with bubble insertion for hazards

lw  $2, 20($1) and $4, $2,$5
or  $4, $4,$2
add $9, $4,$2
Cycle 2

lw $2, 20($1)
and $4, $2,$5
or $4, $4,$2
add $9, $4,$2

and $4,$2,$5

Clock 2
Cycle 3

lw $2, 20($1)
and $4, $2,$5
or $4, $4,$2
add $9, $4,$2

or $4,$4,$2
and $4,$2,$5

before<1>
before<2>
Cycle 4

lw $2, 20($1)
and $4, $2,$5
or $4, $4,$2
add $9, $4,$2
Cycle 5

```plaintext
add  $9, $4,$2
or  $4, $4,$2
and $4, $2,$5

Bubble
lw  $2, 20($1)
and $4, $2,$5
or  $4, $4,$2
add $9, $4,$2
```

Clock 5
Cycle 6

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2
Cycle 7

lw $2, 20($1)
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

Clock 7

S. Reda EN164 Sp '11
Data hazard summary

- Compiler can arrange code to avoid hazards and stalls. Requires knowledge of the pipeline structure.

- Forwarding can sometimes avoid stalls at the expense of extra hardware complexity.

- Stalls reduce performance by increasing the average cycles per instruction (CPI). But sometimes are absolutely necessary to get correct results.
If branch evaluation does not occur until the MEM pipeline stage then 3 cycles could be wasted. Either *stall be default* or *predict branch target and flush if necessary*.
Reduced branch penalty

- If we move hardware (target address adder and register comparator) that determine branch outcome to ID stage (instead of MEM) then only 1 instruction needs to be flushed or stalled.
Pipeline modification for earlier branch evaluation

- MIPS architecture (beq and bne) – and RISC architectures in general were designed for fast single-cycle branches with a small branch penalty. Hypothetical instructions like bgeq or bseq require bulkier comparators with greater propagation delay.
Branch prediction

- Deeper pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable

- Two possible solutions:
  1. Static branch prediction (taken or not taken)
  2. Dynamic branch prediction
1. A Static branch prediction (not taken case)

Prediction correct

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $4, $5, $6</td>
<td>200</td>
</tr>
<tr>
<td>beq $1, $2, 40</td>
<td>400</td>
</tr>
<tr>
<td>lw  $3, 300($0)</td>
<td>600</td>
</tr>
</tbody>
</table>

Prediction incorrect

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $4, $5, $6</td>
<td>200</td>
</tr>
<tr>
<td>beq $1, $2, 40</td>
<td>400</td>
</tr>
<tr>
<td>lw  $3, 300($0)</td>
<td>600</td>
</tr>
</tbody>
</table>

Convert the incorrectly fetched instruction into a bubble (flush it)
1.B Static branch prediction (taken case)

- Assume branch will be taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
      ...
72:  lw   $4, 50($7)
```
Branch taken: cycle 3

and $12, $2, $5

beq $1, $3, 7

sub $10, $4, $8

before<1>

before<2>

Clock 3
Branch taken: cycle 4

There is still 1 cycle penalty to compute target address
Eliminating 1-cycle stall for taken-prediction policy with branch target buffer

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch

**Branch target buffer**

- Cache of target addresses
- Indexed by PC when instruction fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately – no 1-cycle penalty
2. Dynamic branch prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction:
  - **Branch prediction buffer** (aka branch history table) indexed by recent branch instruction addresses and stores outcome (taken/not taken)
  - To execute a branch:
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction
1-bit predictor

- Inner loop branches mispredicted twice!

```
outer: ...
...
inner: ...
...
beq ..., ..., inner
...
beq ..., ..., outer
```

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around
2-bit predictor

- Only change prediction on two successive mispredictions
Data hazards for branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
... beq $1, $4, target
```

- Hazard can resolved using forwarding and stalling
Data hazards for branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need to modify forwarding hardware to forward from EX/MEM or MEM/WB pipeline registers to the comparator
  - Need 1 stall cycle in this example

```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```
Data hazards for branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles
  - Need to modify forwarding hardware to forward from MEM/WB

```
lw $1, addr
beq stalled
beq stalled
beq $1, $0, target
```
Summary

• Pipelining for speedup

• Ideal speedup = number of stages, but actual speedup depends on delay balance between stages (clock frequency), delays introduced by pipeline registers, and number of stalls (CPI).

• Hazards (structural, data, and control) can increase CPI

• Hazards can be eliminated or mitigated using code reorganization, stalling, flushing, forwarding / bypassing

• Branch prediction, branch prediction buffer, branch target buffer can reduce stalls arising from control hazards