Cycle Time for Non-pipelined & Pipelined processors

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>250ps</td>
<td>350ps</td>
<td>150ps</td>
<td>300ps</td>
<td>200ps</td>
</tr>
</tbody>
</table>

- For a non-pipelined processor, the clock cycle is the sum of the latencies of all the pipeline elements:
  Clock cycle = $250 + 350 + 150 + 300 + 200 = 1250$ ps

- For a pipelined processor, the clock cycle is the time of the pipeline element with the largest latency:
  Clock cycle = 350 ps.
Control Hazard

- conditional branch determines control flow
  - fetching next instruction depends on outcome
- in MIPS pipeline
  - compare registers and compute result early
  - add hardware to ID/Reg stage
  - Is this enough?
Stall on Branch

- shorter wait for branch outcome

Program execution order (in instructions)

- `add $4, $5, $6`
- `beq $1, $2, 40`
- `or $7, $8, $9`

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction fetch</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 ps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction fetch</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>400 ps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 200 ps stall
- 400 ps stall
Branch Prediction

- longer pipelines can't determine outcome early
  - stall penalty becomes more significant
- predict outcome of branch
  - only stall if prediction is wrong
- in MIPS
  - simple static prediction
  - branch not taken, fetch next instruction – no delay
  - stall only when branch is taken
Predict Not Taken

Program execution order (in instructions)

Prediction correct

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

Prediction incorrect

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

or $7, $8, $9

Instruction fetch  Reg  ALU  Data access  Reg

Instruction fetch  Reg  ALU  Data access  Reg

Instruction fetch  Reg  ALU  Data access  Reg

Instruction fetch  Reg  ALU  Data access  Reg

Instruction fetch  Reg  ALU  Data access  Reg

Instruction fetch  Reg  ALU  Data access  Reg
More Realistic Branch Prediction

- static branch prediction
  - based on typical branch behaviour
    - loop: predict backward branches taken
    - if: predict forward branches not taken
- encode in high-level SW
- dynamic branch prediction
  - hardware measures actual branch behaviour
    - e.g., record recent history of each branch
  - assume future behaviour will continue trend
Writing Code to Avoid Hazards

\[
\text{sub } \$2, \ $4, \ $5 \\
\text{slt } \$12, \ $2, \ $5 \\
\text{slt } \$18, \ $6, \ $2 \\
\text{add } \$13, \ $7, \ $4 \\
\]

Consider the pipeline diagram with NO forwarding.

\[
\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{sub } \$2, \ $4, \ $5 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{slt } \$12, \ $2, \ $5 & \text{IF} & \text{--} & \text{---} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{slt } \$18, \ $6, \ $2 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{add } \$13, \ $7, \ $4 & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\end{array}
\]
Consider the pipeline diagram with NO forwarding.

```
    1  2  3  4  5  6  7  8  9  10
sub $2,  $4, $5      IF  ID  EX  MEM  WB
slt $12, $2, $5      IF  --  ---  ID  EX  MEM  WB
slt $18, $6, $2      IF  ID  EX  MEM  WB
add $13, $7, $4      IF  ID  EX  MEM  WB
```

With forwarding, how many cycles do we save?

```
    1  2  3  4  5  6  7  8
sub $2,  $4, $5      IF  ID  EX  MEM  WB
slt $12, $2, $5      IF  ID  EX  MEM  WB
slt $18, $6, $2      IF  ID  EX  MEM  WB
add $13, $7, $4      IF  ID  EX  MEM  WB
```
What if we can't forward? Improve the speed from A by rearranging lines of code WITHOUT changing the final result of the program.

```
sub $2, $4, $5
slt $12, $2, $5
slt $18, $6, $2
add $13, $7, $4
```

This add instruction not affected by the slt

We save 1 stall by moving the add instruction.
Pipeline Summary

- pipelining improves performance by increasing instruction throughput
  - multiple instructions executed in parallel
  - unchanged latency
- subject to hazards
  - structure, data, control
- instruction set affects pipeline complexity
Memory Characteristics

- cost per unit storage
- performance
  - access latency - How long does it take to make one access?
  - Throughput - How many accesses can be done over a unit of time?
- Persistency – volatile/non-volatile
- Being non-volatile means that memory holds its data throughout a power cycle. e.g. if you turn off your computer, then turn it back on the memory holds the same value it did before.
Registers

- very expensive, thus limited
- access at instruction speed
- volatile
Main Memory

- cheap & large
- noticeable access latency
  - approx. factor 100 slower
- volatile
Random Access Memory (RAM)

- array
- index-based direct access
  - via memory bus
- fetching memory
  - CPU sends address to memory controller
  - controller responds with data
Static vs. Dynamic RAM

- **Static RAM (SRAM)**
  - similar to previously shown memory circuit
  - stable storage, as long as power is applied
  - multiple transistors per bit
  - expensive, but faster
  - used for caching

- **Dynamic RAM (DRAM)**
  - bit stored in capacitor, needs refreshing
  - single transistor per bit
  - cheaper, but slower
Non-volatile memory

• Magnetic Disks
  • Very cheap, slow, mechanical
  • Used in desktops, servers, laptops (to a certain extent)

• Flash Memory
  • More expensive, fast, non-mechanical
  • Limited Writes
  • Used in usb drives, smartphones
Memory Technology

- typical performance and cost figures
  - as of 2008

<table>
<thead>
<tr>
<th>Technology</th>
<th>Access Time</th>
<th>$/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>0.5-2.5ns</td>
<td>$2000-$5000</td>
</tr>
<tr>
<td>DRAM</td>
<td>50-70ns</td>
<td>$20-$75</td>
</tr>
<tr>
<td>Disk</td>
<td>5,000,000-7,000,000ns</td>
<td>$0.20-$2</td>
</tr>
</tbody>
</table>

- ideally: have large amounts of fast memory
Memory Hierarchy

CPU

Level 1

Level 2

... 

Level $n$

Levels in the memory hierarchy

Increasing distance from the CPU in access time

Size of the memory at each level
Memory Hierarchy

- going down the hierarchy, memory gets
  - cheaper
  - bigger
  - slower
  - further away

- might consider lowest level as backing storage
  - everything else as a cache...
  - different perspectives exist
Memory Stalls

- delay until response from memory controller
- what does the CPU do in the meantime?
  - compute something else? details later...
- memory compromise: fast vs. cheap
  - satisfy as many requests as fast as possible
Locality Principle

- temporal locality
  - same data item likely to be used again soon
  - e.g., loop

- spatial locality
  - close data item likely to be used soon
  - e.g., iteration through array

- example: books in library and on desk