Pipelining

Making the CPU faster.

- Form of parallel processing.
- New instructions are started before previous ones finish.
- At any given time, several instructions are active at various stages of completion.

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Efficiency

- Does not decrease the execution time of any individual instruction.
- Increases the throughput of instructions.
- Efficient for long streams of instructions: The pipeline must be full for maximum benefit.

Pipeline Stages

Break each instruction up into five stages.
- IF: Instruction Fetch
- ID: Instruction Decode and register read.
- EX: EXecution or address calculation.
- MEM: data MEMory access.
- WB: Write Back to register.
### The Pipeline: An Instruction Assembly Line 3

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$10 = 47$</td>
<td>$16 = 41$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$11 = 16$</td>
<td>$17 = -111$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$12 = 29$</td>
<td>$18 = 45$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$13 = 0x412918$</td>
<td>$19 = 7321$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$14 = 32$</td>
<td>$20 = 499$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$15 = 5$</td>
<td>$21 = 10$</td>
<td></td>
</tr>
</tbody>
</table>

| Instruction Memory |        | $13 = 0x412918$ | $16 = 41$ |            |
| $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |

| Register File |        |        |        |            |
| $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |

| Data Memory |        | $0x412918 = 4981$ | $0x41292c = 79$ |            |
| $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |
|        | $10 = 47$ | $11 = 16$ | $12 = 29$ | $13 = 0x412918$ | $14 = 32$ | $15 = 5$ | $16 = 10$ |

### The Pipeline: An Instruction Assembly Line 4

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$18 = 0x412928$</td>
<td>$16 = 41$</td>
<td>$17 = -111$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
<tr>
<td>$21 = 10$</td>
<td>$0x412918$</td>
<td>$0x41292c$</td>
<td>$79$</td>
<td></td>
</tr>
</tbody>
</table>

### The Pipeline: An Instruction Assembly Line 5

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$18 = 0x412928$</td>
<td>$16 = 41$</td>
<td>$17 = -111$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
<tr>
<td>$21 = 10$</td>
<td>$0x412918$</td>
<td>$0x41292c$</td>
<td>$79$</td>
<td></td>
</tr>
</tbody>
</table>

### The Pipeline: An Instruction Assembly Line 6

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$18 = 0x412928$</td>
<td>$16 = 41$</td>
<td>$17 = -111$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
<tr>
<td>$21 = 10$</td>
<td>$0x412918$</td>
<td>$0x41292c$</td>
<td>$79$</td>
<td></td>
</tr>
</tbody>
</table>
### The Pipeline: An Instruction Assembly Line 7

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$16 = 4981$</td>
<td>$17 = 111$</td>
<td>$18 = 0x412928$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
</tbody>
</table>

Register File:
- $18 = 0x412928$
- $19 = 0x412918$
- $16 = 4981$

Data Memory:
- $0x412918 = 4981$
- $0x41292c = 79$

### The Pipeline: An Instruction Assembly Line 8

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$16 = 4981$</td>
<td>$17 = 111$</td>
<td>$18 = 0x412928$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
</tbody>
</table>

Register File:
- $18 = 0x412928$
- $19 = 0x412918$
- $16 = 4981$

Data Memory:
- $0x412918 = 4981$
- $0x41292c = 79$

### The Pipeline: An Instruction Assembly Line 9

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$16 = 4981$</td>
<td>$17 = 111$</td>
<td>$18 = 0x412928$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
</tbody>
</table>

Register File:
- $18 = 0x412928$
- $19 = 0x412918$
- $16 = 4981$

Data Memory:
- $0x412918 = 4981$
- $0x41292c = 79$

### The Pipeline: An Instruction Assembly Line 10

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = 47$</td>
<td>$11 = 16$</td>
<td>$12 = 29$</td>
<td>$13 = 0x412918$</td>
<td>$14 = 32$</td>
</tr>
<tr>
<td>$16 = 4981$</td>
<td>$17 = 111$</td>
<td>$18 = 0x412928$</td>
<td>$19 = 7321$</td>
<td>$20 = 499$</td>
</tr>
</tbody>
</table>

Register File:
- $18 = 0x412928$
- $19 = 0x412918$
- $16 = 4981$

Data Memory:
- $0x412918 = 4981$
- $0x41292c = 79$
The Pipeline: An Instruction Assembly Line 11

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $10, $11, $13</td>
<td>$10 = 47</td>
<td>$16 = 4981</td>
<td>$16 = 0x413a5d</td>
<td></td>
</tr>
<tr>
<td>be $16, 0x3232</td>
<td>$11 = 79</td>
<td>$17 = -111</td>
<td>$17 = 0x412945</td>
<td></td>
</tr>
<tr>
<td>sub $21, $10, $11</td>
<td>$12 = 29</td>
<td>$18 = 0x412928</td>
<td>$18 = 0x412945</td>
<td></td>
</tr>
<tr>
<td>add $20, $18, $12</td>
<td>$13 = 0x412918</td>
<td>$19 = 0x412938</td>
<td>$19 = 0x412938</td>
<td></td>
</tr>
<tr>
<td>be $11, 4($18)</td>
<td>$14 = 32</td>
<td>$20 = 0x412945</td>
<td>$20 = 0x412945</td>
<td></td>
</tr>
<tr>
<td>sub $10, $18, $16</td>
<td>$15 = -5</td>
<td>$21 = 11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register File

Instruction Memory

Data Memory

M[0x412918] = 4981
M[0x41292c] = 79

Pipeline Registers

- Each is quite large.
- Holds the results of each stage ready for the next.
- Serves the function of the conveyor belt.
Pipeline Flow 5

sub $11, $2, $3; lw $10, 20($1)

Pipeline Control: High Level

Instructions take their control signals with them.

Pipeline Flow 6

sub $11, $2, $3; lw $10, 20($1)

Pipeline Control: Detailed
Pipeline Control Example: 1

IF: lw $10, 20($1)
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>

Pipeline Control Example: 2

IF: sub $11, $2, $3
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>

Pipeline Control Example: 3

IF: and $12, $4, $5
ID: before<1>
EX: lw $10, . . .
MEM: before<2>
WB: before<3>

Pipeline Control Example: 4

IF: or $13, $6, $7
ID: and $12, $2, $3
EX: lw $10, . . .
MEM: before<1>
WB: before<2>
**Pipeline Control Example: 9**

![Pipeline Control Diagram]

**Data Hazards**

An instruction which uses a value may fetch it before the instruction which computes it has stored the value.

\[
\begin{align*}
\text{add} & \; \$0, \; \$t0, \; \$t1 \\
\text{sub} & \; \$t2, \; \$s0, \; \$t3
\end{align*}
\]

**Pipelining**

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores

- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

- We’ll build a simple pipeline and look at these issues

- We’ll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

**Data Hazards in the Pipeline 1**

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $s10, $t10</td>
<td>add $s11, $s11</td>
<td>$s10 = 0xa401208</td>
<td>$s10 = $0x44812</td>
<td></td>
</tr>
<tr>
<td>add $s11, $s11</td>
<td>$s11 = 781</td>
<td>$s16 = 459</td>
<td>$s15 = 783</td>
<td></td>
</tr>
<tr>
<td>$s12 = 4981</td>
<td>$s12 = 4981</td>
<td>$s17 = 70</td>
<td>$s16 = 0</td>
<td></td>
</tr>
<tr>
<td>$s13 = 2</td>
<td>$s13 = 2</td>
<td>$s18 = 4981</td>
<td>$s17 = 0</td>
<td></td>
</tr>
<tr>
<td>$s14 = 1000</td>
<td>$s14 = 1000</td>
<td>$s19 = 0</td>
<td>$s18 = 7</td>
<td></td>
</tr>
<tr>
<td>$s15 = 783</td>
<td>$s15 = 783</td>
<td>$s20 = 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Data Hazards Diagram]
Dealing With Data Hazards

- Stall: Wait for the data.
- Forward: Pass new data backwards along the pipeline.
- Delayed Loads: Say that when a register is set, its value is undefined for some number of steps.

Let the assembler or compiler worry about it.

MIPS Example Solution

- Values computed by the ALU are forwarded.
- Stall one cycle for values fetched from memory.

The computed values are available in the pipeline. The memory contents are not.
Dealing With Control Hazards

- **Stall:** Don’t start a new instruction until everything is known.
  
  *May have to discard instructions fetched after a branch.*

- **Predict:** Guess a result and cancel the calculation if it turns out wrong.

- **Delayed Branch:** Just say that branch is not effective immediately.

  *Instruction(s) after branch executed unconditionally.*

  *Let the assembler or compiler worry about it.*

Control Hazards

Instructions are fetched in address order.

After an unconditional jump, the following instruction is fetched before the jump is decoded and the target computed.

For a conditional branch, we also don’t know yet if we’re actually going to branch.

Delayed Branch

- Can be used with the others to reduce cost of stall or incorrect prediction.

- Usually just one instruction is executed unconditionally after the branch.

  *This is called the delay slot.*

- Compiler optimizer tries to put something useful there.
Filling the Delay Slot

a. From before
   add $t1, $t2, $s3
   if $t2 = 0 then
   \[\text{Delay slot}\]

b. From target
   sub $t4, $t5, $t6
   \[\text{Delay slot}\]
   if $t1 = 0 then
   \[\text{Delay slot}\]
   add $t1, $t2, $s3
   if $t1 = 0 then
   \[\text{Delay slot}\]

c. From full through
   add $t1, $t2, $s3
   if $t1 = 0 then
   \[\text{Delay slot}\]
   sub $t4, $t5, $t6

MIPS Example Solution

- Move target computation to decode station.
- Add a comparator at decode to decide if branches are taken.
- One cycle is lost if the branch is taken.

*Alternative: Define a delay slot. Then no cycle is lost if the delay slot can do useful work.*

Branch Prediction

- Guess which way the branch will go and act accordingly.
- If wrong, discard partial results.
Static Branch Prediction

- Predict all branches the same way all the time.
- Usually, predict not taken.

Dynamic Branch Prediction

- Guess based on past behavior.
- Must record the history of each branch.
- Use a small memory as a hash.
- The hash address is the low part of the branch instruction's memory address.
- Ignore collisions. They reduce performance, not correctness.

Dynamic Branch Prediction Types

- One-bit: Predict it will branch the same as last time. *Wrong twice for a simple loop.*
- Two-bit: Retain prediction until wrong twice. *Wrong once for a simple loop.*

Two-Bit Dynamic Branch Prediction
**Exceptions**

- Must jump to O/S.
- Can occur at any stage.
- Must re-construct a state after the faulting instruction.

**Faster Still: Superscalar**

- Multiple Pipelines
- More than one instruction per cycle.
- All kinds of terrible ordering problems.

**The Whole Thing**

**Faster Still: Dynamic Pipeline Scheduling**

- Dispatch instructions out of order.