Pipelined CPU

Jason Mars
Evolution of Our CPU: Single Cycle
Evolution of Our CPU: Multi Cycle
Instruction Latencies and Throughput
Instruction Latencies and Throughput

• Single-Cycle CPU

| Load | Ifetch | Reg/Dec | Exec | Mem | Wr |
Instruction Latencies and Throughput

• Single-Cycle CPU

Load | Ifetch | Reg/Dec | Exec | Mem | Wr

• Multiple Cycle CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

Load | Ifetch | Reg/Dec | Exec | Mem | Wr
Instruction Latencies and Throughput

• Single-Cycle CPU

Load: Ifetch Reg/Dec Exec Mem Wr

• Multiple Cycle CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

Load: Ifetch Reg/Dec Exec Mem Wr

• Pipelined CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8
Instruction Latencies and Throughput

• Single-Cycle CPU

Load | Ifetch | Reg/Dec | Exec | Mem | Wr

• Multiple Cycle CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5
Load | Ifetch | Reg/Dec | Exec | Mem | Wr

• Pipelined CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8
Load | Ifetch | Reg/Dec | Exec | Mem | Wr
Instruction Latencies and Throughput

- **Single-Cycle CPU**

  Load  | Ifetch | Reg/Dec | Exec | Mem | Wr

- **Multiple Cycle CPU**

  Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5

  Load  | Ifetch | Reg/Dec | Exec | Mem | Wr

- **Pipelined CPU**

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

  Load  | Ifetch | Reg/Dec | Exec | Mem | Wr

  Load  | Ifetch | Reg/Dec | Exec | Mem | Wr
Instruction Latencies and Throughput

• Single-Cycle CPU

| Load | Ifetch | Reg/Dec | Exec | Mem | Wr |

• Multiple Cycle CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

| Load | Ifetch | Reg/Dec | Exec | Mem | Wr |

• Pipelined CPU

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

| Load | Ifetch | Reg/Dec | Exec | Mem | Wr |
| Load | Ifetch | Reg/Dec | Exec | Mem | Wr |
| Load | Ifetch | Reg/Dec | Exec | Mem | Wr |
Instruction Latencies and Throughput

• Single-Cycle CPU

Load

| Ifetch | Reg/Dec | Exec | Mem | Wr |

• Multiple Cycle CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

Load

| Ifetch | Reg/Dec | Exec | Mem | Wr |

• Pipelined CPU

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

Load

| Ifetch | Reg/Dec | Exec | Mem | Wr |

Load

| Ifetch | Reg/Dec | Exec | Mem | Wr |

Load

| Ifetch | Reg/Dec | Exec | Mem | Wr |

Load

| Ifetch | Reg/Dec | Exec | Mem | Wr |
Pipelining Advantages
Pipelining Advantages

• Higher *maximum* throughput
Pipelining Advantages

• Higher maximum throughput

• Higher utilization of CPU resources
Pipelining Advantages

• Higher *maximum* throughput

• Higher *utilization* of CPU resources
Pipelining Advantages

• Higher \textit{maximum} throughput

• Higher \textit{utilization} of CPU resources

• But, more complicated datapath, more complex control
A Pipelined Datapath

- IF: Instruction fetch
- ID: Instruction decode and register fetch
- EX: Execution and effective address calculation
- MEM: Memory access
- WB: Write back
A Rough View of the Datapath
Execution in Pipelined Datapath
Execution in Pipelined Datapath

IF: Instruction Fetch
ID: Instruction Decode
EX: Execute
MEM: Memory Access
WB: Write Back

IM: Instruction Memory
ALU: Arithmetic Logic Unit
DM: Data Memory
Reg: Register

steady state
## Mixed Instructions in the Pipeline

<table>
<thead>
<tr>
<th>CC1</th>
<th>CC2</th>
<th>CC3</th>
<th>CC4</th>
<th>CC5</th>
<th>CC6</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mixed Instructions in the Pipeline

lw
  IM → Reg → ALU → DM → Reg

add
Mixed Instructions in the Pipeline

lw

add
Mixed Instructions in the Pipeline

lw
IM → Reg → ALU → DM → Reg

add
IM → Reg → ALU → Reg
Pipeline Principles

- All instructions that share a pipeline should have the same stages in the same order.
  - therefore, add does nothing during Mem stage
  - sw does nothing during WB stage
- All intermediate values must be latched each cycle.
- There is no functional block reuse
Pipelined Datapath

Instruction Fetch  Decode / Reg. Fetch  Execute  Memory  Write-back
Pipelined Datapath
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back
Pipeline In Execution

Instruction Fetch  Decode / Reg. Fetch  Execute  Memory  Write-back

add $10, $1, $2
Pipeline In Execution

Instruction Fetch          Decode / Reg. Fetch          Execute          Memory          Write-back

add $10, $1, $2

Diagram of pipeline stages with instructions and components labeled.
Pipeline In Execution

Instruction Fetch  Decode / Reg. Fetch  Execute  Memory  Write-back

add $10, $1, $2
**Pipeline In Execution**

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode / Reg. Fetch</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-back</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $12, 1000($4)</td>
<td>add $10, $1, $2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back

lw $12, 1000($4)

add $10, $1, $2
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back

lw $12, 1000($4)

add $10, $1, $2
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back

- **sub $15, $4, $1**
- **lw $12, 1000($4)**
- **add $10, $1, $2**
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back

sub $15, $4, $1
lw $12, 1000($4)
add $10, $1, $2
### Pipeline In Execution

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode / Reg. Fetch</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-back</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $15, $4, $1</td>
<td>lw $12, 1000($4)</td>
<td>add $10, $1, $2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

- **Instruction Fetch:**
  - Instruction memory
  - PC

- **Decode / Reg. Fetch:**
  - 4
  - Address
  - Add

- **Execute:**
  - Read register 1
  - Read register 2
  - Write register

- **Memory:**
  - Address
  - Data memory

- **Write-back:**
  - Write data

---

*Thursday, February 14, 13*
### Pipeline In Execution

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode / Reg. Fetch</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-back</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $15, $4, $1</td>
<td>lw $12, 1000($4)</td>
<td>add $10, $1, $</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Diagram:

- **Instruction Fetch**
  - Instruction memory
  - PC
  - Address

- **Decode / Reg. Fetch**
  - Add
  - Read register 1
  - Read register 2
  - Write register
  - Write data

- **Execute**
  - Shift left 2
  - Alu
  - Zero
  - Add result
  - Read data

- **Memory**
  - Address
  - Data memory
  - Read data
  - Write data

- **Write-back**
  - O Mux 1
  - O Mux 2
Pipeline In Execution

Instruction Fetch | Decode / Reg. Fetch | Execute | Memory | Write-back

- sub $15, $4, $1
- lw $12, 1000($4)
- add $10, $1, $
## Pipeline In Execution

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Decode / Reg. Fetch</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-back</th>
</tr>
</thead>
</table>

- **Instruction Fetch**
  - Instruction memory
  - Address

- **Decode / Reg. Fetch**
  - Read register 1
  - Read register 2
  - Write register
  - Write data

- **Execute**
  - ALU
  - Zero ALU result
  - Shift left 2
  - Add result
  - Read data 1

- **Memory**
  - Address
  - Data memory
  - Read data

- **Write-back**
  - 0 Mux

**Example Instructions**

- `sub $15, $4, $1`
- `lw $12, 1000($4)`

*Thursday, February 14, 13*
Pipeline In Execution

Instruction Fetch  Decode / Reg. Fetch  Execute  Memory  Write-back

lw $12, 1000($4)
sub $15, $4, $1
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back

sub $15, $4, $1
Pipeline In Execution

Instruction Fetch  |  Decode / Reg. Fetch  |  Execute  |  Memory  |  Write-back

sub $15, $4, $1
Pipeline In Execution

Instruction Fetch  Decode / Reg. Fetch  Execute  Memory  Write-back
Pipeline In Execution

Instruction Fetch

Decode / Reg. Fetch

Execute

Memory

Write-back
Pipeline with Controls
Pipeline with Controls

But...
Pipeline Control

- **FSM** not really appropriate.

- **Combinational logic!**
  
  - signals generated once, but follow instruction through the pipeline
Pipeline Control
Pipeline with Control Logic
Pipeline with Control Logic
Pipeline with Control Logic
Pipeline with Control Logic
Pipeline with Control Logic
# Pipeline Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>RegWrite</th>
<th>MemtoReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Format</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
Guess the Signal

add $10, $1, $2

Thursday, February 14, 13
Guess the Signal

lw $12, 1000($4)

add $10, $1, $2
Guess the Signal

sub $15, $4, $1
lw $12, 1000($4)
add $10, $1, $2
Guess the Signal

sub $15, $4, $1
Iw $12, 1000($4)
add $10, $1, $2
Guess the Signal

sub $15, $4, $1
Iw $12, 1000($4)
sub $15, $4, $1
Guess the Signal
Data Hazards

• When a result is needed in the pipeline before it is available, a “data hazard” occurs.

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Data Hazards

• When a result is needed in the pipeline before it is available, a “data hazard” occurs.

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Key Points
Key Points

• We achieve high throughput without reducing instruction latency.
Key Points

• We achieve high throughput without reducing instruction latency.

• Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).
Key Points

• We achieve high throughput without reducing instruction latency.

• Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).

• Pipelining uses combinational logic to generate (and registers to propagate) control signals.
Key Points

• We achieve high throughput without reducing instruction latency.

• Pipelining exploits a special kind of parallelism (parallelism between functionality required in different cycles).

• Pipelining uses combinational logic to generate (and registers to propagate) control signals.

• Pipelining creates potential hazards.