Single-Cycle implementation has poor performance
  • Cycle time longer than necessary for all but slowest instruction

Solution: break the instruction into smaller steps
  • Execute each step in one clock cycle
  • Cycle time: time it takes to execute the longest step
  • Design all the steps to have similar length

Advantages of the multiple cycle processor
  • Cycle time is much shorter
  • Functional units can be used > once/instruction (less HW)

Disadvantages of the multiple cycle processor
  • More timing paths to analyze and tune
  • Additional registers to store intermediate data values
A Simple Datapath

Instruction Fetch (IF)

- pc

Decode (RD)

- I$

- IR

- IR

- IR

Execute (EX)

- regs

- alu

Memory (MEM)

- D$

- lmd

Writeback (WB)

- smd

- smd

CDA 4150 − Pipelining
Execution Time

Time taken for 1 instruction
- Add up the execution times of each phase
- Each phase may take different amounts of time
- One instruction executes at a time

Example
- Pick some execution times out of the air
  - $t_{\text{fetch}}=60\text{ns}$, $t_{\text{decode}}=30\text{ns}$, $t_{\text{exec}}=50\text{ns}$, $t_{\text{mem}}=80\text{ns}$, $t_{\text{wb}}=20\text{ns}$
- Total execution time per instruction = $240\text{ns}$
Pipelining

We can execute multiple instructions at the same time!
Each instruction will be in a different phase of execution
Throughput will increase by the number of pipeline stages
Overlap different steps for consecutive instructions
  • Steps are called *pipeline stages*
  • Need latches after each stage to hold control/data for later stages
A new instruction enters the pipeline at IF on each clock
  • Takes 5 clocks to complete execution and leave the pipeline
  • Potential throughput of 1 CPI
### Pipeline Diagram

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>IF_I  \rightarrow RD_I  \rightarrow EX_I  \rightarrow MEM_I  \rightarrow WB_I</td>
</tr>
<tr>
<td>I+1</td>
<td>IF_I+1  \rightarrow RD_I+1  \rightarrow EX_I+1  \rightarrow MEM_I+1  \rightarrow WB_I+1</td>
</tr>
<tr>
<td>I+2</td>
<td>IF_I+2  \rightarrow RD_I+2  \rightarrow EX_I+2  \rightarrow MEM_I+2  \rightarrow WB_I+2</td>
</tr>
<tr>
<td>I+3</td>
<td>IF_I+3  \rightarrow RD_I+3  \rightarrow EX_I+3  \rightarrow MEM_I+3  \rightarrow WB_I+3</td>
</tr>
<tr>
<td>I+4</td>
<td>IF_I+4  \rightarrow RD_I+4  \rightarrow EX_I+4  \rightarrow MEM_I+4</td>
</tr>
</tbody>
</table>

Like assembly lines in manufacturing
Execution Time

Pipeline stages execute in parallel
  • Must wait for slowest one to finish

Pipeline overhead
  • Introducing pipelining registers adds latency
  • Let’s assume the overhead is 5ns

Our example
  • $t_{\text{fetch}}=60\text{ns}$, $t_{\text{decode}}=30\text{ns}$, $t_{\text{exec}}=50\text{ns}$, $t_{\text{mem}}=80\text{ns}$, $t_{\text{wb}}=20\text{ns}$
  • Longest state is $80\text{ns} + 5\text{ns} = 85\text{ns}$
  • Instruction executes in $5 \times 85 = 425\text{ns}$
  • But, we execute different parts of 5 instructions at same time!

At peak throughput, 1 instruction every 85ns
A Pipelined Datapath

Instruction Fetch (IF) → Decode (RD) → Execute (EX) → Memory (MEM) → Writeback (WB)

pc → 4

I$ → 1R → regs → =? → alu → D$ → lmd

16 → extend → 32
Pipeline Hazards

The major hurdle of pipelining
• Situations where next instruction cannot execute
• Reduce the performance of pipelining

Speedup = Pipeline depth/(1 + pipeline stalls/inst)

Want incredibly long pipelines, with no pipeline stalls

Good luck!

Long pipes increase likelihood of hazards
• Let’s look at pipeline resources used by instruction class
<table>
<thead>
<tr>
<th>Pipe stage</th>
<th>ALU</th>
<th>Memory</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Fetch-PC</td>
<td>Fetch-PC</td>
<td>Fetch-PC</td>
</tr>
<tr>
<td></td>
<td>Inst Cache</td>
<td>Inst Cache</td>
<td>Inst Cache</td>
</tr>
<tr>
<td>RD</td>
<td>Register Read</td>
<td>Register Read</td>
<td>Register Read</td>
</tr>
<tr>
<td>EX</td>
<td>ALU</td>
<td>ALU (address)</td>
<td>ALU (dest addr)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Compare logic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fetch-PC (taken)</td>
</tr>
<tr>
<td>MEM</td>
<td>N/A</td>
<td>Cache Tags</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cache Data</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>Register Write</td>
<td>Register Write (Load)</td>
<td></td>
</tr>
</tbody>
</table>

CDA 4150 – Pipelining
Types of Hazards

Three classes of hazards

- Data hazards
  - One instruction has a source operand that is the result of a previous instruction in the pipeline (Read-After Write: RAW)
  - There are other types of data hazards (later)

- Control hazards
  - The execution of an instruction depends on the resolution of a previous branch instruction in the pipeline
  - Becomes a big problem with deep pipelines

- Structural hazards
  - Two or more Instructions in the pipeline require the same hardware resource to progress
  - Most common instance is non-pipelined FU (multiplier)
Data Hazards

In MIPS R3000 pipeline, a data dependency occurs when an instruction’s source register is the destination register for either of the 2 prior instructions

- The simplest way to handle this is to stall the dependent instruction at RD until the required register has been written back
- This would cause a 2-clock delay when the instructions are consecutive

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $r3,$r1,$r2</td>
<td>IF  RD  EX  MEM  WB</td>
</tr>
<tr>
<td>sub $r5,$r3,$r4</td>
<td>IF  RD  RD  RD  EX  MEM  WB</td>
</tr>
</tbody>
</table>

CDA 4150 – Pipelining
Data Hazards

Instruction Fetch (IF)    Decode (RD)    Execute (EX)    Memory (MEM)    Writeback (WB)

pc → I$ → IR → regs → alu → D$ → lmd

add r1, r2, r3

CDA 4150 – Pipelining
Data Hazards

 Instruction Fetch (IF)  Decode (RD)  Execute (EX)  Memory (MEM)  Writeback (WB)

 CDA 4150 – Pipelining

add r1,r2,r3
add r4,r1,r5
Data Hazards

Instruction Fetch (IF) -> Decode (RD) -> Execute (EX) -> Memory (MEM) -> Writeback (WB)

- pc
- I$
- \text{add } r1, r2, r3
- \text{add } r4, r1, r5
- \text{add } r6, r4, r1

add r1, r2, r3
add r4, r1, r5
add r6, r4, r1

CDA 4150 - Pipelining
Data Hazards

Instruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)

pc

I$

regs

alu

D$

add r1,r2,r3
add r4,r1,r5
add r6,r4,r1

Insntruction Fetch (IF)
Decode (RD)
Execute (EX)
Memory (MEM)
Writeback (WB)

CDA 4150 - Pipelining
Data Hazards

Instruction Fetch (IF) → Decode (RD) → Execute (EX) → Memory (MEM) → Writeback (WB)

pc → + → IS → regs → alu → D$ → lmd

- add r1, r2, r3
- add r4, r1, r5
- add r6, r4, r1

CDA 4150 - Pipelining
## Bypassing

Performance can be improved by \textit{forwarding (bypassing)} a result from a later stage to an earlier stage 

- The result of an ALU instruction is known at the end of EX 
- The result of a Load instruction is known at the end of MEM

There is no delay when an ALU instruction executes 
There is 1 clock delay when a Load instruction is directly followed by a dependent instruction

- The Load instruction is said to have a \textit{latency} of 2 clocks

### Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $t3,$t1,$t2</code></td>
<td>IF  RD  EX  MEM  WB</td>
</tr>
<tr>
<td><code>sub $t5,$t3,$t4</code></td>
<td>IF  RD  EX  MEM  WB</td>
</tr>
<tr>
<td><code>lw $s1,0($t3)</code></td>
<td>IF  RD  EX  MEM  WB</td>
</tr>
<tr>
<td><code>addi $s2,$s1,1</code></td>
<td>IF  RD  RD  EX  MEM  WB</td>
</tr>
</tbody>
</table>

CDA 4150 – Pipelining
In this diagram, the pipeline stages are labeled as follows:

- **Instruction Fetch (IF)**
- **Decode (RD)**
- **Execute (EX)**
- **Memory (MEM)**
- **Writeback (WB)**

The diagram shows the process flow through these stages, with key components like the ALU and registers. The ALU bypass is highlighted, indicating a specific path for processing operations.
Mem to ALU Bypass

Instruction Fetch (IF)  
Decode (RD)  
Execute (EX)  
Memory (MEM)  
Writeback (WB)

- **Add r1, r2, r3**
- **Add r4, r1, r5**
- **Add r6, r4, r1**

**ALU bypass**

**Mem to ALU bypass**

CDA 4150 - Pipelining
Control Hazards

When a branch instruction is executed, execution of subsequent instructions depends on whether the branch is taken and the location of the destination.

A simple, but effective approach is to assume the branch is not taken and follow the sequential path.

The branch is resolved at the end of EX:

- If taken, cancel instructions in the sequential path and start fetching from the destination on the next clock
  - this results in a 2-clock delay for taken branches
- If not taken, continue sequentially

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Cycle =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>IF  RD  EX</td>
</tr>
<tr>
<td>beq $t0,$t1,$l1</td>
<td>IF  RD  EX  MEM  WB</td>
</tr>
<tr>
<td>$3$</td>
<td>IF  RD  --</td>
</tr>
<tr>
<td>$4$</td>
<td>IF  --  --</td>
</tr>
<tr>
<td>$l1$:</td>
<td>IF  RD  --</td>
</tr>
<tr>
<td>$5$</td>
<td>IF  RD  --</td>
</tr>
</tbody>
</table>

CDA 4150 – Pipelining
Load Delay
• Explicit 1-instruction delay in MIPS ISA
  – If no instruction can be scheduled following the load, nop required
    • MIPS == “Microprocessor without Interlocked Pipeline Stages
  – But other implementations may have different load delays!

Branch Delay
• Explicit 1-instruction delay in MIPS, HP-PA, SPARC
  – For MIPS, if no instruction can be scheduled, NOP required
    • Scheduled instruction must be safe to execute whether or not branch is taken (assembler schedules)
  – For HP-PA/SPARC the instruction following the branch is conditionally executed or squashed
Delayed Branches

Instruction Fetch (IF) -> Decode (RD) -> Execute (EX) -> Memory (MEM) -> Writeback (WB)

1. Instruction Fetch (IF)
2. Decode (RD)
3. Execute (EX)
4. Memory (MEM)
5. Writeback (WB)

CDA 4150 – Pipelining
Delayed Branches

Instruction Fetch (IF) → Decode (RD) → Execute (EX) → Memory (MEM) → Writeback (WB)

pc + 4 → IS

bgez r1, offset

16 → extend

alu

smd → D$

lmd
**Delayed Branches**

- **Instruction Fetch (IF)**
- **Decode (RD)**
- **Execute (EX)**
- **Memory (MEM)**
- **Writeback (WB)**

The diagram illustrates the pipeline stages of a computer's instruction execution process, highlighting the flow of instructions through the pipeline. Each stage is connected by arrows showing the dependency and flow of data, with labels such as `bgez r1,offset` and `nop` indicating specific instructions or operations. The pipeline stages are marked with `4` and `+`, indicating the number of stages and the flow of data through the pipeline. The diagram also includes symbols for `pc`, `I$, `regs`, `alu`, `D$`, and `lmd`, representing different components of the pipeline and data processing elements. The text `CDA 4150 - Pipelining` is visible, indicating the specific model or context of the pipeline architecture.
Delayed Branches

Instruction Fetch (IF)  Decode (RD)  Execute (EX)  Memory (MEM)  Writeback (WB)

pc  I$  +  =?  D$

I R  regs  alu  smd  lmd

bgez r1,offset  nop

CDA 4150 – Pipelining
Structural Hazards

Non-pipelined, multi-cycle functional units

- Integer multiply, divide

Can also have structural hazards on data cache

- Loads access tags/data in MEM
- Stores access tags in MEM, data in WB
- What if a load follows a store?

Structural hazards are detected in decode and stalled there

Only way to remove them is to add functional units

- Or pipeline them
- Or dual port them (caches)
Next Time

More complicated (deeper) pipelines
Data hazards revisited
Code scheduling for pipelines

What makes pipelining hard
• Interrupts
• Precise exceptions
• Branches and long pipes