## Comparison

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any instruction may reference memory</td>
<td>Only load/store references memory</td>
</tr>
<tr>
<td>Many instructions &amp; addressing modes</td>
<td>Few instructions &amp; addressing modes</td>
</tr>
<tr>
<td>Variable instruction formats</td>
<td>Fixed instruction formats</td>
</tr>
<tr>
<td>Single register set</td>
<td>Multiple register sets</td>
</tr>
<tr>
<td><strong>Multi-clock</strong> cycle instructions</td>
<td><strong>Single-clock</strong> cycle instructions</td>
</tr>
<tr>
<td>Micro-program interprets instructions</td>
<td>Hardware (FSM) executes instructions</td>
</tr>
<tr>
<td>Complexity is in the micro-program</td>
<td>Complexity is in the compiler</td>
</tr>
<tr>
<td>Less to no pipelining</td>
<td>Highly pipelined</td>
</tr>
<tr>
<td>Program code size small</td>
<td>Program code size large</td>
</tr>
</tbody>
</table>
Pipelining (Designing..., M.J. Quinn, '87)

Instruction Pipelining is the use of pipelining to allow more than one instruction to be in some stage of execution at the same time.

Cache memory is a small, fast memory unit used as a buffer between a processor and primary memory.

Ferranti ATLAS (1963):
- Pipelining reduced the average time per instruction by 375%
- Memory could not keep up with the CPU, needed a cache.
Memory Hierarchy

- Faster
- Cheaper
- More Capacity

 Registers

 Pipelining

 Cache memory

 Primary real memory

 Virtual memory (Disk, swapping)
Pipelining versus Parallelism (Designing..., M.J. Quinn, ‘87)

Most high-performance computers exhibit a great deal of concurrency.

However, it is not desirable to call every modern computer a parallel computer.

Pipelining and parallelism are 2 methods used to achieve concurrency.

Pipelining increases concurrency by dividing a computation into a number of steps.

Parallelism is the use of multiple resources to increase concurrency.
Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?
Pipelined Laundry: Start work ASAP

° Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining Lessons

° Pipelining doesn’t help \textit{latency} of single task, it helps \textit{throughput} of entire workload

° \textbf{Multiple} tasks operating simultaneously using different resources

° Potential speedup = \textit{Number pipe stages}

° Pipeline rate limited by \textit{slowest} pipeline stage

° Unbalanced lengths of pipe stages reduces speedup

° Time to “\textit{fill}” pipeline and time to “\textit{drain}” it reduces speedup

° Stall for Dependences
The Five Stages of Load

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory

- **Reg/Dec**: Registers Fetch and Instruction Decode

- **Exec**: Calculate the memory address

- **Mem**: Read the data from the Data Memory

- **Wr**: Write the data back to the register file
RISCEE 4 Architecture

Clock = load value into register

1. Instruction[7-0]
2. ALUop
   - 1 X+0
   - 2 X-Y
   - 3 0+Y
   - 4 0
   - 5 X+Y

3. ALU
   - ALU src A
   - ALU src B
   - ALU Out

4. Accumulator
   - Read Data
   - Write Data

5. RegWrite

6. MemWrite

7. MemRead

8. IRWrite

9. PCSrc

10. P0 \ (\neg \text{AluZero} \& \ BZ)

11. Clock

12. Clock = load value into register
Single Cycle, Multiple Cycle, vs. Pipeline

**Single Cycle Implementation:**

- Load
- Store
- Waste

**Multiple Cycle Implementation:**

- Load
- Store
- R-type

**Pipeline Implementation:**

- Load
- Store
- R-type
Why Pipeline?

- Suppose we execute 100 instructions

- **Single Cycle Machine**
  - 45 ns/cycle \( \times \) 1 CPI \( \times \) 100 inst = 4500 ns

- **Multicycle Machine**
  - 10 ns/cycle \( \times \) 4.6 CPI (due to inst mix) \( \times \) 100 inst = 4600 ns

- **Ideal pipelined machine**
  - 10 ns/cycle \( \times \) (1 CPI \( \times \) 100 inst + 4 cycle drain) = 1040 ns
Why Pipeline? Because the resources are there!

Time (clock cycles)

Inst 0
Inst 1
Inst 2
Inst 3
Inst 4

Resource
MemInst busy busy busy busy busy idle idle idle idle idle
MemData idle idle idle busy busy busy busy busy busy idle
RegRead idle busy busy busy busy busy idle idle idle
RegWrite idle busy busy busy busy busy busy busy busy idle
ALU idle idle idle busy busy busy busy busy busy idle
Can pipelining get us into trouble?

° Yes: **Pipeline Hazards**
  
  • **structural hazards**: attempt to use the same resource two different ways at the same time
    
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  
  • **data hazards**: attempt to use item before it is ready
    
    - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
    
    - instruction depends on result of prior instruction still in the pipeline
  
  • **control hazards**: attempt to make a decision before condition is evaluated
    
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    
    - branch instructions

° Can always resolve hazards by **waiting**

  • pipeline control must detect the hazard
  
  • take action (or delay action) to resolve hazards
Single Memory (Inst & Data) is a Structural Hazard

Structural hazards:
- attempt to use the same resource two different ways at the same time

Previous example: Separate InstMem and DataMem

<table>
<thead>
<tr>
<th>Load</th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Mem(Inst &amp; Data)</th>
<th>RegRead</th>
<th>RegWrite</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td></td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
<tr>
<td></td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
<td>busy</td>
</tr>
</tbody>
</table>

Detection is easy in this case!
Single Memory (Inst & Data) is a Structural Hazard

structure hazards:
  attempt to use the same resource two different ways at the same time

By change the architecture from a Harvard (separate instruction and data memory) to a von Neuman memory, we actually created a structural hazard!

Structural hazards can be avoid by changing

- hardware: design of the architecture (splitting resources)
- software: re-order the instruction sequence
- software: delay
Pipelining

- Improve performance by increasing instruction throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Stall on Branch

Program execution order (in instructions)

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

Time

- 2 ns
- 4 ns
- 2 ns

Figure 6.4
Predicting branches

Program execution order (in instructions)

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

Program execution order (in instructions)

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

Figure 6.5
Delayed branch

Program execution order (in instructions)

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

Figure 6.6
**Instruction pipeline**

**Pipeline stages**
- **IF** instruction fetch (read)
- **ID** instruction decode and register read (read)
- **EX** execute alu operation
- **MEM** data memory (read or write)
- **WB** Write back to register

**Resources**
- **Mem** instr. & data memory
- **RegRead1** register read port #1
- **RegRead2** register read port #2
- **RegWrite** register write
- **ALU** alu operation

**Figure 6.7**
Forwarding

Program execution order (in instructions)

add $s0, $t0, $t1

sub $t2, $s0, $t3

Time

2 4 6 8 10

Figure 6.8
Load Forwarding

Program execution order (in instructions)

lw $s0, 20($t1)

sub $t2, $s0, $t3

Figure 6.9
Figure 6.9

lw $t0, 0($t1)           $t0=Memory[0+$t1]
lw $t2, 4($t1)           $t2=Memory[4+$t1]
sw $t2, 0($t1)           Memory[0+$t1]=$t2
sw $t0, 4($t1)           Memory[4+$t1]=$t0

lw $t2, 4($t1)
lw $t0, 0($y1)
sw $t2, 0($t1)
sw $t0, 4($t1)

Figure 6.9
Basic Idea: split the datapath

What do we need to add to actually split the datapath into stages?
Can help with answering questions like:
- how many cycles does it take to execute this code?
- what is the ALU doing during cycle 4?
- use this representation to help understand datapaths
Pipeline datapath with registers

Figure 6.12
Load instruction fetch and decode

Figure 6.13
Load instruction execution

Figure 6.14
Load instruction memory and write back

Figure 6.15
Store instruction execution

Figure 6.16
Store instruction memory and write back

Figure 6.17
Load instruction: corrected datapath

Figure 6.18
Figure 6.19
Multi-clock-cycle pipeline diagram

Program execution order (in instructions)

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

Program execution order (in instructions)

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

Figure 6.20-21
Single-cycle #1-2

Figure 6.22
Figure 6.23
Single-cycle #5-6

Figure 6.24
Conventional Pipelined Execution Representation
Structural Hazards limit performance

- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI 1.3
  - otherwise resource is more than 100% utilized
Control Hazard Solutions

° Stall: wait until decision is clear
  • It's possible to move up decision to 2nd stage by adding hardware to check registers as being read

° Impact: 2 clock cycles per branch instruction => slow
Control Hazard Solutions

- Predict: guess one direction then back up if wrong
  - Predict not taken

- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)

- More dynamic scheme: history of 1 branch (- 90%)
Redefine branch behavior (takes place after next instruction) "delayed branch"

- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)

- As launch more instruction per clock cycle, less useful
Data Hazard on r1

add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
Data Hazard on r1:

- Dependencies backwards in time are hazards

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```
Data Hazard Solution:

- “Forward” result from one stage to another

- “or” OK if define read/write properly

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or  r8, r1, r9
xor r10, r1, r11
```
Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

```
Time (clock cycles)
```

```
lw r1,0(r2)
sub r4,r1,r3
```

- Can’t solve with forwarding:
- Must delay/stall instruction dependent on loads
## Pipelining the Load Instruction

The five independent functional units in the pipeline datapath are:

- Instruction Memory for the **Ifetch** stage
- Register File’s Read ports (bus A and busB) for the **Reg/Dec** stage
- ALU for the **Exec** stage
- Data Memory for the **Mem** stage
- Register File’s **Write** port (bus W) for the **Wr** stage
The Four Stages of R-type

- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory

- **Reg/Dec: Registers Fetch and Instruction Decode**

- **Exec:**
  - ALU operates on the two register operands
  - Update PC

- **Wr: Write the ALU output back to the register file**
Pipelining the R-type and Load Instruction

° **We have pipeline conflict or structural hazard:**
  - Two instructions try to write to the register file at the same time!
  - Only one write port
Important Observation

° Each functional unit can only be used once per instruction

° Each functional unit must be used at the same stage for all instructions:
  • Load uses Register File’s Write Port during its 5th stage
    
    | Load | Ifetch | Reg/Dec | Exec | Mem | Wr |
    |------|--------|---------|------|-----|----|
    | 1    | 2      | 3       | 4    | 5   |    |

  • R-type uses Register File’s Write Port during its 4th stage
    
    | R-type | Ifetch | Reg/Dec | Exec | Wr |
    |--------|--------|---------|------|----|
    | 1      | 2      | 3       | 4    |    |

° 2 ways to solve this pipeline hazard.
Solution 1: Insert “Bubble” into the Pipeline

- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.

- No instruction is started in Cycle 6!
Solution 2: Delay R-type’s Write by One Cycle

- Delay R-type’s register write by one cycle:
  - Now R-type instructions also use Reg File’s write port at Stage 5
  - Mem stage is a **NOOP** stage: nothing is being done.

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EECS 322 March 27, 2000
The Four Stages of Store

- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Write the data into the Data Memory**
The Three Stages of Beq

- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory

- **Reg/Dec:**
  - Registers Fetch and Instruction Decode

- **Exec:**
  - compares the two register operand,
  - select correct branch target address
  - latch into PC
Summary: 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.
Summary

- **Pipelining is a fundamental concept**
  - multiple steps using distinct resources

- **Utilize capabilities of the Datapath by pipelined instruction processing**
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards