Review: Models

**Single-cycle model** (non-overlapping)
- The instruction latency executes in a single cycle
- Every instruction and clock-cycle must be stretched to the slowest instruction (p.438)

**Multi-cycle model** (non-overlapping)
- The instruction latency executes in multiple-cycles
- The clock-cycle must be stretched to the slowest step
- Ability to share functional units within the execution of a single instruction

**Pipeline model** (overlapping, p. 522)
- The instruction latency executes in multiple-cycles
- The clock-cycle must be stretched to the slowest step
- The throughput is mainly one clock-cycle/instruction
- Gains efficiency by overlapping the execution of multiple instructions, increasing hardware utilization. (p. 377)
Review: Pipeline Hazards

**Pipeline hazards**
- Solution #1 always works (for non-realtime) applications: stall.

**Structural Hazards** (i.e. fetching same memory bank)
- Solution #2: partition architecture

**Control Hazards** (i.e. branching)
- Solution #1: stall! but decreases throughput
- Solution #2: guess and back-track
- Solution #3: delayed decision: delay branch & fill slot

**Data Hazards** (i.e. register dependencies)
- Worst case situation
- Solution #2: re-order instructions
- Solution #3: forwarding or bypassing: delayed load
Review: Single-Cycle Datapath

2 adders: PC+4 adder, Branch/Jump offset adder

Harvard Architecture: Separate instruction and data memory
Review: Multi vs. Single-cycle Processor Datapath

Combine adders: add 1½ Mux & 3 temp. registers, A, B, ALUOut
Combine Memory: add 1 Mux & 2 temp. registers, IR, MDR

Single-cycle = 1 ALU + 2 Mem + 4 Muxes + 2 adders + OpcodeDecoders
Multi-cycle = 1 ALU + 1 Mem + 5½ Muxes + 5 Reg (IR,A,B,MDR,ALUOut) + FSM
Review: Multi-cycle Processor Datapath

Single-cycle = 1 ALU + 2 Mem + 4 Muxes + 2 adders + OpcodeDecoders

Multi-cycle = 1 ALU + 1 Mem + 5½ Muxes + 5 Reg (IR,A,B,MDR,ALUOut) + FSM

5x32 = 160 additional FFs for multi-cycle processor over single-cycle processor
Figure 6.25

Datapath

Registers

160 FFs

+ 213 FFs

+ 16 FFs

213 + 16 = 229 additional FFs for pipeline over multi-cycle processor
Review: Overhead

**Single-cycle model**
- 8 ns Clock (125 MHz), (non-overlapping)
- 1 ALU + 2 adders
- 0 Muxes
- 0 Datapath Register bits (Flip-Flops)

**Multi-cycle model**
- 2 ns Clock (500 MHz), (non-overlapping)
- 1 ALU + Controller
- 5 Muxes
- 160 Datapath Register bits (Flip-Flops)

**Pipeline model**
- 2 ns Clock (500 MHz), (overlapping)
- 2 ALU + Controller
- 4 Muxes
- 373 Datapath + 16 Controlpath Register bits (Flip-Flops)
Review: Data Dependencies: no forwarding

sub $2,$1,$3

and $12,$2,$5

Suppose every instruction is dependant = 1 + 2 stalls = 3 clocks

\[
\text{MIPS} = \frac{\text{Clock}}{\text{CPI}} = \frac{500 \text{ Mhz}}{3} = 167 \text{ MIPS}
\]
Review: R-Format Data Dependencies: Hazard Conditions

1a Data Hazard (2 stalls):

sub $2, $1, $3
and $12, $2, $5

EX/MEM.$rd = ID/EX.$rs

sub $rd, $rs, $rt
and $rd, $rs, $rt

1b Data Hazard (2 stalls):

sub $2, $1, $3
and $12, $1, $2

EX/MEM.$rd = ID/EX.$rt

sub $rd, $rs, $rt
and $rd, $rs, $rt

2a Data Hazard (1 stall):

sub $2, $1, $3
and $12, $1, $5
or $13, $2, $1

MEM/WB.$rd = ID/EX.$rs

sub $rd, $rs, $rt
and $rd, $rs, $rt
or $13, $6, $2

2b Data Hazard (1 stall):

sub $2, $1, $3
and $12, $1, $5
or $13, $6, $2

MEM/WB.$rd = ID/EX.$rt

sub $rd, $rs, $rt
and $rd, $rs, $rt
or $13, $6, $2
Data Dependencies (hazard 1a and 1b): with forwarding

sub $2,$1,$3

and $12,$2,$5

Can R-Format dependencies completely be resolved by forwarding?

and $12,$2,$5
beq $12,$0,L7
Load Data Hazards: Hazard detection unit (page 490)

Stall Condition

Source

\[ IF/ID.\$rs \]
\[ IF/ID.\$rt \]

\} = ID/EX.\$rt \land ID/EX.MemRead=1

Destination

Stall Example

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{lw} \ $2, \ 20($1) \ and \ \text{and} \ $4, \ $2, \ $5 \ and \ \text{or} \ $8, \ $2, \ $6</td>
<td>\text{lw} \ $rt, \ \text{addr}($rs) \ and \ \text{and} \ $rd, \ $rs, \ $rt \ or \ \text{or} \ $rd, \ $rs, \ $rt</td>
</tr>
</tbody>
</table>

No Stall Example: \text{\textit{(only need to look at next instruction)}}

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{lw} \ $2, \ 20($1) \ and \ \text{and} \ $4, \ $1, \ $5 \ or \ \text{or} \ $8, \ $2, \ $6</td>
<td>\text{lw} \ $rt, \ \text{addr}($rs) \ and \ \text{and} \ $rd, \ $rs, \ $rt \ or \ \text{or} \ $rd, \ $rs, \ $rt</td>
</tr>
</tbody>
</table>
Load Data Dependencies: with forwarding

Load $2,20($1)

and $4,$2,$5

Detected Data Hazard
IF/ID.$rs = EX/M.$rt

Load dependencies cannot be completely resolved by forwarding

Even through the Load stalls the next instruction, the stall time is added to the load instruction and not the next instruction.

Load time = 1 (no dependancy) to 2 (with dependency on next instruction)
Delay slot

Before

```assembly
add $4,$6,$6
beq $1,$3,L7
...
L7: lw $4, 50($7)
```

After

```assembly
beq $1,$3,7
add $4,$6,$6
...
L7: lw $4, 50($7)
```

Can you move the add instruction into the delay slot?

```assembly
add $4,$6,$6
beq $1,$4,L7
```

No - but a delay slot still requires an instruction

```assembly
add $4,$6,$6
beq $1,$4,L7
add $0,$0,$0
```
Branch Hazards: Soln #3, Delayed Decision

beq $1,$3,7

instruction was before the branch

add $4,$6,$6

Do not need to discard instruction

Iw $4, 50($7)

Decision made in ID stage: branch
## Summary: Instruction Hazards

<table>
<thead>
<tr>
<th></th>
<th>No-Forwarding</th>
<th>Forwarding</th>
<th>Hazard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R-Format</strong></td>
<td>1-3</td>
<td>1</td>
<td>Data</td>
</tr>
<tr>
<td><strong>Load</strong></td>
<td>1-3</td>
<td>1-2</td>
<td>Data, Structural</td>
</tr>
<tr>
<td><strong>Store</strong></td>
<td>1</td>
<td>1-2</td>
<td>Structural</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>No Delay Slot</th>
<th>Delay Slot</th>
<th>Hazard</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch</strong></td>
<td>2</td>
<td>1</td>
<td>Control</td>
</tr>
<tr>
<td>(decision is made in the ID stage)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>3</td>
<td>1</td>
<td>Control</td>
</tr>
<tr>
<td>(decision is made in the EX stage)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Structural Hazard:** Instruction & Data memory combined.
### Performance, page 504

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single-Cycle</th>
<th>Multi-Cycle Clocks</th>
<th>Pipeline Cycles</th>
<th>Instruction Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads</td>
<td>1</td>
<td>5</td>
<td>1.5 (50% dependancy)</td>
<td>23%</td>
</tr>
<tr>
<td>stores</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>13%</td>
</tr>
<tr>
<td>arithmetic</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>43%</td>
</tr>
<tr>
<td>branches</td>
<td>1</td>
<td>3</td>
<td>1.25 (25% dependancy)</td>
<td>19%</td>
</tr>
<tr>
<td>jumps</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2%</td>
</tr>
<tr>
<td>Clock speed</td>
<td>125 Mhz 8 ns</td>
<td>500 Mhz 2 ns</td>
<td>500 Mhz 2 ns</td>
<td>= Σ Cycles*Mix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>= Clock/CPI</td>
</tr>
<tr>
<td>MIPS</td>
<td>125 MIPS</td>
<td>125 MIPS</td>
<td>424 MIPS</td>
<td></td>
</tr>
</tbody>
</table>

- **Load instruction time**: $50\% \times (1 \text{ clock}) + 50\% \times (2 \text{ clocks}) = 1.5$
- **Branch time**: $75\% \times (1 \text{ clock}) + 25\% \times (2 \text{ clocks}) = 1.25$

Also known as the instruction **latency** with in a pipeline.
Instruction Pipelining is the use of pipelining to allow more than one instruction to be in some stage of execution at the same time.

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

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

- **Principle of Locality**
  states that programs access a relatively small portion of their address space at any instance of time

- **Two types of locality**
  - **Temporal locality** (locality in time)
    If an item is referenced, then the same item will tend to be referenced soon
    “the tendency to reuse recently accessed data items”
  
  - **Spatial locality** (locality in space)
    If an item is referenced, then nearby items will be referenced soon
    “the tendency to reference nearby data items”
Memories Technology and Principle of Locality

- Faster Memories are more expensive per bit
- Slower Memories are usually smaller in area size per bit

<table>
<thead>
<tr>
<th>Memory Technology</th>
<th>Typical access time</th>
<th>$ per Mbyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>5-25 ns</td>
<td>$100-$250</td>
</tr>
<tr>
<td>DRAM</td>
<td>60-120 ns</td>
<td>$5-$10</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>10-20 million ns</td>
<td>$0.10-$0.20</td>
</tr>
</tbody>
</table>
Memory Hierarchy

- CPU
- Registers
- Pipelining
- Cache memory
- Primary real memory
- Virtual memory (Disk, swapping)

Faster

Cheaper Cost $$$

More Capacity
Basic Cache System

Figure 1. Basic Cache System

- Processor
  - L2 Cache
  - L2 Cache Controller
  - PCI Bridge Memory Controller
    - PCI Bus
    - PCI-ISA Bridge
      - ISA Bus
    - Main Memory
Cache Terminology

**A hit** if the data requested by the CPU is in the upper level

**Hit rate** or **Hit ratio**

is the fraction of accesses found in the upper level

**Hit time**

is the time required to access data in the upper level

\[ \text{Hit time} = \text{<detection time for hit or miss>} + \text{<hit access time>} \]

**A miss** if the data is not found in the upper level

**Miss rate** or \((1 - \text{hit rate})\)

is the fraction of accesses **not** found in the upper level

**Miss penalty**

is the time required to access data in the lower level

\[ \text{Miss penalty} = \text{<lower access time>} + \text{<reload processor time>} \]
Cache Example

Figure 7.2

Time 1: Hit: in cache

Time 1: Miss

Time 2: fetch from lower level into cache

Time 3: deliver to CPU

Data are transferred

Hit time = Time 1

Miss penalty = Time 2 + Time 3
Cache Memory Technology: SRAM

• Why use SRAM (Static Random Access Memory)?

• **Speed.**
  The primary advantage of an SRAM over DRAM is speed.

  The fastest DRAMs on the market still require 5 to 10 processor clock cycles to access the first bit of data.

  SRAMs can operate at processor speeds of 250 MHz and beyond, with access and cycle times equal to the clock cycle used by the microprocessor.

• **Density.**
  when 64 Mb DRAMs are rolling off the production lines, the largest SRAMs are expected to be only 16 Mb.

Cache Memory Technology: SRAM (con’t)

• **Volatility.**
  Unlike DRAMs, SRAM cells do not need to be refreshed. SRAMs are available 100% of the time for reading & writing.

• **Cost.**
  If cost is the primary factor in a memory design, then DRAMs win hands down.

  If, on the other hand, performance is a critical factor, then a well-designed SRAM is an effective cost performance solution.
Cache Memory Technology: SRAM Block diagram

Figure 2. Simplified Block Diagram of a Synchronous SRAM
Cache Memory Technology: SRAM timing diagram

Figure 4. Reading from Memory (Flow Thru mode)

Note: DQ0 is the data associated with Address 0 (A0). DQ1 is the data associated with Address 1 (A1).
Cache Memory Technology: SRAM 1 bit cell layout

Figure 3. IBM’s 6-Transistor Memory Cell
Real transistor

- 3-D structure
- Real materials

Ref: http://www.msm.cam.ac.uk/dmg/teaching/m101999/Ch8/index.htm
Basic DRAM design

- DRAM replaces all but one transistors of flip-flop with a capacitor
- \(\Rightarrow\) smaller!
- Capacitor stores information
- Charge leakage requires periodic refreshment (sense & rewrite)
256Mb DRAM

- Increased vertical integration
- Word line passes over capacitor and contact
- Cell area $\sim 0.5 \mu m^2$
- Capacitor area smaller - dielectric must be thinner
- $\Rightarrow$ higher quality dielectric required
Memory Technology: DRAM Evolution

DRAM evolution (II)
DRAM development

- 4-KBIT
- 16-KBIT
- 64-KBIT
- 256-KBIT
- 1-MBIT
- 4-MBIT
- 16-MBIT
- 64-MBIT

Timeline:
- 1975
- 1980
- 1985
- 1990
- 1995
- 2000
Direct Mapped Cache

- **Direct Mapped:** assign the cache location based on the address of the word in memory

  \[
  \text{cache\_address} = \text{memory\_address} \mod \text{cache\_size};
  \]

Observe there is a Many-to-1 memory to cache relationship
Direct Mapped Cache: Data Structure

There is a Many-to-1 relationship between memory and cache.

How do we know whether the data in the cache corresponds to the requested word?

tags

- contain the address information required to identify whether a word in the cache corresponds to the requested word.

- tags need only to contain the upper portion of the memory address (often referred to as a page address)

valid bit

- indicates whether an entry contains a valid address
### Direct Mapped Cache: Temporal Example

#### Figure 7.6

<table>
<thead>
<tr>
<th>Index</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Memory[11010]</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Memory[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache: Worst case, always miss!

<table>
<thead>
<tr>
<th>Index</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>00</td>
<td>Memory[00110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache: Mips Architecture

Figure 7.7
## Modern Systems: Pentium Pro and PowerPC

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Pentium Pro</th>
<th>PowerPC 604</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache organization</td>
<td>Split instruction and data caches</td>
<td>Split instruction and data caches</td>
</tr>
<tr>
<td>Cache size</td>
<td>8 KB each for instructions/data</td>
<td>16 KB each for instructions/data</td>
</tr>
<tr>
<td>Cache associativity</td>
<td>Four-way set associative</td>
<td>Four-way set associative</td>
</tr>
<tr>
<td>Replacement</td>
<td>Approximated LRU replacement</td>
<td>LRU replacement</td>
</tr>
<tr>
<td>Block size</td>
<td>32 bytes</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Write policy</td>
<td>Write-back</td>
<td>Write-back or write-through</td>
</tr>
</tbody>
</table>