NOTE: The material for this lecture was taken from several different sources. They are listed in the corresponding sections.
Pipelining

From the Hennessy and Patterson Book: Computer Organization and Design: the hardware software interface, 3rd edition
Overview

- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.
Basic Concepts
Making the Execution of Programs Faster

- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.
Traditional Pipeline Concept

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes
Traditional Pipeline Concept

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?
Traditional Pipeline Concept

- Pipelined laundry takes 3.5 hours for 4 loads
Traditional Pipeline Concept

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences
Use the Idea of Pipelining in a Computer

Fetch + Execution

(a) Sequential execution

Instruction fetch unit → Execution unit

(b) Hardware organization

Interstage buffer B1

(c) Pipelined execution

Figure 8.1. Basic idea of instruction pipelining.
Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write

Clock cycle 1 2 3 4 5 6 7

Instruction

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>I₁</td>
<td>F₁</td>
<td>D₁</td>
<td>E₁</td>
<td>W₁</td>
<td>F₂</td>
<td>D₂</td>
<td>E₂</td>
</tr>
<tr>
<td>I₂</td>
<td>F₃</td>
<td>D₃</td>
<td>E₃</td>
<td>W₃</td>
<td>F₄</td>
<td>D₄</td>
<td>E₄</td>
</tr>
</tbody>
</table>

(a) Instruction execution divided into four steps

(b) Hardware organization

Figure 8.2. A 4-stage pipeline.
## Ideal Pipelining

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr:</td>
<td></td>
<td>i</td>
<td>i+1</td>
<td>i+2</td>
<td>i+3</td>
<td>i+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
</tbody>
</table>

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining
Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.
Role of Cache Memory

- Each pipeline stage is expected to complete in one clock cycle.
  - The clock period should be long enough to let the slowest pipeline stage to complete.
  - Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.
  (We’ll talk about memory in a moment)
Pipelining Idealisms

- Uniform subcomputations
  - Can pipeline into stages with equal delay

- Identical computations
  - Can fill pipeline with identical work

- Independent computations
  - No relationships between work units

- Are these practical?
  - No, but can get close enough to get significant speedup
Pipeline Performance

Figure 8.3 Effect of an execution operation taking more than one clock cycle
Pipeline Performance

- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.
Figure 8.4. Pipeline stall caused by a cache miss in F2.
Figure 8.5. Effect of a Load instruction on pipeline timing.
Pipeline Performance

- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.
Program Data Dependences

- **True dependence (RAW)**
  - j cannot execute until i produces its result
  
- **Anti-dependence (WAR)**
  - j cannot write its result until i has read its sources

- **Output dependence (WAW)**
  - j cannot write its result until i has written its result

\[
W_r(i) \cap R_d(j) \neq \emptyset
\]

\[
R_d(i) \cap W_r(j) \neq \emptyset
\]

\[
W_r(i) \cap W_r(j) \neq \emptyset
\]

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining
Data Hazards

- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.

- Hazard occurs
  - A ← 3 + A
  - B ← 4 × A

- No hazard
  - A ← 5 × C
  - B ← 20 + C

- When two operations depend on each other, they must be executed sequentially in the correct order.

- Another example:
  - Mul R2, R3, R4
  - Add R5, R4, R6
Control Dependences

- **Conditional branches**
  - Branch must execute to determine which instruction to fetch next
  - A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
  - The decision to branch cannot be made until the execution of that instruction has been completed.
  - Branch instructions represent about 20% of the dynamic instruction count of most programs.

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining
Resolution of Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining
Instruction Level Parallelism

Taken from Soner Onder
Michigan Technological University, Houghton MI and Notes from Hennessy and Patterson’s book
http://www.eecs.berkeley.edu/~pattrsn
Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance

- 2 approaches to exploit ILP:
  1) Rely on hardware to help discover and exploit the parallelism dynamically, and
  2) Rely on software technology to find parallelism, statically at compile-time
Forms of parallelism

- Process-level
  - How do we exploit it? What are the challenges?

- Thread-level
  - How do we exploit it? What are the challenges?

- Loop-level
  - What is really loop level parallelism? What percentage of a program’s time is spent inside loops?

- Instruction-level
  - Lowest level
Instruction-level parallelism (ILP)

- Briefly, ability to execute more than one instruction simultaneously.
- In order to achieve this goal, we should not have dependencies among instructions which are executing in parallel:
  - H/W terminology Data hazards
    (I.e. RAW WAR WAW)
  - S/W terminology Data dependencies
    Name & true dependencies
Dependencies

- Name dependencies
  - Output dependence
  - Anti-dependence
- Data
  - True dependence
- Control

Do you remember the hazards they may lead to?
Increasing ILP

- Techniques
  - Loop unrolling
  - Static Branch Prediction
    - Compiler
  - Dynamic Branch Prediction
    - At runtime
  - Dynamic Scheduling – Tomasulo’s Algorithm
    - Register renaming
Memory Hierarchy

The following sources are used for preparing the slides on memory:

- Lecture 14 from the course Computer architecture ECE 201 by Professor Mike Schulte.


- Lecture 6 from the course Systems Architectures II by Professors Jeremy R. Johnson and Anatole D. Ruslanov

- Some of figures are from Computer Organization and Design: The Hardware/Software Approach, Third Edition, by David Patterson and John Hennessy, are copyrighted material (COPYRIGHT 2004 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED).
The Big Picture

- The Five Classic Components of a Computer
- Memory is usually implemented as:
  - Dynamic Random Access Memory (DRAM) - for main memory
  - Static Random Access Memory (SRAM) - for cache
Memory Hierarchy

<table>
<thead>
<tr>
<th>Memory technology</th>
<th>Typical access time</th>
<th>$ per GB in 2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>0.5–5 ns</td>
<td>$4000–$10,000</td>
</tr>
<tr>
<td>DRAM</td>
<td>50–70 ns</td>
<td>$100–$200</td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>5,000,000–20,000,000 ns</td>
<td>$0.50–$2</td>
</tr>
</tbody>
</table>

Levels in the memory hierarchy:
- Level 1
- Level 2
- ... (repeated levels)
- Level n

Increasing distance from the CPU in access time:

Size of the memory at each level:

- Processor
- Data are transferred
Memory

- **SRAM:**
  - Value is stored on a pair of inverting gates
  - Very fast but takes up more space than DRAM (4 to 6 transistors)

- **DRAM:**
  - Value is stored as a charge on capacitor (must be refreshed)
  - Very small but slower than SRAM (factor of 5 to 10)
Memory Hierarchy: How Does it Work?

- **Temporal Locality** (Locality in Time):
  => Keep most recently accessed data items closer to the processor

- **Spatial Locality** (Locality in Space):
  => Move blocks consists of contiguous words to the upper levels
Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
  - **Hit Rate**: the fraction of memory access found in the upper level
  - **Hit Time**: Time to access the upper level which consists of RAM access time + Time to determine hit/miss

- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block the processor

- **Hit Time << Miss Penalty**
Memory Hierarchy of a Modern Computer System

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

---

**Speed (ns):**
- Processor: 1s
- Control: 10s
- Second Level Cache (SRAM): 100s
- Main Memory (DRAM): 10,000,000s (10s ms)
- Secondary Storage (Disk): 10,000,000,000s (10s sec)
- Tertiary Storage (Tape):

**Size (bytes):**
- Processor: 100s
- Control: Ks
- Second Level Cache (SRAM): Ms
- Main Memory (DRAM): Gs
- Secondary Storage (Disk): Ts
General Principles of Memory

- **Locality**
  - *Temporal Locality*: referenced memory is likely to be referenced again soon (e.g. code within a loop)
  - *Spatial Locality*: memory close to referenced memory is likely to be referenced soon (e.g., data in a sequentially access array)

- **Locality + smaller HW is faster = memory hierarchy**
  - *Levels*: each smaller, faster, more expensive/byte than level below
  - *Inclusive*: data found in upper level also found in the lower level
Cache

- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module
Cache operation - overview

- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot
Cache Design

- Size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Block Size
- Number of Caches
Relationship of Caches and Pipeline

Memory

I-$

Next PC

Adder

Address

Next PC

SEQ PC

MEM/WB

D-$

Next PC

Address

IF/ID

Reg File

ID/EX

EX/MEM

MEM/WB

WB Data

RD

RD

RD

Memory

Zero?

Sign

Extend

Imm

RS1

RS2

ALU

Adder

Data

Memory

RD

RD

RD
Cache/memory structure

(a) Cache

(b) Main memory
Block Placement

- Direct Mapped: Each block has only one place that it can appear in the cache.
- Fully associative: Each block can be placed anywhere in the cache.
- Set associative: Each block can be placed in a restricted set of places in the cache.
  - If there are n blocks in a set, the cache placement is called n-way set associative
Example: Direct Mapped Cache

- **Mapping:** memory mapped to one location in cache:
  
  \[(\text{Block address}) \mod (\text{Number of blocks in cache})\]

- **Number of blocks** is typically a power of two, i.e., cache location obtained from low-order bits of address.
Summary

- **Pipelines**
  - Increase throughput of instructions
  - May have stalls

- **ILP**
  - Exploit multiple units and pipeline
  - Avoid dependencies

- **Memory Hierarchy and cache**
  - Support ILP and pipelines