Advanced Computer Architecture

Course Goal:

Understanding important and emerging design techniques, machine structures, technology factors, evaluation methods that will determine the form of programmable processors in 21st Century.

Topics we will cover include:

- Support for Simultaneous Multithreading (SMT).  Alpha EV8.
- Vector processing. Vector Intelligent RAM (VIRAM).
- Digital Signal Processing (DSP) & Media Architectures & Processors.
- Introduction to Multiprocessors:
  - Single-Chip Multiprocessors: The Hydra Project
- Re-Configurable Computing and Processors.
- Advanced Branch Prediction Techniques.
- Storage: Redundant Arrays of Disks (RAID).
EECC551 Review

- Recent Trends in Computer Design.
- A Hierarchy of Computer Design.
- Computer Architecture’s Changing Definition.
- Computer Performance Measures.
- Instruction Pipelining.
- Branch Prediction.
- Instruction-Level Parallelism (ILP).
- Loop-Level Parallelism (LLP).
- Dynamic Pipeline Scheduling.
- Multiple Instruction Issue (CPI < 1): Superscalar vs. VLIW
- Dynamic Hardware-Based Speculation
- Cache Design & Performance.
Recent Trends in Computer Design

- The cost/performance ratio of computing systems have seen a steady decline due to advances in:
  - Integrated circuit technology: decreasing feature size, $\lambda$
    - Clock rate improves roughly proportional to improvement in $\lambda$
    - Number of transistors improves proportional to $\lambda^2$ (or faster).
  - Architectural improvements in CPU design.

- Microprocessor systems directly reflect IC improvement in terms of a yearly 35 to 55% improvement in performance.

- Assembly language has been mostly eliminated and replaced by other alternatives such as C or C++

- Standard operating Systems (UNIX, NT) lowered the cost of introducing new architectures.

- Emergence of RISC architectures and RISC-core architectures.

- Adoption of quantitative approaches to computer design based on empirical performance observations.
Microprocessor Architecture Trends

CISC Machines
instructions take variable times to complete

RISC Machines (microcode)
simple instructions, optimized for speed

RISC Machines (pipelined)
same individual instruction latency
greater throughput through instruction "overlap"

Superscalar Processors
multiple instructions executing simultaneously

Multithreaded Processors
additional HW resources (regs, PC, SP)
each context gets processor for x cycles

VLIW
"Superinstructions" grouped together
decreased HW control complexity

Single Chip Multiprocessors
duplicate entire processors
(tech soon due to Moore's Law)

SIMULTANEOUS MULTITHREADING
multiple HW contexts (regs, PC, SP)
each cycle, any context may execute
1988 Computer Food Chain

- Mainframe
- Supercomputer
- Mini-supercomputer
- Mini-computer
- Workstation
- PC
- Massively Parallel Processors
1997 Computer Food Chain

Mainframe

Supercomputer

Massively Parallel Processors

Mini-supercomputer

Mini-computer

Server

Work-station

PC

PDA
Mass-produced microprocessors a cost-effective high-performance replacement for custom-designed mainframe/minicomputer CPUs
Microprocessor Frequency Trend

1. Frequency doubles each generation
2. Number of gates/clock reduce by 25%

Processor freq scales by 2X per generation

Gate delays/clock

Intel
IBM Power PC
DEC

Gate delays/clock scales by 2X per generation

Intel
IBM Power PC
DEC

Processor freq scales by 2X per generation

Gate delays/clock scales by 2X per generation
**Microprocessor Transistor Count Growth Rate**

**Moore’s Law**
- 2X transistors/Chip
- Every 1.5 years

**Devices**:
- i4004
- i8080
- i8086
- i80286
- i80486
- i80386
- Pentium
- Pentium Pro
- PowerPC 620
- Sparc Ultra
- Alpha 21264
- Alpha 21164

**Growth Rates**
- Alpha 21264: 15 million
- Pentium Pro: 5.5 million
- PowerPC 620: 6.9 million
- Alpha 21164: 9.3 million
- Sparc Ultra: 5.2 million
### Increase of Capacity of VLSI Dynamic RAM Chips

<table>
<thead>
<tr>
<th>Year</th>
<th>Size (Megabit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>0.0625</td>
</tr>
<tr>
<td>1983</td>
<td>0.25</td>
</tr>
<tr>
<td>1986</td>
<td>1</td>
</tr>
<tr>
<td>1989</td>
<td>4</td>
</tr>
<tr>
<td>1992</td>
<td>16</td>
</tr>
<tr>
<td>1996</td>
<td>64</td>
</tr>
<tr>
<td>1999</td>
<td>256</td>
</tr>
<tr>
<td>2000</td>
<td>1024</td>
</tr>
</tbody>
</table>

1.55X/yr, or doubling every 1.6 years
## Recent Technology Trends (Summary)

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Speed (latency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>2x in 3 years</td>
<td>2x in 3 years</td>
</tr>
<tr>
<td>DRAM</td>
<td>4x in 3 years</td>
<td>2x in 10 years</td>
</tr>
<tr>
<td>Disk</td>
<td>4x in 3 years</td>
<td>2x in 10 years</td>
</tr>
</tbody>
</table>
Computer Technology Trends: *Evolutionary but Rapid Change*

- **Processor:**
  - 2X in speed every 1.5 years; 1000X performance in last decade.

- **Memory:**
  - DRAM capacity: > 2x every 1.5 years; 1000X size in last decade.
  - Cost per bit: Improves about 25% per year.

- **Disk:**
  - Capacity: > 2X in size every 1.5 years.
  - Cost per bit: Improves about 60% per year.
  - 200X size in last decade.
  - Only 10% performance improvement per year, due to mechanical limitations.

- **Expected State-of-the-art PC by end of year 2000:**
  - Processor clock speed: > 1500 MegaHertz (1.5 GigaHertz)
  - Memory capacity: > 500 MegaByte (0.5 GigaBytes)
  - Disk capacity: > 100 GigaBytes (0.1 TeraBytes)
## A Hierarchy of Computer Design

<table>
<thead>
<tr>
<th>Level</th>
<th>Name</th>
<th>Modules</th>
<th>Primitives</th>
<th>Descriptive Media</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Electronics</td>
<td>Gates, FF’s</td>
<td>Transistors, Resistors, etc.</td>
<td>Circuit Diagrams</td>
</tr>
<tr>
<td>2</td>
<td>Logic</td>
<td>Registers, ALU’s ...</td>
<td>Gates, FF’s ….</td>
<td>Logic Diagrams</td>
</tr>
<tr>
<td>3</td>
<td>Organization</td>
<td>Processors, Memories</td>
<td>Registers, ALU’s …</td>
<td>Register Transfer Notation (RTN)</td>
</tr>
<tr>
<td>4</td>
<td>Microprogramming</td>
<td>Assembly Language</td>
<td>Microinstructions</td>
<td>Microprogram</td>
</tr>
<tr>
<td>5</td>
<td>Assembly language programming</td>
<td>OS Routines</td>
<td>Assembly language Instructions</td>
<td>Assembly Language Programs</td>
</tr>
<tr>
<td>6</td>
<td>Procedural Programming</td>
<td>Applications</td>
<td>OS Routines</td>
<td>High-level Language Programs</td>
</tr>
<tr>
<td>7</td>
<td>Application</td>
<td>Systems</td>
<td>Procedural Constructs</td>
<td>Problem-Oriented Programs</td>
</tr>
</tbody>
</table>

Low Level - Hardware

Firmware

High Level - Software

---

**EECC722 - Shaaban**

#14  Lec #1  Fall 2000  9-6-2000
Hierarchy of Computer Architecture

High-Level Language Programs

Software

Application

Operating System

Compiler

Firmware

Instr. Set Proc.

I/O system

Datapath & Control

Digital Design

Circuit Design

Layout

Hardware

Software/Hardware Boundary

Machine Language Program

Logic Diagrams

Circuit Diagrams

Assembly Language Programs

Instruction Set Architecture

Microprogram

Register Transfer Notation (RTN)
Computer Architecture Vs. Computer Organization

- The term **Computer architecture** is sometimes erroneously restricted to computer instruction set design, with other aspects of computer design called implementation.

- More accurate definitions:
  - **Instruction set architecture (ISA)**: The actual programmer-visible instruction set and serves as the boundary between the software and hardware.
  - Implementation of a machine has two components:
    - **Organization**: includes the high-level aspects of a computer’s design such as: The memory system, the bus structure, the internal CPU unit which includes implementations of arithmetic, logic, branching, and data transfer operations.
    - **Hardware**: Refers to the specifics of the machine such as detailed logic design and packaging technology.

- In general, **Computer Architecture** refers to the above three aspects: Instruction set architecture, organization, and hardware.
Computer Architecture’s Changing Definition

- 1950s to 1960s: Computer Architecture Course = Computer Arithmetic
- 1970s to mid 1980s: Computer Architecture Course = Instruction Set Design, especially ISA appropriate for compilers
- 1990s: Computer Architecture Course = Design of CPU, memory system, I/O system, Multiprocessors
Recent Architectural Improvements

- Increased optimization and utilization of cache systems.
- Memory-latency hiding techniques.
- Optimization of pipelined instruction execution.
- Dynamic hardware-based pipeline scheduling.
- Improved handling of pipeline hazards.
- Improved hardware branch prediction techniques.
- Exploiting Instruction-Level Parallelism (ILP) in terms of multiple-instruction issue and multiple hardware functional units.
- Inclusion of special instructions to handle multimedia applications.
- High-speed bus designs to improve data transfer rates.
Current Computer Architecture Topics

**Input/Output and Storage**
- Disks, WORM, Tape
- RAID

**Emerging Technologies**
- Interleaving
- Bus protocols
- RAID

**Memory Hierarchy**
- DRAM
- L2 Cache
- L1 Cache
- Coherence, Bandwidth, Latency
- Addressing, Protection, Exception Handling

**VLSI**
- Instruction Set Architecture

**Pipelining, Hazard Resolution, Superscalar, Reordering, Branch Prediction, Speculation, VLIW, Vector, DSP, ...**

**Multiprocessing, Simultaneous CPU Multi-threading**

**Pipelining and Instruction Level Parallelism (ILP)**

**Thread Level Parallelism (TLB)**
Computer Performance Measures:
Program Execution Time

- For a specific program compiled to run on a specific machine “A”, the following parameters are provided:
  - The total instruction count of the program.
  - The average number of cycles per instruction (average CPI).
  - Clock cycle of machine “A”

- How can one measure the performance of this machine running this program?
  - Intuitively the machine is said to be faster or has better performance running this program if the total execution time is shorter.
  - Thus the inverse of the total measured program execution time is a possible performance measure or metric:

\[
\text{Performance}_A = \frac{1}{\text{Execution Time}_A}
\]

How to compare performance of different machines?
What factors affect performance? How to improve performance?
CPU Execution Time: The CPU Equation

- A program is comprised of a number of instructions, $I$
  - Measured in: instructions/program

- The average instruction takes a number of cycles per instruction (CPI) to be completed.
  - Measured in: cycles/instruction

- CPU has a fixed clock cycle time $C = 1/$clock rate
  - Measured in: seconds/cycle

- CPU execution time is the product of the above three parameters as follows:

  $$\text{CPU Time} = I \times \text{CPI} \times C$$

  $$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$
## Factors Affecting CPU Performance

The CPU time can be calculated using the formula:

\[ \text{CPU time} = \text{Seconds} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \]

### Table

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count I</th>
<th>CPI</th>
<th>Clock Cycle C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program</strong></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Instruction Set Architecture (ISA)</strong></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Organization</strong></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

*Note: X indicates the factor influences the corresponding aspect.*
Metrics of Computer Performance

Each metric has a purpose, and each can be misused.

- Execution time: Target workload, SPEC95, etc.
- (millions) of Instructions per second – MIPS
- (millions) of (F.P.) operations per second – MFLOP/s
- Megabytes per second.
- Cycles per second (clock rate).
Quantitative Principles of Computer Design

• Amdahl’s Law:

The performance gain from improving some portion of a computer is calculated by:

\[
\text{Speedup} = \frac{\text{Performance for entire task using the enhancement}}{\text{Performance for the entire task without using the enhancement}}
\]

or

\[
\text{Speedup} = \frac{\text{Execution time without the enhancement}}{\text{Execution time for entire task using the enhancement}}
\]
Performance Enhancement Calculations: Amdahl's Law

- The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used.

- Amdahl’s Law:

  Performance improvement or speedup due to enhancement E:

  \[
  \text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E} = \frac{\text{Performance with } E}{\text{Performance without } E}
  \]

  Suppose that enhancement E accelerates a fraction F of the execution time by a factor S and the remainder of the time is unaffected then:

  \[
  \text{Execution Time with } E = ((1-F) + \frac{F}{S}) \times \text{Execution Time without } E
  \]

  Hence speedup is given by:

  \[
  \text{Speedup}(E) = \frac{\frac{\text{Execution Time without } E}{(1-F) + \frac{F}{S} \times \text{Execution Time without } E}}{1} = \frac{1}{(1 - F) + \frac{F}{S}}
  \]
Pictorial Depiction of Amdahl’s Law

Enhancement E accelerates fraction F of execution time by a factor of S

Before:
Execution Time without enhancement E:

<table>
<thead>
<tr>
<th>Unaffected, fraction: (1 - F)</th>
<th>Affected fraction: F</th>
</tr>
</thead>
</table>

Unchanged

Unaffected, fraction: (1 - F)

F/S

After:
Execution Time with enhancement E:

\[ \text{Speedup}(E) = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + \frac{F}{S}} \]
Performance Enhancement Example

- For the RISC machine with the following instruction mix given earlier:

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

- If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:

\[
\text{Fraction enhanced} = F = 45\% \text{ or } .45 \\
\text{Unaffected fraction} = 100\% - 45\% = 55\% \text{ or } .55 \\
\text{Factor of enhancement} = \frac{5}{2} = 2.5 \\
\text{Using Amdahl’s Law:} \\
\text{Speedup(E)} = \frac{1}{(1 - F) + \frac{F}{S}} = \frac{1}{.55 + \frac{.45}{2.5}} = 1.37
\]

\[\text{CPI} = 2.2\]
Extending Amdahl's Law To Multiple Enhancements

Suppose that enhancement $E_i$ accelerates a fraction $F_i$ of the execution time by a factor $S_i$ and the remainder of the time is unaffected then:

$$Speedup = \frac{Original\ Execution\ Time}{\left(\left(1-\sum_i F_i\right) + \sum_i \frac{F_i}{S_i}\right) \times Original\ Execution\ Time}$$

$$Speedup = \frac{1}{\left(\left(1-\sum_i F_i\right) + \sum_i \frac{F_i}{S_i}\right)}$$

Note: All fractions refer to original execution time.
Amdahl's Law With Multiple Enhancements:
Example

- Three CPU performance enhancements are proposed with the following speedups and percentage of the code execution time affected:

  Speedup_1 = S_1 = 10  \quad \text{Percentage}_1 = F_1 = 20\%
  Speedup_2 = S_2 = 15  \quad \text{Percentage}_2 = F_2 = 15\%
  Speedup_3 = S_3 = 30  \quad \text{Percentage}_3 = F_3 = 10\%

- While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.

- What is the resulting overall speedup?

  \[
  \text{Speedup} = \frac{1}{(1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}}
  \]

  \[
  \text{Speedup} = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30] \\
  = 1 / [ .55 + .0333 ] \\
  = 1 / .5833 = 1.71
  \]
### Pictorial Depiction of Example

**Before:**
Execution Time with no enhancements: 1

- Unaffected, fraction: .55

**After:**
Execution Time with enhancements: \( .55 + .02 + .01 + .00333 = .5833 \)

- Speedup = \( 1 / .5833 = 1.71 \)

Note: All fractions refer to original execution time.
Evolution of Instruction Sets

- Single Accumulator (EDSAC 1950)
  - Accumulator + Index Registers
    (Manchester Mark I, IBM 700 series 1953)
  - Separation of Programming Model from Implementation

- High-level Language Based
  (B5000 1963)

- Concept of a Family
  (IBM 360 1964)

- General Purpose Register Machines

- Complex Instruction Sets
  (Vax, Intel 432 1977-80)

- Load/Store Architecture
  (CDC 6600, Cray 1 1963-76)
  - RISC
    (Mips, SPARC, HP-PA, IBM RS6000, ... 1987)
A "Typical" RISC

- 32-bit fixed format instruction (3 formats I,R,J)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions (based on register values)
- Delayed branch
Example: MIPS (- DLX)

Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td>Rs2/Opx</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EECC722 - Shaaban
Pipelining: Definitions

• Pipelining is an implementation technique where multiple operations on a number of instructions are overlapped in execution.

• An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction.

• Each step is called a pipe stage or a pipe segment.

• The stages or steps are connected one to the next to form a pipe -- instructions enter at one end and progress through the stage and exit at the other end.

• Throughput of an instruction pipeline is determined by how often an instruction exists the pipeline.

• The time to move an instruction one step down the line is equal to the machine cycle and is determined by the stage with the longest processing delay.
Simple DLX Pipelined Instruction Processing

<table>
<thead>
<tr>
<th>Instruction Number</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>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction I</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction I+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time to fill the pipeline

DLX Pipeline Stages:
- **IF** = Instruction Fetch
- **ID** = Instruction Decode
- **EX** = Execution
- **MEM** = Memory Access
- **WB** = Write Back

First instruction, I Completed

Last instruction, I+4 completed
FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.
A Pipelined DLX Datapath

- Obtained from multi-cycle DLX datapath by adding buffer registers between pipeline stages
- Assume register writes occur in first half of cycle and register reads occur in second half.

FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.
Pipeline Hazards

- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle.
- Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  - **Structural hazards**: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - **Data hazards**: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - **Control hazards**: Arise from the pipelining of conditional branches and other instructions that change the PC.
Performance of Pipelines with Stalls

• Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}
\]

• If pipelining overhead is ignored and we assume that the stages are perfectly balanced then:

\[
\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}
\]

• When all instructions take the same number of cycles and is equal to the number of pipeline stages then:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}
\]
### Structural Hazards

- In pipelined machines, overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

- If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:
  - when a machine has only one register file write port
  - or when a pipelined machine has a shared single-memory pipeline for data and instructions.
  → stall the pipeline for one cycle for register writes or memory data access
DLX with Memory
Unit Structural Hazards

3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.
Resolving A Structural Hazard with Stalling

FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.
Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined machine resulting in incorrect execution.
- Data hazards usually require one or more instructions to be stalled to ensure correct execution.
- Example:

  ADD R1, R2, R3
  SUB R4, R1, R5
  AND R6, R1, R7
  OR R8, R1, R9
  XOR R10, R1, R11

  - All the instructions after ADD use the result of the ADD instruction
  - SUB, AND instructions need to be stalled for correct execution.
Figure 3.9 The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.
Minimizing Data hazard Stalls by Forwarding

- Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.

- Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)

- For example, in the DLX pipeline with forwarding:
  - The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
  - Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed.
  - If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.
Minimizing Data hazard Stalls by Forwarding

• Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.

• Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)

• For example, in the DLX pipeline with forwarding:
  – The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
  – Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed.
  – If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.
Pipelined DLX with Forwarding

3.10 A set of instructions that depend on the ADD result use forwarding paths to avoid the data hazard.
FIGURE 3.20 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.
Data Hazard Classification

Given two instructions $I$, $J$, with $I$ occurring before $J$ in an instruction stream:

- **RAW (read after write):** A true data dependence
  $J$ tried to read a source before $I$ writes to it, so $J$ incorrectly gets the old value.

- **WAW (write after write):** A name dependence
  $J$ tries to write an operand before it is written by $I$.
  The writes end up being performed in the wrong order.

- **WAR (write after read):** A name dependence
  $J$ tries to write to a destination before it is read by $I$,
  so $I$ incorrectly gets the new value.

- **RAR (read after read):** Not a hazard.
Data Hazard Classification

- I (Write) → J (Read) → Shared Operand → Read after Write (RAW)
- I (Write) → Shared Operand → J (Write) → Write after Write (WAW)
- J (Write) → I (Read) → Shared Operand → Write after Read (WAR)
- J (Read) → I (Write) → Shared Operand → Read after Read (RAR) not a hazard
Data Hazards Requiring Stall Cycles

Program execution order (in instructions)

LW R1, 0(R2)

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6

IM  Reg  ALU  DM  Reg  ALU

Bubble

Bubble

Bubble

Bubble

Figure 3.13 The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.
Compiler Instruction Scheduling for Data Hazard Stall Reduction

- Many types of stalls resulting from data hazards are very frequent. For example:

  \[ A = B + C \]

  produces a stall when loading the second data value (B).

- Rather than allow the pipeline to stall, the compiler could sometimes schedule the pipeline to avoid stalls.

- Compiler pipeline or instruction scheduling involves rearranging the code sequence (instruction reordering) to eliminate the hazard.
## Compiler Instruction Scheduling Example

- For the code sequence:
  
  \[
  a = b + c \\
  d = e - f
  \]

- Assuming loads have a latency of one clock cycle, the following code or pipeline compiler schedule eliminates stalls:

  Original code with stalls:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Rb, b</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>Rc, c</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Ra, Rb, Rc</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>a, Ra</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>Re, e</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>Rf, f</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Rd, Re, Rf</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>d, Rd</td>
<td></td>
</tr>
</tbody>
</table>

  Scheduled code with no stalls:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Rb, b</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>Rc, c</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Ra, Rb, Rc</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>a, Ra</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Ra, Rb, Rc</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>Re, e</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>a, Ra</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>Rd, Re, Rf</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>d, Rd</td>
<td></td>
</tr>
</tbody>
</table>

- a, b, c, d, e, and f are in memory
## Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known.
- In current DLX pipeline, the conditional branch is resolved in the MEM stage resulting in three stall cycles as shown below:

<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch successor</td>
<td>IF</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Branch successor + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Branch successor + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch successor + 4</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch successor + 5</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three clock cycles are wasted for every branch for current DLX pipeline.
Reducing Branch Stall Cycles

Pipeline hardware measures to reduce branch stall cycles:

1- Find out whether a branch is taken earlier in the pipeline.
2- Compute the taken PC earlier in the pipeline.

In DLX:

– In DLX branch instructions BEQZ, BNEZ, test a register for equality to zero.
– This can be completed in the ID cycle by moving the zero test into that cycle.
– Both PCs (taken and not taken) must be computed early.
– Requires an additional adder because the current ALU is not useable until EX cycle.
– This results in just a single cycle stall on branches.
FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.
Static Compiler Branch Prediction

Two basic methods exist to statically predict branches at compile time:

1. By examination of program behavior and the use of information collected from earlier runs of the program.
   - For example, a program profile may show that most forward branches and backward branches (often forming loops) are taken. The simplest scheme in this case is to just predict the branch as taken.

2. To predict branches on the basis of branch direction, choosing backward branches as taken and forward branches as not taken.
Reduction of Branch Penalties: Delayed Branch

• When delayed branch is used, the branch is delayed by $n$ cycles, following this execution pattern:
  - conditional branch instruction
  - sequential successor$_1$
  - sequential successor$_2$
  - ........
  - sequential successor$_n$
  - branch target if taken

• The sequential successor instruction are said to be in the branch delay slots. These instructions are executed whether or not the branch is taken.

• In Practice, all machines that utilize delayed branching have a single instruction delay slot.

• The job of the compiler is to make the successor instructions valid and useful instructions.
# Delayed Branch Example

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction $(i + 1)$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 2$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 3$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 4$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction $(i + 1)$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**FIGURE 3.27** The behavior of a delayed branch is the same whether or not the branch is taken.
Branch-delay Slot: Canceling Branches

- In a canceling branch, a static compiler branch direction prediction is included with the branch-delay slot instruction.

- When the branch goes as predicted, the instruction in the branch delay slot is executed normally.

- When the branch does not go as predicted the instruction is turned into a no-op.

- Canceling branches eliminate the conditions on instruction selection in delay instruction strategies B, C

- The effectiveness of this method depends on whether we predict the branch correctly.
<table>
<thead>
<tr>
<th>Scheduling strategy</th>
<th>Requirements</th>
<th>Improves performance when?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) From before branch</td>
<td>Branch must not depend on the rescheduled instructions.</td>
<td>Always.</td>
</tr>
<tr>
<td>(b) From target</td>
<td>Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.</td>
<td>When branch is taken. May enlarge program if instructions are duplicated.</td>
</tr>
<tr>
<td>(c) From fall through</td>
<td>Must be OK to execute instructions if branch is taken.</td>
<td>When branch is not taken.</td>
</tr>
</tbody>
</table>

**FIGURE 3.29** Delayed-branch scheduling schemes and their requirements.

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 4)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**FIGURE 3.30** The behavior of a predicted-taken cancelling branch depends on whether the branch is taken or not.
Pipeline Performance Example

• Assume the following DLX instruction mix:

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>40%</td>
</tr>
<tr>
<td>Load</td>
<td>30% of which 25% are followed immediately by an instruction using the loaded value</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
</tr>
<tr>
<td>branch</td>
<td>20% of which 45% are taken</td>
</tr>
</tbody>
</table>

• What is the resulting CPI for the pipelined DLX with forwarding and branch address calculation in ID stage when using a branch not-taken scheme?

• CPI = Ideal CPI + Pipeline stall clock cycles per instruction

  = 1 + stalls by loads + stalls by branches

  = 1 + .3 x .25 x 1 + .2 x .45 x 1

  = 1 + .075 + .09

  = 1.165
Pipelining and Exploiting Instruction-Level Parallelism (ILP)

• Pipelining increases performance by overlapping the execution of independent instructions.

• The CPI of a real-life pipeline is given by:

\[
\text{Pipeline CPI} = \text{Ideal Pipeline CPI} + \text{Structural Stalls} + \text{RAW Stalls} + \text{WAR Stalls} + \text{WAW Stalls} + \text{Control Stalls}
\]

• A basic instruction block is a straight-line code sequence with no branches in, except at the entry point, and no branches out except at the exit point of the sequence.

• The amount of parallelism in a basic block is limited by instruction dependence present and size of the basic block.

• In typical integer code, dynamic branch frequency is about 15% (average basic block size of 7 instructions).
Increasing Instruction-Level Parallelism

- A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop (i.e. Loop Level Parallelism, LLP).

- This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present.

- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

```plaintext
for (i=1; i<=1000; i=i+1;)
x[i] = x[i] + y[i];
```

- In vector machines, utilizing vector instructions is an important alternative to exploit loop-level parallelism,

- Vector instructions operate on a number of data items. The above loop would require just four such instructions.
DLX Loop Unrolling Example

- For the loop:

```c
for (i=1; i<=1000; i++)
x[i] = x[i] + s;
```

The straightforward DLX assembly code is given by:

```
Loop:  LD         F0, 0 (R1)        ;F0=array element
       ADDD   F4, F0, F2        ;add scalar in F2
       SD          0(R1), F4         ;store result
       SUBI     R1, R1, 8          ;decrement pointer 8 bytes
       BENZ    R1, Loop         ;branch R1!=zero
```
## DLX FP Latency Assumptions

- All FP units assumed to be pipelined.
- The following FP operations latencies are used:

<table>
<thead>
<tr>
<th>Instruction Producing Result</th>
<th>Instruction Using Result</th>
<th>Latency In Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU Op</td>
<td>Another FP ALU Op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU Op</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FP ALU Op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>
Loop Unrolling Example (continued)

- This loop code is executed on the DLX pipeline as follows:

<table>
<thead>
<tr>
<th></th>
<th>Clock cycle</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F4, F0, F2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 0 (R1), F4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBI R1, R1, #8</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BENZ R1, Loop</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With delayed branch scheduling (swap SUBI and SD)

<table>
<thead>
<tr>
<th></th>
<th>Clock cycle</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0, 0(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F4, F0, F2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBI R1, R1, #8</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BENZ R1, Loop</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 8 (R1), F4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9 cycles per iteration

6 cycles per iteration
Loop Unrolling Example (continued)

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers:

<table>
<thead>
<tr>
<th>No scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
</tr>
<tr>
<td>LD</td>
</tr>
<tr>
<td>ADDD</td>
</tr>
<tr>
<td>SD</td>
</tr>
<tr>
<td>LD</td>
</tr>
<tr>
<td>ADDD</td>
</tr>
<tr>
<td>SD</td>
</tr>
<tr>
<td>LD</td>
</tr>
<tr>
<td>ADDD</td>
</tr>
<tr>
<td>SD</td>
</tr>
<tr>
<td>LD</td>
</tr>
<tr>
<td>ADDD</td>
</tr>
<tr>
<td>SD</td>
</tr>
<tr>
<td>SUBI</td>
</tr>
<tr>
<td>BNEZ</td>
</tr>
</tbody>
</table>

Three branches and three decrements of R1 are eliminated.

Load and store addresses are changed to allow SUBI instructions to be merged.

The loop runs in 27 assuming LD takes 2 cycles, each ADDD takes 3 cycles, the branch 2 cycles, other instructions 1 cycle, or 6.8 cycles for each of the four elements.
Loop Unrolling Example (continued)

When scheduled for DLX

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F6, -8 (R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F10, -16(R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F14, -24(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>ADDD</td>
<td>F8, F6, F2</td>
</tr>
<tr>
<td>ADDD</td>
<td>F12, F10, F2</td>
</tr>
<tr>
<td>ADDD</td>
<td>F16, F14, F2</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>SD</td>
<td>-8(R1), F8</td>
</tr>
<tr>
<td>SD</td>
<td>-16(R1), F12</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1,#32</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, Loop</td>
</tr>
<tr>
<td>SD</td>
<td>8(R1), F16;8-32 =-24</td>
</tr>
</tbody>
</table>

The execution time of the loop has dropped to 14 cycles, or 3.5 clock cycles per element compared to 6.8 before scheduling and 6 when scheduled but unrolled.

Unrolling the loop exposed more computation that can be scheduled to minimize stalls.
Loop-Level Parallelism (LLP) Analysis

• LLP analysis is normally done at the source level or close to it since assembly language and target machine code generation introduces a loop-carried dependence, in the registers used for addressing and incrementing.

• Instruction level parallelism (ILP) analysis is usually done when instructions are generated by the compiler.

• Analysis focuses on whether data accesses in later iterations are data dependent on data values produced in earlier iterations.

  e.g. in
  
  \[\text{for (i=1; i<=1000; i++)}\]
  
  \[x[i] = x[i] + s;\]

  the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of \(X[i]\) twice is within a single iteration.
LLP Analysis Examples

• In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1];} /* S2 */
```

– **S1** uses a value computed in an earlier iteration, since iteration i computes \( A[i+1] \) read in iteration \( i+1 \) (loop-carried dependence, prevents parallelism).

– **S2** uses the value \( A[i+1] \), computed by **S1** in the same iteration (not loop-carried dependence).
LLP Analysis Examples

- In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];           /*  S1  */
    B[i+1] = C[i] + D[i];        /*  S2  */
}
```

- $S_1$ uses a value computed by $S_2$ in a previous iteration (loop-carried dependence)

- This dependence is not circular (neither statement depend on itself; $S_1$ depends on $S_2$ but $S_2$ does not depend on $S_1$.

- Can be made parallel by replacing the code with the following:

```c
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```
Reduction of Data Hazards Stalls with Dynamic Scheduling

• So far we have dealt with data hazards in instruction pipelines by:
  – Result forwarding and bypassing to reduce latency and hide or reduce the effect of true data dependence.
  – Hazard detection hardware to stall the pipeline starting with the instruction that uses the result.
  – Compiler-based static pipeline scheduling to separate the dependent instructions minimizing actual hazards and stalls in scheduled code.

• Dynamic scheduling:
  – Uses a hardware-based mechanism to rearrange instruction execution order to reduce stalls at runtime.
  – Enables handling some cases where dependencies are unknown at compile time.
  – Similar to the other pipeline optimizations above, a dynamically scheduled processor cannot remove true data dependencies, but tries to avoid stalling.
Dynamic Pipeline Scheduling: *The Concept*

- Dynamic pipeline scheduling overcomes the limitations of in-order execution by allowing out-of-order instruction execution.

- Instruction are allowed to start executing out-of-order as soon as their operands are available.

Example:

In the case of in-order execution

SUBD must wait for DIVD to complete which stalled ADDD before starting execution

In out-of-order execution SUBD can start as soon as the values of its operands F8, F14 are available.

- This implies allowing out-of-order instruction commit (completion).
- May lead to imprecise exceptions if an instruction issued earlier raises an exception.
- This is similar to pipelines with multi-cycle floating point units.

```
DIVD    F0, F2, F4  
ADDD    F10, F0, F8  
SUBD    F12, F8, F14
```
Dynamic Pipeline Scheduling

- Dynamic instruction scheduling is accomplished by:
  
  - Dividing the Instruction Decode ID stage into two stages:
    - Issue: Decode instructions, check for structural hazards.
    - Read operands: Wait until data hazard conditions, if any, are resolved, then read operands when available.
      (All instructions pass through the issue stage in order but can be stalled or pass each other in the read operands stage).
  
  - In the instruction fetch stage IF, fetch an additional instruction every cycle into a latch or several instructions into an instruction queue.
  
  - Increase the number of functional units to meet the demands of the additional instructions in their EX stage.

- Two dynamic scheduling approaches exist:
  - Dynamic scheduling with a Scoreboard used first in CDC6600
  - The Tomasulo approach pioneered by the IBM 360/91
Dynamic Scheduling With A Scoreboard

• The scoreboard is a hardware mechanism that maintains an execution rate of one instruction per cycle by executing an instruction as soon as its operands are available and no hazard conditions prevent it.

• It replaces ID, EX, WB with four stages: ID1, ID2, EX, WB

• Every instruction goes through the scoreboard where a record of data dependencies is constructed (corresponds to instruction issue).

• A system with a scoreboard is assumed to have several functional units with their status information reported to the scoreboard.

• If the scoreboard determines that an instruction cannot execute immediately it executes another waiting instruction and keeps monitoring hardware units status and decide when the instruction can proceed to execute.

• The scoreboard also decides when an instruction can write its results to registers (hazard detection and resolution is centralized in the scoreboard).
FIGURE 4.3 The basic structure of a DLX processor with a scoreboard.
Instruction Execution Stages with A Scoreboard

1 **Issue (ID1):** If a functional unit for the instruction is available, the scoreboard issues the instruction to the functional unit and updates its internal data structure; **structural** and **WAW** hazards are resolved here. (this replaces part of **ID** stage in the conventional DLX pipeline).

2 **Read operands (ID2):** The scoreboard monitors the availability of the source operands when no earlier active instruction will written it and then tells the functional unit to read the the operands from the registers and start execution (**RAW** hazards resolved here dynamically).

3 **Execution (EX):** The functional unit starts execution upon receiving operands. When the results are ready it notifies the scoreboard (replaces **EX** in DLX).

4 **Write result (WB):** Once the scoreboard senses that a functional unit completed execution, it checks for **WAR** hazards and stalls the completing instruction if needed otherwise the write back is completed.
Three Parts of the Scoreboard

1. **Instruction status:** Which of 4 steps the instruction is in.

2. **Functional unit status:** Indicates the state of the functional unit (FU). Nine fields for each functional unit:
   - **Busy** Indicates whether the unit is busy or not
   - **Op** Operation to perform in the unit (e.g., + or –)
   - **Fi** Destination register
   - **Fj, Fk** Source-register numbers
   - **Qj, Qk** Functional units producing source registers Fj, Fk
   - **Rj, Rk** Flags indicating when Fj, Fk are ready

3. **Register result status:** Indicates which functional unit will write to each register, if one exists. Blank when no pending instructions will write that register.
Dynamic Scheduling: The Tomasulo Algorithm

- Developed at IBM and first used in IBM 360/91 in 1966, about 3 years after the debut of the scoreboard in the CDC 6600.
- Dynamically schedule the pipeline in hardware to reduce stalls.
- Differences between IBM 360 & CDC 6600 ISA.
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600.
  - IBM has 4 FP registers vs. 8 in CDC 6600.
- Current CPU architectures that can be considered descendants of the IBM 360/91 which implement and utilize a variation of the Tomasulo Algorithm include:

  Alpha 21264, HP 8000, MIPS 10000, Pentium III, Xeon, PowerPC G3
Tomasulo Algorithm Vs. Scoreboard

- Control & buffers distributed with Function Units (FU) Vs. centralized in Scoreboard:
  - FU buffers are called “reservation stations” which have pending instructions and operands and other instruction status info.
- Registers in instructions are replaced by values or pointers to reservation stations (RS):
  - This process is called register renaming.
  - Avoids WAR, WAW hazards.
  - Allows for hardware-based loop unrolling.
  - More reservation stations than registers are possible, leading to optimizations that compilers can’t achieve and prevents the number of registers from becoming a bottleneck.
- Instruction results go to FUs from RSs, not through registers, over Common Data Bus (CDB) that broadcasts results to all FUs.
- Loads and Stores are treated as FUs with RSs as well.
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue.
Dynamic Scheduling: The Tomasulo Approach

FIGURE 4.8 The basic structure of a DLX FP unit using Tomasulo's algorithm.
Reservation Station Components

• **Op**  Operation to perform in the unit (e.g., + or –)
• **Vj, Vk**  Value of Source operands
  – Store buffers have a single V field indicating result to be stored.
• **Qj, Qk**  Reservation stations producing source registers.
  (value to be written).
  – No ready flags as in Scoreboard; Qj,Qk=0 => ready.
  – Store buffers only have Qi for RS producing result.
• **Busy:**  Indicates reservation station or FU is busy.
• **Register result status:**  Indicates which functional unit will write each register, if one exists.
  – Blank when no pending instructions exist that will write to that register.
Three Stages of Tomasulo Algorithm

1 Issue: Get instruction from pending Instruction Queue.
   - Instruction issued to a free reservation station (no structural hazard).
   - Selected RS is marked busy.
   - Control sends available instruction operands to assigned RS.
     (renaming registers).

2 Execution (EX): Operate on operands.
   - When both operands are ready then start executing on assigned FU.
   - If all operands are not ready, watch Common Data Bus (CDB) for needed result.

3 Write result (WB): Finish execution.
   - Write result on Common Data Bus to all awaiting units
   - Mark reservation station as available.

- Normal data bus: data + destination (“go to” bus).
- Common Data Bus (CDB): data + source (“come from” bus):
  - 64 bits for data + 4 bits for Functional Unit source address.
  - Write if matches expected Functional Unit (produces result).
  - Does the result broadcast to waiting RSs.
Hardware Dynamic Branch Prediction

• Simplest method:
  – A branch prediction buffer or Branch History Table (BHT) indexed by low address bits of the branch instruction.
  – Each buffer location (or BHT entry) contains one bit indicating whether the branch was recently taken or not.
  – Always mispredicts in first and last loop iterations.

• To improve prediction accuracy, two-bit prediction is used:
  – A prediction must miss twice before it is changed.
  – Two-bit prediction is a specific case of n-bit saturating counter incremented when the branch is taken and decremented otherwise.

• Based on observations, the performance of two-bit BHT prediction is comparable to that of n-bit predictors.
Basic Dynamic Two-Bit Branch Prediction:

Two-bit Predictor State Transition Diagram

FIGURE 4.13 The states in a two-bit prediction scheme.
FIGURE 4.14 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.
From The Analysis of Static Branch Prediction:

DLX Performance Using Canceling Delay Branches

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% conditional branches</th>
<th>% conditional branches with empty slots</th>
<th>% conditional branches that are cancelling</th>
<th>% cancelling branches that are cancelled</th>
<th>% branches with cancelled delay slots</th>
<th>Total % branches with empty or cancelled delay slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>14%</td>
<td>18%</td>
<td>31%</td>
<td>43%</td>
<td>13%</td>
<td>31%</td>
</tr>
<tr>
<td>eqntott</td>
<td>24%</td>
<td>24%</td>
<td>50%</td>
<td>24%</td>
<td>12%</td>
<td>36%</td>
</tr>
<tr>
<td>espresso</td>
<td>15%</td>
<td>29%</td>
<td>19%</td>
<td>21%</td>
<td>4%</td>
<td>33%</td>
</tr>
<tr>
<td>gcc</td>
<td>15%</td>
<td>16%</td>
<td>33%</td>
<td>34%</td>
<td>11%</td>
<td>27%</td>
</tr>
<tr>
<td>li</td>
<td>15%</td>
<td>20%</td>
<td>55%</td>
<td>48%</td>
<td>26%</td>
<td>46%</td>
</tr>
<tr>
<td><strong>Integer average</strong></td>
<td><strong>17%</strong></td>
<td><strong>21%</strong></td>
<td><strong>38%</strong></td>
<td><strong>34%</strong></td>
<td><strong>13%</strong></td>
<td><strong>35%</strong></td>
</tr>
<tr>
<td>doduc</td>
<td>8%</td>
<td>33%</td>
<td>12%</td>
<td>62%</td>
<td>8%</td>
<td>41%</td>
</tr>
<tr>
<td>ear</td>
<td>10%</td>
<td>37%</td>
<td>36%</td>
<td>14%</td>
<td>5%</td>
<td>42%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>12%</td>
<td>0%</td>
<td>69%</td>
<td>24%</td>
<td>16%</td>
<td>17%</td>
</tr>
<tr>
<td>mdljdp2</td>
<td>9%</td>
<td>0%</td>
<td>86%</td>
<td>10%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>su2cor</td>
<td>3%</td>
<td>7%</td>
<td>17%</td>
<td>57%</td>
<td>10%</td>
<td>17%</td>
</tr>
<tr>
<td>FP average</td>
<td>8%</td>
<td>16%</td>
<td>44%</td>
<td>34%</td>
<td>9%</td>
<td>25%</td>
</tr>
<tr>
<td><strong>Overall average</strong></td>
<td><strong>12%</strong></td>
<td><strong>18%</strong></td>
<td><strong>41%</strong></td>
<td><strong>34%</strong></td>
<td><strong>11%</strong></td>
<td><strong>30%</strong></td>
</tr>
</tbody>
</table>

**FIGURE 3.31** Delayed and cancelling delay branches for DLX allow branch hazards to be hidden 70% of the time on average for these 10 SPEC benchmarks.
Prediction Accuracy of Basic Two-Bit Branch Predictors:

4096-entry buffer Vs. An Infinite Buffer Under SPEC89
Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch.

Example:

B1 if (aa==2)
    aa=0;
B2 if (bb==2)
    bb=0;
B3 if (aa!==bb)
    SUBI R3, R1, #2
    BENZ R3, L1 ; b1 (aa!=2)
    ADD R1, R0, R0 ; aa==0
    L1: SUBI R3, R1, #2
    BNEZ R3, L2 ; b2 (bb!=2)
    ADD R2, R0, R0 ; bb==0
    L2 SUB R3, R1, R2 ; R3=aa-bb
    BEQZ R3, L3 ; b3 (aa==bb)

Branch B3 is correlated with branches B1, B2. If B1, B2 are both not taken, then B3 will be taken. Using only the behavior of one branch cannot detect this behavior.
Correlating Two-Level Dynamic Branch Predictors

- Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches:
  - Record the pattern of the m most recently executed branches as taken or not taken.
  - Use that pattern to select the proper branch history table.

- In general, the notation: (m,n) predictor means:
  - Record last m branches to select between $2^m$ history tables.
  - Each table uses n-bit counters (each table entry has n bits).

- Basic two-bit BHT is then a (0,2) predictor.
Dynamic Branch Prediction: Example

if (d==0)  d=1;
if (d==1)

...  
L2:

FIGURE 4.16 Possible execution sequences for a code fragment.

<table>
<thead>
<tr>
<th>Initial value of d</th>
<th>d==0?</th>
<th>b1</th>
<th>Value of d before b2</th>
<th>d==1?</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>Not taken</td>
<td>1</td>
<td>Yes</td>
<td>Not taken</td>
</tr>
<tr>
<td>1</td>
<td>No</td>
<td>Taken</td>
<td>1</td>
<td>Yes</td>
<td>Not taken</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Taken</td>
<td>2</td>
<td>No</td>
<td>Taken</td>
</tr>
</tbody>
</table>

FIGURE 4.17 Behavior of a one-bit predictor initialized to not taken.

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>
Dynamic Branch Prediction: Example (continued)

if (d==0)  
  d=1;  
if (d==1)

\begin{verbatim}
BNEZ R1, L1          ; branch b1 (d!=0)
ADDI R1, R0, #1    ; d==0, so d=1
L1:  SUBI     R3, R1, #1
     BNEZ    R3, L2          ; branch b2 (d!=1)
     ...
L2:
\end{verbatim}

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Initial value of d & d==0? & b1 & Value of d before b2 & d==1? & b2 \\
\hline
0 & Yes & Not taken & 1 & Yes & Not taken \\
1 & No & Taken & 1 & Yes & Not taken \\
2 & No & Taken & 2 & No & Taken \\
\hline
\end{tabular}
\caption{Combinations and meaning of the taken/not taken prediction bits.}
\end{figure}

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
d=? & b1 prediction & b1 action & New b1 prediction & b2 prediction & b2 action & New b2 prediction \\
\hline
2 & NT/NT & T & T/NT & NT/NT & T & NT/T \\
0 & T/NT & NT & T/NT & NT/T & NT & NT/T \\
2 & T/NT & T & T/NT & NT/T & T & NT/T \\
0 & T/NT & NT & T/NT & NT/T & NT & NT/T \\
\hline
\end{tabular}
\caption{The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken.}
\end{figure}
Organization of A Correlating Two-level (2,2) Branch Predictor

FIGURE 4.20 A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.
Prediction Accuracy of Two-Bit Dynamic Predictors Under SPEC89
Further Reducing Control Stalls:

**Branch-Target Buffers**

- **PC of instruction to fetch**
- **Look up**
- **Predicted PC**

**Number of entries in branch-target buffer**

- **No:** instruction is not predicted to be branch. Proceed normally.
- **Yes:** then instruction is branch and predicted PC should be used as the next PC.

**Branch predicted taken or untaken**

**FIGURE 4.22** A branch-target buffer.
FIGURE 4.23 The steps involved in handling an instruction with a branch-target buffer.
Multiple Instruction Issue: CPI < 1

• To improve a pipeline’s CPI to be better [less] than one, and to utilize ILP better, a number of independent instructions have to be issued in the same pipeline cycle.

• Multiple instruction issue processors are of two types:
  – **Superscalar**: A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or dynamically (Tomasulo).
    • PowerPC, Sun UltraSparc, Alpha, HP 8000
  – **VLIW** (Very Long Instruction Word):
    A fixed number of instructions (3-6) are formatted as one long instruction word or packet (statically scheduled by the compiler).
    • Joint HP/Intel agreement (Itanium, Q2 2000).
    • Intel Architecture-64 (IA-64) 64-bit address:
      • Explicitly Parallel Instruction Computer (EPIC).

• Both types are limited by:
  – Available ILP in the program.
  – Specific hardware implementation difficulties.
Multiple Instruction Issue:

Superscalar Vs. VLIW

- Smaller code size.
- Binary compatibility across generations of hardware.
- Simplified Hardware for decoding, issuing instructions.
- No Interlock Hardware (compiler checks?)
- More registers, but simplified hardware for register ports.
## Superscalar Pipeline Operation

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Pipe stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer instruction</td>
<td>IF, ID, EX, MEM, WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF, ID, EX, MEM, WB</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF, ID, EX, MEM, WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF, ID, EX, MEM, WB</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF, ID, EX, MEM, WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF, ID, EX, MEM, WB</td>
</tr>
</tbody>
</table>

**FIGURE 4.26** Superscalar pipeline in operation.
Intel/HP VLIW “Explicitly Parallel Instruction Computing (EPIC)”

- Three instructions in 128 bit “Groups”; instruction template fields determines if instructions are dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show dependencies of more than three instructions.

- 128 integer registers + 128 floating point registers
  - No separate register files per functional unit as in old VLIW.

- Hardware checks dependencies
  (interlocks ⇒ binary compatibility over time)

- Predicated execution: An implementation of conditional instructions used to reduce the number of conditional branches used in the generated code ⇒ larger basic block size

- **IA-64**: Name given to instruction set architecture (ISA);
- **Merced**: Name of the first implementation (2000/2001??)
Intel/HP EPIC VLIW Approach

original source code

Exposé Instruction Parallelism

compiler

Optimize

Instruction Dependency Analysis

Exploit Parallelism: Generate VLIWs

Instruction 2
Instruction 1
Instruction 0
Template

128-bit bundle

Instruction 2
Instruction 1
Instruction 0
Template

EECC722 - Shaaban
Unrolled Loop Example for Scalar Pipeline

1 Loop: 
1. LD F0, 0(R1)  
2. LD F6, -8(R1)  
3. LD F10, -16(R1)  
4. LD F14, -24(R1)  
5. ADDD F4, F0, F2  
6. ADDD F8, F6, F2  
7. ADDD F12, F10, F2  
8. ADDD F16, F14, F2  
9. SD 0(R1), F4  
10. SD -8(R1), F8  
11. SD -16(R1), F12  
12. SUBI R1, R1, #32  
13. BNEZ R1, LOOP  
14. SD 8(R1), F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration

LD to ADDD: 1 Cycle  
ADDD to SD: 2 Cycles
## Loop Unrolling in Superscalar Pipeline:
(1 Integer, 1 FP/Cycle)

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F6,-8(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>LD F14,-24(R1)</td>
<td>ADDD F8,F6,F2</td>
<td>4</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F16,F14,F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1),F8</td>
<td>ADDD F20,F18,F2</td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>SUBI R1,R1,#40</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNEZ R1,LOOP</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)
Loop Unrolling in VLIW Pipeline
(2 Memory, 2 FP, 1 Integer / Cycle)

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op / branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td>ADDD F24,F22,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F8</td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td>SUBI R1,R1,#48</td>
<td>6</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>7</td>
</tr>
<tr>
<td>SD -0(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)
Superscalar Dynamic Scheduling

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume: 1 integer + 1 floating-point operations.
  - 1 Tomasulo control for integer, 1 for floating point.
- Issue at 2X Clock Rate, so that issue remains in order.
- Only FP loads might cause a dependency between integer and FP issue:
  - Replace load reservation station with a load queue; operands must be read in the order they are fetched.
  - Load checks addresses in Store Queue to avoid RAW violation
  - Store checks addresses in Load Queue to avoid WAR, WAW.
- Called “Decoupled Architecture”
Multiple Instruction Issue Challenges

• While a two-issue single Integer/FP split is simple in hardware, we get a CPI of 0.5 only for programs with:
  – Exactly 50% FP operations
  – No hazards of any type.

• If more instructions issue at the same time, greater difficulty of decode and issue operations arise:
  – Even for a 2-issue superscalar machine, we have to examine 2 opcodes, 6 register specifiers, and decide if 1 or 2 instructions can issue.

• VLIW: tradeoff instruction space for simple decoding
  – The long instruction word has room for many operations.
  – By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  – E.g. 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    • 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  – Need compiling technique that schedules across several branches.
Limits to Multiple Instruction Issue Machines

• Inherent limitations of ILP:
  – If 1 branch exist for every 5 instruction: How to keep a 5-way VLIW busy?
  – Latencies of unit adds complexity to the many operations that must be scheduled every cycle.
  – For maximum performance multiple instruction issue requires about:
    Pipeline Depth $\times$ No. Functional Units
    independent instructions per cycle.

• Hardware implementation complexities:
  – Duplicate FUs for parallel execution are needed.
  – More instruction bandwidth is essential.
  – Increased number of ports to Register File (datapath bandwidth):
    • VLIW example needs 7 read and 3 write for Int. Reg.
    & 5 read and 3 write for FP reg
  – Increased ports to memory (to improve memory bandwidth).
  – Superscalar decoding complexity may impact pipeline clock rate.
Hardware Support for Extracting More Parallelism

- Compiler ILP techniques (loop-unrolling, software Pipelining etc.) are not effective to uncover maximum ILP when branch behavior is not well known at compile time.

- Hardware ILP techniques:
  - **Conditional or Predicted Instructions**: An extension to the instruction set with instructions that turn into no-ops if a condition is not valid at run time.
  - **Speculation**: An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls:
    - **Static Speculation** by the compiler with hardware support:
      - The compiler labels an instruction as speculative and the hardware helps by ignoring the outcome of incorrectly speculated instructions.
      - Conditional instructions provide limited speculation.
    - **Dynamic Hardware-based Speculation**:
      - Uses dynamic branch-prediction to guide the speculation process.
      - Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction.
Conditional or Predicted Instructions

- Avoid branch prediction by turning branches into conditionally-executed instructions:
  
  ```markdown
  if (x) then (A = B op C) else NOP
  ```
  
  - If false, then neither store result nor cause exception: instruction is annulled (turned into NOP).
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move.
  - HP PA-RISC can annul any following instruction.
  - IA-64: 64 1-bit condition fields selected so conditional execution of any instruction.

- Drawbacks of conditional instructions
  
  - Still takes a clock cycle even if “annulled”.
  - Must stall if condition is evaluated late.
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline.
Dynamic Hardware-Based Speculation

- Combines:
  - Dynamic hardware-based branch prediction
  - Dynamic Scheduling: of multiple instructions to issue and execute out of order.

- Continue to dynamically issue, and execute instructions passed a conditional branch in the dynamically predicted branch direction, before control dependencies are resolved.
  - This overcomes the ILP limitations of the basic block size.
  - Creates dynamically speculated instructions at run-time with no compiler support at all.
  - If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory).
    - Addition of commit (retire or re-ordering) stage and forcing instructions to commit in their order in the code (i.e. to write results to registers or memory).
    - Precise exceptions are possible since instructions must commit in order.
Hardware-Based Speculation

Speculative Execution + Tomasulo’s Algorithm
Four Steps of Speculative Tomasulo Algorithm

1. **Issue** — Get an instruction from FP Op Queue
   
   If a reservation station and a reorder buffer slot are free, issue instruction & send operands & reorder buffer number for destination (this stage is sometimes called “dispatch”)

2. **Execution** — Operate on operands (EX)
   
   When both operands are ready then execute; if not ready, watch CDB for result; when both operands are in reservation station, execute; checks RAW (sometimes called “issue”)

3. **Write result** — Finish execution (WB)
   
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit** — Update registers, memory with reorder buffer result
   
   - When an instruction is at head of reorder buffer & the result is present, update register with result (or store to memory) and remove instruction from reorder buffer.
   
   - A mispredicted branch at the head of the reorder buffer flushes the reorder buffer (sometimes called “graduation”)

   $\Rightarrow$ Instructions issue, execute (EX), write result (WB) out of order but must commit in order.
Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

• HW determines address conflicts.
• HW provides better branch prediction.
• HW maintains precise exception model.
• HW does not execute bookkeeping instructions.
• Works across multiple implementations
• SW speculation is much easier for HW design.
Memory Hierarchy: The motivation

- The gap between CPU performance and main memory has been widening with higher performance CPUs creating performance bottlenecks for memory access instructions.

- The memory hierarchy is organized into several levels of memory with the smaller, more expensive, and faster memory levels closer to the CPU: registers, then primary Cache Level ($L_1$), then additional secondary cache levels ($L_2, L_3$…), then main memory, then mass storage (virtual memory).

- Each level of the hierarchy is a subset of the level below: data found in a level is also found in the level below but at lower speed.

- Each level maps addresses from a larger physical memory to a smaller level of physical memory.

- This concept is greatly aided by the principal of locality both temporal and spatial which indicates that programs tend to reuse data and instructions that they have used recently or those stored in their vicinity leading to working set of a program.
Memory Hierarchy: Motivation

Processor-Memory (DRAM) Performance Gap

μProc 60%/yr.

Processor-Memory Performance Gap:
(grows 50% / year)

DRAM 7%/yr.
Cache Design & Operation Issues

• Q1: Where can a block be placed cache? (Block placement strategy & Cache organization)
  – Fully Associative, Set Associative, Direct Mapped.

• Q2: How is a block found if it is in cache? (Block identification)
  – Tag/Block.

• Q3: Which block should be replaced on a miss? (Block replacement)
  – Random, LRU.

• Q4: What happens on a write? (Cache write policy)
  – Write through, write back.
Cache Organization & Placement Strategies

Placement strategies or mapping of a main memory data block onto cache block frame addresses divide cache into three organizations:

1. **Direct mapped cache:** A block can be placed in one location only, given by:
   \[(\text{Block address}) \mod (\text{Number of blocks in cache})\]

2. **Fully associative cache:** A block can be placed anywhere in cache.

3. **Set associative cache:** A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:
   \[(\text{Block address}) \mod (\text{Number of sets in cache})\]

If there are \(n\) blocks in a set the cache placement is called \(n\)-way set-associative.
Locating A Data Block in Cache

- Each block frame in cache has an address tag.
- The tags of every cache block that might contain the required data are checked in parallel.
- A valid bit is added to the tag to indicate whether this entry contains a valid address.
- The address from the CPU to cache is divided into:
  - A block address, further divided into:
    - An index field to choose a block set in cache.
      (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - A block offset to select the data from the block.
Address Field Sizes

Physical Address Generated by CPU

Block Address

- Tag
- Index
- Block Offset

\[
\text{Block offset size} = \log_2(\text{block size})
\]

\[
\text{Index size} = \log_2(\text{Total number of blocks/associativity})
\]

\[
\text{Tag size} = \text{address size} - \text{index size} - \text{offset size}
\]
Direct-Mapped Cache Example

1024 Blocks
Each block = one word
Can cache up to $2^{32}$ bytes of memory
Four-Way Set Associative Cache: DLX Implementation Example

Address

31 30 ... 12 11 10 9 8 ... 3 2 1 0

Index Field

22
8

Index Field

256 sets
1024 block frames
Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

Sample Data

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU Random</td>
<td>LRU Random</td>
<td>LRU Random</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.18% 5.69%</td>
<td>4.67% 5.29%</td>
<td>4.39% 4.96%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88% 2.01%</td>
<td>1.54% 1.66%</td>
<td>1.39% 1.53%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15% 1.17%</td>
<td>1.13% 1.13%</td>
<td>1.12% 1.12%</td>
</tr>
</tbody>
</table>
Cache Read/Write Operations

- Statistical data suggest that reads (including instruction fetches) dominate processor cache accesses (writes account for 25% of data cache traffic).

- In cache reads, a block is read at the same time while the tag is being compared with the block address. If the read is a hit the data is passed to the CPU, if a miss it ignores it.

- In cache writes, modifying the block cannot begin until the tag is checked to see if the address is a hit.

- Thus for cache writes, tag checking cannot take place in parallel, and only the specific data (between 1 and 8 bytes) requested by the CPU can be modified.

- Cache is classified according to the write and memory update strategy in place: write through, or write back.
Cache Write Strategies

1. Write Though: Data is written to both the cache block and to a block of main memory.
   - The lower level always has the most updated data; an important feature for I/O and multiprocessing.
   - Easier to implement than write back.
   - A write buffer is often used to reduce CPU write stall while data is written to memory.

2. Write back: Data is written or updated only to the cache block. The modified cache block is written to main memory when it’s being replaced from cache.
   - Writes occur at the speed of cache
   - A status bit called a dirty bit, is used to indicate whether the block was modified while in cache; if not the block is not written to main memory.
   - Uses less memory bandwidth than write through.
Cache Write Miss Policy

• Since data is usually not needed immediately on a write miss two options exist on a cache write miss:

Write Allocate:
The cache block is loaded on a write miss followed by write hit actions.

No-Write Allocate:
The block is modified in the lower level (lower cache level, or main memory) and not loaded into cache.

While any of the above two write miss policies can be used with either write back or write through:

• Write back caches use write allocate to capture subsequent writes to the block in cache.

• Write through caches usually use no-write allocate since subsequent writes still have to go to memory.
Cache Performance

For a CPU with a single level (L1) of cache and no stalls for cache hits:

CPU time = (CPU execution clock cycles + Memory stall clock cycles) \times \text{clock cycle time}

Memory stall clock cycles =

(Reads \times \text{Read miss rate} \times \text{Read miss penalty}) + (Writes \times \text{Write miss rate} \times \text{Write miss penalty})

If write and read miss penalties are the same:

Memory stall clock cycles =

Memory accesses \times \text{Miss rate} \times \text{Miss penalty}
Cache Performance Example

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- \( \text{CPI}_{\text{execution}} = 1.1 \)
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{mem stalls per instruction}
\]

\[
\text{Mem Stalls per instruction} = \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}
\]

\[
\text{Mem accesses per instruction} = 1 + .3 = 1.3
\]

\[
\text{Mem Stalls per instruction} = 1.3 \times .015 \times 50 = 0.975
\]

\[
\text{CPI} = 1.1 + .975 = 2.075
\]

The ideal CPU with no misses is \( 2.075/1.1 = 1.88 \) times faster
Typical Cache Performance Data Using SPEC92

<table>
<thead>
<tr>
<th>Size</th>
<th>Instruction cache</th>
<th>Data cache</th>
<th>Unified cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>3.06%</td>
<td>24.61%</td>
<td>13.34%</td>
</tr>
<tr>
<td>2 KB</td>
<td>2.26%</td>
<td>20.57%</td>
<td>9.78%</td>
</tr>
<tr>
<td>4 KB</td>
<td>1.78%</td>
<td>15.94%</td>
<td>7.24%</td>
</tr>
<tr>
<td>8 KB</td>
<td>1.10%</td>
<td>10.19%</td>
<td>4.57%</td>
</tr>
<tr>
<td>16 KB</td>
<td>0.64%</td>
<td>6.47%</td>
<td>2.87%</td>
</tr>
<tr>
<td>32 KB</td>
<td>0.39%</td>
<td>4.82%</td>
<td>1.99%</td>
</tr>
<tr>
<td>64 KB</td>
<td>0.15%</td>
<td>3.77%</td>
<td>1.35%</td>
</tr>
<tr>
<td>128 KB</td>
<td>0.02%</td>
<td>2.88%</td>
<td>0.95%</td>
</tr>
</tbody>
</table>

FIGURE 5.7 Miss rates for instruction, data, and unified caches of different sizes.
Cache Performance Example

To compare the performance of either using a 16-KB instruction cache and a 16-KB data cache as opposed to using a unified 32-KB cache, we assume a hit to take one clock cycle and a miss to take 50 clock cycles, and a load or store to take one extra clock cycle on a unified cache, and that 75% of memory accesses are instruction references. Using the miss rates for SPEC92 we get:

Overall miss rate for a split cache = (75% x 0.64%) + (25% x 6.74%) = 2.1%

From SPEC92 data a unified cache would have a miss rate of 1.99%

Average memory access time =

= % instructions ( Read hit time + Read miss rate x Miss penalty) + % data x ( Write hit time + Write miss rate x Miss penalty)

For split cache:

Average memory access time_{split} = 75% x ( 1 + 0.64 x 50) + (1+6.47% x 50) = 2.05 cycles

For unified cache:

Average memory access time_{unified} = 75% x ( 1 + 1.99% x 50) + 25% x ( 1 + 1+ 1.99% x 50)

= 2.24 cycles
3 Levels of Cache

CPU

L1 Cache
Hit Rate = $H_1$, Hit time = 1 cycle

L2 Cache
Hit Rate = $H_2$, Hit time = $T_2$ cycles

L3 Cache
Hit Rate = $H_3$, Hit time = $T_3$

Main Memory

Memory access penalty, $M$
3-Level Cache Performance

CPUtime = IC x \( (CPI_{\text{execution}} + \text{Mem Stall cycles per instruction}) \times C \)

Mem Stall cycles per instruction = Mem accesses per instruction \( \times \) Stall cycles per access

• For a system with 3 levels of cache, assuming no penalty when found in \( L_1 \) cache:

Stall cycles per memory access =

\[
[\text{miss rate } L_1] \times [\text{Hit rate } L_2 \times \text{Hit time } L_2 \\
+ \text{Miss rate } L_2 \times (\text{Hit rate } L_3 \times \text{Hit time } L_3 \\
+ \text{Miss rate } L_3 \times \text{Memory access penalty}) ] = \\
[1 - H_1] \times [ H_2 \times T_2 \\
+ (1-H_2) \times (H_3 \times (T_2 + T_3) \\
+ (1 - H_3) \times M ) ]
\]
Three Level Cache Performance Example

- CPU with \( CPI_{\text{execution}} = 1.1 \) running at clock rate = 500 MHZ
- 1.3 memory accesses per instruction.
- \( L_1 \) cache operates at 500 MHZ with a miss rate of 5%
- \( L_2 \) cache operates at 250 MHZ with miss rate 3%, \( (T_2 = 2 \text{ cycles}) \)
- \( L_3 \) cache operates at 100 MHZ with miss rate 1.5%, \( (T_3 = 5 \text{ cycles}) \)
- Memory access penalty, \( M = 100 \text{ cycles} \)

Find CPI.

With single \( L_1 \), \( CPI = 1.1 + 1.3 \times 0.05 \times 100 = 7.6 \)

\[
CPI = CPI_{\text{execution}} + \text{Mem Stall cycles per instruction}
\]

\[
\text{Mem Stall cycles per instruction} = \text{Mem accesses per instruction} \times \text{Stall cycles per access}
\]

\[
\text{Stall cycles per memory access} = [1 - H_1] \times [H_2 \times T_2 + (1 - H_2) \times H_3 \times (T_2 + T_3) + (1 - H_3) \times M]
\]

\[
= [0.05] \times [0.97 \times 2 + (0.03) \times (0.985 \times (2+5) + 0.015 \times 100)]
\]

\[
= 0.05 \times [1.94 + 0.03 \times (6.895 + 1.5)]
\]

\[
= 0.05 \times [1.94 + 0.274] = 0.11
\]

\[
CPI = 1.1 + 1.3 \times 0.11 = 1.24
\]
## Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>MR</th>
<th>MP</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td></td>
<td>+</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Subblock Placement</td>
<td></td>
<td>+</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td></td>
<td>+</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td></td>
<td>+</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Second Level Caches</td>
<td></td>
<td>+</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Small &amp; Simple Caches</td>
<td></td>
<td></td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Avoiding Address Translation</td>
<td></td>
<td></td>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>Pipelining Writes</td>
<td></td>
<td></td>
<td>+</td>
<td>1</td>
</tr>
</tbody>
</table>

### Miss rate

- Larger Block Size: +
- Higher Associativity: +
- Victim Caches: +
- Pseudo-Associative Caches: +
- HW Prefetching of Instr/Data: +
- Compiler Controlled Prefetching: +
- Compiler Reduce Misses: +
- Priority to Read Misses: +
- Subblock Placement: +
- Early Restart & Critical Word 1st: +
- Non-Blocking Caches: +
- Second Level Caches: +
- Small & Simple Caches: -

### Miss Penalty

- Compiler Reduce Misses: 0
- Small & Simple Caches: 0

### Hit time

- Larger Block Size: +
- Higher Associativity: +
- Victim Caches: +
- Pseudo-Associative Caches: +
- HW Prefetching of Instr/Data: +
- Compiler Controlled Prefetching: +
- Compiler Reduce Misses: +
- Priority to Read Misses: +
- Subblock Placement: +
- Early Restart & Critical Word 1st: +
- Non-Blocking Caches: +
- Second Level Caches: +
- Small & Simple Caches: -
- Avoiding Address Translation: +
- Pipelining Writes: +