Advanced Computer Architecture

Course Goal:
Understanding important emerging design techniques, machine structures, technology factors, evaluation methods that will determine the form of high-performance programmable processors and computing systems in 21st Century.

Important Factors:
• Driving Force: Applications with diverse and increased computational demands even in mainstream computing (multimedia etc.)
• Techniques must be developed to overcome the major limitations of current computing systems to meet such demands:
  – ILP limitations, Memory latency, IO performance.
  – Increased branch penalty other stalls in deeply pipelined CPUs.
  – General-purpose processors as only homogeneous system computing resource.
• Enabling Technology for many possible solutions:
  – Increased density of VLSI logic (> one billion transistors in 2005)
  – Enables a high-level of system-level integration.
Course Topics

Topics we will cover include:

• Overcoming inherent ILP & clock scaling limitations by exploiting Thread-level Parallelism (TLP):
  – Support for Simultaneous Multithreading (SMT).
    • Alpha EV8. Intel P4 Xeon (aka Hyper-Threading), IBM Power5.
  – Chip Multiprocessors (CMPs):
    • The Hydra Project: An example CMP with Hardware Data/Thread Level Speculation (TLS) Support. IBM Power4, 5 ….

• Instruction Fetch Bandwidth/Memory Latency Reduction:
  – Conventional & Block-based Trace Cache (Intel P4).

• Advanced Dynamic Branch Prediction Techniques.

• Towards micro heterogeneous computing systems:
  – Vector processing. Vector Intelligent RAM (VIRAM).
  – Digital Signal Processing (DSP), Media Processors.
  – Graphics Processor Units (GPUs).
  – Re-Configurable Computing and Processors.

• Virtual Memory Design/Implementation Issues.

• High Performance Storage: Redundant Arrays of Disks (RAID).
Mainstream Computer System Components

Central Processing Unit (CPU): General Propose Processor (GPP)

1000MHZ - 3.8 GHz  (a multiple of system bus speed)
Pipelined (7 - 30 stages)
Superscalar (max ~ 4 instructions/cycle) single-threaded
Dynamically-Scheduled or VLIW
Dynamic and static branch prediction

Examples: Alpha, AMD K7: EV6, 400MHZ
Intel PII, PIII: GTL+ 133MHZ
Intel P4 800MHZ

Support for one or more CPUs

SDRAM
PC100/PC133
100-133MHz
64-128 bits wide
2-way interleaved
~ 900 MBYTES/SEC

Double Data Rate (DDR) SDRAM
PC3200
400MHZ (effective 200x2)
64-128 bits wide
4-way interleaved
~3.2 GBYTES/SEC
(second half 2002)

RAMbus DRAM (RDRAM)
PC800, PC1060
400-533MHz (DDR)
16-32 bits wide channel
~ 1.6 - 3.2 GBYTES/SEC
(per channel)

Caches

Front Side Bus (FSB)

Memory Controller

Memory

Controllers

I/O Devices:

Disks
Displays
Keyboards

NICs

Networks

I/O Buses

North Bridge

South Bridge

Chipset

Fast Ethernet
Gigabit Ethernet
ATM, Token Ring ..
Computing Engine Choices

• General Purpose Processors (GPPs): Intended for general purpose computing (desktops, servers, clusters..)

• Application-Specific Processors (ASPs): Processors with ISAs and architectural features tailored towards specific application domains
  – E.g Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs), Vector Processors...

• Co-Processors: A hardware (hardwired) implementation of specific algorithms with limited programming interface (augment GPPs or ASPs)

• Configurable Hardware:
  – Field Programmable Gate Arrays (FPGAs)
  – Configurable array of simple processing elements

• Application Specific Integrated Circuits (ASICs): A custom VLSI hardware solution for a specific computational task

• The choice of one or more depends on a number of factors including:
  – Type and complexity of computational algorithm (general purpose vs. Specialized)
  – Desired level of flexibility
  – Development cost
  – Power requirements
  – Performance requirements
  – System cost
  – Real-time constrains
Computing Engine Choices

Programmability / Flexibility

General Purpose Processors (GPPs):

Application-Specific Processors (ASPs):

E.g. Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs)

Configurable Hardware

Selection Factors:
- Type and complexity of computational algorithms (general purpose vs. Specialized)
- Desired level of flexibility
- Development cost
- Power requirements
- Performance
- System cost
- Real-time constrains

Specialization, Development cost/time
Performance/Chip Area/Watt (Computational Efficiency)

Processor = Programmable computing element that runs programs written using a pre-defined set of instructions
Computer System Components

Enhancing Computing Performance & Capabilities:

- Support for Simultaneous Multithreading (SMT): Intel HT.
- VLIW & intelligent compiler techniques: Intel/HP EPIC IA-64.
- More Advanced Branch Prediction Techniques.
- Chip Multiprocessors (CMPs): The Hydra Project. IBM Power 4,5
- Vector processing capability: Vector Intelligent RAM (VIRAM).
  Or Multimedia ISA extension.
- Digital Signal Processing (DSP) capability in system.
- Re-Configurable Computing hardware capability in system.

Memory Latency Reduction:
- Conventional & Block-based Trace Cache.
- Integrate Memory Controller & a portion of main memory with CPU: Intelligent RAM
- Integrated memory Controller:
  - AMD Opetron
  - IBM Power5

Recent Trend:
- More system components integration (lowers cost, improves system performance)
  System On Chip (SOC) approach

CPU

Caches

L1

L2

L3

Memory

controllers

Memory Bus

Front Side Bus (FSB)

Memory Controller

North Bridge

South Bridge

adapters

Controllers

NICs

Disks (RAID)

Displays

Keyboards

Networks

I/O Devices:

EECC722 - Shaaban

#6 Lec # 1 Fall 2006 9-4-2006
EECC551 Review

- Recent Trends in Computer Design.
- 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.
- Basic Virtual memory Issues
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).
    - Rate of clock speed improvement have decreased in recent years.
  - Architectural improvements in CPU design.
- Microprocessor-based systems directly reflect IC and architectural 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, Windows) 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.
- Increased importance of exploiting thread-level parallelism (TLP) in main-stream computing systems.
Mass-produced microprocessors are a cost-effective high-performance replacement for custom-designed mainframe/minicomputer CPUs.

Microprocessor: Single-chip VLSI-based processor
Microprocessor Performance 1987-97

Integer SPEC92 Performance

> 100x performance increase in the last decade
Microprocessor Transistor Count Growth Rate

Moore’s Law:
(circa 1970)
2X transistors/Chip
Every 1.5 years
Still valid today

Alpha 21264: 15 million
Pentium Pro: 5.5 million
PowerPC 620: 6.9 million
Alpha 21164: 9.3 million
Sparc Ultra: 5.2 million

> One billion in 2005

How to best exploit increased transistor count?
• Keep increasing cache capacity/levels?
• Multiple GPP cores?
• Integrate other types of computing elements?

~ 500,000x transistor density increase in the last 35 years
Microprocessor Frequency Trend

Result:
- Deeper Pipelines
  - Longer stalls
  - Higher CPI (lowers effective performance per cycle)

1. Frequency used to double each generation
2. Number of gates/clock reduce by 25%
3. Leads to deeper pipelines with more stages
   (e.g. Intel Pentium 4E has 30+ pipeline stages)

Possible Solutions?
- Exploit Thread-Level Parallelism (TLB) at the chip level (SMT/CMP)
- Utilize/integrate more-specialized computing elements other than GPPs

T = I x CPI x C
Parallelism in Microprocessor VLSI Generations

Bit-level parallelism

Instruction-level (ILP)

Thread-level (?)

Multiple micro-operations per cycle (multi-cycle non-pipelined)

Not Pipelined CPI >> 1

Single-issue Pipelined CPI =1

Superscalar /VLIW CPI <1

Simultaneous Multithreading SMT:
- e.g. Intel’s Hyper-threading

Chip-Multiprocessors (CMPs)
- e.g IBM Power 4, 5
- Intel Pentium D, Core Duo
- AMD Athlon 64 X2
- Dual Core Opteron
- Sun UltraSparc T1 (Niagara)

Crossbar (Matrix)

Even more important due to slowing clock rate increase

Chip-Level Parallel Processing

Thread-Level Parallelism (TLP)

Single Thread

Improving microprocessor generation performance by exploiting more levels of parallelism
Microprocessor Architecture Trends

General Purpose Processor (GPP)

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

"Superinstructions" grouped together
decreased HW control complexity
(Single or Multi-Threaded)

VLIW

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

SIMULTANEOUS MULTITHREADING (SMT)
multiple HW contexts (regs, PC, SP)
each cycle, any context may execute
e.g. Intel’s HyperThreading (P4)

SMT/CMPs
(e.g. IBM Power5, AMD X2, Intel Core 2)

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

"Superinstructions" grouped together
decreased HW control complexity
(Single or Multi-Threaded)

VLIW

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

SIMULTANEOUS MULTITHREADING (SMT)
multiple HW contexts (regs, PC, SP)
each cycle, any context may execute
e.g. Intel’s HyperThreading (P4)

SMT/CMPs
(e.g. IBM Power5, AMD X2, Intel Core 2)
Computer Technology Trends: 
*Evolutionary but Rapid Change*

- **Processor:**
  - 1.5-1.6 performance improvement every year; Over 100X performance in last decade.

- **Memory:**
  - DRAM capacity: > 2x every 1.5 years; 1000X size in last decade.
  - Cost per bit: Improves about 25% or more per year.
  - Only 15-25% performance improvement 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 2006:**
  - Processor clock speed: ~ 3000 MegaHertz (3 Giga Hertz)
  - Memory capacity: > 4000 MegaByte (4 Giga Bytes)
  - Disk capacity: > 500 GigaBytes (0.5 Tera Bytes)

Performance gap compared to CPU performance causes system performance bottlenecks

With 2-4 processor cores on a single chip
Architectural Improvements

• Increased optimization, utilization and size of cache systems with multiple levels (currently the most popular approach to utilize the increased number of available transistors).

• Memory-latency hiding techniques. Including Simultaneous Multithreading (SMT)

• 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 system and memory bus designs to improve data transfer rates and reduce latency.

• Increased exploitation of Thread-Level Parallelism in terms of Simultaneous Multithreading (SMT) and Chip Multiprocessors (CMPs)
Current Computer Architecture Topics

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

Emerging Technologies
- Interleaving
- Bus protocols

Memory Hierarchy
- DRAM
- L2 Cache
- L1 Cache

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)

VLSI
- Addressing, Protection, Exception Handling

Memory Hierarchy
- Coherence, Bandwidth, Latency

VLSI
- Optimizing Memory Hierarchy

ECC722 - Shaaban
Metrics of Computer Performance
(Measures)

Application
Programming Language
Compiler

ISA

Datapath
Control
Function Units
Transistors Wires Pins

Execution time: Target workload, SPEC95, SPEC2000, etc.

(millions) of Instructions per second – MIPS
(millions) of (F.P.) operations per second – MFLOP/s

Megabytes per second.

Cycles per second (clock rate).

Each metric has a purpose, and each can be misused.
CPU Execution Time: The CPU Equation

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

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

- 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:

  $$T = I \times CPI \times C$$

  (This equation is commonly known as the CPU performance equation)

  Or Instructions Per Cycle (IPC):
  $$IPC = 1/CPI$$

#19 Lec #1 Fall 2006 9-4-2006
Factors Affecting CPU Performance

CPU time = \( \frac{\text{Seconds}}{\text{Program}} = \text{Instructions} \times \text{Cycles} \times \text{Seconds} \)

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

\( T = I \times CPI \times C \)
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:

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

  Hence speedup is given by:

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

  F (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied.
Pictorial Depiction of Amdahl’s Law

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

Before:
Execution Time without enhancement E: (Before enhancement is applied)
  • shown normalized to 1 = (1-F) + F = 1

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

After:
Execution Time with enhancement E:

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

EECC722 - Shaaban
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:

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

Note: All fractions $F_i$ refer to original execution time before the enhancements are applied.
Amdahl's Law With Multiple Enhancements: Example

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

  Speedup_1 = S_1 = 10  Percentage_1 = F_1 = 20%
  Speedup_2 = S_2 = 15  Percentage_2 = F_2 = 15%
  Speedup_3 = S_3 = 30  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?

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

- \[
\text{Speedup} = 1 / \left[\left(1 - .2 - .15 - .1\right) + .2/10 + .15/15 + .1/30\right]
\]

\[
= 1 / \left[.55 + .0333\right]
\]

\[
= 1 / .5833 = 1.71
\]
Pictorial Depiction of Example

Before:
Execution Time with no enhancements: 1

<table>
<thead>
<tr>
<th>Unaffected, fraction: .55</th>
<th>F1 = .2</th>
<th>F2 = .15</th>
<th>F3 = .1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/10</td>
<td>/15</td>
<td>/30</td>
</tr>
</tbody>
</table>

Unchanged

Unaffected, fraction: .55

After:
Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

Note: All fractions (F_i, i = 1, 2, 3) refer to original execution time.

What if the fractions given are after the enhancements were applied? How would you solve the problem?
Instruction Pipelining Review

- Instruction pipelining is CPU implementation technique where multiple operations on a number of instructions are overlapped.
  - Instruction pipelining exploits Instruction-Level Parallelism (ILP)
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called a pipeline stage or a pipeline segment.
- The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline -- instructions enter at one end and progress through the stages and exit at the other end.
- The time to move an instruction one step down the pipeline is equal to the machine cycle and is determined by the stage with the longest processing delay.
- Pipelining increases the CPU instruction throughput: The number of instructions completed per cycle.
  - Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or $\text{ideal CPI} = 1$
- Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).
  - Minimum instruction latency = $n$ cycles, where $n$ is the number of pipeline stages
MIPS In-Order Single-Issue Integer Pipeline

**Ideal Operation** (No stall cycles)

Fill Cycles = number of stages -1

<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>

**MIPS 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

4 cycles = n -1

Time to fill the pipeline

Ideal CPI = 1

(or IPC = 1)

In-order = instructions executed in original program order

Ideal pipeline operation without any stall cycles

n= 5 pipeline stages
A Pipelined MIPS Datapath

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

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

Branch Penalty = 4 - 1 = 3 cycles
Pipeline Hazards

- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.

- Hazards reduce the ideal speedup (increase CPI > 1) 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 result 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.
One shared memory for instructions and data

MIPS with Memory Unit Structural Hazards

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

One shared memory for instructions and data

The structural hazard causes pipeline bubbles to be inserted.

\[ CPI = 1 + \text{fraction of loads and stores} \times 1 \]
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 may require one or more instructions to be stalled to ensure correct execution.

- Example:

  1. DADD R1, R2, R3
  2. DSUB R4, R1, R5
  3. AND R6, R1, R7
  4. OR R8, R1, R9
  5. XOR R10, R1, R11

  Arrows represent data dependencies between instructions.

  Instructions that have no dependencies among them are said to be parallel or independent.

  A high degree of Instruction-Level Parallelism (ILP) is present in a given code sequence if it has a large number of parallel instructions.

  - All the instructions after DADD use the result of the DADD instruction.
  - DSUB, AND instructions need to be stalled for correct execution.

  i.e Correct operand data not ready yet when needed in EX cycle.
Figure A.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Two stall cycles are needed here.
Minimizing Data hazard Stalls by Forwarding

• Data 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 MIPS integer 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.
A set of instructions that depend on the DADD result uses forwarding paths to avoid the data hazard.
Data Hazard/Dependence Classification

True Data Dependence

I (Write)  
\[\text{Shared Operand}\]

J (Read)  
\[\text{Program Order}\]

Read after Write (RAW)  
if data dependence is violated

I (Write)  
\[\text{Shared Operand}\]

J (Write)  
\[\text{Shared Operand}\]

Write after Write (WAW)  
if output dependence is violated

A name dependence: output dependence

A name dependence: antidependence

I (Read)  
\[\text{Shared Operand}\]

J (Write)  
\[\text{Shared Operand}\]

Write after Read (WAR)  
if antidependence is violated

I (Read)  
\[\text{Shared Operand}\]

J (Read)  
\[\text{Shared Operand}\]

Read after Read (RAR)  
not a hazard

No dependence
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 (branch is resolved).
  - Otherwise the PC may not be correct when needed in IF.
- In current MIPS pipeline, the conditional branch is resolved in stage 4 (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 instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></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>WB</td>
</tr>
<tr>
<td>Branch successor + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Correct PC available</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming we stall or flush the pipeline on a branch instruction:
Three clock cycles are wasted for every branch for current MIPS pipeline.

Branch Penalty = stage number where branch is resolved - 1
here Branch Penalty = 4 - 1 = 3 Cycles

i.e Correct PC is not available when needed in IF.
Pipeline Performance Example

• Assume the following MIPS instruction mix:

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

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

• \( \text{CPI} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \)

\[
\begin{align*}
\text{CPI} & = 1 + \text{stalls by loads} + \text{stalls by branches} \\
& = 1 + 0.3 \times 0.25 \times 1 + 0.2 \times 0.45 \times 1 \\
& = 1 + 0.075 + 0.09 \\
& = 1.165
\end{align*}
\]
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 (assuming ideal memory):

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

- 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).
Basic Blocks/Dynamic Execution Sequence (Trace) Example

Static Program Order

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
</table>

- A-O = Basic Blocks terminating with conditional branches
- The outcomes of branches determine the basic block dynamic execution sequence or trace

Trace: Sequence of basic blocks executed

Program Control Flow Graph (CFG)

If all three branches are taken the execution trace will be basic blocks: ACGO

Average Basic Block Size = 5-7 instructions

Type of branches in example above: “If Then …. Else” Branches
Increasing Instruction-Level Parallelism (ILP)

- 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. This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler to eliminate more stall cycles.
- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

```c
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.
MIPS Loop Unrolling Example

For the loop:

\[
\text{for (i=1000; i>0; i=i-1) } \\
x[i] = x[i] + s;
\]

The straightforward MIPS assembly code is given by:

\[
\begin{align*}
\text{Loop: } & \quad \text{L.D } F0, 0 (R1) \quad ;F0=\text{array element} \\
& \quad \text{ADD.D } F4, F0, F2 \quad ;\text{add scalar in F2 (constant)} \\
& \quad \text{S.D } F4, 0(R1) \quad ;\text{store result} \\
& \quad \text{DADDUI } R1, R1, \# -8 \quad ;\text{decrement pointer 8 bytes} \\
& \quad \text{BNE } R1, R2,\text{Loop} \quad ;\text{branch R1!=R2}
\end{align*}
\]

R1 is initially the address of the element with highest address.  
8(R2) is the address of the last element to operate on.

X[ ] array of double-precision floating-point numbers (8-bytes each)
MIPS 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>

Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used
**Loop Unrolling Example (continued)**

- This loop code is executed on the MIPS pipeline as follows:
  
  (Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used)

### No scheduling

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td>2</td>
<td>stall</td>
</tr>
<tr>
<td>3</td>
<td>ADD.D F4, F0, F2</td>
</tr>
<tr>
<td>4</td>
<td>stall</td>
</tr>
<tr>
<td>5</td>
<td>stall</td>
</tr>
<tr>
<td>6</td>
<td>S.D F4, 0(R1)</td>
</tr>
<tr>
<td>7</td>
<td>DADDUI R1, R1, # -8</td>
</tr>
<tr>
<td>8</td>
<td>stall</td>
</tr>
<tr>
<td>9</td>
<td>BNE R1, R2, Loop</td>
</tr>
<tr>
<td>10</td>
<td>stall</td>
</tr>
</tbody>
</table>

- **10 cycles per iteration**

- **6 cycles per iteration**

\[
\frac{10}{6} = 1.7 \text{ times faster}
\]

- Ignoring Pipeline Fill Cycles
- No Structural Hazards

(Chapter 4.1)
### Loop Unrolling Example (continued)

#### Loop: 1

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0, 0(R1)</td>
<td>Stall</td>
</tr>
<tr>
<td>2</td>
<td>ADD.D F4, F0, F2</td>
<td>Stall</td>
</tr>
<tr>
<td>3</td>
<td>SD F4, 0(R1)</td>
<td>; drop DADDUI &amp; BNE</td>
</tr>
<tr>
<td>4</td>
<td>LD F6, -8(R1)</td>
<td>Stall</td>
</tr>
<tr>
<td>5</td>
<td>ADDD F8, F6, F2</td>
<td>Stall</td>
</tr>
<tr>
<td>6</td>
<td>SD F8, -8(R1),</td>
<td>; drop DADDUI &amp; BNE</td>
</tr>
<tr>
<td>7</td>
<td>LD F10, -16(R1)</td>
<td>Stall</td>
</tr>
<tr>
<td>8</td>
<td>ADDD F12, F10, F2</td>
<td>Stall</td>
</tr>
<tr>
<td>9</td>
<td>SD F12, -16(R1)</td>
<td>; drop DADDUI &amp; BNE</td>
</tr>
<tr>
<td>10</td>
<td>LD F14, -24(R1)</td>
<td>Stall</td>
</tr>
<tr>
<td>11</td>
<td>ADDD F16, F14, F2</td>
<td>Stall</td>
</tr>
<tr>
<td>12</td>
<td>SD F16, -24(R1)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>DADDUI R1, R1, # -32</td>
<td>Stall</td>
</tr>
<tr>
<td>14</td>
<td>BNE R1, R2, Loop</td>
<td>Stall</td>
</tr>
</tbody>
</table>

#### Loop unrolled 4 times

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers.
- The size of the basic block increased from 5 instructions in the original loop to 14 instructions.

**No scheduling**

- Three branches and three decrements of R1 are eliminated.
- Load and store addresses are changed to allow DADDUI instructions to be merged.

The unrolled loop runs in 28 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycle, or 28/4 = 7 cycles to produce each of the four elements.

---

**Register Renaming**

Note use of different registers for each iteration (register renaming)
Loop Unrolling Example (continued)

When scheduled for pipeline

Loop:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F6, -8  (R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F10, -16(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F14, -24(R1)</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8, F6, F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12, F10, F2</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16, F14, F2</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F8, -8(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1,# -32</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F12, 16(R1),F12</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2, Loop</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F16, 8(R1), F16</td>
<td>;8-32 = -24</td>
</tr>
</tbody>
</table>

The execution time of the loop has dropped to 14 cycles, or $14/4 = 3.5$ clock cycles per element compared to 7 before scheduling and 6 when scheduled but unrolled. 

Speedup = $6/3.5 = 1.7$

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

Note: No stalls

Larger Basic Block → More ILP

Exposed
Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent (parallel).

  e.g. in
  \[
  \text{for (i=1; i<=1000; i++)}
  \]
  \[
  x[i] = x[i] + s;
  \]
  Usually: Data Parallelism $\rightarrow$ LLP

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.

$\Rightarrow$ Thus loop iterations are parallel (or independent from each other).

- Loop-carried Data Dependence: A data dependence between different loop iterations (data produced in an earlier iteration used in a later one).
- Not Loop-carried Data Dependence: Data dependence within the same loop iteration.
- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent (and in vector processing).
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used in the loop.
  - Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.

(In Chapter 4.4)
LLP Analysis Example 1

- 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 */
```

(Where A, B, C are distinct non-overlapping arrays)

- **S2** uses the value `A[i+1]`, computed by **S1** in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).
  - **S1** uses a value computed by **S1** in the earlier iteration, since iteration `i` computes `A[i+1]` read in iteration `i+1` (loop-carried dependence, prevents parallelism). The same applies for **S2** for `B[i]` and `B[i+1]`

⇒ These two data dependencies are loop-carried spanning more than one iteration (two iterations) preventing loop parallelism.
LLP Analysis Example 2

• 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 */
}
```

– **S1** uses the value **B[i]** computed by **S2** in the previous iteration (loop-carried dependence)

– This dependence is not circular:
  - **S1** depends on **S2** but **S2** does not depend on **S1**.
  - 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]; /* Loop Completion code
```

---

**Dependency Graph**

- **S1**
- **S2**

**Iteration #** → **i**
- **B** \(_{i+1}\)

**Loop-carried Dependence**

\[ i \rightarrow i+1 \]

- **S2 → S1** on **B[i]**  
  - Loop-carried dependence

---

**Parallel loop iterations**  
(data parallelism in computation exposed in loop code)

---

**Data**

- **A[i]**
- **B[i]**
- **C[i]**
- **D[i]**
LLP Analysis Example 2

Original 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 */
}
```  

```c
```

```
B[100] = C[99] + D[99];
B[101] = C[100] + D[100];
```

Modified Parallel Loop:

(One less iteration)

```
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];
```  

Loop Start-up code

```
```

```
B[100] = C[99] + D[99];
B[101] = C[100] + D[100];
```  

Not Loop Carried Dependence

Loop-carried Dependence

Loop Completion code
Reduction of Data Hazards Stalls with Dynamic Scheduling

- So far we have dealt with data hazards in instruction pipelines by:
  - Result forwarding (register bypassing) to reduce or eliminate stalls needed to prevent RAW hazards as a result 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 hazard-prevention stalls in scheduled code.
    - Loop unrolling to increase basic block size: More ILP exposed.

- Dynamic scheduling: (out-of-order execution)
  - Uses a hardware-based mechanism to reorder or rearrange instruction execution order to reduce stalls dynamically at runtime.
    - Better dynamic exploitation of instruction-level parallelism (ILP).
  - Enables handling some cases where instruction dependencies are unknown at compile time (ambiguous dependencies).
  - Similar to the other pipeline optimizations above, a dynamically scheduled processor cannot remove true data dependencies, but tries to avoid or reduce stalling.

Why?

i.e. Start of instruction execution is not in program order

(In Appendix A.8, Chapter 3.2, 3.3)
Dynamic Pipeline Scheduling: *The Concept*

(Out-of-order execution) i.e Start of instruction execution is not in program order

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

- Instruction are allowed to start executing out-of-order as soon as their operands are available.
  - Better dynamic exploitation of instruction-level parallelism (ILP).

**Example:**

In the case of in-order pipelined execution

SUB.D must wait for DIV.D to complete which stalled ADD.D before starting execution

In out-of-order execution SUB.D 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.

(In Appendix A.8, Chapter 3.2) Order = Program Instruction Order

EECC722 - Shaaban
Dynamic Scheduling: The Tomasulo Algorithm

- Developed at IBM and first implemented in IBM’s 360/91 mainframe 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:

  RISC CPUs: Alpha 21264, HP 8600, MIPS R12000, PowerPC G4
  RISC-core x86 CPUs: AMD Athlon, Pentium III, 4, Xeon ….
Dynamic Scheduling: The Tomasulo Approach

The basic structure of a MIPS floating-point unit using Tomasulo’s algorithm

(In Chapter 3.2) Pipelined FP units are used here
Three Stages of Tomasulo Algorithm

1 **Issue:** Get instruction from pending Instruction Queue (IQ).
   - Instruction issued to a free reservation station (RS) (no structural hazard).
   - Selected RS is marked busy.
   - Control sends available instruction operands values (from ISA registers) to assigned RS.
   - Operands not available yet are renamed to RSs that will produce the operand (register renaming). (Dynamic construction of data dependency graph)

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 (forwarding done via CDB). (i.e. wait on any remaining operands, no RAW)

3 **Write result (WB):** Finish execution.
   - Write result on Common Data Bus (CDB) to all awaiting units (RSs)
   - 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 data to waiting RS if source matches expected RS (that produces result).
  - Does the result forwarding via broadcast to waiting RSs.

(In Chapter 3.2)
Dynamic Conditional Branch Prediction

• Dynamic branch prediction schemes are different from static mechanisms because they utilize hardware-based mechanisms that use the run-time behavior of branches to make more accurate predictions than possible using static prediction.

• Usually information about outcomes of previous occurrences of branches (branching history) is used to dynamically predict the outcome of the current branch. Some of the proposed dynamic branch prediction mechanisms include:
  – **One-level or Bimodal:** Uses a Branch History Table (BHT), a table of usually two-bit saturating counters which is indexed by a portion of the branch address (low bits of address). (First proposed mid 1980s)
  – **Two-Level Adaptive Branch Prediction.** (First proposed early 1990s),
  – MCFarling’s Two-Level Prediction with index sharing (gshare, 1993).
  – **Hybrid or Tournament Predictors:** Uses a combinations of two or more (usually two) branch prediction mechanisms (1993).

• To reduce the stall cycles resulting from correctly predicted taken branches to zero cycles, a Branch Target Buffer (BTB) that includes the addresses of conditional branches that were taken along with their targets is added to the fetch stage.

(Dynamic Branch Prediction in Chapter 3.4, 3.5)
Branch Target Buffer (BTB)

- Effective branch prediction requires the target of the branch at an early pipeline stage. (resolve the branch early in the pipeline)
- One can use additional adders to calculate the target, as soon as the branch instruction is decoded. This would mean that one has to wait until the ID stage before the target of the branch can be fetched, taken branches would be fetched with a one-cycle penalty (this was done in the enhanced MIPS pipeline Fig A.24).
- To avoid this problem one can use a Branch Target Buffer (BTB). A typical BTB is an associative memory where the addresses of taken branch instructions are stored together with their target addresses.
- Some designs store n prediction bits as well, implementing a combined BTB and Branch history Table (BHT).
- Instructions are fetched from the target stored in the BTB in case the branch is predicted-taken and found in BTB. After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created once it is resolved as taken.
- Branch Target Instruction Cache (BTIC): A variation of BTB which caches also the code of the branch target instruction in addition to its address. This eliminates the need to fetch the target instruction from the instruction cache or from memory.

Goal of BTB: Zero stall taken branches
Basic Branch Target Buffer (BTB)

Goal of BTB: Zero stall taken branches

Fetch instruction from instruction memory (I-L1 Cache)

Instruction Fetch

IF

BTB is accessed in Instruction Fetch (IF) cycle

A branch-target buffer.

0 = NT = Not Taken
1 = T = Taken

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 Targets

Predicted PC

Branch Address

Branch Target if predicted taken

Number of entries in branch-target buffer

PC of instruction to fetch

Look up
One-Level (Bimodal) Branch Predictors

- One-level or bimodal branch prediction uses only one level of branch history.
- These mechanisms usually employ a table which is indexed by lower N bits of the branch address. Pattern History Table (PHT)
- Each table entry (or predictor) consists of \( n \) history bits, which form an \( n \)-bit automaton or saturating counters.
- Smith proposed such a scheme, known as the Smith Algorithm, that uses a table of two-bit saturating counters. (1985)
- One rarely finds the use of more than 3 history bits in the literature.
- Two variations of this mechanism:
  - Pattern History Table: Consists of directly mapped entries.
  - Branch History Table (BHT): Stores the branch address as a tag. It is associative and enables one to identify the branch instruction during IF by comparing the address of an instruction with the stored branch addresses in the table (similar to BTB).
One-Level Bimodal Branch Predictors

Pattern History Table (PHT)

Sometimes referred to as
Decode History Table (DHT)
or
Branch History Table (BHT)

N Low Bits of Branch Address

Table has $2^N$ entries
(also called predictors).

Example:

For $N = 12$
Table has $2^N = 2^{12}$ entries
= 4096 = 4k entries

Number of bits needed = $2 \times 4k = 8k$ bits

What if different branches map to the same predictor (counter)?
This is called branch address aliasing and leads to interference with current branch prediction by other branches and may lower branch prediction accuracy for programs with aliasing.

High bit determines branch prediction
0 = NT = Not Taken
1 = T = Taken

2-bit saturating counters (predictors)

Update counter after branch is resolved:
- Increment counter used if branch is taken
- Decrement counter used if branch is not taken

EECC722 - Shaaban
Prediction Accuracy of A 4096-Entry Basic One-Level Dynamic Two-Bit Branch Predictor

Specifications:
- \( N = 12 \), \( 2^N = 4096 \)

**FP**
- nasa7: 1%
- matrix300: 0%
- tomcatv: 1%
- doduc: 5%
- spice: 9%
- fppp: 9%
- gcc: 12%
- espresso: 5%
- eqntott: 18%
- li: 10%

**Integer**
- Misprediction Rate: (Lower misprediction rate due to more loops)
  - Integer average 11%
  - FP average 4%

Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.

Has, more branches involved in IF-Then-Else constructs than FP
Correlating Branches

Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch.

**Example:**

<table>
<thead>
<tr>
<th>Branch</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>if (aa==2)</td>
<td>DSUBUI R3, R1, #2 ; R3 = R1 - 2</td>
</tr>
<tr>
<td></td>
<td>aa=0; (not taken)</td>
<td>BNEZ R3, L1 ; B1 (aa!=2)</td>
</tr>
<tr>
<td>B2</td>
<td>if (bb==2)</td>
<td>DSUBUI R3, R2, #2 ; R3 = R2 - 2</td>
</tr>
<tr>
<td></td>
<td>bb=0; (not taken)</td>
<td>BNEZ R3, L2 ; B2 (bb!=2)</td>
</tr>
<tr>
<td>B3</td>
<td>if (aa!==bb)</td>
<td>DSUBUI R3, R1, R2 ; R3=aa-bb</td>
</tr>
</tbody>
</table>

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 GAp Branch Predictors

- Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches using two levels of branch history.
- Uses two levels of branch history:

1. **First level (global):**
   - Record the global pattern or history of the $m$ most recently executed branches as taken or not taken. Usually an $m$-bit shift register.

2. **Second level (per branch address):**
   - $2^m$ prediction tables, each table entry has $n$ bit saturating counter.
   - The branch history pattern from first level is used to select the proper branch prediction table in the second level.
   - The low $N$ bits of the branch address are used to select the correct prediction entry (predictor) within a the selected table, thus each of the $2^m$ tables has $2^N$ entries and each entry is 2 bits counter.
   - Total number of bits needed for second level = $2^m \times n \times 2^N$ bits
   - In general, the notation: **GAp** $(m,n)$ predictor means:
     - Record last $m$ branches to select between $2^m$ history tables.
     - Each second level table uses $n$-bit counters (each table entry has $n$ bits).
   - Basic two-bit single-level Bimodal BHT is then a $(0,2)$ predictor.
Organization of A Correlating Two-level GAp (2,2) Branch Predictor

First Level
Branch History Register (BHR)
(2 bit shift register)

Second Level
Pattern History Tables (PHTs)

High bit determines branch prediction
0 = Not Taken
1 = Taken

(m = # of branches tracked in first level = 2)
Thus $2^m = 2^2 = 4$ tables in second level

(N = # of low bits of branch address used = 4)
Thus each table in 2nd level has $2N = 24 = 16$ entries

(n = # number of bits of 2nd level table entry = 2)
Number of bits for 2nd level = $2^m \times n \times 2^N$
$= 4 \times 2 \times 16 = 128$ bits

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

Basic
Single (one) Level

Correlating
Two-level
Gap (2, 2)

SPEC89 benchmarks

N = 12

N = 10

4096 entries: 2 bits per entry
Unlimited entries: 2 bits per entry
1024 entries (2,2)

FP

Integer
Multiple Instruction Issue: CPI < 1

- To improve a pipeline’s CPI to be better [less] than one, and to better exploit Instruction Level Parallelism (ILP), a number of instructions have to be issued in the same 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 -more commonly- dynamically (Tomasulo).
    - PowerPC, Sun UltraSparc, Alpha, HP 8000, Intel PII, III, 4 ...
  - **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).
    - Example: Explicitly Parallel Instruction Computer (EPIC)
      - Originally a joint HP/Intel effort.
      - ISA: Intel Architecture-64 (IA-64) 64-bit address:

- Limitations of the approaches:
  - Available ILP in the program (both).
  - Specific hardware implementation difficulties (superscalar).
  - VLIW optimal compiler design issues.

Most common = 4 instructions/cycle called 4-way superscalar processor

CPI < 1 or Instructions Per Cycle (IPC) > 1

Chapter 3.6, 3.7, 4.3, 4.5
Two instructions can be issued per cycle (\textit{static two-issue or 2-way superscalar}).
One of the instructions is integer (including load/store, branch). The other instruction is a floating-point operation.
   - This restriction reduces the complexity of hazard checking.
Hardware must fetch and decode two instructions per cycle.
Then it determines whether zero (a stall), one or two instructions can be issued (in decode stage) per cycle.

<table>
<thead>
<tr>
<th>Instruction Type</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>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-issue statically scheduled pipeline in operation
FP instructions assumed to be adds (EX takes 3 cycles)

(Ch. 3.6) Ideal CPI = 0.5 Ideal Instructions Per Cycle (IPC) = 2

Instructions assumed independent (no stalls)
Intel IA-64: VLIW “Explicitly Parallel Instruction Computing (EPIC)”

- Three 41-bit instructions in 128 bit “Groups” or bundles; an instruction bundle template field (5-bits) determines if instructions are dependent or independent and statically specifies the functional units to used by the instructions:
  - 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

- 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).
- **Itanium**: Name of the first implementation (2001).
Intel/HP EPIC VLIW Approach

original source code

Sequential Code

Expose Instruction Parallelism (dependency analysis)

compiler

Dependency Graph

Optimize

Instruction Dependency Analysis

Exploit Instruction Parallelism: Generate VLIWs

128-bit bundle

Instruction 2
41 bits

Instruction 1
41 bits

Instruction 0
41 bits

Template
5 bits

Template field has static assignment/scheduling information
Unrolled Loop Example for Scalar (single-issue) Pipeline

1 Loop: L.D  F0,0(R1)
2       L.D  F6,-8(R1)
3       L.D  F10,-16(R1)
4       L.D  F14,-24(R1)
5  ADD.D  F4,F0,F2
6  ADD.D  F8,F6,F2
7  ADD.D  F12,F10,F2
8  ADD.D  F16,F14,F2
9  S.D   F4,0(R1)
10  S.D  F8,-8(R1)
11  DADDUI R1,R1,#-32
12  S.D   F12,16(R1)
13  BNE   R1,R2,LOOP
14  S.D   F16,8(R1) ; 8-32 = -24

Latency:
L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

Unrolled and scheduled loop from loop unrolling example

14 clock cycles, or 3.5 per original iteration (result) (unrolled four times)

Recall that loop unrolling exposes more ILP by increasing size of resulting basic block

No stalls in code above: CPI = 1 (ignoring initial pipeline fill cycles)
Loop Unrolling in 2-way 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>L.D F0,0(R1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDUI R1,R1,#-40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F16,-24(R1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R1,R2,LOOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays and expose more ILP (unrolled one more time)
- 12 cycles, or 12/5 = 2.4 cycles per iteration (3.5/2.4 = 1.5X faster than scalar)
- CPI = 12/17 = .7 worse than ideal CPI = .5 because 7 issue slots are wasted

Recall that loop unrolling exposes more ILP by increasing basic block size
# 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>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td></td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
<td>7</td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td>BNE R1,R2,LOOP</td>
<td>8</td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays and expose more ILP
7 results in 9 cycles, or 1.3 cycles per iteration
(2.4/1.3 = 1.8X faster than 2-issue superscalar, 3.5/1.3 = 2.7X faster than scalar)
Average: about 23/9 = 2.55 IPC (instructions per clock cycle) Ideal IPC = 5,
CPI = .39  Ideal CPI = .2 thus about 50% efficiency, 22 issue slots are wasted
Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)

(In chapter 4.3 pages 317-318)
Superscalar Architecture Limitations:

Issue Slot Waste Classification

- Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:
  - **Vertical waste** is introduced when the processor issues no instructions in a cycle.
  - **Horizontal waste** occurs when not all issue slots can be filled in a cycle.

Example:

<table>
<thead>
<tr>
<th>Issue Slot</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- Full issue slot
- Empty issue slot

Horizontal waste = 9 slots
Vertical waste = 12 slots

Instructions Per Cycle = IPC = 1/CPI

Also applies to VLIW

Result of issue slot waste: Actual Performance << Peak Performance
Conditional Instructions and Speculation

• 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.

• Techniques to further reduce the impact of branches on performance:
  – **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 (e.g. canceling branch delay instruction).
  – **Speculation**: An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls (i.e. branch not resolved yet):
    • **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** (Ch. 3.7)
      – Uses dynamic branch-prediction to guide the speculation process.
      – Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction. (Speculative Tomasulo)

ISA Support Needed

ISA/Compiler Support Needed

No ISA or Compiler Support Needed

EECC722 - Shaaban
Dynamic Hardware-Based Speculation

(Speculative Execution Processors, Speculative Tomasulo)

- Dynamic hardware-based branch prediction
- Dynamic Scheduling: issue multiple instructions in order and execute out of order. (Tomasulo)

- 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 ISA/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).

Why?
- Precise exceptions are possible since instructions must commit in order.

How?
- Addition of commit (retire, completion, or re-ordering) stage and forcing instructions to commit in their order in the code (i.e. to write results to registers or memory in program order).

(Ch. 3.7)
Hardware-Based Speculation

Speculative Execution + Tomasulo’s Algorithm

= Speculative Tomasulo

Speculative Tomasulo-based Processor
Four Steps of Speculative Tomasulo Algorithm

1. **Issue** — (In-order) Get an instruction from Instruction 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** — (out-of-order) 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** — (out-of-order) Finish execution (WB)
   
   Write on Common Data Bus (CDB) to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit** — (In-order) 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 (cancels speculated instructions after the branch)

   ⇒ Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.
Memory Hierarchy: 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, faster memory levels closer to the CPU: registers, then primary Cache Level (L1), then additional secondary cache levels (L2, L3…), then main memory, then mass storage (virtual memory).

- Each level of the hierarchy is usually a subset of the level below: data found in a level is also found in the level below (farther from CPU) but at lower speed (longer access time).

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

- 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.

(Chapter 5.1-5.3)
Memory Hierarchy: Motivation
Processor-Memory (DRAM) Performance Gap

(i.e. Gap between memory access time and CPU cycle time)

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

Ideal Memory Access Time = 1 CPU Cycle
Real Memory Access Time >> 1 CPU cycle
Basic 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, FIFO.

• 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 frames divide cache designs into three organizations:

1. **Direct mapped cache**: A block can be placed in only one location (cache block frame), given by the mapping function:
   \[ \text{index} = (\text{Block address}) \mod (\text{Number of blocks in cache}) \]
   Least complex to implement

2. **Fully associative cache**: A block can be placed anywhere in cache. (no mapping function).
   Most complex cache organization to implement

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:
   \[ \text{index} = (\text{Block address}) \mod (\text{Number of sets in cache}) \]
   Most common cache organization

If there are \( n \) blocks in a set the cache placement is called \( n \)-way set-associative.
Address Field Sizes/Mapping

Physical Memory Address Generated by CPU
(size determined by amount of physical main memory cacheable)

Block Address

Tag

Index

Block Offset

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

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

Tag size = address size - index size - offset size

Mapping function:

Cache set or block frame number = Index =

= (Block Address) MOD (Number of Sets)

No index/mapping function for fully associative cache
4K Four-Way Set Associative Cache: MIPS Implementation Example

1024 block frames
Each block = one word
4-way set associative
1024 / 4 = $2^8$ = 256 sets

Can cache up to
$2^{32}$ bytes = 4 GB of memory

Set associative cache requires parallel tag matching and more complex hit logic which may increase hit time

Mapping Function:
Cache Set Number = index = (Block address) MOD (256)

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay
Memory Hierarchy Performance:
Average Memory Access Time (AMAT), Memory Stall cycles

- **The Average Memory Access Time (AMAT):** The number of cycles required to complete an average memory access request by the CPU.
- **Memory stall cycles per memory access:** The number of stall cycles added to CPU execution cycles for one memory access.
- **Memory stall cycles per average memory access = (AMAT -1)**
- **For ideal memory:** AMAT = 1 cycle, this results in zero memory stall cycles.
- **Memory stall cycles per average instruction =**
  \[
  \text{Number of memory accesses per instruction} \times \text{Memory stall cycles per average memory access} = (1 + \text{fraction of loads/stores}) \times (\text{AMAT} -1)
  \]

\[
\text{Base CPI} = \text{CPI}_{\text{execution}} = \text{CPI with ideal memory}
\]

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}
\]
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 or dirty 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 or modified bit, is used to indicate whether the block was modified while in cache; if not the block is not written back to main memory when replaced.
- Advantage: 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 missed cache block is loaded into cache 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 (written or updated) into cache.

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

- Write back caches always 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.
Memory Access Tree, Unified L₁
Write Through, No Write Allocate, No Write Buffer

CPU Memory Access

L₁
Read
% reads x H₁
L₁ Read Hit:
Hit Access Time = 1
Stalls = 0
Ideal access on a hit

% reads x (1 - H₁)
L₁ Read Miss:
Access Time = M + 1
Stalls Per access = M
Stalls = % reads x (1 - H₁) x M

Write
% write x H₁
L₁ Write Hit:
Access Time: M + 1
Stalls Per access = M
Stalls = % write x (H₁) x M

% write x (1 - H₁)
L₁ Write Miss:
Access Time: M + 1
Stalls per access = M
Stalls = % write x (1 - H₁) x M

Stall Cycles Per Memory Access = % reads x (1 - H₁) x M + % write x M

AMAT = 1 + % reads x (1 - H₁) x M + % write x M
CPI = CPI_{execution} + (1 + fraction of loads/stores) x Stall Cycles per access
Stall Cycles per access = AMAT - 1

M = Miss Penalty
H₁ = Level 1 Hit Rate
1 - H₁ = Level 1 Miss Rate

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H₁ = Level 1 Hit Rate
1 - H₁ = Level 1 Miss Rate

Exercise:
Create memory access tree for split level 1
Memory Access Tree Unified L₁
Write Back, With Write Allocate

**CPU Memory Access**

- **L₁ Hit:**
  - \( \% = H₁ \)
  - Hit Access Time = 1
  - Stalls = 0

- **L₁ Miss**
  - \( (1 - H₁) \times \% \) clean
  - \( (1 - H₁) \times \% \) dirty

- **L₁ Miss, Clean**
  - Access Time = \( M + 1 \)
  - Stalls per access = \( M \)
  - Stall cycles = \( M \times (1 - H₁) \times \% \) clean

- **L₁ Miss, Dirty**
  - Access Time = \( 2M + 1 \)
  - Stalls per access = \( 2M \)
  - Stall cycles = \( 2M \times (1 - H₁) \times \% \) dirty

**Stall Cycles Per Memory Access**

\[
\text{Stall Cycles Per Memory Access} = (1 - H₁) \times (M \times \% \text{ clean} + 2M \times \% \text{ dirty})
\]

**AMAT** = 1 + Stall Cycles Per Memory Access

**CPI = CPI_{\text{execution}} + (1 \ + \ \text{fraction of loads/stores}) \times \text{Stall Cycles per access}**

- **M** = Miss Penalty = stall cycles per access resulting from missing in cache
- **M + 1** = Miss Time = Main memory access time
- **H₁** = Level 1 Hit Rate
- **1 - H₁** = Level 1 Miss Rate
Miss Rates For Multi-Level Caches

- **Local Miss Rate:** This rate is the number of misses in a cache level divided by the number of memory accesses to this level (i.e., those memory accesses that reach this level).

  Local Hit Rate = 1 - Local Miss Rate

- **Global Miss Rate:** The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.

- Since level 1 receives all CPU memory accesses, for level 1:

  Local Miss Rate = Global Miss Rate = 1 - H1

- For level 2 since it only receives those accesses missed in 1:

  Local Miss Rate = Miss rate\(_{L2}\) = 1 - H2
  
  Global Miss Rate = Miss rate\(_{L1}\) x Miss rate\(_{L2}\)
  
  = (1 - H1) x (1 - H2)

For Level 3, global miss rate?
Common Write Policy For 2-Level Cache

• **Write Policy For Level 1 Cache:**
  – Usually Write through to Level 2.
  – Write allocate is used to reduce level 1 read misses.
  – Use write buffer to reduce write stalls to level 2.

• **Write Policy For Level 2 Cache:**
  – Usually write back with write allocate is used.
    • To minimize memory bandwidth usage.

• The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2
  • Common in the majority of all CPUs with 2-levels of cache
  • As opposed to exclusive L1, L2 (e.g AMD Athlon XP, A64)
    As if we have a single level of cache with one portion (L1) is faster than remainder (L2)
    i.e what is in L1 is not duplicated in L2
L1: Write Through to L2, Write Allocate, With Perfect Write Buffer

L2: Write Back with Write Allocate

CPU Memory Access

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0

L1 Miss:
(1-H1) x H2

L1 Miss, L2 Hit:
Hit Access Time = T2 +1
Stalls per L2 Hit = T2
Stalls = (1-H1) x H2 x T2

L1 Miss, L2 Miss, Clean
Access Time = M +1
Stalls per access = M
Stall cycles = M x (1-H1) x (1-H2) x % clean

L1 Miss, L2 Miss, Dirty
Access Time = 2M +1
Stalls per access = 2M
Stall cycles = 2M x (1-H1) x (1-H2) x % dirty

Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1-H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty

= (1-H1) x H2 x T2 + (1-H1) x (1-H2) x ( % clean x M) + % dirty x 2M

AMAT = 1 + Stall Cycles Per Memory Access
CPI = CPI\text{execution} + (1 + fraction of loads and stores) x Stall Cycles per access

Assuming:
Ideal access on a hit in L1
T1 = 0

2-Level (Both Unified) Memory Access Tree
### 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>
X86 CPU Cache/Memory Performance Example:
AMD Athlon XP/64/FX Vs. Intel P4/Extreme Edition

Main Memory: Dual (64-bit) Channel PC3200 DDR SDRAM
peak bandwidth of 6400 MB/s

Source: The Tech Report 1-21-2004
Virtual Memory: Overview

- Virtual memory controls two levels of the memory hierarchy:
  - Main memory (DRAM).
  - Mass storage (usually magnetic disks).
- Main memory is divided into blocks allocated to different running processes in the system by the OS:
  - **Fixed size blocks**: Pages (size 4k to 64k bytes). *(Most common)*
  - **Variable size blocks**: Segments (largest size $2^{16}$ up to $2^{32}$).
  - **Paged segmentation**: Large variable/fixed size segments divided into a number of fixed size pages (X86, PowerPC).
- At any given time, for any running process, a portion of its data/code is loaded (allocated) in main memory while the rest is available only in mass storage.
- A program code/data block needed for process execution and not present in main memory result in a page fault (address fault) and the page has to be loaded into main memory by the OS from disk *(demand paging)*.
- A program can be run in any location in main memory or disk by using a relocation/mapping mechanism controlled by the operating system which maps (translates) the address from virtual address space (logical program address) to physical address space (main memory, disk).

**Using page tables**
Basic Virtual Memory Management

- Operating system makes decisions regarding which virtual (logical) pages of a process should be allocated in real physical memory and where (demand paging) assisted with hardware Memory Management Unit (MMU).

- On memory access -- If no valid virtual page to physical page translation (i.e page not allocated in main memory)
  - Page fault to operating system (e.g system call to handle page fault)
  - Operating system requests page from disk
  - Operating system chooses page for replacement
    - writes back to disk if modified
  - Operating system allocates a page in physical memory and updates page table w/ new page table entry (PTE). Then restart faulting process

Paging is assumed
Virtual Memory Basic Strategies

- **Main memory page placement (allocation):** Fully associative placement or allocation (by OS) is used to lower the miss rate.

- **Page replacement:** The least recently used (LRU) page is replaced when a new page is brought into main memory from disk.

- **Write strategy:** Write back is used and only those pages changed in main memory are written to disk (dirty bit scheme is used).

- **Page Identification and address translation:** To locate pages in main memory a page table is utilized to translate from virtual page numbers (VPNs) to physical page numbers (PPNs). The page table is indexed by the virtual page number and contains the physical address of the page.
  - **In paging:** Offset is concatenated to this physical page address.
  - **In segmentation:** Offset is added to the physical segment address.

- **Utilizing address translation locality,** a **translation lookaside buffer (TLB)** is usually used to cache recent address translations (PTEs) and prevent a second memory access to read the page table.

PTE = Page Table Entry
Direct Page Table Organization

Two memory accesses needed:
- First to page table.
- Second to item.

Page table usually in main memory.

How to speedup virtual to physical address translation?

Paging is assumed
Speeding Up Address Translation:
Translation Lookaside Buffer (TLB)

- Translation Lookaside Buffer (TLB): Utilizing address reference temporal locality, a small on-chip cache used for address translations (PTEs).
  - TLB entries usually 32-128
  - High degree of associativity usually used
  - Separate instruction TLB (I-TLB) and data TLB (D-TLB) are usually used.
  - A unified larger second level TLB is often used to improve TLB performance and reduce the associativity of level 1 TLBs.

- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.
- TLB-Refill: If a virtual address is not found in TLB, a TLB miss (TLB fault) occurs and the system must search (walk) the page table for the appropriate entry and place it into the TLB this is accomplished by the TLB-refill mechanism.

- Types of TLB-refill mechanisms:
  - Hardware-managed TLB: A hardware finite state machine is used to refill the TLB on a TLB miss by walking the page table. (PowerPC, IA-32)
  - Software-managed TLB: TLB refill handled by the operating system. (MIPS, Alpha, UltraSPARC, HP PA-RISC, ...)

Fast but not flexible
Speeding Up Address Translation:

Translation Lookaside Buffer (TLB)

- TLB: A small on-chip cache that contains recent address translations (PTEs).
- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.

Paging is assumed

TLB (on-chip)
32-128 Entries

Single-level Unified TLB shown