Advanced Computer Architecture

Subject Code	:	**06CS81**	IA Marks	:	25
No. of Lecture Hrs./ Week	:	04	Exam Hours	:	03
Total No. of Lecture Hrs.	:	52	Exam Marks	:	100

PART - A

UNIT - 1

FUNDAMENTALS OF COMPUTER DESIGN: Introduction; Classes of computers; Defining computer architecture; Trends in Technology, power in Integrated Circuits and cost; Dependability; Measuring, reporting and summarizing Performance; Quantitative Principles of computer design.

6 hours

UNIT - 2

PIPELINING: Introduction; Pipeline hazards; Implementation of pipeline; What makes pipelining hard to implement?

6 Hours

UNIT - 3

INSTRUCTION –LEVEL PARALLELISM – 1: ILP: Concepts and challenges; Basic Compiler Techniques for exposing ILP; Reducing Branch costs with prediction; Overcoming Data hazards with Dynamic scheduling; Hardware-based speculation.

7 Hours

UNIT - 4

INSTRUCTION –LEVEL PARALLELISM – 2: Exploiting ILP using multiple issue and static scheduling; Exploiting ILP using dynamic scheduling, multiple issue and speculation; Advanced Techniques for instruction delivery and Speculation; The Intel Pentium 4 as example. **7 Hours**

PART - B

UNIT - 5

MULTIPROCESSORS AND THREAD –LEVEL PARALLELISM: Introduction; Symmetric shared-memory architectures; Performance of symmetric shared–memory

multiprocessors; Distributed shared memory and directory-based coherence; Basics of synchronization; Models of Memory Consistency.

7 Hours

UNIT - 6

REVIEW OF MEMORY HIERARCHY: Introduction; Cache performance; Cache Optimizations, Virtual memory.

6 Hours

UNIT - 7

MEMORY HIERARCHY DESIGN: Introduction; Advanced optimizations of Cache performance; Memory technology and optimizations; Protection: Virtual memory and virtual machines.

6 Hours

UNIT - 8

HARDWARE AND SOFTWARE FOR VLIW AND EPIC: Introduction: Exploiting Instruction-Level Parallelism Statically; Detecting and Enhancing Loop-Level Parallelism; Scheduling and Structuring Code for Parallelism; Hardware Support for Exposing Parallelism: Predicated Instructions; Hardware Support for Compiler Speculation; The Intel IA-64 Architecture and Itanium Processor; Conclusions.

7 Hours

TEXT BOOK:

1. **Computer Architecture, A Quantitative Approach** – John L. Hennessey and David A. Patterson:, 4th Edition, Elsevier, 2007.

REFERENCE BOOKS:

1. **Advanced Computer Architecture Parallelism, Scalability** – Kai Hwang:, Programability**,** Tata Mc Grawhill, 2003.

2. **Parallel Computer Architecture, A Hardware / Software Approach** – David E. Culler, Jaswinder Pal Singh, Anoop Gupta:, Morgan Kaufman, 1999.

Table of Contents

PART - A

UNIT - 1

FUNDAMENTALS OF COMPUTER DESIGN:

Introduction; Classes of computers

Defining computer architecture

Trends in Technology, power in Integrated Circuits and cost

Dependability

Measuring reporting and summarizing Performance

Quantitative Principles of computer design.

6 hours

UNIT I

FUNDAMENTALS OF COMPUTER DESIGN

Introduction

Today' s desktop computers (less than $500 cost) ar e having more performance, larger memory and storage than a computer bought in 1085 for 1 million dollar. Highest performance microprocessors of today outperform Supercomputers of less than 10 years ago. The rapid improvement has come both from advances in the technology used to build computers and innovations made in the computer design or in other words, the improvement made in the computers can be attributed to innovations of technology and architecture design.

During the first 25 years of electronic computers, both forces made a major contribution, delivering performance improvement of about 25% per year. Microprocessors were evolved during late 1970s and their ability along with improvements made in the Integrated Circuit (IC) technology y contributed to 35% performance growth per year.

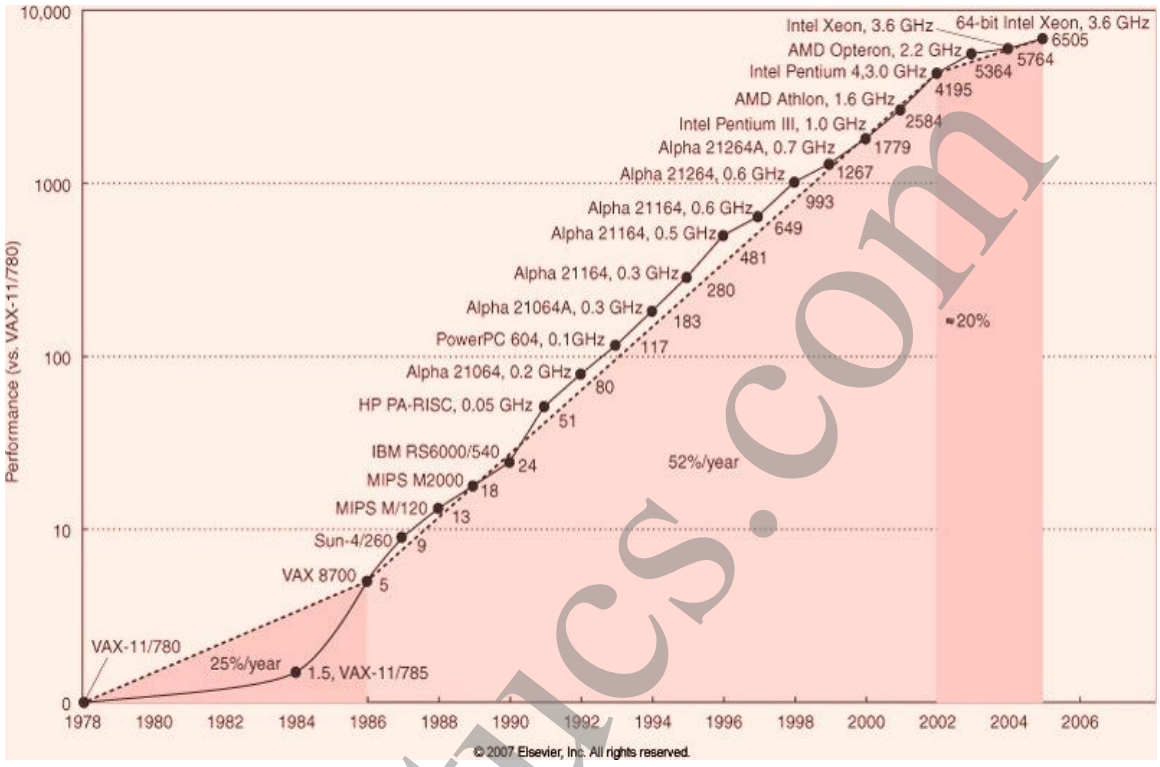
The virtual elimination of assembly language programming reduced the n eed for object-code compatibility. The creation of standardized vendor-independent operating system lowered the cost and risk of bringing out a new architecture.

In the yearly 1980s, the Reduced Instruction Set Computer (RISC) based machines focused the attention of designers on two critical performance techniques, the exploitation Instruction Level Parallelism (ILP) and the use of caches. The figu re 1.1 shows the growth in processor performance since the mid 1980s. The graph plots performance relative to the VAX-11/780 as measured b y the SPECint benchmarks. From the figure it is clear that architectural and organizational enhancements led to 16 years of sustained growth in performance at an annual rate of over 50%. Since 2002, processor performance improvement has dropped to about 20% per year due to the following hurdles:

•Maximum power dissipation of air-cooled chips
•Little ILP left to exploit efficiently
•Limitations laid by memory latency

The hurdles signals historic switch from relying solely on ILP to Thread Level Parallelism (TLP) and Data Level Parallelism (DLP).

Figure 1.1 The evolution of various classes of computers:



Classes of Computers

1960: Large Main frames (Millions of $)
(Applications: Business Data processing, large Scientific computin g)
1970: Minicomputers (Scientific laboratories, Time sharing concepts)
1980: Desktop Computers (µPs) in the form of Personal computers and workstations.
(Larger Memory, more computing power, Replaced Time sharing g systems)
1990: Emergence of Internet and WWW, PDAs, emergence of high performance digital consumer electronics
2000: Cell phones
These changes in computer use have led to three different computing classes each characterized by different applications, requirements and computing technologies.owth in processor performance since 1980s

Desktop computing

The first and still the largest market in dollar terms is desktop computing. Desktop computing system cost range from $ 500 (low end) to $ 5000 (high-end configuration). Throughout this range in price, the desktop market tends to drive to optimize price- performance. The perf ormance concerned is compute performance and graphics performance. The combination of performance and price are the driving factors to the customers and the computer designer. Hence, the newest, high performance and cost effective processor often appears first in desktop computers.

Servers:

Servers provide large-scale and reliable computing and file services and are mainly used in the large-scale en terprise computing and web based services. The three important

characteristics of servers are:

•**Dependability**: Severs must operate 24x7 hours a week. Failure of server system is far more catastrophic than a failure of desktop. Enterprise will lose revenue if the server is unavailable.

•**Scalability:** as the business grows, the server may have to provide more functionality/ services. Thus ability to scale up the computin g capacity, memory, storage and I/O bandwidth is crucial.

•**Throughput**: transactions completed per minute or web pages served per second are crucial for servers.

Embedded Computers

Simple embedded microprocessors are seen in washing machines, printers, network switches, handheld devices such as cell phones, smart cards video game devices etc. embedded computers have the widest spread of processing power and cost. The primary goal is often meeting the performance need at a minimum price rather than achieving higher performance at a higher price. The other two characteristic requirements are to minimize the memory and power.

In many embedded applications, the memory can be substantial portion of the systems cost and it is very important to optimize the memory size in such cases. The application is expected to fit totally in the memory on the p rocessor chip or off chip memory. The importance of memory size translates to an emphasis on code size which is dictated by the application. Larger memory consumes more power. All these aspects are considered while choosing or designing processor for the embedded applications.

Defining C omputer Arch itecture

The computer designer has to ascertain the attributes that are important for a new computer and design the system to maximize the performance while staying within cost, power and availability constraints. The task has few important aspects such as Instruction Set design, Functional organization, Logic design and implementation.

Instruction Set Architecture (ISA)

ISA refers to the actual programmer visible Instruction set. The ISA serves as boundary between the software and hardware. Th e seven dimensions of the ISA are:

i)Class of ISA: Nearly all ISAs today ar e classified as General-Purpose-Register architectures. The operands are either Registers or Memory locations.
The two popular versions of this class are:
Register-Memory ISAs : ISA of 80x86, can access memory as part of many instructions.
Load -Store ISA Eg. ISA of MIPS, can access memory only with Load or Store instructions.

ii)**Memory addressing**: Byte addressing scheme is most widely used in all desktop and server computers. Both 80x86 and MIPS use byte addressing. Incase of MIPS the object must be aligned. An access to an object of s b yte at byte address A is aligned if A mod s =0. 80x86 does not require alignment. Accesses are faster if operands are aligned.

iii) **Addressing modes**:Specify the address of a M object apart from register and constant operands.
MIPS Addressing modes:
•Register mode addressing
•Immediate mode addressing
•Displacement mode addressing

80x86 in addition to the above addressing modes supports the additional modes of addressing:

i. Register Indirect
ii. Indexed
iii,Based with Scaled index

iv)Types and sizes of operands:

MIPS and x86 support:
•8 bit (ASCII character), 16 bit(Unicode character)
•32 bit (Integer/word)
•64 bit (long integer/ Double word)
•32 bit (IEEE-754 floating point)
•64 bit (Double precision floating point)
•80x86 also supports 80 bit floating point operand.(extended double Precision

v)Operations:The general category o f operations are:

- oData Transfer
- oArithmetic operations
- oLogic operations
- oControl operations
- oMIPS ISA: simple & easy to implement
- ox86 ISA: richer & larger set of operations

vi) Control flow instructions:All ISAs support:
Conditional & Unconditional Branches
Procedure C alls & Returns MIPS 80x86

- Conditional Branches tests content of Register Condition code bits
- Procedure C all JAL CALLF
- Return Address in a R Stack in M

vii) Encoding an ISA

Fixed Length ISA	**Variable Length ISA**
MIPS 32 Bit long	80x86 (1-18 bytes)
Simplifies decoding	Takes less space

Number of Registers and number of Addressing modes hav e significant impact on the length of instruction as the register field and addressing mode field can appear many times in a single instruction.

Trends in Technology

The designer must be aware of the following rapid changes in implementation technology.

- •Integrated C ircuit (IC) Logic technology
- •Memory technology (semiconductor DRAM technology)
- •Storage o r magnetic disk technology
- •Network technology

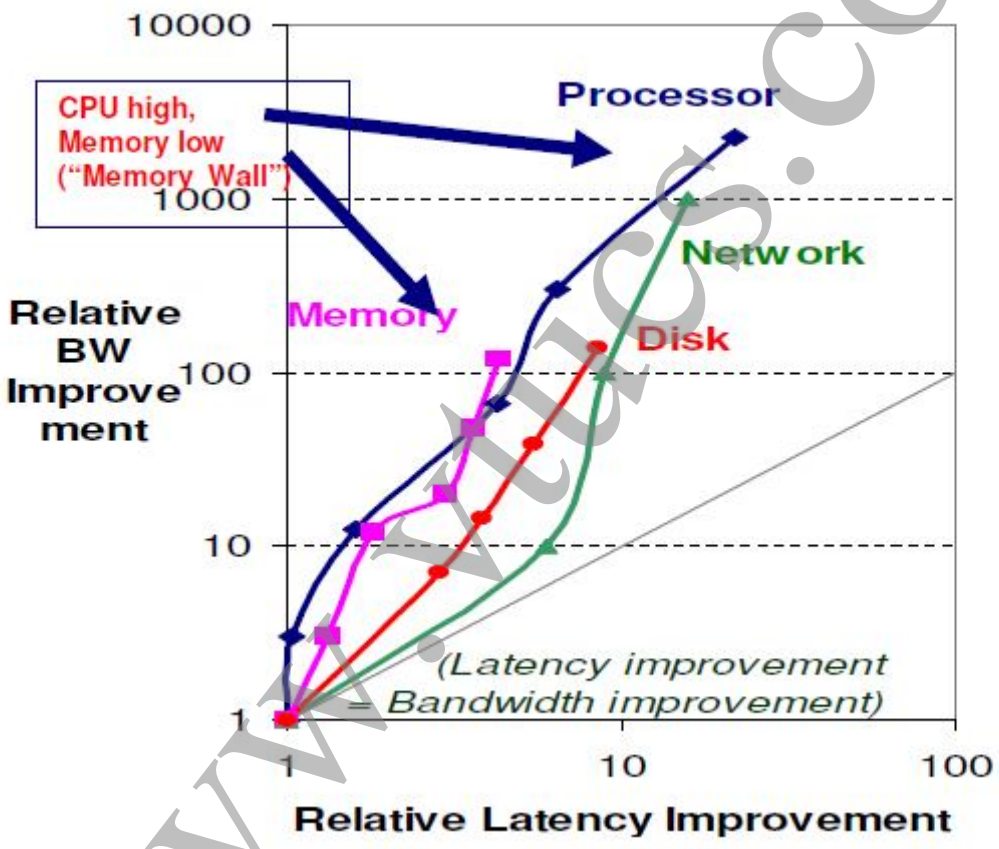
IC Logic technology:

Transistor density increases by about 35%per year. Increase in die size corresponds to about 10 % to 20% per year. The combined effect is a growth rate in transistor count on a chip of about 40% to 55% per year. Semiconductor DRAM technology:cap acity increases by about 40% per year.
Storage Technology:
Before 1990: the storage density increased by about 30% per year.
After 1990: the storage density increased by about 60 % per year.
Disks are still 50 to 100 times cheaper per bit than DRAM.

Network Technology:

Network performance depends both on the per formance of the switches and on the performance of the transmission system. Although the technology improves continuously, the impact of these improvements can be in discrete leaps.

Performance trends: Bandwidth or throughput is the total amount of work done in given time.
Latency or response time is the time between the start and the completion of an event. (for eg. Millisecond for disk access)



A simple rule of thumb is that bandwidth gro ws by at least the square of the improvement in latency. Computer designers should make plans accordingly.

•IC Processes are characterizes by the f ature sizes.

•Feature sizes decreased from 10 microns(1971) to 0.09 microns(2006)

•Feature sizes shrink, devices shrink quadr atically.

•Shrink in vertical direction makes the operating v oltage of the transistor to reduce.

•Transistor performance improves linearly with decreasing feature size

.

•Transistor count improves quadratically with a linear improvement in Transistor performance.

•!!! Wire delay scales poo rly comp ared to Transistor performance.

•Feature sizes shrink, wires get shorter.

•Signal delay fo r a wire increases in proportion to the product of Resistance and Capacitance.

Trends in Power in Integrated Circuits

For CMOS chips, the dominant source of energy consumption is due to switching transistor, also called as Dynamic power and is given b y the following equation.

Power = (1/2)*Capacitive load* Voltage

* Frequency switched dynamic

•For mobile devices, energy is the better metric

Energy *dynamic* = Capacitive load x Voltage 2

•For a fix ed task, slowing clock rate (frequency switched) reduces power, but not energy

•Capacitive load a function of number of transistors connected to output and technology, which determines capacitance of wires and transistors

•Dropping voltage helps both, so went from 5V down to 1V

•To save energy & dynamic power, most CPUs now turn off clock of inactive modules

•Distributing the power, removing the heat and preventing hot spots have become increasingly difficult challenges.

• The leakage current flows even when a transistor is off. Therefore *static power* is equally important.

$$\text{Power}_{\text{static}} = \text{Current}_{\text{static}} * \text{Voltage}$$

•Leakage current increases in processors with smaller transistor sizes

•Increasing the number of transistors increases power even if they are turned off

•In 2006, goal for leakage is 25% of total power consumption; high performance designs at 40%

•Very low power systems even gate voltage to inactive modules to control loss due to leakage

Trends in Cost

• The underlying principle that drives the cost down is the learning curvemanufacturing costs decrease over time.

• Volume is a second key factor in determining cost. Volume decreases cost since it increases purchasing manufacturing efficiency. As a rule of thumb, the cost decreases

about 10% for each doubling of volume.

• Cost of an Integrated Circuit

Although the cost of ICs have dropped exponentially, the basic process of silicon manufacture is unchanged. A wafer is still tested and chopped into dies that are packaged.

Cost of IC = Cost of [die+ testing die+ Packaging and final test] / (Final test yoeld)

Cost of die = Cost of wafer/ (Die per wafer x Die yield)

The number of dies per wafer is approximately the area of the wafer divided by the area of the die.

Die per wafer = [_ * (Wafer Dia/2)2/Die area]-[_* wafer dia/_(2*Die area)]

The first term is the ratio of wafer area to die area and the second term compensates for the rectangular dies near the periphery of round wafers(as shown in figure).

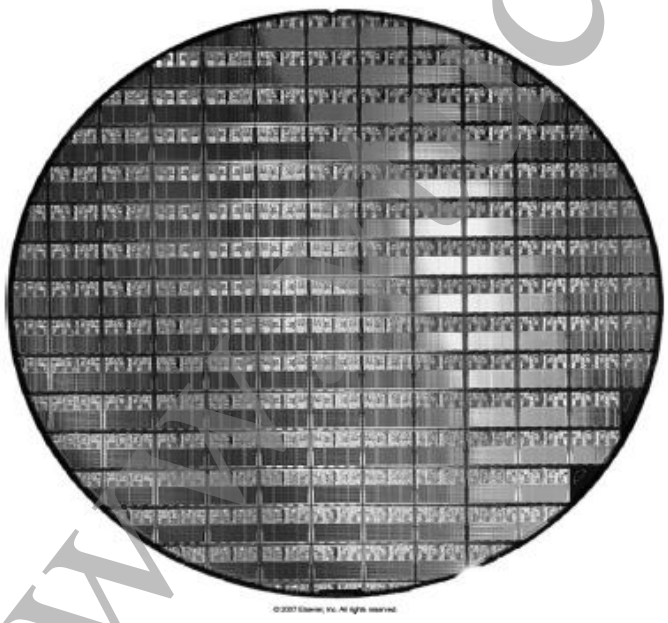


Figure 1.2 Close view of 300 mm wafer

Dependability:

The Infrastructure providers offer Service Level Agreement (SLA) or Service Level Objectives (SLO) to guarantee that their networking or power services would be dependable.

• Systems alternate between 2 states of service with respect to an SLA:
1. **Service accomplishment**, where the service is delivered as specified in SLA
2. **Service interruption**, where the delivered service is different from the SLA
• Failure = transition from state 1 to state 2
• Restoration = transition from state 2 to state 1

The two main measures of Dependability are Module Reliability and Module Availability. *Module reliability* is a measure of continuous service accomplishment (or time to failure) from a reference initial instant.

1. *Mean Time To Failure* (*MTTF*) measures Reliability
2. *Failures In Time* (*FIT*) = 1/MTTF, the rate of failures
• Traditionally reported as failures per billion hours of operation
• *Mean Time To Repair* (*MTTR*) measures Service Interruption
– *Mean Time Between Failures* (*MTBF*) = MTTF+MTTR
• *Module availability* measures service as alternate between the 2 states of accomplishment and interruption (number between 0 and 1, e.g. 0.9)
• *Module availability = MTTF / (MTTF + MTTR)*

Performance:

The *Execution time* or *Response time* is defined as the time between the start and completion of an event. The total amount of work done in a given time is defined as the *Throughput*.

The Administrator of a data center may be interested in increasing the *Throughput*. The computer user may be interested in reducing the *Response time*.

Computer user says that computer is faster when a program runs in less time.

$$\text{Performance} = \frac{1}{\text{Execution Time (X)}}$$

The phrase **"X is faster than Y"** is used to mean that the response time or execution time is lower on X than Y for the given task. **"X is n times faster than Y"** means

$$\text{Execution Time}_y = n * \text{Execution time}_x$$

$$\text{Performance}_x = n * \text{Perfromance}_y$$

The routinely executed programs are the best candidates for evaluating the performance of the new computers. To evaluate new system the user would simply compare the execution time of their workloads.

Benchmarks

The real applications are the best choice of benchmarks to evaluate the performance. However, for many of the cases, the workloads will not be known at the time of evaluation. Hence, the benchmark program which resemble the real applications are chosen. The three types of benchmarks are:

- KERNELS, which are small, key pieces of real applications;
- Toy Programs: which are 100 line programs from beginning programming assignments, such Quicksort etc.,
- Synthetic Benchmarks: Fake programs invented to try to match the profile and behavior of real applications such as Dhrystone.

To make the process of evaluation a fair justice, the following points are to be followed.

- Source code modifications are not allowed.
- Source code modifications are allowed, but are essentially impossible.
- Source code modifications are allowed, as long as the modified version produces the same output.
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
- SPECCPU: popular desktop benchmark suite given by Standard Performance Evaluation committee (SPEC)
  - CPU only, split between integer and floating point programs
  - SPECint2000 has 12 integer, SPECfp2000 has 14 integer programs
  - SPECCPU2006 announced in Spring 2006.

  SPECSFS (NFS file server) and SPECWeb (WebServer) added as server benchmarks

- Transaction Processing Council measures server performance and costperformance for databases
  - TPC-C Complex query for Online Transaction Processing
  - TPC-H models ad hoc decision support
  - TPC-W a transactional web benchmark
  - TPC-App application server and web services benchmark

• **SPEC Ratio:** Normalize execution times to reference computer, yielding a ratio proportional to performance = time on reference computer/time on computer being rated

• If program SPECRatio on Computer A is 1.25 times bigger than Computer B, then

$$1.25 = \frac{SPECRatio_A}{SPECRatio_B} = \frac{\frac{ExecutionTime_{reference}}{ExecutionTime_A}}{\frac{ExecutionTime_{reference}}{ExecutionTime_B}}$$

$$= \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

• **Note :** when comparing 2 computers as a ratio, execution times on the reference computer drop out, so choice of reference computer is irrelevant.

Quantitative Principles of Computer Design

While designing the computer, the advantage of the following points can be exploited to enhance the performance.

* **Parallelism:** is one of most important methods for improving performance.

- One of the simplest ways to do this is through pipelining ie, to over lap the instruction Execution to reduce the total time to complete an instruction sequence.
- Parallelism can also be exploited at the level of detailed digital design.
- Set- associative caches use multiple banks of memory that are typically searched n parallel. Carry look ahead which uses parallelism to speed the process of computing.

* **Principle of locality:** program tends to reuse data and instructions they have used recently. The rule of thumb is that program spends 90 % of its execution time in only 10% of the code. With reasonable good accuracy, prediction can be made to find what instruction and data the program will use in the near future based on its accesses in the recent past.

* **Focus on the common case** while making a design trade off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, since the impact of the improvement is higher if the occurrence is frequent.

Amdahl's Law: Amdahl's law is used to find the performance gain that can be obtained by improving some portion or a functional unit of a computer Amdahl's law defines the speedup that can be gained by using a particular feature.

Speedup is the ratio of performance for entire task without using the enhancement when possible to the performance for entire task without using the enhancement. Execution time is the reciprocal of performance. Alternatively, speedup is defined as thee ratio of execution time for entire task without using the enhancement to the execution time for entair task using the enhancement when possible.

Speedup from some enhancement depends an two factors:

i. The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement. Fraction enhanced is always less than or equal to

Example: If 15 seconds of the execution time of a program that takes 50 seconds in total can use an enhancement, the fraction is 15/50 or 0.3

ii. The improvement gained by the enhanced execution mode; ie how much faster the task would run if the enhanced mode were used for the entire program. Speedup enhanced is the time of the original mode over the time of the enhanced mode and is always greater then 1.

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speed up}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}}$$

The Processor performance Equation:

Processor is connected with a clock running at constant rate. These discrete time events are called clock ticks or clock cycle.

CPU time for a program can be evaluated:

CPU time = CPU clock cycles for a program X clock cycle time

$$\text{CPU time} = \frac{\text{CPU Clock cycles for a program}}{\text{Clock rate}}$$

Using the number of clock cycle and the Instruction count (IC), it is possible to determine the average number of clock cycles per instruction (CPI). The reciprocal of CPI gives Instruction per clock (IPC)

$$\text{CPI} = \frac{\text{CPU clock cycle for a program}}{\text{Instruction count}}$$

CPU time = IC X CPI X Clock cycle time

$$\text{CPU time} = \frac{\text{Seconds}}{\text{program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{clock cycle}}$$

Processor performance depends on IC, CPI and clock rate or clock cycle. There 3 parameters are dependent on the following basic technologies.

Clock Cycle time – H/W technology and organization

CPI- organization and ISA

IC- ISA and compiler – technology

Example:

A System contains Floating point (FP) and Floating Point Square Root (FPSQR) unit. FPSQR is responsible for 20% of the execution time. One proposal is to enhance the FPSQR hardware and speedup this operation by a factor of 15 second alternate is just to try to make all FP instructions run faster by a factor of 1.6 times faster with the same effort as required for the fast FPSQR, compare the two design alternative

Option 1

$$\text{Speedup}_{FPSQR} = \frac{1}{(1-0.2) + (0.2/15)} = 1.2295$$

Option 2

$$\text{Speedup}_{FP} = \frac{1}{(1-0.5) + (0.5/1.6)} = 1.2307$$

Option 2 is relatively better.

UNIT - 2

PIPELINING:

Introduction

Pipeline hazards

Implementation of pipeline

What makes pipelining hard to implement?

6 Hours

UNIT II

Pipelining: Basic and Intermediate concepts

Pipeline is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. Pipeline allows to overlapping the execution of multiple instructions. A Pipeline is like an assembly line each step or pipeline stage completes a part of an instructions. Each stage of the pipeline will be operating an a separate instruction. Instructions enter at one end progress through the stage and exit at the other end. If the stages are perfectly balance.
(assuming ideal conditions), then the time per instruction on the pipeline processor is given by the ratio:

Time per instruction on unpipelined machine/ Number of Pipeline stages

Under these conditions, the speedup from pipelining is equal to the number of stage pipeline. In practice, the pipeline stages are not perfectly balanced and pipeline does involve some overhead. Therefore, the speedup will be always then practically less than the number of stages of the pipeline. Pipeline yields a reduction in the average execution time per instruction. If the processor is assumed to take one (long) clock cycle per instruction, then pipelining decrease the clock cycle time. If the processor is assumed to take multiple CPI, then pipelining will aid to reduce the CPI.

A Simple implementation of a RISC instruction set
Instruction set of implementation in RISC takes at most 5 cycles without pipelining.
The 5 clock cycles are:

1. Instruction fetch (IF) cycle:

Send the content of program count (PC) to memory and fetch the current instruction from memory to update the PC.

New PC ← [PC] + 4; Since each instruction is 4 bytes

2. Instruction decode / Register fetch cycle (ID):

Decode the instruction and access the register file. Decoding is done in parallel with reading registers, which is possible because the register specifies are at a fixed location in a RISC architecture. This corresponds to fixed field decoding. In addition it involves:

- Perform equality test on the register as they are read for a possible branch.
- Sign-extend the offset field of the instruction in case it is needed.
- Compute the possible branch target address.

3. Execution / Effective address Cycle (EXE)

The ALU operates on the operands prepared in the previous cycle and performs one of the following function defending on the instruction type.

* Memory reference: Effective address ← [Base Register] + offset

* Register- Register ALU instruction: ALU performs the operation specified in the instruction using the values read from the register file.
* Register- Immediate ALU instruction: ALU performs the operation specified in the instruction using the first value read from the register file and that sign extended immediate.

4. Memory access (MEM)

For a load instruction, using effective address the memory is read. For a store instruction memory writes the data from the 2nd register read using effective address.

5. Write back cycle (WB)

Write the result in to the register file, whether it comes from memory system (for a LOAD instruction) or from the ALU.

Five stage Pipeline for a RISC processor

Each instruction taken at most 5 clock cycles for the execution
* Instruction fetch cycle (IF)
* Instruction decode / register fetch cycle (ID)
* Execution / Effective address cycle (EX)
* Memory access (MEM)
* Write back cycle (WB)

The execution of the instruction comprising of the above subtask can be pipelined. Each of the clock cycles from the previous section becomes a pipe stage – a cycle in the pipeline. A new instruction can be started on each clock cycle which results in the execution pattern shown figure 2.1. Though each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions as illustrated in figure 2.1.

Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB

Figure 2.1 Simple RISC Pipeline. On each clock cycle another instruction fetched

Each stage of the pipeline must be independent of the other stages. Also, two different operations can't be performed with the same data path resource on the same clock. For example, a single ALU cannot be used to compute the effective address and perform a subtract operation during the same clock cycle. An adder is to be provided in the stage 1 to compute new PC value and an ALU in the stage 3 to perform the arithmetic indicatedin the instruction (See figure 2.2). Conflict should not arise out of overlap of instructions using pipeline. In other words, functional unit of each stage need to be independent of other functional unit. There are three observations due to which the risk of conflict is reduced.

- Separate Instruction and data memories at the level of L1 cache eliminates a conflict for a single memory that would arise between instruction fetch and data access.
- Register file is accessed during two stages namely ID stage WB. Hardware should allow to perform maximum two reads one write every clock cycle.
- To start a new instruction every cycle, it is necessary to increment and store the PC every cycle.

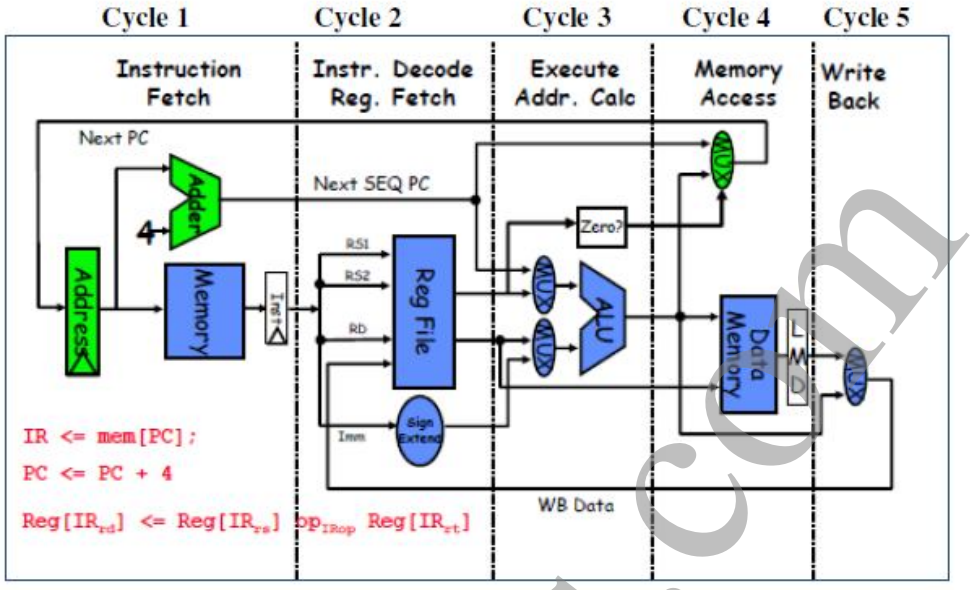


Figure 2.2 Diagram indicating the cycle and functional unit of each stage.

Buffers or registers are introduced between successive stages of the pipeline so that at the end of a clock cycle the results from one stage are stored into a register (see figure 2.3). During the next clock cycle, the next stage will use the content of these buffers as input. Figure 2.4 visualizes the pipeline activity.

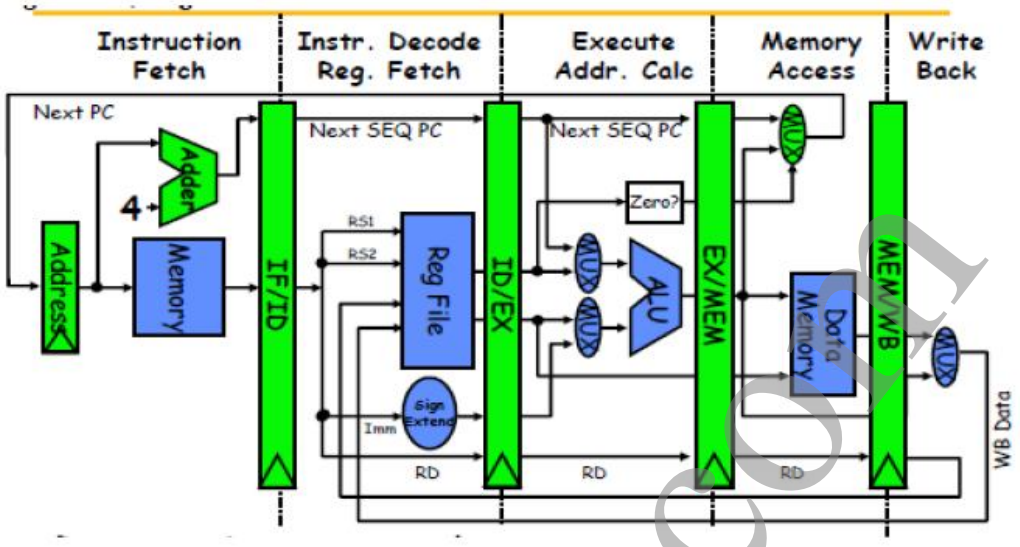


Figure 2.3 Functional units of 5 stage Pipeline. IF/ID is a buffer between IF and ID stage.

Basic Performance issues in Pipelining

Pipelining increases the CPU instruction throughput but, it does not reduce the executiontime of an individual instruction. In fact, the pipelining increases the execution time of each instruction due to overhead in the control of the pipeline. Pipeline overhead arises from the combination of register delays and clock skew. Imbalance among the pipe stages reduces the performance since the clock can run no faster than the time needed for the slowest pipeline stage.

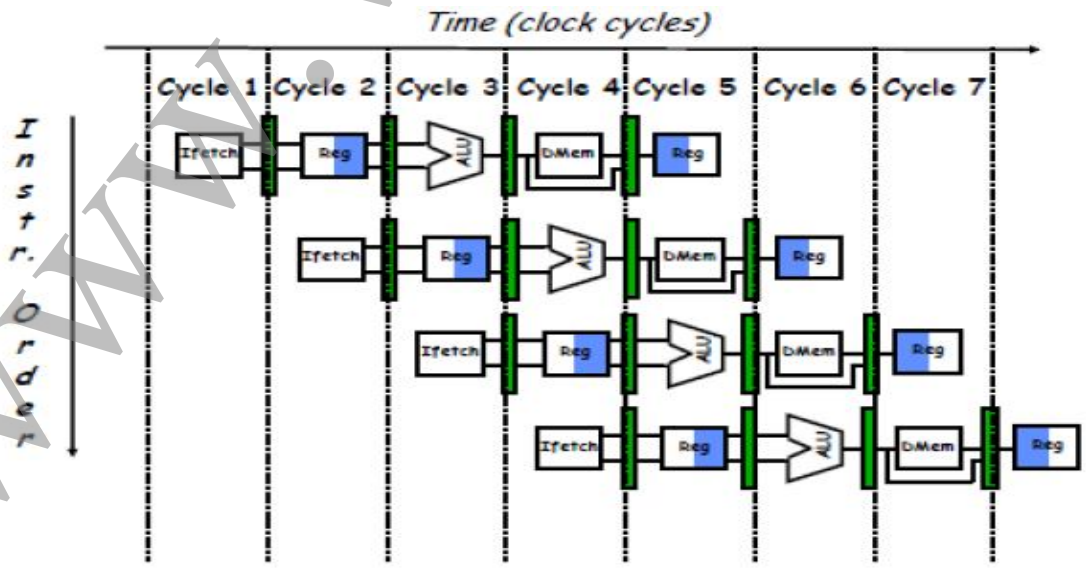


Figure 2.4 Pipeline activity

Pipeline Hazards

Hazards may cause the pipeline to stall. When an instruction is stalled, all the instructions issued later than the stalled instructions are also stalled. Instructions issued earlier than the stalled instructions will continue in a normal way. No new instructions are fetched during the stall. Hazard is situation that prevents the next instruction in the instruction stream fromk executing during its designated clock cycle. Hazards will reduce the pipeline performance.

Performance with Pipeline stall

A stall causes the pipeline performance to degrade from ideal performance. Performance improvement from pipelining is obtained from:

$$\textbf{Speedup} = \frac{\text{Average instruction time un-pipelined}}{\text{Average instruction time pipelined}}$$

$$\textbf{Speedup} = \frac{\text{CPI unpipelined} * \text{Clock cycle unpipelined}}{\text{CPI pipelined} * \text{Clock cycle pipelined}}$$

$$\textbf{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$\textbf{CPI pipelined} = 1 + \text{Pipeline stall clock cycles per instruction}$$

Assume that,
i) cycle time overhead of pipeline is ignored
ii) stages are balanced
With theses assumptions

$$\text{Clock cycle unpipelined} = \text{clock cycle pipelined}$$

$$\text{Therefore, Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

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

If all the instructions take the same number of cycles and is equal to the number of pipeline stages or depth of the pipeline, then,

$$\text{CPI unpipelined} = \text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1+\text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls,
Pipeline stall cycles per instruction = zero
Therefore,
Speedup = Depth of the pipeline.

Types of hazard

Three types hazards are:

1. Structural hazard
2. Data Hazard
3. Control Hazard

Structural hazard

Structural hazard arise from resource conflicts, when the hardware cannot support all possible combination of instructions simultaneously in overlapped execution. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have structural hazard. Structural hazard will arise when some functional unit is not fully pipelined or when some resource has not been duplicated enough to allow all combination of instructions in the pipeline to execute. For example, if memory is shared for data and instruction as a result, when an instruction contains data memory reference, it will conflict with the instruction reference for a later instruction (as shown in figure 2.5a). This will cause hazard and pipeline stalls for 1 clock cycle.

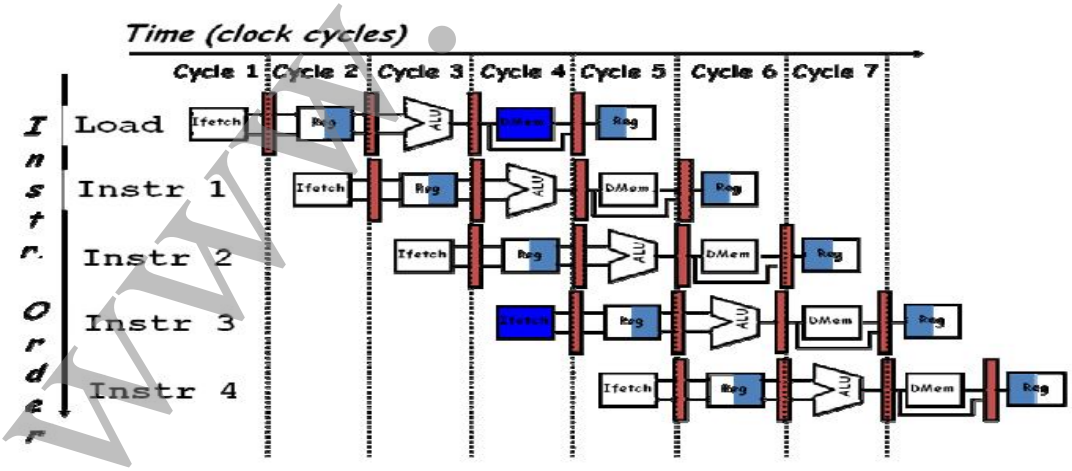
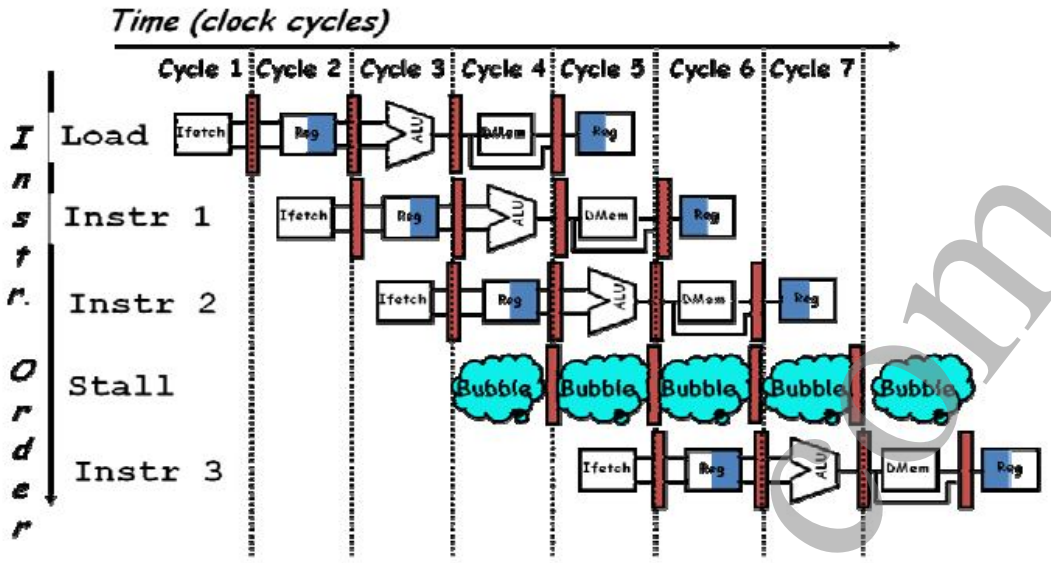


Figure 2.5a Load Instruction and instruction 3 are accessing memory in clock cycle4



Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Load Instruction	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				Stall	IF	ID	EXE	MEM	WB
Instruction I+4						IF	ID	EXE	MEM

Figure 2.5b A Bubble is inserted in clock cycle 4

Pipeline stall is commonly called Pipeline bubble or just simply bubble

Data Hazard

Consider the pipelined execution of the following instruction sequence (Timing diagram shown in figure 2.6)

```
DADD    R1, R2, R3
DSUB    R4,R1,R5
AND     R6,R1,R5
OR      R8, R1,R9
XOR     R10,R1,R11
```

DADD instruction produces the value of R1 in WB stage (Clock cycle 5) but the DSUB instruction reads the value during its ID stage (clock cycle 3). This problem is called Data Hazard. DSUB may read the wrong value if precautions are not taken. AND instruction will read the register during clock cycle 4 and will receive the wrong results. The XOR instruction operates properly, because its register read occurs in clock cycle 6 after DADD writes in clock cycle 5. The OR instruction also operates without incurring a hazard because the register file reads are performed in the second half of the cycle whereas the writes are performed in the first half of the cycle.

Minimizing data hazard by Forwarding

The DADD instruction will produce the value of R! at the end of clock cycle 3. DSUB instruction requires this value only during the clock cycle 4. If the result can be moved from the pipeline register where the DADD store it to the point (input of LAU) where DSUB needs it, then the need for a stall can be avoided. Using a simple hardware technique called Data Forwarding or Bypassing or short circuiting, data can be made available from the output of the ALU to the point where it is required (input of LAU) at the beginning of immediate next clock cycle.

Forwarding works as follows:

i) The output of ALU from EX/MEM and MEM/WB pipeline register is always feedback to the ALU inputs.

ii) If the Forwarding hardware detects that the previous ALU output serves as the source for the current ALU operations, control logic selects the forwarded result as the input rather than the value read from the register file. Forwarded results are required not only from the immediate previous instruction, but also from an instruction that started 2 cycles earlier. The result of i_{th} instruction Is required to be forwarded to $(i+2)_{th}$ instruction also. Forwarding can be generalized to include passing a result directly to the functional unit that requires it.

Data Hazard requiring stalls

```
LD    R1,  0(R2)
DADD R3, R1, R4
AND   R5, R1, R6
OR    R7, R1, R8
```

The pipelined data path for these instructions is shown in the timing diagram (figure 2.7)

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
LD R1, 0(R2)	IF	ID	EXE	MEM	WB				
DADD R3,R1,R4		IF	ID	EXE	MEM	WB			
AND R5, R1, R6			IF	ID	EXE	MEM	WB		
OR R7, R1, R8				IF	ID	EXE	MEM	WB	
LD R1, 0(R2)	IF	ID	EXE	MEM	WB				
DADD R3,R1,R4		IF	ID	Stall	EXE	MEM	WB		
AND R5, R1, R6			IF	Stall	ID	EXE	MEM	WB	
OR R7, R1, R8				Stall	IF	ID	EXE	MEM	WB

Figure 2.7 In the top half, we can see why stall is needed. In the second half, stall created to solve the problem.

The LD instruction gets the data from the memory at the end of cycle 4. even with forwarding technique, the data from LD instruction can be made available earliest during clock cycle 5. DADD instruction requires the result of LD instruction at the beginning of clock cycle 5. DADD instruction requires the result of LD instruction at the beginning of clock cycle 4. This demands data forwarding of clock cycle 4. This demands data forwarding in negative time which is not possible. Hence, the situation calls for a pipeline stall.Result from the LD instruction can be forwarded from the pipeline register to the and instruction which begins at 2 clock cycles later after the LD instruction. The load instruction has a delay or latency that cannot be eliminated by forwarding alone. It is necessary to stall pipeline by 1 clock cycle. A hardware called Pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared. The pipeline interlock helps to preserve the correct execution pattern by introducing a stall or bubble. The CPI for the stalled instruction increases by the length of the stall. Figure 2.7 shows the pipeline before and after the stall. Stall causes the DADD to move 1 clock cycle later in time. Forwarding to the AND instruction now goes through the register file or forwarding is not required for the OR instruction. No instruction is started during the clock cycle 4.

Control Hazard

When a branch is executed, it may or may not change the content of PC. If a branch is taken, the content of PC is changed to target address. If a branch is taken, the content of PC is not changed

The simple way of dealing with the branches is to redo the fetch of the instruction following a branch. The first IF cycle is essentially a stall, because, it never performs useful work. One stall cycle for every branch will yield a performance loss 10% to 30% depending on the branch frequency

Reducing the Brach Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay

1. Freeze or Flush the pipeline, holding or deleting any instructions after the ranch until the branch destination is known. It is a simple scheme and branch penalty is fixed and cannot be reduced by software

2. Treat every branch as not taken, simply allowing the hardware to continue as if the branch were not to executed. Care must be taken not to change the processor state until the branch outcome is known.

Instructions were fetched as if the branch were a normal instruction. If the branch is taken, it is necessary to turn the fetched instruction in to a no-of instruction and restart the fetch at the target address. Figure 2.8 shows the timing diagram of both the situations.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	Idle	Idle	Idle	Idle	Idle		
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.8 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

3. Treat every branch as *taken*: As soon as the branch is decoded and target Address is computed, begin fetching and executing at the target if the branch target is known before branch outcome, then this scheme gets advantage.

For both predicated taken or predicated not taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware choice.

4. Delayed branch technique is commonly used in early RISC processors.

In a delayed branch, the execution cycle with a branch delay of one is
Branch instruction
Sequential successor-1
Branch target if taken

The sequential successor is in the branch delay slot and it is executed irrespective of whether or not the branch is taken. The pipeline behavior with a branch delay is shown in Figure 2.9. Processor with delayed branch, normally have a single instruction delay. Compiler has to make the successor instructions valid and useful there are three ways in

which the to delay slot can be filled by the compiler.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Instruction (i+2)			IF	ID	EXE	MEM	WB		
Instruction (i+3)				IF	ID	EXE	MEM	WB	
Instruction (i+4)					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.9 Timing diagram of the pipeline to show the behavior of a delayed branch is the same whether or not the branch is taken.

The limitations on delayed branch arise from

i) Restrictions on the instructions that are scheduled in to delay slots.

ii) Ability to predict at compiler time whether a branch is likely to be taken or not taken.

The delay slot can be filled from choosing an instruction

a) From before the branch instruction

b) From the target address

c) From fall- through path.

The principle of scheduling the branch delay is shown in fig 2.10

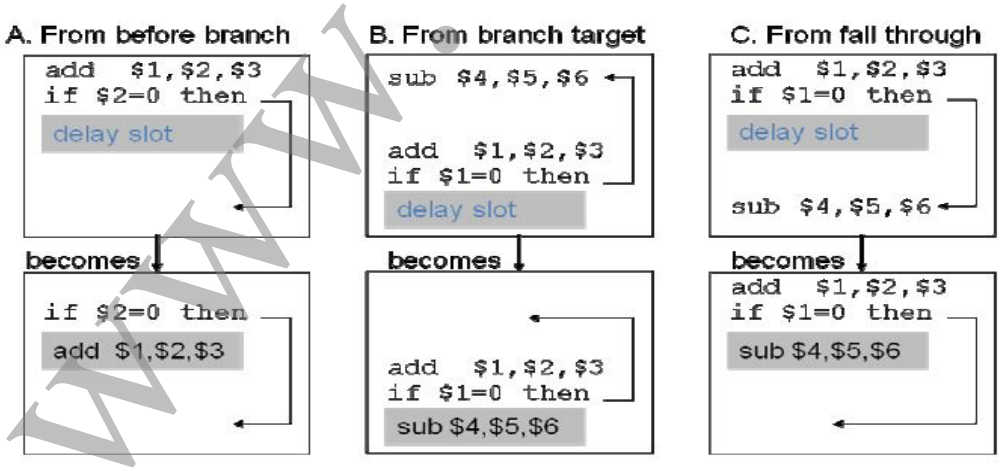


Figure 2.10 Scheduling the Branch delay

What makes pipelining hard to implements?

Dealing with exceptions: Overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the CPU. In a pipelined CPU, an instruction execution extends over several clock cycles. When this instruction is in execution, the other instruction may raise exception that may force the CPU to abort the instruction in the pipeline before they complete

Types of exceptions:

The term exception is used to cover the terms interrupt, fault and exception. I/O device request, page fault, Invoking an OS service from a user program, Integer arithmetic overflow, memory protection overflow, Hardware malfunctions, Power failure etc. are the different classes of exception. Individual events have important characteristics that determine what action is needed corresponding to that exception.

i) Synchronous versus Asynchronous

If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is asynchronous. Asynchronous events are caused by devices external to the CPU and memory such events are handled after the completion of the current instruction.

ii) User requested versus coerced:

User requested exceptions are predictable and can always be handled after the current instruction has completed. Coerced exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable

iii) User maskable versus user non maskable :

If an event can be masked by a user task, it is user maskable. Otherwise it is user non maskable.

iv) Within versus between instructions:

Exception that occur within instruction are usually synchronous, since the instruction triggers the exception. It is harder to implement exceptions that occur withininstructions than those between instructions, since the instruction must be stopped and restarted. Asynchronous exceptions that occurs within instructions arise from catastrophic situations and always causes program termination.

v) Resume versus terminate:

If the program's execution continues after the interrupt, it is a resuming event otherwise if is terminating event. It is easier implement exceptions that terminate execution. 29

Stopping and restarting execution:

The most difficult exception have 2 properties:

1. Exception that occur within instructions
2. They must be restartable

For example, a page fault must be restartable and requires the intervention of OS. Thus pipeline must be safely shutdown, so that the instruction can be restarted in the correct state. If the restarted instruction is not a branch, then we will continue to fetch the sequential successors and begin their execution in the normal fashion. 11) Restarting is usually implemented by saving the PC of the instruction at which to restart. Pipeline control can take the following steps to save the pipeline state safely.

i) Force a trap instruction in to the pipeline on the next IF

ii) Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in pipeline. This prevents any state changes for instructions that will not be completed before the exception is handled.

iii) After the exception – handling routine receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

NOTE:

1. with pipelining multiple exceptions may occur in the same clock cycle because there are multiple instructions in execution.

2 Handling the exception becomes still more complicated when the instructions are allowed to execute in *out of order* fashion.

Pipeline implementation

Every MIPS instruction can be implemented in 5 clock cycle

1. Instruction fetch cycles.(IF)

IR ← Mem [PC]

NPC ← PC+ 4

Operation: send out the [PC] and fetch the instruction from memory in to the Instruction Register (IR). Increment PC by 4 to address the next sequential instruction.

2. Instruction decode / Register fetch cycle (ID)

A ← Regs [rs]

B ← Regs [rt]

Imm ← sign – extended immediate field of IR;

Operation: decode the instruction and access that register file to read the registers (rs and rt). File to read the register (rs and rt). A & B are the temporary registers. Operands are kept ready for use in the next cycle.

Decoding is done in concurrent with reading register. MIPS ISA has fixed length Instructions. Hence, these fields are at fixed locations.

3. Execution/ Effective address cycle (EX)

One of the following operations are performed depending on the instruction type.
* Memory reference:

: ALU output ← A+ Imm;

Operation: ALU adds the operands to compute the effective address and places the result in to the register ALU output.

- Register – Register ALU instruction:

ALU output ← A _*func* B;

Operation: The ALU performs the operation specified by the function code on the value taken from content of register A and register B.
*. Register- Immediate ALU instruction:

ALU output ← A *Op* Imm ;

Operation: the content of register A and register Imm are operated (function Op) and result is placed in temporary register ALU output.
*. Branch:

ALU output ← NPC + (Imm << 2)
Cond ← (A == 0)

UNIT - 3

INSTRUCTION –LEVEL PARALLELISM – 1: ILP

Concepts and challenges

Basic Compiler Techniques for exposing ILP

Reducing Branch costs with prediction

Overcoming Data hazards with Dynamic scheduling

Hardware-based speculation.

7 Hours

UNIT III

Instruction Level Parallelism

The potential overlap among instruction execution is called Instruction Level Parallelism (ILP) since instructions can be executed in parallel. There are mainly two approaches to exploit ILP.

i) Hardware based approach: An approach that relies on hardware to help discover and exploit the parallelism dynamically. Intel Pentium series which has dominated in the market) uses this approach.

ii) Software based approach: An approach that relies on software technology to find parallelism statically at compile time. This approach has limited use in scientific or application specific environment. Static approach of exploiting ILP is found in Intel Itanium.

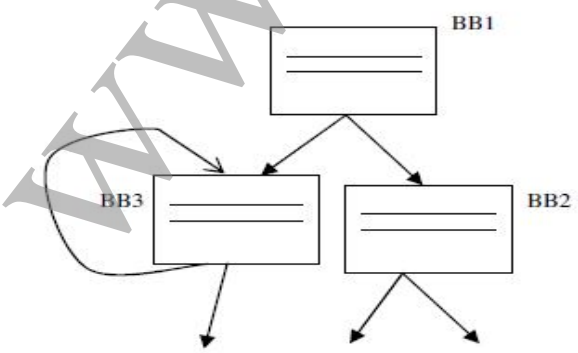
Factors of both programs and processors limit the amount of parallelism that can be exploited among instructions and these limit the performance achievable. The performance of the pipelined processors is given by:

Pipeline CPI= Ideal Pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

By reducing each of the terms on the right hand side, it is possible to minimize the overall pipeline CPI.

To exploit the ILP, the primary focus is on Basic Block (BB). The BB is a straight line code sequence with no branches in except the entry and no branches out except at the exit. The average size of the BB is very small i.e., about 4 to 6 instructions. The flow diagram segment of a program is shown below (Figure 3.1). BB1 , BB2 and BB3 are the Basic Blocks.

Figure 3.1 Flow diagram segment



The amount of overlap that can be exploited within a Basic Block is likely to be less than the average size of BB. To further enhance ILP, it is possible to look at ILP across multiple BB. The simplest and most common way to increase the ILP is to exploit the parallelism among iterations of a loop (Loop level parallelism). Each iteration of a loop can overlap with any other iteration.

Data Dependency and Hazard

If two instructions are parallel, they can execute simultaneously in a pipeline of arbitrary length without causing any stalls, assuming the pipeline has sufficient resources. If two instructions are dependent, they are not parallel and must be executed in sequential order.
There are three different types dependences.

- Data Dependences (True Data Dependency)
- Name Dependences
- Control Dependences

Data Dependences

An instruction j is data dependant on instruction i if either of the following holds:
i) Instruction i produces a result that may be used by instruction j

Eg1: i: L.D **F0**, 0(R1)
j: ADD.D F4, **F0**, F2

ith instruction is loading the data into the F0 and jth instruction use F0 as one the operand. Hence, jth instruction is data dependant on ith instruction.

Eg2: DADD **R1**, R2, R3
DSUB R4, **R1**, R5

ii) Instruction j is data dependant on instruction k and instruction k data dependant on instruction i

Eg: L.D **F4**, 0(R1)
MUL.D **F0**, **F4**, F6
ADD.D F5, **F0**, F7

Dependences are the property of the programs. A Data value may flow between instructions either through registers or through memory locations. Detecting the data flow and dependence that occurs through registers is quite straight forward. Dependences that flow through the memory locations are more difficult to detect. A data dependence convey three things.

a) The possibility of the Hazard.
b) The order in which results must be calculated and
c) An upper bound on how much parallelism can possibly exploited.

Name Dependences

A Name Dependence occurs when two instructions use the same Register or Memory location, but there is no flow of data between the instructions associated with that name.

Two types of Name dependences:

i) **Antidependence**: between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. he original ordering must be preserved to ensure that i reads the correct value.

Eg: L.D F0, 0(R1)
DADDUI R1, R1, R3

ii) **Output dependence:** Output Dependence occurs when instructions i and j write to the same register or memory location.

Ex: ADD.D **F4**, F0, F2
SUB.D **F4**, F3, F5

The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.The above instruction can be reordered or can be executed simultaneously if the name of the register is changed. The renaming can be easily done either statically by a compiler or dynamically by the hardware.

Data hazard: Hazards are named by the ordering in the program that must be preserved by the pipeline

RAW (Read After Write): j tries to read a source before i writes it, so j in correctly gets old value, this hazard is due to true data dependence.

WAW (Write After Write): j tries to write an operand before it is written by i. WAW hazard arises from output dependence.

WAR (Write After Read): j tries to write a destination before it is read by i, so that I incorrectly gets the new value. WAR hazard arises from an antidependence and normally cannot occur in static issue pipeline.

CONTROL DEPENDENCE:
A control dependence determines the ordering of an instruction i with respect to a branch instruction,

```
Ex: if P1 {
      S1;
    }
    if P2 {
    S2;
    }
```

S1 is Control dependent on P1 and

S2 is control dependent on P2 but not on P1.
a)An instruction that is control dependent on a branch cannot be moved before the branch ,so that its execution is no longer controlled by the branch.
b)An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

BASIC PIPELINE SCHEDULE AND LOOP UNROLLING

To keep a pipe line full, parallelism among instructions must be exploited by finding sequence of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall,a dependent instruction must be separated from the source instruction by the distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline.

The compiler can increase the amount of available ILP by transferring loops.

```
for(i=1000; i>0 ;i=i-1)
X[i] = X[i] + s;
```

We see that this loop is parallel by the noticing that body of the each iteration is independent.

The first step is to translate the above segment to MIPS assembly language

```
Loop: L.D F0, 0(R1) : F0=array element
ADD.D F4, F0, F2 : add scalar in F2
S.D F4, 0(R1) : store result
DADDUI R1, R1, #-8 : decrement pointer
: 8 Bytes (per DW)
BNE R1, R2, Loop : branch R1! = R2
```

Without any Scheduling the loop will execute as follows and takes 9 cycles for each iteration.

```
1 Loop: L.D F0, 0(R1) ;F0=vector element
2 stall
3 ADD.D F4, F0, F2 ;add scalar in F2
4 stall
5 stall
6 S.D F4, 0(R1) ;store result
7 DADDUI R1, R1,# -8 ;decrement pointer 8B (DW)
8 stall ;assumes can't forward to branch
        9 BNEZ R1, Loop ;branch R1!=zero
```

We can schedule the loop to obtain only two stalls and reduce the time to 7 cycles:

```
L.D F0, 0(R1)

DADDUI R1, R1, #-8
```

ADD.D F4, F0, F2

Stall

Stall

S.D F4, 0(R1)

BNE R1, R2, Loop

Loop Unrolling can be used to minimize the number of stalls. Unrolling the body of the loop by our times, the execution of four iteration can be done in 27 clock cycles or 6.75 clock cycles per iteration.

1 Loop: L.D F0,0(R1)

3 ADD.D F4,F0,F2

6 S.D 0(R1),F4 ;drop DSUBUI & BNEZ

7 L.D F6,-8(R1)

9 ADD.D F8,F6,F2

12 S.D -8(R1),F8 ;drop DSUBUI & BNEZ

13 L.D F10,-16(R1)

15 ADD.D F12,F10,F2

18 S.D -16(R1),F12 ;drop DSUBUI & BNEZ

19 L.D F14,-24(R1)

21 ADD.D F16,F14,F2

24 S.D -24(R1),F16

25 DADDUI R1,R1,#-32 :alter to 4*8

26 BNEZ R1,LOOP

Unrolled loop that minimizes the stalls to 14 clock cycles for four iterations is given below:

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 0(R1), F4

10 S.D -8(R1), F8

11 S.D -16(R1), F12

12 DSUBUI R1, R1,#32

13 S.D 8(R1), F16 ;8-32 = -24

14 BNEZ R1, LOOP

Summary of Loop unrolling and scheduling

The loop unrolling requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)

2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code

4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent

- Transformation requires analyzing memory addresses and finding that they do not refer to the same address

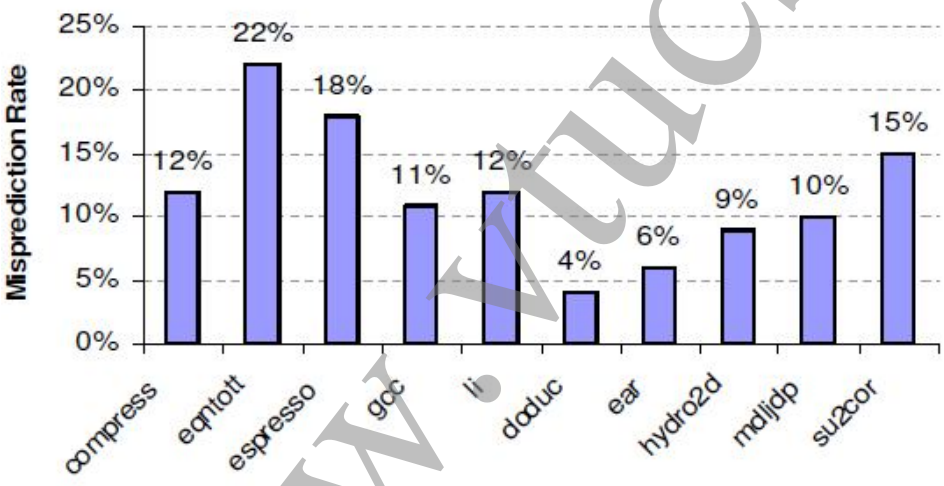
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

To reduce the Branch cost, prediction of the outcome of the branch may be done. The prediction may be done statically at compile time using compiler support or dynamically using hardware support. Schemes to reduce the impact of control hazard are discussed below:

Static Branch Prediction

Assume that the branch will not be ***taken*** and continue execution down the sequential instruction stream. If the branch is ***taken***, the instruction that are being fetched and decoded must be discarded. Execution continues at the branch target. Discarding instructions means we must be able to flush instructions in the IF, ID and EXE stages.
Alternately, it is possible that the branch can be predicted as taken. As soon as the instruction decoded is found as branch, at the earliest, start fetching the instruction from the target address.

- **Average misprediction = untaken branch frequency = 34% for SPEC pgms.**



The graph shows the misprediction rate for set of SPEC benchmark programs

Dynamic Branch Prediction

With deeper pipelines the branch penalty increases when measured in clock cycles. Similarly, with multiple issue, the branch penalty increases in terms of instructions lost. Hence, a simple static prediction scheme is inefficient or may not be efficient in most of the situations. One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and if so, to begin fetching new instruction from the target address.

This technique is called ***Dynamic branch prediction***.

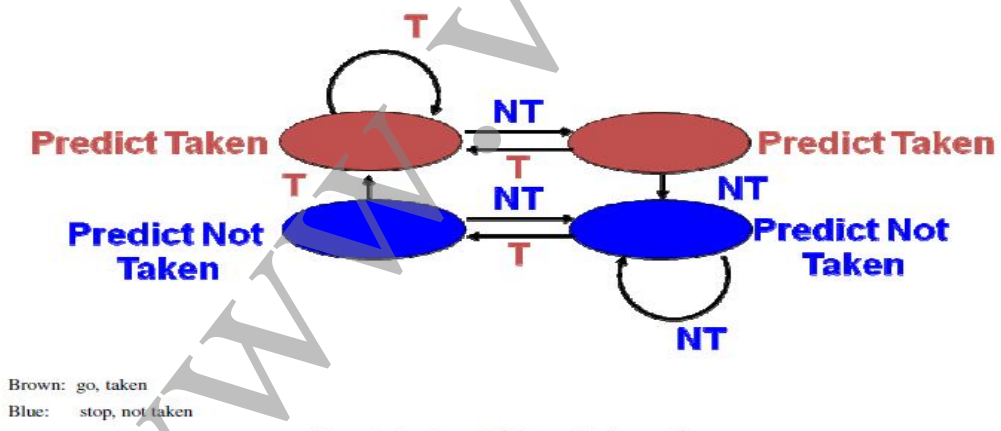
• Why does prediction work?

– Underlying algorithm has regularities
– Data that is being operated on has regularities
– Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
– There are a small number of important branches in programs which have dynamic behavior for which dynamic branch prediction performance will be definitely better compared to static branch prediction.

• Performance = *f*(accuracy, cost of misprediction)
• Branch History Table (BHT) is used to dynamically predict the outcome of the current branch instruction. Lower bits of PC address index table of 1-bit values
– Says whether or not branch taken last time
o - No address check

• Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 iterations before exit):
– End of loop case, when it exits instead of looping as before
– First time through loop on *next* time through code, when it predicts exit instead of looping

• Simple two bit history table will give better performance. The four different states of 2 bit predictor is shown in the state transition diagram.



The states in a 2-bit prediction scheme

Correlating Branch Predictor

It may be possible to improve the prediction accuracy by considering the recent behavior of other branches rather than just the branch under consideration. Correlating predictors are two-level predictors. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

• Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table (BHT)

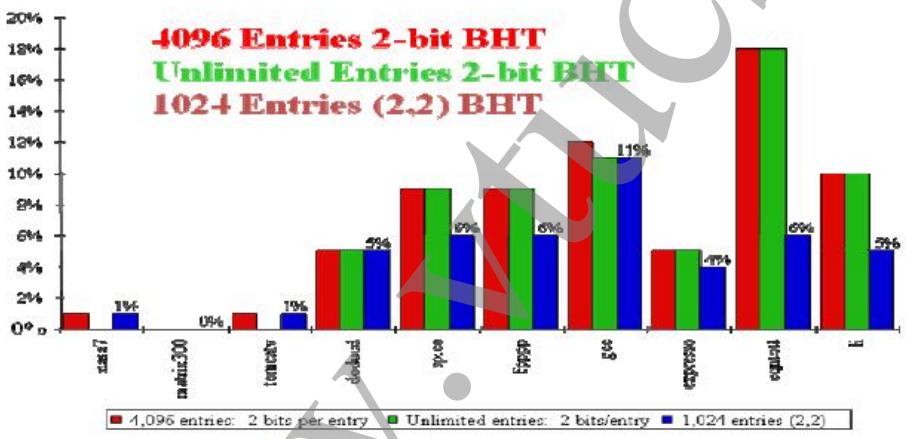
• In general, (*m*,*n*) predictor means record last *m* branches to select between 2m history tables, each with *n*-bit counters

– Thus, old 2-bit BHT is a (0,2) predictor

– Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.

• Each entry in table has *m* *n*-bit predictors. In case of (2,2) predictor, behavior of recent branches selects between four predictions of next branch, updating just that prediction. The scheme of the table is shown:

Comparisons of different schemes are shown in the graph.



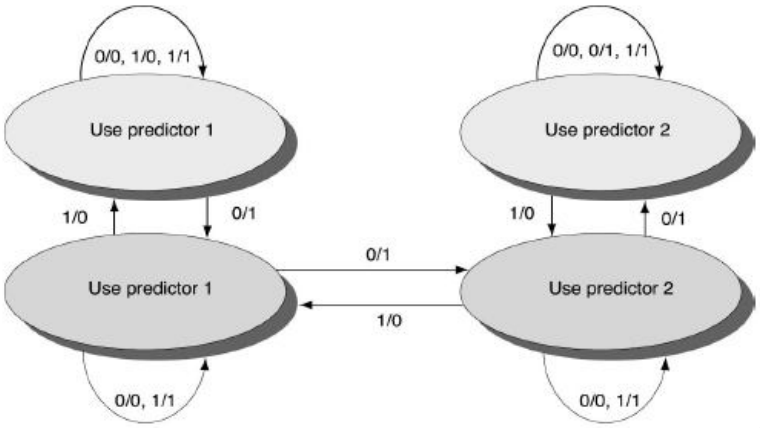
Comparison of 2 bit predictors (y-axis: % frequency of mispredictions, x-axis: SPEC89 programs)

Tournament predictor is a multi level branch predictor and uses n bit saturating counter to chose between predictors. The predictors used are global predictor and local predictor.

– Advantage of tournament predictor is ability to select the right predictor for a particular branch which is particularly crucial for integer benchmarks.

– A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks

– Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective oin recent prediction.



Dynamic Branch Prediction Summary

• Prediction is becoming important part of execution as it improves the performance of the pipeline.

• Branch History Table: 2 bits for loop accuracy

• Correlation: Recently executed branches correlated with next branch
  – Either different branches (GA)
  – Or different executions of same branches (PA)
• Tournament predictors take insight to next level, by using multiple predictors
  – usually one based on global information and one based on local information, and combining them with a selector
  – In 2006, tournament predictors using » 30K bits are in processors like the Power and Pentium 4

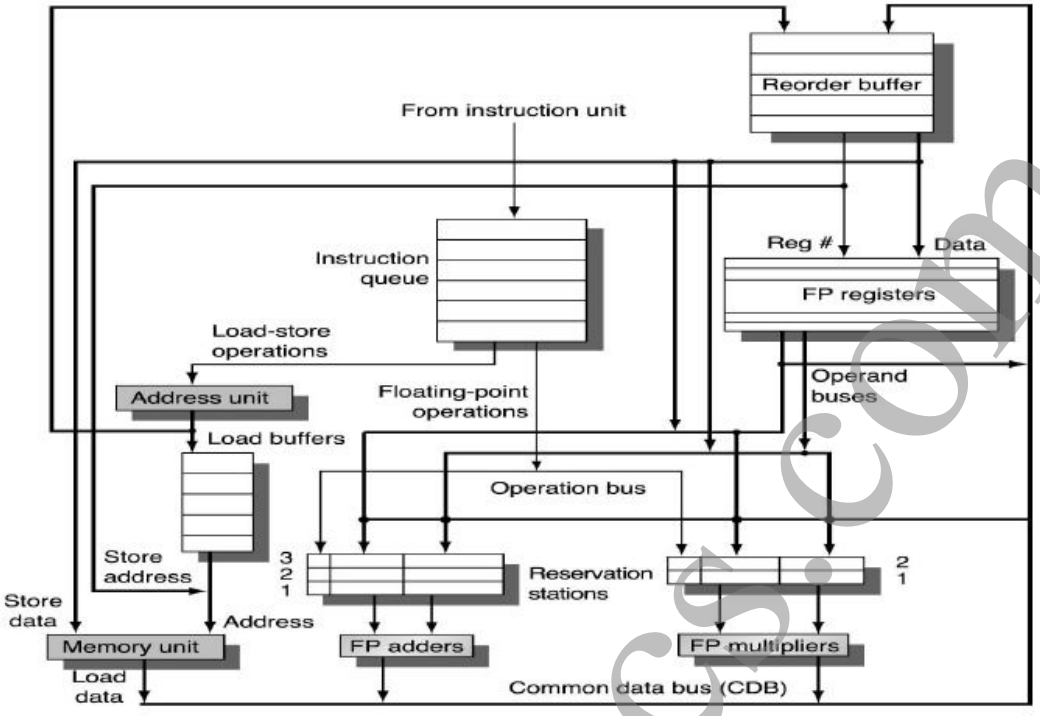
Tomasulu algorithm and Reorder Buffer

Tomasulu idea:

1. Have reservation stations where register renaming is possible
2. Results are directly forwarded to the reservation station along with the final registers. This is also called short circuiting or bypassing.

ROB:

1.The instructions are stored sequentially but we have indicators to say if it is speculative or completed execution.
2. If completed execution and not speculative and reached head of the queue then we commit it.



Speculating on Branch Outcomes

• To optimally exploit ILP (instruction-level parallelism) – e.g. with pipelining, Tomasulo,etc. – it is critical to efficiently maintain *control dependencies* (=branch dependencies)

• Key idea: *Speculate* on the outcome of branches(=predict) and execute instructions *as if* the predictions are correct

• of course, we must proceed in such a manner as to be able to recover if our speculation turns out wrong

Three components of hardware-based speculation

1. *dynamic branch prediction* to pick branch outcome
2. *speculation* to allow instructions to execute before control dependencies are resolved, i.e., before branch outcomes become known – with ability to undo in case of incorrect speculation
3. *dynamic scheduling*

Speculating with Tomasulo
Key ideas:

1. separate execution from completion: instructions to execute speculatively but no *instructions update registers or memory until no more speculative*

2. therefore, add a final step – after an instruction is no longer speculative, called *instruction commit*– when it is allowed to make register and memory updates

3. allow instructions to execute and complete out of order but force them to commit in order

4. Add hardware called the *reorder buffer* (*ROB*), with registers to hold the result of an instruction *between completion and commit*

Tomasulo's Algorithm with Speculation: Four Stages

1. Issue: get instruction from Instruction Queue
   _ if reservation station and ROB slot free (no structural hazard),
   control issues instruction to reservation station and ROB, and sends to reservation station operand values (or reservation station source for values) as well as allocated ROB slot number

2. Execution: operate on operands (EX)
_ when both operands ready then execute;if not ready, watch CDB for result

3. Write result: finish execution (WB)
_ write on CDB to all awaiting units and ROB; mark reservation station available

4. Commit: update register or memory with ROB result
   _ when instruction reaches head of ROB and results present, update register with result or store to memory and remove instruction from ROB
   _ if an incorrectly predicted branch reaches the head of ROB, flush the ROB, and restart at correct successor of branch

ROB Data Structure

ROB entry fields
• Instruction type: branch, store, register operation (i.e., ALU or load)
• State: indicates if instruction has completed and value is ready
• Destination: where result is to be written – register number for register operation (i.e. ALU or load), memory address for store
• branch has no destination result
Value: holds the value of instruction result till time to commit
Additional reservation station field
• Destination: Corresponding ROB entry number
   Example
   1. L.D F6, 34(R2)

2. L.D F2, 45(R3

3. MUL.D F0, F2, F4

4. SUB.D F8, F2, F6

5. DIV.D F10, F0, F6

6. ADD.D F6, F8, F2

The position of Reservation stations, ROB and FP registers are indicated below:

Assume latencies load 1 clock, add 2 clocks, multiply 10 clocks, divide 40 clocks
Show data structures just before MUL.D goes to commit…

Reservation Stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV		Mem[34+Regs[R2]]	#3		#5	

At the time MUL.D is ready to commit only the two L.D instructions have already committed,though others have completed execution
Actually, the MUL.D is at the head of the ROB – the L.D instructions are shown only for understanding purposes #X represents value field of ROB entry number X

Floating point registers

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder#	3						6		4	5
Busy	yes	no	no	no	no	no	yes	…	yes	yes

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F6, 34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	no	L.D F2, 45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	yes	MUL.D F0, F2, F4	Write result	F0	#2 × Regs[F4]
4	yes	SUB.D F8, F6, F2	Write result	F8	#1 – #2
5	yes	DIV.D F10, F0, F6	Execute	F10	
6	yes	ADD.D F6, F8, F2	Write result	F6	#4 + #2

Example

```
Loop: LD    F0    0    R1
MULTD       F4    F0   F2
SD          F4    0    R1
SUBI        R1    R1   #8
BNEZ        R1    Loop
```

Assume instructions in the loop have been issued twice
Assume L.D and MUL.D from the first iteration have committed and all other instructions have completed
Assume effective address for store is computed prior to its issue
Show data structures

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
1	no	L.D F0, 0(R1)	Commit	F0	Mem[0+Regs[R1]]
2	no	MUL.D F4, F0, F2	Commit	F4	#1 × Regs[F2]
3	yes	S.D F4, 0(R1)	Write result	0 + Regs[R1]	#2
4	yes	DADDUI R1, R1, #-8	Write result	R1	Regs[R1] – 8
5	yes	BNE R1, R2, Loop	Write result		
6	yes	L.D F0, 0(R1)	Write result	F0	Mem[#4]
7	yes	MUL.D F4, F0, F2	Write result	F4	#6 × Regs[F2]
8	yes	S.D F4, 0(R1)	Write result	0 + #4	#7
9	yes	DADDUI R1, R1, #-8	Write result	R1	#4 – 8
10	yes	BNE R1, R2, Loop	Write result		

Notes

• If a branch is mispredicted, recovery is done by flushing the ROB of all entries that appear after the mispredicted branch

- entries before the branch are allowed to continue
- restart the fetch at the correct branch successor

• When an instruction commits or is flushed from the ROB then the corresponding slots become available for subsequent instructions

Advantages of hardware-based speculation:

- -able to disambiguate memory references;
- -better when hardware-based branch prediction is better than software-based branch
- prediction done at compile time; - maintains a completely precise exception model even for speculated instructions;
- does not require compensation or bookkeeping code;
  main disadvantage:
- complex and requires substantial hardware resources;

UNIT - IV

INSTRUCTION –LEVEL PARALLELISM – 2:

Exploiting ILP using multiple issue and static scheduling

Exploiting ILP using dynamic scheduling

Multiple issue and speculation

Advanced Techniques for instruction delivery and Speculation

The Intel Pentium 4 as example. **7 Hours**

UNIT IV

INSTRUCTION –LEVEL PARALLELISM – 2

What is ILP?

• Instruction Level Parallelism
  – Number of operations (instructions) that can be performed in parallel
• Formally, two instructions are parallel if they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls assuming that the pipeline has sufficient resources
  – Primary techniques used to exploit ILP
• Deep pipelines
• Multiple issue machines
• Basic program blocks tend to have 4-8 instructions between branches
  – Little ILP within these blocks
  – Must find ILP between groups of blocks

Example Instruction Sequences

• Independent instruction sequence:

```
lw $10, 12($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
```

• Dependent instruction sequence:

```
lw $10, 12($1)
sub $11, $2, $10
and $12, $11, $10
or $13, $6, $7
add $14, $8, $13
```

Finding ILP:

• Must deal with groups of basic code blocks
• Common approach: loop-level parallelism
– Example:

– In MIPS (assume $s0 initialized properly):

for (i=1000; i > 0; i--)
x[i] = x[i] + s;
Loop: lw $t0, 0($s1) # t0 = array element
addu $t0, $t0, $s2 # add scalar in $s2
sw $t0, 0($s1) # store result
addi $s1, $s1, -4 # decrement pointer
bne $s1, $0, Loop # branch $s1 != 0

Loop Unrolling:

• Technique used to help scheduling (and performance)
• Copy the loop body and schedule instructions from different iterations of the loop gether
• MIPS example (from prev. slide):

Loop: lw $t0, 0($s1) # t0 = array element
addu $t0, $t0, $s2 # add scalar in $s2
sw $t0, 0($s1) # store result
lw $t1, -4($s1)
addu $t1, $t1, $s2
sw $t1, -4($s1)
addi $s1, $s1, -8 # decrement pointer
bne $s1, $0, Loop # branch $s1 != 0

Note the new register & counter adjustment!
• Previous example, we unrolled the loop once
– This gave us a second copy
• Why introduce a new register ($t1)?
– Antidependence (name dependence)
• Loop iterations would reuse register $t0
• No data overlap between loop iterations!
• Compiler *RENAMED* the register to prevent a "dependence"
– Allows for better instruction scheduling and identification of true dependencies
• In general, you can unroll the loop as much as you want
– A factor of the loop counter is generally used
– Limited advantages to unrolling more than a few times

Loop Unrolling: Performance:

• Performance (dis)advantage of unrolling
– Assume basic 5-stage pipeline
• Recall lw requires a bubble if value used immediately after
• For original loop
– 10 cycles to execute first iteration
– 16 cycles to execute two iterations

• Assuming perfect prediction
• For unrolled loop
– 14 cycles to execute first iteration -- without reordering
• Gain from skipping addi, bne
– 12 cycles to execute first iteration -- with reordering
• Put lw together, avoid bubbles after ea

Loop Unrolling: Limitations

• Overhead amortization decreases as loop is unrolled more
• Increase in code size
– Could be bad if ICache miss rate increases
• Register pressure
– Run out of registers that can be used in renaming process
–

Exploiting ILP: Deep Pipelines

Deep Pipelines
• Increase pipeline depth beyond 5 stages
– Generally allows for higher clock rates
– UltraSparc III -- 14 stages
– Pentium III -- 12 stages
– Pentium IV -- 22 stages
• Some versions have almost 30 stages
– Core 2 Duo -- 14 stages
– AMD Athlon -- 9 stages
– AMD Opteron -- 12 stages
– Motorola G4e -- 7 stages
– IBM PowerPC 970 (G5) -- 14 stages
• Increases the number of instructions executing at the same time
• Most of the CPUs listed above also issue multiple instructions per cycle

Issues with Deep Pipelines

• Branch (Mis-)prediction
– Speculation: Guess the outcome of an instruction to remove it as a dependence to other instructions
– Tens to hundreds of instructions “in flight”
– Have to flush some/all if a branch is mispredicted
• Memory latencies/configurations
– To keep latencies reasonable at high clock rates, need fast caches
– Generally smaller caches are faster
– Smaller caches have lower hit rates
• Techniques like way prediction and prefetching can help lower latencies

Optimal Pipelining Depths

• Several papers published on this topic
– Esp. the 29th International Symposium on Computer Architecture (ISCA)
– Intel had one pushing the depth to 50 stages

– Others have shown ranges between 15 and 40
– Most of the variation is due to the intended workload

Exploiting ILP: Multiple Issue Computers

Multiple Issue Computers

• **Benefit**
– CPIs go below one, use IPC instead (instructions/cycle)
– Example: Issue width = 3 instructions, Clock = 3GHz
• Peak rate: 9 billion instructions/second, IPC = 3
• For our 5 stage pipeline, 15 instructions "in flight" at any given time
• Multiple Issue types
– Static
• Most instruction scheduling is done by the compiler
– Dynamic (superscalar)
• CPU makes most of the scheduling decisions
• Challenge: overcoming instruction dependencies
– Increased latency for loads
– Control hazards become worse
• Requires a more ambitious design
– Compiler techniques for scheduling
– Complex instruction decoding logic

Exploiting ILP:Multiple Issue Computers Static Scheduling

Instruction Issuing
• Have to decide which instruction types can issue in a cycle
– Issue packet: instructions issued in a single clock cycle
– Issue slot: portion of an issue packet
• Compiler assumes a large responsibility for hazard checking, scheduling, etc.

Static Multiple Issue

For now, assume a "souped-up" 5-stage MIPS pipeline that can issue a packet with:
– One slot is an ALU or branch instruction
One slot is a load/store instruction

Instruction Type	Pipeline Stages						
ALU or Branch instruction	IF	ID	EX	MEM	WB		
Load or Store instruction	IF	ID	EX	MEM	WB		
ALU or Branch instruction		IF	ID	EX	MEM	WB	
Load or Store instruction		IF	ID	EX	MEM	WB	
ALU or Branch instruction			IF	ID	EX	MEM	WB
Load or Store instruction			IF	ID	EX	MEM	WB

–

Static Multiple Issue

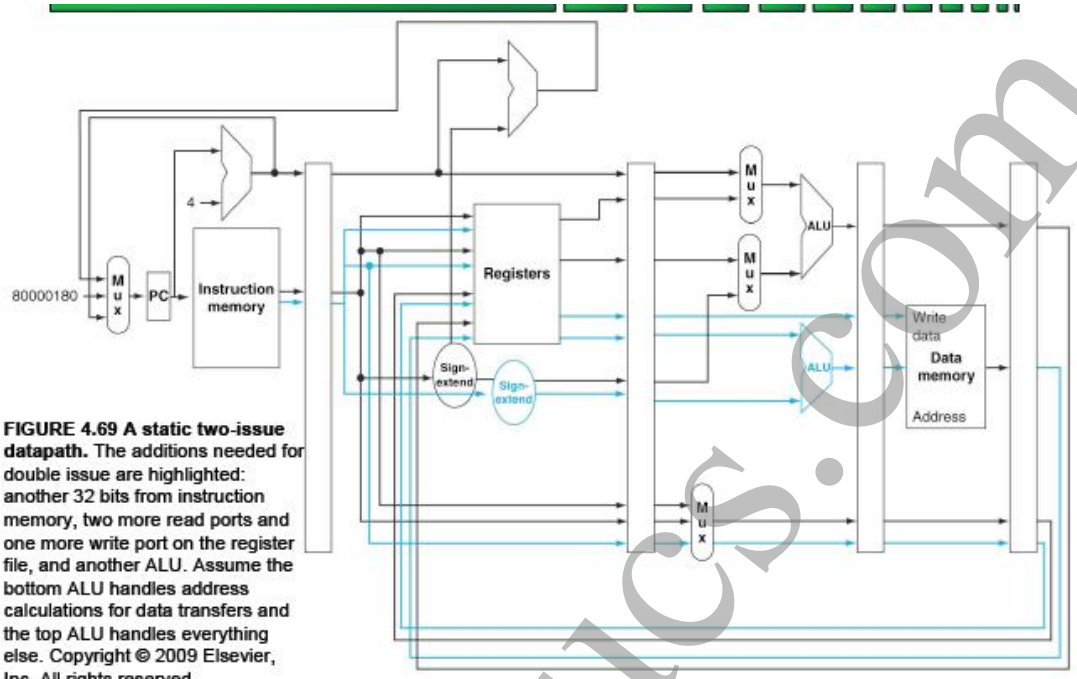


FIGURE 4.69 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Static Multiple Issue Scheduling

```
add  $1, $2, $3
add  $5, $2, $2
load $4, $3(100)
load $3, $2(100)
sub  $2, $5, $3
add $2, $2, $4
```

Becomes

Cycle	ALU/Branch Instruction	Load/Store Instruction
1		
2		
3		
4		
5		
6		

Static Mult. Issue w/Loop Unrolling

Original loop schedule for a 2-issue MIPS

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addi  $s1, $s1, -4
      bne   $s1, $0, Loop
```

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4

Unrolled (once) loop schedule for a 2-issue MIPS

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      lw    $t1, -4($s1)
      addu  $t1, $t1, $s2
      sw    $t1, -4($s1)
      addi  $s1, $s1, -8
      bne   $s1, $0, Loop
```

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4
			5
			6
			7

Static Mult. Issue w/Loop Unrolling

```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addi  $s1, $s1, -4
      bne   $s1, $0, Loop
```

```
Loop:
```

Unroll loop 3x, (4 iterations total) and schedule for a 2-issue MIPS

	ALU/Branch	Load/Store	Cycle
Loop:			1
			2
			3
			4
			5
			6
			7
			8
			9
			10
			11
			12

Exploiting ILP:Multiple Issue Computers Dynamic Scheduling

Dynamic Multiple Issue Computers

• Superscalar computers
• CPU generally manages instruction issuing and ordering
  – Compiler helps, but CPU dominates
• Process
  – Instructions issue in-order
  – Instructions can execute out-of-order
• Execute once all operands are ready
  – Instructions commit in-order
• Commit refers to when the architectural register file is updated (current completed state of program

Aside: Data Hazard Refresher

• Two instructions (i and j), j follows i in program order
• Read after Read (RAR)
• Read after Write (RAW)
  – Type:
  – Problem:
• Write after Read (WAR)
  – Type:

– Problem:

• Write after Write (WAW)

– Type: Problem:

Superscalar Processors

• Register Renaming

– Use more registers than are defined by the architecture

• Architectural registers: defined by ISA
• Physical registers: total registers
  – Help with name dependencies
• Antidependence
  – Write after Read hazard
• Output dependence
  – Write after Write hazard

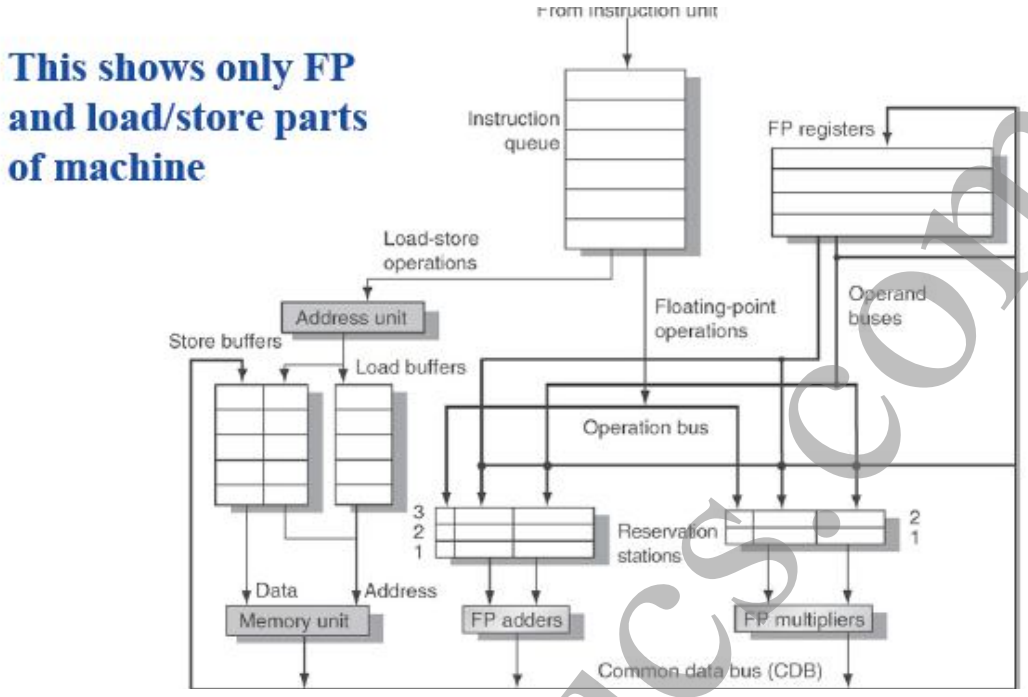
Tomasulo's Superscalar Computers

• R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM J. of Research and Development*, Jan. 1967
• See also: D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 model 91: Machine philosophy and instruction-handling," *IBM J. of Research and evelopment*, Jan. 1967
• Allows out-of-order execution
• Tracks when operands are available

– Minimizes RAW hazards

• Introduced renaming for WAW and WAR hazards

Tomasulo's Superscalar Computers



Instruction Execution Process

• Three parts, arbitrary number of cycles/part
• Above does not allow for speculative execution
• Issue (aka Dispatch)
– If empty reservation station (RS) that matches instruction, send to RS with operands rom register file and/or know which functional unit will send operand
  – If no empty RS, stall until one is available

Rename registers as appropriate
Instruction Execution Process
• Execute
– All branches before instruction must be resolved
• Preserves exception behavior
– When all operands available for an instruction, send it to functional unit
• Monitor common data bus (CDB) to see if result is needed by RS entry
– For non-load/store reservation stations
• If multiple instructions ready, have to pick one to send to functional unit
– For load/store
• Compute address, then place in buffer
• Loads can execute once memory is free
• Stores must wait for value to be stored, then execute

Write Back

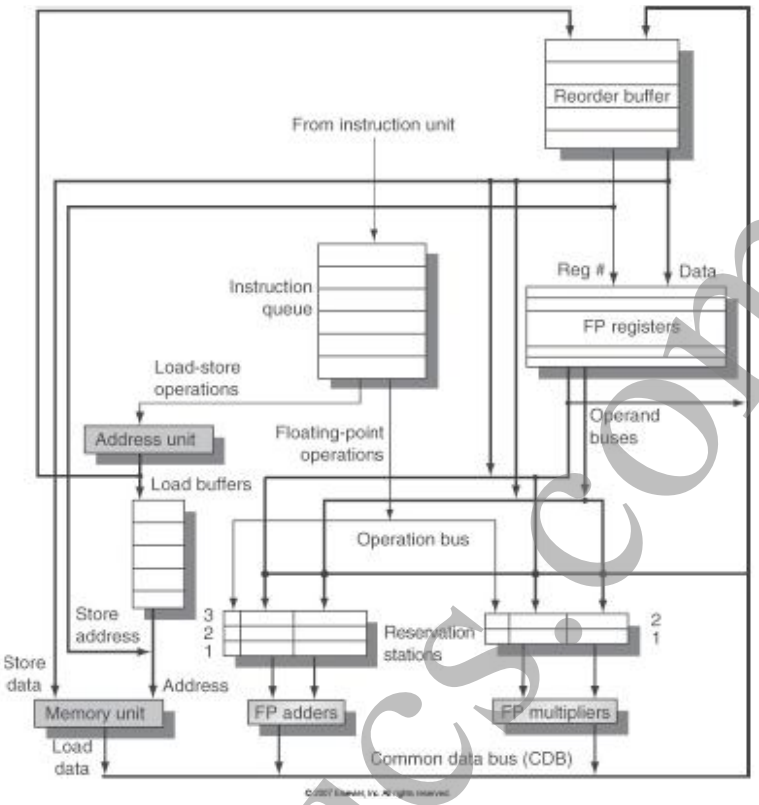
– Functional unit places on CDB
• Goes to both register file and reservation stations
– Use of CDB enables forwarding for RAW hazards
– Also introduces a latency between result and use of a value

Reservation Stations

• Require 7 fields
– Operation to perform on operands (2 operands)
– Tags showing which RS/Func. Unit will be producing operand (or zero if operand available/unnecessary)
– Two source operand values
– A field for holding memory address calculation data
• Initially, immediate field of instruction
• Later, effective address
– Busy
• Indicates that RS and its functional unit are busy
• Register file support
– Each entry contains a field that identifies which RS/func. unit will be writing into this entry (or blank/zero if noone will be writing to it) Limitation of Current Machine

Instruction execution requires branches to be resolved
• For wide-issue machines, may issue one branch per clock cycle!
• Desire:
– Predict branch direction to get more ILP
– Eliminate control dependencies
• Approach:
– Predict branches, utilize *speculative* instruction execution
  – Requires mechanisms for "fixing" machine when speculation is incorrect Tomasulo's w/Hardware Speculation

This shows only FP and load/store parts of machine



Tomasulo's w/HW Speculation

• Key aspects of this design
– Separate forwarding (result bypassing) from actual instruction completion
• Assuming instructions are executing speculatively
• Can pass results to later instructions, but prevents instruction from performing updates that can't be "undone"
– Once instruction is no longer speculative it can update register file/memory
• New step in execution sequence: instruction commit
• Requires instructions to wait until they can commit Commits still happen in order Reorder Buffer (ROB)

Instructions hang out here before committing
• Provides extra registers for RS/RegFile
– Is a source for operands
• Four fields/entry
– Instruction type
• Branch, store, or register operation (ALU & load)
– Destination field
• Register number or store address
– Value field
• Holds value to write to register or data for store
– Ready field
• Has instruction finished executing?

• Note: store buffers from previous version now in ROB

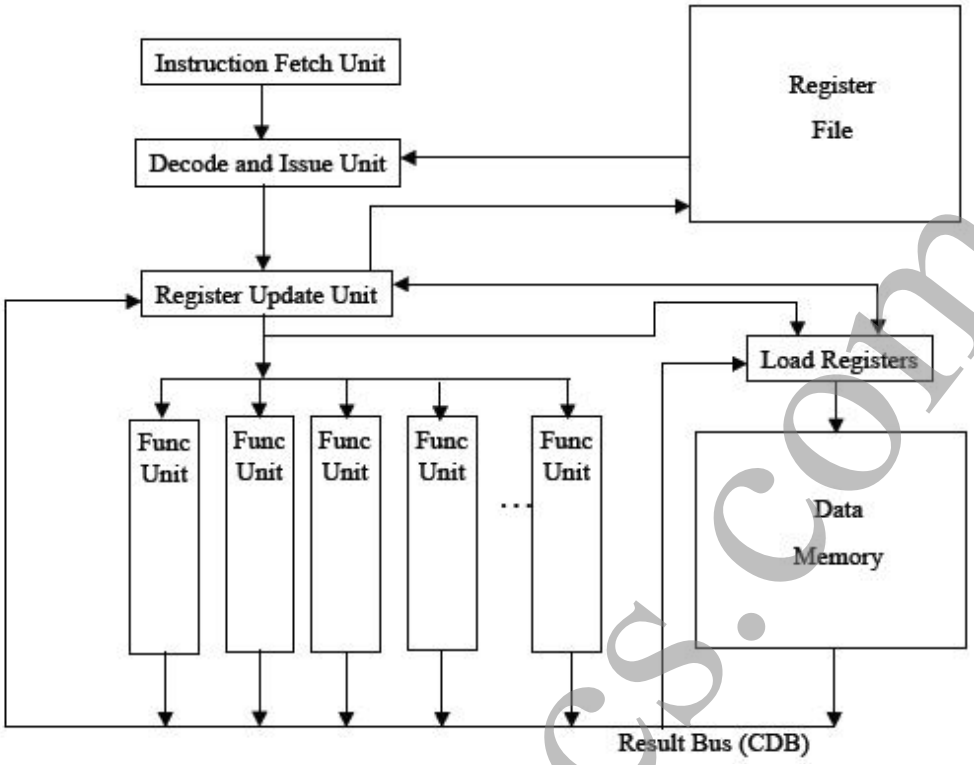
Instruction Execution Sequence

• Issue
– Issue instruction if opening in RS & ROB
– Send operands to RS from RegFile and/or ROB
• Execute
– Essentially the same as before
• Write Result
– Similar to before, but put result into ROB
• Commit (next slide)

Committing Instructions

Look at head of ROB
• Three types of instructions
– Incorrectly predicted branch
• Indicates speculation was wrong
• Flush ROB
• Execution restarts at proper location – Store
• Update memory
• Remove store from ROB
– Everything else
• Update registers
• Remove instruction from ROB

RUU Superscalar Computers

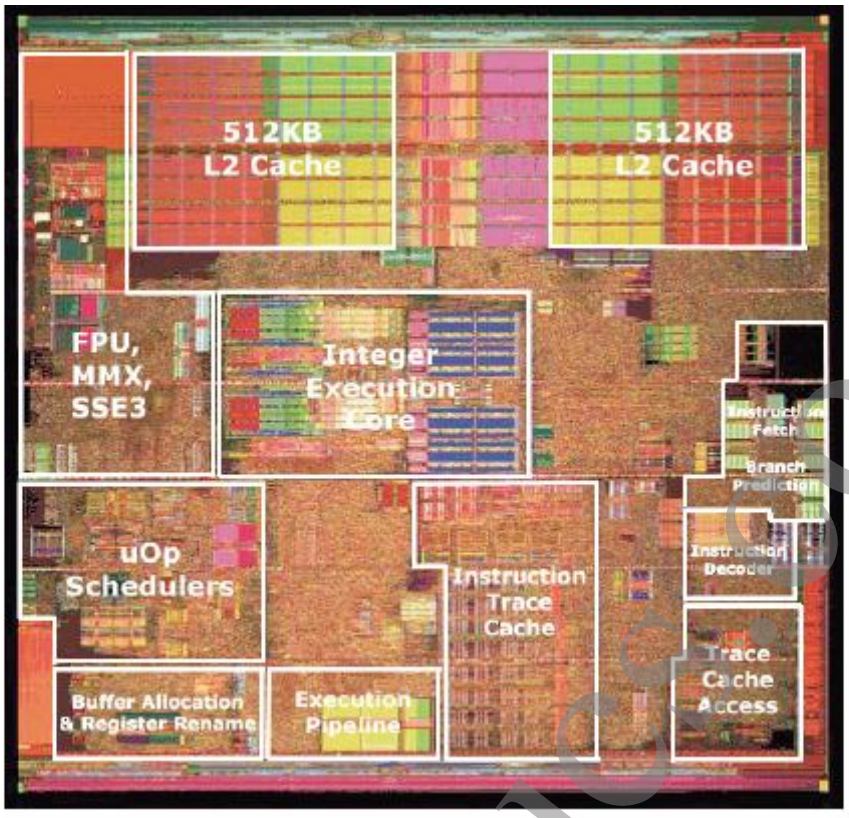


Modeling tool Simple Scalar implements an RUU style processor

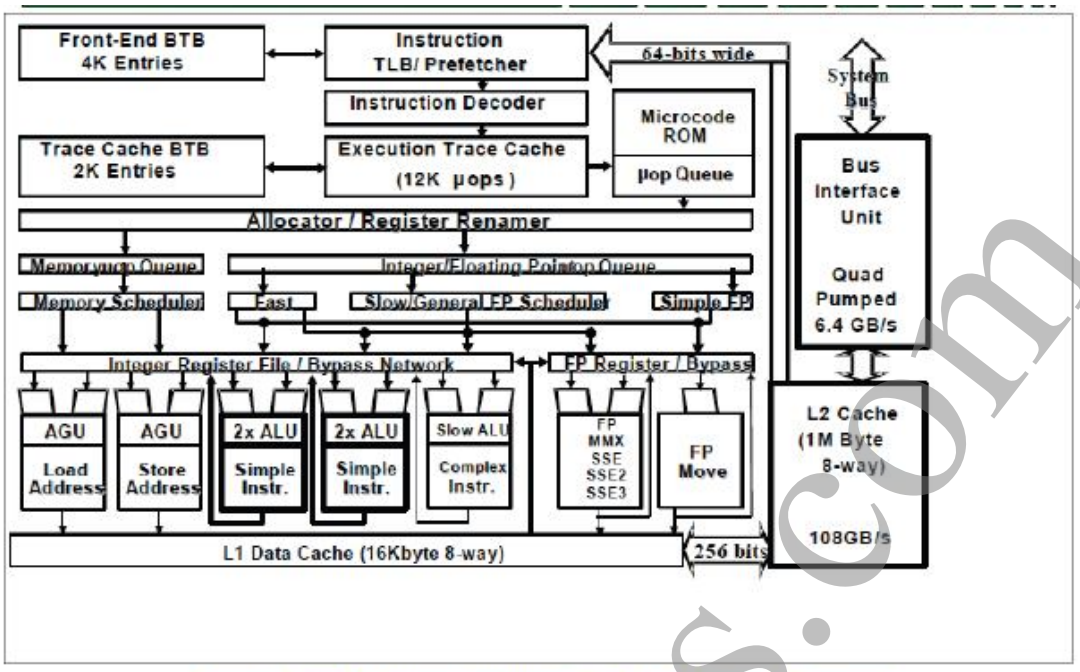
– You will be using this tool after Spring Break
• Architecture similar to speculative Tomasulo's
• Register Update Unit (RUU)
– Controls instructions scheduling and dispatching to functional units
– Stores intermediate source values for instructions
– Ensures instruction commit occurs in order!
– Needs to be of appropriate size
• Minimum of issue width * number of pipeline stages
• Too small of an RUU can be a structural hazard!
• Result bus could be a structural hazard

A Real Computer:Intel Pentium 4

Pentium 4 Die Photo



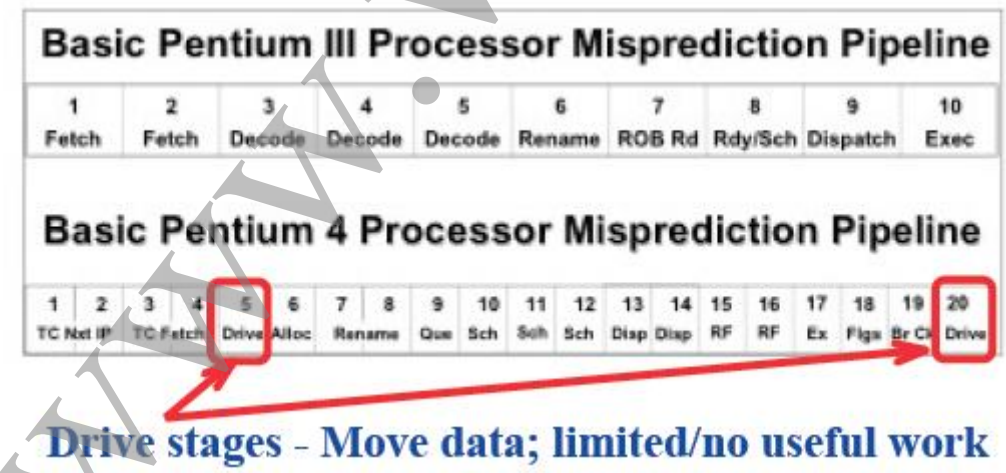
Overview of P4



Boggs et al, "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology," Intel Tech. J. Vol. 8, Num. 1, 2004

Pentium 4 Pipeline

• See handout for overview of major steps
• Prescott (90nm version of P4) had 31 pipeline stages
  – Not sure how pipeline is divided up
  –



P4: Trace Cache

Non-traditional instruction cache
• Recall x86 ISA

– CISC/VLIW: ugly assembly instructions of varying lengths
– Hard for HW to decode
– Ended up translating code into RISC-like microoperations to execute
• Trace Cache holds sequences of RISC-like micro-ops
– Less time decoding, more time executing
– Sequence storage similar to “normal” instruction cache

P4: Branch Handling

BTBs (Branch Target Buffers)
– Keep both branch history and branch target addresses
• Target address is instruction immediately after branch
– Predict if no entry in BTB for branch
• Static prediction
• If a backwards branch, see how far target is from current; if within a threshold, predict taken, else predict not taken
• If a forward branch, predict not taken
• Also some other rules
• Front-end BTB is L2 (like) for the trace cache BTB (L1 like)

P4: Execution Core

• Tomasulo’s algorithm-like
• Can have up to 126 instructions in-flight
– Max of 3 micro-ops sent to core/cycle
– Max of 48 loads, 32 stores
• Send up to 6 instructions to functional units per cycle via 4 ports
– Port 0: Shared between first fast ALU and FP/Media move scheduler
– Port 1: Shared between second fast ALU and Complex integer and FP/Media scheduler
– Port 2: Load
Port 3: Store

P4: Rapid Execution Engine
Execute 6 micro-ops/cycle
– Simple ALUs run at 2x machine clock rate
– Can generate 4 simple ALU results/cycle
– Do one load and one store per cycle
• Loads involve data speculation
– Assume that most loads hit L1 and Data Translation Look-aside Buffer (DTLB)
– Get data into execution, while doing address check
• Fix if L1 miss occurred

P4: Memory Tricks

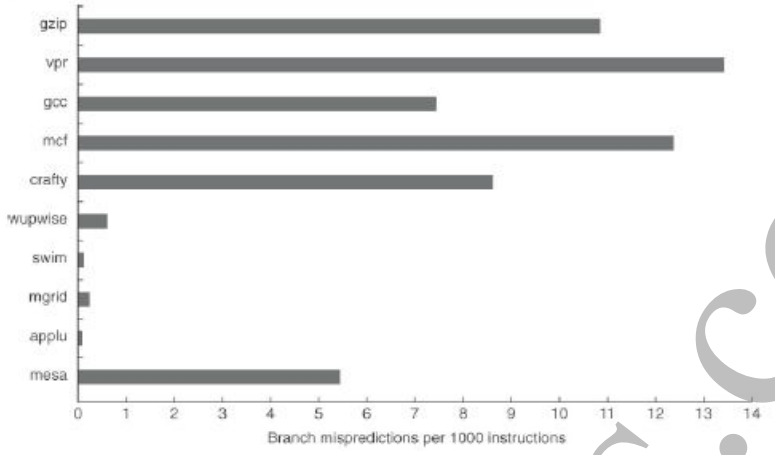
• Store-to-Load Forwarding
– Stores must wait to write until non-speculative
– Loads occasionally want data from store location
– Check both cache and Store Forwarding Buffer
• SFB is where stores are waiting to be written
– If hit when comparing load address to SFB address, use SFB data, not cache data
• Done on a partial address
• Memory Ordering Buffer
– Ensures that store-to-load forwarding was correct
• If not, must re-execute load
– Force forwarding
• Mechanism for forwarding in case addresses are misaligned
• MOB can tell SFB to forward or not
– False forwarding
• Fixes partial address match between load and SFB

P4: Specs for Rest of Slides

• For one running at 3.2 GHz
– From grad arch book
• L1 Cache
– Int: Load to use - 4 cycles
– FP: Load to use - 12 cycles
– Can handle up to 8 outstanding load misses
• L2 Cache (2 MB)
18 cycle access time

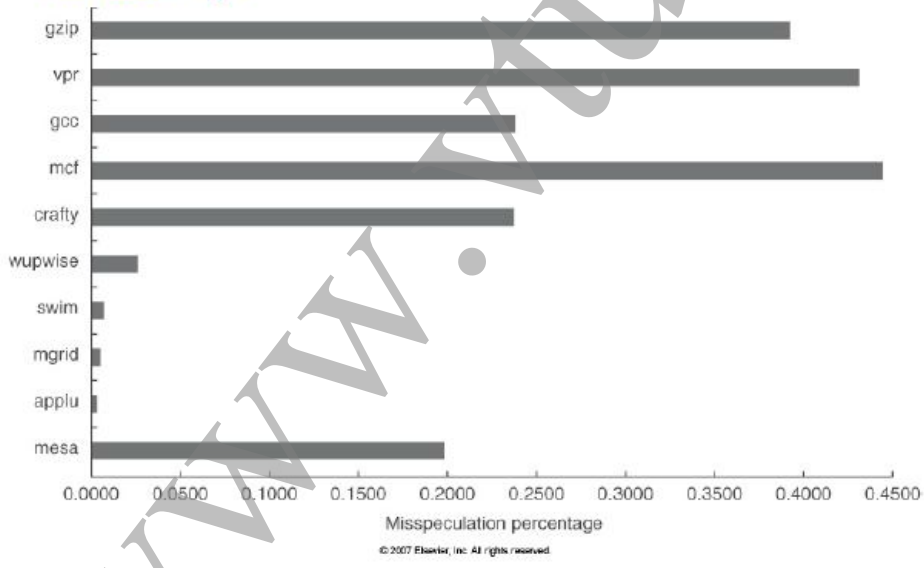
P4: Branch Prediction

- **Graph results from subset of SPEC 2000 Benchmarks**
  - **Integer: gzip, vpr, gcc, mcf, crafty**
    - **168 branches/1000 instructions**
  - **FP: wupwise, swim, mgrid, applu, mesa**
    - **48 branches/1000 instructions**



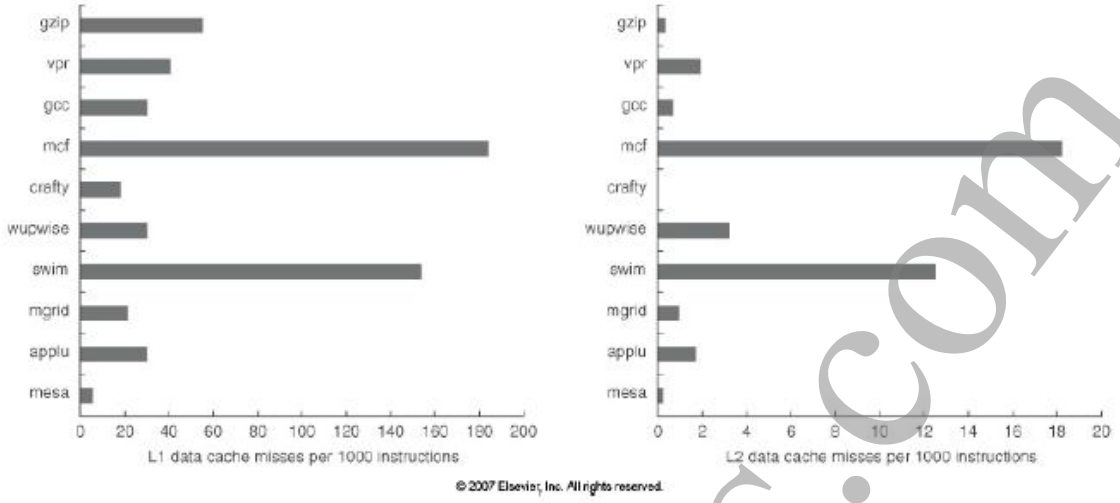
P4: Misspeculation Percentages

- **For micro-ops**



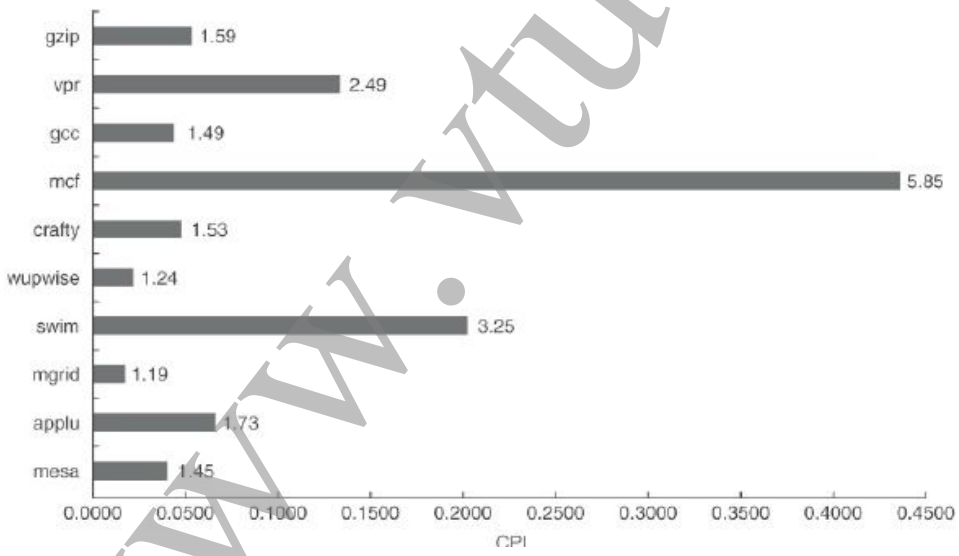
P4: Data Cache Miss Rates

- **This L2 is 2MB, not 1MB (as in paper you read)**
- **Note scale is 10x for L1 as compared to L2**



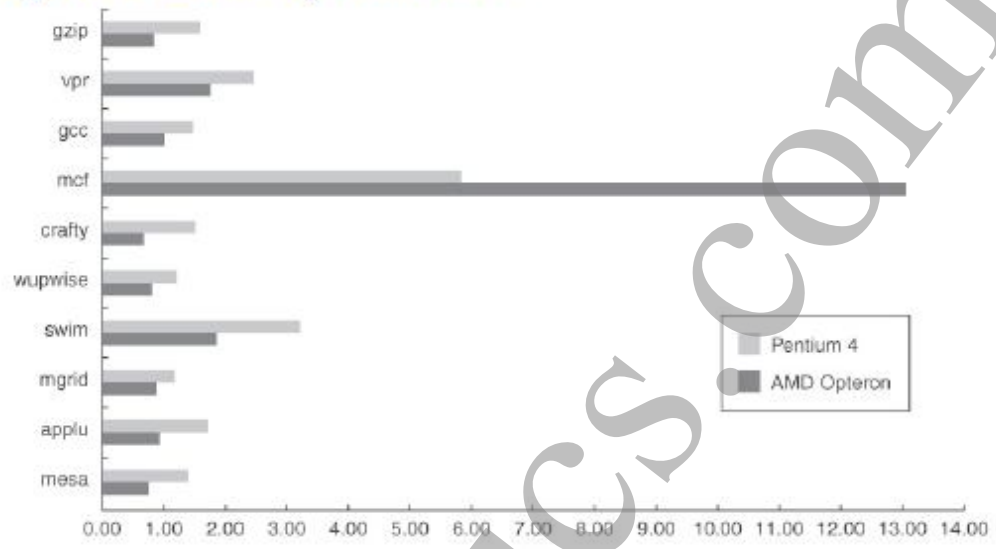
P4: CPI

- **Read values from lines, not sure why X-axis is scaled like it is**
- **1.29 micro-ops per IA-32 instruction**



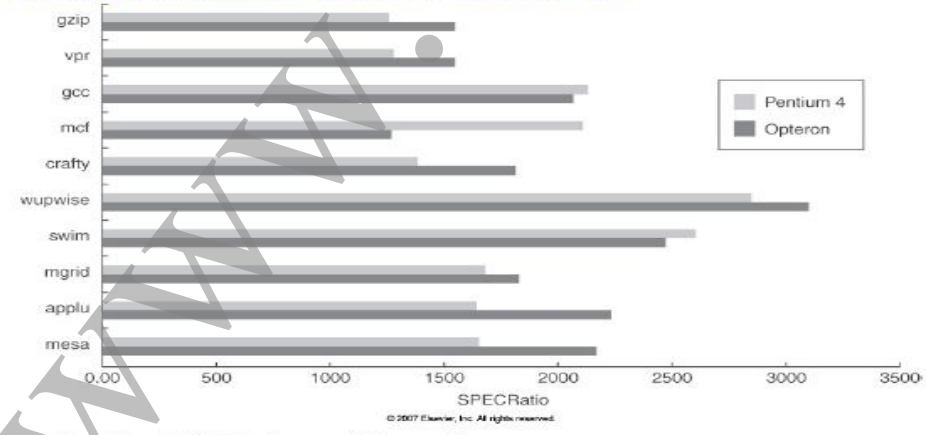
P4 vs. AMD Opteron

- **Architecturally similar**
  - **Opteron pipeline much shorter (12 stages)**
  - **P4 seems to have a larger cache**
- **CPI comparison below (Opteron at 2.6GHz)**
  - **Opteron CPI lower by factor of 1.27**



P4 vs. Opteron: Real Performance

- **Clock rates (2005 comparison)**
  - **P4: 3.8 GHz**
  - **Opteron: 2.8 GHz**
- **Opteron performance advantage of about 1.08**



PART - B

UNIT - 5

MULTIPROCESSORS AND THREAD –LEVEL PARALLELISM:
Introduction

Symmetric shared-memory architectures

Performance of symmetric shared–memory multiprocessors

Distributed shared memory and directory-based coherence

Basics of synchronization

Models of Memory Consistency. **7 Hours**

UNIT V

Multiprocessors and Thread-Level Parallelism

We have seen the renewed interest in developing multiprocessors in early 2000:
- The slowdown in uniprocessor performance due to the diminishing returns in exploring instruction-level parallelism.
- Difficulty to dissipate the heat generated by uniprocessors with high clock rates.
- Demand for high-performance servers where thread-level parallelism is natural.

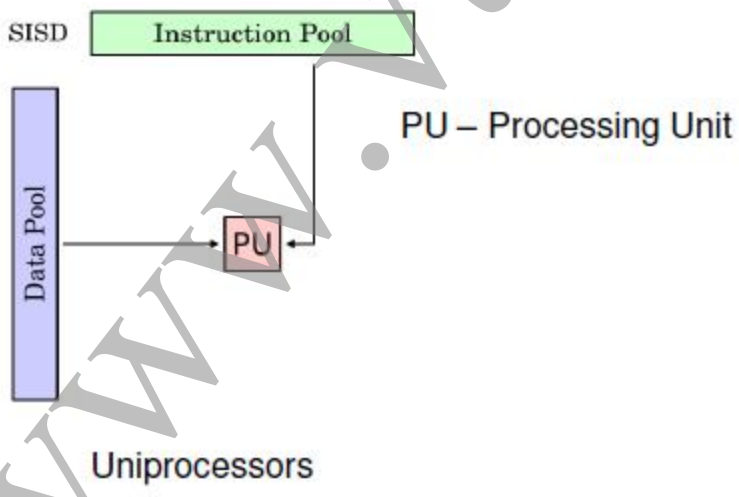
For all these reasons multiprocessor architectures has become increasingly attractive.

A Taxonomy of Parallel Architectures

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 30 years ago, Flynn proposed a simple model of categorizing all computers that is still useful today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers in one of four categories:

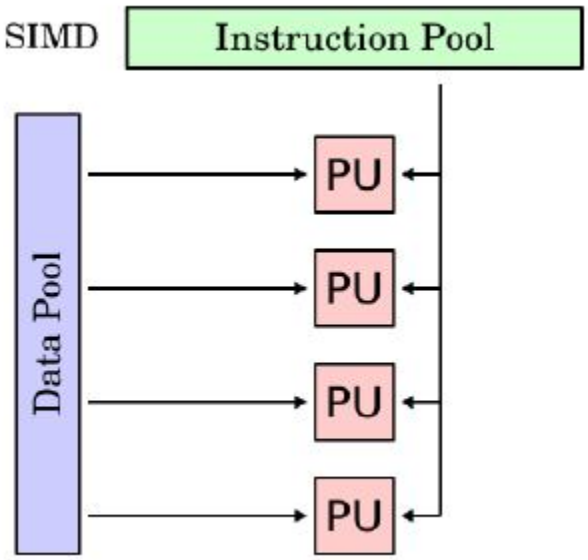
1.*Single instruction stream, single data stream*

(SISD)—This category is the uniprocessor.

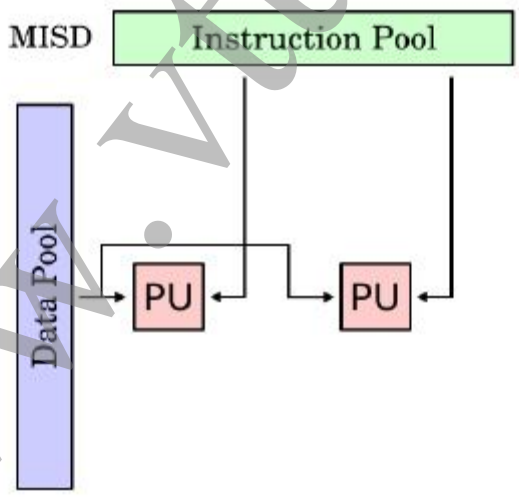


2.*Single instruction stream, multiple data streams*

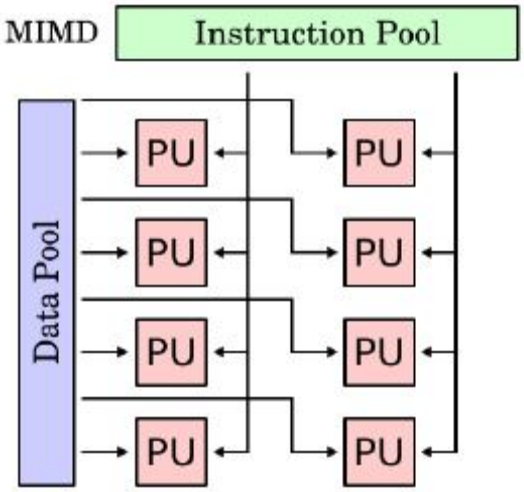
(SIMD)—The same instruction is executed by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. Vector architectures are the largest class of processors of this type.



3.*Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date, but may be in the future. Some special purpose stream processors approximate a limited form of this (there is only a single data stream that is operated on by successive functional units).



4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors. This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.



1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.

2. MIMDs can build on the cost/performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers.

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. Recall from the last chapter, that a process is an segment of code that may be run independently, and that the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of the processes on other processors. It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*

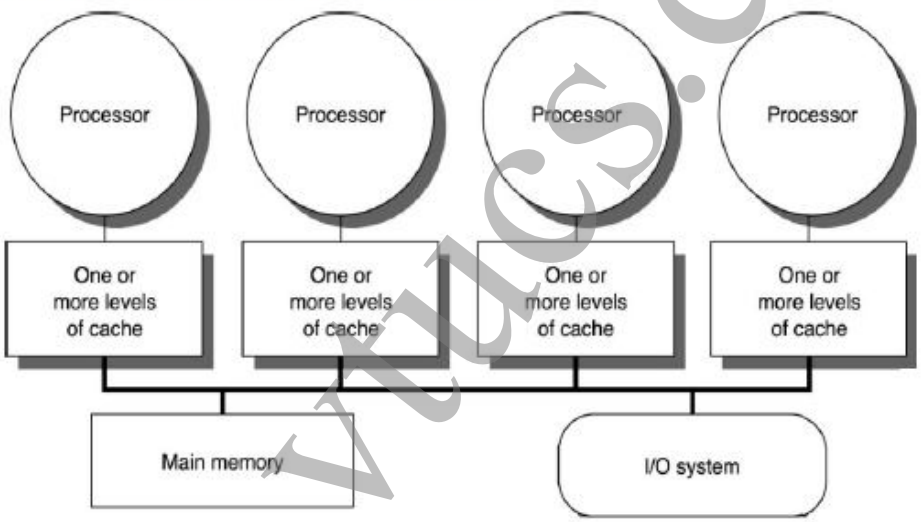
. Today, the term thread is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space. To take advantage of an MIMD multiprocessor with *n* processors, we must usually have at least *n* threads or processes to execute. The independent threads are typically identified by the programmer or created by the compiler. Since the parallelism in this situation is contained in the threads, it is called *thread-level parallelism*.

Threads may vary from large-scale, independent processes–for example, independent programs running in a multiprogrammed fashion on different processors– to parallel iterations of a loop, automatically generated by a compiler and each executing for perhaps less than a thousand instructions. Although the size of a thread is important in considering how to exploit thread-level parallelism efficiently, the important qualitative

distinction is that such parallelism is identified at a high-level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel. In contrast, instruction level parallelism is identified by primarily by the hardware, though with software help in some cases, and is found and exploited one instruction at a time.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization, because what constitutes a small or large number of processors is likely to change over time.
The first group, which we call

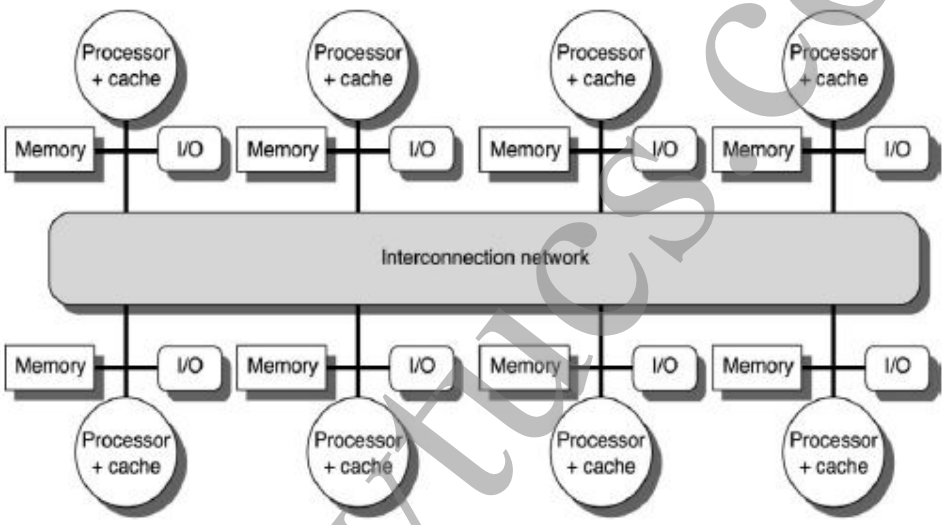
centralized shared-memory architectures



Centralized shared memory architectures have at most a few dozen processors in 2000. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even a switch, a centralized shared memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable, sharing a centralized memory, even organized as multiple banks, becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processos and a uniform access time from any processor, these multiprocessors are often called *symmetric (shared-memory) multiprocessors* (*SMPs*), and this style of architecture is sometimes called *UMA* for *uniform memory access*. This type of centralized sharedmemory architecture is currently by far the most popular organization.

The second group consists of multiprocessors with physically distributed memory. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the scale of multiprocessor for which distributed memory is preferred over a single, centralized memory continues to decrease in number (which is another reason not to use small and large scale). Of course, the larger number of processors raises the need for a high bandwidth interconnect.



Distributed-memory multiprocessor

Distributing the memory among the nodes has two major benefits. First, it is a costeffective way to scale the memory bandwidth, if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantage for a distributed memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency, at least when there is no contention, because the processors no longer share a single centralized memory. As we will see shortly, the use of distributed memory leads to two different paradigms for interprocessor communication. Typically, I/O as well as memory is distributed among the nodes of the multiprocessor, and the nodes may be small SMPs (2–8 processors). Although the use of multiple processors in a node together with a memory and a network interface is quite useful from the cost-efficiency viewpoint.

Challenges for Parallel Processing

• Limited parallelism available in programs

– Need new algorithms that can have better parallel performance

• Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

$$\text{Speedup}_{\text{Overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$80 = \frac{1}{\frac{Fraction_{Parallel}}{100} + (1 - Fraction_{Parallel})}$$

$$Fraction_{Parallel} = 0.9975$$

Data Communication Models for Multiprocessors

– shared memory: access shared address space implicitly via load and store operations.

– message-passing: done by explicitly passing messages among the processors

- can invoke software with Remote Procedure Call (RPC)
- often via library, such as MPI: Message Passing Interface
- also called "Synchronous communication" since communication causes synchronization between 2 processes

Message-Passing Multiprocessor

- The address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor

- The same physical address on two different processors refers to two different locations in two different memories.

Multicomputer (cluster):

- can even consist of completely separate computers connected on a LAN.

- cost-effective for applications that require little or no communication

Symmetric Shared-Memory Architectures

Multilevel caches can substantially reduce the memory bandwidth demands of a processor.

This is extremely

- Cost-effective
- This can work as plug in play by placing the processor and cache sub-system on a board into the bus backplane.

Developed by

- IBM – One chip multiprocessor
- AMD and INTEL- Two –Processor
- SUN – 8 processor multi core

Symmetric shared – memory support caching of

- Shared Data
- Private Data

Private data: used by a single processor

When a private item is cached, its location is ***migrated*** to the cache Since no other processor uses the data, the program behavior is identical to that in a uniprocessor.

Shared data: used by multiple processor

When shared data are cached, the shared value may be ***replicated*** in multiple caches

advantages: reduce access latency and memory contention induces a new problem: cache coherence.

Cache Coherence

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. I.e, If two different processors have two different values for the same location, this difficulty is generally referred to as cache coherence problem

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Cache coherence problem for a single memory location

• **Informally:**

– "Any read must return the most recent write"
– Too strict and too difficult to implement
–

• **Better:**

– "Any write must eventually be seen by a read"
– All writes are seen in proper order ("serialization")
–

• **Two rules to ensure this:**

– "If P writes x and then P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
– Writes to a single location are serialized: seen in one order
• Latest write will be seen
• Otherwise could see writes in illogical order (could see older value after a newer value)

The definition contains two different aspects of memory system:
• Coherence
• Consistency

A memory system is coherent if,
• Program order is preserved.
• Processor should not continuously read the old data value.
• Write to the same location are serialized.

The above three properties are sufficient to ensure coherence,*When a written value will be seen is also important.* This issue is defined by memory consistency model. Coherence and consistency are complementary.

Basic schemes for enforcing coherence

Coherence cache provides:

• migration: a data item can be moved to a local cache and used there in a transparent fashion.
• replication for shared data that are being simultaneously read.
• both are critical to performance in accessing shared data.

To over come these problems, adopt a hardware solution by introducing a protocol tomaintain coherent caches named as Cache Coherence Protocols
These protocols are implemented for tracking the state of any sharing of a data block.
Two classes of Protocols
• Directory based
• Snooping based

Directory based

- Sharing status of a block of physical memory is kept in one location called the directory.
- Directory-based coherence has slightly higher implementation overhead than snooping.
- It can scale to larger processor count.

Snooping

- Every cache that has a copy of data also has a copy of the sharing status of the block.
- No centralized state is kept.
- Caches are also accessible via some broadcast medium (bus or switch)
- Cache controller monitor or snoop on the medium to determine whether or not they have a copy of a block that is represented on a bus or switch access.

Snooping protocols are popular with multiprocessor and caches attached to single shared memory as they can use the existing physical connection- bus to memory, to interrogate the status of the caches. Snoop based cache coherence scheme is implemented on a shared bus. Any communication medium that broadcasts cache misses to all the processors.

Basic Snoopy Protocols

- Write strategies
  - Write-through: memory is always up-to-date
  - Write-back: snoop in caches to find most recent copy
- Write Invalidate Protocol
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
- Read miss: further read will miss in the cache and fetch a new copy of the data.
- Write Broadcast/Update Protocol (typically write through)
  - Write to shared data: broadcast on bus, processors snoop, and *update* any copies
  - Read miss: memory/cache is always up-to-date.
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

Examples of Basic Snooping Protocols

Write Invalidate

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

Write Update

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

Assume neither cache initially holds X and the value of X in memory is 0

Example Protocol

• Snooping coherence protocol is usually implemented by incorporating a finitestate controller in each node

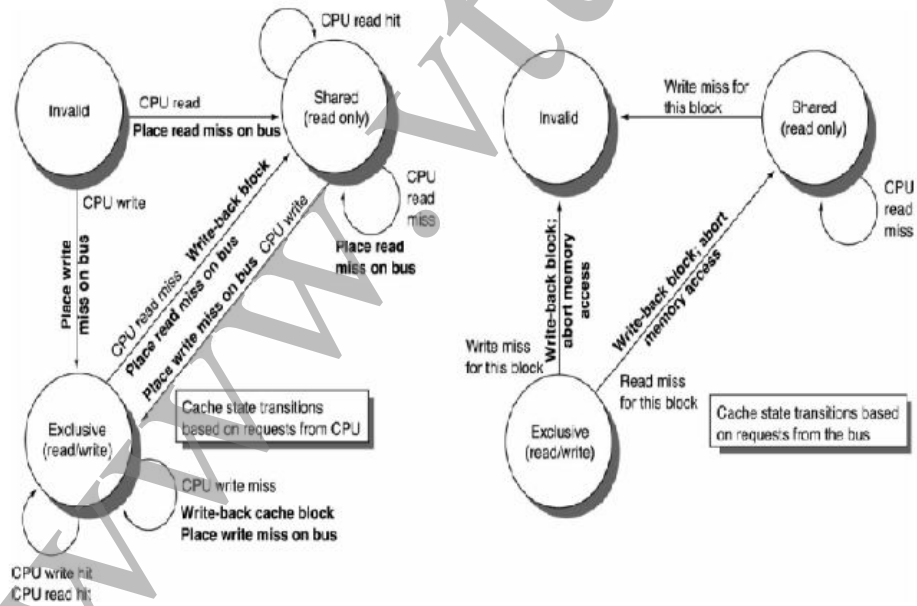
• Logically, think of a separate controller associated with each cache block
  – That is, snooping operations or cache requests for different blocks can proceed independently
• In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  – that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - – Snoops every address on bus
  - – If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state:
  - – Clean in all caches and up-to-date in memory (Shared)
  - – OR Dirty in exactly one cache (Exclusive)
  - – OR Not in any caches
- Each cache block is in one state (track these):
  - – Shared : block can be read
  - – OR Exclusive : cache has only copy, its writeable, and dirty
  - – OR Invalid : block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

Write-Back State Machine – CPU

State Transitions for Each Cache Block is as shown below



- CPU may read/write hit/miss to the block
- May place write/read miss on bus
- May receive read/write miss from bus

Request	Source	State of block	Function and explanation
Read hit	processor	shared or exclusive	Read data in cache.
Read miss	processor	invalid	Place read miss on bus.
Read miss	processor	shared	Address conflict miss: place read miss on bus.
Read miss	processor	exclusive	Address conflict miss: write back block, then place read miss on bus
Write hit	processor	exclusive	Write data in cache.
Write hit	processor	shared	Place write miss on bus.
Write miss	processor	invalid	Place write miss on bus.
Write miss	processor	shared	Address conflict miss: place write miss on bus.
Write miss	processor	exclusive	Address conflict miss: write back block, then place write miss on bus.
Read miss	bus	shared	No action; allow memory to service read miss.
Read miss	bus	exclusive	Attempt to share data: place cache block on bus and change state to shared.
Write miss	bus	shared	Attempt to write shared block; invalidate the block.
Write miss	bus	exclusive	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid.

Conclusion

- "End" of uniprocessors speedup => Multiprocessors
- Parallelism challenges: % parallalizable, long latency to remote memory
- Centralized vs. distributed memory
  - – Small MP vs. lower latency, larger BW for Larger MP
- Message Passing vs. Shared Address
  - – Uniform access time vs. Non-uniform access time
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data _ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Shared medium serializes writes _ Write consistency

Implementation Complications

- Write Races:
  - – Cannot update cache until bus is obtained
- Otherwise, another processor may get bus first, and then write the same cache block!
  - – Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - – If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - – Split transaction bus:
- Bus transaction is not atomic:

can have multiple outstanding transactions for a block

• Multiple misses can interleave, allowing two caches to grab block in the Exclusive state

• Must track and prevent multiple misses for one block

• Must support interventions and invalidations

Performance Measurement

• Overall cache performance is a combination of

– Uniprocessor cache miss traffic

– Traffic caused by communication – invalidation and subsequent cache misses

• Changing the processor count, cache size, and block size can affect these two components of miss rate

• Uniprocessor miss rate: compulsory, capacity, conflict

• Communication miss rate: coherence misses

– True sharing misses + false sharing misses

True and False Sharing Miss

• **True sharing miss**

– The first write by a PE to a shared cache block causes an invalidation to establish ownership of that block

– When another PE attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred

• **False sharing miss**

– Occur when a block a block is invalidate (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written to

– **The block is shared, but no word in the cache is actually shared, and this miss would not occur if the block size were a single word**

• Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss or a false sharing miss.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

Example Result

• True sharing miss (invalidate P2)
• 2: False sharing miss
– x2 was invalidated by the write of P1, but that value of x1 is not used in P2
• 3: False sharing miss
— The block containing x1 is marked shared due to the read in P2, but P2 did not read x1. A write miss is required to obtain exclusive access to the block
• 4: False sharing miss
• 5: True sharing miss

Distributed Shared-Memory Architectures

Distributed shared-memory architectures
• Separate memory per processor
– Local or remote access via memory controller
– The physical address space is statically distributed Coherence Problems
• Simple approach: uncacheable
– shared data are marked as uncacheable and only private data are kept in caches
– very long latency to access memory for shared data
• Alternative: directory for memory blocks
▪ The directory per memory tracks state of every block in every cache
• which caches have a copies of the memory block, dirty vs. clean, ...

Two additional complications
• The interconnect cannot be used as a single point of arbitration like the bus
• Because the interconnect is message oriented, many messages must have explicit responses

To prevent directory becoming the bottleneck, we distribute directory entries with memory, each keeping track of which processors have copies of their memory blocks

Directory Protocols

• Similar to Snoopy Protocol: Three states
– ***Shared***: 1 or more processors have the block cached, and the value in memory is up-to-date (as well as in all the caches)
– ***Uncached***: no processor has a copy of the cache block (not valid in any cache)
– ***Exclusive***: Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date

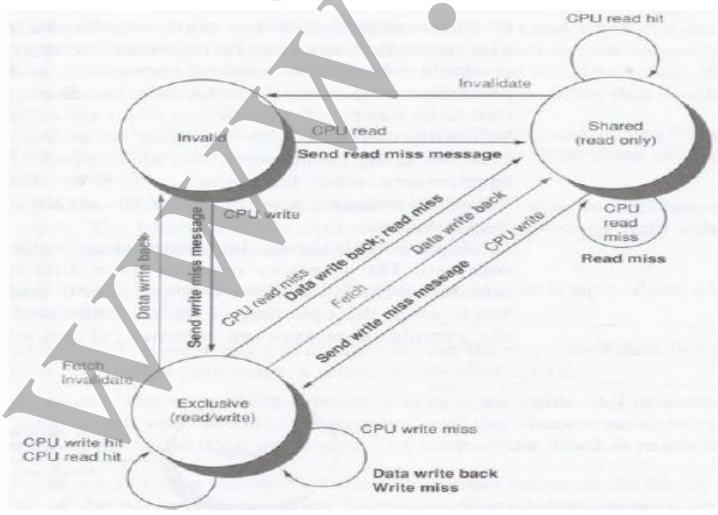
• The processor is called the owner of the block
• In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared (usually a bit vector for each memory block: 1 if processor has copy)
• Keep it simple(r):
– Writes to non-exclusive data => write miss
– Processor blocks until access completes
– Assume messages received and acted upon in order sent

Messages for Directory Protocols

Message type	Source	Destination	Message contents	Function of this message
Read miss	local cache	home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	local cache	home directory	P, A	Processor P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	home directory	remote cache	A	Invalidate a shared copy of data at address A.
Fetch	home directory	remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	home directory	remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	home directory	local cache	D	Return a data value from the home memory.
Data write back	remote cache	home directory	A, D	Write back a data value for address A.

• local node: the node where a request originates
• home node: the node where the memory location and directory entry of an address reside
• remote node: the node that has a copy of a cache block (exclusive or shared)

State Transition Diagram for Individual Cache Block



• Comparing to snooping protocols:
– identical states

– stimulus is almost identical
– write a shared cache block is treated as a write miss (without fetch the block)
– cache block must be in exclusive state when it is written
– any shared block must be up to date in memory

• write miss: data fetch and selective invalidate operations sent by the directory controller (broadcast in snooping protocols)

Directory Operations: Requests and Actions

• Message sent to directory causes two actions:
– Update the directory
– More messages to satisfy request

• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:

– Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.

– Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

• Block is Shared => the memory value is up-to-date:
– Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
– Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

• Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:

– Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.

Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.

– Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.

– Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

Synchronization: The Basics

Synchronization mechanisms are typically built with user-level software routines that rely on hardware –supplied synchronization instructions.

• Why Synchronize?
Need to know when it is safe for different processes to use shared data
• Issues for Synchronization:
– Uninterruptable instruction to fetch and update memory (atomic operation);
– User level synchronization operation using this primitive;
– For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory

• Atomic exchange: interchange a value in a register for a value in memory
0 _ synchronization variable is free
1 _ synchronization variable is locked and unavailable
– Set register to 1 & swap
– New value in register determines success in getting lock
0 if you succeeded in setting the lock (you were first)
1 if other processor had already claimed access
– Key is that exchange operation is indivisible
• Test-and-set: tests a value and sets it if the value passes the test
• Fetch-and-increment: it returns the value of a memory location and atomically increments it
– 0 _ synchronization variable is free
• Hard to have read & write in 1 instruction: use 2 instead
• Load linked (or load locked) + store conditional
– Load linked returns the initial value
– Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
• Example doing atomic swap with LL & SC:

```
try:  mov  R3,R4 ;       mov exchange value

      ll   R2,0(R1) ; load linked

      sc   R3,0(R1) ; store conditional

      beqz R3,try ; branch store fails (R3 = 0)

      mov  R4,R2 ; put load value in R4
```

• Example doing fetch & increment with LL & SC:

```
try:  ll   R2,0(R1) ;   load linked
      addi R2,R2,#1 ;    increment (OK if reg–reg)
      sc   R2,0(R1) ;   store conditional
      beqz R2,try ;     branch store fails (R2 = 0)
```

User Level Synchronization—Operation Using this Primitive

• Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
li  R2,#1
lockit: exch    R2,0(R1) ;      atomic exchange
        bnez    R2,lockit ;     already locked?
```

• What about MP with cache coherency?

– Want to spin on cache copy to avoid full memory latency

– Likely to get cache hits for such variables

• Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

• Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:    li      R2,#1
lockit: lw      R3,0(R1)        ;load var
        bnez    R3,lockit ;             _ 0 _ not free _ spin
        exch    R2,0(R1) ;              atomic exchange
        bnez    R2,try ;                already locked?
```

Memory Consistency Models

• What is consistency? When must a processor see the new value? e.g., seems that P1: A = 0; P2: B = 0;

```
..... .....
        A = 1; B = 1;
   L1: if (B == 0) ... L2: if (A == 0) ...
```

• Impossible for both if statements L1 & L2 to be true?

– What if write invalidate is delayed & processor continues?

• Memory consistency models:

what are the rules for such cases?

• Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved _ assignments before ifs above

– SC: delay all memory accesses until all invalidates done

• Schemes faster execution to sequential consistency

• Not an issue for most programs; they are synchronized

– A program is synchronized if all access to shared data are ordered by synchronization operations

```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

• Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)

• Several Relaxed Models for Memory Consistency since most programs are

synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed Consistency Models : The Basics

• Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent

– By relaxing orderings, may obtain performance advantages

– Also specifies range of legal compiler optimizations on shared data

– Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program

• 3 major sets of relaxed orderings:

1. W_R ordering (all writes completed before next read)

• Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency

2. W _ W ordering (all writes completed before next write)

3. R _ W and R _ R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering

• Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

UNIT - VI

REVIEW OF MEMORY HIERARCHY:

Introduction

Cache performance

Cache Optimizations

Virtual memory.

6 Hours

UNIT VI

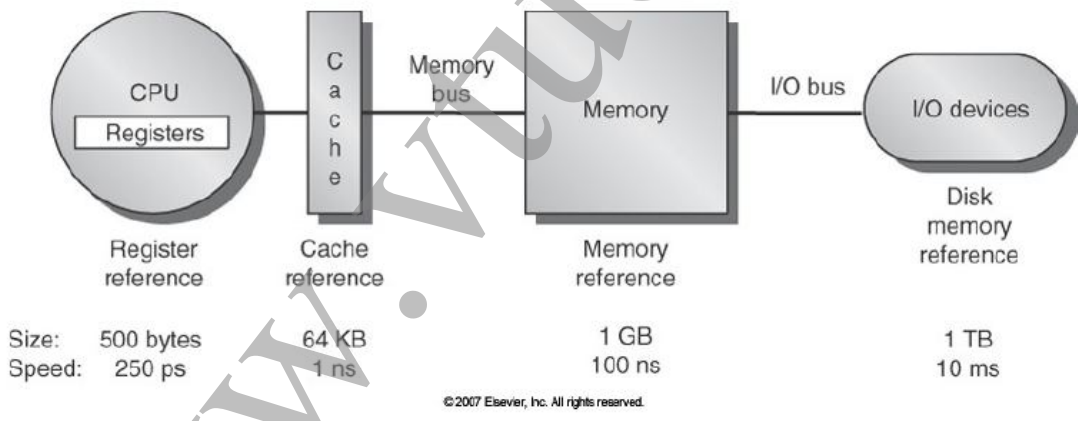
REVIEW OF MEMORY HIERARCHY

• Unlimited amount of fast memory
- Economical solution is memory hierarchy
- Locality
- Cost performance

Principle of locality
- most programs do not access all code or data uniformly.

• Locality occurs
- Time (Temporal locality)
- Space (spatial locality)

• Guidelines
– Smaller hardware can be made faster
– Different speed and sizes

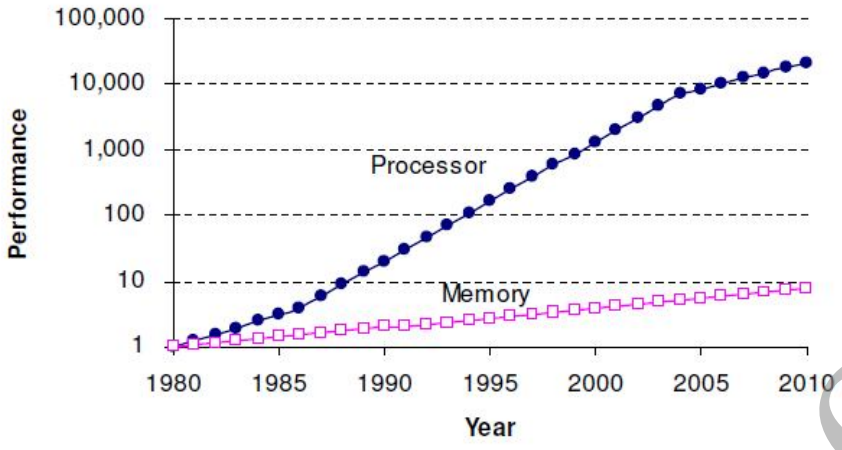


Goal is provide a memory system with cost per byte than the next lower level

• Each level maps addresses from a slower, larger memory to a smaller but faster memory higher in the hierarchy.
– Address mapping
– Address checking.

• Hence protection scheme for address for scrutinizing addresses are also part of the memory hierarchy.

Why More on Memory Hierarchy?



• The importance of memory hierarchy has increased with advances in performance of processors.

• **Prototype**

`– When a word is not found in cache

• Fetched from memory and placed in cache with the address tag.

• Multiple words(block) is fetched for moved for efficiency reasons.

– key design

• Set associative

– Set is a group of block in the cache.

– Block is first mapped on to set.

» Find mapping

» Searching the set

Chosen by the address of the data:

(Block address) MOD(Number of sets in cache)

• n-block in a set

– Cache replacement is called n-way set associative.

Cache data

- Cache read.
- Cache write.

Write through: update cache and writes through to update memory.

Both strategies

- Use write buffer.

this allows the cache to proceed as soon as the data is placed in the buffer rather than wait the full latency to write the data into memory.

Metric

used to measure the benefits is ***miss rate***

No of access that miss

No of accesses

Write back: updates the copy in the cache.

• Causes of high miss rates

– Three Cs model sorts all misses into three categories

• Compulsory: every first access cannot be in cache

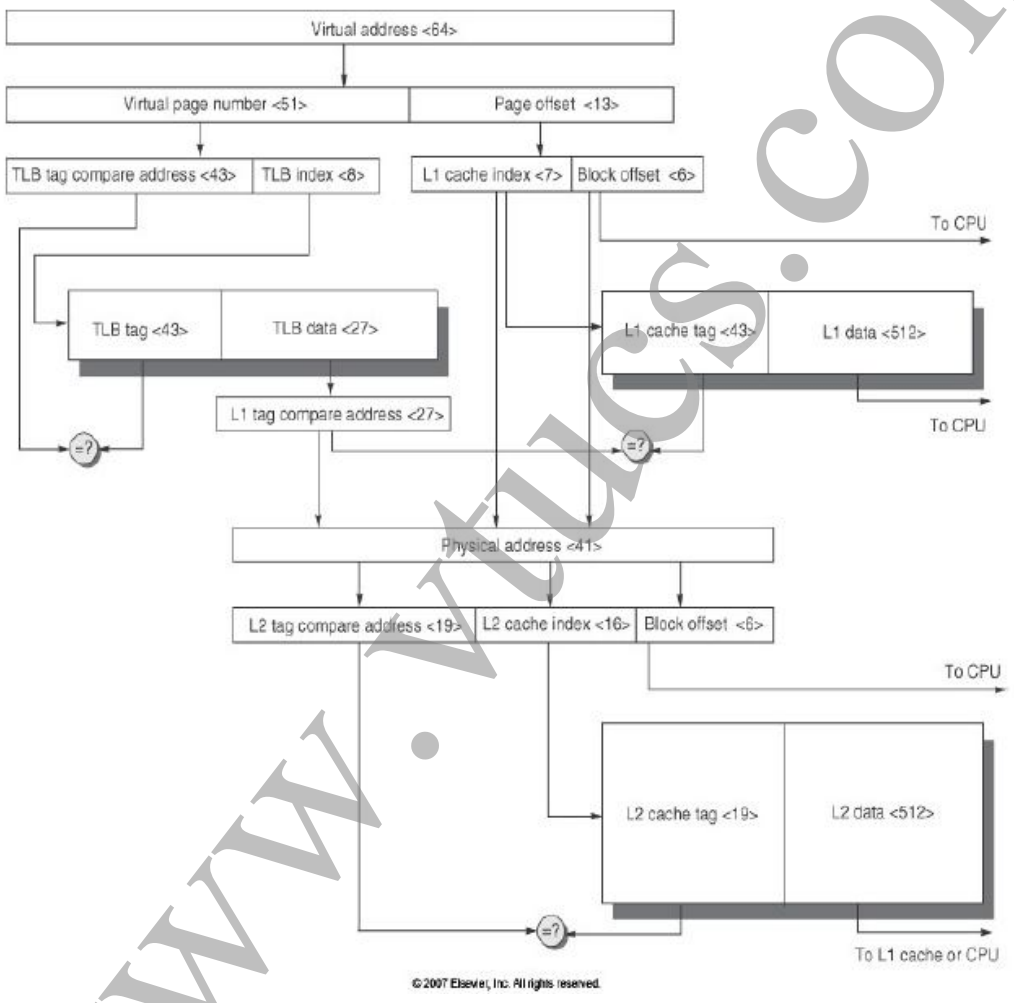
– Compulsory misses are those that occur if there is an infinite cache

• Capacity: cache cannot contain all that blocks that are needed for the program.

– As blocks are being discarded and later retrieved.

• Conflict: block placement strategy is not fully associative

– Block miss if blocks map to its set.



Miss rate can be a misleading measure for several reasons

So, misses per instruction can be used per memory reference

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate X Memory accesses}}{\text{Instruction count}}$$

$$= \text{Miss rate X } \frac{\text{Mem accesses}}{\text{Instruction}}$$

Cache Optimizations

Six basic cache optimizations

1. Larger block size to reduce miss rate:
- To reduce miss rate through spatial locality.
- Increase block size.
- Larger block size reduce compulsory misses.
- But they increase the miss penalty.

2. Bigger caches to reduce miss rate:
- capacity misses can be reduced by increasing the cache capacity.
- Increases larger hit time for larger cache memory and higher cost and power.

3. Higher associativity to reduce miss rate:
- Increase in associativity reduces conflict misses.

4. Multilevel caches to reduce penalty:
- Introduces additional level cache
- Between original cache and memory.
- L1- original cache

L2- added cache.

L1 cache: - small enough
- speed matches with clock cycle time.

L2 cache: - large enough
- capture many access that would go to main memory.

Average access time can be redefined as

Hit timeL1+ Miss rate L1 X (Hit time L2 + Miss rate L2 X Miss penalty L2)

5. Giving priority to read misses over writes to reduce miss penalty:
- write buffer is a good place to implement this optimization.
- write buffer creates hazards: read after write hazard.

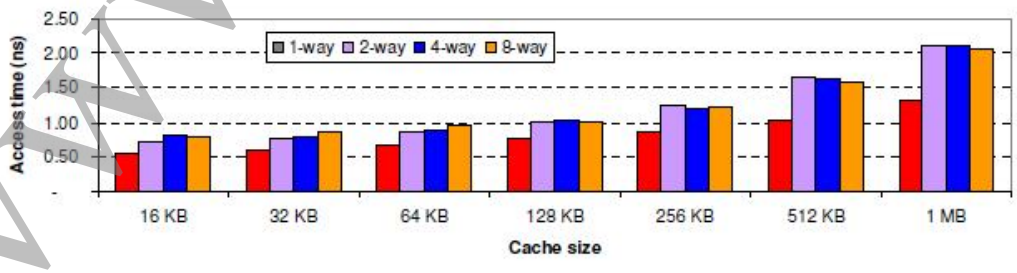
6. Avoiding address translation during indexing of the cache to reduce hit time:
- Caches must cope with the translation of a virtual address from the processor to a physical address to access memory.
- common optimization is to use the page offset.
- part that is identical in both virtual and physical addresses- to index the cache.

Advanced Cache Optimizations

- **Reducing hit time**
  - – Small and simple caches
  - – Way prediction
  - – Trace caches
- **Increasing cache bandwidth**
  - – Pipelined caches
  - – Multibanked caches
  - – Nonblocking caches
- **Reducing Miss Penalty**
  - – Critical word first
  - – Merging write buffers
- **Reducing Miss Rate**
  - – Compiler optimizations
- **Reducing miss penalty or miss rate via parallelism**
  - – Hardware prefetching
- – Compiler prefetching
- –

First Optimization : Small and Simple Caches

- Index tag memory and then compare takes time
- _ Small cache can help hit time since smaller memory takes less time to index
  - – E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
  - – Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple _ direct mapping
  - – Can overlap tag check with data transmission since no choice
- Access time estimate for 90 nm using CACTI model 4.0
- – Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches



Second Optimization: Way Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?

• Way prediction: keep extra bits in cache to predict the "way," or block within the set, of next cache access.

– Multiplexer is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data

– Miss _ 1st check other blocks for matches in next clock cycle
- Accuracy » 85%
- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Used for instruction caches vs. data caches

Third optimization: Trace Cache

• Find more instruction level parallelism?
How to avoid translation from x86 to microops?
• Trace cache in Pentium 4
1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
– Built-in branch predictor
2. Cache the micro-ops vs. x86 instructions
– Decode/translate from x86 to micro-ops on trace cache miss
+ 1. _ better utilize long blocks (don't exit in middle of block, don't enter at label in middle of block)
- 1. _ complicated address mapping since addresses no longer aligned to powerof-
2 multiples of word size
- 1. _ instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

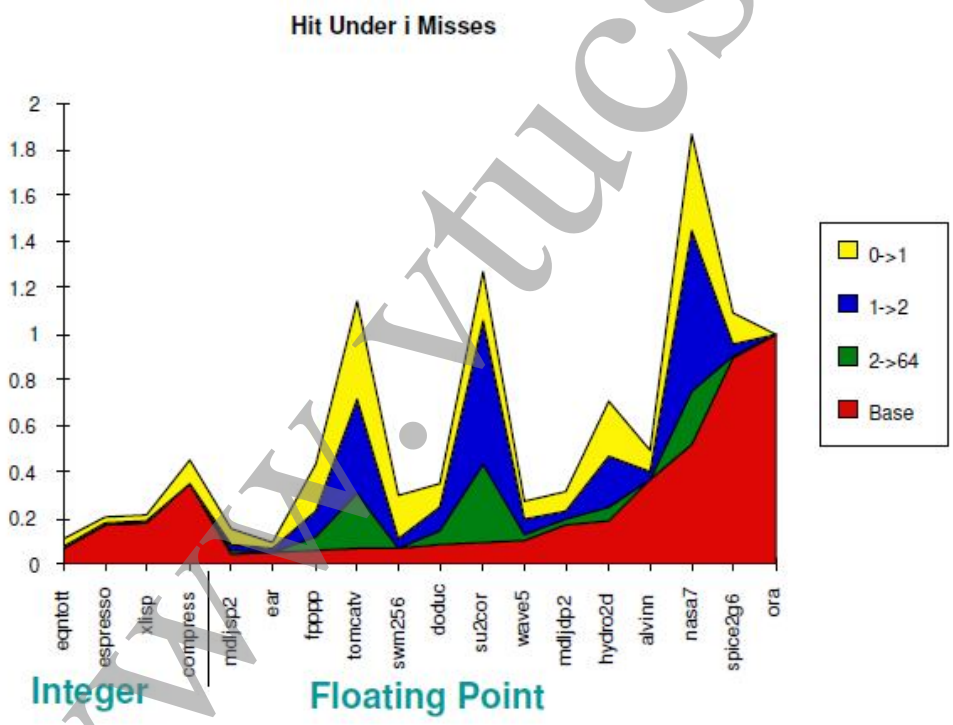
Fourth optimization: pipelined cache access to increase bandwidth

• Pipeline cache access to maintain bandwidth, but higher latency
• Instruction cache access pipeline stages:
1: Pentium
2: Pentium Pro through Pentium III
4: Pentium 4
- _ greater penalty on mispredicted branches
- _ more clock cycles between the issue of the load and the use of the data

Fifth optimization: Increasing Cache Bandwidth Non-Blocking Caches

• *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  – requires F/E bits on registers or out-of-order execution
  – requires multi-bank memories
• "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
• "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
  – Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  – Requires multiple memory banks (otherwise cannot support)
– Pentium Pro allows 4 outstanding memory misses

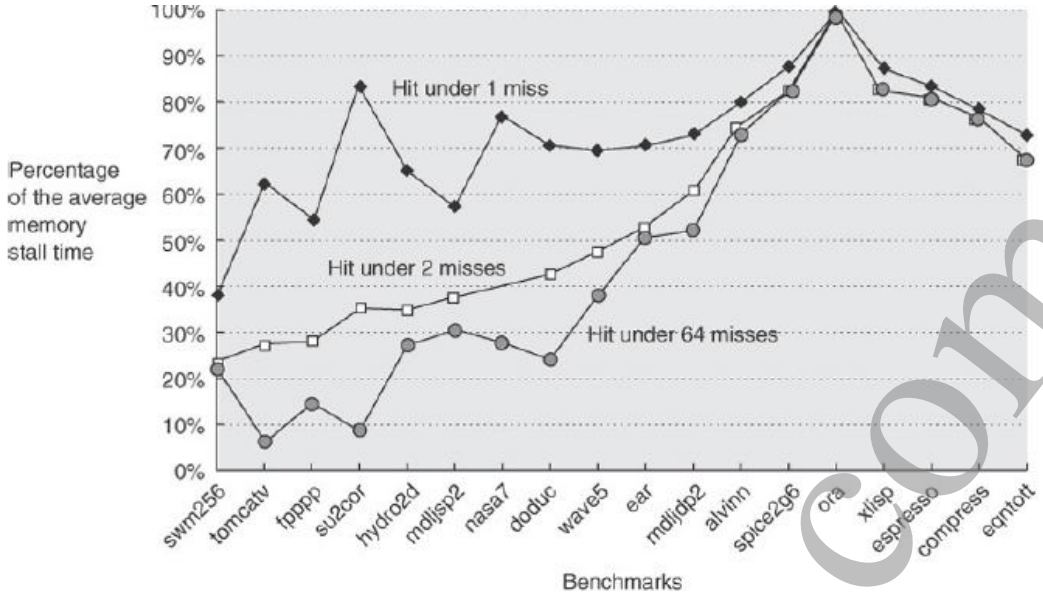
Value of Hit Under Miss for SPEC



• FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26

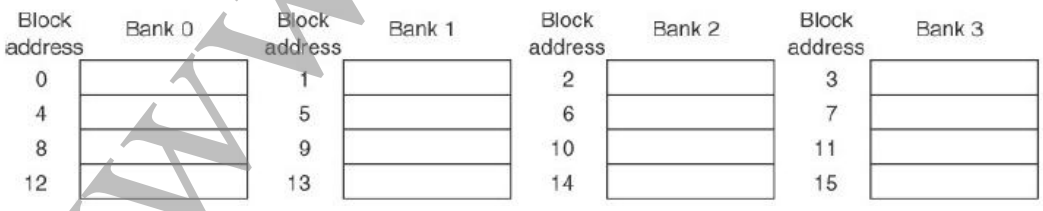
• Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19

• 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92



Sixth optimization: Increasing Cache Bandwidth via Multiple Banks

• Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses
  – E.g.,T1 (“Niagara”) L2 has 4 banks
• Banking works best when accesses naturally spread themselves across banks _ mapping of addresses to banks affects behavior of memory system
• Simple mapping that works well is “sequential interleaving”
  – Spread block addresses sequentially across banks
  – E,g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; …



Seventh optimization :Reduce Miss Penalty: Early Restart and Critical Word First

• Don’t wait for full block before restarting CPU
• *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  – Spatial locality _ tend to want next sequential word, so not clear size of benefit of just early restart

• *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block

– Long blocks more popular today _ Critical Word 1st Widely used

Eight optimization: Merging Write Buffer to Reduce Miss Penalty-

• Write buffer to allow processor to continue while waiting to write to memory

• If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry

• If so, new data are combined with that entry

• Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory

• The Sun T1 (Niagara) processor, among many others, uses write merging

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

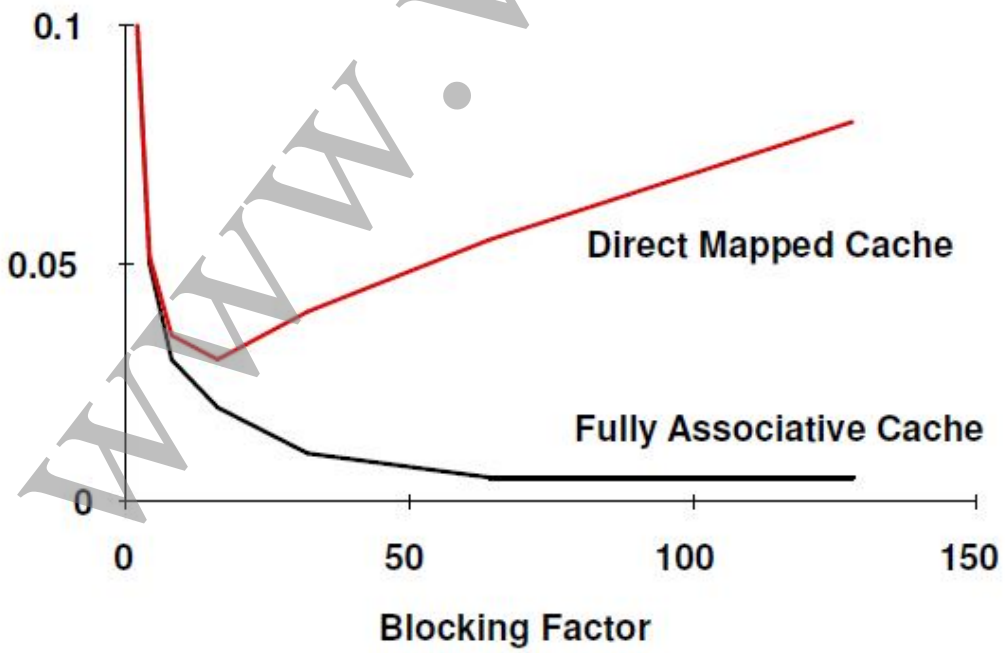
Ninth optimization: Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];
/* After: 1 array of stuctures */
struct merge {
int val;
int key;
};
struct merge merged_array[SIZE];
```

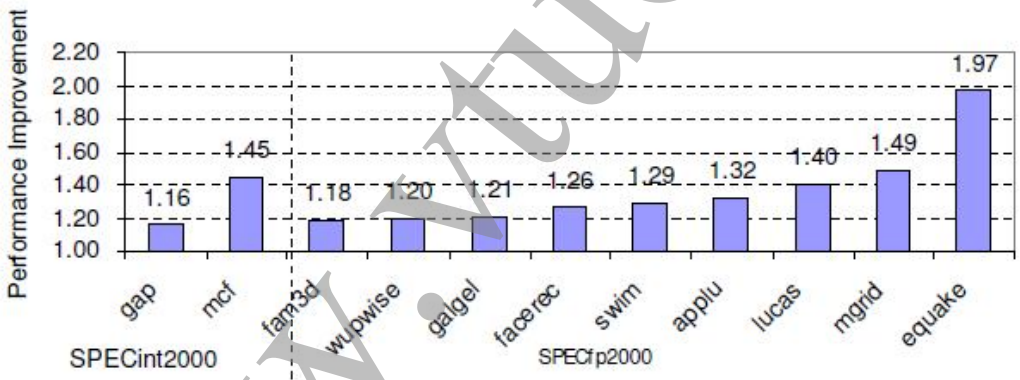
Reducing conflicts between val & key; improve spatial locality



• Conflict misses in caches not FA vs. Blocking size
– Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Tenth optimization Reducing Misses by Hardware Prefetching of Instructions & Data

• Prefetching relies on having extra memory bandwidth that can be used without penalty
• Instruction Prefetching
– Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
– Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer
• Data Prefetching
– Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
– Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes



Eleventh optimization: Reducing Misses by Software Prefetching Data

• Data Prefetch
– Load data into register (HP PA-RISC loads)
– Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
– Special prefetching instructions cannot cause faults; a form of speculative execution
• Issuing Prefetch Instructions takes time
– Is cost of prefetch issues < savings in reduced misses?
– Higher superscalar reduces difficulty of issue bandwidth

The techniques to improve hit time, bandwidth, miss penalty and miss rate generally affect the other components of the average memory access equation as well as the complexity of the memory hierarchy.

UNIT - VII

MEMORY HIERARCHY DESIGN:

Introduction

Advanced optimizations of Cache performance

Memory technology and optimizations

Protection

Virtual memory and virtual machines.

6 Hours

UNIT VII

MEMORY HIERARCHY DESIGN

AMAT and Processor Performance

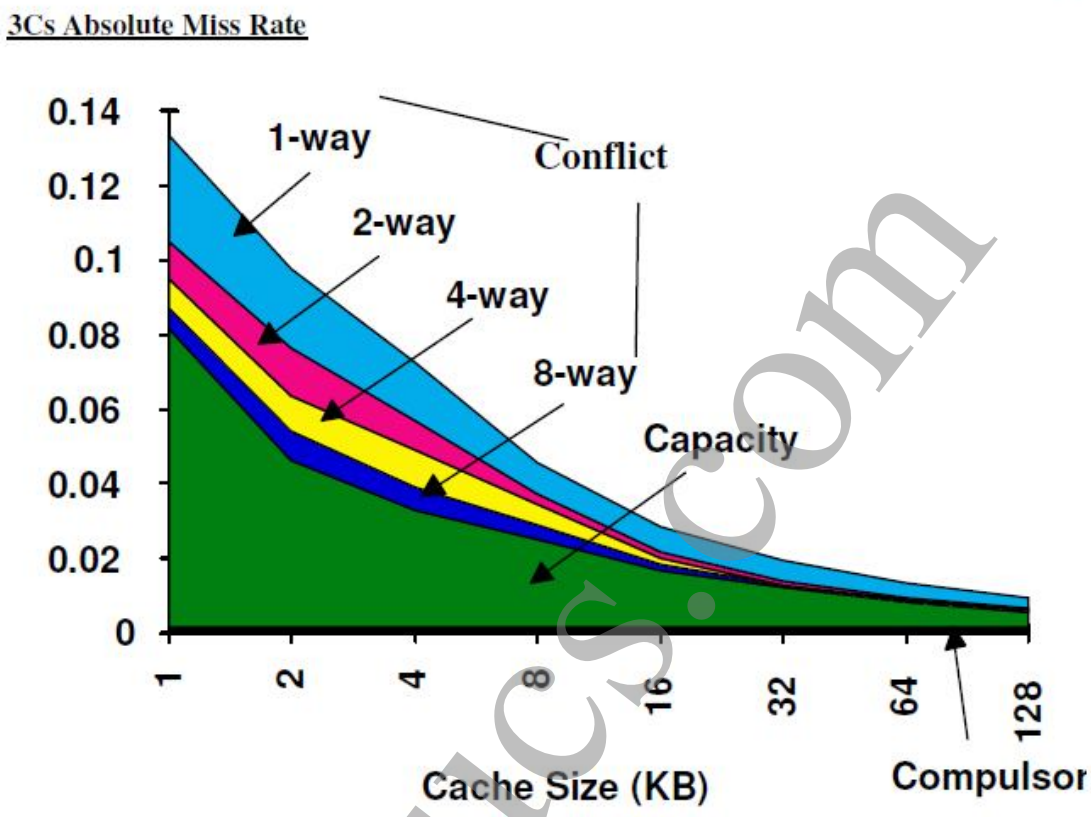
•AMAT = Average Memory Access Time
•Miss-oriented Approach to Memory Access
–CPIExec includes ALU and Memory instructions
•Separating out Memory component entirely
–CPIALUOps does not include memory instructions

Summary: Caches

•The Principle of Locality:
–Program access a relatively small portion of the address space at any instant of time.
•Temporal Locality OR Spatial Locality:
•Three Major Categories of Cache Misses:
–Compulsory Misses: sad facts of life. Example: cold start misses.
–Capacity Misses: increase cache size
–Conflict Misses: increase cache size and/or associativity

Where Misses Come From?

•Classifying Misses: 3 Cs
–Compulsory — The first access to a block is not in the cache,
Also called cold start misses or first reference misses.
(Misses in even an Infinite Cache)
–Capacity — If the cache cannot contain all the blocks needed during execution of a program,
–Conflict — If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set.
(Misses in N-way Associative, Size X Cache)
More recent, 4th "C":
–Coherence — Misses caused by cache coherence



•**Write Policy:**

–Write Through: needs a write buffer.
–Write Back: control can be complex

Summary:

The Cache Design Space

–Several interacting dimensions
–cache size
–block size
–associativity
–replacement policy
–write-through vs write-back
–The optimal choice is a compromise
–Simplicity often wins

Cache Organization?

•Assume total cache size not changed

•What happens if: Which of 3Cs is obviously affected?

–Change Block Size
–Change Cache Size
–Change Cache Internal Organization
–Change Associativity
–Change Compiler

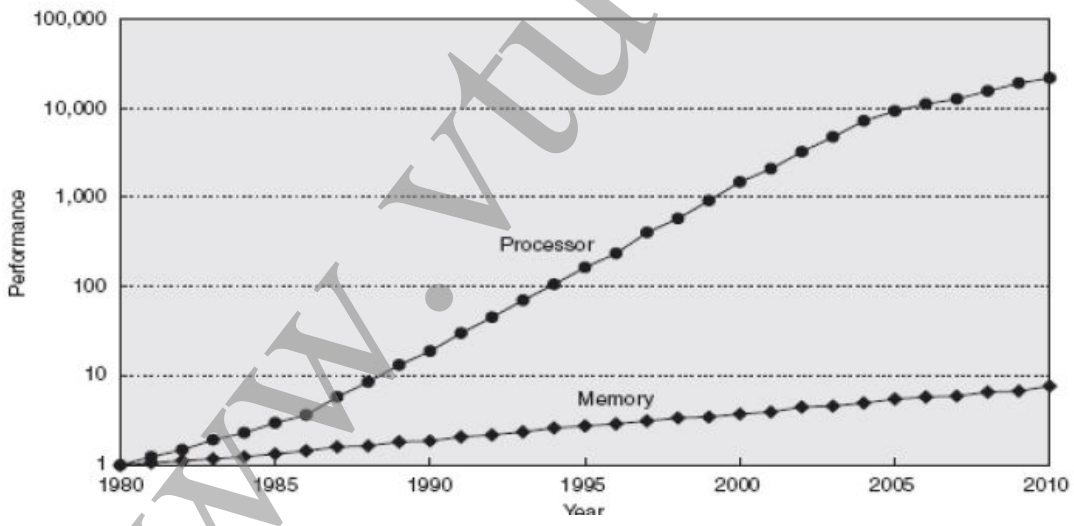
Cache Optimization Summary

How to Improve Cache Performance?

•Cache optimizations
–1. Reduce the miss rate
–2. Reduce the miss penalty
–3. Reduce the time to hit in the cache

Cache Optimisation

Why improve Cache performance:



Performance improvement of CPU vs Memory- CPU fabrication has advanced much more than memory- hence need to use cache optimization techniques.

Review: 6 Basic Cache Optimizations

• Reducing hit time

1.Address Translation during Cache Indexing

• Reducing Miss Penalty

2. Multilevel Caches

3. Giving priority to read misses over write misses

• Reducing Miss Rate

4. Larger Block size (Compulsory misses)

5. Larger Cache size (Capacity misses)

6. Higher Associativity (Conflict misses)

11 Advanced Cache Optimizations

- **Reducing hit time**
  1. Small and simple caches
  2. Way prediction
  3. Trace caches
- **Increasing cache bandwidth**
  4. Pipelined caches
  5. Multibanked caches
  6. Nonblocking caches
- **Reducing Miss Penalty**
  7. Critical word first
  8. Merging write buffers
- **Reducing Miss Rate**
  9.Compiler optimizations
- **Reducing miss penalty or miss rate via parallelism**
  10.Hardware prefetching
  11.Compiler prefetching

1. Fast Hit times via Small and Simple Caches

Index tag memory and then compare takes time

• **Small** cache can help hit time since smaller memory takes less time to index

– E.g., L1 caches same size for 3 generations of AMD icroprocessors: K6, Athlon, and Opteron

– Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip

• **Simple** direct mapping

Can overlap tag check with data transmission since no choice

2. Fast Hit times via Way Prediction

• How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?

• Way prediction: keep extra bits in cache to predict the "way," or block within the set, ofnext cache access.

– Multiplexer is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data

– Miss - 1st check other blocks for matches in next clock cycle

3. Fast Hit times via Trace Cache

Find more instruction level parallelism?

How avoid translation from x86 to microops?- Trace cache in Pentium 4

1. Dynamic traces of the executed instructions vs. static sequence of instructions as determined by layout in memory

– Built-in branch predictor

2. Cache the micro-ops vs. x86 instructions - **Decode/translate from x86 to micro-ops on trace cache miss**

+ 1. 1 better utilize long blocks (don't exit in middle of block, don't enter at label in middle of block)

- 1. 1 complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size

- 1. 1 instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

4: Increasing Cache Bandwidth by Pipelining

–Pipeline cache access to maintain bandwidth, but higher latency

• Instruction cache access pipeline stages:

1: Pentium

2: Pentium Pro through Pentium III

4: Pentium 4

- greater penalty on mispredicted branches
- more clock cycles between the issue of the load and the use of the data

5. Increasing Cache Bandwidth:

Non-Blocking Caches- Reduce Misses/Penalty

• Non-blocking cache or lockup-free cache allow data cache to continue to supply cache hits during a m iss

– requires F/E bits on registers or out-of-order execution

– requires multi-bank memories

• "hit under miss" reduces the effective miss penalty by working during miss vs. ignoring CPU requests

• "hit under multiple miss" or "miss under miss" may further lower the effective miss penalty by overlapping multiple misses

– Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses

– Requires muliple memory banks (otherwise cannot support)

– Penium Pro allows 4 outstanding memory misses

6: Increasing Cache Bandwidth via Multiple Banks

Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses

– E.g.,T1 ("Niagara") L2 has 4 banks

• Banking works best when accesses naturally spread themselves across banks ı mapping of addresses to banks affects behavior of memory system

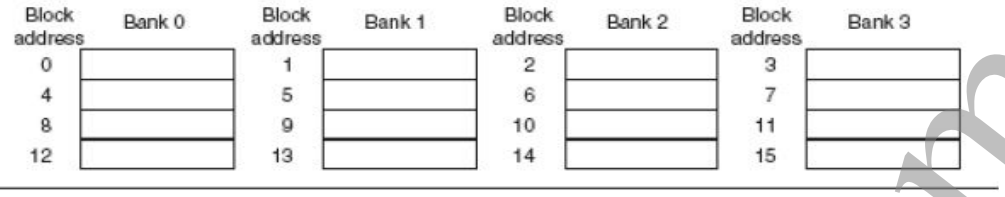


Figure 5.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Simple mapping that works well is "sequential interleaving"

– Spread block addresses sequentially across banks

– E,g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; ….

7. Reduce Miss Penalty:

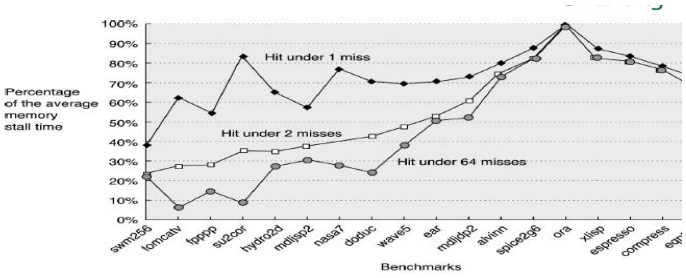
Early Restart and Critical Word First

Don't wait for full block before restarting CPU

Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution

– Spatial locality - tend to want next sequential word, so not clear size of benefit of just early restart

Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block



– Long blocks more popular today □ Critical Word 1st Widely used

8. Merging Write Buffer to Reduce Miss Penalty

•Write buffer to allow processor to continue while waiting to write to memory

•If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry -If so, new data are combined with that entry

•Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory

•The Sun T1 (Niagara) processor, among many others, uses write merging

To illustrate write merging

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

9. Reducing Misses by Compiler Optimizations

•McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software

• **Instructions**
– Reorder procedures in memory so as to reduce conflict misses
– Profiling to look at conflicts (using tools they developed)
• **Data**
– Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
– Loop Interchange: change nesting of loops to access data in order stored in memory
– Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
– Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows
Compiler Optimizations- Reduction comes from software (no Hw ch.)

Loop Interchange

•Motivation: some programs have nested loops that access data in nonsequential order
•Solution: Simply exchanging the nesting of the loops can make the code access the data in the order it is stored =>
reduce misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded
Loop Interchange Example

```
/* Before */
for (j = 0; j < 100; j = j+1)
for (i = 0; i < 5000; i = i+1)
x[i][j] = 2 * x[i][j];
/* After */
for (i = 0; i < 5000; i = i+1)
for (j = 0; j < 100; j = j+1)
x[i][j] = 2 * x[i][j];
```

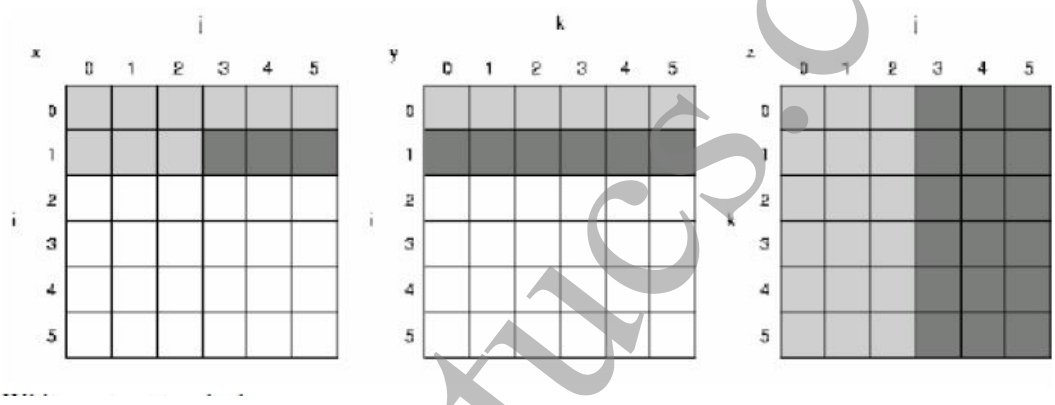
Blocking
•Motivation: multiple arrays, some accessed by rows and some by columns
•Storing the arrays row by row (row major order) or column by column (column major order) does not help: both rows and columns are used in every iteration of the loop (Loop Interchange cannot help)
•Solution: instead of operating on entire rows and columns of an array, blocked algorithms operate on submatrices or blocks => maximize accesses to the data loaded into the cache before the data is replaced

Blocking Example

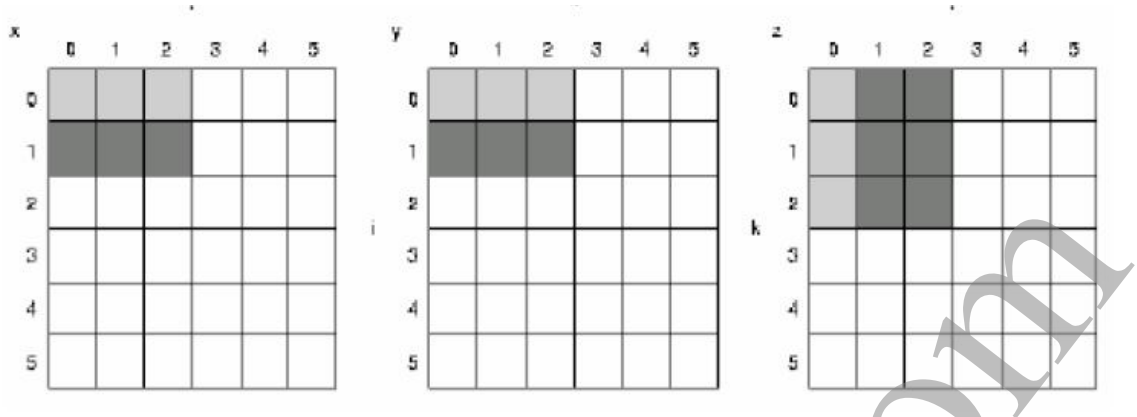
```
/* Before */
for (i = 0; i < N; i = i+1)
for (j = 0; j < N; j = j+1)
{r = 0;
for (k = 0; k < N; k = k+1){
r = r + y[i][k]*z[k][j];};
x[i][j] = r;
};
```

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
for (j = jj; j < min(jj+B,N); j = j+1)
{r = 0;
for (k = kk; k < min(kk+B,N); k = k + 1)
r = r + y[i][k]*z[k][j];
x[i][j] = x[i][j] + r;
};
```

Snapshot of x, y, z when
i=1
White:



White: not yet touched
Light: older access
Dark: newer access Before….
The Age of Accesses to x, y, Z



Note in contrast to previous Figure, the smaller number of elements accessed

Merging Arrays

•Motivation: some programs reference multiple arrays in the same dimension with the same indices at the same time =>
these accesses can interfere with each other,leading to conflict misses

•Solution: combine these independent matrices into a single compound array, so that a single cache block can contain the desired elements

Merging Arrays Example

Loop Fusion

• Some programs have separate sections of code that access with the same loops, performing different computations on the common data

• Solution:
"Fuse" the code into a single loop =>
the data that are fetched into the cache can be used repeatedly before being swapped out => reducing misses via improved temporal locality

Loop Fusion Example

Summary of Compiler Optimizations- to Reduce Cache Misses (by hand)

10. Reducing Misses by Hardware Prefetching of Instructions & Data

Prefetching relies on having extra memory bandwidth that can be used without penalty

• Instruction Prefetching
– Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
– Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer

Data Prefetching

– Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages

– Prefetching invoked if 2 successive L2 cache misses to a page,if distance between those cache blocks is < 256 bytes

11. Reducing Misses by Software Prefetching Data

•Data Prefetch

•– Load data into register (HP PA-RISC loads)

•– Cache Prefetch: load into cache

•(MIPS IV, PowerPC, SPARC v. 9)

•– Special prefetching instructions cannot cause faults;

•a form of speculative execution

•• Issuing Prefetch Instructions takes time

•– Is cost of prefetch issues < savings in reduced misses?

•– Higher superscalar reduces difficulty of issue bandwi

Compiler Optimization vs. Memory Hierarchy Search

Compiler tries to figure out memory hierarchy optimizations

• New approach: "Auto-tuners" 1st run variations of program on computer to find best combinations of optimizations (blocking, padding, …) and algorithms, then produce C code to be compiled for that computer

• "Auto-tuner" targeted to numerical method

– E.g., PHiPAC (BLAS), Atlas (BLAS), Sparsity (Sparse linear algebra), Spiral (DSP), FFT-W

Cache Optimization Summary

Comparison of the 11 techniques

Technique	Hit Time	Band-width	Miss penalty	Miss rate	HW cost/ complexity	Comment
Small and simple caches	+			–	0	Trivial; widely used
Way-predicting caches	+				1	Used in Pentium 4
Trace caches	+				3	Used in Pentium 4
Pipelined cache access	–	+			1	Widely used
Nonblocking caches		+	+		3	Widely used
Banked caches		+			1	Used in L2 of Opteron and Niagara
Critical word first and early restart			+		2	Widely used
Merging write buffer			+		1	Widely used with write through
Compiler techniques to reduce cache misses				+	0	Software is a challenge; some computers have compiler option
Hardware prefetching of instructions and data			+	+	2 instr., 3 data	Many prefetch instructions; AMD Opteron prefetches data
Compiler-controlled prefetching			+	+	3	Needs nonblocking cache; in many CPUs

Main Memory Background

Performance of Main Memory:

Latency: Cache Miss Penalty

- Access Time: time between request and word arrives
- Cycle Time: time between requests

Bandwidth: I/O & Large Block Miss Penalty (L2)

Main Memory is DRAM: Dynamic Random Access Memory

Dynamic since needs to be refreshed periodically (8 ms, 1% time)

Addresses divided into 2 halves (Memory as a 2D matrix):

- RAS or Row Access Strobe
- CAS or Column Access Strobe

Cache uses SRAM: Static Random Access Memory

No refresh (6 transistors/bit vs. 1 transistor

- Size: DRAM/SRAM - 4-8,
- Cost/Cycle time: SRAM/DRAM - 8-16

Main Memory Deep Background

- "Out-of-Core", "In-Core," "Core Dump"?
- "Core memory"?
- Non-volatile, magnetic
- Lost to 4 Kbit DRAM (today using 512Mbit DRAM)
- Access time 750 ns, cycle time 1500-3000 ns

DRAM logical organization (4 Mbit)

Quest for DRAM Performance

1. Fast Page mode

– Add timing signals that allow repeated accesses to row buffer without nother row access time

– Such a buffer comes naturally, as each array will buffer 1024 to 2048 bits for each access

2. Synchronous DRAM (SDRAM)

– Add a clock signal to DRAM interface, so that the repeated transfers would not bear overhead to synchronize with DRAM controller

3. Double Data Rate (DDR SDRAM)

– Transfer data on both the rising edge and falling edge of the DRAM clock signal I doubling the peak data rate

– DDR2 lowers power by dropping the voltage from 2.5 to 1.8 volts + offers higher clock rates: up to 400 MHz

– DDR3 drops to 1.5 volts + higher clock rates: up to 800 MHz

4.Improved Bandwidth, not Latency

DRAM name based on Peak Chip Transfers / Sec

DIMM name based on Peak DIMM MBytes / Sec

Need for Error Correction!

• Motivation:
  – Failures/time proportional to number of bits!
  – As DRAM cells shrink, more vulnerable
• Went through period in which failure rate was low enough without error correction that people didn't do correction
  – DRAM banks too large now
  – Servers always corrected memory systems
• Basic idea: add redundancy through parity bits
  – Common configuration: Random error correction
• SEC-DED (single error correct, double error detect)
• One example: 64 data bits + 8 parity bits (11% overhead)
  – Really want to handle failures of physical components as well
• Organization is multiple DRAMs/DIMM, multiple DIMMs
• Want to recover from failed DRAM and failed DIMM!
• "Chip kill" handle failures width of single DRAM chip

DRAM Technology

• Semiconductor Dynamic Random Access Memory
• Emphasize on cost per bit and capacity
• Multiplex address lines ı cutting # of address pins in half
– Row access strobe (RAS) first, then column access strobe (CAS)
– Memory as a 2D matrix – rows go to a buffer
– Subsequent CAS selects subrow
• Use only a single transistor to store a bit
– Reading that bit can destroy the information
– Refresh each bit periodically (ex. 8 milliseconds) by writing back
• Keep refreshing time less than 5% of the total time
• DRAM capacity is 4 to 8 times that of SRAM
• DIMM: Dual inline memory module
– DRAM chips are commonly sold on small boards called DIMMs
– DIMMs typically contain 4 to 16 DRAMs
• Slowing down in DRAM capacity growth
– Four times the capacity every three years, for more than 20 years
– New chips only double capacity every two year, since 1998
• DRAM performance is growing at a slower rate
– RAS (related to latency): 5% per year
  – CAS (related to bandwidth): 10%+ per year
  –

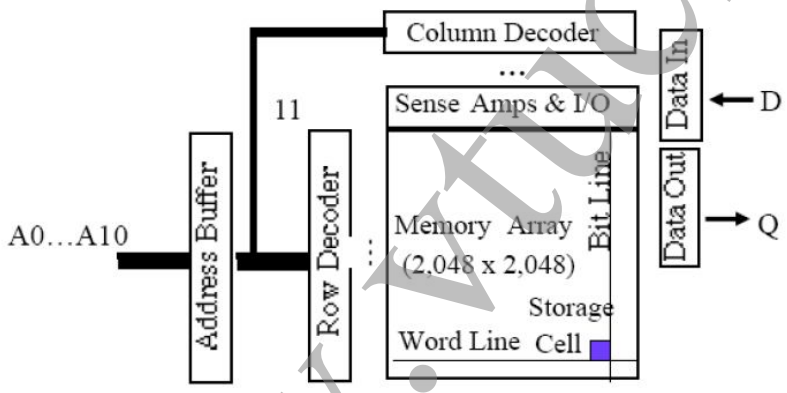
RAS improvement

SRAM Technology

• Cache uses SRAM: Static Random Access Memory
• SRAM uses six transistors per bit to prevent the information from being disturbed when read
_no need to refresh
– SRAM needs only minimal power to retain the charge in
the standby mode _ good for embedded applications

– No difference between access time and cycle time for
SRAM
• Emphasize on speed and capacity
– SRAM address lines are not multiplexed
• SRAM speed is 8 to 16x that of DRAM

Improving Memory Performance
in a Standard DRAM Chip
• Fast page mode: time signals that allow repeated accesses to buffer without another row access time
• Synchronous RAM (SDRAM): add a clock signal to DRAM interface, so that the repeated transfer would not bear overhead to synchronize with the controller
– Asynchronous DRAM involves overhead to sync with controller
– Peak speed per memory module 800—1200MB/sec in 2001
• Double data rate (DDR): transfer data on both the rising edge and falling edge of DRAM clock signal
– Peak speed per memory module 1600—2400MB/sec in 2001



Protection:
Virtual Memory and Virtual Machines

Slide Sources: Based on "Computer Architecture" by Hennessy/Patterson.
Supplemented from various freely downloadable sources
Security and Privacy
- •Innovations in Computer Architecture and System software
- •Protection through Virtual Memory
- •Protection from Virtual Machines
  - –Architectural requirements
  - –Performance

Protection via Virtual Memory
- •Processes
  - –Running program
  - –Environment (state) needed to continue running it
- •Protect Processes from each other

–Page based virtual memory including TLB which caches page table entries –Example: Segmentation and paging in 80x86

Processes share hardware without interfering with each other

- •Provide User Process and Kernel Process
- •Readable portion of Processor state:
  - –User supervisor mode bit
  - –Exception enable/disable bit
  - –Memory protection information
- •System call to transfer to supervisor mode
  - –Return like normal subroutine to user mode
- •Mechanism to limit memory access

Memory protection

•Virtual Memory
–Restriction on each page entry in page table
–Read, write, execute privileges
–Only OS can update page table
–TLB entries also have protection field
•Bugs in OS
–Lead to compromising security
–Bugs likely due to huge size of OS code
Protection via Virtual Machines
Virtualization
• Goal:
– Run multiple instances of different OS on the same hardware
– Present a transparent view of one or more environments (M-to-N mapping of M "real" resources, N "virtual" resources)
Protection via Virtual Machines
Virtualization- cont.
• Challenges:
– Have to split all resources (processor, memory, hard drive, graphics card, networking card etc.) among the different OS -> virtualize the resources

- – The OS can not be aware that it is using virtual resources instead of real resources

Problems with virtualization

• Two components when using virtualization:
– Virtual Machine Monitor (VMM)
– Virtual Machine(s) (VM)
• Para-virtualization:
– Operating System has been modified in order to run as a VM
• 'Fully' Virtualized:

- – No modification required of an OS to run as a VM
- –

Virtual Machine Monitor-'hypervisor'

• Isolates the state of each guest OS from each other
• Protects itself from guest software

• Determines how to map virtual resources to physical resources
– Access to privileged state
– Address translation
– I/O
– Exceptions and interrupts
• Relatively small code (compared to an OS)
• VMM must run in a higher privilege mode than guest OS

Managing Virtual Memory
• Virtual memory offers many of the features required for hardware virtualization
– Separates the physical memory onto multiple processes
– Each process 'thinks' it has a linear address space of full size
– Processor holds a page table translating virtual addresses used by a process and the according physical memory
– Additional information restricts processes from
• Reading a page of on another process or
• Allow reading but not modifying a memory page or
• Do not allow to interpret data in the memory page as instructions and do not try to execute them
• Virtual Memory management thus requires
– Mechanisms to limit memory access to protected memory
– At least two modes of execution for instructions
• Privileged mode: an instruction is allowed to do what it whatever it wants -> kernel mode for OS
• Non-privileged mode: user-level processes
• Intel x86 Architecture: processor supports four levels
– Level 0 used by OS
– Level 3 used by regular applications
•Provide mechanisms to go from non-privileged mode to privileged mode -> system call
•Provide a portion of processor state that a user process can read but not modify
• E.g. memory protection information
• Each guest OS maintains its page tables to do the mapping from virtual address to physical address
• Most simple solution: VMM holds an additional table which maps the physical address of a guest OS onto the 'machine address'
– Introduces a third level of redirection for every memory access
• Alternative solution: VMM maintains a shadow page table of each guest OS
– Copy of the page table of the OS
– Page tables still works with regular physical addresses
– Only modifications to the page table are intercepted by the VMM

Protection via Virtual Machines
-some definitions
• VMs include all emulation methods to provide a standard software interface
• Different ISAs can be used (emulated) on the native machine

• When the ISAs match the hardware we call it (operating) system virtual machines
• Multiple OSes all share the native hardware

Cost of Processor Virtualisation
•VM is much smaller than traditional OS
•Isolation portion is only about 10000 lines for a VMM
•Processor bound programs have very little virtualisation overhead
•I/O bound jobs have more overhead
•ISA emulation is costly

Other benefits of VMs
• Managing software
–Complete software stack
–Old Oses like DOS
–Current stable OS
–Next OS release
• Managing Hardware
–Multiple servers avoided
–VMs enable hardware sharing
–Migration of a running VM to another m/c
•For balancing load or evacuate from failing HW
Requirements of a VMM
•Guest sw should behave exactly on VM as if on native hw
•Guest sw should not be able to change allocation of RT resources directly
•Timer interrupts should be virtualised
•Two processor modes- system and user
•Priveleged subset of instruction available only in system mode

More issues on VM usage
•ISA support for virtual machine
–IBM360 support
–80x86 do no support
•Use of virtual memory
–Concept of **virtual- real- physical** memories
–Instead of extra indirection use shadow page table
•Virtualising I/Os
–More i/o
–More diversity
–Physical disks to partitioned virtual disks
–Network cards time sliced

UNIT - VIII

HARDWARE AND SOFTWARE FOR VLIW AND EPIC:

Introduction

Exploiting Instruction-Level Parallelism Statically

Detecting and Enhancing Loop-Level Parallelism

Scheduling and Structuring Code for Parallelism

Hardware Support for Exposing Parallelism

Predicated Instructions; Hardware Support for Compiler Speculation

The Intel IA-64 Architecture and Itanium Processor; Conclusions.

7 Hours

UNIT VIII

HARDWARE AND SOFTWARE FOR VLIW AND EPIC

Loop Level Parallelism- Detection and Enhancement

Static Exploitation of ILP

•Use compiler support for increasing parallelism
–Supported by hardware
•Techniques for eliminating some types of dependences
–Applied at compile time (no run time support)
•Finding parallelism
•Reducing control and data dependencies
•Using speculation
Unrolling Loops – High-level
–**for (i=1000; i>0; i=i-1) x[i] = x[i] + s;**
–C equivalent of unrolling to block four iterations into one:
–for (i=250; i>0; i=i-1)
{
x[4*i] = x[4*i] + s;
x[4*i-1] = x[4*i-1] + s;
x[4*i-2] = x[4*i-2] + s;
x[4*i-3] = x[4*i-3] + s;
}
Enhancing Loop-Level Parallelism
•Consider the previous running example:
–for (i=1000; i>0; i=i-1) x[i] = x[i] + s;
–there is no *loop-carried dependence* – where data used in a later iteration depends on data produced in an earlier one
–in other words, all iterations could (conceptually) be executed in parallel
•Contrast with the following loop:
–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 */ }
–*what are the dependences?*
A Loop with Dependences
•**For the loop:**
–**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 */ }

–***what are the dependences?***
•There are two different dependences:
–**loop-carried: (prevents parallel operation of iterations)**
•S1 computes A[i+1] using value of A[i] computed in previous iteration
•S2 computes B[i+1] using value of B[i] computed in previous iteration
–**not loop-carried: (parallel operation of iterations is ok)**

•S2 uses the value A[i+1] computed by S1 in the *same* iteration
•The loop-carried dependences in this case *force* successive iterations of the loop to execute in series. *Why?*
–S1 of iteration *i* depends on S1 of iteration *i-1* which in turn depends on …, etc.
Another Loop with Dependences
•Generally, loop-carried dependences *hinder* ILP
–if there are no loop-carried dependences all iterations could be executed in parallel
–even if there are loop-carried dependences it may be possible to parallelize the loop – an analysis of the dependences is required…

•For the loop:
–for (i=1; i<=100; i=i+1) { A[i] = A[i] + B[i]; /* S1 */
B[i+1] = C[i] + D[i]; /* S2 */ }

–***what are the dependences?***
•There is one loop-carried dependence:
–S1 uses the value of B[i] computed in a previous iteration by S2
–but this *does not force* iterations to execute in series. *Why…?*
–…because S1 of iteration *i* depends on S2 of iteration *i-1*…, and the *chain of dependences stops here*!

Parallelizing Loops with Short Chains of Dependences

•Parallelize the loop:
–for (i=1; i<=100; i=i+1) { A[i] = A[i] + B[i]; /* S1 */
B[i+1] = C[i] + D[i]; /* S2 */ }
•Parallelized code:
–A[1] = A[1] + B[1];
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];
–the dependence between the two statements in the loop is no longer loop-carried and iterations of the loop may be executed in parallel
–

Loop-Carried Dependence Detection: affine array index: a x i+b

To detect loop-carried dependence in a loop, the Greatest Common Divisor (GCD) test can be used by the compiler, which is based on the following:
If an array element with index: a x i + b is stored and element: c x i + d of the same array is loaded later where index runs from m to n, a dependence exists if the following two conditions hold:
1.There are two iteration indices, j and k , m <= j , k <= n
(within iteration limits)
2.The loop stores into an array element indexed by:
a x j + b
and later loads from the same array the element indexed by:
c x k + d

Thus:
a x j + b = c x k + d

The Greatest Common Divisor (GCD) Test

If a loop carried dependence exists, then :

GCD(c, a) must divide (d-b)

The GCD test is sufficient to guarantee no loop carried dependence
However there are cases where GCD test succeeds but no dependence exits because GCD test does not take loop bounds into account
Example:

```
for (i=1; i<=100; i=i+1) {
x[2*i+3] = x[2*i] * 5.0;
}
```

a = 2 b = 3 c = 2 d = 0
GCD(a, c) = 2
d - b = -3
2 does not divide -3 _ No loop carried dependence possible.

Example- Loop Iterations to be Independent

Finding multiple types of dependences

```
for (i=1; i<=100; i=i+1) {
Y[i] = X[i] / c; /* S1 */
X[i] = X[i] + c; /* S2 */
Z[i] = Y[i] + c; /* S3 */
Y[i] = c - Y[i]; /* S4 */ }
```

Answer The following dependences exist among the four statements:
1. There are true dependences from S1 to S3 and from S1 to S4 because of Y[i]. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on X[i].
3. There is an antidependence from S3 to S4 for Y[i].
4. There is an output dependence from S1 to S4, based on Y[i].

Eliminating false dependencies

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=1; i<=100; i=i+1 {
/* Y renamed to T to remove output dependence */
T[i] = X[i] / c;
/* X renamed to X1 to remove antidependence */
X1[i] = X[i] + c;
/* Y renamed to T to remove antidependence */
Z[i] = T[i] + c;
Y[i] = c - T[i];
}
```

Drawback of dependence analysis

•When objects are referenced via pointers rather than array indices (but see discussion

below)
• When array indexing is indirect through another array, which happens with many representations of sparse arrays
• When a dependence may exist for some value of the inputs, but does not exist in actuality when the code is run since the inputs never take on those values
• When an optimization depends on knowing more than just the possibility of a dependence, but needs to know on *which* write of a variable does a read of that variable depend

Points-to analysis
Relies on information from three major sources:
1. Type information, which restricts what a pointer can point to.
2. Information derived when an object is allocated or when the address of an object is taken, which can be used to restrict what a pointer can point to. For example, if *p* always points to an object allocated in a given source line and *q* never points to that object, then *p* and *q* can never point to the same object.
3. Information derived from pointer assignments. For example, if *p* may be assigned the value of *q,* then *p* may point to anything *q* points to.

Eliminating dependent computations
copy propagation, used to simplify sequences like the following:

```
DADDUI R1,R2,#4
DADDUI R1,R1,#4
to
DADDUI R1,R2,#8
```

Tree height reduction
•they reduce the height of the tree structure representing a computation, making it wider but shorter.

Recurrence
Recurrences are expressions whose value on one iteration is given by a function that depends onthe previous iterations.
sum = sum + x;
sum = sum + x1 + x2 + x3 + x4 + x5;
If unoptimized requires five dependent operations, but it can be rewritten as
sum = ((sum + x1) + (x2 + x3)) + (x4 + x5);
evaluated in only three dependent operations.

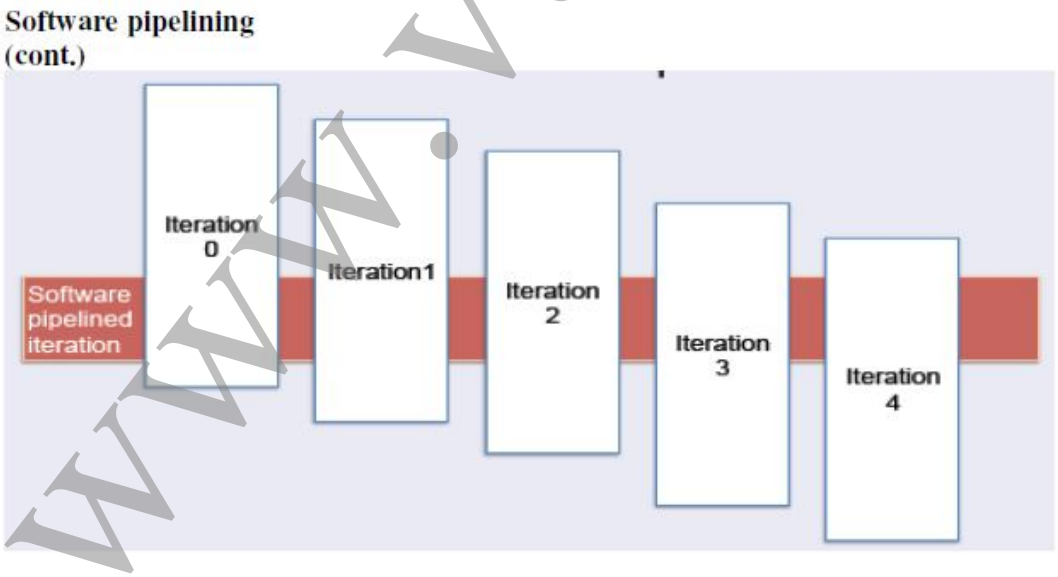
Scheduling and Structuring Code for Parallelism
Static Exploitation of ILP
•Use compiler support for increasing parallelism
–Supported by hardware
•Techniques for eliminating some types of dependences
–Applied at compile time (no run time support)

•Finding parallelism
•Reducing control and data dependencies
•Using speculation

Techniques to increase the amount of ILP
•For processor issuing more than one instruction on every clock cycle.
–Loop unrolling,
–software pipelining,
–trace scheduling, and
–superblock scheduling

Software pipelining
•Symbolic loop unrolling
•Benefits of loop unrolling with reduced code size
•Instructions in loop body selected from different loop iterations
•Increase distance between dependent instructions in

Software pipelined loop
Loop: SD F4,16(R1) #store to v[i]
ADDD F4,F0,F2 #add to v[i-1]
LD F0,0(R1) #load v[i-2]
ADDI R1,R1,-8
BNE R1,R2,Loop
5 cycles/iteration (with dynamic scheduling and renaming)
Need startup/cleanup code



SW pipelining example
Iteration i: L.D F0,0(R1)

```
                ADD.D       F4,F0,F2
                S.D         F4,0(R1)
Iteration i+1:  L.D         F0,0(R1)
                ADD.D       F4,F0,F2
                S.D         F4,0(R1)
Iteration i+2:  L.D         F0,0(R1)
                ADD.D       F4,F0,F2
                S.D         F4,0(R1)
```

SW pipelined loop with startup and cleanup code

```
#startup, assume i runs from 0 to n
        ADDI    R1,R1-16            #point to v[n-2]
        LD      F0,16(R1)           #load v[n]
        ADDD    F4,F0,F2            #add v[n]
        LD      F0,8(R1)            #load v[n-1]
        #body for (i=2;i<=n-2;i++)
Loop:   SD      F4,16(R1)           #store to v[i]
        ADDD    F4,F0,F2            #add to v[i-1]
        LD F0,0(R1)                 #load v[i-2]
        ADDI    R1,R1,-8
        BNE     R1,R2,Loop
#cleanup
        SD      F4,8(R1)            #store v[1]
        ADDD    F4,F0,F2            #add v[0]
        SD      F4,0(R1)            #store v[0]
```

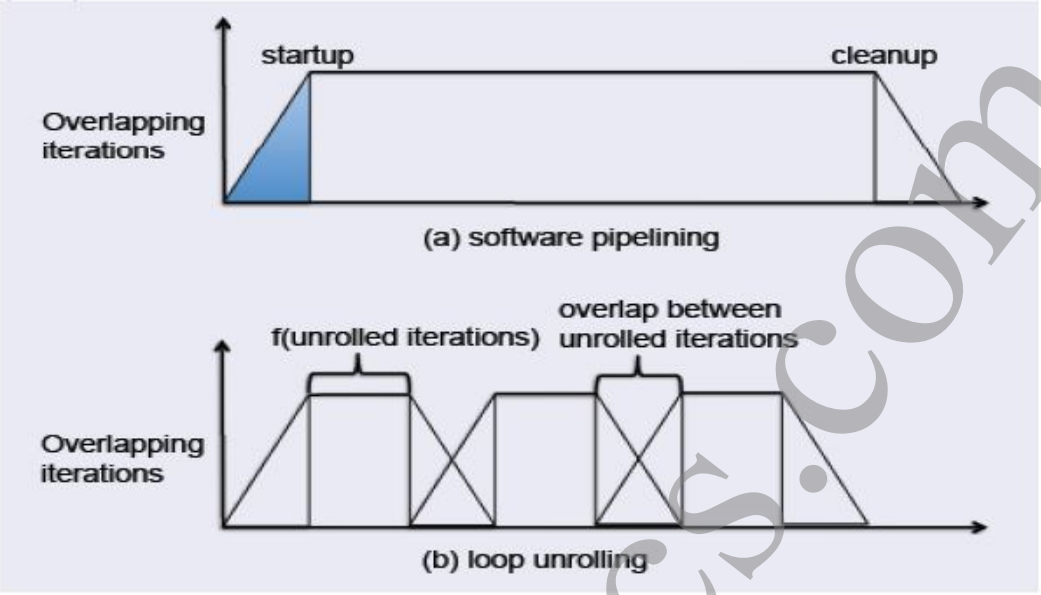
Software pipelining versus unrolling

•**Performance effects of SW pipelining vs. unrolling**

–Unrolling reduces loop overhead per iteration

–SW pipelining reduces startup-cleanup pipeline overhead

Software pipelining versus unrolling (cont.)
Software pipelining



Advantages

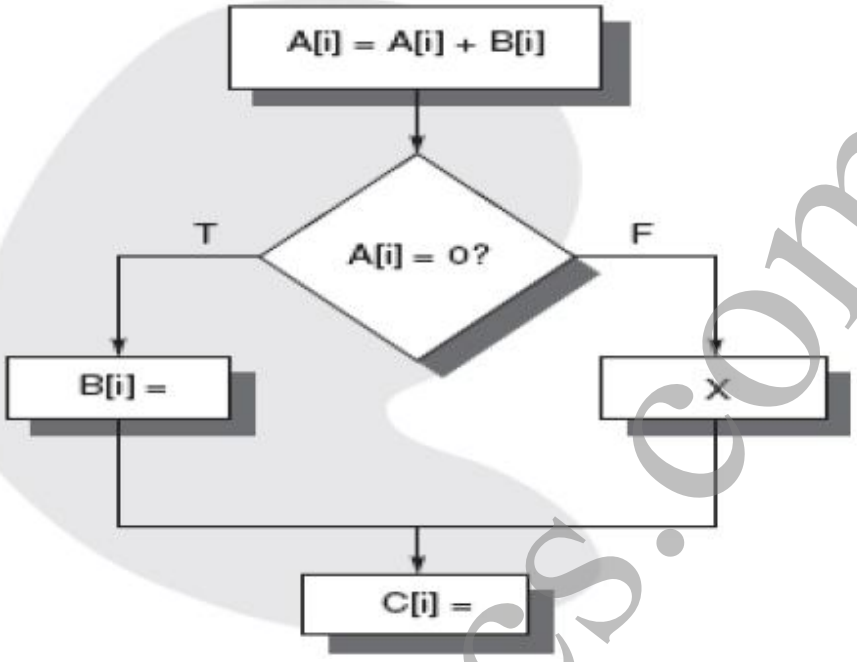
- Less code space than conventional unrolling
- Loop runs at peak speed during steady state
- Overhead only at loop initiation and termination
- Complements unrolling

Disadvantages

- Hard to overlap long latencies
- Unrolling combined with SW pipelining
- Requires advanced compiler transformations

Global Code Scheduling

•Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences.



Global code scheduling

•aims to compact a code fragment with internal control
- –structure into the shortest possible sequence
- –that preserves the data and control dependences

•Data dependences are overcome by unrolling

•In the case of memory operations, using dependence analysis to determine if two references refer to the same address.

•Finding the shortest possible sequence of dependent instructions- critical path

•Reduce the effect of control dependences arising from conditional nonloop branches by moving code.

•Since moving code across branches will often affect the frequency of execution of such code, effectively using global code motion requires estimates of the relative frequency of different paths.

•if the frequency information is accurate, is likely to lead to faster code.

Global code scheduling- cont.

•Global code motion is important since many inner loops contain conditional statements.

•Effectively scheduling this code could require that we move the assignments to B and C to earlier in the execution sequence, before the test of A.

Factors for compiler

•Global code scheduling is an extremely complex problem
- –What are the relative execution frequencies
- –What is the cost of executing the computation

–How will the movement of B change the execution time
–Is B the best code fragment that can be moved
–What is the cost of the compensation code

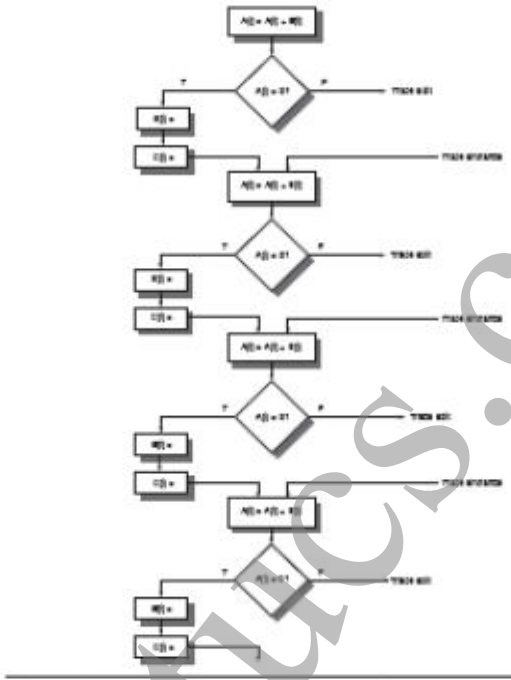
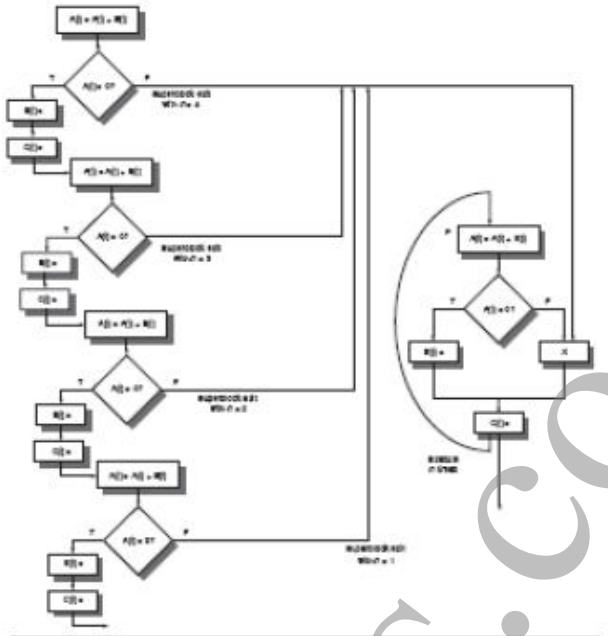


Figure G.4 This trace is obtained by assuming that the program fragment in Figure G.3 is the inner loop and unwinding it four times, treating the shaded portion in Figure G.3 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.

Using super blocks

Trace Scheduling:
•*Focusing on the Critical Path*



Using super blocks

Code generation sequence

```
        LD R4,0(R1)         ;load A
        LD R5,0(R2)         ;load B
        DADDU R4,R4,R5      ;Add to A
        SD R4,0(R1)         ;Store A
...
        BNEZ R4,elsepart    ;Test A
... ;then part
        SD ...,0(R2)        ;Stores to B
...
        J join              ;jump over else
elsepart:   ...             ;else part
        X                   ;code for X
...
join:       ...             ;after if
                SD ...,0(R3)        ;store C[i]
```

Trace Scheduling,
Superblocks and Predicated Instructions

•For processor issuing more than one instruction on every clock cycle.
- –Loop unrolling,
- –software pipelining,

–trace scheduling, and
–superblock scheduling

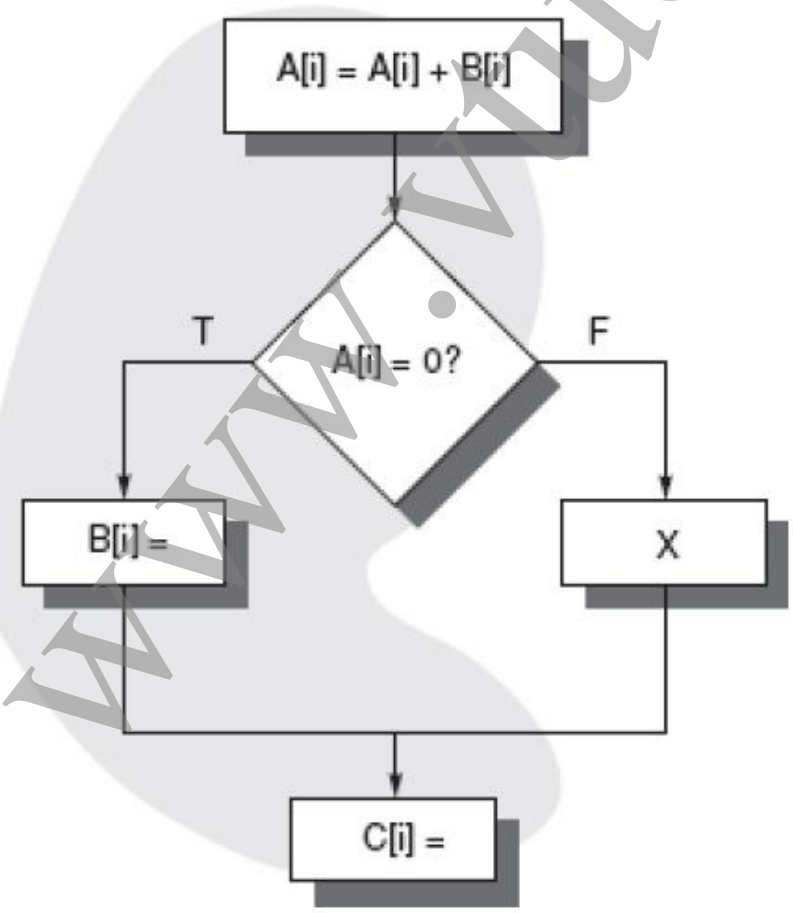
Trace Scheduling

•Used when
– Predicated execution is not supported
– Unrolling is insufficient
• Best used
– If profile information clearly favors one path over the other
• Significant overheads are added to the infrequent path
• Two steps :
– Trace Selection
– Trace Compaction
–

Trace Selection

Likely sequence of basic blocks that can be put together
– Sequence is called a trace
• What can you select?
– Loop unrolling generates long traces
– Static branch prediction forces some straight-line code behavior

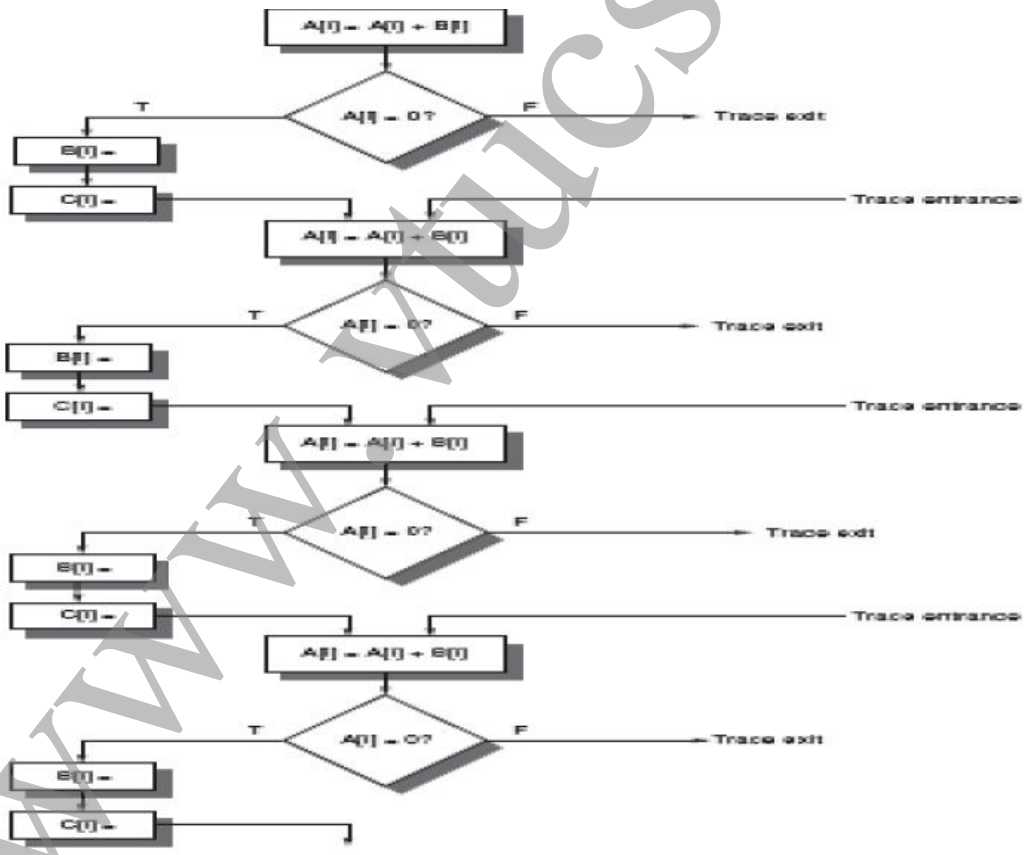
Trace Selection



(cont.)
Trace Example
If the shaded portion in previous code was frequent path and it was unrolled 4 times : _
• Trace exits are jumps off the frequent path
• Trace entrances are returns to the trace

Trace Compaction

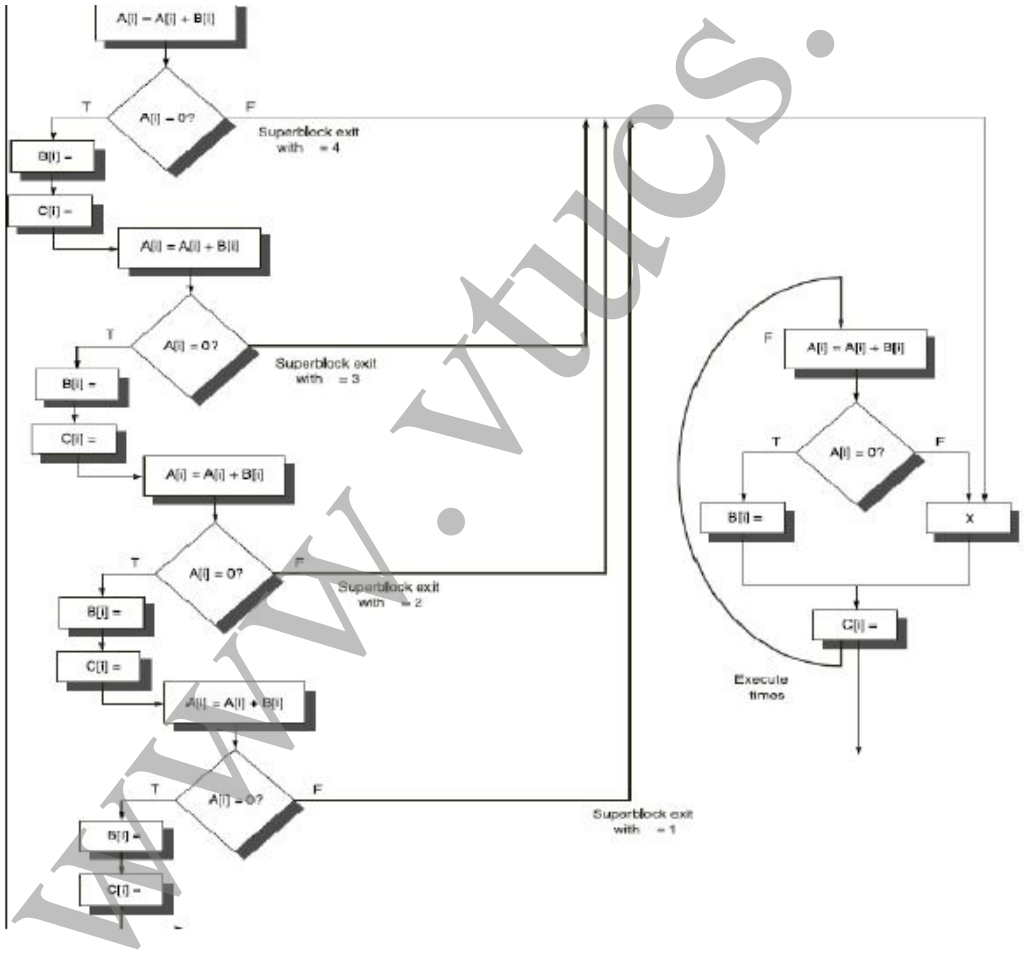
•Squeeze these into smaller number of wide instructions
• Move operations as early as it can be in a trace
• Pack the instructions into as few wide instructions as possible
• Simplifies the decisions concerning global code motion
– All branches are viewed as jumps into or out of the trace
• Bookkeeping
– Cost is assumed to be little
• Best used in scientific code with extensive loops



Super Blocks for Global Scheduling

- Motivation :
  - Entries and exits into trace schedule code are complicated
  - Compiler cannot do a good cost analysis about compensation code
- Superblocks
  - Are like traces
  - One entry point
- Unlike traces
  - Different exit points
  - Common in for loops
- Single entry and exit points
- Code motion across exit only need be considered

Superblock Construction



• **Tail duplication**
– Creates a separate block that corresponds to the portion of trace after the entry
• When proceeding as per prediction – Take the path of superblock code
• When exit from
superblock
– Residual loop that handles rest of the iterations
–

Analysis on Superblocks
• Reduces the complexity of bookkeeping and scheduling
– Unlike the trace approach
• Can have larger code size though
• Assessing the cost of duplication
• Compilation process is not simple any more

H/W Support : Conditional Execution
• Also known as Predicated Execution
– Enhancement to instruction set
– Can be used to eliminate branches
– All control dependences are converted to data dependences
• Instruction refers to a condition
– Evaluated as part of the execution
• True?
– Executed normally
• False?
– Execution continues as if the instruction were a no-op
• Example :
– Conditional move between registers
–

Example
if (A==0)
S = T;
Straightforward Code
BNEZ R1, L;
ADDU R2, R3, R0
L:
Conditional Code
CMOVZ R2, R3, R1
Annulled if R1 is not 0
Conditional Instruction …
• Can convert control to data dependence
• In vector computing, it's called *if conversion*.
• Traditionally, in a pipelined system
– Dependence has to be resolved closer to front of pipeline
• For conditional execution
– Dependence is resolved at end of pipeline, closer to the register write

Another example

• A = abs(B)
if (B < 0)
A = -B;
else
A = B;

• Two conditional moves
• One unconditional and one conditional move
• The branch condition has moved into the instruction
– Control dependence becomes data dependence
–

Limitations of Conditional Moves

• Conditional moves are the simplest form of predicated instructions
• Useful for short sequences
• For large code, this can be inefficient
– Introduces many conditional moves
• Some architectures support full predication
– All instructions, not just moves
• Very useful in global scheduling
– Can handle nonloop branches nicely
– Eg : The whole if portion can be predicated if the frequent path is not taken

Example

LW	R1, 40(R2)	ADD	R3,R4,R5
		ADD	R6, R3, R7
BEQZ	R10, L		
LW	R8, 0(R10)		
LW	R9, 0(R8)		

• Assume : Two issues, one to ALU and one to memory; or branch by itself
• Wastes a memory operation slot in second cycle
• Can incur a data dependence stall if branch is not taken
– R9 depends on R8

Predicated Execution

Assume : LWC is predicated load and loads if third operand is not 0

LW	R1, 40(R2)	ADD	R3,R4,R5
LWC	R8, 0(R10), R10	ADD	R6, R3, R7
BEQZ	R10, L		
LW	R9, 0(R8)		

• One instruction issue slot is eliminated
• On mispredicted branch, predicated instruction will not have any effect
• If sequence following the branch is short, the entire block of the code can be predicated

Some Complications

- Exception Behavior
  - – Must not generate exception if the predicate is false
- If R10 is zero in the previous example
  - – LW R8, 0(R10) can cause a protection fault
- If condition is satisfied
  - – A page fault can still occur
- Biggest Issue – Decide when to annul an instruction
  - – Can be done during issue
- Early in pipeline
- Value of condition must be known early, can induce stalls
  - – Can be done before commit
- Modern processors do this
- Annulled instructions will use functional resources
- Register forwarding and such can complicate implementation

Limitations of Predicated Instructions

• Annulled instructions still take resources
  – Fetch and execute atleast
  – For longer code sequences, benefits of conditional move vs branch is not clear
• Only useful when predicate can be evaluated early in the instruction stream
• What if there are multiple branches?
  – Predicate on two values?
• Higher cycle count or slower clock rate for predicated instructions
  – More hardware overhead
• MIPS, Alpha, Pentium etc support partial predication
• IA-64 has full predication

Hardware support for Compiler Speculation

H/W Support : Conditional Execution

• Also known as Predicated Execution

– Enhancement to instruction set

– Can be used to eliminate branches

– All control dependences are converted to data dependences

• Instruction refers to a condition

– Evaluated as part of the execution

• True?

– Executed normally

• False?

– Execution continues as if the instruction were a no-op

• Example :

– Conditional move between registers

Example

```
if (A==0)
S = T;
```

Straightforward Code

```
BNEZ R1, L;
ADDU R2, R3, R0
L:
```

Conditional Code

```
CMOVZ R2, R3, R1
```

Annulled if R1 is not 0

Conditional Instruction …

• Can convert control to data dependence

• In vector computing, it's called *if conversion*.

• Traditionally, in a pipelined system

– Dependence has to be resolved closer to front of pipeline

• For conditional execution

– Dependence is resolved at end of pipeline, closer to the register write

Another example

• A = abs(B)

```
if (B < 0)
A = -B;
else
A = B;
```

• Two conditional moves

• One unconditional and one conditional move

• The branch condition has moved into the instruction

– Control dependence becomes data dependence

Limitations of Conditional Moves

• Conditional moves are the simplest form of predicated instructions

• Useful for short sequences

• For large code, this can be inefficient
– Introduces many conditional moves
• Some architectures support full predication
– All instructions, not just moves
• Very useful in global scheduling
– Can handle nonloop branches nicely
– Eg : The whole if portion can be predicated if the frequent path is not taken
Example
• Assume : Two issues, one to ALU and one to memory; or branch by itself
• Wastes a memory operation slot in second cycle
• Can incur a data dependence stall if branch is not taken
  – R9 depends on R8

Predicated Execution
Assume : LWC is predicated load and loads if third operand is not 0
• One instruction issue slot is eliminated
• On mispredicted branch, predicated instruction will not have any effect
• If sequence following the branch is short, the entire block of the code can be predicated

Predication
Some Complications
• Exception Behavior
– Must not generate exception if the predicate is false
• If R10 is zero in the previous example
– LW R8, 0(R10) can cause a protection fault
• If condition is satisfied
– A page fault can still occur
• Biggest Issue – Decide when to annul an instruction
– Can be done during issue
-- Early in pipeline
• Value of condition must be known early, can induce stalls
– Can be done before commit
• Modern processors do this
• Annulled instructions will use functional resources
• Register forwarding and such can complicate implementation

Limitations of Predicated Instructions
• Annulled instructions still take resources
– Fetch and execute atleast
– For longer code sequences, benefits of conditional move vs branch is not clear
• Only useful when predicate can be evaluated early in the instruction stream
• What if there are multiple branches?
– Predicate on two values?
• Higher cycle count or slower clock rate for predicated instructions
– More hardware overhead
• MIPS, Alpha, Pentium etc support partial predication

• IA-64 has full predication

Preserve control and data flow, precise interrupts in Predication
•Speculative predicated instructions may not throw illegal exceptions
–LWC may not throw exception if R10 == 0
–LWC may throw recoverable page fault if R10 6= 0
•Instruction conversion to nop
–Early condition detection may not be possible due to data dependence
–Late condition detection incurs stalls and consumes pipeline resources needlessly
•Instructions may be dependent on multiple branches
•Compiler able to find instruction slots and reorder

Hardware support for speculation
Alternatives for handling speculative exceptions
•Hardware and OS ignore exceptions from speculative instructions
•Mark speculative instructions and check for exceptions
–Additional instructions to check for exceptions and recover
•Registers marked with poison bits to catch exceptions upon read
•Hardware buffers instruction results until instruction is no longer speculative

Exception classes
•Recoverable: exception from speculative instruction may harm performance, but not preciseness
•Unrecoverable: exception from speculative instruction compromises preciseness

Solution I: Ignore exceptions
HW/SW solution
•Instruction causing exception returns undefined value
•Value not used if instruction is speculative
•Incorrect result if instruction is non-speculative
–Compiler generates code to throw regular exception
•Rename registers receiving speculative results

Solution I: Ignore exceptions
Example

Non-speculative

```
# if (A==0) A=B; else A=A+4;
        LD   R1,0(R3)  ;load A
        BNEZ R1,L1     ;test A
        LD   R1,0(R2)  ;then load B
        J    L2
L1:     ADDI R1,R1,4   ;else
L2:     SD   R1,0(R3)  ;store A
```

Speculative

```
# if (A==0) A=B; else A=A+4;
        LD   R1,0(R3)  ;load A
        LD   R4,0(R2)  ;speculative load B
        BEQZ R1,L3     ;test A
        ADDI R4,R1,4   ;else
L3:     SD   R4,0(R3)  ;non-speculative store
```

Solution II: mark speculative instructions

```
# if (A==0) A=B; else A=A+4;
                LD      R1,0(R3)    ;load A
                SLD     R4,0(R2)    ;speculative load B
                BNEZ R1,L1  ;test A
                CHK     R4,recover  ;speculation check
                J       L2          ;skip else
        L1:     ADDI    R4,R1,4     ;else
        L2:     SD      R4,0(R3)    ;store A
recover:..
```

```
# if (A==0) A=B; else A=A+4;
                LD      R1,0(R3)    ;load A
                SLD     R4,0(R2)    ;speculative load B
                BNEZ R1,L1  ;test A
                CHK     R4,recover  ;speculation check
                J       L2          ;skip else
        L1:     ADDI    R4,R1,4     ;else
        L2:     SD      R4,0(R3)    ;store A
recover:..
```

•R4 marked with poison bit

•Use of R4 in SD raises exception if SLD raises exception
•Generate exception when result of offending instruction is used for the first time
•OS code needs to save poison bits during context switching

Solution IV HW mechanism like a ROB

- •Instructions are marked as speculative
- •How many branches speculatively moved
- •Action (T/NT) assumed by compiler
- •Usually only one branch
- •Other functions like a ROB

HW support for Memory Reference Speculation

- •Moving stores across loads
  - –To avoid address conflict
  - –Special instruction checks for address conflict
- •Left at original location of load instruction
- •Acts like a guardian
- •On speculative load HW saves address
  - –Speculation failed if a stores changes this address before check nstruction
- •Fix-up code re-executes all speculated instructions

IA-64 and Itanium Processor

Introducing The IA-64 Architecture

Itanium and Itanium2 Processor

Slide Sources: Based on "Computer Architecture" by Hennessy/Patterson.
Supplemented from various freely downloadable sources
IA-64 is an EPIC

•IA-64 largely depends on software for parallelism
• VLIW – Very Long Instruction Word
• EPIC – Explicitly Parallel Instruction Computer

VLIW points

•VLIW – Overview
– RISC technique
– Bundles of instructions to be run in parallel
– Similar to superscaling

- **Uses compiler instead of branch prediction hardware**

EPIC

•EPIC – Overview

– Builds on VLIW

– Redefines instruction format

– Instruction coding tells CPU how to process data

– Very compiler dependent

– Predicated execution

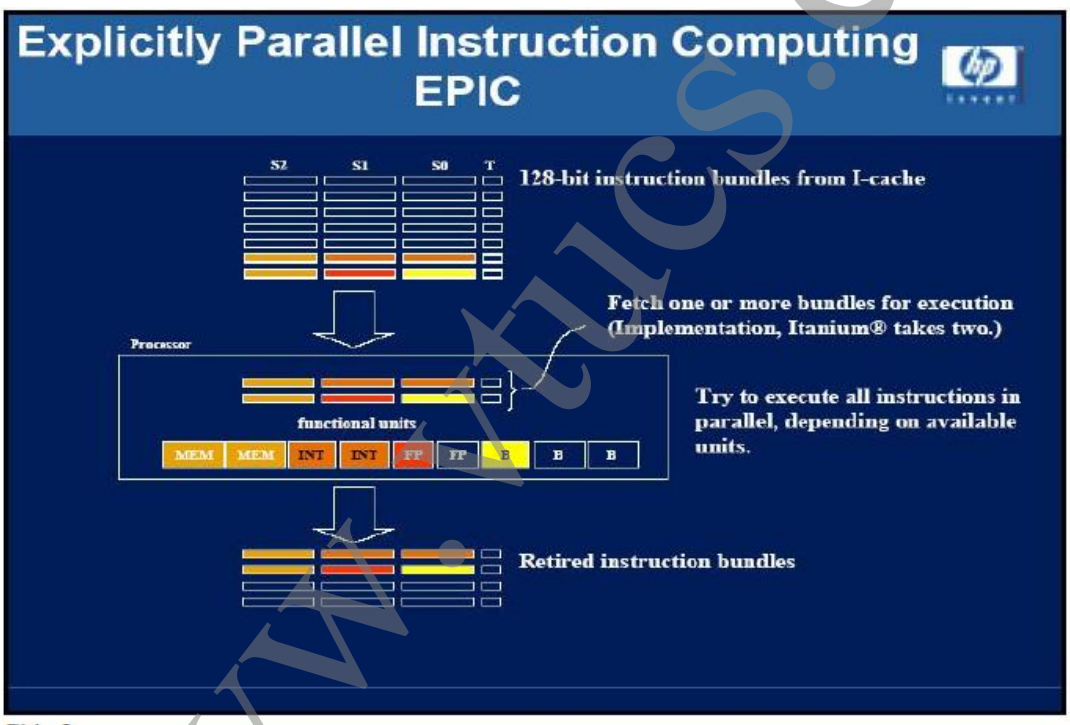
EPIC pros and cons

•EPIC – Pros:

– Compiler has more time to spend with code

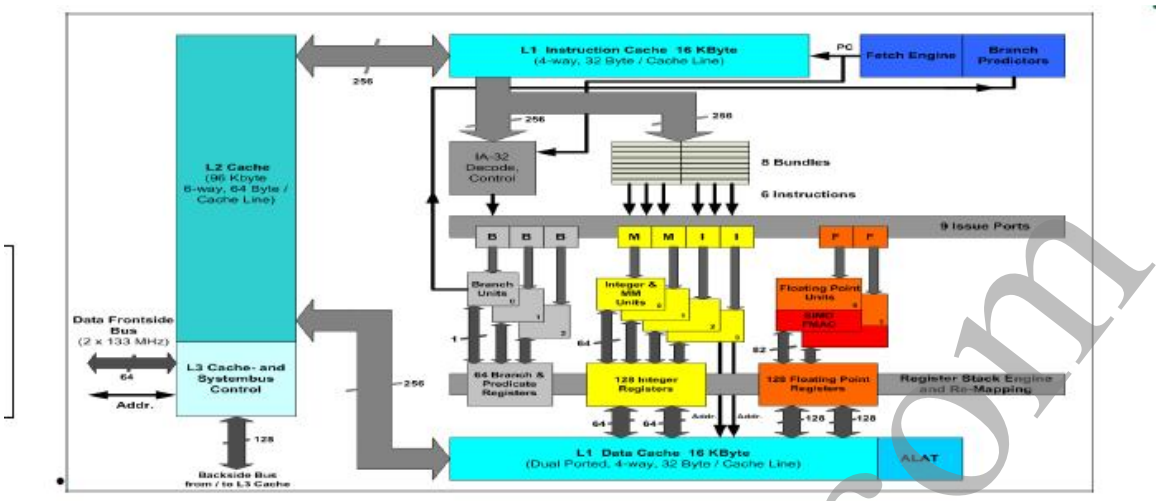
– Time spent by compiler is a one-time cost

- **Reduces circuit complexity**



Chip Layout

•*Itanium* Architecture Diagram

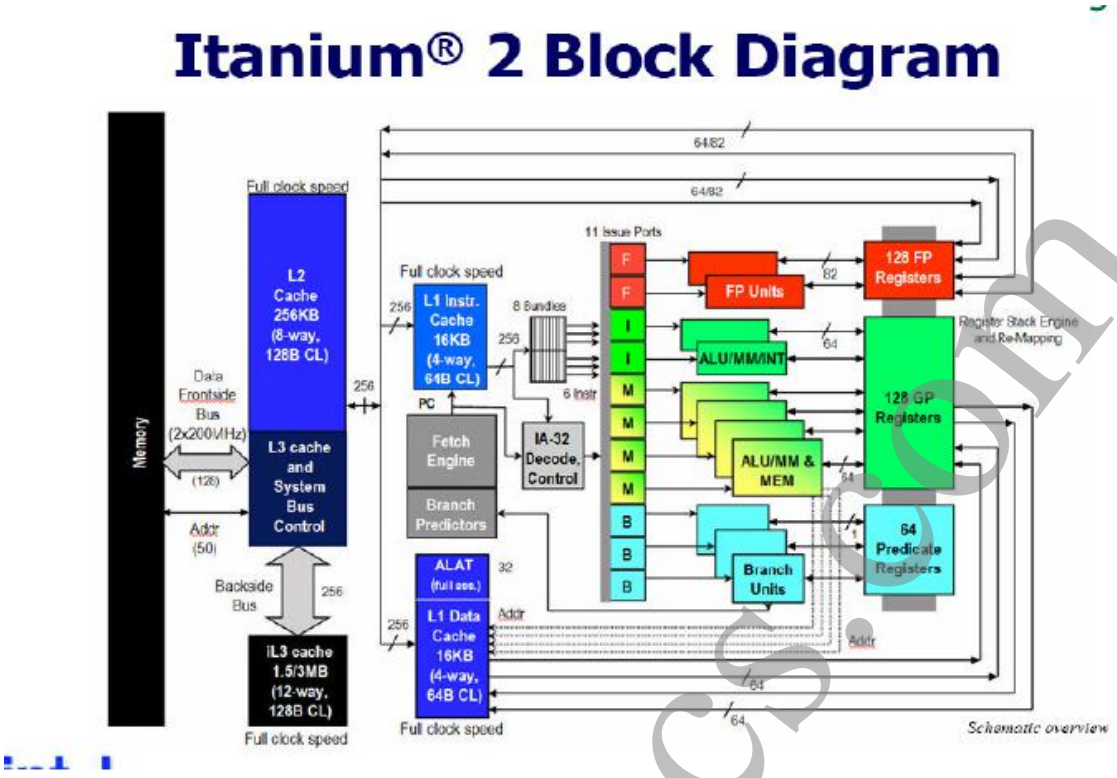


Itanium Specs

- •4 Integer ALU's
- •4 multimedia ALU's
- •2 Extended Precision FP Units
- •2 Single Precision FP units
- •2 Load or Store Units
- •3 Branch Units
- •10 Stage 6 Wide Pipeline
- •32k L1 Cache
- •96K L2 Cache
- •4MB L3 Cache(extern)þ
- •800Mhz Clock

Intel Itanium

•800 MHz
•10 stage pipeline
•Can issue 6 instructions (2 bundles) per cycle
•4 Integer, 4 Floating Point, 4 Multimedia, 2 Memory, 3 Branch Units
•32 KB L1, 96 KB L2, 4 MB L3 caches
•2.1 GB/s memory bandwidth



Itanium2 Specs

•6 Integer ALU's
•6 multimedia ALU's
•2 Extended Precision FP Units
•2 Single Precision FP units
•2 Load and Store Units
•3 Branch Units
•8 Stage 6 Wide Pipeline
•32k L1 Cache
•256K L2 Cache
•3MB L3 Cache(on die)þ
•1Ghz Clock initially
–Up to 1.66Ghz on Montvale

Itanium2 Improvements

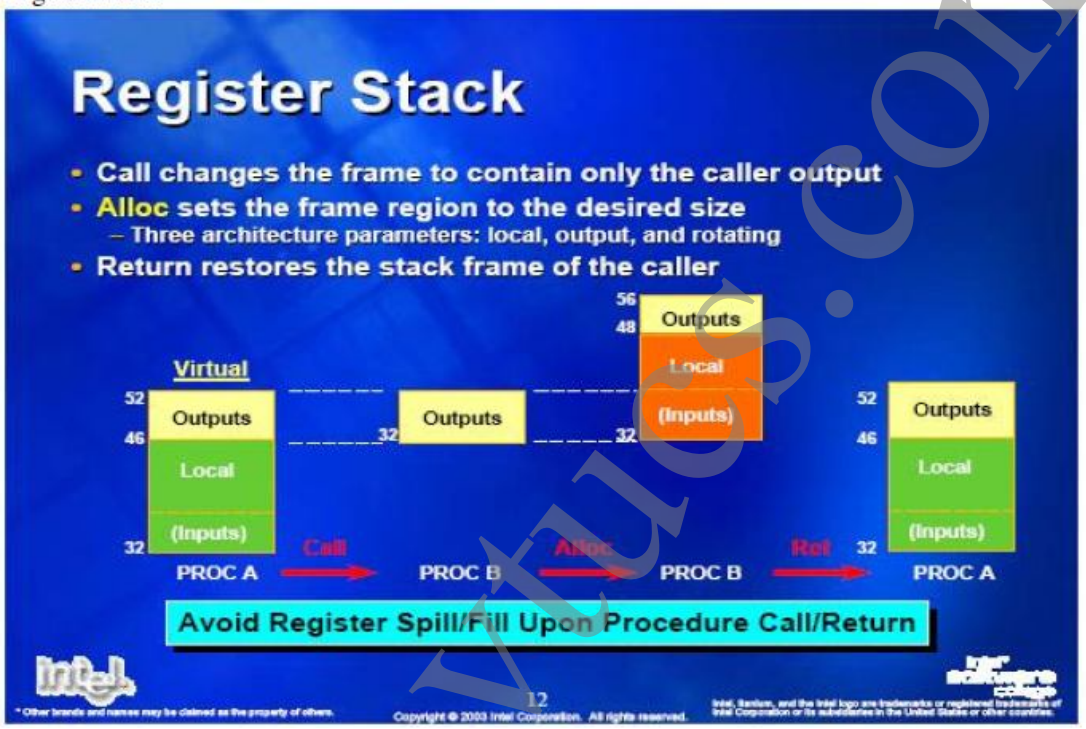
•Initially a 180nm process
–Increased to 130nm in 2003
–Further increased to 90nm in 2007
•Improved Thermal Management
•Clock Speed increased to 1.0Ghz
•Bus Speed Increase from 266Mhz to 400Mhz

•L3 cache moved on die
–Faster access rate

IA-64 Pipeline Features
•Branch Prediction
–Predicate Registers allow branches to be turned on or off
–Compiler can provide branch prediction hints
•Register Rotation
–Allows faster loop execution in parallel
•Predication Controls Pipeline Stages
Cache Features
•L1 Cache
–4 way associative
–*16Kb* Instruction
–16Kb Data
•L2 Cache
–*Itanium*
•6 way associative
•96 Kb
–*Itanium2*
•8 way associative
•256 Kb Initially
–256Kb Data and 1Mb Instruction on Montvale!
Cache Features
•L3 Cache
–*Itanium*
•4 way associative
•Accessible through FSB
•2-4Mb
–*Itanium2*
•2 – 4 way associative
•On Die
•3Mb
–Up to 24Mb on Montvale chips(12Mb/core)!
Register
Specification
_128, 65-bit General Purpose Registers
_128, 82-bit Floating Point Registers
_128, 64-bit Application Registers
_8, 64-bit Branch Registers
_64, 1-bit Predicate Registers
Register Model
- _128 General and Floating Point Registers
- _32 always available, 96 on stack
- _As functions are called, compiler allocates a specific number of local and output

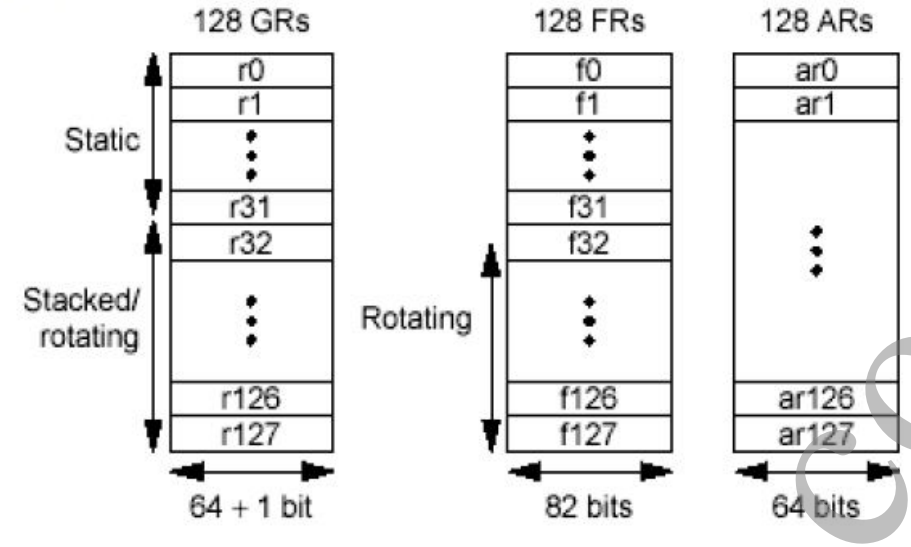
- registers to use in the function by using register allocation instruction "Alloc".
- _Programs renames registers to start from 32 to 127.
- _Register Stack Engine (RSE) automatically saves/restores stack to memory when needed
- _RSE may be designed to utilize unused memory bandwidth to perform register spill and fill operations in the background

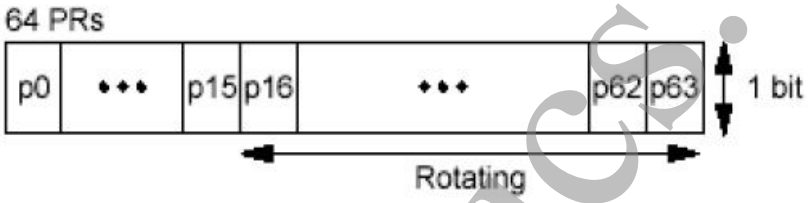
Register Stack

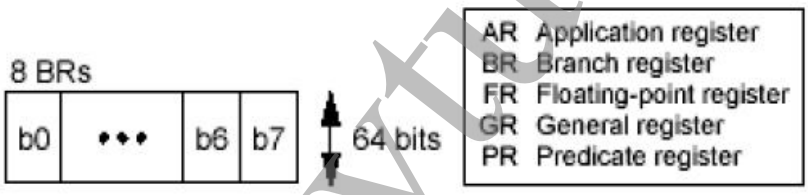


On function call, machine shifts register window such that previous output registers become new locals starting at r32

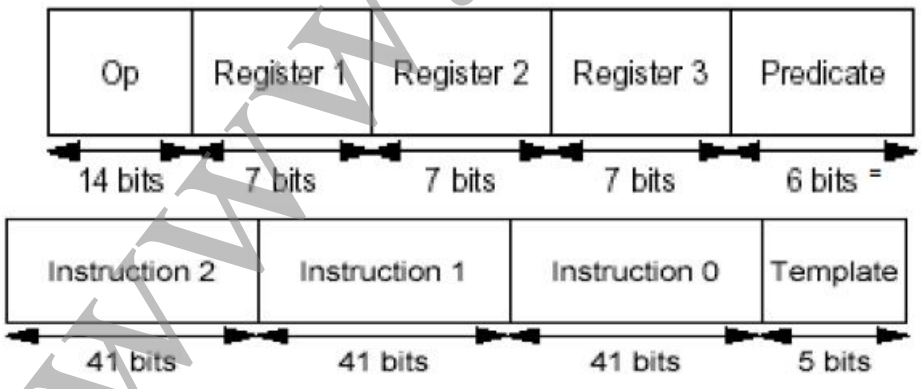
❑Registers







Instruction Encoding



Five execution unit slots

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Possible Template Values

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F

Figure G.7 The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines

Straightforward MIPS code

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
```

```
DADDUI    R1,R1,#-8     ;decrement pointer
                        ;8 bytes (per DW)
BNE       R1,R2,Loop    ;branch R1!=R2
```

The code scheduled to minimize the number of bundles

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
9: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
14: M M F	L.D F10,-16(R1)	L.D F14,-24(R1)	ADD.D F4,F0,F2	3
15: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F8,F6,F2	4
15: M M F	L.D F26,-48(R1)	S.D F4,0(R1)	ADD.D F12,F10,F2	6
15: M M F	S.D F8,-8(R1)	S.D F12,-16(R1)	ADD.D F16,F14,F2	9
15: M M F	S.D F16,-24(R1)		ADD.D F20,F18,F2	12
15: M M F	S.D F20,-32(R1)		ADD.D F24,F22,F2	15
15: M M F	S.D F24,-40(R1)		ADD.D F28,F26,F2	18
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	21

(a) The code scheduled to minimize the number of bundles

The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
8: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
9: M M I	L.D F10,-16(R1)	L.D F14,-24(R1)		2
14: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	3
14: M M F	L.D F26,-48(R1)		ADD.D F8,F6,F2	4
15: M M F			ADD.D F12,F10,F2	5
14: M M F		S.D F4,0(R1)	ADD.D F16,F14,F2	6
14: M M F		S.D F8,-8(R1)	ADD.D F20,F18,F2	7
15: M M F		S.D F12,-16(R1)	ADD.D F24,F22,F2	8
14: M M F		S.D F16,-24(R1)	ADD.D F28,F26,F2	9
9: M M I	S.D F20,-32(R1)	S.D F24,-40(R1)		11
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	12

(b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Unrolled loop after it has been scheduled for the pipeline

```
Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
```

```
ADD.D       F4,F0,F2
ADD.D       F8,F6,F2
ADD.D       F12,F10,F2
ADD.D       F16,F14,F2
S.D     F4,0(R1)
S.D     F8,-8(R1)
DADDUI R1,R1,#-32
S.D     F12,16(R1)
S.D     F16,8(R1)
BNE     R1,R2,Loop
```

Instruction Encoding

- **Each instruction includes the opcode and three operands**
- **Each instructions holds the identifier for a corresponding Predicate Register**
- **Each bundle contains 3 independent instructions**
- **Each instruction is 41 bits wide**
- **Each bundle also holds a 5 bit template field**

Distributing Responsibility

_ILP Instruction Groups

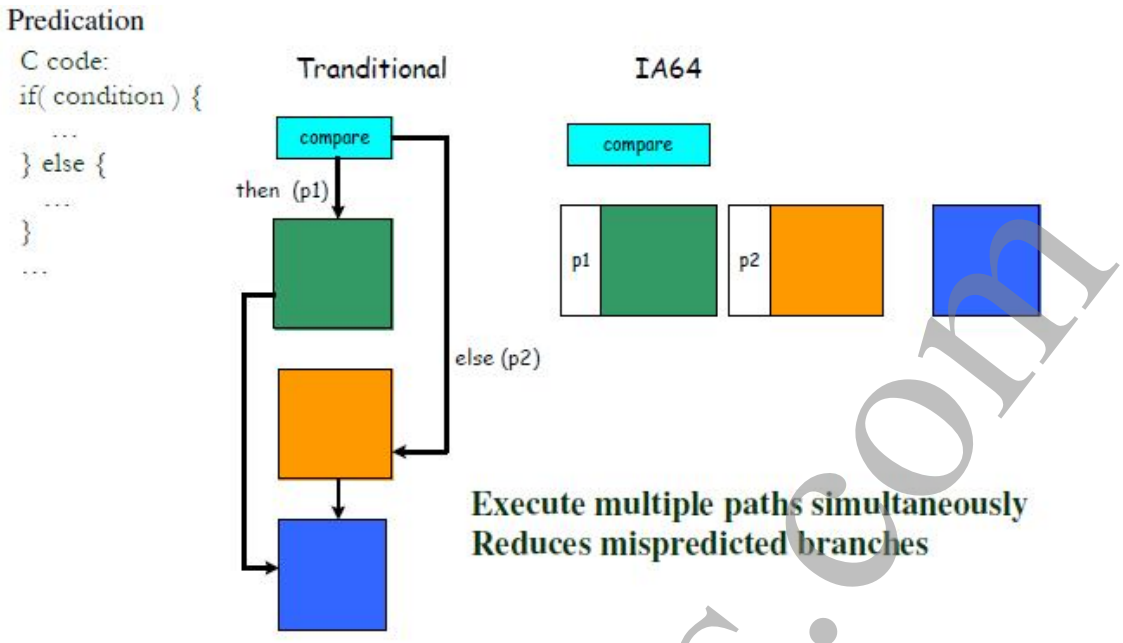
_Control flow parallelism

Parallel comparison

Multiway branches

_Influencing dynamic events

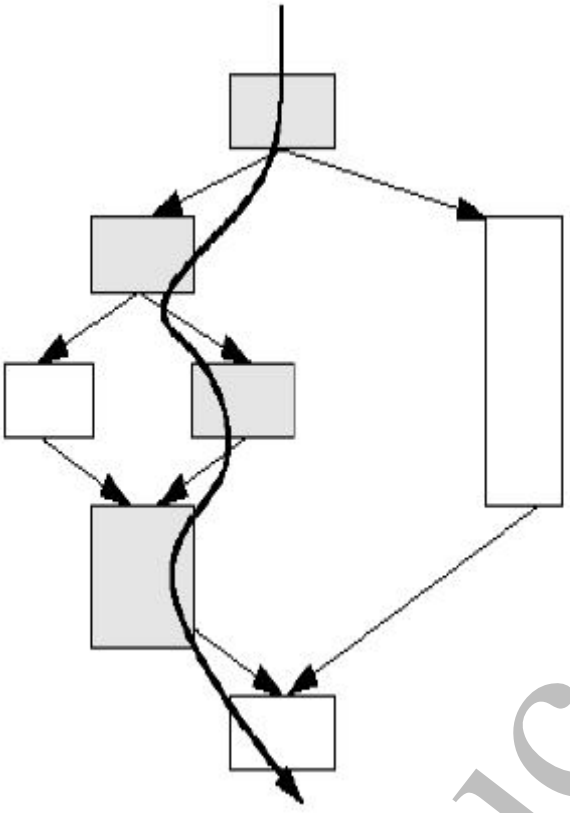
Provides an extensive set of hints that the compiler uses to tell the hardware about likely branch behavior (taken or not taken, amount to fetch at branch target) and memory operations (in what level of the memory hierarchy to cache data).



- _Use predicates to eliminate branches, move instructions across branches
- _Conditional execution of an instruction based on predicate register (64 1-bit predicate registers)
- _Predicates are set by compare instructions
- _Most instructions can be predicated – each instruction code contains predicate field
- _If predicate is true, the instruction updates the computation state; otherwise, it behaves like a nop
-

Scheduling and Speculation

• Basic block: code with single entry and exit, exit point can be multiway branch
• Control Improve ILP by statically move ahead long latency code blocks.
• path is a frequent execution path
• Schedule for control paths
• Because of branches and loops, only small percentage of code is executed regularly
• Analyze dependences in blocks and paths
• Compiler can analyze more efficiently - more time, memory, larger view of the program
• Compiler can locate and optimize the commonly executed blocks



Control speculation

_ Not all the branches can be removed using predication.

_ Loads have longer latency than most instructions and tend to start timecritical chains of instructions

_ Constraints on code motion on loads limit parallelism

_ Non-EPIC architectures constrain motion of load instruction

_ IA-64: Speculative loads, can safely schedule load instruction before one or more prior branches

Control Speculation

_Exceptions are handled by setting NaT (Not a Thing) in target register

_Check instruction-branch to fix-up code if NaT flag set

_Fix-up code: generated by compiler, handles exceptions

_NaT bit propagates in execution (almost all IA-64 instructions)

_NaT propagation reduces required check points

Speculative Load

_ Load instruction (ld.s) can be moved outside of a basic block even if branch target is not known

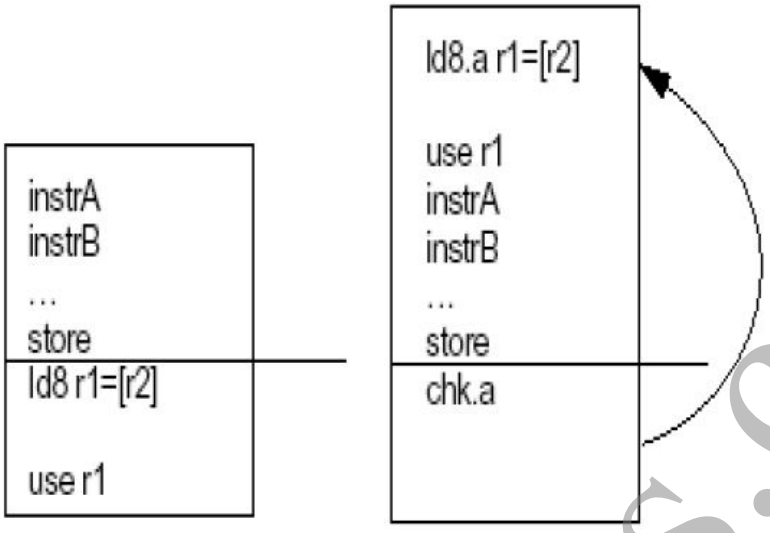
_ Speculative loads does not produce exception - it sets the NaT

_ Check instruction (chk.s) will jump to fix-up code if NaT is set

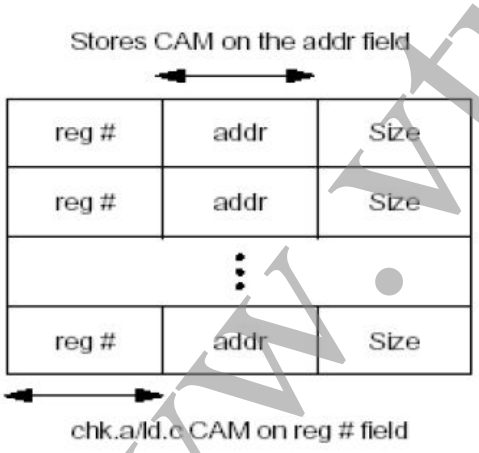
Data Speculation

_ The compiler may not be able to determine the location in memory being referenced (pointers)
_ Want to move calculations ahead of a possible memory dependency
_ Traditionally, given a store followed by a load, if the compiler cannot determine if the addresses will be equal, the load cannot be moved ahead of the store.
_ IA-64: allows compiler to schedule a load before one or more stores
_ Use advance load (ld.a) and check (chk.a) to implement
_ ALAT (Advanced Load Address Table) records target register, memory address accessed, and access size

Data Speculation
1. Allows for loads to be moved ahead of stores even if the compiler is unsure if addresses are the same
2. A speculative load generates an entry in the ALAT
3. A store removes every entry in the ALAT that have the same address
4. Check instruction will branch to fix-up if the given address is not in the ALAT



ALAT

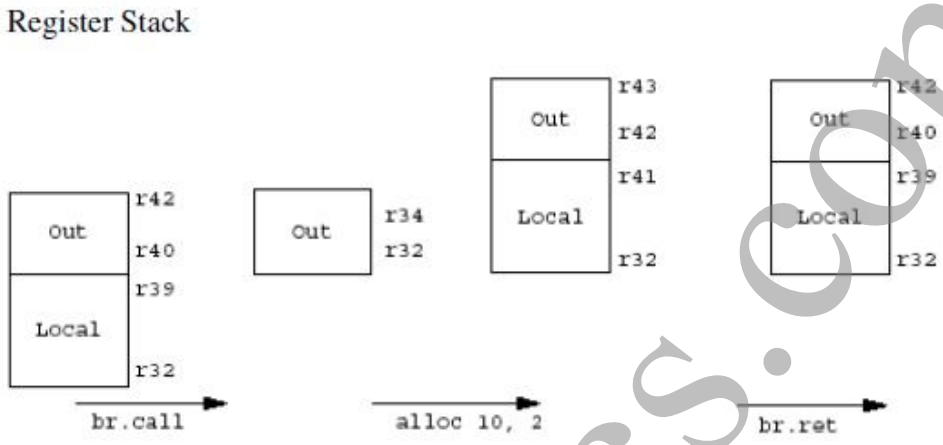


• Use address field as the key for comparison
• If an address cannot be found, run recovery code
• ALAT are smaller and simpler implementation than equivalent structures for superscalars

Register Model

- _128 General and Floating Point Registers
- _32 always available, 96 on stack
- _As functions are called, compiler allocates a specific number of local and output registers to use in the function by using register allocation instruction “Alloc”.

- _Programs renames registers to start from 32 to 127.
- _Register Stack Engine (RSE) automatically saves/restores stack to memory when needed
- _RSE may be designed to utilize unused memory bandwidth to perform register spill and fill operations in the background



On function call, machine shifts register window such that previous output registers become new locals starting at r32

Software Pipelining

_loops generally encompass a large portion of a program's execution time, so it's important to expose as much loop-level parallelism as possible.

_Overlapping one loop iteration with the next can often increase the parallelism.

Software Pipelining

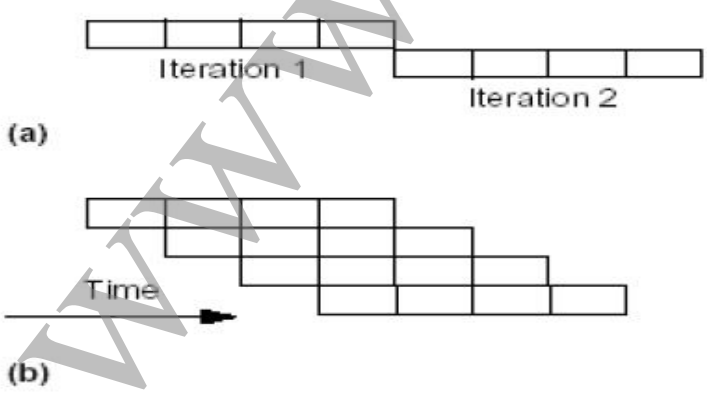


Figure 15. Sequential (a) versus pipelined (b) execution.

We can implement loops in parallel by resolve some problems.

_Managing the loop count,
_Handling the renaming of registers for the pipeline,
_Finishing the work in progress when the loop ends,
_Starting the pipeline when the loop is entered, and
_Unrolling to expose cross-iteration parallelism.

•IA-64 gives hardware support to compilers managing a software pipeline
•Facilities for managing loop count, loop termination, and rotating registers

"The combination of these loop features and predication enables the compiler to generate compact code, which performs the essential work of the loop in a highly parallel form."