EC6009 ADVANCED COMPUTER ARCHITECTURE **L T P C**

3 0 0 3

UNIT I FUNDAMENTALS OF COMPUTER DESIGN 9

Review of Fundamentals of CPU, Memory and IO – Trends in technology, power, energy and cost, Dependability - Performance Evaluation

UNIT II INSTRUCTION LEVEL PARALLELISM 9

ILP concepts – Pipelining overview - Compiler Techniques for Exposing ILP – Dynamic Branch Prediction – Dynamic Scheduling – Multiple instruction Issue – Hardware Based Speculation – Static scheduling - Multi-threading - Limitations of ILP – Case Studies.

UNIT III DATA-LEVEL PARALLELISM 9

Vector architecture – SIMD extensions – Graphics Processing units – Loop level parallelism.

UNIT IV THREAD LEVEL PARALLELISM 9

Symmetric and Distributed Shared Memory Architectures – Performance Issues – Synchronization – Models of Memory Consistency – Case studies: Intel i7 Processor, SMT & CMP Processors.

UNIT V MEMORY AND I/O 9

Cache Performance – Reducing Cache Miss Penalty and Miss Rate – Reducing Hit Time – Main Memory and Performance – Memory Technology. Types of Storage Devices – Buses – RAID – Reliability, Availability and Dependability – I/O Performance Measures.

TOTAL: 45 PERIODS

TEXT BOOK:

1. John L Hennessey and David A Patterson, “Computer Architecture A Quantitative Approach”, Morgan Kaufmann/ Elsevier, Fifth Edition, 2012.

REFERENCES:

1. Kai Hwang and Faye Briggs, “Computer Architecture and Parallel Processing”, Mc Graw-Hill International Edition, 2000.
2. Sima D, Fountain T and Kacsuk P, ”Advanced Computer Architectures: A Design Space Approach”, Addison Wesley, 2000.

EC 6009 ADVANCED COMPUTER ARCHITECTURE 3 0 0 3

1. **Aim and Objective of the subject**
    - Understand the micro-architectural design of processors
    - Learn about the various techniques used to obtain performance improvement and power savings in current processors
    - To familiarize the students with Instruction Level Parallelism and Data-Level Parallelism
    - To expose the students to the concept of Thread Level Parallelism
    - To familiarize the students with Memory and I/O
2. **Need and Importance for the study of the subject**
    - Evaluate performance of different processor architectures with respect to various parameters
    - Analyze performance of different Instruction Level Parallelism techniques
    - Evaluate performance of Data-Level Parallelism andThread Level Parallelism
    - Identify cache and memory related issues in multi-processors
3. **Industry Connectivity and Latest Developments**
    - Latest processor's architecture in computers (SMT, CMP, Intel i7 Processor's) are analyzed.
    - VLIW and Vector architectures play a vital role in industries
4. **Industry Visit (Planned if any)** --NIL---

TEXT BOOK:

1. John L Hennessey and David A Patterson, “Computer Architecture A Quantitative Approach”, Morgan Kaufmann/ Elsevier, Fifth Edition, 2012.

REFERENCES:

1. Kai Hwang and Faye Briggs, “Computer Architecture and Parallel Processing”, Mc Graw-Hill International Edition, 2000.
2. Sima D, Fountain T and Kacsuk P, ”Advanced Computer Architectures: A Design Space Approach”, Addison Wesley, 2000.

Si. No	Unit	Topics to be covered	Hours required / planned	Cumulative Hrs	Books Referred	Page No.
UNITI FUNDAMENTALS OF COMPUTER DESIGN						
1	I	Review of Fundamentals of CPU, Memory, IO	3	3	T1	2-17
2	I	Trends in technology Instructions	1	4	T1	17-21
3	I	Power, Energy	2	6	T1	21-26
4	I	Cost	1	7	T1	27-33
5	I	Dependability	1	8	T1	33-36
6	I	Performance Evaluation	1	9	T1	36-44
UNITII INSTRUCTION LEVEL PARALLELISM						
7	II	ILP concepts, Pipelining overview	1	10	T1	148-156
8	II	Compiler Techniques for Exposing ILP	2	12	T1	156-162
9	II	Dynamic Branch Prediction	1	13	T1	162-167
10	II	Dynamic Scheduling	1	14	T1	167-183
11	II	Hardware Based Speculation	1	15	T1	183-192
12	II	Multiple instruction Issue - Static scheduling	1	16	T1	192-202
13	II	Multi-threading	1	17	T1	223-232
14	II	Limitations of ILP	1	18	T1	213-221
15	II	Case Studies	1	19	T1	247-254
UNIT III DATA-LEVEL PARAL1LELISM						
16	III	Vector architecture	2	21	T1	264-282
17	III	SIMD extensions	3	24	T1	282-288
18	III	Graphics Processing units	2	26	T1	288-315
19	III	Loop level parallelism	2	28	T1	315-322
UNITIV THREAD LEVEL PARALLELISM						
20	IV	Symmetric Shared Memory Architectures	2	30	T1	366-378
21	IV	Distributed Shared Memory Architectures	2	32	T1	378-386
22	IV	Performance Issues	1	33	T1	395-400
23	IV	Synchronization	1	34	T1	386-391
24	IV	Models of Memory Consistency	1	35	T1	392-395
25	IV	Case studies: Intel i7 Processor, SMT Processor, CMP Processor	2	37	T1	401-405
UNITV MEMORY ANDI/O						
26	V	Cache Performance- Reducing Cache Miss Penalty, Miss Rate, Reducing Hit Time	3	40	T1	78-96
27	V	Main Memory and Performance	1	41	T1	72-78
28	V	Memory Technology	1	42	T1	96-105
29	V	Types of Storage Devices	1	43	T1	D12-35
30	V	Buses	1	44	T1	I16
31	V	RAID – Reliability, Availability and Dependability	1	45	T1	D44-59
32	V	I/O Performance Measures	1	46	T1	D15-16

Total: 46 Hours

INDEX

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

EC6009 ADVANCED COMPUTER ARCHITECTURE

QUESTION BANK

UNIT I - FUNDAMENTALS OF COMPUTER DESIGN

PART A

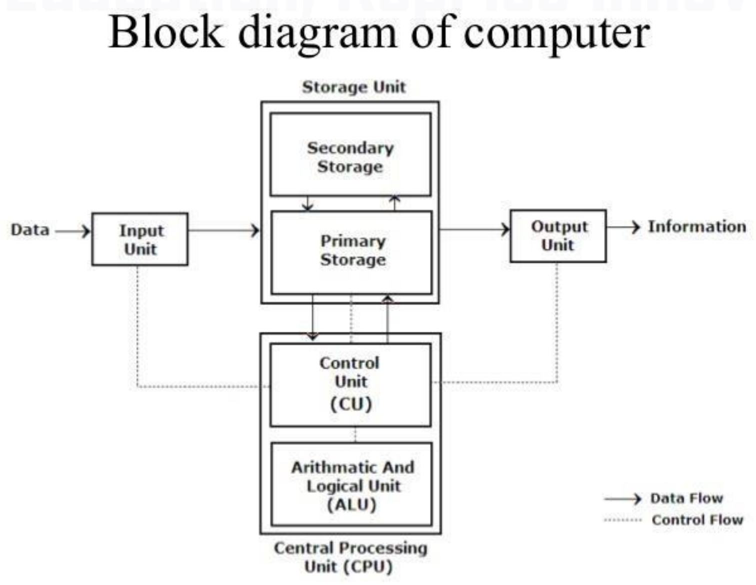
1. What are the five trends in computer Technology? (NOV/DEC 2016)

- **Multi processors**
- **VLSI Technology Trends**
- **Advanced machine learning**
- **Internet of Things (IOT)**
- **Cloud computing**

2. What are the components of a computer? [May/June 2014]

- Input unit
- Memory unit
- Arithmetic and Logic Unit
- Output unit
- Control unit

3. Draw the block diagram of computer. [R-2013]



Block Diagram of Computer

1. What is Execution time/Response time??[May/June 2015]

Response time also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

2. How to find the cost of an Integrated circuit ? (NOV/DEC 2016)

The cost of an integrated circuit varies between computers. In case of personal mobile devices increasing the performance of the whole system on chip (SOC), the cost of IC is greater than cost of PMD.

Cost of a packaged IC = (cost of die + cost of testing die + cost of packing and final test) / final test field.

4. What is CPU execution time, user CPU time and system CPU time and Write the expression for CPU time? ?[May/June 2014]

CPU time: The actual time the CPU spends computing for a specific task.

User CPU time: The CPU time spent in a program itself.

System CPU time: The CPU time spent in the operating system performing tasks on behalf the program.

CPU time= CPU clock cycles for a program x Clock cycle time

or

CPU time= CPU clock cycles for a program / Clock rate

5. What is clock cycle and clock period? [R-2013]

C**lock cycle**: The time for one clock period, usually of the processor clock, this runs at a constant rate.

Clock period: The length of each clock cycle.

6. Define CPI and Write the expression for CPI time. [R-2013]

The term Clock Cycles Per Instruction Which is the average number of clock cycles each 3 instruction takes to execute, is often abbreviated as CPI.

CPI time= CPU clock cycles for a program / Instruction count

7. State and explain the performance equation?[R-2013]

Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock cycle rate is R cycles per second, the processor time is given by T = (N x S) / R. This is often referred to as the basic performance equation. Where N denotes number of machine Instructions.

8. Define MIPS and MIPS Rate.?[Nov/Dec 2014]

MIPS: Million Instructions Per Second (MIPS) is a measurement of program execution speed based on the number of millions of instructions. MIPS is computed

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

as:

This MIPS measurement is also called Native MIPS to distinguish it from some alternative definitions of MIPS.

MIPS Rate. The rate at which the instructions are executed at a given time.

9. Define Throughput and Throughput rate. [R-2013]

Throughput -The total amount of work done in a given time.

Throughput rate-The rate at which the total amount of work done at a given time.

10. Define Amdahl’s law. [R-2013]

It States that performance improvement to be gained by using faster mode of execution limited by the fraction of time the faster mode can be used. Amdahl’s law defines the term speedup which is given by,

$$Speedup = \frac{Performanceofentiretaskusingtheenhancementwhenpossible}{Performanceforentiretaskwithoutusingtheenhancement}$$

$$speedup = \frac{Executiontimeforentiretaskwithoutusingtheenhancement}{Executiontimeofentiretaskusingtheenhancementwhenpossible}$$

Amdahl’s law states that in parallelization, if P is the proportion of a system or program that can be made parallel, and 1-P is the proportion that remains serial, then the maximum speed up that can be achieved using N number of processors is 1/((1-P)+(P/N).

11. Calculate speedup $_{\text{overall}}$ using Amdahl's law given that a new CPU, which is 10 times faster than the original CPU on computation, is additionally introduced. The original CPU is busy with computations 40% of time and I/O wait time is 60%.[R-2013]

Given: Fraction $_{\text{enhanced}}$=0.4, Speedup $_{\text{enhanced}}$=10

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

$$\text{speedup}_{\text{overall}} = \frac{1}{(0.6)+\frac{0.4}{10}} = 1.5625$$

12. What do you infer from the term dependability?(Apr/may 2017)

Computers not only need to be fast, they need to be dependable. Since any physical device can fail, the system must dependable by including redundant components that can take over when a failure occurs and to help to detect failures.

13. Find the number of dies per 30 cm wafer for a die that is 0.7 cm on a side. [R-2013]

Solution:

Total die area is 0.49cm^2

$$\boldsymbol{Diesperwafer = \frac{\pi x(\frac{30}{2})^2}{0.49} - \frac{\pi x 30}{\sqrt{2x0.49}} = 1347}$$

14. Find the yield for dies that are I cm on a side and 0.7 cm on a side, assuming a defect density of 0.6 cm2. [R-2013]

Solution:

Total die area is 0.49cm^2

For larger die

$\boldsymbol{Dieyield = 1 + (\frac{0.6x1}{4})^{-4}}$**=0.57**

For smaller die

$\boldsymbol{Dieyield = 1 + (\frac{0.6x0.49}{4})^{-4}}$**=0.75**

15. What are the two main measures of dependability? [R-2013]

Module reliability: It is a measure of the continuous service accomplishment from a reference initial instant.

Module Availability: It is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption.

PART B

1. Explain the Components of a computer system with the block diagram in detail.

[R-2013]

A computer consists of five functionally independent main parts. They are

1. Input
2. Memory
3. Arithmetic and logic
4. Output
5. Control unit

Basic functional units of a computer

The computer accepts programs and the data through an input and stores them in the memory. The stored data are processed by the arithmetic and logic unit under program control. The processed data is delivered through the output unit. All above activities are directed by control unit.

a. Input unit

The computer accepts coded information through input unit. The input can be from human operators, electromechanical devices such as keyboards or from other computer over communication lines.

Examples of input devices are Keyboard, joysticks, trackballs and mouse are used as graphic input devices in conjunction with display

Keyboard

- It is a common input device.
- Whenever a key is pressed; the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over cable to the memory of the computer.

Memory unit

Memory unit is used to store programs as well as data. Memory is classified into primary and secondary storage.

Primary storage

It also called main memory. It operates at high speed and it is expensive. It is made up of large number of semiconductor storage cells, each capable of storing one bit of information. These cells are grouped together in a fixed size called word.

This facilitates reading and writing the content of one word (n bits) in single basic operation instead of reading and writing one bit for each operation.

Secondary storage

It is slow in speed. It is cheaper than primary memory. Its capacity is high. It is used to store information that is not accessed frequently. Various secondary devices are magnetic tapes and disks, optical disks (CD-ROMs), floppy etc.

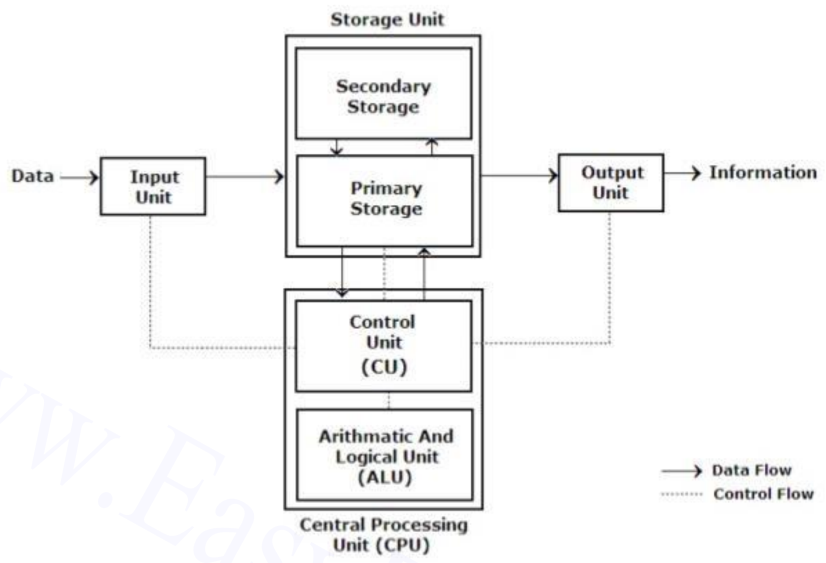


Figure: The basic components of computer system

b. Arithmetic and logic unit

Arithmetic and logic unit (ALU) and control unit together form a processor. Actual execution of most computer operations takes place in arithmetic and logic unit of the processor. Example: Suppose two numbers located in the memory are to be added. They are brought into the processor, and the actual addition is carried out by the ALU.

Registers:

Registers are high speed storage elements available in the processor. Each register can store one word of data. When operands are brought into the processor for any operation, they are stored in the registers. Accessing data from register is faster than that of the memory.

c. Output unit

The function of output unit is to produce processed result to the outside world in human understandable form. Examples of output devices are Graphical display, Printers such as inkjet, laser, dot matrix and so on. The laser printer works faster.

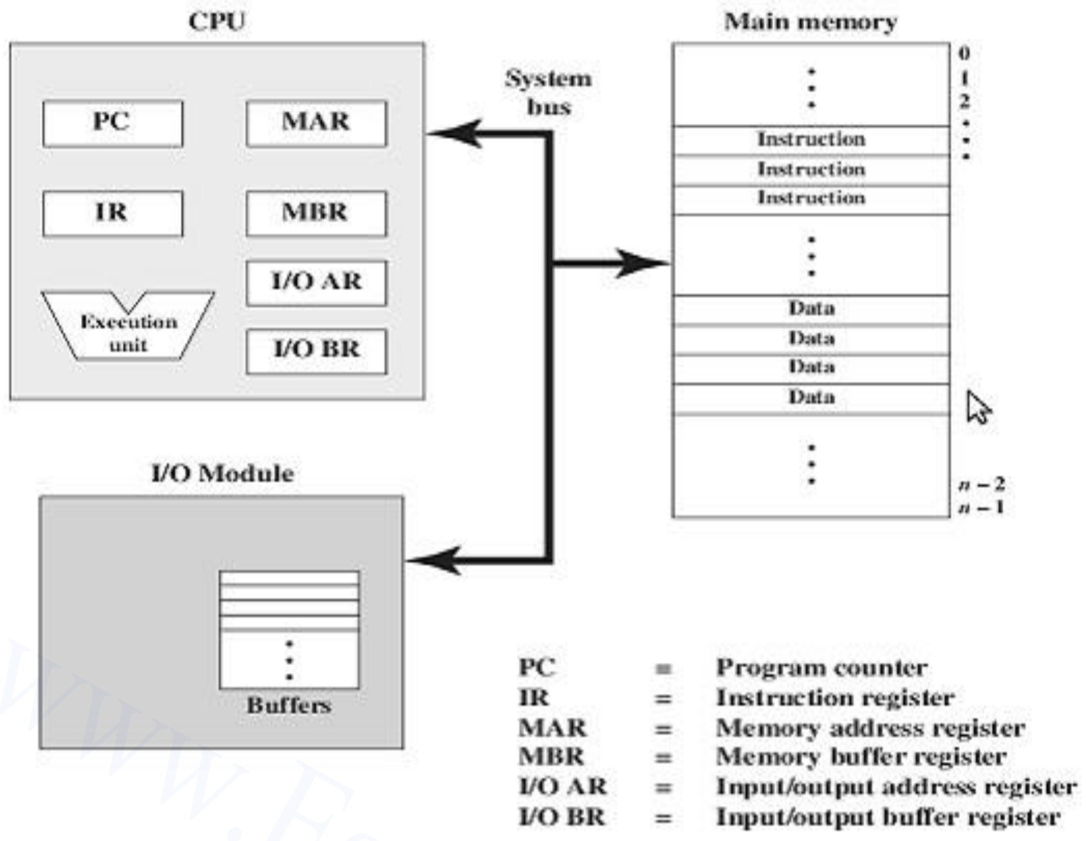


Figure: Computer Components

d. Control unit

Control unit coordinates the operation of memory, arithmetic and logic unit, input unit, and output unit in some proper way. The **control unit** issues **control signals** that cause the CPU (and other components of the computer) to fetch the instruction to the IR (Instruction Register) and then execute the actions dictated by the machine language instruction that has been stored there. Control units are well defined, physically separate unit that interact with other parts of the machine.

A set of control lines carries the signals used for timing and synchronization of events in all units Example: Data transfers between the processor and the memory are controlled by the control unit through timing signals**. Timing signals** are the signals that determine when a given action is to take place.

Basic Operational Concept

Computer Components:

Top-Level view

PC the **program counter** contains the address of the assembly language instruction to be executed next. IR the **instruction register** contains the binary word corresponding to the machine language version of the instruction currently being executed.

MAR the **memory address register** contains the address of the word in main memory that is being accessed. The word being addressed contains either data or a machine language instruction to be executed.

MBR the **memory buffer register** (also called MDR for memory data register) is the register used to communicate data to and from the memory.

The operation of a processor is characterized by a *fetch-decode-execute* cycle. In the first phase of the cycle, the processor fetches an instruction from memory. The address of the instruction to fetch is stored in an internal register named the *program counter*, or PC. As the processor is waiting for the memory to respond
with the instruction, it increments the PC. This means the fetch phase of the next cycle will fetch the instruction in the next sequential location in memory. In the decode phase the processor stores the information returned by the memory in another internal register, known as the instruction register, or IR. The IR now holds a single machine instruction, encoded as a binary number. The processor decodes the value in the IR in order to figure out which operations to perform in the next stage.In the execution stage the processor actually carries out the instruction. This step often requires further memory operations; for example, the instruction may direct the processor to fetch two operands from memory, add them, and store the result in a third location (the addresses of the operands and the result are also encoded as part of the instruction). At the end of this phase the machine starts the cycle
over again by entering the fetch phase for the next instruction. The CPU exchanges data with memory. For this purpose, it typically makes use of **two internal (to the CPU) register:**

- A memory address register (MAR), which specifies the address in memory for the next read or write, and
- A memory buffer register (MBR), which contains the data to be written into memory or receives the data read from memory.

An I/O addresses register (I/OAR) specifies a particular I/O device. An I/O buffer (I/OBR) register is used for the exchange of data between an I/O module and the CPU.A memory module consists of a set of locations, defined by sequentially numbered address. Each location contains a binary number that can be

interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa.
It contains internal buffers for temporarily holding these data until they can be sent on. Instructions can be classified as one of three major types: arithmetic/logic, data transfer, and control. Arithmetic and logic instructions apply primitive functions of one or two arguments, for example addition, multiplication, or logical AND.

2. State the CPU performance equation and discuss the factors that affect the performance of a computer. [R-2013]?[May/June 2014]

Response time: The time between the start and the completion of an event also referred to as *execution time*.

Throughput: The total amount of work done in a given time. In comparing design alternatives, we often want to relate the performance of two different machines, say X and Y. The machine X is faster than Y is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, if X is *n* times faster than Y

$$\frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = n$$

Since execution time is the reciprocal of performance, the following relationship

$$n = \frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = \frac{\dfrac{1}{\text{Performance}_{\text{Y}}}}{\dfrac{1}{\text{Performance}_{\text{X}}}} = \frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}}$$

The performance and execution time are reciprocals, increasing performance decreases execution time. To help avoid confusion between the terms Increasing and decreasing, we usually say improve performance or improve execution time when we mean increase performance and decrease execution time.

CPU performance equation:

All computers are constructed using a clock running at a constant rate. These discrete time events are called ticks, clock ticks, clock periods, clocks, cycles, or clock cycles. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

CPU time = CPU clock cycles for a program x Clock cycle time

CPU time=CPU clock cycle to exe a pgm / clock rate

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed, the instruction path length or instruction count (IC).

CPI is computed as:

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction Count}}$$

By transposing instruction count in the above formula, clock cycles can be defined As IC * CPI. This allows us to use CPI in the execution time formula

$$\text{CPU time} = \text{Instruction Count} \times \text{Clock cycle time} \times \text{Cycles per Instruction}$$

$$\text{CPU time} = \frac{\text{Instruction Count} \times \text{Clock cycle time}}{\text{Clock rate}}$$

CPU to calculate the number of total CPU clock cycles as**,**

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

Basic performance equation

Let T be the time required for the processor to execute a program in high level language. The compiler generates machine language object program corresponding to the source program. Assume that complete execution of the program requires the execution of N machine language instructions.

Assume that average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock rate is R cycles per second, the program execution time is given by T = (N x S) / R this is often called Basic performance equation.

To achieve high performance, the performance parameter T should be reduced. T value can be reduced by reducing N and S, and increasing R.

- Value of N is reduced if the source program is compiled into fewer number of machine instructions.
- Value of S is reduced if instruction has a smaller no of basic steps to perform or if the execution of the instructions is overlapped.

- Value of R can be increased by using high frequency clock, ie. Time required to complete a basic execution step is reduced.
- N, S and R are dependent factors. Changing one may affect another.

Choosing Programs to Evaluate Performance

There are five levels of programs used in such circumstances, listed below in decreasing order of accuracy of prediction.

1. Real applications-Real applications have input, output, and options that a user can select when running the program. There is one major downside to using real applications as benchmarks: Real applications often encounter portability problems arising from dependences on the operating system or compiler. Enhancing portability often means modifying the source and sometimes eliminating some important activity, such as interactive graphics, which tends to be more system-dependent.**Modified (or scripted) applications-**In many cases, real applications are used as the building block for a benchmark either with modifications to the application or with a script that acts as stimulus to the application. Applications are modified for two primary reasons: to enhance portability or to focus on one particular aspect of system performance. For example, to create a CPU-oriented benchmark, I/O may be removed or restructured to minimize its impact on execution time. Scripts are used to reproduce interactive behavior, which might occur on a desktop system, or to simulate complex multiuser interaction, which occurs in a server system.

2. Kernels-Several attempts have been made to extract small, key pieces from real programs and use them to evaluate performance. Kernels are best used to isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs.

4. Toy benchmarks-Toy benchmarks are typically between 10 and 100 lines of code and produce a result the user already knows before running the toy program. Programs like Sieve of Eratosthenes, Puzzle, and Quicksort are popular because they are small, easy to type, and run on almost any computer. The best use of such programs is beginning programming assignments.

5. Synthetic benchmarks- Synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs. Whetstone and Dhrystone are the most popular synthetic benchmarks.

3. (i)Explain the need to switch from uniprocessors to multiprocessors and draw the performance chart for processors over years. [R-2013]?[May/June 2013]

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

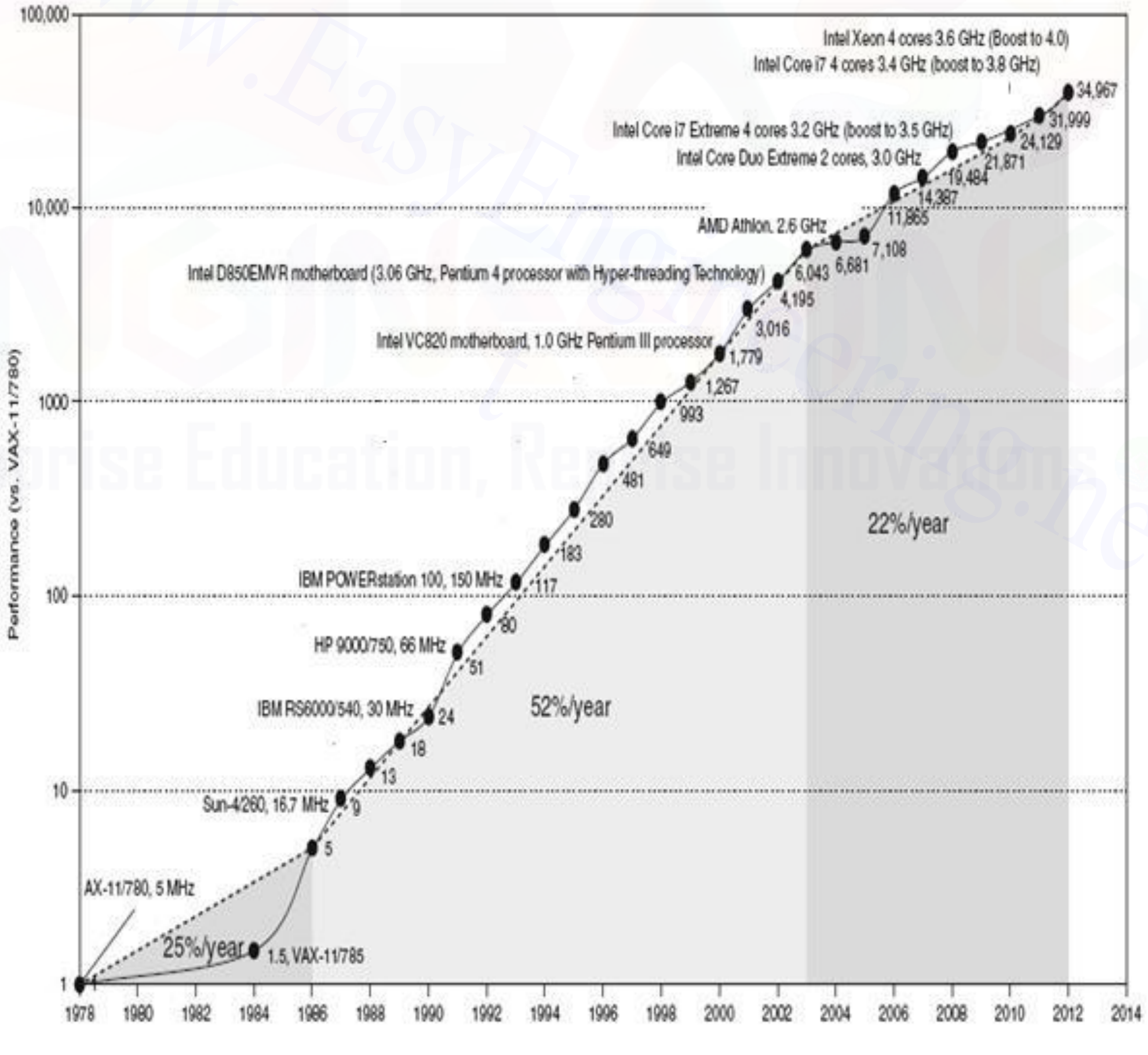


Figure Growth in processor performance since the mid-1980s

(ii) Explain how clock rate and power are related to each other in microprocessor over years with a neat graph (or) Elaborate about power wall with neat sketch. [R-2013]

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of energy consumption is so-called dynamic energy— that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied.

$$Energy \propto Capacitive\ load \times Voltage^2$$

This equation is the energy of a pulse during the logic transition of $0 \to 1 \to 0$ or $1 \to 0 \to 1$. The energy of a single transition is then

$$Energy \propto 1/2 \times Capacitive\ load \times Voltage^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

$$Power \propto 1/2 \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the fanout) and the technology, which determines the capacitance of both wires and transistors.

(iii) Explain the fundamentals of computer Design. [R-2013]

Computer technology has made incredible progress in the roughly from last 55 years. This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer design.

During the first 25 years of electronic computers, both forces made a major contribution; but beginning in about 1970, computer designers became largely dependent upon integrated circuit technology. During the 1970s, performance continued to improve at about 25% to 30% per year for the mainframes and minicomputers that dominated the industry.

The late 1970s after invention of microprocessor the growth roughly increased 35% per year in performance. This growth rate, combined with the cost advantages of a mass-produced microprocessor, led to an increasing fraction of the computer business. In addition, two significant changes are observed in computer industry.

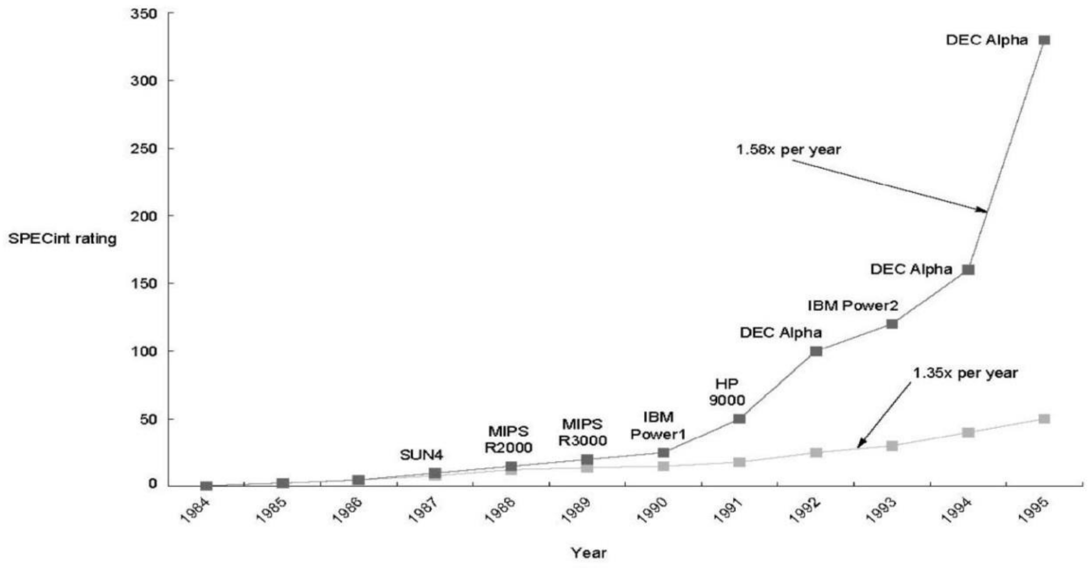


Fig. Growth in microprocessor Performance since the mid 1980s has been substantially higher than in earlier years as shown by plotting SPECint performance.

- First, the virtual elimination of assembly language programming reduced the need for object-code compatibility.
- Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to successfully develop a new set of architectures, called RISC (Reduced Instruction Set Computer) architectures. In the early 1980s. The RISC-based machines focused the attention of designers on
two critical performance techniques, the exploitation of instruction-level parallelism and the use of caches.

The combination of architectural and organizational enhancements has led to 20 years of sustained growth in performance at an annual rate of over 50%. Figure shows the effect of this difference in performance growth rates.

The effect of this dramatic growth rate has been twofold.

- First, it has significantly enhanced the capability available to computer users. For many applications, the highest performance microprocessors of today outperform the supercomputer of less than 10 years ago.
- Second, this dramatic rate of improvement has led to the dominance of microprocessor-based computers across the entire range of the computer design.

4 (i) Explain the trends in technologies for building computer over time. (or) Explain various Technology trends in computer industry. [R-2013]

The designer must be especially aware of rapidly occurring changes in implementation technology. The following Four implementation technologies changed the computer industry.

1. Integrated circuit logic technology: Transistor density increases by about 35% per year, and die size increases 10% to 20% per year. The combined effect is a growth rate in transistor count on a chip of about 55% per year.

2. Semiconductor DRAM: Density increases by between 40% and 60% per year and Cycle time has improved very slowly, decreasing by about one-third in 10 years. Bandwidth per chip increases about twice as fast as latency decreases. In addition, changes to the DRAM interface have also improved the bandwidth.

3. Magnetic disk technology: it is improving more than 100% per year. Prior to 1990, density increased by about 30% per year, doubling in three years. It appears that disk technology will continue the faster density growth rate for some time to come. Access time has improved one-third in 10 years.

4. Network Technology: Network Performance depends both on the performance of switches and on the performance of the transmission system, both latency and bandwidth can be improved, though recently bandwidth has been the primary focus.

Scaling of Transistor Performance, Wires, and Power in Integrated Circuits

Transistor Performance:

Integrated circuit processes are characterized by the feature size, which is decreased from 10 microns in 1971 to 0.18 microns in 2001. Since a transistor is a 2-dimensional object, the density of transistors increases quadratically with a linear decrease in feature size. The increase in transistor performance, this combination of scaling factors leads to a complex interrelationship between transistor performance and process feature size.

Wires

The signal delay for a wire increases in proportion to the product of its resistance and capacitance. As feature size shrinks wires get shorter, but the resistance and capacitance per unit length gets worse.

Since both resistance and capacitance depend on detailed aspects of the process, the geometry of a wire, the loading on a wire, and even the adjacency to other structures.

Power

Power also provides challenges as devices are scaled. For modern CMOS microprocessors, the dominant energy consumption is in switching transistors. The energy required per transistor is proportional to the product of the load capacitance of the transistor, the frequency of switching, and the square of the voltage.

Moore's Law

Gordon Moore (Founder of Intel) observed in 1965 that the number of transistors that could be crammed on a chip doubles every year.

Based on SPEED, the CPU has increased dramatically, but memory and disk have increased only a little. This has led to dramatic changed in architecture, Operating Systems, and Programming practices.

Capacity	**Speed (latency)**
Logic 2x in 3 years	2x in 3 years
DRAM 4x in 3 years	2x in 10 years
Disk 4x in 3 years	2x in 10 years

Electronics technology continues to evolve Increased capacity and performance Reduced cost

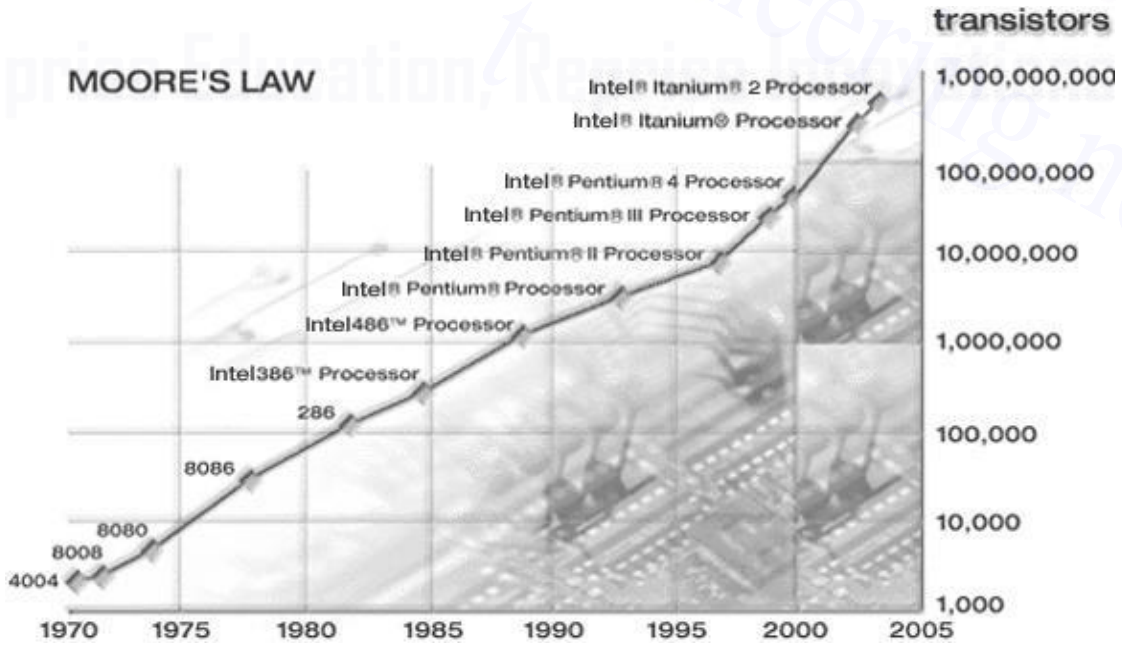


Figure: Moore's law

(ii) Explain the trends in Cost, Price for building computer over time. (or) Explain the impact of Time, volume and commodification on Cost and Price. [R-2013]

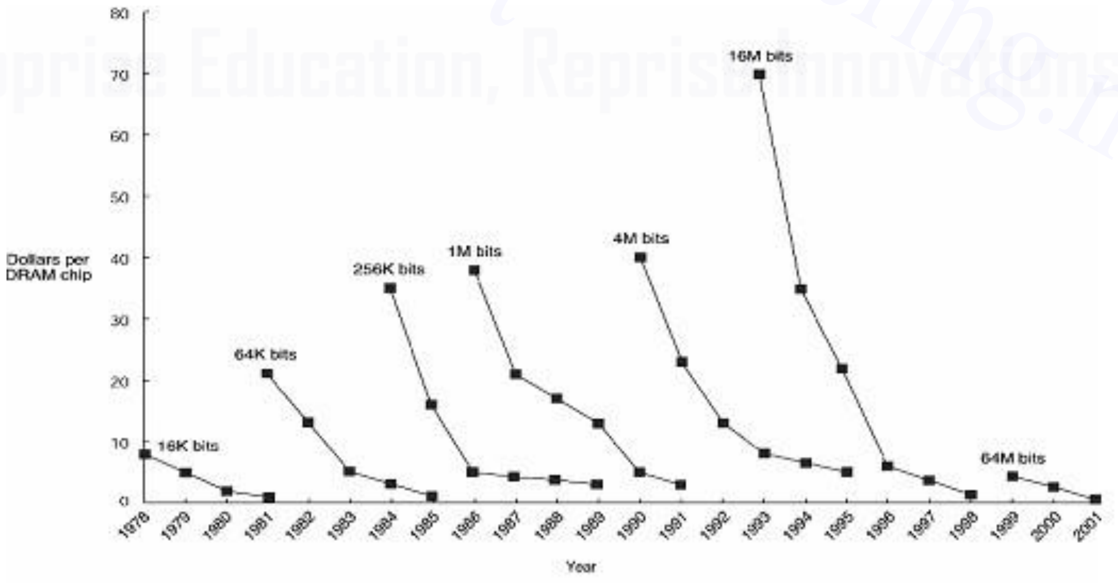
- Price: selling a finished good
  - 1999: more than half the PCs sold were priced at less than $1000
- Cost: the amount spent to produce it, including overhead

The Impact of Time, Volume, Commodification:

- Time
  - The cost of a manufactured computer component decreases over time even without major improvements in the basic implementation technology.
  - The underlying principle that drives costs down is the learning curve manufacturing costs decrease over time.
  - Yield: the percentage of the manufactured devices that survive the testing procedure.

    As the technology matures over time, the yield improves and hence, things get cheaper.
  - Prices of six generations of DRAMs

    The following Figure plots the price of a new DRAM chip over its lifetime.



Between the start of a project and the shipping of a produce, say, two years, the cost of a new DRAM drops by a factor of between 5 and 10 in constant dollars.

- Price of an Intel Pentium III at a Given Frequency

It decreases over time as yield enhancements decrease the cost of a good die and competition forces price reductions.

- Wafer and Dies

Exponential cost decrease since silicon manufacture technology basically the same:
A wafer is tested and chopped into dies that are packaged

- Volume
  - Volume is the second key factor in determining cost. Increasing volume affecting the cost in several ways.
  - Increasing volume decreases cost (time for learning curve),
  - Increases purchasing and manufacturing efficiency
- Commodities
  - Commodities are Products are sold by multiple vendors in large volumes are essentially identical
  - Virtually all the products sold on the shelves of grocery stores are commodities, as are standard DRAMs, disks, monitors, and keyboards.
  - In the past 10 years, much of the low end of the computer business has become a commodity business focused on building IBM-compatible PCs.
  - Improved competition leads to reduced cost
- Cost of an Integrated Circuit (IC)

The cost of packaged integrated circuit is

$$\text{Cost of IC} = \frac{\boldsymbol{costofdie + Costoftestingdie + Costofpackagingandfinaltest}}{\boldsymbol{Finaltestyield}}$$

A greater portion of the cost that varies between machines.

$$\boldsymbol{costofdie} = \frac{\boldsymbol{costofwafer}}{\#\,\boldsymbol{DiesperwaferxDieyield}}$$

$$\#\,\boldsymbol{Diesperwafer} = \frac{\boldsymbol{\pi xWaferradius^2}}{\boldsymbol{DieArea}} - \frac{\boldsymbol{\pi xWaferdiameter}}{\boldsymbol{\sqrt{2}xDieArea}}$$

Number of Dies per wafer is sensitive to die size.

$$Dieyield = \text{Wafer yield}x\ (1 + (\frac{DefectsperunitareaxDieArea}{\alpha})^{-\alpha}$$

In today's technology, α=4, $Defects\ per\ unit\ area = 0.4\ and\ 0.8\ per\ cm^2$

5.(i) Suppose we have made the following measurements:

Frequency of FP operations (other than FPSQR) = 25% Average CPI of FP operations = 4.0 Average CPI of other instructions = 1.33 Frequency of FPSQR= 2% CPI of FPSQR = 20 Assume that the two design alternatives are to decrease the CPI of FPSQR to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the CPU performance equation. [R-2013]

$$\begin{aligned}\text{CPI}_{\text{original}} &= \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Instruction count}}\right)\\ &= (4 \times 25\%) + (1.33 \times 75\%) = 2.0\end{aligned}$$

$$\begin{aligned}\text{CPI}_{\text{with new FPSQR}} &= \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}})\\ &= 2.0 - 2\% \times (20 - 2) = 1.64\end{aligned}$$

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625$$

$$\begin{aligned}\text{Speedup}_{\text{new FP}} &= \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}}\\ &= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23\end{aligned}$$

(ii)A program runs in 20 seconds on machine A with a clock speed of 200 MHz. A computer architect wants to build machine B, which will run this program in 6 seconds. The architect has determined that a substantial increase in the clock rate possible, but this will affect the design of the rest of the CPU, causing machine B to require 1.2 times as many clock cycle as machine A for this program. What clock rate should he target for a best design? [R-2013]

Solution:

$$\text{CPU time for A} = \frac{\text{CPU clock cycles}_A}{\text{clock rate}_A}$$

$$20 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{200 \times 10^6 \frac{\text{cycles}}{\textit{seconds}}}$$

$$\text{CPU clock cycle}_A = 20 \times 200 \times 10^6 \text{ cycles}$$

CPU time for B may be found as shown below

$$\text{CPU time for B} = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 4000 \times 10^6 \text{ cycles}}{\text{clock rate}_B}$$

$$\text{clock rate}_B = \frac{1.2 \times 4000 \times 10^6 \text{ cycles}}{6 \text{ seconds}} = 800 \times 10^6 \frac{\text{cycles}}{\text{seconds}} = 800\text{MHz}$$

Therefore ,Machine B must have four times the clock rate of A to run the program in 6 seconds

UNIT II
INSTRUCTION LEVEL PARALLELISM
PART A

1. What is ILP? [Nov/Dec 2012][May/June 2012]

ILP = Instruction level parallelism

• Multiple operations (or instructions) can be executed in parallel

2. What are the needs of ILP? [R-2013]

• Sufficient resources

• Parallel scheduling

- ➢ Hardware solution
- ➢ Software solution

• Application should contain ILP

3. What are the various hazards?[May/June 2014] [April/May 2017]

Three types of hazards

1.Structural 2.Data dependence 3.Control dependence

4. What is dynamic scheduling? [May/June 2013](NOV/DEC 2016)

• *Dynamic scheduling*: hardware rearranges instruction execution to reduce stalls Allow instructions behind stall to proceed

5. What are the advantages of dynamic scheduling? (Apr/may 2017)

• Handles cases when dependences unknown at compile time

– e.g., because they may involve a memory reference

• It simplifies the compiler

• Allows code compiled for one machine to run efficiently on a different machine, with different number of function units (FUs), and different pipelining

6. What is branch prediction? [May/June 2013]

• High branch penalties in pipelined processors:

– With on average 20% of the instructions being a branch, the maximum ILP is five

• CPI = CPIbase + fbranch * fmisspredict * penalty

– Large impact if:

– Penalty high: long pipeline

– CPIbase low: for multiple-issue processors

7. What is speculation? [May/June 2012]

Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a *data flow execution:* Operations execute as soon as their operands are available.

8. What are the four steps involved in instruction execution? [R-2013]

1.Issue 2. Execute 3.Write result 4.Commit

9. How to calculate the value of CPI. [R-2013]

The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls: Pipeline CPI = Ideal pipeline CPI+Structuralstalls+Data hazard stalls + Control stalls

10. What is ideal pipeline CPI? [Nov/Dec 2013]

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock).

11. What are the types of data dependencies?

[April/May 2015] (NOV/DEC 2016)

There are three different types of dependences: *data dependences* (also called true data dependences), *name dependences,* and *control dependences.*

12. What is instruction commit? [R-2013]

When an instruction is no longer speculative, we allow it to update the register file or memory; we call this additional step in the instruction execution sequence *instruction commit.*

13. What is reorder buffer? [R-2013]

Instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer is called the *reorder buffer.*

14. What is control speculation? [R-2013]

Loads incur high latency

– Need to schedule loads as early as possible

– Two barriers – branches and stores

Control speculation – move loads above branches

15. What is principle of locality? [Nov/Dec 2016]

An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

PART B

1. (i)What is multithreading and explain the approaches of multithreading. [May/Jun 2014], [Apr/ May 2015]

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread.

Software and hardware multithreading

- A "hardware thread" is a physical CPU or core. So, a 4 core CPU can support 4 hardware threads at once - the CPU really is doing 4 things at the same time.
- One hardware thread can run many software threads. In modern operating systems, this is often done by time-slicing - each thread gets a few milliseconds to execute before the OS schedules another thread to run on that CPU. Since the OS switches back and forth between the threads quickly, it appears as if one CPU is doing more than one thing at once, but in reality, a core is still running only one hardware thread, which switches between many software threads.
- Modern JVMs map java threads directly to the native threads provided by the OS, so there is no inherent overhead introduced by java threads vs native threads. As to hardware threads, the OS tries to map threads to cores, if there are sufficient cores. So, if you have a java program that starts 4 threads, and have more 4 or more cores, there's a good chance your 4 threads will run truly in parallel on 4 separate cores, if the cores are idle.
- **Multithreading** computers have hardware support to efficiently execute multiple threads. These are distinguished from multiprocessing systems (such as multi-core systems) in that the threads have to share the resources of a single core: the computing units, the CPU caches and the translation look aside buffer (TLB).
- Where multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by leveraging thread-level as well as instruction-level parallelism. As the two techniques

are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.

There are two main approaches to multithreading.

Cycle-by-cycle interleaving (Fine Grained Multithreading)

Fine-grained multithreading switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.

The next instruction of a thread is fed into the pipeline after the retirement of the previous instruction. This eliminates the need for forwarding data paths, but implies that there must be as many threads as pipeline stages. This can be a problem for contemporary super pipelined processors. Furthermore, in order to fully hide the memory latency, the number of threads must be larger than the memory latency in cycles.

Block interleaving (Coarse Grained Multithreading)

Coarse-grained multithreadingwas invented as an alternative to fine-grained multithreading. A coarse-grained multithreading switches thread only on costly stalls, such as level two caches misses.

It is also called as block interleaving; the processor starts executing another thread if the current thread experiences an event that is predicted to have a significantly long latency. If it can be predicted that the latency is larger than the cost of a thread switch, then the processor can at least hide part of the latency by executing another thread.

Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction-Level Parallelism:

Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

- Simultaneous multithreading (SMT) tries to eliminate *horizontalwaste*(unused instruction slots in a cycle) because it fetches and issues instructions from different threads simultaneously.
- SMT is designed for superscalar processors, however. To implement this technique in a VLIW processor, several VLIW instructions have to be combined at runtime. This can be very difficult and may increase the cycle time because resource conflicts may occur since not every operation can be placed in each instruction slot. a bit needs to be added to every VLIW instruction to indicate that it can be issued across multiple cycles without violating dependencies.

Figure illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

n a superscalar with no multithreading support,

n a superscalar with coarse-grained multithreading,

n a superscalar with fine-grained multithreading, and

n a superscalar with simultaneous multithreading.

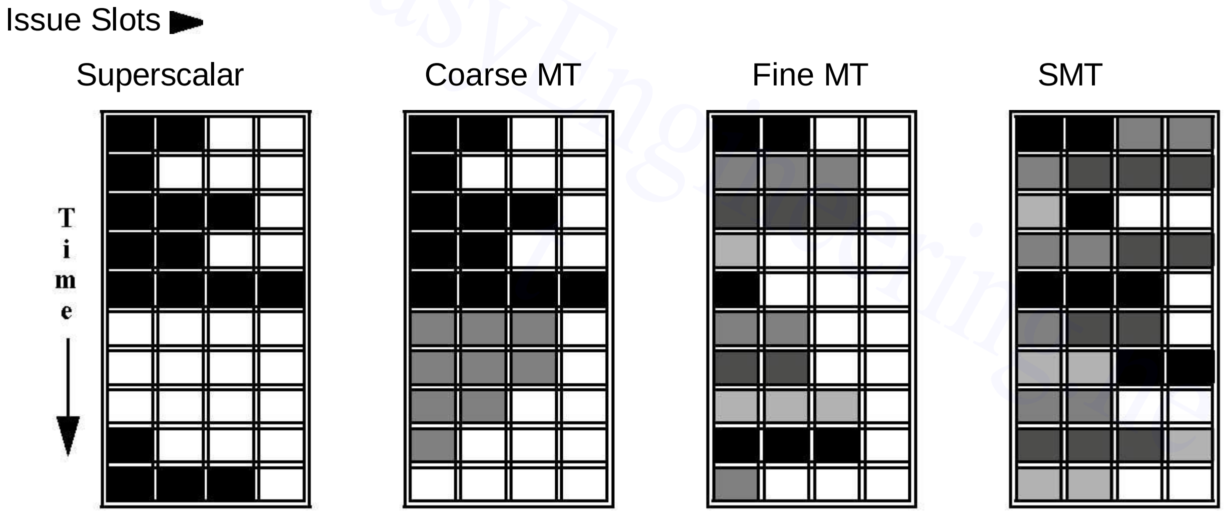


FIGURE **This illustration shows how these four different approaches use the issue slots of a superscalar processor.** The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP.

Hardware Multithreading Techniques

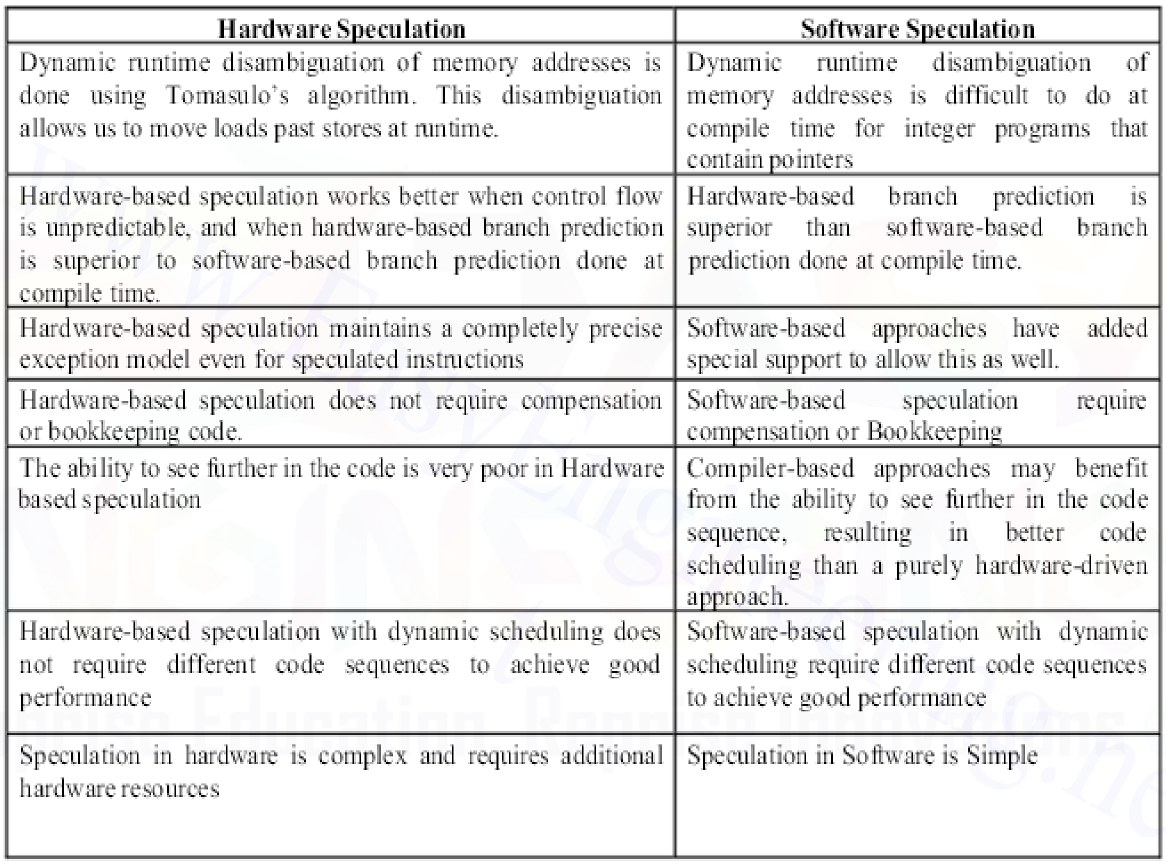
Basically, three different hardware multithreading techniques can be distinguished:

- cycle-by-cycle interleaving (Fine Grained Multithreading)
- block interleaving (Coarse Grain Multithreading)

- Simultaneous multithreading. (SMT)

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor.
In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle.
In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously; with multiple threads using the issue slots in a single clock cycle.

(ii)Compare hardware and software speculation mechanisms.



Hardware Speculation	Software Speculation
Dynamic runtime disambiguation of memory addresses is done using Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime.	Dynamic runtime disambiguation of memory addresses is difficult to do at compile time for integer programs that contain pointers
Hardware-based speculation works better when control flow is unpredictable, and when hardware-based branch prediction is superior to software-based branch prediction done at compile time.	Hardware-based branch prediction is superior than software-based branch prediction done at compile time.
Hardware-based speculation maintains a completely precise exception model even for speculated instructions	Software-based approaches have added special support to allow this as well.
Hardware-based speculation does not require compensation or bookkeeping code.	Software-based speculation require compensation or Bookkeeping
The ability to see further in the code is very poor in Hardware based speculation	Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling than a purely hardware-driven approach.
Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance	Software-based speculation with dynamic scheduling require different code sequences to achieve good performance
Speculation in hardware is complex and requires additional hardware resources	Speculation in Software is Simple

2. Explain hardware based speculation to overcome the control dependencies. [Apr/ May 2015], [Nov/ Dec 2014], [May/Jun 2013]

Hardware-based speculation combines three key ideas:

- **Dynamic branch prediction to choose which instructions to execute**
- **Speculation to allow the execution of instructions before the control dependences are resolved** (with the ability to undo the effects of an incorrectly speculated sequence)

- **Dynamic scheduling to deal with the scheduling of different combinations of basic blocks**.

  Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a data flow execution: Operations execute as soon as their operands are available.

Role of commit stage:

Using the bypassed value is like performing a speculative register read, since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory we call this additional step in the instruction execution sequence instruction *commit.*

KEY IDEA BEHIND HARDWARE SPECULATION:

- The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit in order and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits.
- Hence, when we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to commit. Adding this commit phase to the instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer, which we call the reorder buffer, is also used to pass results among instructions that may be speculated.

  ROLE OF RE-ORDER BUFFER:

- The reorder buffer (ROB) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set. **The ROB holds the result of an instruction between the times the operation associated with the instruction completes and the time the instruction commits**. Hence, the ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm.

- The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the register file.
- With speculation, the register file is not updated until the instruction commits (and we know definitively that the instruction should execute); thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit.

  Each entry in the ROB contains four fields:
- Instruction type
  - The instruction type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which has register destinations).
- Destination field
  - The destination field supplies the register number (for loads and ALU operations) or the memory address (for stores) where the instruction result should be written.
- Value field
  - The value field is used to hold the value of the instruction result until the instruction commits.
- Ready field.
  - The ready field indicates that the instruction has completed execution, and the value is ready.

Here are the four steps involved in instruction execution:

1. **Issue**—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB; send the operands to the reservation station if they are available in either the registers or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB entry allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries.
2. **Execute**—If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards.

When both operands are available at a reservation station, execute the operation. Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation.

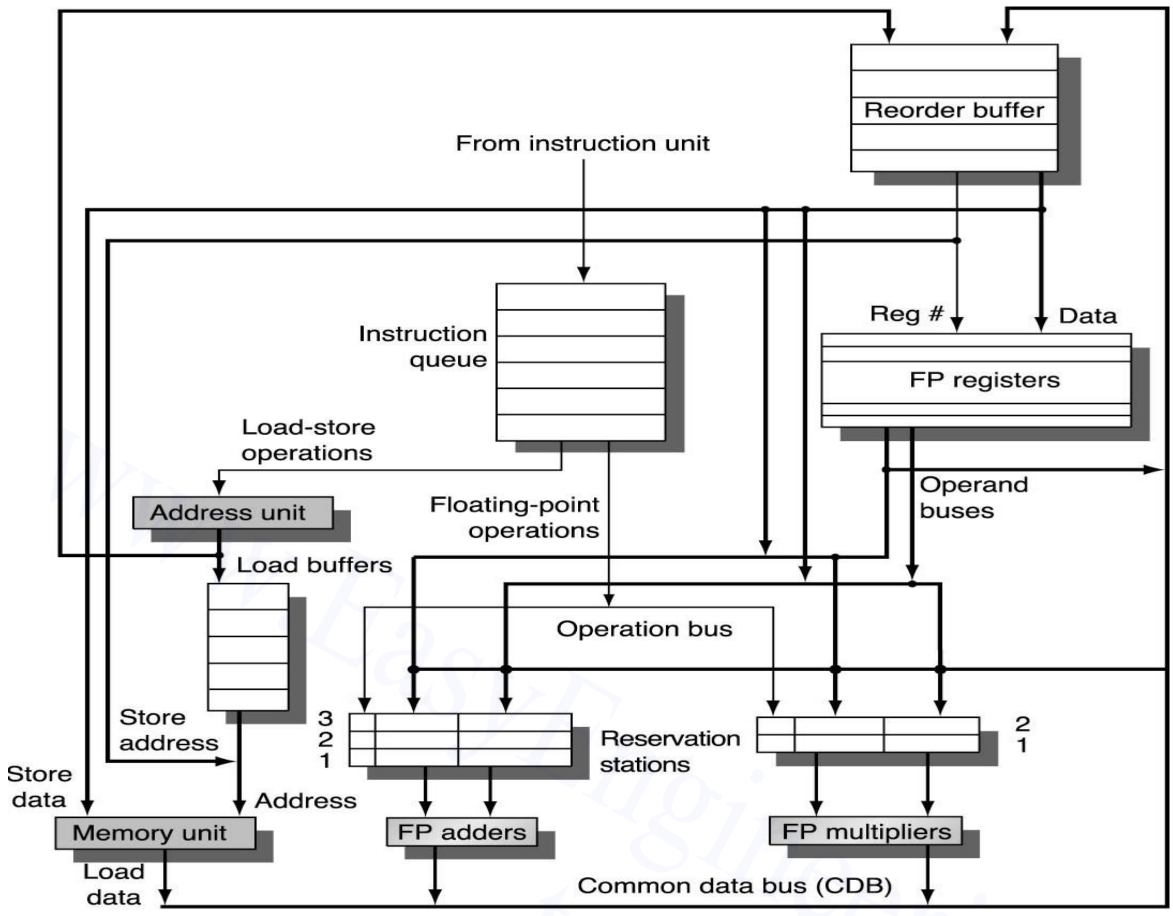


Figure: The basic structure of a MIPS floating-point unit using Tomasulo's algorithm.

3. **Write result**—When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity we assume that this occurs during the Write Results stage of a store; we discuss relaxing this requirement later.

4. **Commit**—this is the final stage of completing an instruction, after which only its result remains. (Some processors call this commit phase "completion" or

"graduation.") There are three different sequences of actions at commit depending on whether the committing instruction is a branch with an incorrect prediction, a store, or any other instruction (normal commit).

- The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB.
- When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished.

3. i)Describe the basic compiler techniques for exploiting Instruction Level Parallelism (ILP). [Apr/ May 2015], [Nov/ Dec 2014], [May/Jun 2013], [May/Jun 2014]

Basic Pipeline Scheduling and Loop Unrolling

- To keep a pipeline full, parallelism among instructions must be exploited by finding sequences 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 a distance in clock cycles equal to the pipeline latency of that source instruction.
- Consider the following code segment, which adds a scalar to a vector:

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

We can see that this loop is parallel by noticing that the body of each iteration is independent. First, let's look at the performance of this loop, showing how we can use the parallelism to improve its performance for a MIPS pipeline with the latencies shown above.

Instruction producing result	**Instruction using result**	**Latency in clock cycles**
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

Load double Store double 0

The first step is to translate the above segment to MIPS assembly language. In the following code segment, RI is initially the address of the element in the array with the highest address, and F2 contains the scalar value s. Register R2 is precompiled, so that 8(R2) is the address of the last element to operate on. The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop:   L.D FO, O(R1)                   ;F0=array
element
        ADD.D F4, F0, F2            ; add scalar in F2
        S.D F4,0(R1)                    ;store result
        DADDUI Rl,Rl,#-8          ;decrement pointer
```

also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together.

In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus, we will want to use different registers for each iteration, increasing the required number of registers.

The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the code above has no stalls.

Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Without any scheduling the loop will execute as follows:

```
                              Clock cycle issued
Loop:  L.D     F0,0(R1)        1
stall                          2
       ADD.D F4,F0,F2          3
stall                          4
stall                          5
       S.D F4,0(R1)                 6
       DADDUI R1,R1,#-8        7
stall                          8
       BNE R1,R2,Loop          9
stall                          10
```

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop: L.D  F0,0(R1)
DADDUI R1,R1,#-8
ADD.D F4,F0,F2
stall
BNE R1,R2,Loop ;   delayed branch
S.D F4,8(R1) ;                altered & interchanged with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6. The stall after ADD.D is for the use by the S.D.

WITH UNROLLING:

```
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)
BNE R1,R2,Loop
S.D F16,8(R1) ;8-32 = -24
```

ii) Explain the various dynamic branch prediction schemes. [Nov/ Dec 2014], [May/Jun 2013]

Dynamic Branch Prediction and Branch-Prediction Buffers

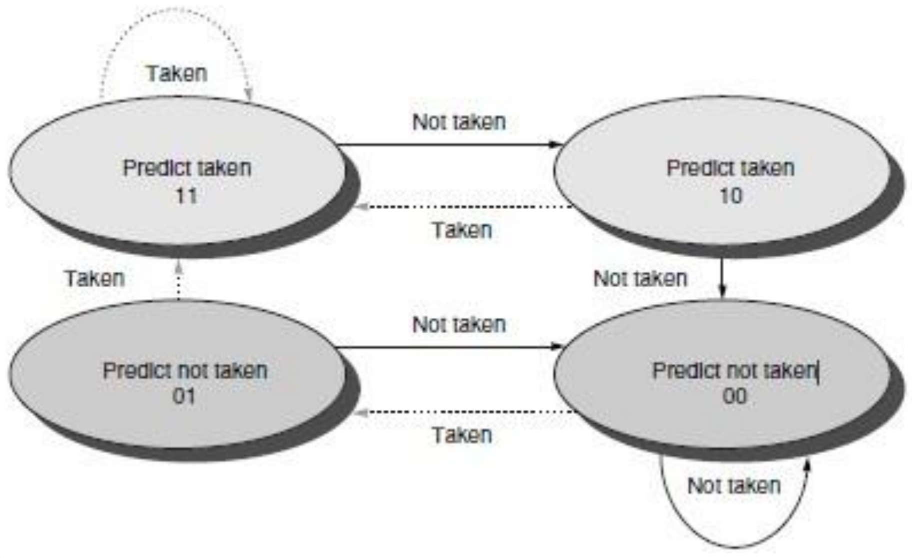
The simplest dynamic branch-prediction scheme is a branch-prediction buffer or ***branch history table****. A* ***branch-prediction buffer is a small memory indexed*** *by* the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs. With such a buffer, we don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits.

- But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This simple 1-bit prediction scheme ---- has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used. In a **2-bit scheme**, a prediction must miss twice before it is changed. Figure shows the finite-state processor for a 2-bit prediction scheme.

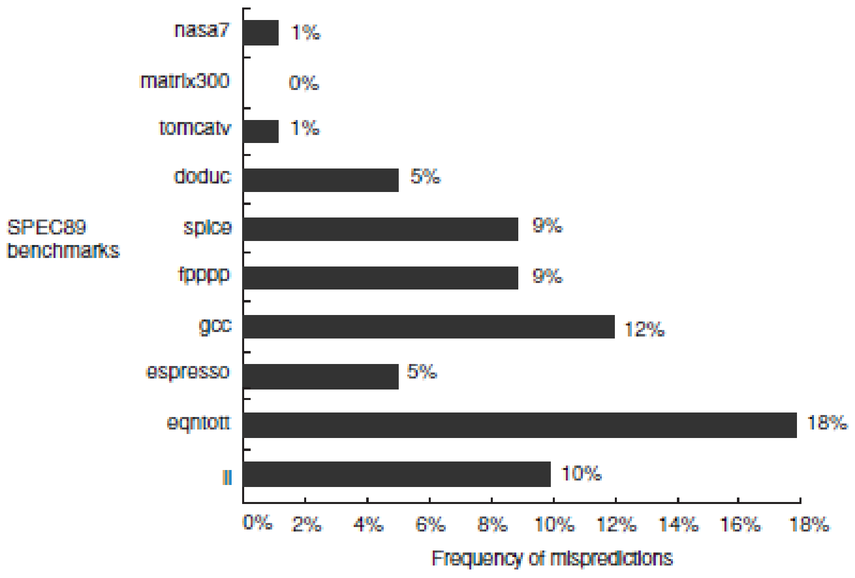
A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue., if the prediction turns out to be wrong, the prediction bits are changed.



Correlating Branch Predictors

The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of other branches rather than just the branch we are trying to predict. Consider a small code fragment from the eqntott benchmark, a member of early SPEC benchmark suites that displayed particularly bad branch prediction behavior:

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```



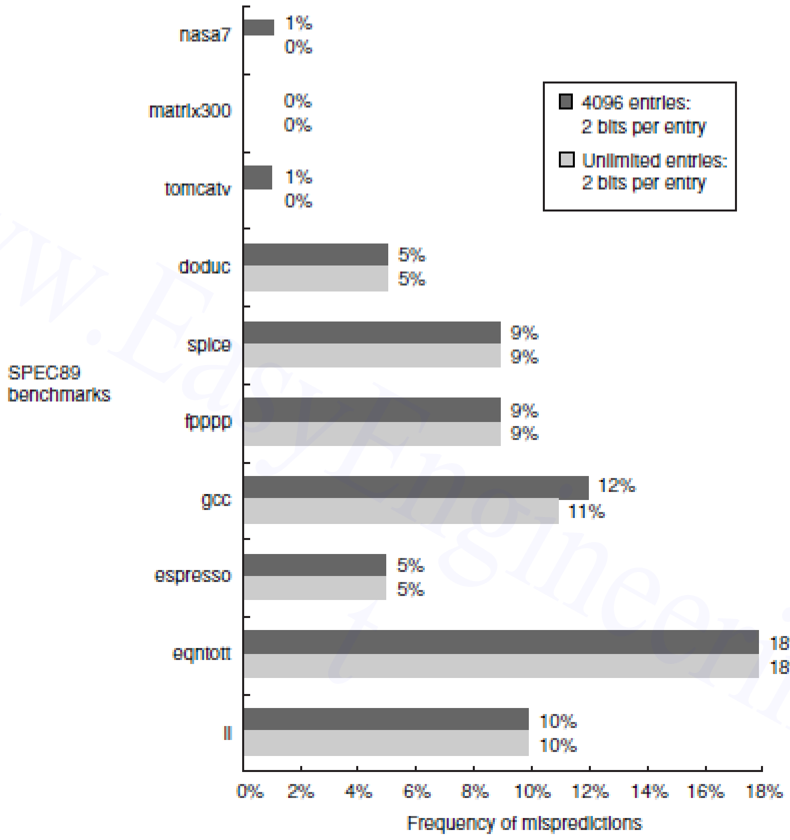
- **Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors**.
- Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch. For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch. In the general case an (m,n) predictor uses the behavior of the last m branches to choose from 2m branch predictors, each of which is an n-bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the 2-bit scheme and requires only a trivial amount of additional hardware.

Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that, by adding global information, the performance could be improved. Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector.

Tournament predictors can achieve both better accuracy at medium sizes (8K- 32K bits) and also make use of very large numbers of prediction bits effectively. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some mix) was most effective in recent predictions.

As in a simple 2-bit predictor, the saturating counter requires two mispredictions before changing the identity of the preferred predictor. The advantage of a tournament predictor is its ability to select the right predictor for a particular branch, which is particularly crucial for the 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.



The above figure looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. As we saw earlier, the prediction capability of the local predictor does not improve beyond a certain size. The correlating predictor shows a significant improvement, and the tournament predictor generates slightly better performance. For more recent versions of the SPEC, the results would be similar, but the asymptotic behavior would not be reached until slightly larger-sized predictors.

4. What is Dynamic Scheduling? Explain how it is used to reduce data hazards. Also explain dynamic scheduling using Tomosulo's approach with an example. [Nov/ Dec 2013]

Dynamic scheduling

- A major limitation of simple pipelining techniques is that they use in-order instruction issue and execution: Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed.
- Thus, if there is dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle. If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and 7 can execute.

For example, consider this code:

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

- The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet SUB. D is not data dependent on anything in the pipeline.
- This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order. In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved.
- To allow us to begin executing the SUB. D in the above example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. Thus, we still use in-order instruction issue (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operands are available. Such a pipeline does ***out-of-order execution, which implies out-of-order completion***.

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. Consider the following MIPS floating-point code sequence:

```
DIV.D F0,F2,F4
ADD.D F6,F0,F8
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```

- There is antidependence between the ADD. D and the SUB.D, and if the pipeline executes the SUB. D before the ADD. D (which is waiting for the DIV. D), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of F6 by MUL.D, WAW hazards must be handled. As we will see, both these hazards are avoided by
  the use of register renaming.
- Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise.
- Dynamically scheduled processors **preserve exception behavior** by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed; we will see shortly how this property can be guaranteed.
- Although exception behavior must be preserved, dynamically scheduled processors may generate **imprecise exceptions**. **An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order.** Imprecise exceptions can occur because of two possibilities:
  1. The pipeline may have already completed instructions that are later in pro-gram order than the instruction causing the exception.
  2. The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception. Imprecise exceptions make it difficult to restart execution after an exception. Rather than address these problems in this section, we will discuss a solution that provides precise exceptions in the context of a processor with speculation To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

- Issue—Decode instructions, check for structural hazards.
- Read operands—Wait until no data hazards, then read operands.

We distinguish when an instruction begins execution and when it completes ***execution; between the two times, the instruction is in execution. Our pipeline*** allows multiple instructions to be in execution at the same time, and without this capability, a major advantage of dynamic scheduling is lost. Having multiple instructions in execution at once requires multiple functional units, pipelined functional units, or both.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order.

Dynamic Scheduling using Tomosulo's approach

- **RAW hazards are avoided by executing an instruction only when its operands are available**.
- **WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. Register renaming eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.**

To better understand how register renaming eliminates WAR and WAW hazards; consider the following example code sequence that includes both a potential WAR and WAW hazard:

```
DIV.D F0.F2.F4
ADD.D F6,F0,F8
S.D F6,0(R1)
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```

There is antidependence between the ADD.D and the SUB.D and an output dependence between the ADD.D and the MilL.D, leading to two possible hazards: a WAR hazard on the use of F8 by ADD. D and a WAW hazard since the ADD. D may finish later than the MUL.D. There are also three true data dependences:

between the DIV.D and the ADD.D, between the SUB.D and the MUL.D, and between the ADD.D and the S.D.

These two name dependences can both be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, S and T. Using S and T, the sequence can be rewritten without any dependences as

```
DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D S,0(R1)
SUB.D T.F10.F14
MUL.D F6,F10,T
```

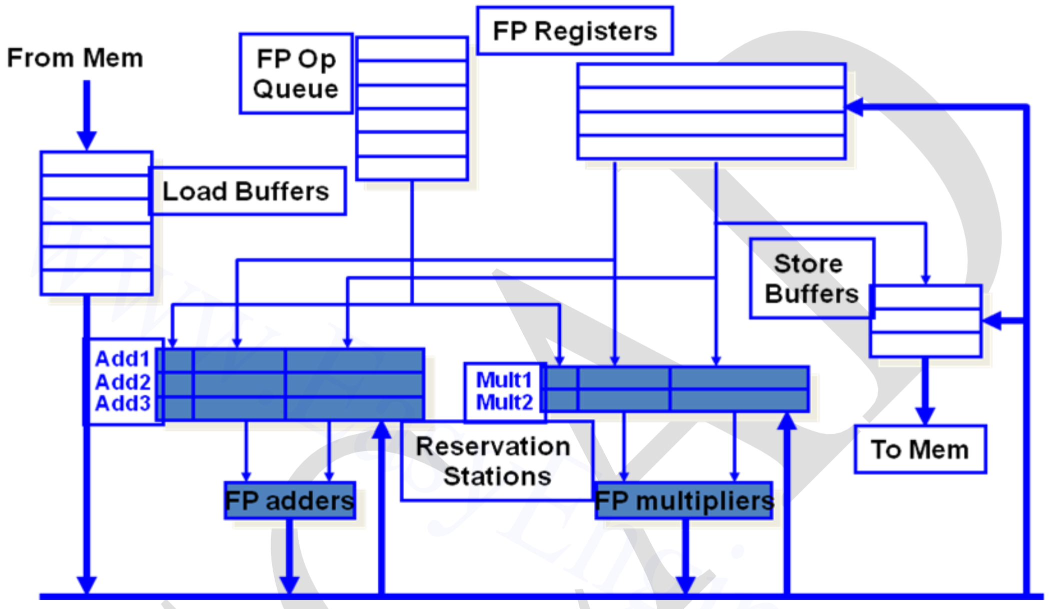
- In addition, any subsequent uses of F8 must be replaced by the register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of F8 that are later in the code requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the above code segment and a later use of F8.
- In Tomasulo's scheme, register renaming is provided by reservation stations which buffer the operands of instructions waiting to issue. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.
- In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming.
- Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler.

USE OF RESERVATION STATIONS:

- The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: The information held in the reservation stations at each functional unit determine when an instruction can begin execution at that unit.

- Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91 this is called the common data bus, or CDB). In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

The basic structure of a Tomasulo-based processor, including both the floating-point unit and the load-store unit; none of the execution control tables are shown.



COMPONENTS OF TOMOSULO'S ARCHITECTURE:

RESERVATION STATION:

- Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit, and either the operand values for that instruction, if they have already been computed, or else the names of the reservation stations that will provide the operand values.
- The load buffers and store buffers hold data or addresses coming from and going to memory and behave almost exactly like reservation stations, so we distinguish them only when necessary. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers.

RESERVATION STATION COMPONENTS:

- Op: operation to perform in the unit (e.g., + or –)
- Vj, Vk: value of Source operands

  - Store buffers has V field, result to be stored
- Qj, Qk: reservation stations producing source registers (value to be written)
  - Note: Qj,Qk=0 => ready
  - Store buffers only have Qi for RS producing result
- Busy: indicates reservation station or FU is busy
- Register result status: Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Steps an involved in tomosulo algorithm. There are only three steps, although each one can now take an arbitrary number of clock cycles:

- Issue—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards. (This stage is sometimes called dispatch in a dynamically scheduled processor.)
- Execute—If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into any reservation station awaiting it. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available, RAW hazards are avoided. (Some dynamically scheduled processors call this step "issue," but we use the name "execute," which was used in the first dynamically scheduled processor, the CDC 6600.)
  - Loads and stores require a two-step execution process. The first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait for the value to be stored before being sent to the memory unit.

- Write result—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores are buffered in the store buffer until both the value to be stored and the store address are available, then the result is written as soon as the memory unit is free.

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...
56	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Result		

5(i) Explain the concept of ILP with various types of dependencies in ILP. [Nov/ Dec 2013], [May/Jun 2013]

Data Dependences and Hazards

- If two instructions are paral*lel,* they can execute simultaneously in a pipeline of arbitrary depth withoutcausing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

There are three different types of dependences:

- Data dependences (also called true data dependences)
- Name dependences(also called anti dependence)
- Control dependences.

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j. Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program. Note that dependence within a single instruction (such as ADDD R1.R1.R1) is not considered dependence.

Loop: L.D F0,0(R1)

↘

ADD.D F4 ,F0,F2

↙

S.D F4,0(R1)

Explanation:

- Both of the above dependent sequences, as shown by the arrows, have each instruction depending on the previous one. The arrows here and in above examples show the order that must be preserved for correct execution.
- The arrow points from an instruction that must precede the instruction that the arrowhead points to. If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instruction

Name Dependences

The second type of dependence is name dependence. Name dependence occurs when two instructions use the same register or memory location, called a ***name, but there is no flow of data between the instructions associated with that*** **name**. There are two types of name dependences between an instruction i that *precedes* ***instruction j in program order***

1. An anti dependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the

correct value. In the example on page 69, there is an anti dependence between S. D and DADDIU on register Rl.

2. An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

   Example:

- Instr$_J$ writes operand *before* Instr$_I$ writes it.

I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7

- Called an "output dependence" by compiler writers. This also results from the reuse of name "r1"

Data Hazards

A hazard is created whenever there is a dependence between instructions*, and* they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called program order, that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order only where it affects the **out-*come of the program*.** ***Detecting and avoiding hazards ensures that necessary pro*gram order** is preserved.

<u>RAW (read after write)</u> —j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that j receives the value from i.

<u>WAW (write after write)</u> *—j tries to write an operand before it is written by i.* The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

WAR (write after read)—j tries to write a destination before it is read by i, so ***i incorrectly gets the new value. This hazard arises from an anti dependence***. WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID) and all writes are late (in WB).

Example: for WAR Hazard: Floating-point data part

```
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
```

Integer data part

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

Control Dependences

A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order.

```
if P1 {
      S1;
};

if P2 {
      S2;
}
```

S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

1. 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. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.

2. 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. For example,* we cannot take a statement before the if statement and move it into the then portion.

(ii) Briefly discuss the limitations of ILP also discuss the techniques to overcome these limitations.[Nov/ Dec 2014], [Nov/ Dec 2013]

Limits to ILP

- Conflicting studies of amount of ILP
  - ➢ Benchmarks - vectorized Fortran FP vs. integer C programs
  - ➢ Hardware sophistication
  - ➢ Compiler sophistication
- How much ILP is available using existing mechanisms with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - ➢ Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - ➢ Intel SSE2: 128 bit, including 2 64-bit FP per clock
  - ➢ Motorola AltiVec: 128 bit ints and FPs
  - ➢ Supersparc Multimedia ops, etc.

Overcoming Limits

- Advances in compiler technology and different hardware techniques may be able to overcome limitations assumed in studies
- However, unlikely such advances when coupled with realistic hardware will overcome these limits in near future

UNIT III
DATA LEVEL PARALLELISM
PART A

1. Define SIMD.?[May/June 2014]

Single instruction issue, single operation, but multiple data sets per operation. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.

2. What do you mean by Loop level parallelism? ?[May/June 2015]

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level Parallelism*

3. What is the need to detect loop dependencies? How does the compiler detect it? [R-2013]

- To support loop unrolling, the compiler must detect any dependencies that exist both within and between loop iterations
- Different compilers may use different loop dependency test like GCD test, Banerjee test to detect loop dependencies.

4. What is vector processing? [R-2013]

Vector Processing**:** Explicit coding of independent loops as operations on large vectors of numbers

– Multimedia instructions being added to many processors

5. Compare multi vector and SIMD computers. [R-2013](Apr/may 2017)

Multi vector	**SIMD**
It processes multiple word at a time through pipelined Processors	It consists multiple processing elements that perform the same operation on multiple data points simultaneously
Process in concurrent	It exploits data level parallelism, but not concurrency
Multimedia instructions being added to many processors	Used to improve the performance of multimedia applications

6. Name some SIMD languages. [R-2013]

- HLSL
- Cg,
- GLSL
- OpenCL
-

7. Compare CPU and GPU. [R-2013](NOV/DEC 2016)

GPU

- Same operations on many primitives (SIMD)
- Focus on throughput over latency
- Lots of special purpose hardware

CPU

- Focus on reducing Latency
- Designed to handle a wider range of problems

8. What are advantages and disadvantages of GPUs? [R-2013]

Advantages

- Supercomputer-like FP performance on commodity processors

Disadvantages

- Performance tuning difficult
- Large speed gap between compiler-generated and hand-tuned code.

9. Mention the limitations of GPUs. ?[Nov/Dec 2014]

GPU Limitations

- Relatively small amount of memory, less than 4GB in current GPUs
- I/O directly to GPU memory has complications – It Must transfer to host memory, and then back

10. What is Chaining? [R-2013]

Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available.

11. Mention the Advantages of SIMD over vector architecture. Apr/may 2017)

- Cost little to add to the standard ALU and easy to implement

- Require little extra state → easy for context-switch
- Require little extra memory bandwidth
- No virtual memory problem of cross-page access and page-fault

12. What are the Disadvantage of Vector Processor? [R-2013]

- limited to regular data and control structures
- Vector Registers and buffers
- memory BW

13. Why vector processors popular in scientific calculations? [R-2013]

A computer designed to apply arithmetic operations to long vectors or arrays. Most vector processors rely heavily on *pipelining* to achieve high performance.

14. What is a vector? [R-2013]

Vectorisan ordered list of items in a computer's memory. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as indices to the base.

15. How does the compiler now which code to compile for CPU and which one for GPU?
[R-2013]

- Specifier tells compiler where function will be executed
- Executed on CPU, called form CPU
- CUDA kernel to be executed on GPU
- CUDA kernel to be executed on GPU

16. What are the primary components of instruction set architecture of VMIPS ? (NOV/DEC 2016)

- **Vector registers**
- **Vector fuctional unit**
- **Vector load/store unit**
- **Set a scalar registers.**

PART B

1. Explain how do you detect and enhance Loop Level Parallelism? [R-2013]

Loop Carried Dependence:

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations, such dependence is called a loop-carried dependence.

To see that a loop is parallel, let us first look at the source representation:

```
for (i=1000; i>0; i=i–1)
x[i] = x[i] + s;
```

In this loop, there is dependence between the two uses of x[i], but this dependence is within a single iteration and is not loop-carried.

There is dependence between successive uses of i in different iterations, which is loop-carried, but this dependence involves an induction variable and can be easily recognized and eliminated.

EXAMPLE Consider a loop like this one:

```
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 */
}
```

Assume that A, B, and C are distinct, non-overlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure, which includes this loop, determining whether arrays overlap or are identical often requires sophisticated, inter-procedural analysis of the program.)

What are the data dependences among the statements S1 and S2 in the loop?

There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1], which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
2. S2 uses the value, A[i+1], computed by S1 in the same iteration.

These two dependences are different and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement S1 on an earlier iteration of S1, this dependence is loop-carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence above (S2 depending on S1) is within an it-21eration and is not loop-carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in iteration were kept in order.

Consider a loop like this one:

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

Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.

2. On the first iteration of the loop, statement S1 depends on the value of B[1] computed prior to initiating the loop.

These two observations allow us to replace the loop above with the following code sequence:

```
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];22
}
B[101] = C[100] + D[100];
```

Finding Dependences

Finding the dependences in a program is an important part of three tasks:

- good scheduling of code,
- determining which loops might contain parallelism
- Eliminating name dependences.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are affined. In simplest terms, a one-dimensional array index is affine if it can be written in the form a × i + b, where ‘a’ and ‘b’ are constants, and i is the loop index variable.

Dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop. That is m ≤ j ≤ n, m ≤ k ≤ n.
2. The loop stores into an array element indexed by a × j + b and later fetches from that same array element when it is indexed by c × k + d. That is, a × j + b = c × k + d.24

GCD TEST:

As an example, a simple and sufficient test for the absence of a dependence is the greatest common divisor, or GCD, test. It is based on the observation that if a loop-carried dependence exists, then GCD (c,a) must divide (d – b). (Recall that an integer, x, divides another integer, y, if there is no remainder when we do the division y/x and get an integer quotient.)

EXAMPLE Use the GCD test to determine whether dependences exist in the following loop:

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

ANSWER Given the values a = 2, b = 3, c = 2, and d = 0, then GCD(a,c) = 2, and d – b = –3. Since 2 does not divide –3, no dependence is possible.

The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not take the loop bounds into account.

Eliminating Dependent Computations:

Compilers can reduce the impact of dependent computations so as to achieve more ILP. The key technique is to eliminate or reduce a dependent computation by back substitution, which increases the amount of parallelism and sometimes increases the amount of computation required.

Within a basic block, algebraic simplifications of expressions and an optimization called **copy propagation**, which eliminates operations that copy values, can be used to simplify sequences like the following:

```
DADDUI R1,R2,#4
DADDUI R1,R1,#4
```

to:

```
DADDUI R1,R2,#8
```

assuming this is the only use of R1. In fact, the techniques we used to reduce multiple increments of array indices during loop unrolling and to move the increments across memory addresses.

In these examples, computations are actually eliminated, but it also possible that we may want to increase the parallelism of the code, possibly even increasing the number of operations. Such optimizations are called tree height reduction, since they reduce the height of the tree structure representing a computation, making it wider but shorter. Consider the following code sequence:

```
ADD R1,R2,R3
ADD R4,R1,R6
ADD R8,R4,R7
```

Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as:

```
ADD R1,R2,R3
ADD R4,R6,R7
ADD R8,R1,R4
```

This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Software Pipelining: Symbolic Loop Unrolling

There are two other important techniques that have been developed for this purpose: software pipelining and trace scheduling.

Software pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. This approach is most easily understood by looking at the scheduled code for the superscalar version of MIPS, which appeared in Figure.

The scheduler in this example essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within single loop iteration. By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.

Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1,#-8
BNE R1, R2, Loop
```

Software pipelining symbolically unrolls the loop and then selects instructions from each iteration. Since the unrolling is symbolic, the loop overhead instructions (the DADDUI and BNE) need not be replicated. Here’s the body of the unrolled loop without overhead instructions, highlighting the instructions taken from each iteration:

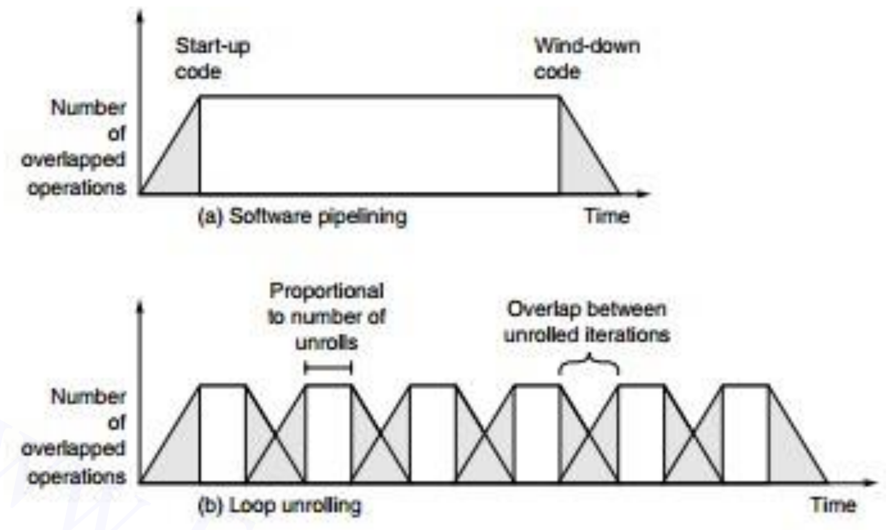
```
Iteration i: L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1)
Iteration i+1: L.D F0,0(R1)
ADD.D F4,F0,F2
S.D 0(R1),F4

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

The selected instructions from different iterations are then put together in the loop with the loop control instructions:

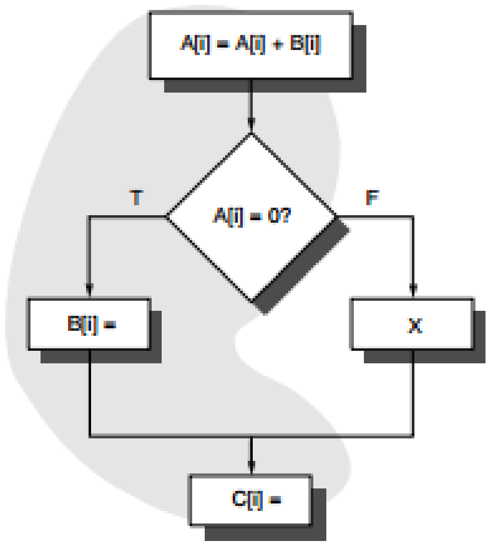
```
Loop: S.D F4,16(R1) ;stores into M[i]
ADD.D F4,F0,F2 ;adds to M[i-1]
L.D F0,0(R1) ;loads M[i-2]
DADDUI R1,R1,#-8
BNE R1,R2,Loop
```

This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions, and assuming that DADDUI is scheduled after the ADD.D and the L.D instruction, with an adjusted offset, is placed in the branch delay slot. Because the load and store are separated by offsets of 16 (two iterations), the loop should run for two fewer iterations.


Start-up code
Wind-down code
Number of overlapped operations
(a) Software pipelining
Time
Proportional to number of unrolls
Overlap between unrolled iterations
Number of overlapped operations
(b) Loop unrolling
Time

Global Code Scheduling

- **In general, effective scheduling of a loop body with internal control flow will require moving instructions across branches, which is global code scheduling. In this section, we first examine the challenge and limitations of 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. The data dependences force a partial order on operations, while the control dependences dictate instructions across which code cannot be easily moved.
- Finding the shortest possible sequence for a piece of code means finding the shortest sequence for the critical path, this is the longest sequence of dependent instructions.
- Control dependences arising from loop branches are reduced by unrolling.



Global code scheduling can reduce the effect of control dependences arising from conditional non-loop 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.

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. Such global code motion must satisfy a set of constraints to be legal. In addition, the movement of the code associated with B, unlike that associated with C, is speculative:

To perform the movement of B, we must ensure that neither the data flow nor the exception behavior is changed. Compilers avoid changing the exception behavior by not moving certain classes of instructions, such as memory references, that can cause exceptions.

```
LD R4,0(R1) ;            load A
LD R5,0(R2) ;            load B
DADDU R4,R4,R5 ;         Add to A
SD 0(R1),R4 ;            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: Focusing on the Critical Path

Trace scheduling is useful for processors with a large number of issues per clock, where conditional or predicated execution is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. **Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths.**

There are two steps to trace scheduling.

- The first step, called **trace selection**, tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a trace. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Additionally, by using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks.
- Once a trace is selected, the second process, called trace compaction, tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

Advantage of Trace Scheduling:

The advantage of the trace scheduling approach is that it simplifies the decisions concerning global code motion. In particular, branches are viewed as jumps into or out of the selected trace, which is assumed to the most probable path.

2. Explain in detail about how SIMD extensions exploit data-level parallelism. [R-2013]

<u>SIMD Extensions</u>

- **Media applications operate on data types narrower than the native word size**

➢ Example: disconnect carry chains to "partition" adder

- **Limitations, compared to vector instructions:**

➢ Number of data operands encoded into op code

➢ No sophisticated addressing modes (strided, scatter-gather)

➢ No mask registers

<u>SIMD Implementations</u>

- **Intel MMX (1996)**

➢ Eight 8-bit integer ops or four 16-bit integer ops

- **Streaming SIMD Extensions (SSE) (1999)**

➢ Eight 16-bit integer ops

➢ Four 32-bit integer/fpops or two 64-bit integer/fpops

- **Advanced Vector Extensions (2010)**

➢ Four 64-bit integer/fpops

- **Operands must be consecutive and aligned memory locations**
- **Generally designed to accelerate carefully written libraries rather than for compilers**
- **Advantages over vector architecture:**

➢ Cost little to add to the standard ALU and easy to implement

➢ Require little extra state □easy for context-switch

➢ Require little extra memory bandwidth

➢ No virtual memory problem of cross-page access and page-fault

<u>Example SIMD Code</u>

- **Example DXPY:**

```
L.DF0,a;load scalar a
MOVF1, F0;copya into F1 for SIMD MUL
MOVF2, F0;copya into F2 for SIMD MUL
MOVF3, F0;copya into F3 for SIMD MUL
```

```
DADDIUR4,Rx,#512;last address to load
Loop:L.4D F4,0[Rx];load X[i], X[i+1], X[i+2], X[i+3]
MUL.4DF4,F4,F0;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
L.4DF8,0[Ry];load Y[i], Y[i+1], Y[i+2], Y[i+3]
ADD.4DF8,F8,F4;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
S.4DF8,0[Ry] ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
DADDIURx,Rx,#32;increment index to X
DADDIURy,Ry,#32;increment index to Y
DSUBUR20,R4,Rx;compute bound
BNEZR20,Loop;check if done
```

<u>Roofline Performance Model</u>

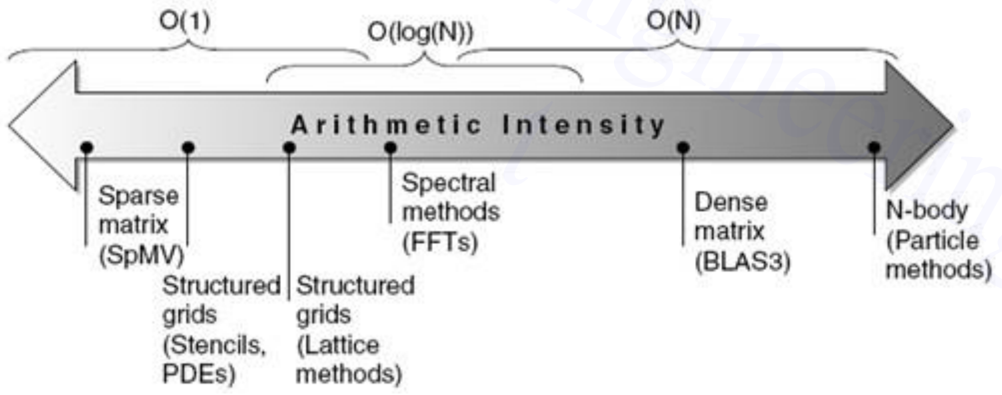
- **Basic idea:**

  Plot peak floating-point throughput as a function of arithmetic intensity

  Ties together floating-point performance and memory performance for a target machine

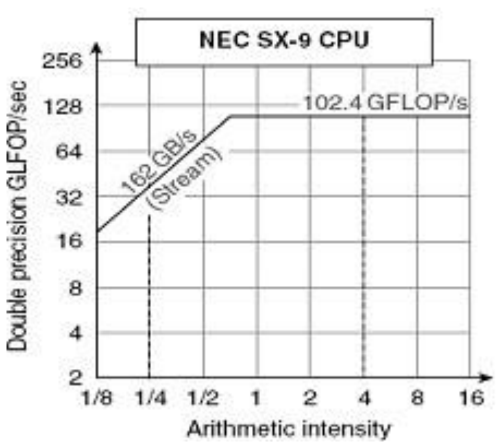
- **Arithmetic intensity**

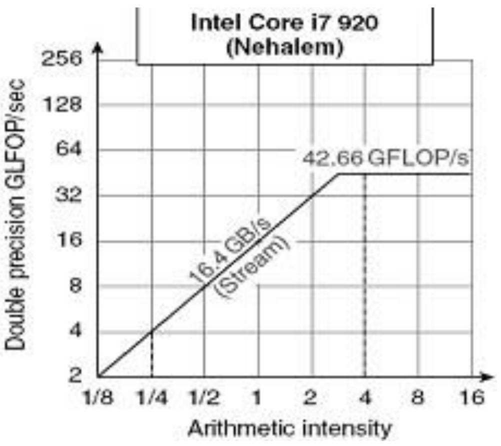
  Floating-point operations per byte read



Examples

Attainable GFLOPs/sec Min = (Peak Memory BW ×Arithmetic Intensity, Peak Floating Point Perf.)





3. Explain in detail about Vector Architectures of data-level parallelism . [R-2013]

<u>**Vector Architectures**</u>

- **Basic idea:**

➢ Read sets of data elements into "vector registers"

➢ Operate on those registers

➢ Disperse the results back into memory

- **Registers are controlled by compiler**

➢ Used to hide memory latency

➢ Leverage memory bandwidth

<u>**VMIPS**</u>

- **Example architecture: VMIPS**
- **Loosely based on Cray-1**
- **Vector registers**

➢ Each register holds a 64-element, 64 bits/element vector

➢ Register file has 16 read ports and 8 write ports

- **Vector functional units**

➢ Fully pipelined

➢ Data and control hazards are detected

- **Vector load-store unit**

➢ Fully pipelined

➢ One word per clock cycle after initial latency

- **Scalar registers**

➢ 32 general-purpose registers

This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

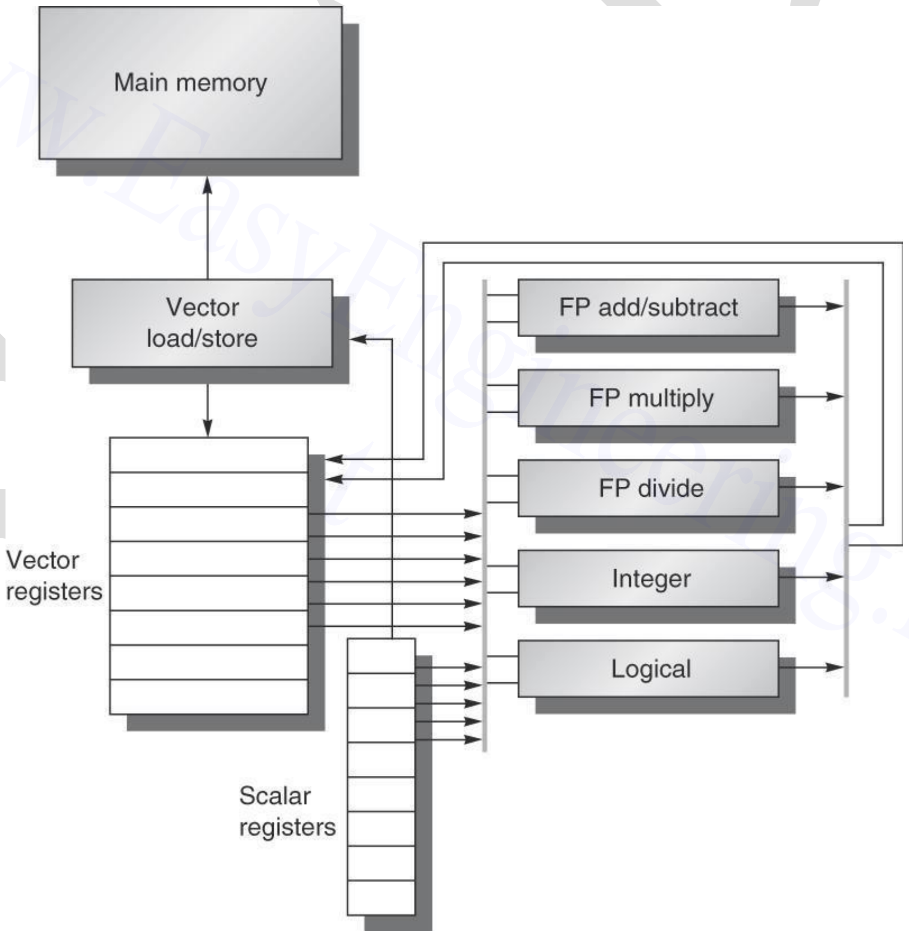


Fig. The basic structure of vector architecture, VMIPS.

<u>**VMIPS Instructions**</u>

- **ADDVV.D: add two vectors**

- **ADDVS.D: add vector to a scalar**
- **LV/SV: vector load and vector store from address**
- **Example: DAXPY (double precision a*X+Y)**

➢ L.DF0,a; load scalar a
➢ LVV1,Rx; load vector X
➢ MULVS.DV2,V1,F0; vector-scalar multiply
➢ LVV3,Ry; load vector Y
➢ ADDVVV4,V2,V3; add
➢ SVRy,V4; store the result

- **Requires 6 instructions**
- **<u>DAXPY in MIPS Instructions</u>**

Example: DAXPY (double precision a*X+Y)

L.DF0,a; load scalar a
DADDIUR4,Rx,#512; last address to load
Loop: L.DF2,0(Rx); load X[i]
MUL.DF2,F2,F0; a x X[i]
L.DF4,0(Ry); load Y[i]
ADD.DF4,F2,F2; a x X[i] + Y[i]
S.DF4,9(Ry); store into Y[i]
DADDIURx,Rx,#8; increment index to X
DADDIURy,Ry,#8; increment index to Y
SUBBUR20,R4,Rx; compute bound
BNEZR20,Loop; check if done

- **Requires almost 600 MIPS ops**

<u>Vector Execution Time</u>

- **Execution time depends on three factors:**

➢ Length of operand vectors
➢ Structural hazards
➢ Data dependencies

- **VMIPS functional units consume one element per clock cycle**
- **Execution time is approximately the vector length**
- **Convoy**

➢ Set of vector instructions that could potentially execute together

Chimes

- **Sequences with read-after-write dependency hazards can be in the same convey via chaining**
- **Chaining**

Allows a vector operation to start as soon as the individual elements of its vector source operand become available

- **Chime**

➢ Unit of time to execute one convey

➢ mconveys executes in mchimes

➢ For vector length of n, requires mx nclock cycles

Example

LVV1,Rx;load vector X

MULVS.DV2,V1,F0;vector-scalar multiply

LVV3,Ry;load vector Y

ADDVV.DV4,V2,V3;add two vectors

SVRy,V4;store the sum

Convoys:

1LVMULVS.D

2LVADDVV.D

3SV

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 64 x 3 = 192 clock cycles

Challenges

- **Start up time**

➢ Latency of vector functional unit

➢ Assume the same as Cray-1

❖ Floating-point add => 6 clock cycles

❖ Floating-point multiply => 7 clock cycles

❖ Floating-point divide => 20 clock cycles

❖ Vector load => 12 clock cycles

- **Optimizations:**

➢ Multiple Lanes: > 1 element per clock cycle

- Vector Length Registers: Non-64 wide vectors
- Vector Mask Registers: IF statements in vector code
- Memory Banks: Memory system optimizations to support vector processors
- Stride: Multiple dimensional matrices
- Scatter-Gather: Sparse matrices
- Programming Vector Architectures: Program structures affecting performance

<u>Multiple Lanes</u>

- **Element n of vector register A is "hardwired" to element n of vector register B**

- Allows for multiple hardware lanes

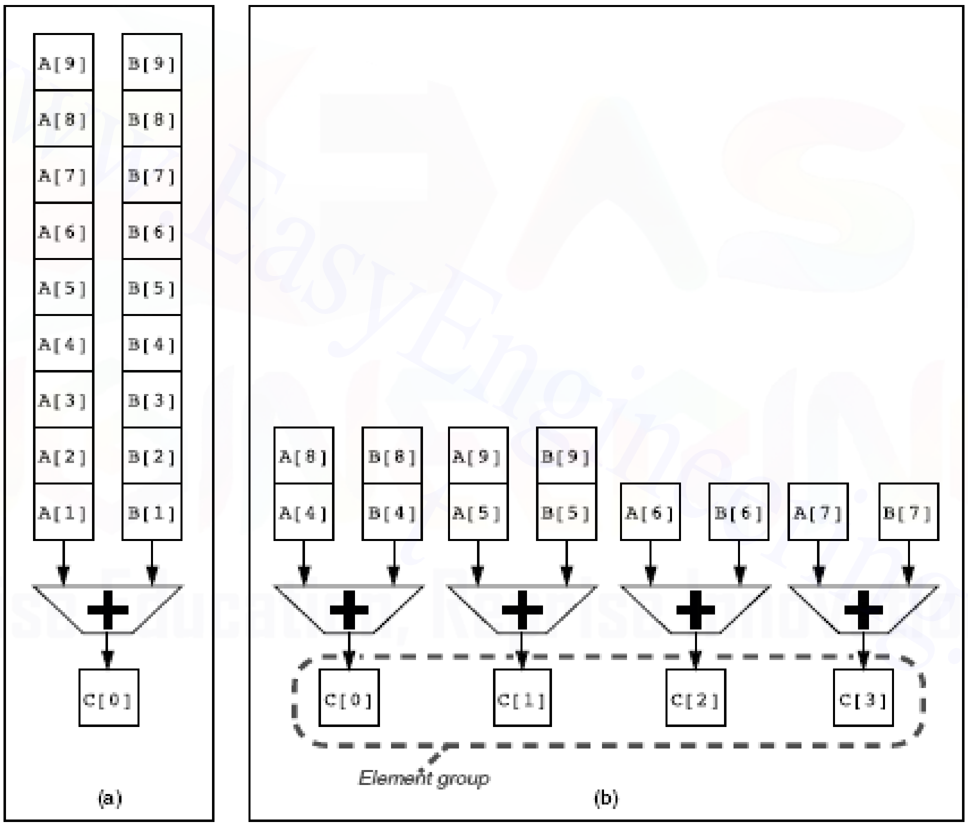
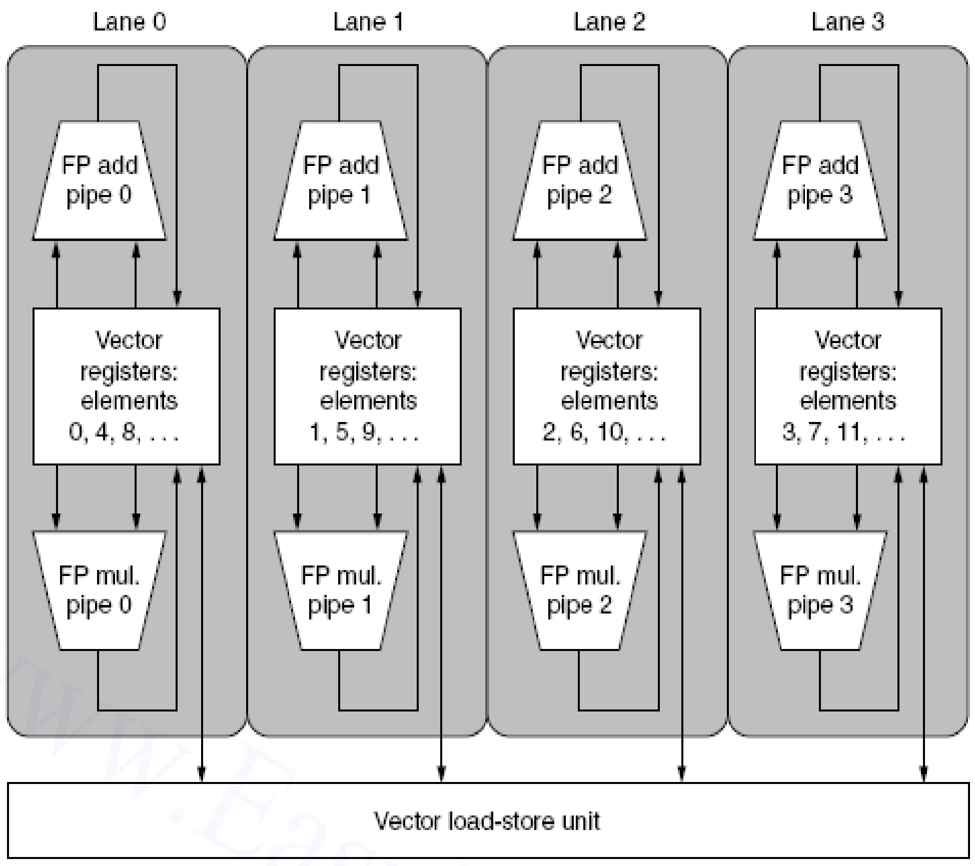


Figure: Element n of vector register A is "hardwired" to element n of vector register B



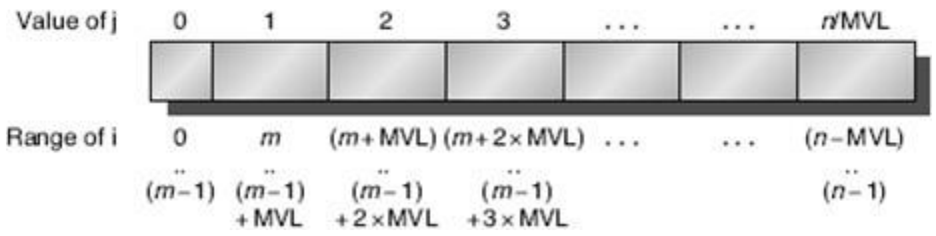
Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Figure: Vector load store-unit with pipelining

<u>Vector Length Registers</u>

- **Vector length not known at compile time?**
- **Use Vector Length Register (VLR)**
- **Use strip mining for vectors over the maximum length:**

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
for (i= low; i< (low+VL); i=i+1) /*runs for length VL*/
Y[i] = a * X[i] + Y[i] ; /*main operation*/
low = low + VL; /*start of next vector*/
VL = MVL; /*reset the length to maximum vector length*/
}
```



Vector Mask Registers

- **Consider:**

for (i = 0; i < 64; i=i+1)

if (X[i] != 0)

X[i] = X[i] –Y[i];

- **Use vector mask register to "disable" elements (if conversion):**

LVV1, Rx; load vector X into V1

LVV2, Ry; load vector Y

L.DF0, #0; load FP zero into F0

SNEVS.DV1, F0; sets VM (i) to 1 if V1 (i)! =F0

SUBVV.DV1, V1, V2; subtract under vector mask

SVRx, V1; store the result in X

- **GFLOPS rate decreases**

Memory Banks

- **Memory system must be designed to support high bandwidth for vector loads and stores**
- **Spread accesses across multiple banks**

➢ Control bank addresses independently

➢ Load or store non sequential words

➢ Support multiple vector processors sharing the same memory

- **Example:**

➢ 32 processors, each generating 4 loads and 2 stores/cycle

➢ Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns

➢ How many memory banks needed?

❖ 32x6=192 accesses,

- 15/2.167≈7 processor cycles
- 1344!

Stride

- **Consider:**

```
for (i = 0; i < 100; i=i+1)
for (j = 0; j < 100; j=j+1) {
A[i][j] = 0.0;
for (k = 0; k < 100; k=k+1)
A[i][j] = A[i][j] + B[i][k] * D[k][j];
}
```

- **Must vectorize multiplication of rows of B with columns of D**
- **Use non-unit stride**
- **Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:**
  - #banks / LCM(stride, #banks) < bank busy time (in # of cycles)
- **Example:**

  8 memory banks with a bank busy time of 6 cycles and a total memory latency of 12 cycles.

  How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?
- **Answer:**
- **Stride of 1: number of banks is greater than the bank busy time, so it takes**

  12+64 = 76 clock cycles 1.2 cycle per element
- **Stride of 32: the worst case scenario happens when the stride value is a multiple of the number of banks, which this is!**

  Every access to memory will collide with the previous one! Thus, the total time will be:

  12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element!

Scatter-Gather

- **Consider sparse vectors A & C and vector indices K & M, and A and C have the same** number (n) of non-zeros:

  for (i = 0; i < n; i=i+1)

A[K[i]] = A[K[i]] + C[M[i]];

Ra, Rc, RkandRmthe starting addresses of vectors

- **Use index vector:**

LVVk, Rk;load K

LVIVa, (Ra+Vk);load A[K[]]

LVVm, Rm;load M

LVIVc, (Rc+Vm);load C[M[]]

ADDVV.DVa, Va, Vc;add them

SVI(Ra+Vk), Va;store A[K[]]

<u>Programming Vec. Architectures</u>

- **Compilers can provide feedback to programmers**
- **Programmers can provide hints to compiler**

<u>Summary of Vector Architecture</u>

- **Optimizations:**

➢ Multiple Lanes: > 1 element per clock cycle

➢ Vector Length Registers: Non-64 wide vectors

➢ Vector Mask Registers: IF statements in vector code

➢ Memory Banks: Memory system optimizations to support vector processors

➢ Stride: Multiple dimensional matrices

➢ Scatter-Gather: Sparse matrices

➢ Programming Vector Architectures: Program structures affecting performance

4. Explain in detail about Graphics Processing Units and its Programming. [R-2013]

Graphical Processing Units

Given the hardware invested to do graphics well, how can it be supplemented to improve performance of a wider range of applications?

Basic idea:

Heterogeneous execution model

CPU is the *host*, GPU is the *device*

Develop a C-like programming language for GPU

Compute Unified Device Architecture (CUDA)

OpenCLfor vendor-independent language

Unify all forms of GPU parallelism as *CUDA thread*

Programming model is “Single Instruction Multiple Thread” (SIMT)

Threads and Blocks

A thread is associated with each data element

CUDA threads, with thousands of which being utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP

Threads are organized into blocks

Thread Blocks: groups of up to 512 elements

Multithreaded SIMD Processor: hardware that executes a whole thread block (32 elements executed per thread at a time)

Blocks are organized into a grid

Blocks are executed independently and in any order

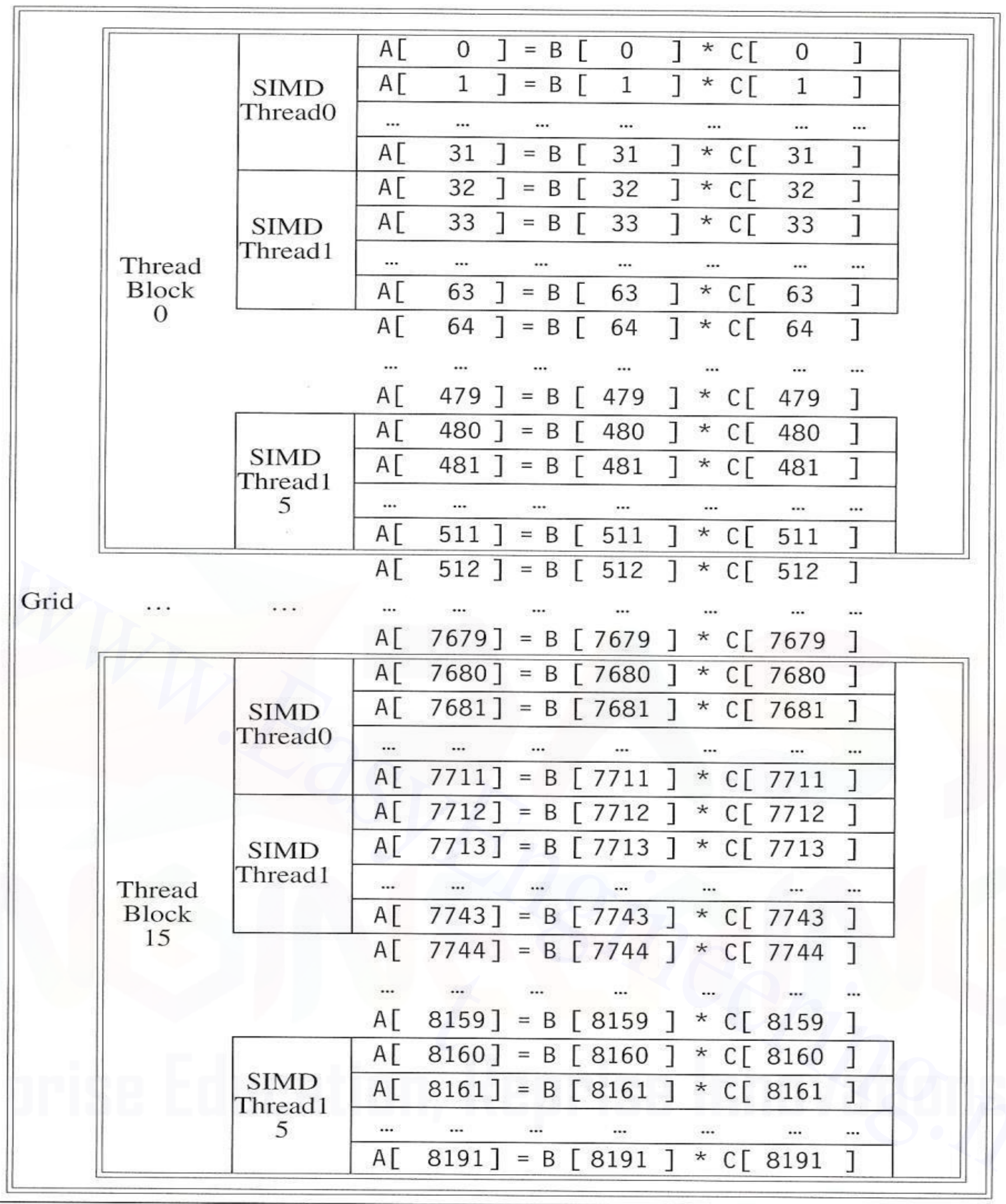
Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory

GPU hardware handles thread management, not applications or OS

A multiprocessor composed of multithreaded SIMD processors

A Thread Block Scheduler

Grid, Threads, and Blocks


Grid
Thread Block 0
SIMD Thread0
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
SIMD Thread1
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
SIMD Thread15
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
Thread Block 15
SIMD Thread0
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
SIMD Thread1
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
SIMD Thread15
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor

Uses multithreading to hide memory latency

Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

Multiply two vectors of length 8192

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes
    - 512 elements/block, 16 SIMD threads/block 32 ele/thread
- SIMD instruction executes 32 elements at a time
- Thus grid size = 16 blocks
- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
- Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

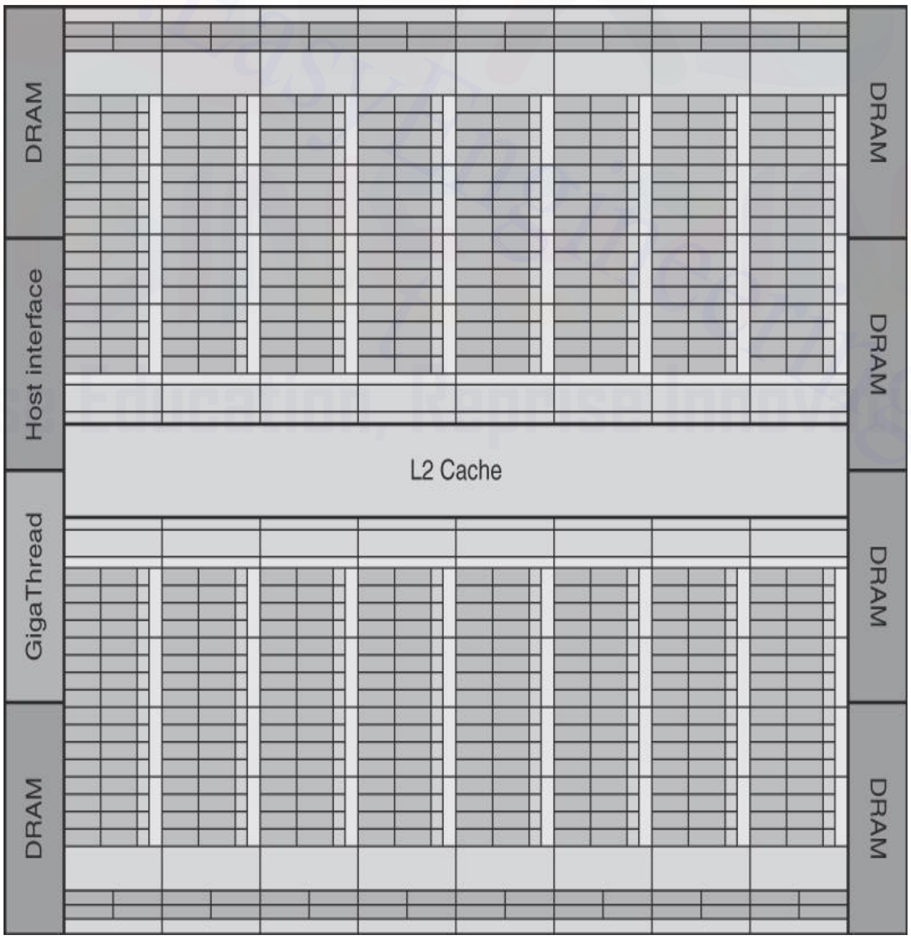


Fig. Floor plan of the Fermi GTX 480 GPU

This diagram shows 16 multithreaded SIMD Processors. The Thread Block

Scheduler is highlighted on the left. The GTX 480 has 6 GDDR5 ports, each 64 bits wide, supporting up to 6 GB of capacity. The Host Interface is PCI Express 2.0 x 16. Giga Thread is the name of the scheduler that distributes thread blocks to Multiprocessors, each of which has its own SIMD Thread Scheduler.

Terminology

Threads of SIMD instructions

- Each has its own PC
- Thread scheduler uses scoreboard to dispatch
- No data dependencies between threads!
- Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency

Thread block scheduler schedules blocks to SIMD processors

Within each SIMD processor:

- 32 SIMD lanes
- Wide and shallow compared to vector processors

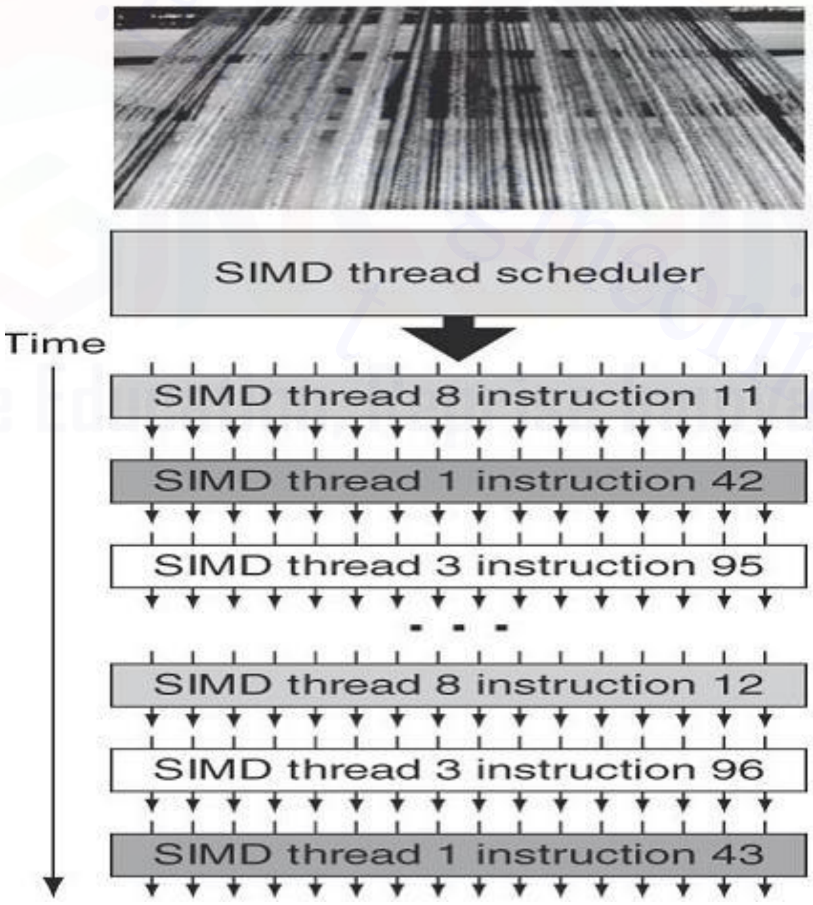

SIMD thread scheduler
Time
SIMD thread 8 instruction 11
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
- - -
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43

Fig .Scheduling of threads of SIMD instructions.

The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

Example

NVIDIA GPU has 32,768 registers

- Divided into lanes
- Each SIMD thread is limited to 64 registers
- SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
- Fermi has 16 physical SIMD lanes, each containing 2048 registers

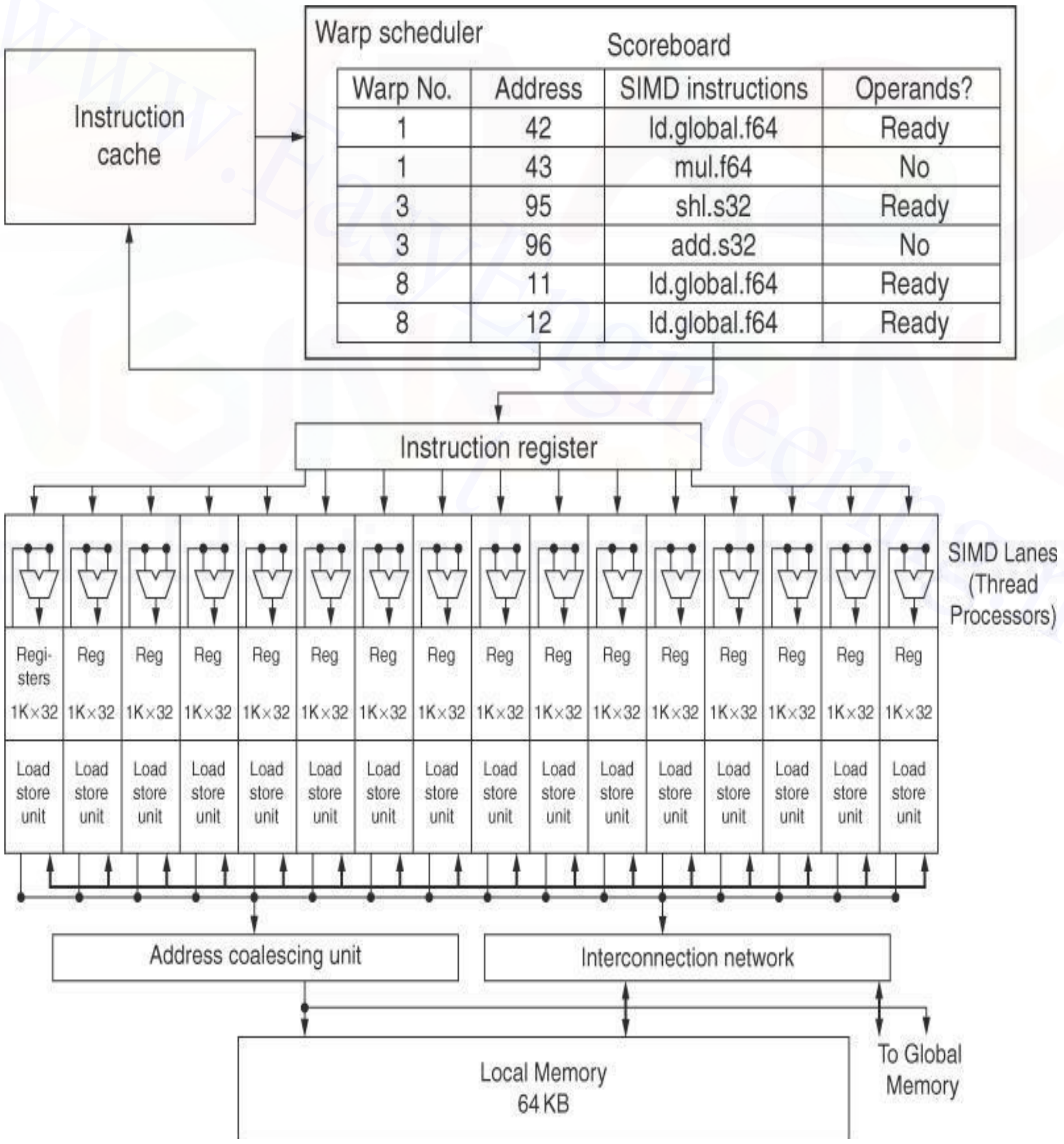


Fig. Simplified block diagram of a Multithreaded SIMD Processor.

It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

NVIDIA Instruction Set Arch.

ISA is an abstraction of the hardware instruction set

- "Parallel Thread Execution (PTX)"
- Uses virtual registers
- Translation to machine code is performed in software
- Example: one CUDA thread, 8192 of these created!

shl.s32R8, blockIdx, 9; Thread Block ID * Block size (512 or 29)

add.s32R8, R8, threadIdx; R8 = i= my CUDA thread ID

ld.global.f64RD0, [X+R8]; RD0 = X[i]

ld.global.f64RD2, [Y+R8]; RD2 = Y[i]

mul.f64 R0D, RD0, RD4; Product in RD0 = RD0 * RD4 (scalar a)

add.f64 R0D, RD0, RD2; Sum in RD0 = RD0 + RD2 (Y[i])

st.global.f64 [Y+R8], RD0; Y[i] = sum(X[i]*a + Y[i])

Conditional Branching

Like vector architectures, GPU branch hardware uses internal masks

Also uses

- Branch synchronization stack
  - Entries consist of masks for each SIMD lane
  - I.e. which threads commit their results (all threads execute)
- Instruction markers to manage when a branch diverges into multiple execution paths
  - Push on divergent branch
- …and when paths converge
  - Act as barriers
  - Pops stack

Per-thread-lane 1-bit predicate register, specified by programmer

Example

if (X[i] != 0)

X[i] = X[i] –Y[i];

else X[i] = Z[i];

ld.global.f64RD0, [X+R8]; RD0 = X[i]

setp.neq.s32P1, RD0, #0; P1 is predicate register 1

@!P1, braELSE1, *Push; Push old mask, set new mask bits*

; if P1 false, go to ELSE1

ld.global.f64RD2, [Y+R8]; RD2 = Y[i]

sub.f64RD0, RD0, RD2; Difference in RD0

st.global.f64[X+R8], RD0; X[i] = RD0

@P1, braENDIF1, **Comp; complement mask bits*

; if P1 true, go to ENDIF1

ELSE1:ld.global.f64 RD0, [Z+R8]; RD0 = Z[i]

st.global.f64 [X+R8], RD0; X[i] = RD0

ENDIF1: *<next instruction>, *Pop; pop to restore old mask*

NVIDIA GPU Memory Structures

Each SIMD Lane has private section of *off-chip* DRAM

- "Private memory", not shared by any other lanes
- Contains stack frame, spilling registers, and private variables
- Recent GPUs cache this in L1 and L2 caches

Each multithreaded SIMD processor also has local memory that is *on-chip*

- Shared by SIMD lanes / threads within a block only

The *off-chip* memory shared by SIMD processors is *GPU Memory*

Host can read and write GPU memory

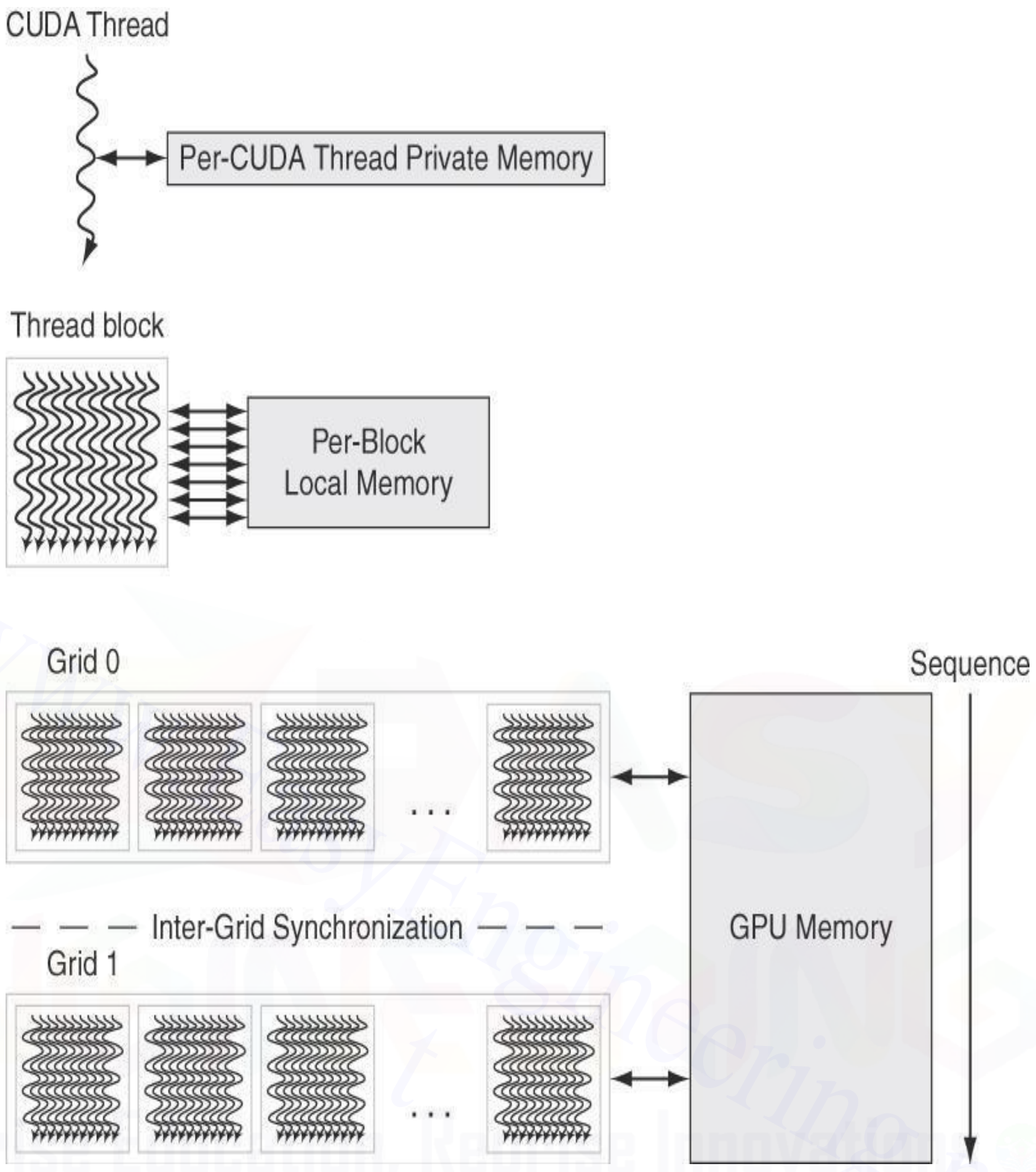


Fig.GPU Memory structures

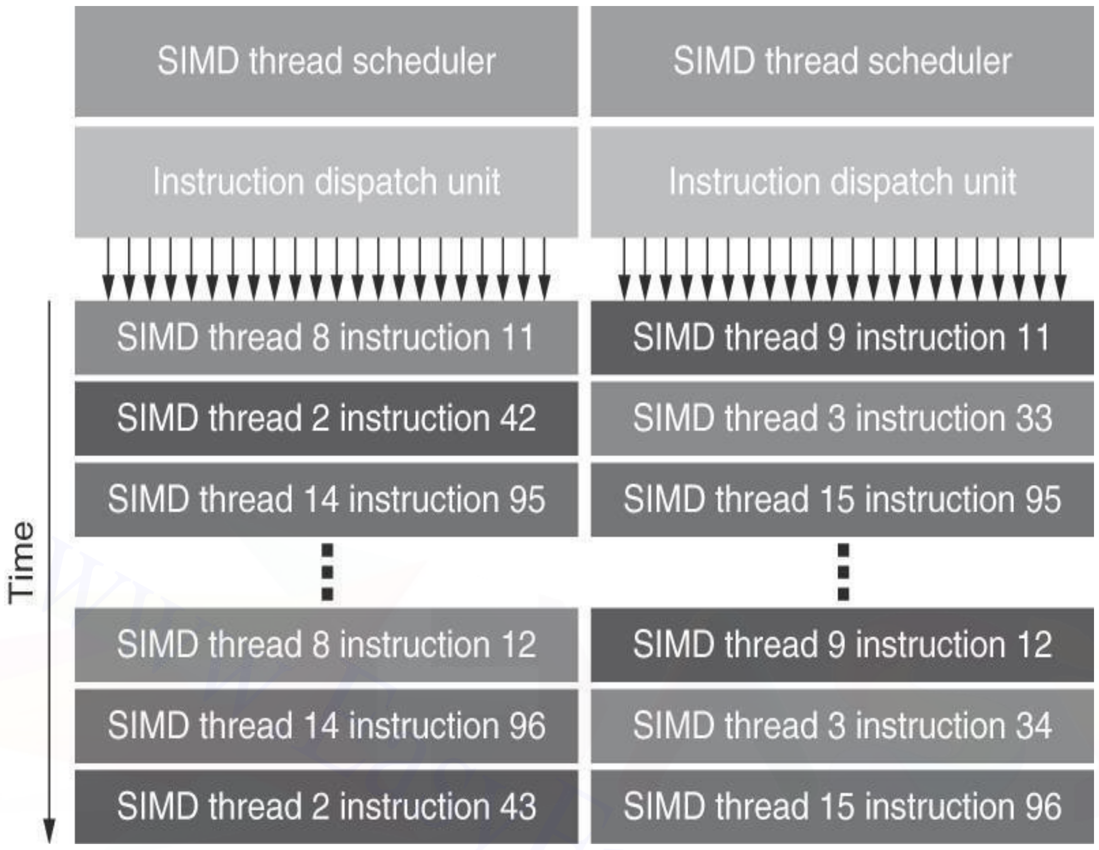
GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Fermi Architecture Innovations

Each SIMD processor has

- Two SIMD thread schedulers, two instruction dispatch units
- 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units

➢ Thus, two threads of SIMD instructions are scheduled every two clock cycles



SIMD thread scheduler
SIMD thread scheduler
Instruction dispatch unit
Instruction dispatch unit
SIMD thread 8 instruction 11
SIMD thread 9 instruction 11
SIMD thread 2 instruction 42
SIMD thread 3 instruction 33
SIMD thread 14 instruction 95
SIMD thread 15 instruction 95
SIMD thread 8 instruction 12
SIMD thread 9 instruction 12
SIMD thread 14 instruction 96
SIMD thread 3 instruction 34
SIMD thread 2 instruction 43
SIMD thread 15 instruction 96
Time

Fig. Block Diagram of Fermi's Dual SIMD Thread Scheduler.

Fast double precision: gen-78 515 GFLOPs for DAXPY

Caches for GPU memory: I/D L1/SIMD proc and shared L2

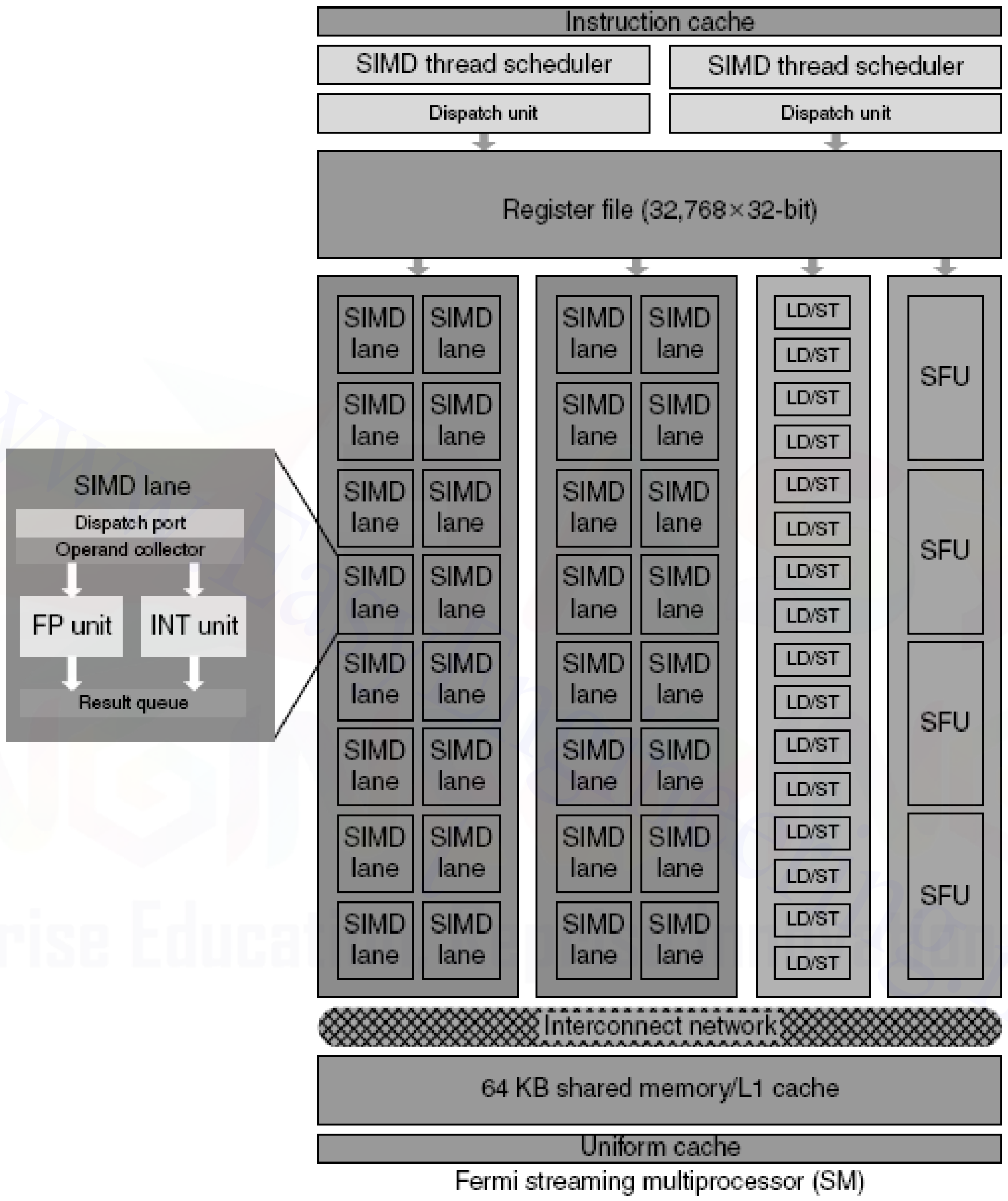
64-bit addressing and unified address space: C/C++ ptrs

Error correcting codes: dependability for long-running apps

Faster context switching: hardware support, 10X faster

Faster atomic instructions: 5-20X faster than gen-

Fermi Multithreaded SIMD Proc.



5. (i)**Consider the following code, which multiplies two vectors that contain single-precision complex values: [R-2013]**

For (i=0; i<300; i++) {

c_re[i] = a_re[i] * b_re[i] –a_im[i] * b_im[i];

c_im[i] = a_re[i] * b_im[i] –a_im[i] * b_re[i];

Assume that the processor runs at 700 MHz and has a maximum vector length of 64.

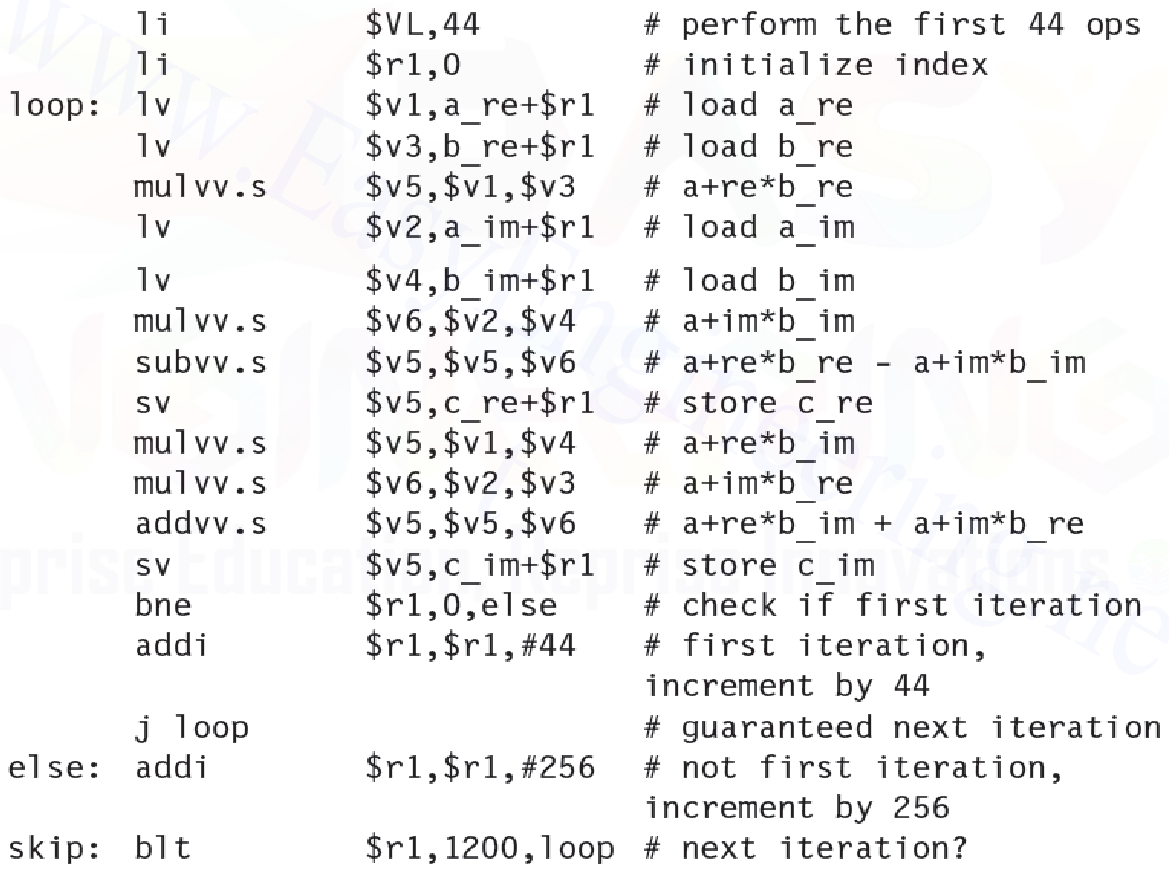
A. What is the arithmetic intensity of this kernel (i.e., the ratio of floating-point operations per byte of memory accessed)?

B. Convert this loop into VMIPS assembly code using strips mining.

C.Assuming chaining and a single memory pipeline, how many chimes are required?

A. This code reads four floats and writes two floats for every six FLOPs, so the arithmetic intensity = 6/6 = 1.

B. Assume MVL = 64□300 mod 64 = 44



```
        li          $VL,44          # perform the first 44 ops
        li          $r1,0           # initialize index
loop:   lv          $v1,a_re+$r1    # load a_re
        lv          $v3,b_re+$r1    # load b_re
        mulvv.s     $v5,$v1,$v3     # a+re*b_re
        lv          $v2,a_im+$r1    # load a_im
        lv          $v4,b_im+$r1    # load b_im
        mulvv.s     $v6,$v2,$v4     # a+im*b_im
        subvv.s     $v5,$v5,$v6     # a+re*b_re - a+im*b_im
        sv          $v5,c_re+$r1    # store c_re
        mulvv.s     $v5,$v1,$v4     # a+re*b_im
        mulvv.s     $v6,$v2,$v3     # a+im*b_re
        addvv.s     $v5,$v5,$v6     # a+re*b_im + a+im*b_re
        sv          $v5,c_im+$r1    # store c_im
        bne         $r1,0,else      # check if first iteration
        addi        $r1,$r1,#44     # first iteration,
                                    increment by 44
        j loop                      # guaranteed next iteration
else:   addi        $r1,$r1,#256    # not first iteration,
                                    increment by 256
skip:   blt         $r1,1200,loop   # next iteration?
```

C.Identify convoys:

1. mulvv.slv   # a_re* b_re

# (assume already loaded),

                          # loada_im

2. lvmulvv.s# load b_im, a_im* b_im

3. subvv.ssv# subtract and store c_re

4. mulvv.slv   # a_re* b_re,

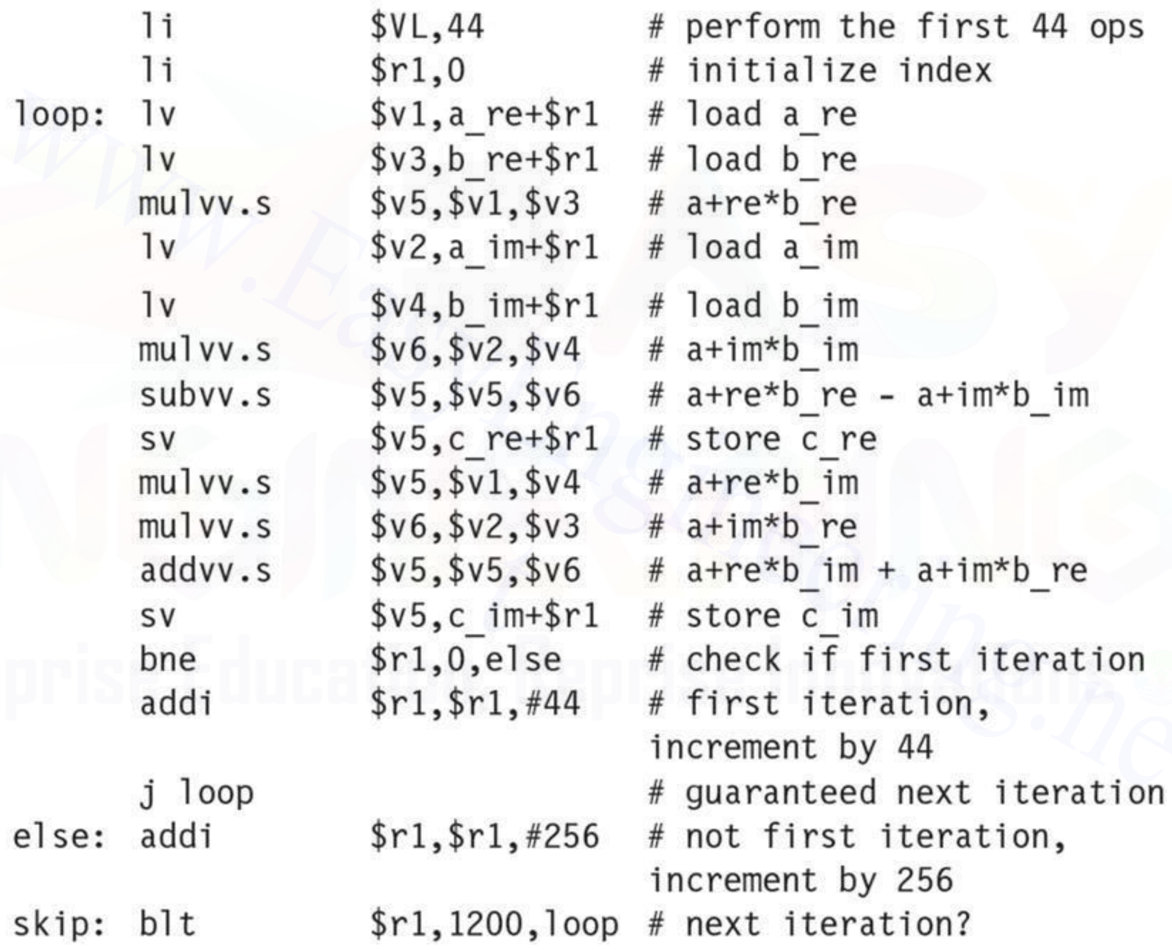
                          # load next a_revector

5. mulvv.slv   # a_im* b_re,

                           # load next b_revector

6. addvv.ssv# add and store c_im

6 chimes



```
      li        $VL,44         # perform the first 44 ops
      li        $r1,0          # initialize index
loop: lv        $v1,a_re+$r1   # load a_re
      lv        $v3,b_re+$r1   # load b_re
      mulvv.s   $v5,$v1,$v3    # a+re*b_re
      lv        $v2,a_im+$r1   # load a_im
      lv        $v4,b_im+$r1   # load b_im
      mulvv.s   $v6,$v2,$v4    # a+im*b_im
      subvv.s   $v5,$v5,$v6    # a+re*b_re - a+im*b_im
      sv        $v5,c_re+$r1   # store c_re
      mulvv.s   $v5,$v1,$v4    # a+re*b_im
      mulvv.s   $v6,$v2,$v3    # a+im*b_re
      addvv.s   $v5,$v5,$v6    # a+re*b_im + a+im*b_re
      sv        $v5,c_im+$r1   # store c_im
      bne       $r1,0,else     # check if first iteration
      addi      $r1,$r1,#44    # first iteration,
                               increment by 44
      j loop                   # guaranteed next iteration
else: addi      $r1,$r1,#256   # not first iteration,
                               increment by 256
skip: blt       $r1,1200,loop  # next iteration?
```

(ii)With examples, explain how do you detect and enhance Loop Level Parallelism?

[R-2013]

Loop-Level Parallelism

Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations

Loop-carried dependence

Example 1:

```
for (i=999; i>=0; i=i-1)
x[i] = x[i] + s;
```

No loop-carried dependence

Example 2:

```
for (i=0; i<100; i=i+1) {
A[i+1] = A[i] + C[i]; /* S1 */
B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

S1 and S2 use values computed by S1 in previous iteration

S2 uses value computed by S1 in same iteration

Example 3:

```
for (i=0; i<100; i=i+1) {
A[i] = A[i] + B[i]; /* S1 */
B[i+1] = C[i] + D[i]; /* S2 */
}
```

S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel

Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
B[i+1] = C[i] + D[i];
A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Example 4:

```
for (i=0;i<100;i=i+1) {
A[i] = B[i] + C[i];
D[i] = A[i] * E[i];
}
```

No loop-carried dependence

Example 5:

```
for (i=1;i<100;i=i+1) {
Y[i] = Y[i-1] + Y[i];
}
```

Loop-carried dependence in the form of *recurrence*

Finding dependences

Assume that a *1-D*array index *i*is*affine*:

*a*x *i*+ *b* (with constants *a* and *b*)

An index in an *n-D* array index is *affine* if it is affine in each dimension

Assume:

- Store to *a*x *i*+ *b*, then
- Load from *c*x *i*+ *d*
- *i*runs from *m*to*n*
- Dependence exists if:

Given j, ksuch that m≤ j≤ n, m≤ k≤ n

to ax j+ b, load from ax k+ d, and ax j+ b= cx k+ d

Finding dependences

Generally cannot determine at compile time

Test for absence of dependence:

GCD test:

If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)

Example 1:

```
for (i=0; i<100; i=i+1) {
X[2*i+3] = X[2*i] * 5.0;
}
```

Answer: a=2, b=3, c=2, d=0 GCD(c,a)=2, d-b=-3 no dependence possible.

Example 2:

```
for (i=0; 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 */
```

}

Watch for antidependences and output dependencies:

RAW: S1S3, S1S4 on Y[i], not loop-carried

WAR: S1S2 on X[i]; S3S4 on Y[i]

WAW: S1S4 on Y[i]

Reductions

Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
sum = sum + x[i] * y[i];
```

Transform to…

```
for (i=9999; i>=0; i=i-1)
sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
finalsum= finalsum+ sum[i];
```

Do on p processors:

```
for (i=999; i>=0; i=i-1)
finalsum[p] = finalsum[p] + sum[i+1000*p];
```

Note: assumes associativity!

UNIT IV
THREAD LEVEL PARALLELISM
PART A

1. Define sequential consistency memory model(NOV/DEC 2016)

The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by
that access are completed.

2. What is loop unrolling? What are the major limitations of loop unrolling? [Nov/Dec 2012]

To control the various dependencies the loop is unrolled as many times as possible.

Limitations

- Increased program code size, which can be undesirable, particularly for embedded applications.
- The code is less readable.
- Increased register usage in a single iteration to store temporary variables
- Unrolled loops that contain branches are even slower than recursions

3. What is the disadvantage of having a distributed memory? [R-2013]

The key disadvantages for distributed memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories.

4. What is the importance of memory consistency model? [Nov/Dec 2013]

Consistency model defines correct behavior

Coherence protocol is only a means to an

Consistency model restricts ordering of loads/stores

5. Why do we need synchronization? [May/June 2014]

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases.

5. When is a memory said to be coherent in a multi-processor system? [R-2013]

In multi- processor system, two or more processing elements work concurrently. A memory system is coherent if any read of a data item returns the most recently written value of that data item when two processors access the same memory location.

7. Write a note on multiprocessor cache coherence problem? [Nov/Dec 2012][April/May 2014][May/June 2012](Apr/may 2017)

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. Two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem.*

8. What are the advantages of CMP architecture? ? [Nov/Dec 2012]

- CMP is easier to implement.
- A combination of CMP with SMT is superior.
- A multicore design takes several processor cores and packages them as a single processor.
- The goal is to enable system to run more tasks simultaneously and thereby achieve greater overall system performance.

9. Enlist the features of SMT Architecture.[April/May 2015]

- Fully exploit thread-level parallelism and instruction-level parallelism.

- Better Performance
  - Mix of independent programs
  - Programs that are parallelizable
  - Single threaded program

10. Why are design issues of SMT and CMP architecture are important? [May/June 2014]

To improve parallelism, throughput-and application performance design issues of SMT and CMP architecture are important

11. What is the disadvantage of having a distributed memory? [R-2013]

The key disadvantages for distributed memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories.

12. What is fine grained multithreading? [May/June 2013]

Fine-grained multithreading switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.

13.What are the advantages of SMT? [R-2013]

- SMT also increases hardware design flexibility.
- Simultaneous multithreading increases the complexity of instruction scheduling

14. What are the design challenges in SMT? [May/June 2012]

- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in Instruction issue - more candidate instructions need to be considered

15. What is multithreading? What is SMT?[Nov/Dec 2014]

Threads to share the functional units of one processor via overlapping .Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit TLP at the same time it exploits ILP.

16. List the methods for providing synchronization in threads?

(NOV/DEC 2016)

The method for providing synchronization in threads are spinlock, data coherency, semaphores and readers-writers lock.

17. Define sequential consistency ?(NOV/DEC 2016)

The most straightforward model for memory consistency is called sequential consistency. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved.

PART B

1. Explain the symmetric shared memory architecture. (or) Explain the UMA architecture. (or) Explain the snoopy based Cache coherence protocols with neat diagram. [Apr/ May 2015], [May/Jun 2014] [Nov/ Dec 2014], [Nov/ Dec 2013]

Symmetric Shared Memory Architectures:

The Symmetric Shared Memory Architecture consists of several processors with a single physical memory shared by all processors through a shared bus which is shown below.

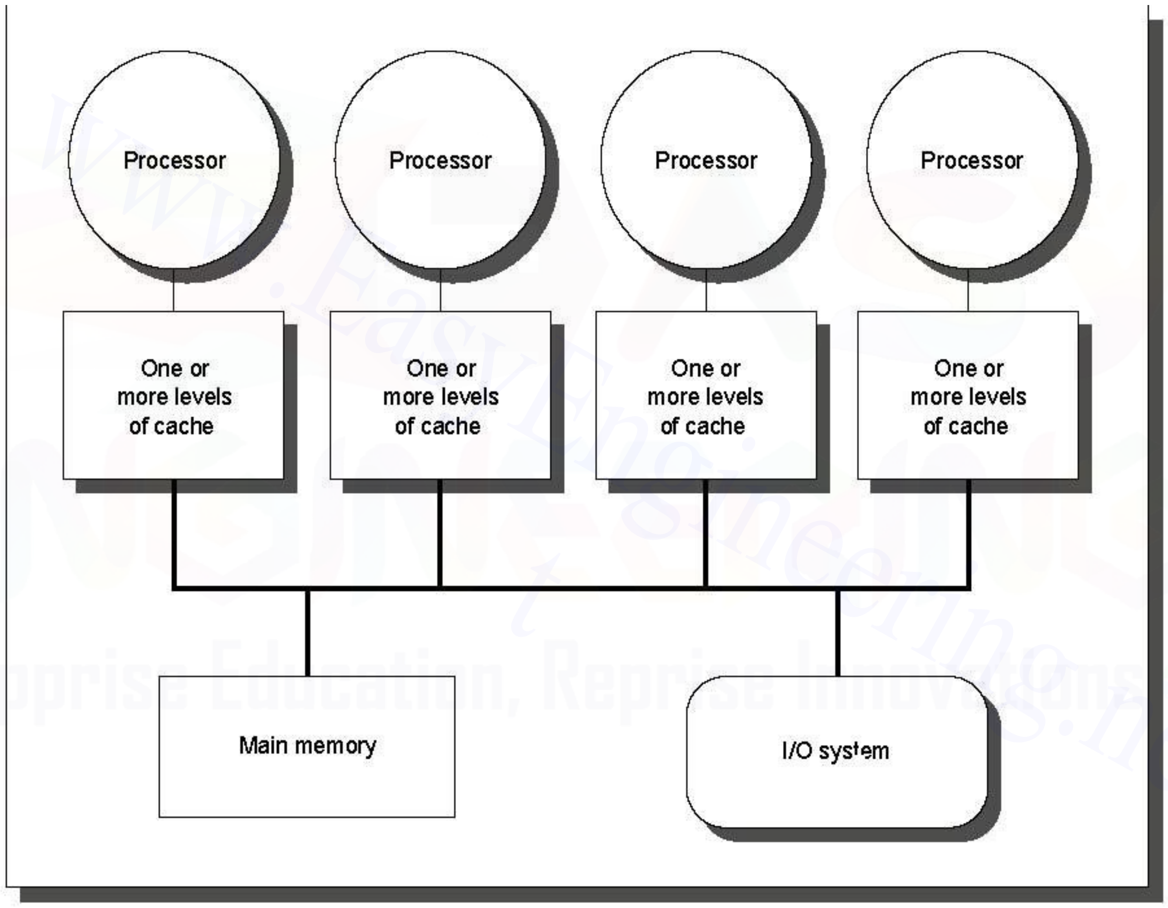

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

FIGURE Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus. In larger designs, multiple buses, or even a switch may be used, but the key architectural property: uniform access time o all memory from all processors remains.

Small-scale shared-memory machines usually support the caching of both shared and private data. *Private data* is used by a single processor, while *shared data* is used by multiple processors; essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required.

Cache Coherence in Multiprocessors:

Introduction of caches caused a coherence problem for I/O operations; the same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches. The table illustrates **the problem and shows how two different processors can have two different values for the same location. This difficulty s generally referred to as the *cache-coherence* problem.**

Table. The cache-coherence problem for a single memory location (X), read and written by two processors (A and B).

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

The first aspect, called *coherence,* defines what values can be returned by a read. The second aspect, called *consistency,* determines when a written value will be returned by a read.

Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

Basic Schemes for Enforcing Coherence

The protocols to maintain coherence for multiple processors are called *cache-coherence protocols*. There are two classes of protocols, which use different techniques to track the sharing status, in use:

Directory based—the sharing status of a block of physical memory is kept in just one location, called the *directory;* we focus on this approach in section 6.5, when we discuss scalable shared-memory architecture.

[IF THE QUESTION ASKED LIKE: EXPLAIN SNOOPING BASED PROTOCOL, START WRITE FROM HERE]

Snooping—every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of a block that is requested on the bus.

Snooping Protocols

The methods which ensure that a processor has exclusive access to a data item before it write that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

WRITE INVALIDATE:

The following table shows an example of an invalidation protocol for a snooping bus with write-back caches in action To see how this protocol ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name).

Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization.

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

		Contents of	**Contents of**	**Contents of memory**
Processor activity	**Bus activity**	**CPU A's cache**	**CPU B's cache**	**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

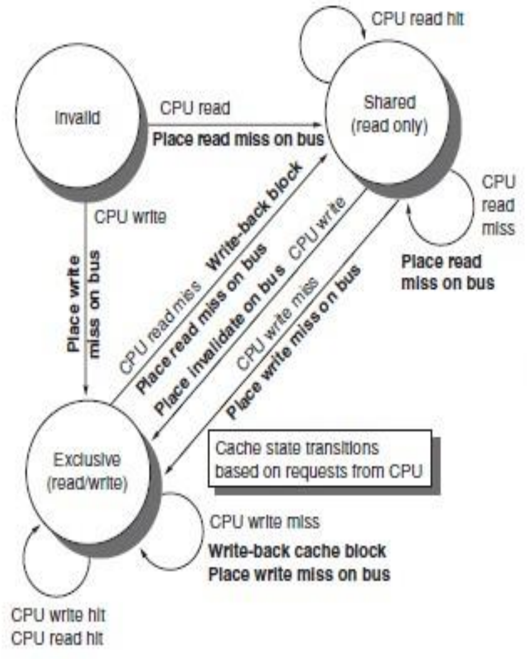
WRITE UPDATE PROTOCOL:

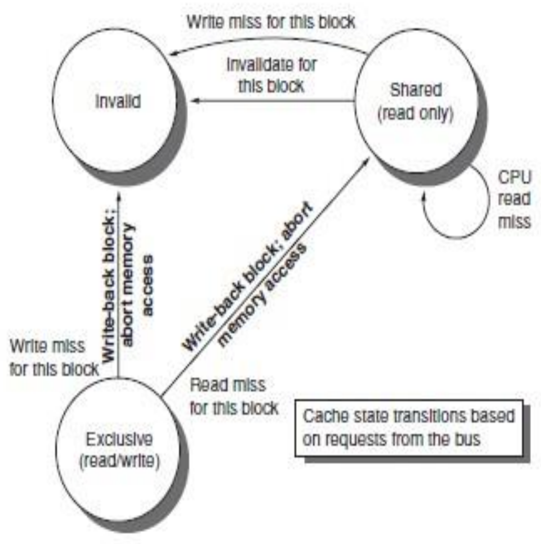
The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *writes broadcast* protocol. Table shows an example of a write update protocol in operation. In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs.

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

		Contents of	**Contents of**	**Contents of memory**
Processor activity	**Bus activity**	**CPU A's cache**	**CPU B's cache**	**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

State transition diagram:




Write miss for this block
Invalidate for this block
Invalid
Shared (read only)
CPU read miss
Write-back block; abort memory access
Write-back block; abort memory access
Write miss for this block
Read miss for this block
Cache state transitions based on requests from the bus
Exclusive (read/write)

Basic Implementation Techniques

The serialization of access enforced by the bus also forces serialization of writes, since when two processors compete to write to the same location, one must obtain bus access before the other. The first processor to obtain bus access will cause the other processor's copy to be invalidated, causing writes to be strictly serialized. One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access.

For a write-back cache, however, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for caches misses and for writes: Each processor snoops every address placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. Since write-back caches generate lower requirements for memory bandwidth, they are greatly preferable in a multiprocessor, despite the slight increase in complexity. Therefore, we focus on implementation with write-back caches.

2. Explain the concept of distributed shared memory and also explain directory based cache coherence protocols with an example. (or) Explain the

NUMA architecture with neat diagram. [Nov/ Dec 2014] [Apr/ May 2015], [May/Jun 2014]

There are several disadvantages in Symmetric Shared Memory architectures.

- First, compiler mechanisms for transparent software cache coherence are very limited.
- Second, without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word.
- Third, mechanisms for tolerating latency such as pre-fetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; These disadvantages are magnified by the large latency of access to remote memory versus a local cache.
- For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors. For larger-scale architectures, there are new challenges to extending the cache-coherent shared-memory model. Although the bus can certainly be replaced with a more scalable interconnection network and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed is known as Distributed Shared Memory architecture.

The first coherence protocol is known as a directory protocol. A *directory* keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on.

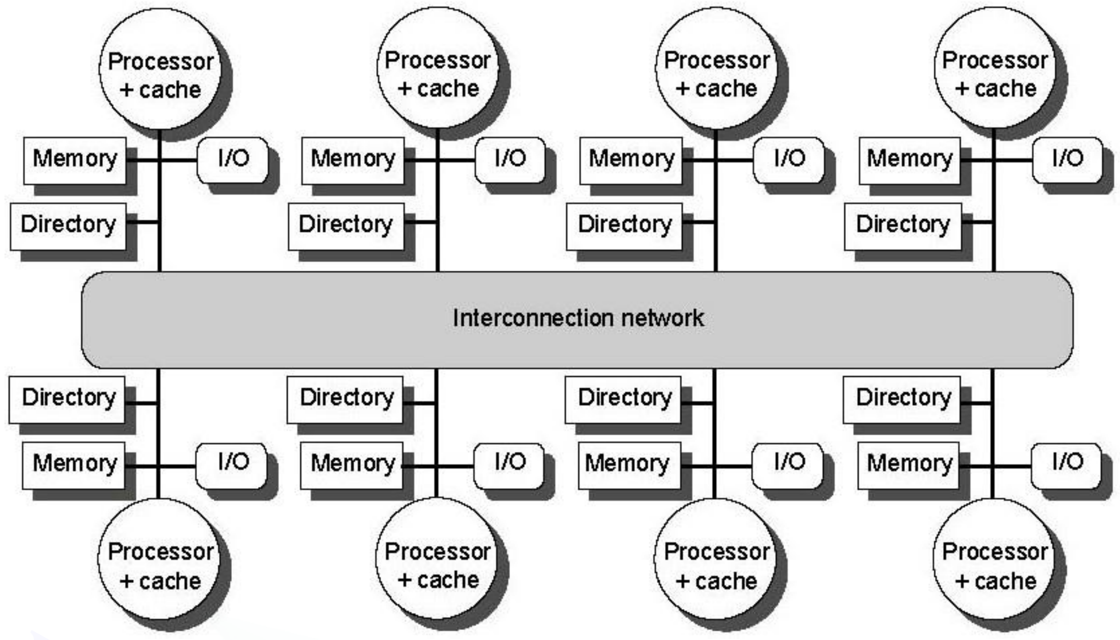


Figure: A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor.

Directory-Based Cache-Coherence Protocols: The Basics

- There are two primary operations that a directory protocol must implement:
  - Handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.)
  - To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

Shared—one 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

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.

A catalog of the message types that may be sent between the processors and the directories. Figure shows the type of messages sent among nodes.

- The *local* node is the node where a request originates.
- The *home* node is the node where the memory location and the directory

entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a remote node.

- A *remote* node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

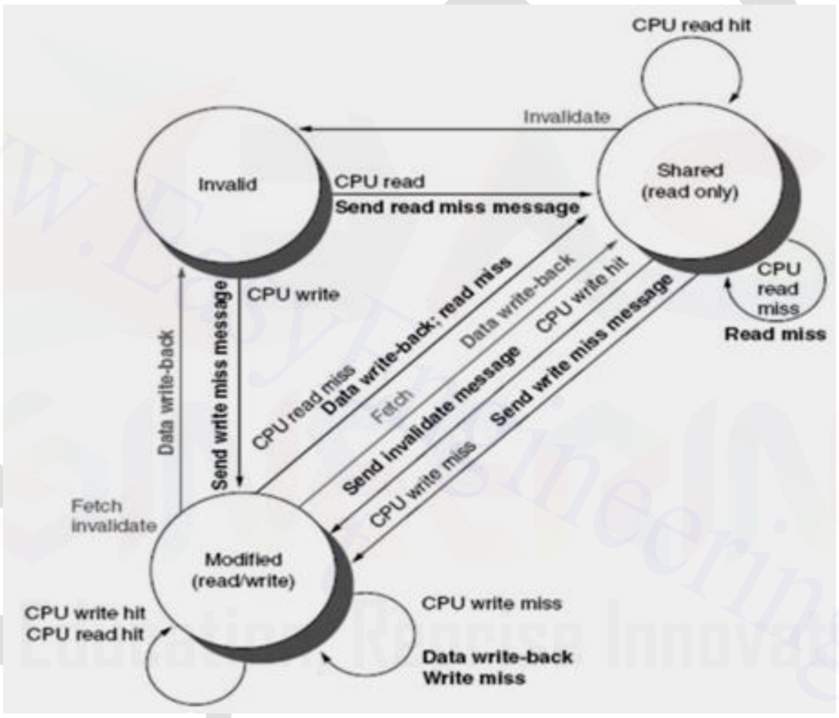
Message type	**Source**	**Destination**	**Message contents**	**Function of this message**
Read miss	Local cache	Home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Processor P has a write miss at address A; — 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.

(where P=requesting processor number), A=requested address, and D=data contents).

Example :

State transition diagram for an individual cache block in a directory-based system.



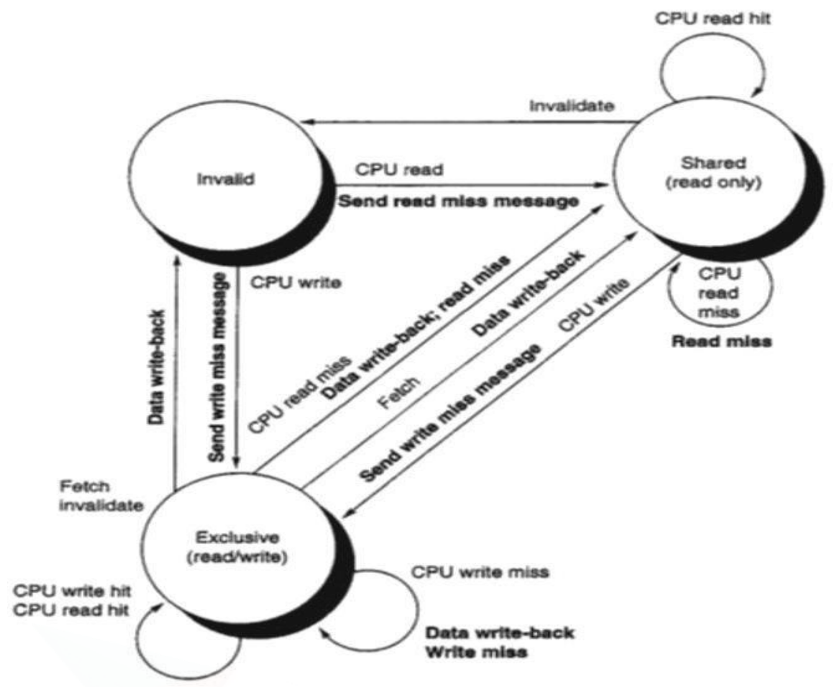


Figure: State transition diagram for an individual cache block in a directory-based system

To understand these directory operations, let’s examine the requests received and actions taken state by state. When a block is in the uncached state, the copy in memory is the current value, so the only possible requests for that block are

_ *Read miss*—The requesting processor is sent the requested data from memory, and the requestor is made the only sharing node. The state of the block is made shared.

_ *Write miss*—The requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicate the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

_ *Read miss*—The requesting processor is sent the requested data from memory, and the requesting processor is added to the sharing set.

_ *Write miss*—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state, the current value of the block is held in the cache of the processor identified by the set Sharers (the owner), so there are three possible directory requests:

_ *Read miss*—The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).

_ *Data write back*—The owner processor is replacing the block and therefore must write it back. This write back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.

_ *Write miss*—The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

3. Discuss about synchronization process used in multiprocessor system. [May/Jun 2013], [Nov/ Dec 2013]

Synchronization

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. The efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases.

There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers.

One typical operation for building synchronization operations is the *atomic exchan*ge, which interchanges a value in a register for a value in memory. Use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that can tell whether or not the two operations executed atomically. One operation, present in many older multiprocessors, is *test-and-set,* which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange.

Another atomic synchronization primitive is *fetch-and-increment:* it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment.

Implementing Locks Using Coherence

We can use the coherence mechanisms of a multiprocessor to implement *spin locks:* locks that a processor continuously tries to acquire, spinning around a loop until it succeeds. Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available. Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
                DADDUI   R2,R0,#1
        lockit:  EXCH                R2,0(R1)        ; atomic exchange
                BNEZ                 R2,lockit        ; already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

Synchronization Performance Challenges

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier.

Synchronization Mechanisms for Larger-Scale Multiprocessors

Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails. Figure 6.41 shows how a spin lock with *exponential back-off* is implemented.

Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses. This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
ADDUI           R3,R0,#1    ;R3 = initial delay
lockit:  LL     R2,0(R1)    ;load linked
         BNEZ   R2,lockit   ;not available-spin
         DADDUI R2,R2,#1    ;get locked value
         SC     R2,0(R1)    ;store conditional
         BNEZ   R2,gotit    ;branch if store succeeds
         DSLL   R3,R3,#1    ;increase delay by factor of 2
         PAUSE  R3          ;delays by value in R3
         J      lockit
gotit:   use data protected by lock
```

Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits Before we
look at hardware primitives,

Hardware Primitives

In this section we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases we can create hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 20 lock/unlock sequences.

Improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, or in software using an array to keep track of the waiting processes.

4. Explain in detail about the needs and types of memory consistency models. [Apr/ May 2015], [Nov/ Dec 2014], [May/Jun 2014], [Nov/ Dec 2013], [May/Jun 2013]

- Cache coherence ensures that multiple processors see a consistent view of memory.
- It does not answer the question of how consistent the view of memory must be. By "how consistent" we mean, when must a processor see a value that has been updated by another processor? Since processors communicate through shared variables (used both for data values and for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor? Since the only way to "observe the writes of another processor" is through reads, the question becomes, what properties must be enforced among reads and writes to different locations by different processors?
- Although the question of how consistent memory must be seems simple, it is remarkably complicated, as we can see with a simple example.

Here are two code segments from processes P1 and P2, shown side by side:

```
P1: A = 0;              P2: B = 0;
.....                   .....
A = 1;                  B = 1;
L1: if (B == 0) ...     L2: if (A == 0)...
```

- Assume that the processes are running on different processors, and that locations A and B are originally cached by both processors with the initial value of 0.
- If writes always take immediate effect and are immediately seen by other processors, it will be impossible for *both* if statements (labelled L1 and L2) to evaluate their conditions as true, since reaching the if statement means that either A or B must have been assigned the value 1. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay; then it is possible that both P1 and P2 have not seen the invalidations for B and A (respectively) *before* they attempt to read the values. The question is, Should this behaviour be allowed, and if so, under what conditions?

SEQUENTIAL CONSISTENCY:

- The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved.
- Sequential consistency eliminates the possibility of some nonobvious execution in the previous example because the assignments must be completed before the if statements are initiated.
- The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally effective to delay the next memory access until the previous one is completed.
- Remember that memory consistency involves operations among different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of A or B (A == 0 or B == 0) until the previous write has completed (B = 1 or A = 1). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read.

Relaxed Consistency Models:

- The key idea in relaxed consistency models is to 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.

There are a variety of relaxed models that are classified according to what read and write orderings they relax. We specify the orderings by a set of rules of the form X→Y, meaning that operation X must complete before operation Y is done. Sequential consistency requires maintaining all four possible orderings:

R→W,

R→R,

W→R,

and W→W.

The relaxed models are defined by which of these four sets of orderings they relax:

1. Relaxing the W→R ordering yields a model known as *total store ordering* or *processor consistency*. Because this ordering retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.

2. Relaxing the W→W ordering yields a model known as *partial store order*.

3. Relaxing the R→W and R→R orderings yields a variety of models including *weak ordering*, the PowerPC consistency model, and *release consistency*, depending on the details of the ordering restrictions and how synchronization

operations enforce ordering.

By relaxing these orderings, the processor can possibly obtain significant performance advantages. There are, however, many complexities in describing relaxed consistency models, including the advantages and complexities of relaxing different orders, defining precisely what it means for a write to complete, and deciding when processors can see values that the processor itself has written.

5. Explain in detail about CMP and SMT architecture and its performance.

[Nov/ Dec 2013], [May/Jun 2013]

CMP architecture (MULTICORE ARCHITECTURE or CHIP MULTIPROCESSORS)

Single-core computer

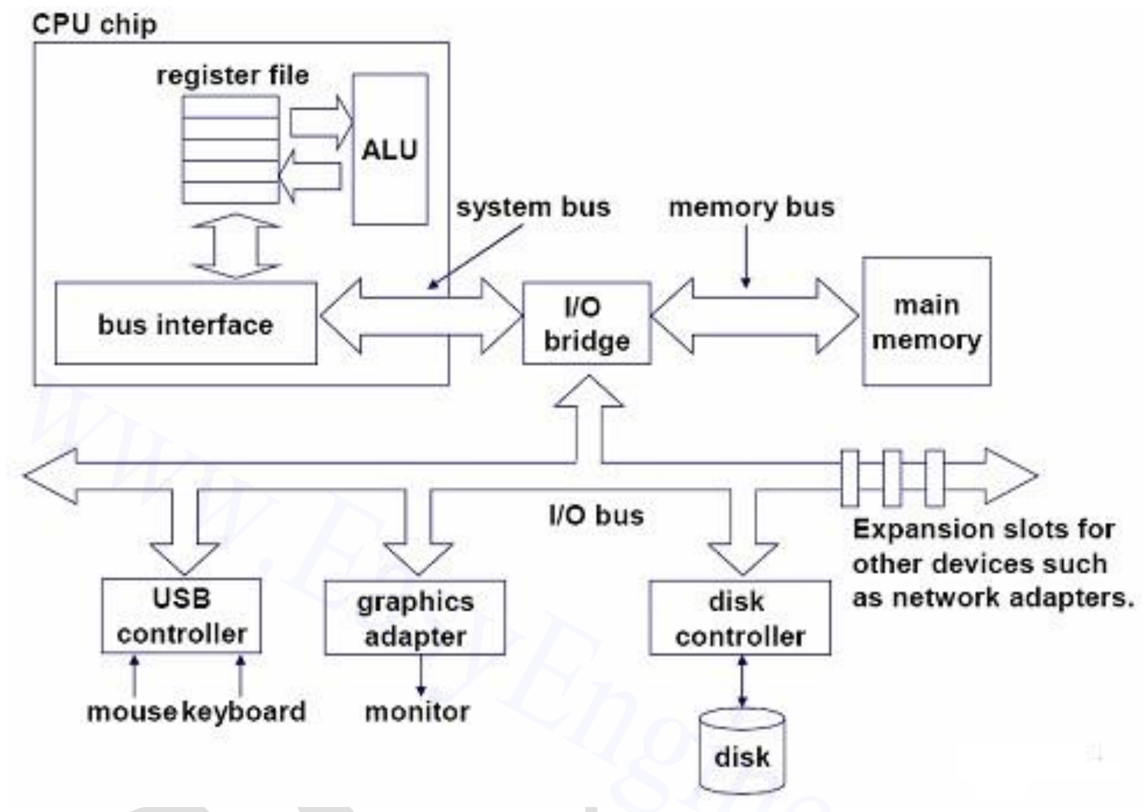


Figure: Single core computer architecture

CMP ARCHITECTURE:

Chip level multiprocessing (CMP or Multicore):

- Integrates two or more independent cores into a single package composed of a single integrated circuit, called a die, or more dies packaged, each executing threads independently.
- Every functional units of a processor is duplicated.
- Multiple processors, each with a full set of architectural resources, result on the same die.
- Processor may share an on-chip cache or each can have its own cache.

Example: IBM Power4, HP Mako

Challenges:

Power, Die area, Cost

CHIP MULTITHREADING:

- Chip multithreading is defined as the combination of chip multiprocessing and the hardware multithreading.
- Chip multithreading is the capability of a processor to process multiple software threads and simultaneous Hardware threads of execution.
- CMP is achieved by multiple cores on a single chip or multiple threads on a single core.
- CMP processors are especially suited to server workloads, which generally have high levels of Thread Level Parallelism (TLP).
- CMT Processors support many hardware strands through efficient sharing of on chip resources such as pipelines, caches and predictors.
- CMT Processors are a good match for server workloads, which have high levels of TLP and relatively low levels of ILP.

Multi-core architectures

Replicate multiple processor cores on a single die

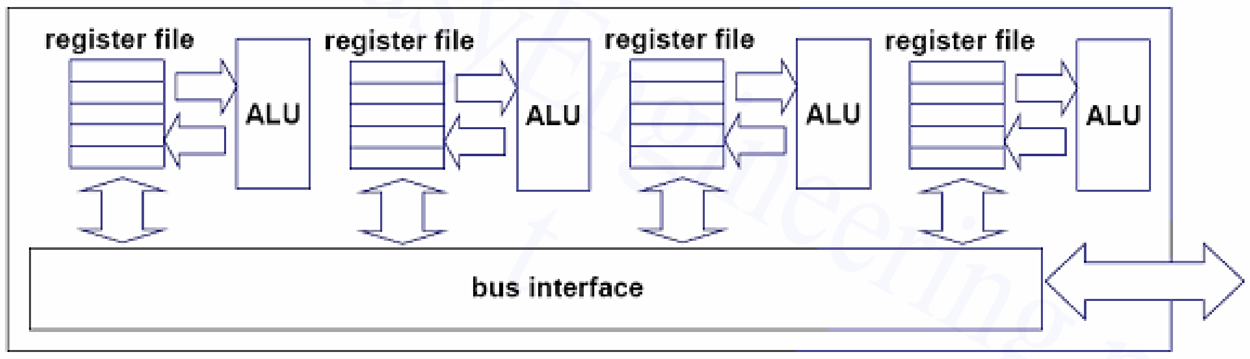


Fig: Multicore architecture showing multiple register and ALU

Why Multicore?

1. Difficult to make single-core clock frequencies even higher
2. Deeply pipelined circuits:
   a. heat problems
   b. speed of light problems
   c. difficult design and verification
   d. large design teams necessary
   e. server farms need expensive air-conditioning
3. Many new applications are multithreaded
4. General trend in computer architecture (shift towards more parallelism)

What applications benefit from multicore?

1. Database servers
2. Web servers (Web commerce)
3. Compilers
4. Multimedia applications
5. Scientific applications, CAD/CAM
6. In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)

SMT architecture (Simultaneous Multi-Threading concept for converting thread level parallelism into instruction level parallelism).

Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available.

In the SMT case, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads.

As mentioned earlier, simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the integrated exploitation of TLP through multithreading.
In particular, dynamically scheduled superscalar have a large set of virtual registers that can be used to hold the register sets of independent threads (assuming separate renaming tables are kept for each thread). Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads.

Design Challenges in SMT

Because a dynamically scheduled superscalar processor is likely to have a deep pipeline, SMT will be unlikely to gain much in performance if it were coarse grained. Since SMT makes sense only in a fine-grained implementation, we must worry about the impact of fine-grained scheduling on single-thread performance.

This effect can be minimized by having a preferred thread, which still permits multithreading to preserve some of its performance advantage with a smaller compromise in single-thread performance.

Unfortunately, mixing many threads will inevitably compromise the execution time of individual threads. Similar problems exist in instruction fetch. To maximize single-thread performance, we should fetch as far ahead as possible in that single thread and always have the fetch unit free when a branch is miss-predicted and a miss occurs in the prefetch buffer. Unfortunately, this limits the number of instructions available for scheduling from other threads, reducing throughput.

There are a variety of other design challenges for an SMT processor, including the following:

_ Dealing with a larger register file needed to hold multiple contexts
_ Not affecting the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging
_ Ensuring that the cache and TLB conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation In viewing these problems, two observations are important. First, in many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current superscalar is low enough that there is room for significant improvement, even at the cost of some overhead.

The IBM Power5 used the same pipeline as the Power4, but it added SMT support. In adding SMT, the designers found that they had to increase a number of structures in the processor so as to minimize the negative performance consequences from fine-grained thread interaction. These changes included the following:

Increasing the associativity of the L1 instruction cache and the instruction address translation buffers
_ Adding per-thread load and store queues
_ Increasing the size of the L2 and L3 caches
_ Adding separate instruction prefetch and buffering
_ Increasing the number of virtual registers from 152 to 240
_ Increasing the size of several issue queues

Because SMT exploits thread-level parallelism on a multiple-issue superscalar, it is most likely to be included in high-end processors targeted at server markets.

In addition, it is likely that there will be some mode to restrict the multithreading, so as to maximize the performance of a single thread.

SMT PERFORMANCE:

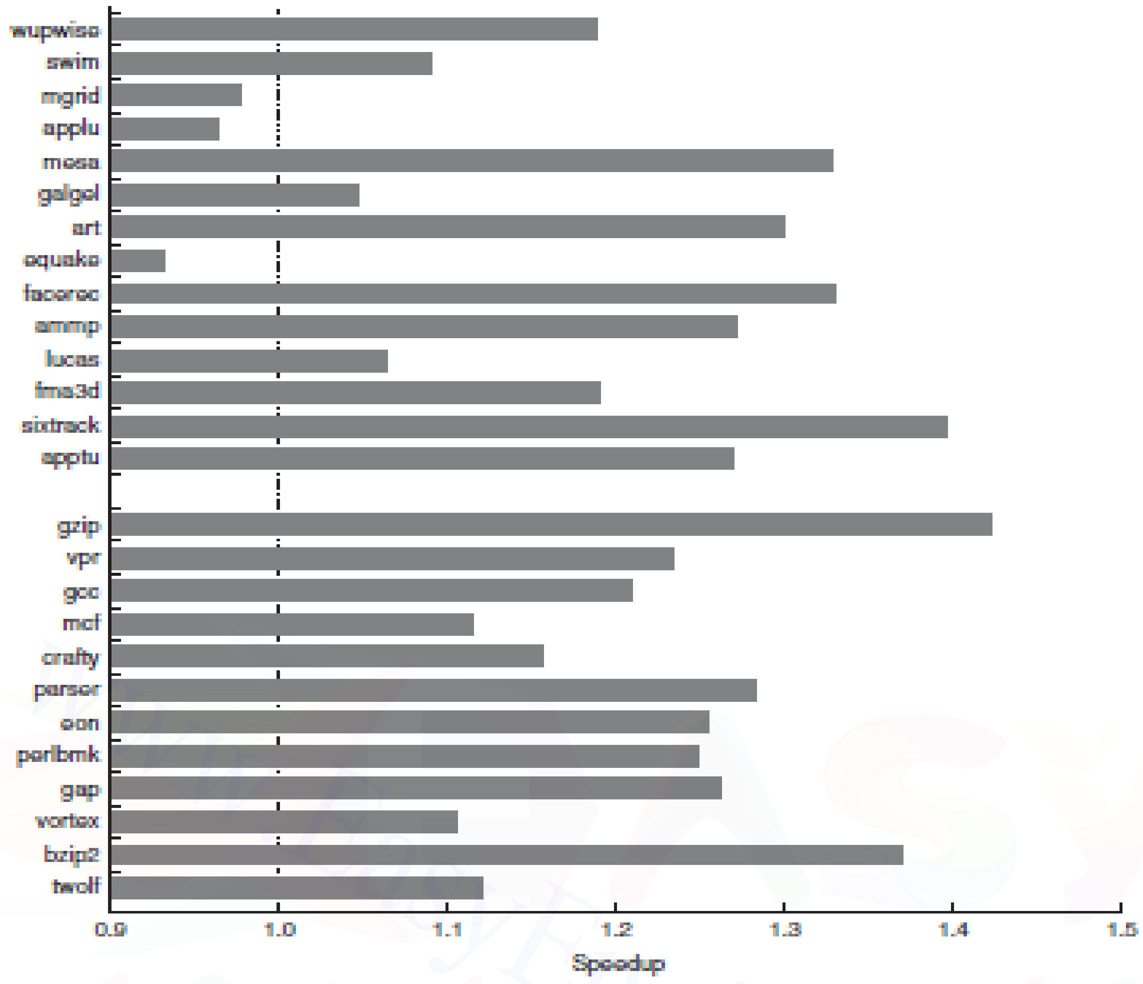


Fig: A comparison of SMT and single-thread (ST) performance on the 8-processor IBM eServer p5575

UNIT V

MEMORY ANDI/O

PART A

1. What is temporal locality and spatial locality? [May/June 2014](Apr/may 2017)

Temporal locality

It’s an accessed item has a high probability being accessed in the near future

Spatial locality

These are items close in space to a recently accessed item have a high probability of being accessed next

2. What are the basic cache optimizations? (NOV/DEC 2016)

Reduces miss rate

Larger block size

Bigger cache

Higher associativity

Reduces conflict rate

Reduce miss penalty

3. What are the advanced cache optimizations? [R-2013]

• Reducing hit time

• Increasing cache bandwidth

• Reducing Miss Penalty

• Reducing Miss Rate

• Reducing miss penalty or miss rate via parallelism

4. What is non-blocking cache? [R-2013]

Non-blocking cache or lockup-free cache

– allow data cache to continue to supply cache hits during a miss

– requires out-of-order execution CPU

5. What is hit under miss and hit under multiple miss? [May/June 2013]

“Hit under miss” reduces the effective miss penalty by continuing during a miss.

“Hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses

6. How to calculate Average memory access time2-way and Average memory access time4-way? [R-2013]

Average memory access time2-way = Hit time + Miss rate × Miss Penalty

Average memory access time4-way = Hit time × 1.1 + Miss rate × Miss Penalty

7. What is way prediction? [R-2013]

In *way prediction,* extra bits are kept in the cache to predict the way, or block within the set of the *next* cache access. This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data.

8. What is sequential interleaving? [R-2013]

A simple mapping that works well is to spread the addresses of the block sequentially across the banks, called *sequential interleaving*.

9 .What are the two basic strategies in the seventh optimization? [R-2013]

Critical word first—Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block. *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the processor and let the processor continue execution.

10. What is a victim buffer and victim cache? [R-2013]

In a write-back cache, the block that is replaced is sometimes called the *victim*. Write buffer is called as a *victim buffer*. The write victim buffer or victim buffer contains the dirty blocks that are discarded from a cache because of a miss. Rather than stall on a subsequent cache miss, the contents of the buffer are checked on a miss to see if they have the desired data before going to the next lower-level memory. This is a *victim cache*.

11. What is access time and cycle time?[May/June 2014]

Memory latency is traditionally quoted using two measures *Access time* is the time between when a read is requested and when the desired word arrives.
Cycle time is the minimum time between requests.

12. What does RAID stands for and what is JBOD? What is mirroring? [R-2013]

RAID stands for redundant array of inexpensive disks. RAID0has no redundancy and is sometimes nicknamed JBOD, for “just a bunch of disks,” although the data may be striped across the disks in the array.

RAID 1—Also called *mirroring* or *shadowing,* there are two copies of every piece of data

13. What is I/O bandwidth and latency and transaction time? [Nov/Dec 2014], [May/June 2014]

I/O throughput is sometimes called *I/O bandwidth,* and response time is sometimes called *latency.*
Transaction time is the sum of entry time , system response time and think time.

14. Differentiate between SRAM AND DRAM [R-2013]

SRAM	**DRAM**
SRAMs don’t need to refresh and so the access time is very close to the cycle time.	DRAM need to be refreshed
Capacity is low	Capacity is 4-8 times of SRAM
SRAM needs only minimal power to retain the charge in standby mode	Need more power to retain the charge.
SRAM designs are concerned with speed and capacity	DRAM designs the emphasis is on cost per bit and capacity

15. How RAID can improve the performance of I/O? [April/May 2015]

RAID - Redundant Array of Inexpensive Disks, (now commonly Redundant Array of Independent Disks) is a data storage virtualization technology that combines multiple physical disk drive components into a single logical unit for the purposes of data redundancy, performance improvement, or both. Thus improves I/O Performance.

16. What are the types of storage devices ?(NOV/DEC 2016)

- Magnetic Disk
- Magnetic Tapes
- CD-ROMs
- Flash Memory

PART B

1. Describe the need of cache optimization scheme. Give a description about schemes to reduce cache miss penalty and miss rate. [Apr/ May 2015], [Nov/ Dec 2014], [May/Jun 2014], [Nov/ Dec 2013]

Cache performance

The *average memory access time* is calculated as follows

Average memory access time = hit time + Miss rate x Miss Penalty.

Where Hit Time is the time to deliver a block in the cache to the processor (includes time to determine whether the block is in the cache), Miss Rate is the fraction of memory references not found in cache (misses/references) and Miss Penalty is the additional time required because of a miss.

The average memory access time due to cache misses predicts processor performance.

- First, there are other reasons for stalls, such as contention due to I/O devices using memory and due to cache misses.
- Second, The CPU stalls during misses, and the memory stall time is strongly correlated to average memory access time.

CPU time = (CPU execution clock cycles + Memory stall clock cycles) × Clock cycle time

There are 17 cache optimizations into four categories:

1 **Reducing the miss penalty**: multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches;

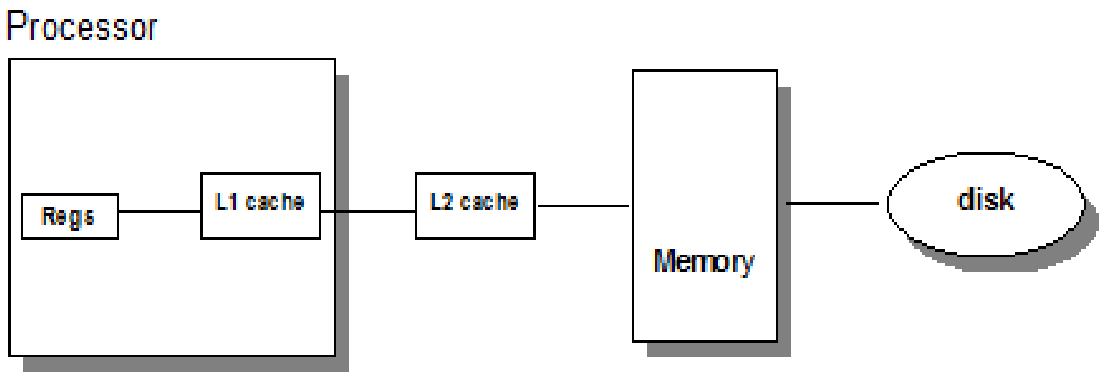
2 **Reducing the miss rate:** larger block size, larger cache size, higher associativity, pseudo-associativity, and compiler optimizations;

3 **Reducing the miss penalty or miss rate via parallelism**: nonblocking caches, hardware prefetching, and compiler prefetching;

4 **Reducing the time to hit in the cache**: small and simple caches, avoiding address translation, and pipelined cache access.

There are five optimizations techniques to reduce miss penalty.

i) First Miss Penalty Reduction Technique: Multi-Level Caches



The First Miss Penalty Reduction Technique follows the Adding another level of cache between the original cache and memory. The first-level cache can be small enough to match the clock cycle time of the fast CPU and the second- level cache can be large enough to capture many accesses that would go to main memory, thereby the effective miss penalty.

The definition of *average memory access time* for a two-level cache. Using the subscripts L_1 and L_2 to refer, respectively, to a first-level and a second-level cache, the formula is

Average memory access time = Hit time$_{L1}$ + Miss rate$_{L1}$ × Miss penalty$_{L1}$

and Miss penalty$_{L1}$ = Hit time$_{L2}$ + Miss rate$_{L2}$ × Miss penalty$_{L2}$

so Average memory access time = Hit time$_{L1}$ + Miss rate$_{L1}$× (Hit time$_{L2}$ + Miss rate$_{L2}$ × Miss penalty$_{L2}$)

Local miss rate—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate$_{L1}$ and for the second-level cache it is Miss rate$_{L2}$.

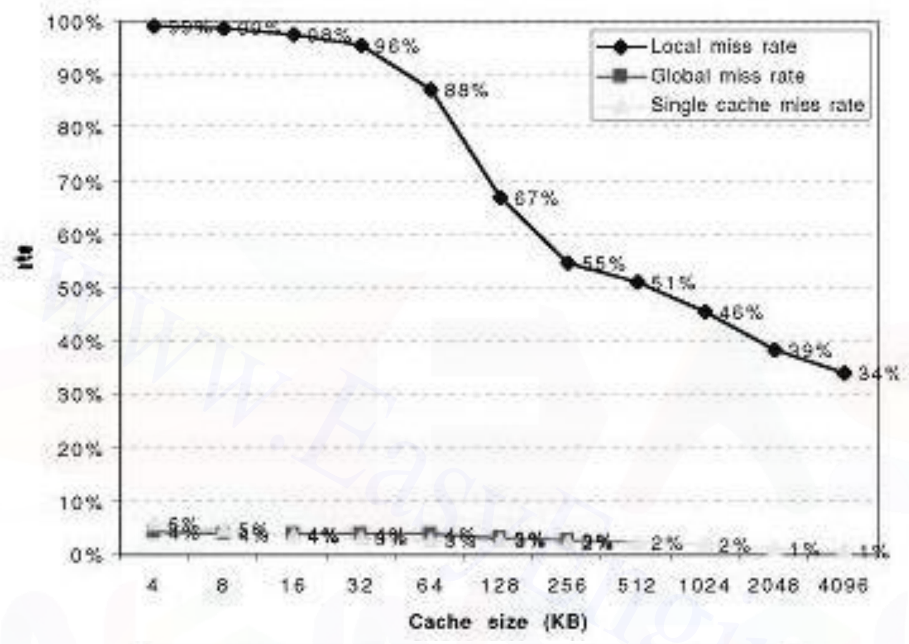
Global miss rate—The number of misses in the cache divided by the total num- ber of memory accesses generated by the CPU. Using the terms above, the global miss rate for the first-level cache is still just Miss rate$_{L1}$but for the second-level cache it is Miss rate$_{L1}$ × Miss rate$_{L2}$.

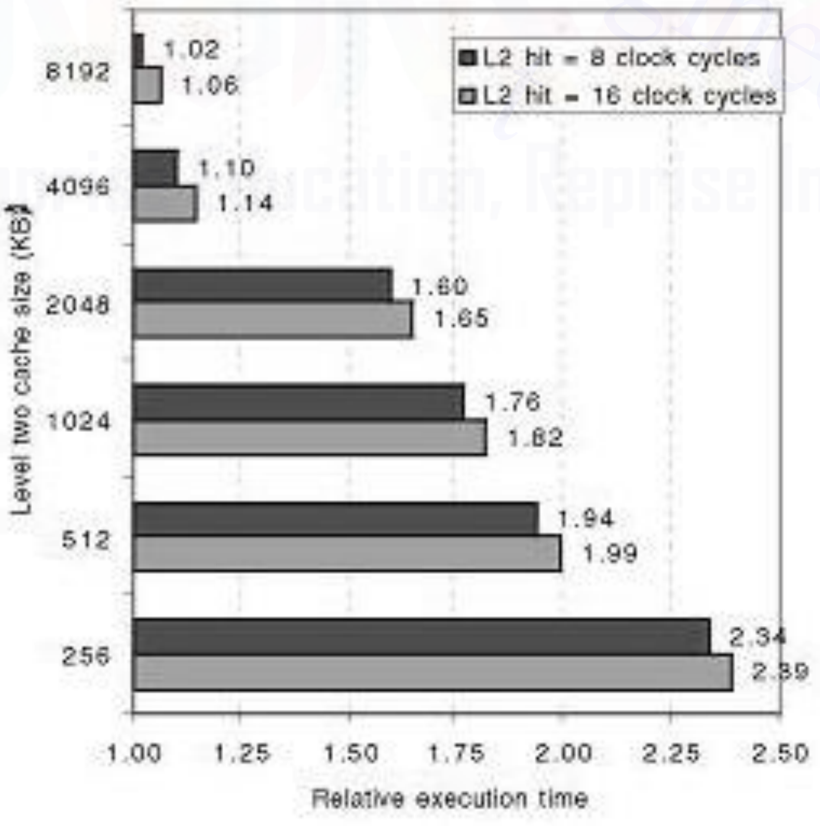
Average memory stalls per instruction = Misses per instruction$_{L1}$× Hit time$_{L2}$ +

Misses per instruction$_{L2}$ × Miss penalty$_{L2}$.

Consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the CPU, while the speed of the second-level cache only affects the miss penalty of the first-level cache.

Figures show how miss rates and relative execution time change with the size of a second-level cache for one design.





ii) Second Miss Penalty Reduction Technique: Critical Word First and Early Restart

Multilevel caches require extra hardware to reduce miss penalty, but not this second technique. It is based on the observation that the CPU normally needs just one word of the block at a time. This strategy is impatience: Don’t wait for the full block to be loaded before sending the requested word and restarting the CPU. Here are two specific strategies:

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. Critical-word-first fetch is also called *wrapped* fetch and *requested word first*.

Early restart—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

iii) Third Miss Penalty Reduction Technique: Giving Priority to Read Misses over Writes

This optimization serves reads before writes have been completed. We start with looking at complexities of a write buffer. With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss. The simplest way out of this is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue.

iv) Fourth Miss Penalty Reduction Technique: Merging Write Buffer

This technique also involves write buffers, this time improving their efficiency. Write through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. As mentioned above, even write back caches use a simple buffer when a block is replaced. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the CPU's perspective; the CPU continues working while the write buffer prepares to write the word to memory. If the buffer contains other modified blocks, the addresses can be checked to see if the address of this new data matches the address of the valid write buffer entry. If so, the new data are combined with that entry, called *write merging*.

If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time.

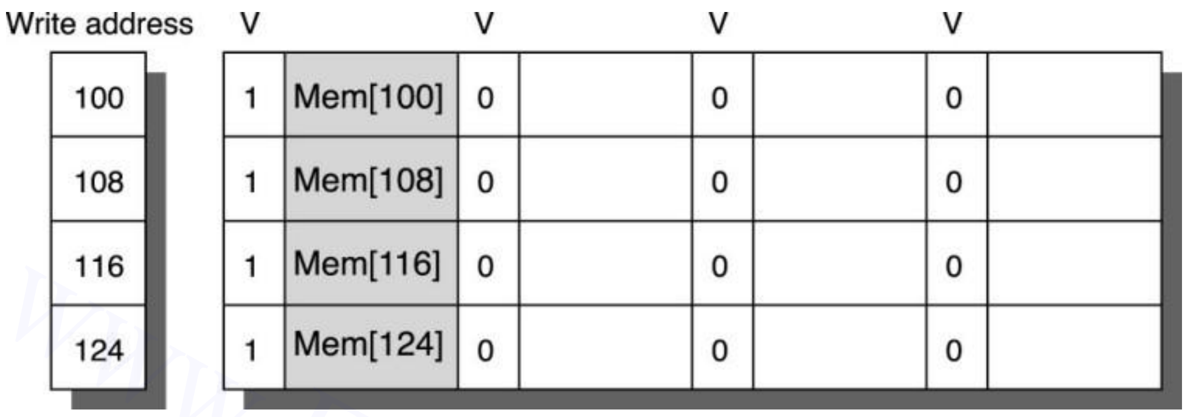


Figure: write buffer with write merging

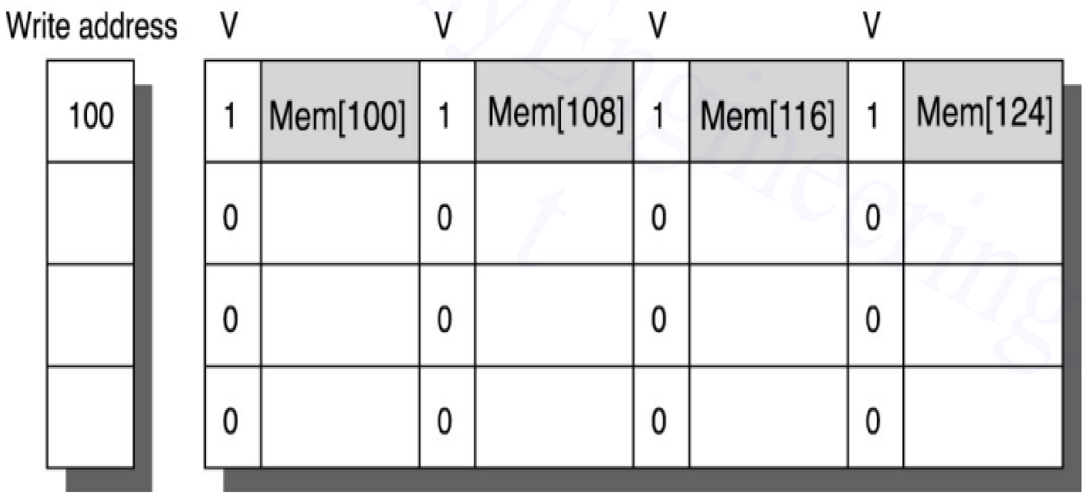


Figure: write buffer without write merging

The optimization also reduces stalls due to the write buffer being full. Figure shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted.

The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left; with valid bits (V) indicating whether or not the next sequential eight bytes are occupied in this entry. (Without write merging, the words to the right in the upper drawing would only be used for instructions which wrote multiple words at the same time.)

v) Fifth Miss Penalty Reduction Technique: Victim Caches

One approach to lower miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost.

Such "recycling" requires a small, fully associative cache between a cache and its refill path. Figure shows the organization. This *victim cache* contains only blocks that are discarded from a cache because of a miss "victims" and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped. The AMD Athlon has a victim cache with eight entries.

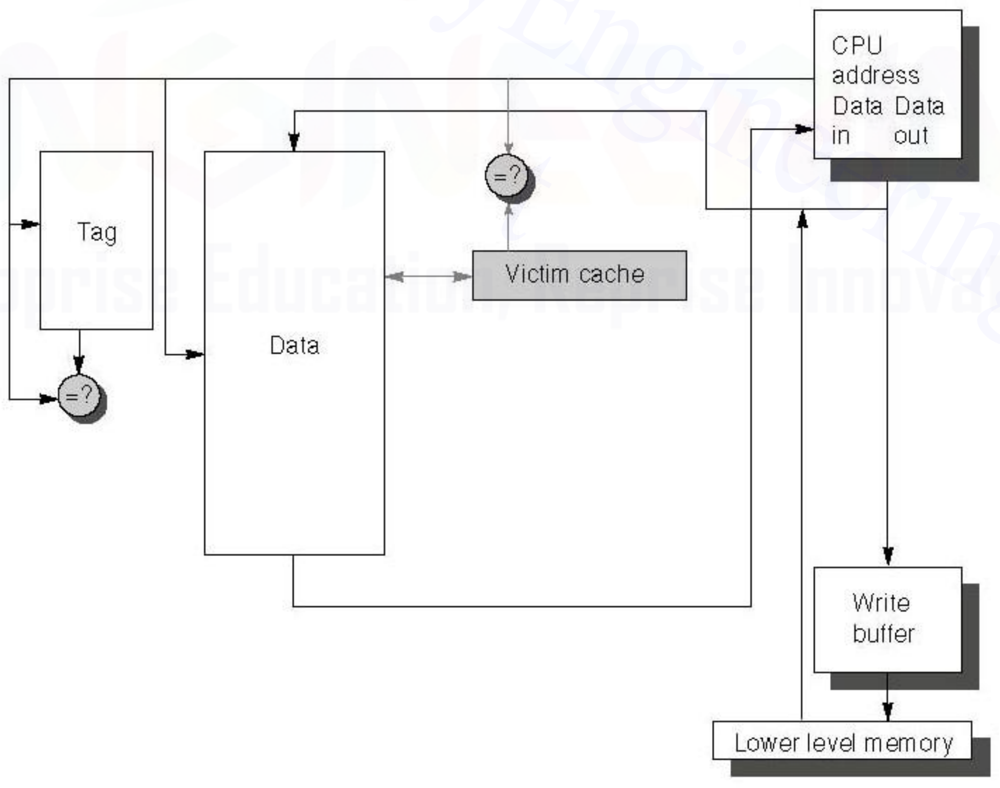


Fig: Placement of victim cache in the memory hierarchy

Reducing cache miss penalty and miss rate

The classical approach to improving cache behavior is to reduce miss rates, and there are five techniques to reduce miss rate. first start with a model that sorts all misses into three simple categories:

Compulsory—The very first access to a block *cannot be* in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.

Capacity—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur be-cause of blocks being discarded and later retrieved.

Conflict—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur be-cause a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses* or *interference misses*. The idea is that hits in a fully associative cache which become misses in an N-way set associative cache are due to more than N requests on some popular sets.

i) First Miss Rate Reduction Technique: Larger Block Size

The simplest way to reduce miss rate is to increase the block size. Figure shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

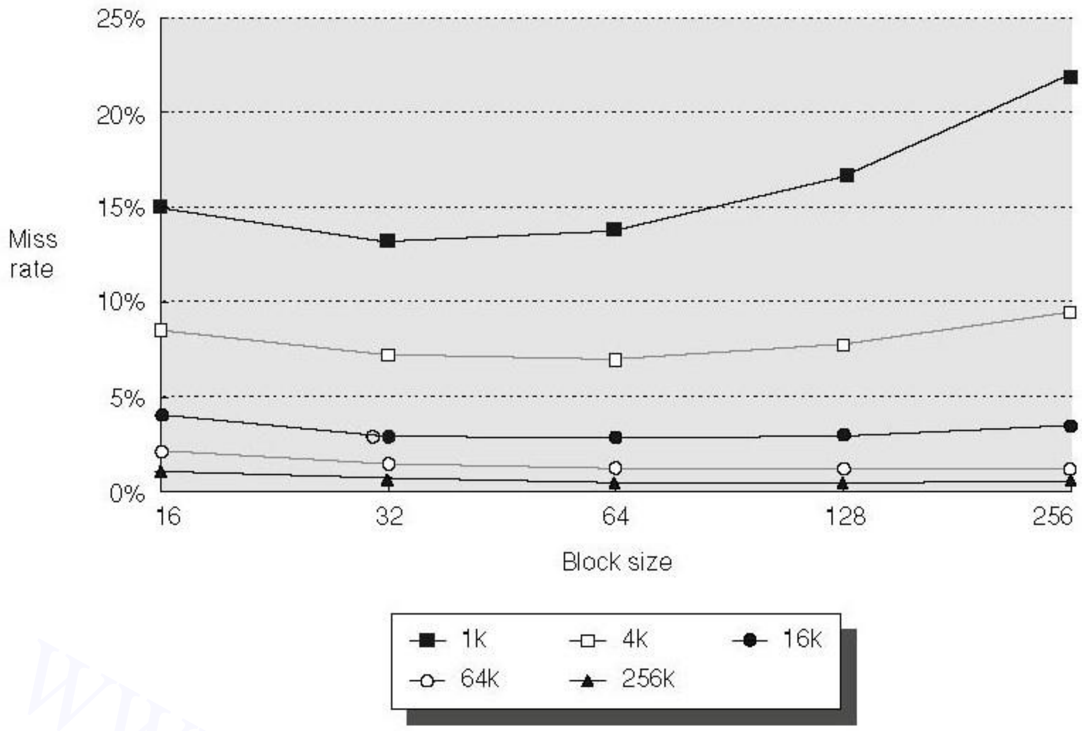


Fig: Miss rate versus block size for five different-sized caches

At the same time, larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly, there is little reason to increase the block size to such a size that it *increases* the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

ii) Second Miss Rate Reduction Technique: Larger caches

The obvious way to reduce capacity misses in the above is to increases capacity of the cache. The obvious drawback is longer hit time and higher cost. This technique has been especially popular in off-chip caches: The size of second or third level caches in 2001 equals the size of main memory in desktop computers.

iii) Third Miss Rate Reduction Technique: Higher Associativity:

Generally the miss rate improves with higher associativity. There are two general rules of thumb that can be drawn. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the 8-way entries to the capacity miss, since capacity misses are calculated using fully associative cache.

iv) Fourth Miss Rate Reduction Technique: Way Prediction and Pseudo-Associative Caches

In *way-prediction*, extra bits are kept in the cache to predict the set of the *next* cache access. This prediction means the multiplexer is set early to select the desired set, and only a single tag comparison is performed that clock cycle. A miss results in checking the other sets for matches in subsequent clock cycles.

The Alpha 21264 uses way prediction in its instruction cache. (Added to each block of the instruction cache is a set predictor bit. The bit is used to select which of the two sets to try on the *next* cache access. If the predictor is correct, the instruction cache latency is one clock cycle. If not, it tries the other set, changes the set predictor, and has a latency of three clock cycles.

v) Fifth Miss Rate Reduction Technique: Compiler Optimizations

Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses. Reordering the instructions reduced misses by 50% for a 2-KB direct-mapped instruction cache with 4-byte blocks, and by 75% in an 8-KB cache.

Another code optimization aims for better efficiency from long cache blocks. Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.

Loop Interchange:

Some programs have nested loops that access data in memory in non-sequential order. Simply exchanging the nesting of the loops can make the code access the data in the order it is stored. Assuming the arrays do not fit in cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded.

```
/* 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];
```

This optimization improves cache performance without affecting the number of instructions executed.

2.(i) Discuss in detail about various hit time reduction techniques to improve cache performance. [Nov/ Dec 2013]

Reducing hit time

- On many machines, cache access time limits the clock cycle rate !
- Therefore, cache design affects more than average memory access time, it affects everything.

1. Small& simple caches
   - The less hardware that is necessary to implement a cache, the shorter the critical path through the hardware.
   - **Direct-mapped** is faster than set associative for both reads and writes.
     - In particular, tag checking can overlap data transmission (there is only one piece of data for each index).
   - Fitting the cache on the chip with the CPU is also very important for fast access times.
   - Therefore, fast clock cycle time encourages small direct-mapped caches.

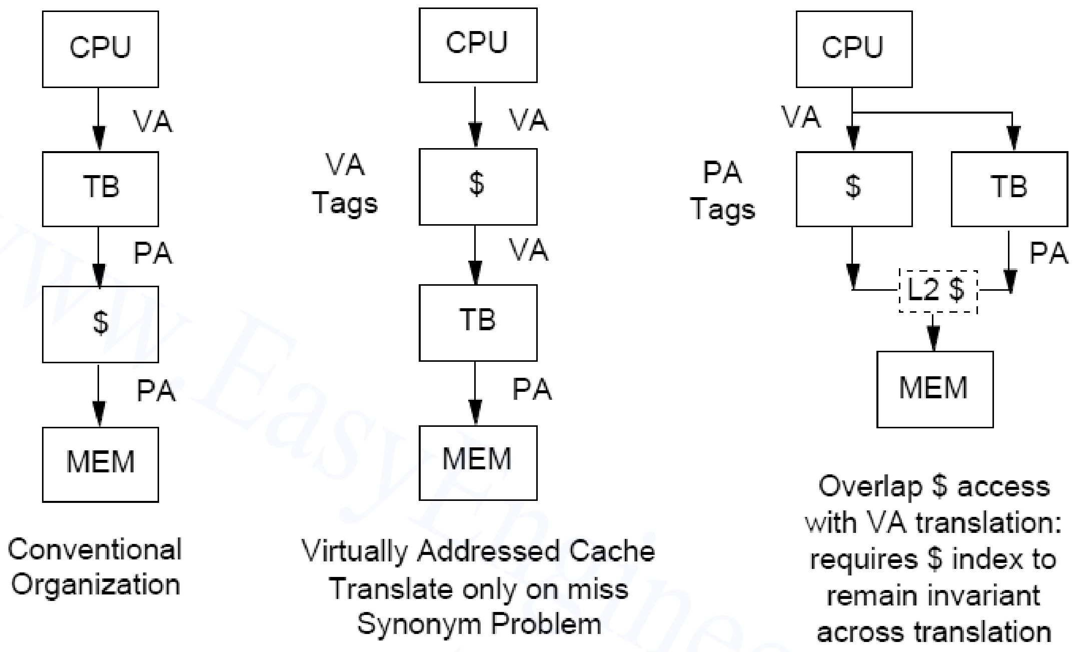
2.(i)Avoid address translation during indexing

- The CPU uses virtual addresses that must be mapped to a physical address.
- The cache may either use virtual or physical addresses.

- A cache that indexes by virtual addresses is called a virtual cache, as opposed to a physical cache.

o A virtual cache reduces hit time since a translation from a virtual address to a physical address is not necessary on hits.

o Also, address translation can be done in parallel with cache access, so penalties for misses are reduced as



CPU
VA
TB
PA
$
PA
MEM
Conventional Organization
CPU
VA
VA Tags
$
VA
TB
PA
MEM
Virtually Addressed Cache Translate only on miss Synonym Problem
CPU
VA
PA Tags
$
TB
PA
L2 $
MEM
Overlap $ access with VA translation: requires $ index to remain invariant across translation

Fig: Virtually Addressed Cache

2(ii).Avoid address translation during indexing with physical portion of address

o Virtual cache difficulties include:

- Process switches require cache purging

o In virtual caches, different processes share the same virtual addresses even though they map to different physical addresses.

o When a process is swapped out, the cache must be purged of all entries to make sure that the new process gets the correct data.

- One solution: PID tags
- Increase the width of the cache address tags to include a process ID (instead of purging the cache.)
- The current process PID is specified by a register.

- If the PID does not match, it is **not** a hit even if the address matches.

21 12 11 0

Page Address	Page offset	
Address Lag	Index	Block offset

Figure: Page address and Page offset

- Virtual cache difficulties include:

Aliasing Two **different virtual** addresses may have the **same physical** address.

- In such a case, it is possible to end up with two copies of the same block!
- **Solution to aliasing**

(i)**Anti-aliasing hardware**

- A hardware solution called **anti-aliasing** guarantees every cache block a unique physical address.
  - Every virtual address maps to the same location in the cache.

(ii)**Page coloring**

- This software technique forces aliases to share some address bits.
  - Therefore, the virtual address and physical address match over these bits.

- A direct-mapped cache that is 2^k bytes (where k is the number of matching bits) or smaller can **never** have duplicate physical addresses for blocks.

(iii)**Using the page offset**

An alternative to get the best of both virtual and physical caches.

- If we use the page offset to index the cache, then we can overlap the virtual address translation process with the time required to **read** the tags.

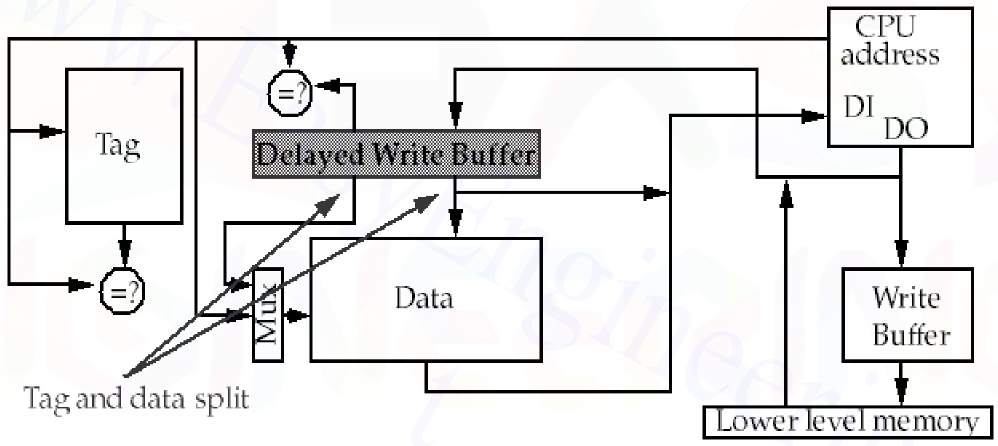
- Note that the page offset is unaffected by address translation.

- However, this restriction forces the cache size to be smaller than the page size.
    - Since the index comes only from the "physical" portion of the virtual address (the page offset).

- After doing both in parallel, the tag is checked against the **physical** address stored in the cache.
- High associativity allows for larger cache sizes.

3. Pipelined writes

- Write hits take longer than read hits because tag checking is required before the data is written.
    - One solution is to pipeline the writes (Alpha AXP 21064):



- The second stage of the write (cache is updated with new data) occurs during the first stage of the next write.
- Allows tag checking and data writing to occur simultaneously.

4. Fast Writes on Misses via Small Subblocks

- If most writes are 1 word, subblock size is 1 word, & write through then always write subblock& tag immediately
    - Tag match and valid bit already set: Writing the block was proper,& nothing lost by setting valid bit on again.
    - Tag match and valid bit not set: The tag match means that this is the proper block; writing the data into the subblock makes it appropriate to turn the valid bit on.

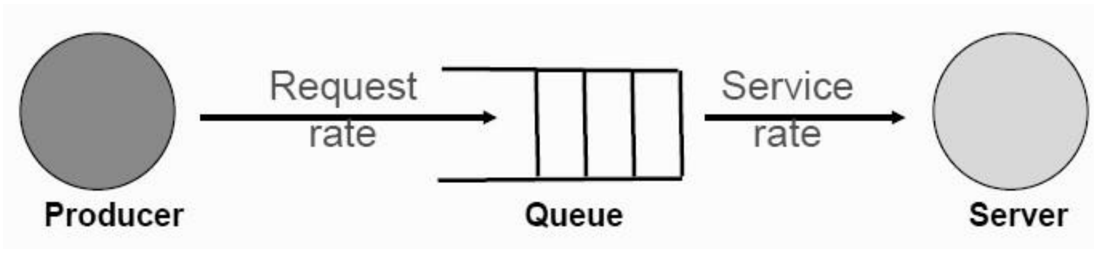
- Tag mismatch: This is a miss and will modify the data portion of the block. As this is a write-through cache, however, no harm was done; memory still has an up-to-date copy of the old value. Only the tag to the address of the write and the valid bits of the other subblock need be changed because the valid bit for this subblock has already been set

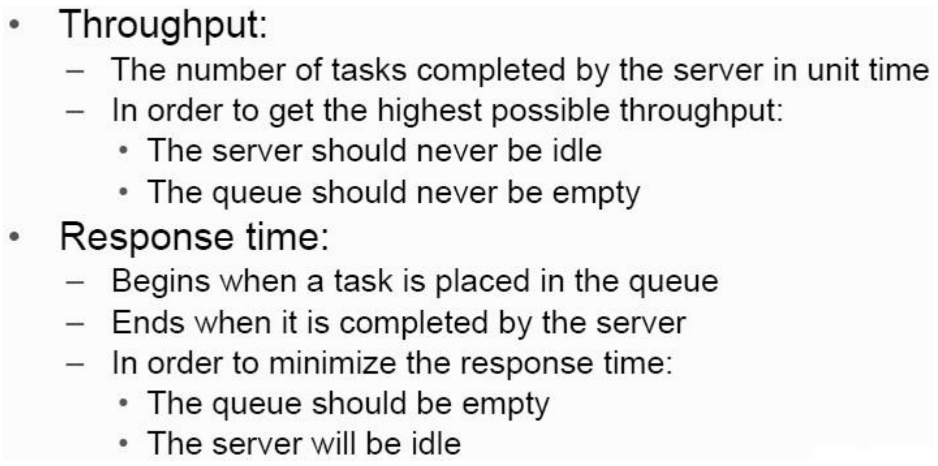
(ii) Elaborate on different methods to measure the performance of I/O.[Apr/ May 2015], [May/Jun 2014], [May/Jun 2013]

I/O PERFORMANCE:

- I/O System performance depends on many aspects of the system ("limited by weakest link in the chain"):
  - The CPU
  - The memory system:
    - Internal and external caches, Main Memory
  - The underlying interconnection (buses)
  - The I/O controller
  - The I/O device
  - The speed of the I/O software (Operating System)
  - The efficiency of the software's use of the I/O devices
- Two common performance metrics:
  - Throughput: I/O bandwidth
  - Response time: Latency
- Throughput (I/O processes per second)
  - Useful for file serving and transaction processing
- Latency - total time for an I/O process from start to finish
  - Most important to users
  - Latency = controller time + wait time + (NumOfBytes / bandwidth) + CPU time – overlap

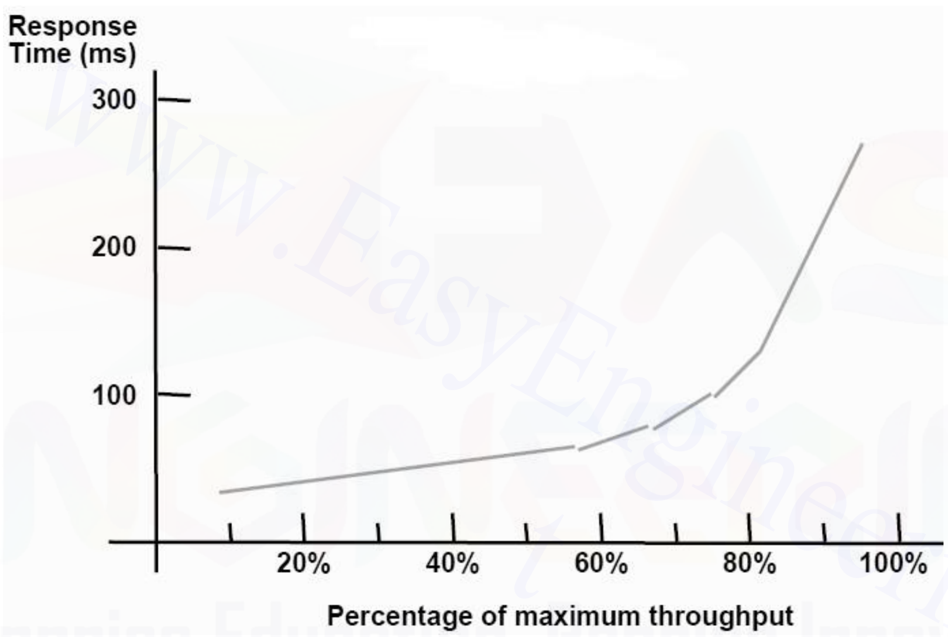
Simple – Producer Server Model



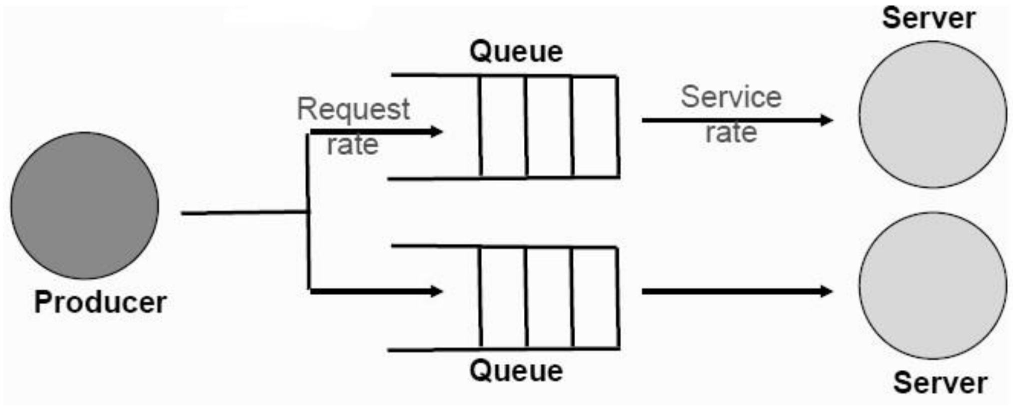


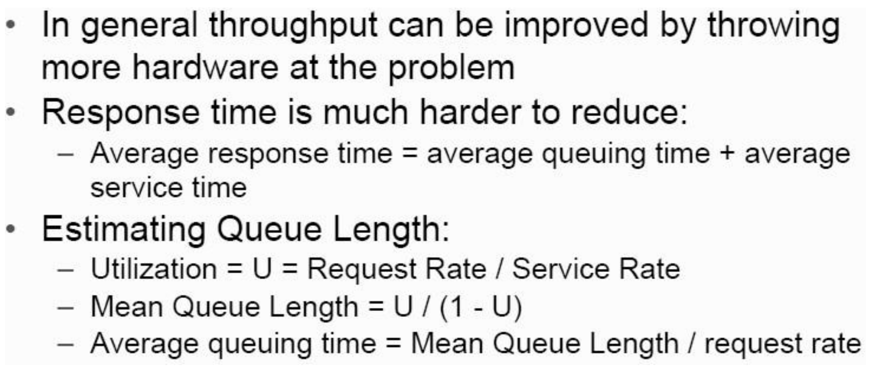
- Throughput:
  - The number of tasks completed by the server in unit time
  - In order to get the highest possible throughput:
    - The server should never be idle
    - The queue should never be empty
- Response time:
  - Begins when a task is placed in the queue
  - Ends when it is completed by the server
  - In order to minimize the response time:
    - The queue should be empty
    - The server will be idle

Throughput Vs Respond Time



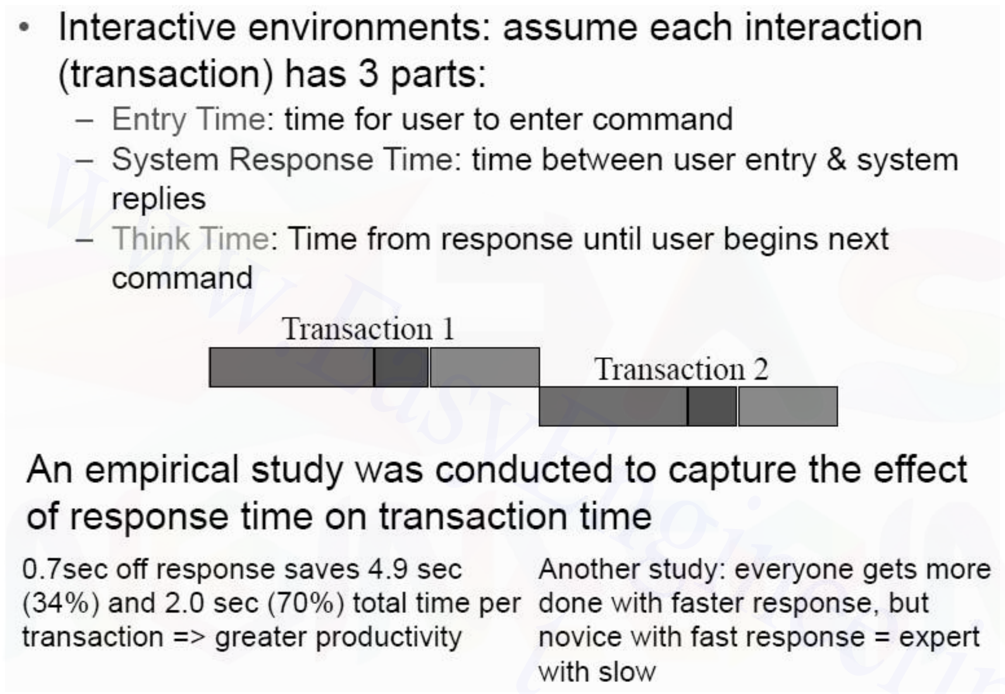
Throughput Enhancement





- In general throughput can be improved by throwing more hardware at the problem
- Response time is much harder to reduce:
  - Average response time = average queuing time + average service time
- Estimating Queue Length:
  - Utilization = U = Request Rate / Service Rate
  - Mean Queue Length = U / (1 - U)
  - Average queuing time = Mean Queue Length / request rate

Response Time Vs Productivity



- Interactive environments: assume each interaction (transaction) has 3 parts:
  - Entry Time: time for user to enter command
  - System Response Time: time between user entry & system replies
  - Think Time: Time from response until user begins next command

Transaction 1
Transaction 2

An empirical study was conducted to capture the effect of response time on transaction time

0.7sec off response saves 4.9 sec (34%) and 2.0 sec (70%) total time per transaction => greater productivity

Another study: everyone gets more done with faster response, but novice with fast response = expert with slow

3. How will you improve the main memory performance?

Main Memory

- Main memory is usually made from DRAM while caches use SRAM.
- SRAM is faster (by almost an order of magnitude).
  - However, it's also more expensive per bit and 1/4 to 1/8 as dense as DRAM (1 transistor versus 6 transistors).
- We now turn to optimizing DRAM performance.
- Performance measures include:
  - Latency
    - Important for caches.
  - Bandwidth
    - Important for I/O.

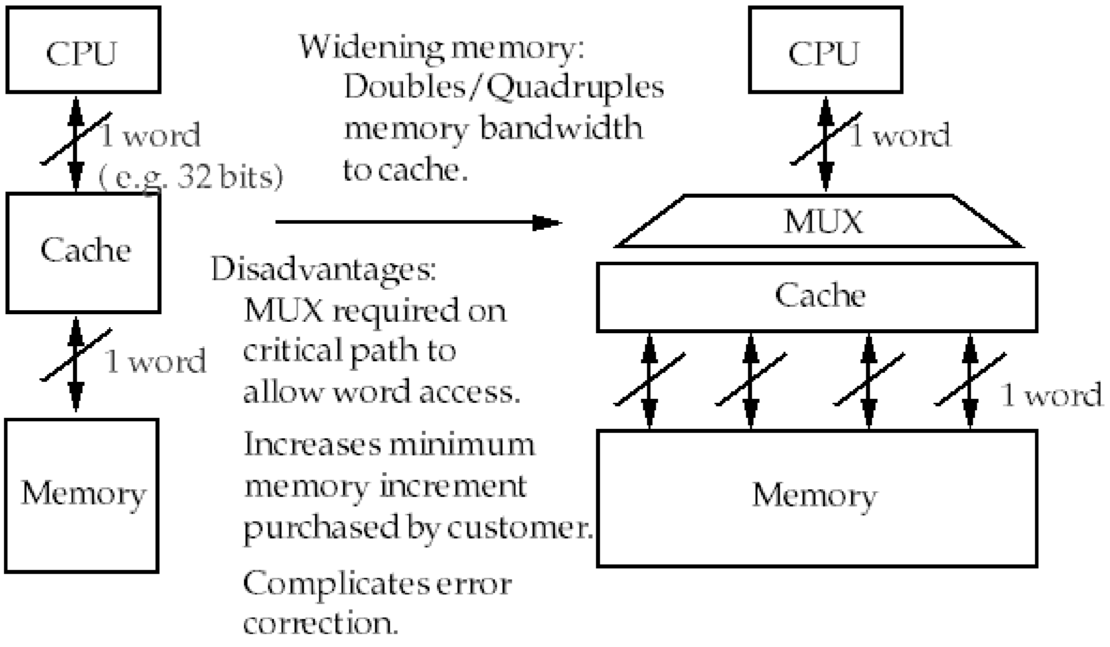
        - Also for cache with second-level and larger block sizes.

Improving Main Memory Performance

Latency measures:

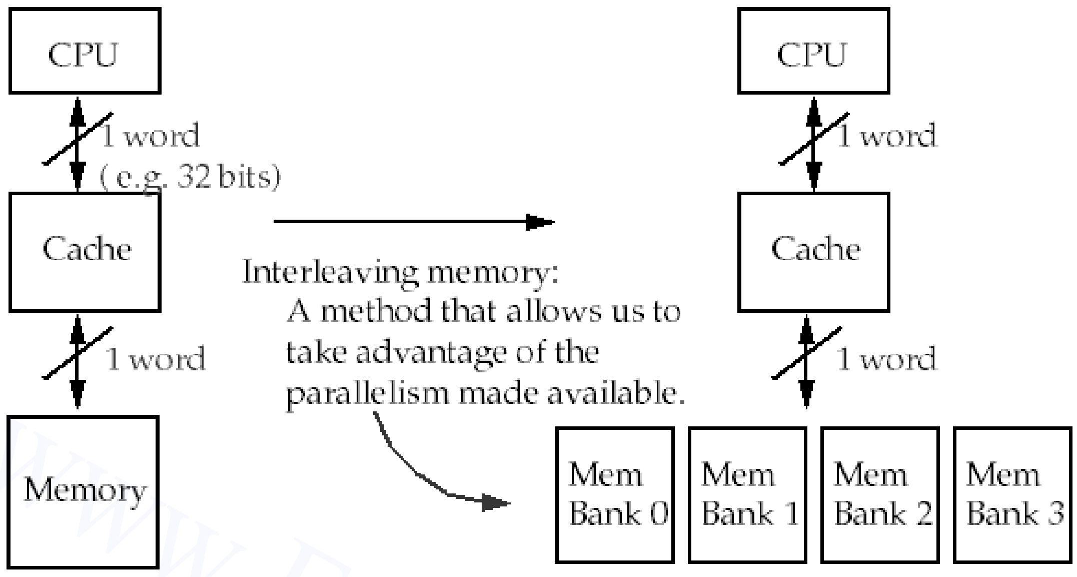
- Access time
    - Time between when a read is requested and when the desired word arrives.
- Cycle time
    - This is the minimum time between the starts of two accesses to memory.
    - This is at least as long as access time, and is usually longer.
- Refresh time
    - DRAMs must occasionally refresh their data.
        - This is done by reading all of the cells in a row and writing them back.
    - This must be done every few milliseconds.
    - However, this operation consumes less than 5% of total time.
    - This is true because the time necessary to refresh is proportional to the **square root** of the size of the DRAM.
    - Amdahl suggested that memory capacity should grow linearly with CPU speed.
    - Memory capacity grows **four-fold** every **three** years to supply this demand.
- The CPU-DRAM performance gap is a problem, however, since DRAM performance improvement is only about 7% per year.
    - Cache innovations have addressed this problem to some degree.
    - We will now look at innovations in main memory organizations that are more cost effective.

(i) Wider main memory


CPU
1 word (e.g. 32 bits)
Cache
1 word
Memory
Widening memory: Doubles/Quadruples memory bandwidth to cache.
Disadvantages: MUX required on critical path to allow word access.
Increases minimum memory increment purchased by customer.
Complicates error correction.
CPU
1 word
MUX
Cache
1 word
Memory

- DRAM chips are typically 1-8 bits wide.
    - Any number of them can be accessed in parallel without extra delay.
    - By increasing the width of memory, the CPU can get more bits in a single cycle.
- This increases bandwidth between cache and memory.
- For example, consider a cache with 4 word blocks.
- Main memory might require:
    - 4 cycles to send the address.
    - 40 cycles to access memory.
    - 4 cycles to transfer over the bus.
- If the memory is only **one word** wide, a miss would require 4 x (4 + 40 + 4) = 192 cycles!
- If the memory is enlarged to **4 words wide**, miss time is only 48 cycles.

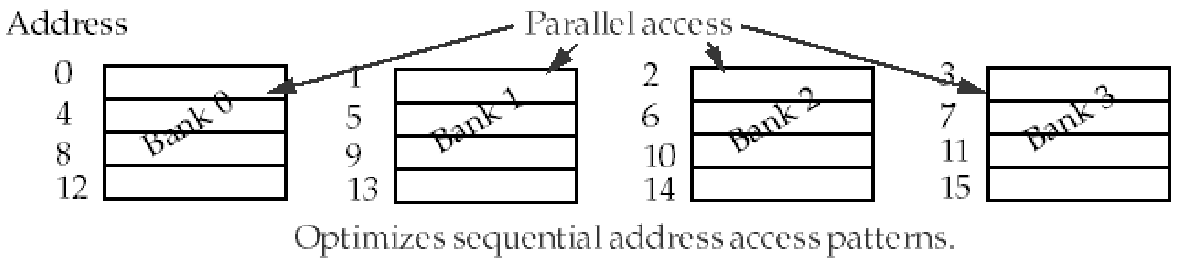
(ii) Interleaved memory



- Banks are often one word wide, so bus width need not be changed.
  - However, several independent areas of memory can be accessed simultaneously.
  - For example, we could fetch a block by
    - Sending 1 address.
    - Waiting for a single memory cycle.
    - Transferring 4 words for a total time of 4 + 40 + (4 x 4) = 60 cycles.
    - Which is a little slower than wider memory (due to bus limitations) but it has several advantages.

  - Individual writes can also be overlapped if they are addressed to different banks.

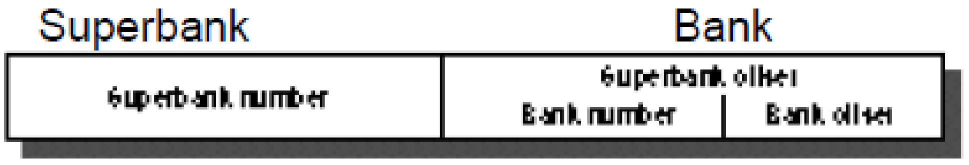
  - One possible interleaving strategy: **Word interleaving** :



- Read access optimization is possible if, for example, cache block size is four words since parallel access is possible (no conflicts).
- Also, write-back caches make writes sequential as well as reads, improving efficiency even further.
- How many banks are sufficient?
  - One rule might be `# of banks >= # of clocks to access a word in a bank'.
  - This allows up to 1 word per clock cycle in best case.

(iii) Independent memory banks

- The interleaved memory concept can be extended to remove **all** restrictions on memory access.
  - We assumed for interleaved memory that only a **single memory controller** was present.
  - This allowed the interleaving of sequential access patterns.
  - Address line sharing among the banks is possible in this scheme.
- We can also use **multiple independent controllers**, e.g. one for I/O devices, one for cache reads and one for cache writes.
  - Banks are still accessed in parallel, but now there may be multiple independent requests serviced simultaneously.
- This can be particularly useful with **nonblocking** caches (caches that allow multiple outstanding reads misses).
- And multiprocessors.



(iv) Avoiding memory bank conflicts

- As with caches, programs can be modified to improve memory performance.
  - The most important is to keep all the banks running.

- Programs that access all banks evenly will perform best.
    - However, data memory references are **not** random and may end up going to the same bank.
- Using a prime number of memory banks makes this work well.
    - However, using a prime number makes the division operation expensive:

$$\text{Bank number} = \text{Address MOD Number of banks}$$

$$\text{Address within bank} = \frac{\text{Address}}{\text{Number of banks}} \quad \longleftarrow$$

- There are schemes that use a prime number of banks and fast modulo arithmetic to distribute memory accesses to many banks of memory.

- For example, the following can be used:

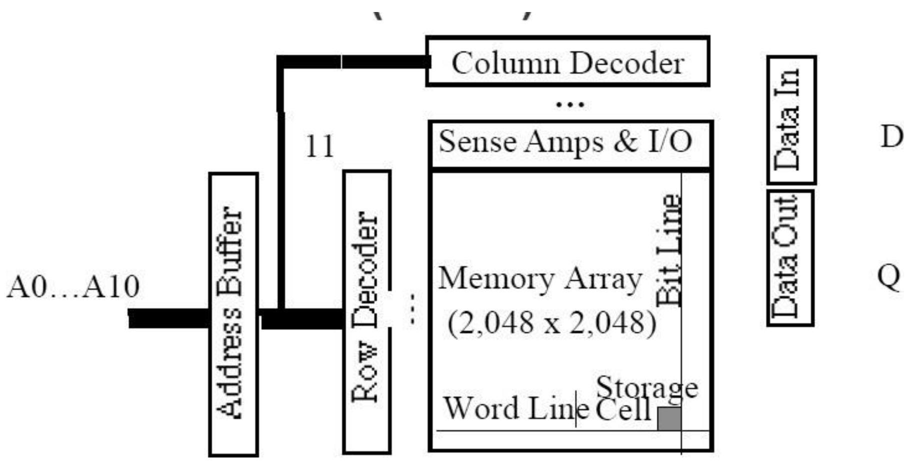
$$\text{Bank number} = \text{Address MOD Number of banks}$$

$$\text{Address within bank} = \text{Address MOD Number of words in bank}$$

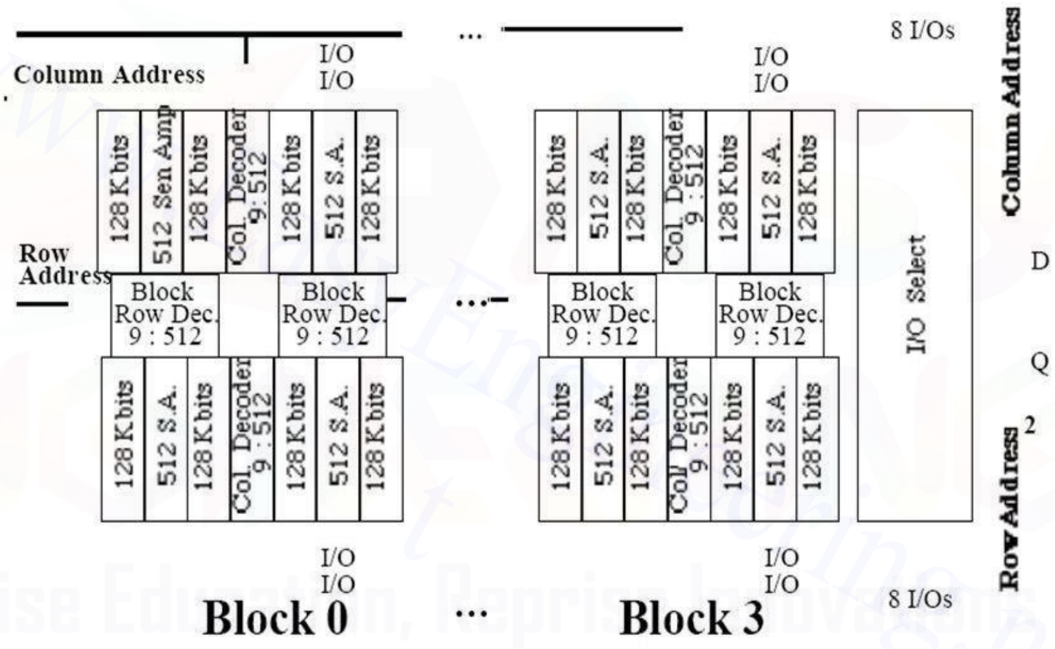
- This avoids the use of an expensive `non power of 2' division operation shown previously.
- There is a proof that guarantees that the above mapping provides a unique mapping between an address and a memory location.
- For numbers of the form $2^N -1$, there is fast hardware to implement the operation.

(v) DRAM-specific interleaving

- The previous methods work with any memory technology.
- We now look at techniques that take advantage of the nature of DRAMs.
- The first three take advantage of the individual row access and column access operations that occur on a memory access.
    - DRAMs buffer a row of bits inside the DRAM for column access.
    - The size of the buffer is usually the square root of the DRAM size, e.g. 16Kbits for 64MBits.
    - In order to improve performance, DRAMs are designed to allow multiple accesses to this buffer, **eliminating** the row access time.



DRAM Logical Organization



DRAM Physical Organization

- Nibble mode
  - The DRAM can supply three extra bits from locations sequential to the one just accessed.
  - This can be done after each RAS (Row Access Strobe).
- Page mode
  - The DRAM can act as an SRAM once a row has been selected.

- For example, random bits from the row can be selected by changing just the column address.
- This can occur until the next RAS or refresh.

- Static column mode (Extended Data Out [**EDO**] RAM)
  - Very similar to page mode, except that it is not necessary to toggle (clock) the column access strobe line every time the column address changes.
- These optimizations can improve bandwidth by a factor of **four** .
- Synchronous DRAM (SDRAM)
  - In this type of DRAM, the clock is supplied to the RAM chip, and all signals are synchronized to it.
  - This allows the RAM to run at higher speeds.
  - Similarly, sequential data can be retrieved faster, at the rate of one bit per clock cycle.
- VRAM
  - Video RAM is used to drive displays.
  - It can be written or read using a normal interface or a special interface that **outputs rows** one bit at a time (good for video displays!).

4. Explain RAID architecture in detail and its various levels. [May/Jun 2014], [Nov/ Dec 2013], [May/Jun 2013]

An innovation that improves both dependability and performance of storage systems is *disk arrays*. One argument for arrays is that potential throughput can be increased by having many disk drives and, hence, many disk arms, rather than one large drive with one disk arm. Although a disk array would have more faults than a smaller number of larger disks when each disk has the same reliability, dependability can be improved by adding redundant disks to the array to tolerate faults. That is, if a single disk fails, the lost information can be reconstructed from redundant information.

The only danger is in having another disk fail between the time the first disk fails and the time it is replaced (termed *mean time to repair,* or MTTR). Since the *mean time to failure* (MTTF) of disks is tens of years, and the MTTR is measured in hours, redundancy can make the measured reliability of 100 disks much higher than that of a single disk. These systems have become known by the acronym *RAID*, stand-ing originally for *redundant array of inexpensive disks*, although some have re-named it to *redundant array of independent disks*

The several approaches to redundancy have different overhead and performance.

Figure shows the standard RAID levels. It shows how eight disks of user data must be supplemented by redundant or check disks at each RAID level. It also shows the minimum number of disk failures that a system would survive.

RAID level	Minimum number of Disk faults survived	Example Data disks	Corresponding Check disks	Corporations producing RAID products at this level
0 Non-redundant striped	0	8	0	Widely used
1 Mirrored	1	8	8	EMC, Compaq (Tandem), IBM
2 Memory-style ECC	1	8	4	
3 Bit-interleaved parity	1	8	1	Storage Concepts
4 Block-interleaved parity	1	8	1	Network Appliance
5 Block-interleaved	1	8	1	Widely used

distributed parity				
6 P+Q redundancy	2	8	2	

FIGURE RAID levels, their fault tolerance, and their overhead in redundant disks.

No Redundancy (RAID 0)

This notation is refers to a disk array in which data is striped but there is no redundancy to tolerate disk failure. Striping across a set of disks makes the collection appear to software as a single large disk, which simplifies storage management. It also improves performance for large accesses, since many disks can operate at once. Video editing systems, for example, often stripe their data.

RAID 0 something of a misnomer as there is no redundancy, it is not in the original RAID taxonomy, and striping predates RAID. However, RAID levels are often left to the operator to set when creating a storage system, and RAID 0 is often listed as one of the options. Hence, the term RAID 0 has become widely used.

Mirroring (RAID 1)

This traditional scheme for tolerating disk failure, called *mirroring* or *shadowing*, uses twice as many disks as does RAID 0. Whenever data is written to one disk, that data is also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" to get the desired information. Mirroring is the most expensive RAID solution, since it requires the most disks.

The RAID terminology has evolved to call the former RAID 1+0 or RAID 10

("striped mirrors") and the latter RAID 0+1 or RAID 01 ("mirrored stripes").

Bit-Interleaved Parity (RAID 3)

The cost of higher availability can be reduced to 1/*N*, where *N* is the number of disks in a *protection group*. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure. RAID 3 is popular in applications with large data sets, such as multimedia and some scientific codes.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo two. The assumption behind this technique is that failures are so rare that taking longer to recover from failure but reducing redundant storage is a good trade-off.

Block-Interleaved Parity and Distributed Block-Interleaved Parity (RAID 4 and RAID 5)

In RAID 3, every access went to all disks. Some applications would prefer to do smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the next RAID levels. Since error-detection information in each sector is checked on reads to see if data is correct, such “small reads” to each disk can occur independently as long as the minimum access is one sector.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in Figure. A “small write” would require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.

RAID 4 efficiently supports a mixture of large reads, large writes, small reads, and small writes. One drawback to the system is that the parity disk must be updated n every write, so it is the bottleneck for back-to-back writes. To fix the parity-write bottleneck, the parity information can be spread throughout all the disks so that there is no single bottleneck for writes. The distributed parity organization is RAID 5.

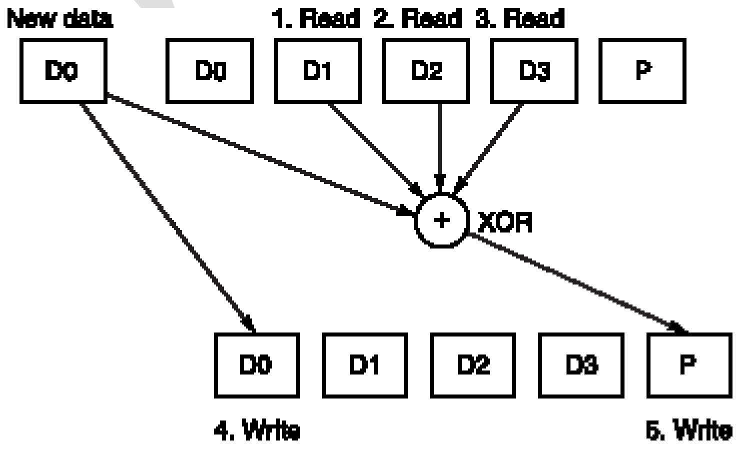


Figure: Data distribution in RAID 4

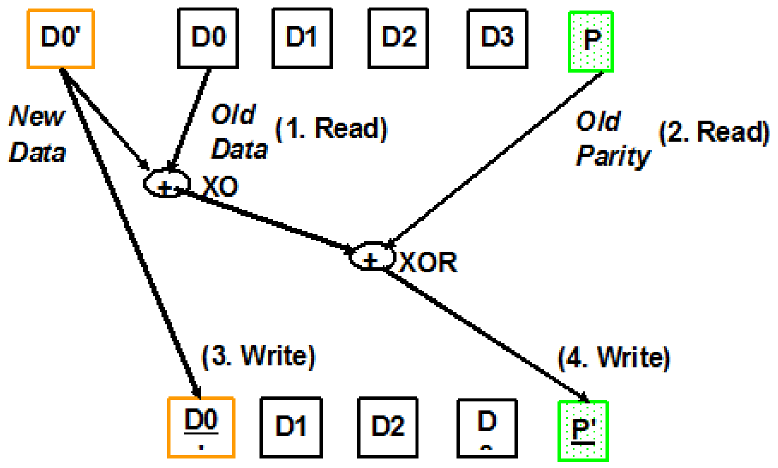


Figure: Data distribution in RAID 5

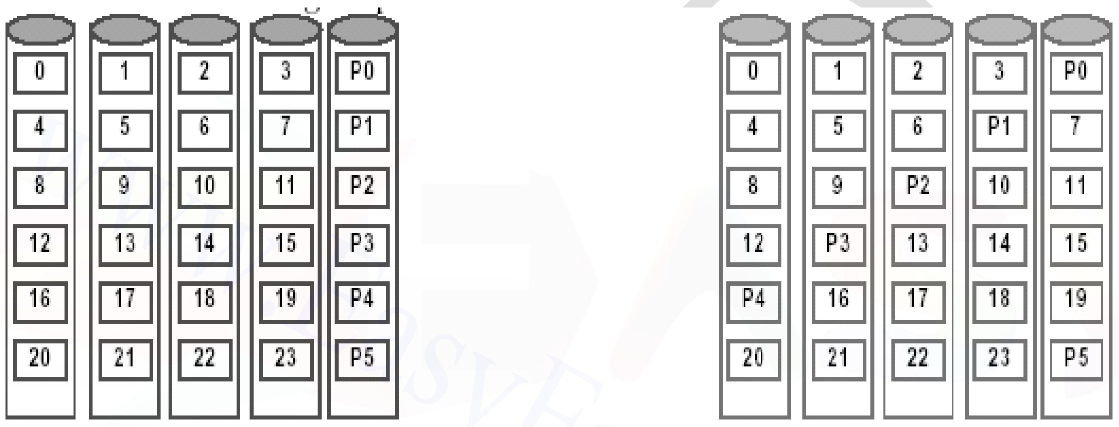


Fig: Data distribution in RAID4 vs Raid5

Figure shows how data are distributed in RAID 4 vs. RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the stripe units are not located in the same disks.

For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur at the same time as the write to block 8. Those same writes to the organization on the left would result in changes to blocks P1 and P2, both on the fifth disk, which would be a bottleneck.

P+Q redundancy (RAID 6)

Parity based schemes protect against a single, self-identifying failures. When a single failure is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. Yet another parity block is added to allow recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure works as well, except now there are six disk accesses instead of four to update both P and Q information.

Errors and Failures in Real Systems

Publications of real error rates are rare for two reasons. First academics rarely have access to significant hardware resources to measure. Second industrial, researchers are rarely allowed to publish failure information for fear that it would be used against their companies in the marketplace. Below are four exceptions.

Berkeley’s Tertiary Disk

The Tertiary Disk project at the University of California created an art-image server for the Fine Arts Museums of San Francisco. This database consists of high quality images of over 70,000 art works. The database was stored on a cluster, which consisted of 20 PCs containing 368 disks connected by a switched Ethernet. It occupied in seven 7-foot high racks.

Component	Total in System	Total Failed	% Failed
SCSI Controller	44	1	2.3%
SCSI Cable	39	1	2.6%
SCSI Disk	368	7	1.9%
IDE Disk	24	6	25.0%
Disk Enclosure - Backplane	46	13	28.3%
Disk Enclosure - Power Supply	92	3	3.3%
Ethernet Controller	20	1	5.0%
Ethernet Switch	2	1	50.0%
Ethernet Cable	42	1	2.3%
CPU/Motherboard	20	0	0%

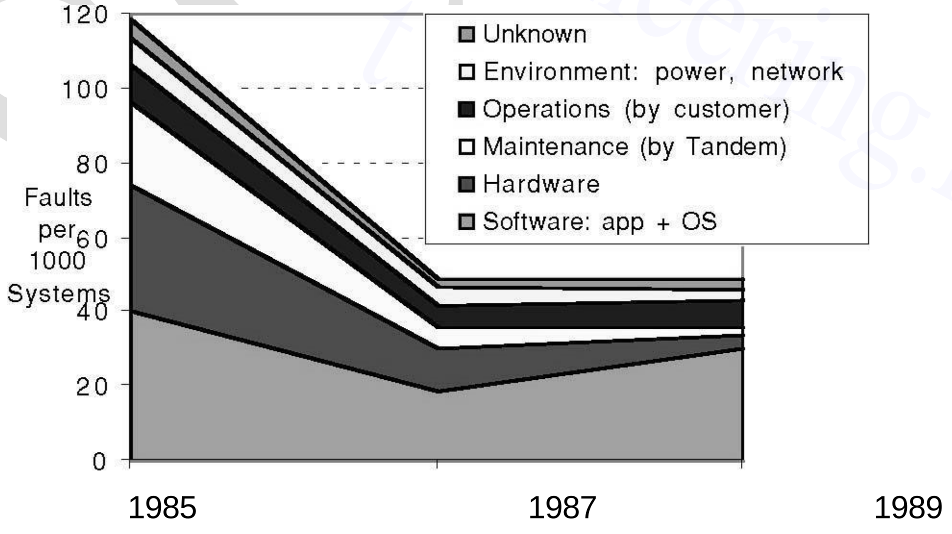
FIGURE Failures of components in Tertiary Disk over eighteen months of operation.

Figure shows the failure rates of the various components of Tertiary Disk. In advance of building the system, the designers assumed that data disks would be the least reliable part of the system, as they are both mechanical and plentiful. As Tertiary Disk was a large system with many redundant components, it had the potential to survive this wide range of failures. Components were connected and mirrored images were placed no single failure could make any image unavailable. This strategy, which initially appeared to be overkill, proved to be vital.

This experience also demonstrated the difference between transient faults and hard faults. *Transient faults* are faults that come and go, at least temporarily fixing themselves. *Hard faults* stop the device from working properly, and will continue to misbehave until repaired.

Tandem

The next example comes from industry. Gray [1990] collected data on faults for Tandem Computers, which was one of the pioneering companies in fault tolerant computing. Figure 7.21 graphs the faults that caused system failures between 1985 and 1989 in absolute faults per system and in percentage of faults encountered. The data shows a clear improvement in the reliability of hardware and maintenance. Disks in 1985 needed yearly service by Tandem, but they were re-placed by disks that needed no scheduled maintenance. Shrinking number of chips and connectors per system plus software's ability to tolerate hardware faults reduced hardware's contribution to only 7% of failures by 1989. And when hardware was at fault, software embedded in the hardware device (firmware) was often the culprit. The data indicates that software in 1989 was the major source of reported outages (62%), followed by system operations (15%).



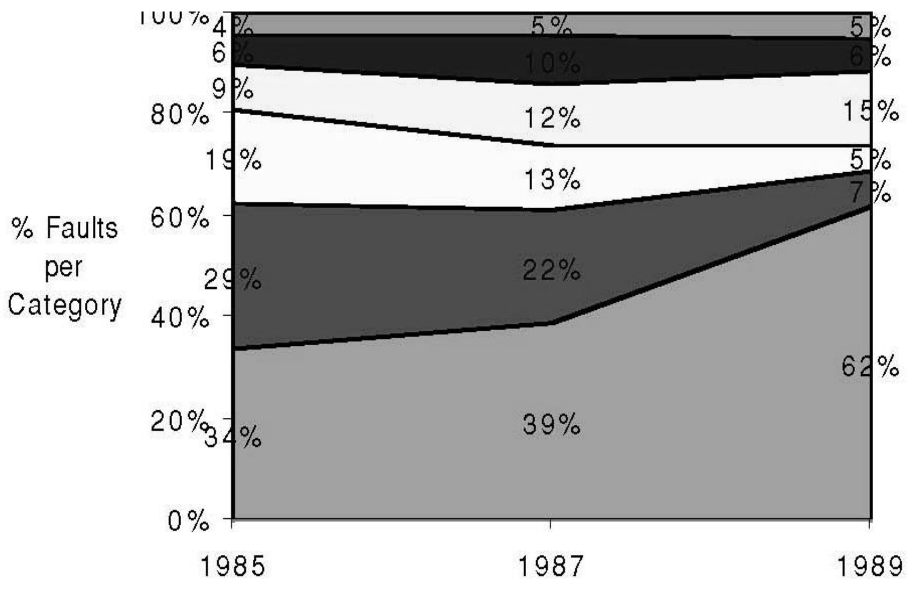


FIGURE **Faults in Tandem between 1985 and 1989.** Gray [1990] collected these data for the fault tolerant Tandem computers based on reports of component failures by customers.

The problem with any such statistics is that these data only refer to what is reported; for example, environmental failures due to power outages were not reported to Tandem because they were seen as a local problem.

VAX

The next example is also from industry. Murphy and Gent [1995] measured faults in VAX systems. They classified faults as hardware, operating system, system management, or application/networking. Figure 7.22 shows their data for 1985 and 1993. They tried to improve the accuracy of data on operator faults by having the system automatically prompt the operator on each boot for the reason for that reboot. They also classified consecutive crashes to the same fault as operator fault. Note that the hardware/operating system went from causing 70% of the failures in 1985 to 28% in 1993. Murphy and Gent expected system management to be the primary dependability challenge in the future.

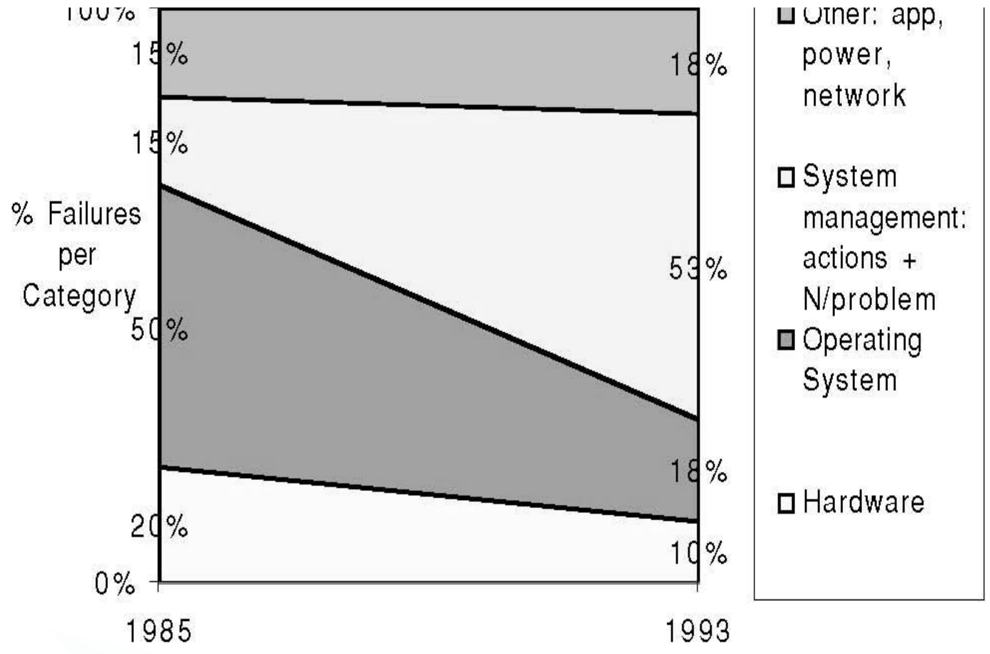


FIGURE Causes of system failures on Digital VAX systems between 1985 and 1993 collected by Murphy and Gent [1995]. System management crashes include having several crashes for the same problem, suggesting that the problem was difficult for the operator to diagnose. It also included operator actions that directly resulted in crashes, such as giving parameters bad values, bad configurations, and bad application installation.

FCC

The final set of data comes from the government. The Federal Communications Commission (FCC) requires that all telephone companies submit explanations when they experience an outage that affects at least 30,000 people or lasts thirty minutes. These detailed disruption reports do not suffer from the self-reporting problem of earlier figures, as investigators determine the cause of the outage rather than operators of the equipment. Kuhn [1997] studied the causes of outages between 1992 and 1994 and Enriquez [2001] did a follow-up study for the first half of 2001. In addition to reporting number of outages, the FCC data includes the number of customers affected and how long they were affected. Hence, we can look at the size and scope of failures, rather than assuming that all are equally important. Figure 7.23 plots the absolute and relative number of customer-outage minutes for those years, broken into four categories:

- Failures due to exceeding the network's capacity (overload).
- Failures due to people (human).
- Outages caused by faults in the telephone network software (software).
- Switch failure, cable failure, and power failure (hardware).

These four examples and others suggest that the primary cause of failures in large systems today is faults by human operators. Hardware faults have declined due to a decreasing number of chips in systems, reduced power, and fewer connectors. Hardware dependability has improved through fault tolerance techniques such as RAID. At least some operating systems are considering reliability implications before new adding features, so in 2001 the failures largely occur elsewhere.

5.Explain in detail about the types of storage devices. [May/Jun 2014]

Types of Storage Devices

There are various types of Storage devices such as magnetic disks, magnetic tapes, automated tape libraries, CDs, and DVDs.

The First Storage device magnetic disks have dominated nonvolatile storage since 1965. Magnetic disks play two roles in computer systems:

- Long Term, nonvolatile storage for files, even when no programs are running
- A level of the memory hierarchy below main memory used as a backing store for virtual memory during program execution.

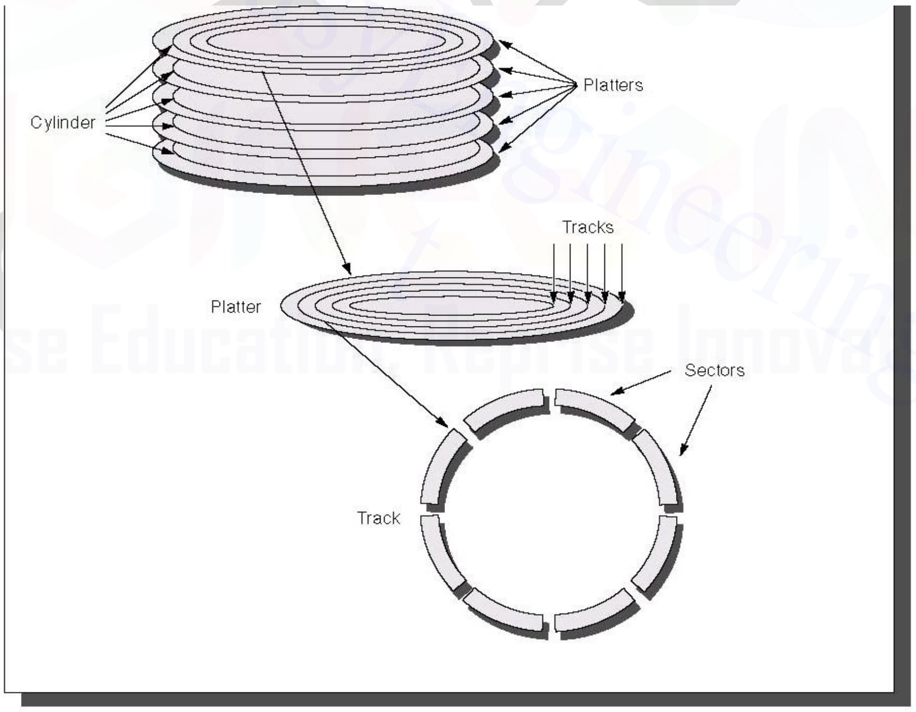


Figure: Disks are organized into platters , tracks and sectors

A magnetic disk consists of a collection of *platters* (generally 1 to 12), rotating on a spindle at 3,600 to 15,000 revolutions per minute (RPM). These platters are metal or glass disks covered with magnetic recording material on both sides, so 10 platters have 20 recording surfaces.

The disk surface is divided into concentric circles, designated *tracks*. There are typically 5,000 to 30,000 tracks on each surface. Each track in turn is divided into *sectors* that contain the information; a track might have 100 to 500 sectors. A sector is the smallest unit that can be read or written. IBM mainframes allow users to select the size of the sectors, although most systems fix their size, typically at 512 bytes of data. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code, a gap, the sector number of the next sector, and so on.

To read and write information into a sector, a movable *arm* containing a *read/ write head* is located over each surface. To read or write a sector, the disk controller sends a command to move the arm over the proper track. This operation is called a *seek*, and the time to move the arm to the desired track is called *seek time*.

Average seek time is the subject of considerable misunderstanding. Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average was open to wide interpretation.

The time for the requested sector to rotate under the head is the *rotation latency* or *rotational delay*. The average latency to the desired information is obviously halfway around the disk; if a disk rotates at 10,000 revolutions per minute (RPM), the average rotation time is therefore

Average Rotation Time = 0.5/10,000RPM = 0.5/(10,000/60)RPM = 3.0ms

The next component of disk access, *transfer time,* is the time it takes to transfer a block of bits, typically a sector, under the read-write head. This time is a function of the block size, disk size, rotation speed, recording density of the track, and speed of the electronics connecting the disk to computer. Transfer rates in 2001 range from 3 MB per second for the 3600 RPM, 1-inch drives to 65 MB per second for the 15000 RPM, 3.5-inch drives.

The Future of Magnetic Disks

The disk industry has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as improvement in *areal density*, measured in bits per square inch:

Areal Density = (Tracks/Inch) on a disk surface X (Bits/Inch) on a track

Through about 1988 the rate of improvement of areal density was 29% per year, thus doubling density every three years. Between then and about 1996, the rate improved to 60% per year, quadrupling density every three years and matching the traditional rate of DRAMs. From 1997 to 2001 the rate increased to 100%, or doubling every year. In 2001, the highest density in commercial products is 20 billion bits per square inch, and the lab record is 60 billion bits per square inch.

Optical Disks:

One challenger to magnetic disks is *optical compact disks,* or *CDs*, and its successor, called *Digital Video Discs* and then *Digital Versatile Discs* or just *DVDs*. Both the *CD-ROM* and *DVD-ROM* are removable and inexpensive to manufacture, but they are read-only mediums. These 4.7-inch diameter disks hold 0.65 and 4.7 GB, respectively, although some DVDs write on both sides to double their capacity. Their high capacity and low cost have led to CD-ROMs and DVD-ROMs replacing floppy disks as the favorite medium for distributing software and other types of computer data.

The popularity of CDs and music that can be downloaded from the WWW led to a market for rewritable CDs, conveniently called CD-RW, and write once CDs, called CD-R. In 2001, there is a small cost premium for drives that can record on CD-RW. The media itself costs about $0.20 per CD-R disk or $0.60 per CD-RW disk. CD-RWs and CD-Rs read at about half the speed of CD-ROMs and CD-RWs and CD-Rs write at about a quarter the speed of CD-ROMs.

Magnetic Tape:

Magnetic tapes have been part of computer systems as long as disks because they use the similar technology as disks, and hence historically have followed the same density improvements. The inherent cost/performance difference between disks and tapes is based on their geometries:

- Fixed rotating platters offer random access in milliseconds, but disks have a limited storage area and the storage medium is sealed within each reader.
- Long strips wound on removable spools of “unlimited” length mean many tapes can be used per reader, but tapes require sequential access that can take seconds.

One of the limits of tapes had been the speed at which the tapes can spin without breaking or jamming. A technology called *helical scan tapes* solves this problem by keeping the tape speed the same but recording the information on a diagonal to the tape with a tape reader that spins much faster than the tape is moving. This technology increases recording density by about a factor of 20 to 50. Helical scan tapes were developed for low-cost VCRs and camcorders, which brought down the cost of the tapes and readers.

Automated Tape Libraries

Tape capacities are enhanced by inexpensive robots to automatically load and store tapes, offering a new level of storage hierarchy. These *near line* tapes mean access to terabytes of information in tens of seconds, without the intervention of a human operator.

Flash Memory

Embedded devices also need nonvolatile storage, but premiums placed on space and power normally lead to the use of Flash memory instead of magnetic recording. Flash memory is also used as a rewritable ROM in embedded systems, typically to allow software to be upgraded without having to replace chips. Applications are typically prohibited from writing to flash memory in such circumstances.

Like electrically erasable and programmable read-only memories (EEPROM), Flash memory is written by inducing the tunneling of charge from transistor gain to a floating gate. The floating gate acts as a potential well which stores the charge, and the charge cannot move from there without applying an external force. The primary difference between EEPROM and Flash memory is that Flash restricts write to multi-kilobyte blocks, increasing memory capacity per chip by reducing area dedicated to control. Compared to disks, Flash memories offer low power consumption (less than 50 milliwatts), can be sold in small sizes, and offer read access times comparable to DRAMs. In 2001, a 16 Mbit Flash memory has a 65 ns access time, and a 128 Mbit Flash memory has a 150 ns access time.

EC6009-ADVANCE COMPUTER ARCHITECTURE

(Regulation 2013)

PART A - (10*2=20 marks)

1. Define spatial and temporal locality.
2. What is dependability.
3. List the major advantage of dynamic scheduling using tomasulo's approach.
4. What is data hazards.
5. What are the omissions in the SIMD extension instruction set.
6. Describe the similarities and difference between multimedia SIMD computer and GPU..
7. What is multicore architecture.
8. What are the major disadvantage of DSM architecture?
10. Explain the need to implement memory as a hierarchy

PART B – (5*13=65 Marks)

11.(a) (i) Explain in detail about trends in power and energy on IC with suitable example..

OR

(b) Discuss about the guideline and principle that are useful in design and evaluate the performance of computer system with example.

12. (a) (i) Describe the basic compiler techniques for exploiting instruction level parallel (10)

(ii) Briefly compare the hardware and software speculations? (6)

OR

(b) (i) Explain the method of exploiting LIP and VLIW processor. (8)

(ii) Discuss the important limitation of ILP. (8)

13.(a) (i) Explain data level parallelism in Vector architecture in detail. (16)

OR

(b) Discuss GPU architecture with neat diagram. (16)

14.(a) With neat diagram explain the distributed shared memory architecture . (16)

OR

(b) (i) Explain about synchronization techniques with example. (10)

(ii) Discuss about models of memory consistency. (6)

15. (a) Discuss various basic cache optimization techniques with example.

OR

(b) (i) Briefly describe about various RAID levels with diagram. (8)

(ii) List and explain various I/O Performance measure. (8)

Electronics and Communication Engineering

EC 6009 – ADVANCED COMPUTER ARCHITECTURE

(Regulation 2013)

Answer all Questions

PART A (10 * 2 = 20)

1. What are the five trends in computer Technology ?
2. How to find the cost of an integrated circuits ?
3. Explain the idea behind dynamic scheduling.
4. Give an example for data dependence.
5. Differentiate GPU and CPU.
6. What are the primary components of instruction set architecture of VMIPS?
7. List the methods for providing synchronization in threads.
8. Define sequential consistency.
9. List the six basic optimizations techniques of cache.
10. What are the types of storage Devices?

PART B (5*16=80)

11. (a) Write short notes on energy and power consumption in a microprocessor (16)

(or)

(b)Discuss the performance evaluation methods of different computers(16)

12. (a) (i) Explain the types of dependencies in ILP (8)

(ii) explain the compilation techniques that can be used to expose instruction level parallelism.(8)

(or)

(b) (i) Explain dynamic scheduling. Explain how it is used to reduce data hazards. (8)

(ii) Define Multithreading. Explain how ILP is achieved using multithreading with an example (8)

13. (a) Discuss similarities and difference between vector architecture and GPUs (16)

(or)

(b) Explain detecting and enhancing loop level parallelism in details.(16).

14. (a) Describe distributed shared memory architecture in detail (16).

(or)

(b) Explain Models of memory consistency in details (16).

15. (a) Explain the categories of misses and how will u reduce the cache miss rate (16)

(or)

(b) (i) Explain the various ways to measure I/O processor (8)

(ii) Explain the various levels of RAID (8)

Question Paper Code:71393

B.E/B.Tech. DEGREE EXAMINATION, APRIL/MAY 2015

Sixth Semester

CS 2354/CS 64/10144/CS 604 - ADVANCED COMPUTER ARCHITECTURE

Regulation 2008/2010

Time: 3 Hours Maximum

: 100 marks

Answer ALL Questions

Part A (10*2= 20 marks)

1. Briefly describe data Hazards.
2. Point out the different type of data dependences.
3. Difference between VLIW and EPIC Processor.
4. Briefly describe about the Register Stack Mechanism in IA -64 Register model.
5. What is the use of branch prediction buffer?
6. Write a note on multiprocessor cache coherence.
7. Point out how RAID can improve the performance of I/O

8 What is the need to implement memory as a hierarchy?

9. Enlist the features of SMT architecture.

10. Point out the advantage and disadvantage of hetrogeneous multi-core processor.

PART B- (5*16=80 Marks)

11. A) Explain how computer technology can be used to enhance a processor ability to exploit ILP.

OR

B) what are the different way for branch prediction? Describe how pipeline performance issues can be reduced by branch prediction.

12) A) Discuss about Itanium processor and its IA 64 instruction set architecture.

OR

B) What is speculative execution? Compare and contrast hardware and software speculation mechanisms.

13) a) Discuss in detail about the performance issues in symmetric and distributed shared memory architectures,

OR

b) What is the need for Memory consistency model? Explain its various types.

14. a) Describe the need for cache optimization schemes. Give a description about the advanced cache optimization schemes to reduce cache miss penalty and miss rate.

OR

b)) Elaborate on the difference methods to measure the performance ofI/O.

15) a) Compare and Contrast Intel Multi core architecture and SUN CMP architecture.

OR

B) What is hardware multithreading ? Compare and contrast Fine grained Multi-Treading and Coarse grained Multi- Treading.

Time: 3 Hours Maximum

: 100 marks

Answer ALL Questions

Part A (10*2= 20 marks)

1. What is the major limitation of pipeline techniques..
2. How many branch selected entries are in a (2,2) branch predictor that has a total of 8K bits in prediction buffer.
3. List down the issues in the design of simultaneous multithreaded processor.
4. What is the key features in the microarchitecture of Itanium 2..
5. What do you mean by Multitreading?
6. Define sequential consistency.
7. Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

8 Define Transaction time?

9. What do you mean by hyperthreading.
10. Define SMT.

PART B- (5*16=80 Marks)

11. A) Describe the types of optimization performed by the modern compilers.

OR

B) (i) Discuss about Tomasulo's Algorithm

(ii). Explain the various dynamic branch prediction schemes.

12) A) (i) Explain the methods of exploiting ILP using VLIW processor.

(ii) Describe the register naming approach for implementation of speculation.

OR

B) (i) Compare the hardware VS software speculation mechanisms.

(ii) What are the important limitations to ILP?

13) a) Explain the models of memory consistency in multiprocessor systems.

OR

b) Explain the snoopy cache coherence protocol for ensuring coherence in symmetric multiprocessor.

14. a) Explain the categories of misses and how will you reduce cache miss rates.

OR

b) (i) Explain the steps in designing an I/O systems.

(ii) Write a short note on fault, failures and errors.

15) a) Explain the SUN CMP architecture.

OR

B) Discuss the design issues and implementation of Intel Multicore architecture.

CS2354/CS64/10144 CS604-ADVANCE COMPUTER ARCHITECTURE

(Regulation 2008/2010)

Time:Three Hours Maximum: 100 Marks

Answer All Questions

PART A-(10 x 2= 20 marks)

1. What are the possible types of data hazards?
2. What do you mean by branch prediction buffer?
3. What is VLIW?
4. List out the major parts of Itanium Processor?
5. When is a memory said to be coherent in a multi-processor system?
6. Why do we need synchronization?
7. Define the term access time , bandwidth.
8. State the principle of locality. List the types of locality.
9. Write the benefits of multi-core architecture.
10. Why are design issues of SMT and CMP architectures important?

PART A-(5 x 16= 80 marks)

11.a)(i)Explain the three types of dependencies in instructions. (10)

(ii)Explain the functions of Tomasulo's approach (6)

(or)

b)(i) List put the decisions and transformation to be considered to obtain unrolled code in loop unrolling and scheduling. (8)

(ii)What is static branch prediction method? Explain its use. (8)

12. a)(i)Explain the problems with VLIW approach. (8)

(ii)Explain about compiler speculation with hardware support.
(8)

(or)

b)(i)Compare hardware with software speculation mechanisms.
(8)

(ii)Describe the various execution units in IA 64 processors.
(8)

13. a) Explain the function of symmetric shared memory architecture and compare it with Distributed Shared memory architecture.
(16)

(or)

b) (i)What is multithreading? Explain the approaches to multithreading?
(8)

(ii)Explain the models of memory consistency.
(8)

14.a)(i)Discuss the techniques for reducing Cache miss Penalty.
(8)

(ii)Breifly explain the types of storage devices.
(8)

(or)

b)(i)Explain various levels of RAID.
(8)

(ii)List and explain various I/O performance measures.
(8)

15. a)(i)Describe the two levels of threads. (8)

(ii)Discuss the factors that limits the issues in simultaneous multithreading.(8)

(or)

b)With a neat sketch, explain the architecture of IBM cell processor in detail. Highlight the feature of it.
(16)

CS2354/CS64/10144 CS604-ADVANCED COMPUTER ARCHITECTURE

(Regulation 2008/2010)

Time: Three Hours Maximum: 100 Marks

Answer All Questions

PART A-(10 x 2= 20 marks)

1. What is pipeline CPI?
2. What are the uses of static branch predictors?
3. What are the advantages of superblock approach?
4. What are the functional units of Itanium Processor?
5. What are the advantages of MIMD multiprocessors?
6. What is the importance of memory consistency model?
7. What are the categories of cache organization based on placing a block?
8. What are the measures of I/O performance?
9. What is multicore?
10. What are the design issues of cell processor?

PART A-(5 x 16= 80 marks)

11.a)(i)What is Dynamic Scheduling? Explain how it is used to reduce data hazards (8)

(ii)Explain how to reduce branch costs with dynamic hardware prediction (8)

(or)

b)(i) Explain the various types of dependencies in ILP (8)

(ii)Explain in detail the concept involved in Instruction level Parallelism (8)

12.a)(i)Distinguish between hardware versus software speculation mechanism in detail(8) (ii)Explain how the instruction format of IA64 is used to achieve the parallelism (8)

(or)

b)(i)Explain the trace scheduling in exploiting ILP. List its advantages (8)

(ii)Briefly discuss the limitations of ILP (8)

13. a) (i) what do you mean by snooping protocol? Explain how it is used to maintain the coherence (8)

(ii)Explain the different models of memory consistency (8)

(or)

b) (i)Discuss the directory based cache coherence protocol (8)

(ii)Explain how the hardware primitives can be used to build synchronization operations (8)

14. a)(i)Explain the various hit time reduction techniques (8)

(ii)Explain the RAID architecture in detail (8)

(or)

b) What are the categories of cache misses? Explain the various techniques available foe reducing cache miss rate. (16)

15. a)(i)Discuss the design challenges of SMT architecture. (8)

(ii)Explain the Intel multicore architecture with its benefits. (8)

(or)

b) (i)Explain in detail about the CMP architecture and its performance. (8)

(ii) Explain the architecture of IBM cell processor with neat block diagram. (8)

Question Paper Code:21315

B.E./B.Tech. DEGREE EXAMINATION , MAY/JUNE 2013

Sixth Semester

Computer Science Engineering

CS2354/CS64/10144 CS604-ADVANCE COMPUTER ARCHITECTURE

(Regulation 2008/2010)

Time:Three Hours Maximum: 100 Marks

Answer All Questions

PART A-(10 x 2= 20 marks)

1. Define Dynamic Scheduling.
2. List the five levels of branch prediction.
3. Define loop-carried dependence.
4. What is the major disadvantage of supporting speculation hardware?
5. What is the disadvantage of using symmetric shared memory?
6. What is consistency?
7. What is cache miss and cache hit?
8. What is the bus master?
9. What are the categories of multiprocessors?
10. What is fine grained multithreading?

PART A-(5 x 16= 80 marks)

11.a)Explain the concept of ILP with various types of dependencies in ILP.

(or)

b)Explain how to reduce branch cost with dynamic hardware prediction.

12. a)Explain how hardware support for exposing more parallelism at compile time.

(or)

b)Explain how hardware based speculation is used to overcome control dependence.

13. a)Discuss about different models for memory consistency.

(or)

b)Define synchronization and explain the different mechanisms employed for synchronization among processors.

14. a)Explain the various levels of RAID.

(or)

b)Explain various ways to measure I/O Performance.

15. a)How is multithreading used to exploit thread level parallelism within a processor? Explain with example.

(or)

b)Discuss SMT and CMP architectures in detail.

CS2354/CS64/10144 CS604-ADVANCE COMPUTER ARCHITECTURE

(Regulation 2008)

Time:Three Hours Maximum: 100 Marks

Answer All Questions

PART A-(10 x 2= 20 marks)

1. Define temporal and spatial locality.
2. What are the major advantages of dynamic scheduling using Tomasulo's approach?
3. What is instruction level parallelism and what are the two major approaches for ILP?
4. What is the major disadvantage of supporting speculation hardware?
5. Define casual consistency memory approach.
6. What is loop unrolling and what are the major limitations of loop unrolling?
7. What is multiprocessor cache coherence problem?
8. What are the differences and similarities between SCSI and IDE?
9. What is meant by simultaneous multithreading?
10. What are the advantages of CMP architectures?

PART A-(5 x 16= 80 marks)

11. a)(i)Discuss about the guidelines and principles that are useful in the design and analysis of computers.(8)

 (ii) Explain dynamic branch prediction with an example. (8)

 (or)

 b) (i)Describe the major factors that influence the cost of a computer and how these factors are changing over time.(8)

 (ii) Explain hardware based speculation to overcome control dependencies.(8)

12. a)(i) Briefly compare CISC, RISC and VLIW. (6)

(ii) Describe the architecture of a typical superscalar VLIW processor with the help of block diagram. (10)

(or)

b)(i) Describe the basic compiler techniques for exploiting instruction level parallelism. (10)

(ii) Briefly compare hardware and software speculation mechanisms. (6)

13. a)(i) What is multi-threaded architecture and what are the advantages of multi-threaded architecture? (6)

(ii) Discuss about the synchronization techniques used in multiprocessor systems. (10)

(or)

b)(i)Explain the basic architecture of a symmetric multiprocessor system with the help of a block diagram. (10)

(ii) Describe the term coarse-grained and fine-grained multithreading. (6)

14.a)Describe various techniques for optimization of cache in detail. (16)

(or)

b)(i) Briefly describe standard RAID level in detail. (10)

(ii) Discuss about the issues in designing I/O system.(6)

15.a)(i) Describe various techniques for hardware multithreading in detail. (8)

(ii) Explain single chip multiprocessor architecture with the help of diagram. (8)

(or)

b)Discuss about the major challenges and issues in the design of multi-core architectures. (16)

CS2354/CS64/10144 CS604-ADVANCE COMPUTER ARCHITECTURE

(Regulation 2008)

Time:Three Hours Maximum: 100 Marks

Answer All Questions

PART A-(10 x 2= 20 marks)

1. Define Instruction level parallelism.
2. What are the advantages of using dynamic scheduling?
3. What are the advantages and disadvantages of trace scheduling method?
4. What are the limits on Instruction Level Parallelism?
5. What is Multiprocessor Cache Coherence?
6. Distinguish between fine-grained and coarse-grained multithreading.
7. Why do DRAMs generally have much larger capacities than SRAMs constructed in the same fabrication technology?
8. What is the average time to read or write a 512-byte sector for a disk?The advertised average seek time is 5 ms.the transfer rate is 40 MB/Second,it rotates at 10000 RPM,and the controller overhead is 0.1 ms,Assume the disk is idle so that there is no queuing delay.
9. What is Multi-Core Architecture?
10. What are the advantages of CMP architecture?

PART A-(5 x 16= 80 marks)

11. a)(i)Explain in Details about the various dependences caused in ILP. (8)

(ii)Explain the static and dynamic branch prediction schemes in detail (8)

(or)

b)(i) Explain the Tomasulo’s Approach used in dynamic scheduling for overcoming data hazards. (8)

(ii)Describe how the compiler technology can be used to improve the performance of instruction level parallelism

(8)

12. a)(i)Explain the software pipelining method used for uncover parallelism.

(8)

(ii)Compare the Hardware speculation with software speculation

(8)

(or)

b)Discuss the essential features of Intel IA-64 Architecture and Itanium Processor(16)

13. a)(i)Discuss the various cache-coherence protocols used in symmetric shared memory architecture.

(8)

(ii)What are the hardware primitives available to resolve synchronization issues in a multi-processor environment? Give examples.

(8)

(or)

b) (i)Discuss the performance of Symmetric Shared-Memory Multiprocessors for a multi-programmed workload consisting of both user activity and OS activity (8)

(ii)Discuss the various memory consistency models.

(8)

14. a)(i)Discuss the various techniques available for reducing cache miss penalty (8)

(ii)Briefly discuss the various levels of RAID

(8)

(or)

b)(i)Write short notes on compiler Optimizations to reduce the miss rate.

(8)

(ii)Explain the steps involved in the designing of an I/O system.
(8)

15. a)(i)DExplain in detail about SMT architecture and its challenges
(8)

(ii)Discuss in detail about heterogeneous multi-core processors.(8)

(or)

b)(i)Explain the CMP architecture in detail
(8)

(ii)Explain the IBM cell processor concept in detail
(8)