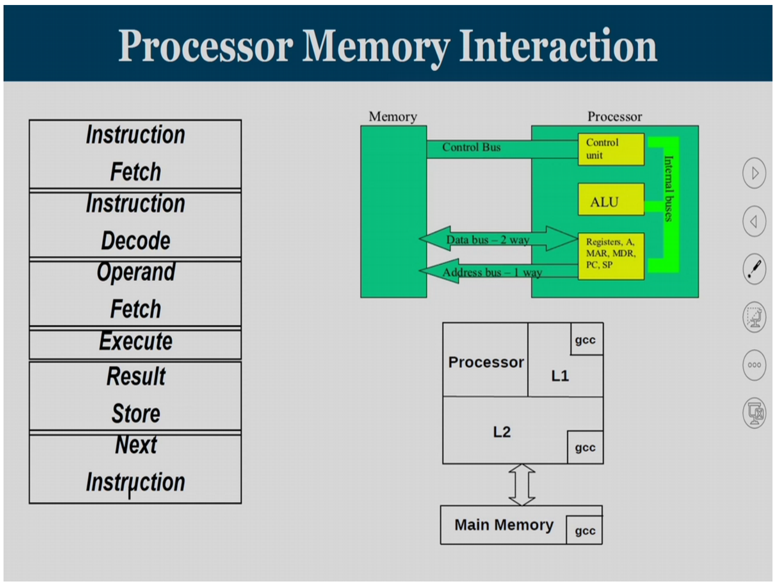
Multi-core Computer Architecture - Storage and Interconnects

Dr. John Jose
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture - 02
Instruction Execution Principles

Welcome to the first lecture of this course and today's lecture is titled as Instruction Execution Principles. This one lecture we will try to review the basic computer organization concepts that you have already learned, this will give a better understanding to appreciate the rest of the contents.

(Refer Slide Time: 00:56)



Processor Memory Interaction

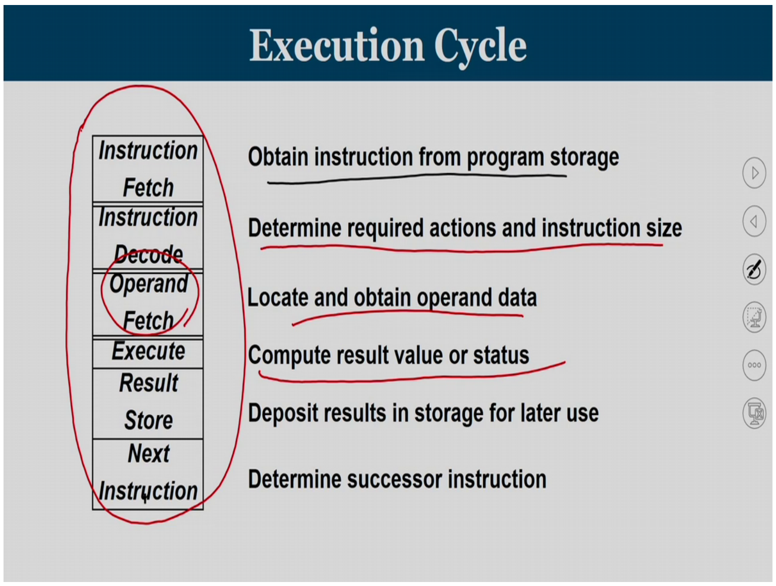
Instruction Fetch
Instruction Decode
Operand Fetch
Execute
Result Store
Next Instruction

So, this is the structure of a processor memory interaction, we have a processor and the processor has a control unit and then it has ALU it has a couple of registers and this is where is our primary memory.

And like we have mentioned in the introductory video, the task that the processor is going to run is going to be represented as a sequence of instructions and these instructions are going to be stored inside your memory. The instructions, which are already stored in the memory has to be fetched to the processor and we how to decode this instructions and carry out this task. This is called the instruction life cycle which consists of an instruction fetch operation, instruction decode, operand fetch, execute and then finding out the result and then you go and compute the next instruction and then the entire process is being repeated.

So, processers may have multiple levels of cache memory and, then there is a main memory upon requirement you fetch the corresponding instructions and data that are currently been used into your L 2 and L 1 caches this is how it is going to work.

(Refer Slide Time: 02:10)



Execution Cycle

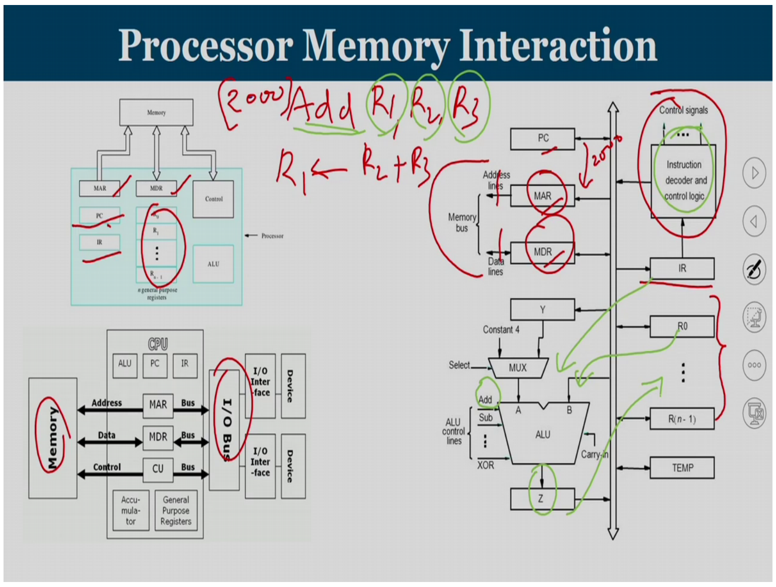
Instruction Fetch	Obtain instruction from program storage
Instruction Decode	Determine required actions and instruction size
Operand Fetch	Locate and obtain operand data
Execute	Compute result value or status
Result Store	Deposit results in storage for later use
Next Instruction	Determine successor instruction

Now, looking deeper into what happens these are the task or the subdivisions of the execution of an instruction. First obtain the instructions from the program storage and that is what is known as your instruction fetch. And then you have to find out, what are the actions to be done on this particular instruction that is called decoding of the instruction.

And then for every instruction there will be an opcode as well as a couple of operands. Now the data is located inside this operand, it can be inside registers or it can be inside memory. So, bringing them that is known as operand fetch, locate and obtain the operand data. Once you get the data when you are fit to execute the task and that is what is called compute the result value. Then once you obtain this results you can put it inside the storage for later use and then you have to find out the successor instruction.

For normal straight line sequencing the very next instruction is going to be fetched and executed, if it is a branch instruction it can be either a conditional instruction or an unconditional instruction, if it is an unconditional instruction go to the target instruction, if it is conditional instruction go to find out certain status flags and based upon whether the flag is set or reset we have to find out which is going to be the next instruction. So, every instruction will be going through this particular sequence of operations and a processor helps in doing this task.

(Refer Slide Time: 03:46)



Processor Memory Interaction

Now, we will see what is inside a generic microprocessor, we have a couple of registers that is available that we call it as general purpose registers, there are two important registers one is called the MAR the other one is called MDR memory address register and memory data register these two registers are going to interact with your memory. And then we have a control unit which controls the entire operation.

A special register which triggers the action is called the program counter, it contains the address of the next instruction to be fetched from memory, and then we have the instruction register once you bring and instruction from memory that we will be going into the instruction register and from the instruction register we are going to decode and find out what is a corresponding action to be done.

This is saying with whether the operation is to be carried out from memory or the operation is going to be carried out on the IO. So, MAR and MDR are the two interface registers which a processor is having with respect to the memory system. Now, looking deeper into what is inside a processor, this is what you see here is the register organization inside a processor, we have a program counter, we have memory address register memory data register and we have the general purpose registers that is R 0 to R n. And then we have the control logic that takes care of the activity here, we have an ALU so, the data is taken from the general purpose registers will comes to ALU produce the result.

Now, these are the two registers MAR and MDR that are important in interfacing with memory, when you want to access a memory location, whether it is for a load operation or whether it is for a store operation. The address of the location has to be put inside MAR. And once you give the control signal whether it is a read or a write signal that data that is either fetched from the memory, or to be written in the memory location will be interfaced through MDR. So, the address portion will be connected to the MAR and the data portion will be arriving through MDR. If the address bus that is starting from this is address bus that starts from MAR it is unidirectional from the processor to the memory and the data bus is bidirectional that is connected to memory.

So, suppose if you have an operation in which you how to fetch an instruction and, then let us say the instruction is going to be add R 1 R 2 and R 3, let us write this instruction add R 1 R 2 and R 3, let us say this means R 1 is R 2 plus R 3. If this particular instruction is going to be stored inside memory, then first I have to find out in which address this instruction stored let us say this address is going to be 2000.

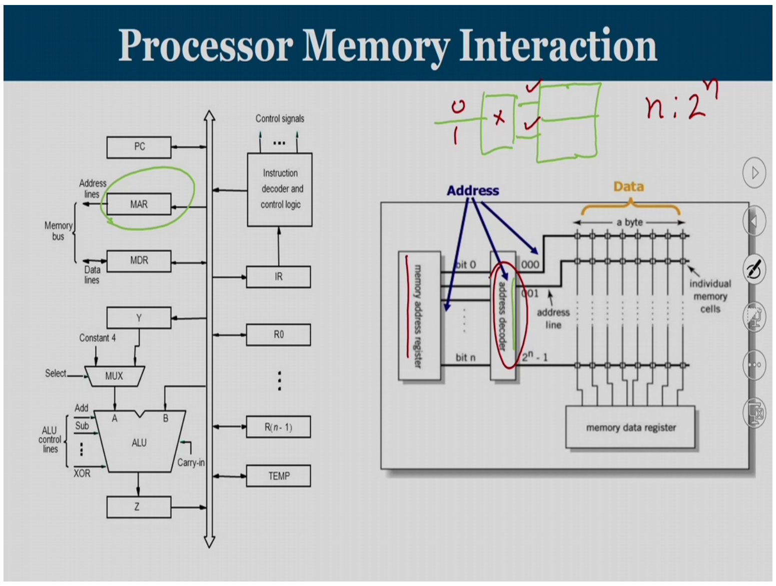
So, your program counter is going to provide this value 2000 into MAR, once it reaches MAR then through the address bus the location 2000 is enabled, the contents of location 2000 are brought to MDR and it will be reaching IR upon decoding. So, certain instructions are single word instruction some instructions will be multi word instructions.

For simplicity now we assume this add R 1 R 2 and R 3 will reach MDR in one fetch in one clock cycle, upon going to IR we are going to decode it we try to understand yeah this is going to be a opcode is add and this is my first operant, second operant and third operant and the control unit knows that the second operands condense is to be added with the third operands condense and you are going to put it insight the first operand.

So, the necessary control signals will be generated from the control logic and you get the data like R 2s data will reach ALU, R 3s data also will reach ALU, ALU has two inputs the corresponding add signal will be activated, you get the result in is it that is a temporary register and then it is been stored back.

Similarly there exist these kind of smaller micro architectural steps for every instruction, let it be a load instruction, or it can be a stored instruction, or it can be a an ALU instruction it can be a branch instruction this is the way how memory and processors are interfaced. In short MAR and MDR are two crucial registers that are going to interface with the memory.

(Refer Slide Time: 08:53)



Processor Memory Interaction

Now, this is the memory address register what we are talking about, the contents of memory address register is being passing through the address bus and it reaches memory. In memory the first unit that is going to interface is which is called an address decoder, depending on the bits that this decoder receives, one of its output location is going to be activated. For example, let us assume a simple memory location with two words. So, you have a decoder the decoder has one input and this is connected to two of this output.

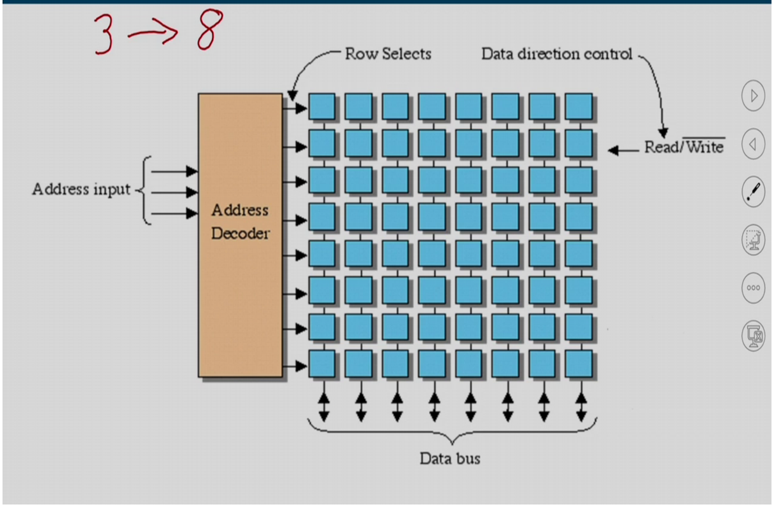
So, if the value is 0 then this will be selected rather than that if the value is going to be one then this is going to be selected. So, when you have more number of locations the number of bits that the processor is sending will increase from 1 bit to 2 bit 3 bit like that. So, the decoder that you are going to use is typically n is to 2 power n decoder if you have 4 inputs 16 outputs. So, using four bits in the address one among 16 locations can be uniquely addressed. So, depending on the size of your MAR let us say your MAR is going to be 32 bits; that means, 2 power 32 address locations can be uniquely addressed and the decoder is basically 32 bit will be there in its input that is coming from MAR and the output will be 2 power 32.

Depending on these 32 bits one among the 2 power 32 locations will be uniquely selected and that is going to be your action point for memory. And once that is selected depending on whether it is a read operation or write operation either data flows from memory to MDR or data flows from MDR to memory, if it is a load operation then after selecting your memory location using MAR, let us say the location 2000 the condense of 2000 will be transferred to MDR; that means, it reaches processor that is called load.

Now, if you wanted to write something to location 2000 put the appropriate bits in MAR we were right signal and put your data inside MDR. So, condense of MDR will get transferred to the memory location, so that is the role of MAR and MDR in this context. So, depending on the increase in size of MAR the address decoders specification also changes.

(Refer Slide Time: 11:39)

How memory works ?

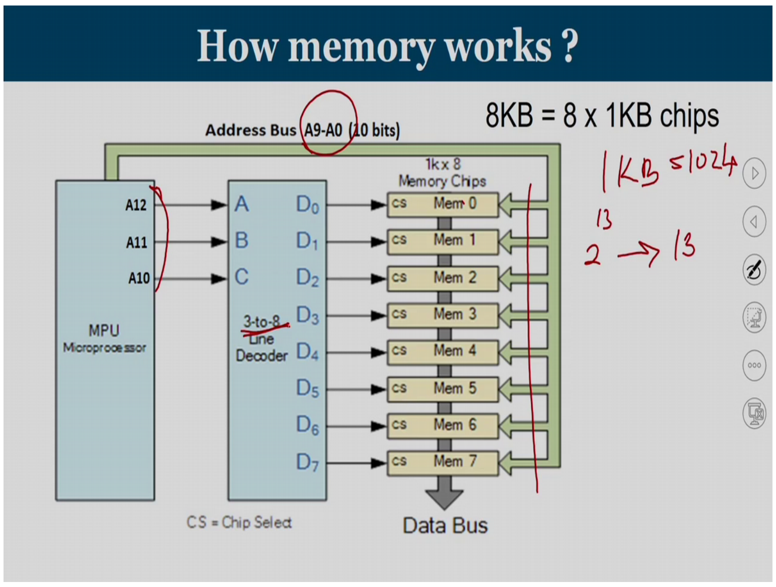


Now, consider this case where you have three bit address. So, using three bit address you can uniquely access 8 locations 2 power 3 equal to 8, if you give 0 0 0 as the input then this location will be selected.

Similarly if you are going to put 1 1 1 as the input, then this will be selected. Similarly depending on the various combinations that you have any of the memory location can be uniquely selected. So, it is the address bits that the processor give will reach MAR, from MAR it is going to reach the corresponding memory the address decoder will take the corresponding bits and activate one and only one of its output, and that is what the unique memory that is been chosen. Now, in this case we have 8 memory locations so, this is basically your address bus here and this is basically your data bus.

And in modern processors multiple memory location condense are compacted together to connected to the database we will see the details once we learn about cache memory topics. This is the way how you connect multiple memory organizations together.

(Refer Slide Time: 13:15)



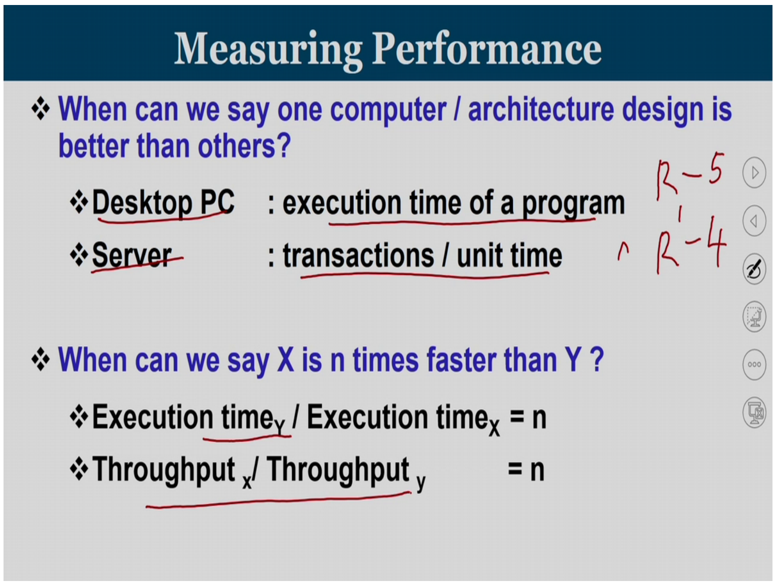
Consider the case that you have memory locations of capacity 1 K byte, each of this one is going to be 1 KB. What does it mean? There are 1024 memory locations and each location can store 8 bit of information or each location is basically 1 byte. Now, to access one unique location from these 1024 locations, we have to give 10 bits. Based upon these 10 bits one among this 1024 locations will be uniquely selected. But I have a requirement of 8 KB, I am not happy with 1 KB memory, in this case I am looking for an 8 KB memory.

But all the memory chips that is available is only of 1 KB size so, I am going to organize this 1 KB into 8, 8 1 KB chips are organized. Now, I have to interface this thing with the processor, when I have 8 KB 8 KB is nothing, but 2 power 13 so, I require 13 bits to come from processor in order to uniquely identify 1 location out of 8 KB locations.

But since I have done an internal layout organization of 1 K into 8 chips, I have 8 chips organized the lower 10 bits are connected to all the chips the higher order 3 bits that is A 10 A 11 and A 12, they will choose one among this chip. So, the lower 10 bits will go one of the 1024 locations within a chip and the higher three bits will determine. So, this is going to a decoder it is a 3 to 8 decoder, it will choose one among this chip.

This is the way how a larger memory is being address; similarly you can still go for larger memory locations by combining multiple smaller memory units.

(Refer Slide Time: 15:39)



Measuring Performance

- When can we say one computer / architecture design is better than others?
  - Desktop PC : execution time of a program
  - Server : transactions / unit time

- When can we say X is n times faster than Y ?
  - Execution time$_Y$ / Execution time$_X$ = n
  - Throughput $_X$/ Throughput $_Y$ = n

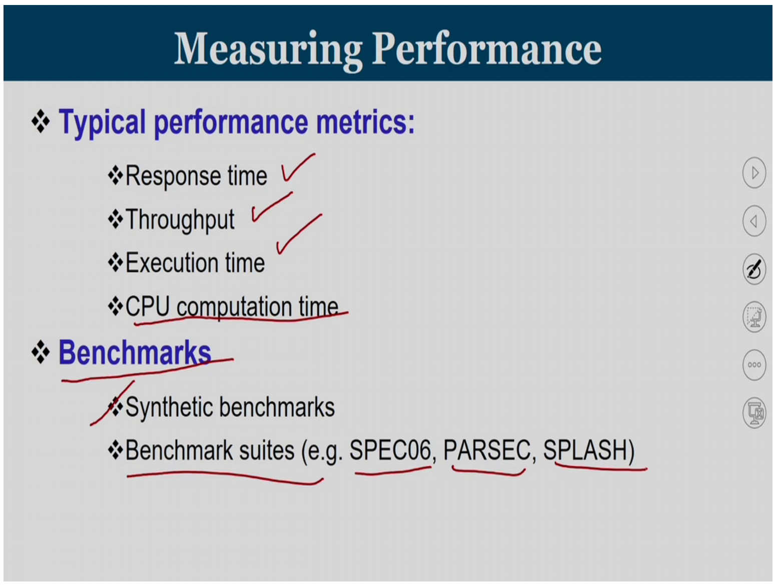
Now, when can we say that one computer architecture or design or one processor is better than others, we have to use certain performance metric, one kind of performance metric that is being typically used in a desktop PC is what is a execution time of a program, considered two architectures let us say R is 1 processor R dash is the new processor. Let us say you are going to take 5 unit of time to complete a program in R and you are going to take only 4 unit of time, if the same program is executed in R dash.

This means R dash is faster than R, this is one metric we are trying to find out what is execution time of a program it is a most commonly used metric. So, one program you are going to run in one machine try to find out, what is the execution time the same program is executed in another machine to find out what is the execution time over there. So, larger the execution time; that means, a particular machine is slower. When it comes to high end computing device like servers, then it is not the execution time because you get request from multiple devices for multiple clients.

So, in this context execution time may not be the right metric to assess a system, in that case transactions per unit time, number of task number of responses I can give in a given unit time, let us say one server is capable of responding to 10 client request in a given time. Another server if it can respond to 15 client request then the second server is considered to be having a better performance than the first one. So, depending on the context in the type in the case of desktop machines, then execution time will be a good metric, when it comes to server machines then the number of transactions per unit time will be a more better metric to represent the performance.

When can we say that a machine X is n times faster than machine Y the common metric that we can use is the execution time ratio, when the execution time of Y is n times that of execution time of X, or the ratio of execution time of Y to the execution time of X will be n, is can also be represented in the hands of a throughput, throughput of x divided by throughput of y can also be represented as n.

(Refer Slide Time: 18:16)



Measuring Performance

- Typical performance metrics:
  - Response time ✓
  - Throughput ✓
  - Execution time ✓
  - CPU computation time
- Benchmarks
  - Synthetic benchmarks
  - Benchmark suites (e.g. SPEC06, PARSEC, SPLASH)

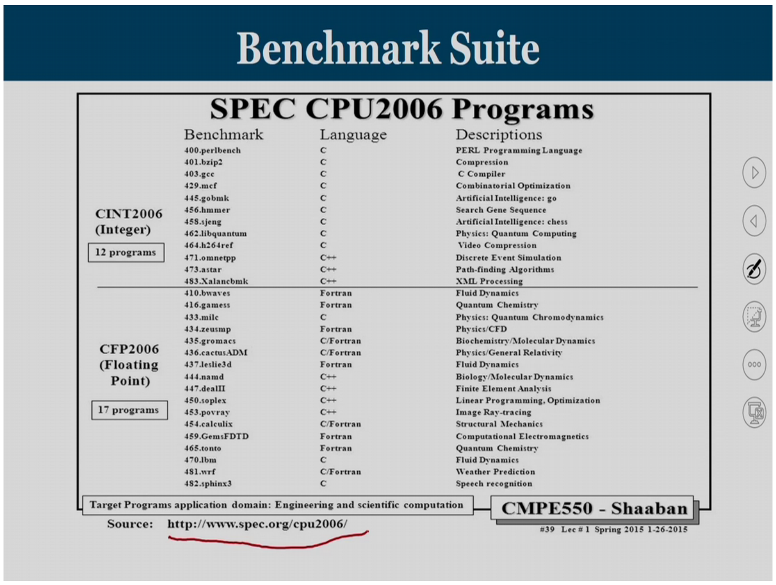
These are the typical performance metrics we can measure in terms of response time, to certain devices ones you give the data when will it respond, or once you give the command by what time it is going to respond with the data.

In certain cases throughput number of task completed in unit time it can be a good performance metrics in some other time it is execution time that is important, in certain task there may be certain time it is going to execute with the CPU sometimes some IO operations need to be there sometimes, we are only bothered about what is going to be the execution time. So, in that case CPU computation time is going to be a more bottom 1.

Now, once you have a processor in hand how will you check whether my processor is better, one way of looking at it to find out execution time, I can always find out a special program which will take less amount of time in my architecture, when compared to the baseline architecture, that need not be true for a different program for a program a my machine maybe better, for a program b the base line machine maybe better.

So, there is a common consensus in computer architecture community how can you rate and that is what is called benchmarks, benchmarks are standard programs and when you run this standard programs on a machine a and a machine b, then you may be able to find out rather than using your own programs, it is always better to rate an architecture with respect to this standard benchmarks programs available. Benchmarks can be classified into synthetic benchmarks which are artificial programs, which wont represent any kind of a real application is are artificial programs, which will create fetch decode and execution effect or we can use real benchmark suits like SPEC 2006 PARSEC and SPLASH, these are some of the commonly used benchmarks in architecture community.

(Refer Slide Time: 20:21)



Benchmark Suite

SPEC CPU2006 Programs

	Benchmark	Language	Descriptions
CINT2006 (Integer) 12 programs	400.perlbench	C	PERL Programming Language
	401.bzip2	C	Compression
	403.gcc	C	C Compiler
	429.mcf	C	Combinatorial Optimization
	445.gobmk	C	Artificial Intelligence: go
	456.hmmer	C	Search Gene Sequence
	458.sjeng	C	Artificial Intelligence: chess
	462.libquantum	C	Physics: Quantum Computing
	464.h264ref	C	Video Compression
	471.omnetpp	C++	Discrete Event Simulation
	473.astar	C++	Path-finding Algorithms
	483.Xalancbmk	C++	XML Processing
CFP2006 (Floating Point) 17 programs	410.bwaves	Fortran	Fluid Dynamics
	416.gamess	Fortran	Quantum Chemistry
	433.milc	C	Physics: Quantum Chromodynamics
	434.zeusmp	Fortran	Physics/CFD
	435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
	436.cactusADM	C/Fortran	Physics/General Relativity
	437.leslie3d	Fortran	Fluid Dynamics
	444.namd	C++	Biology/Molecular Dynamics
	447.dealII	C++	Finite Element Analysis
	450.soplex	C++	Linear Programming, Optimization
	453.povray	C++	Image Ray-tracing
	454.calculix	C/Fortran	Structural Mechanics
	459.GemsFDTD	Fortran	Computational Electromagnetics
	465.tonto	Fortran	Quantum Chemistry
	470.lbm	C	Fluid Dynamics
	481.wrf	C/Fortran	Weather Prediction
	482.sphinx3	C	Speech recognition

Target Programs application domain: Engineering and scientific computation

Source: http://www.spec.org/cpu2006/

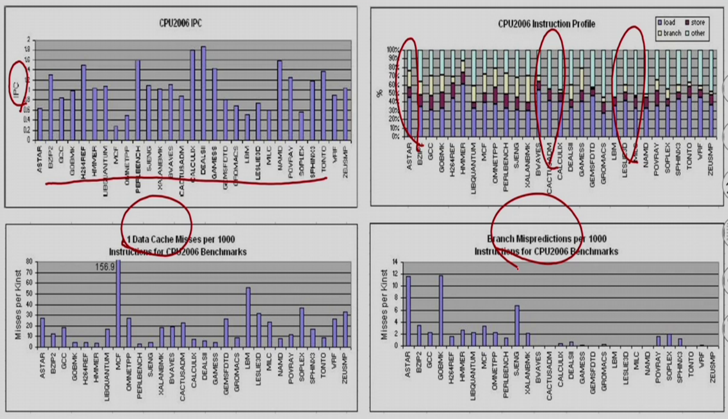
CMPE550 - Shaaban

#39 Lec # 1 Spring 2015 1-26-2015

Now, this is the details of SPEC 2006 we can find the details from the SPEC website, these are the SPEC programs SPEC 2006 programs.

(Refer Slide Time: 20:27)

Benchmark based evaluation

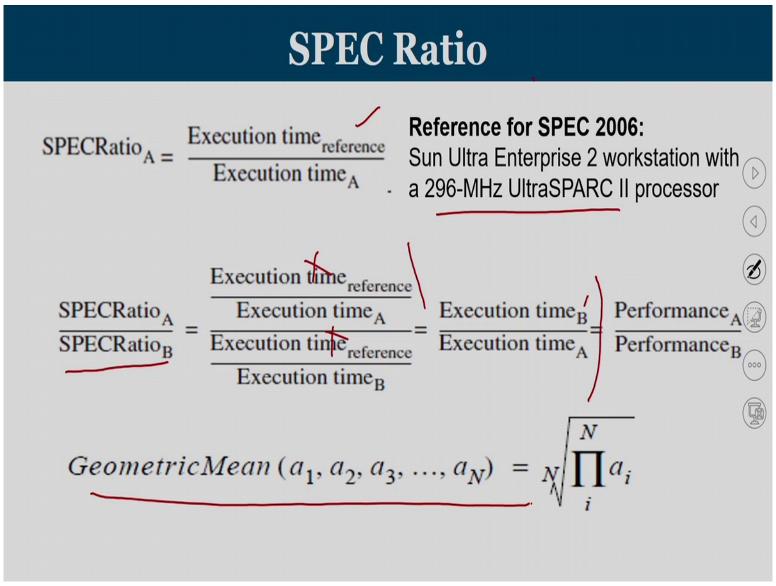


Now, by using architectural simulators we are going to learn of the simulator gem 5 in our course the simulator will help you in finding out lot of internal characteristics of a program, we have heard that there are different SPEC programs. Now, this particular chart is finding out IPC what is the number of instructions that is completed per cycle, for various benchmarks the number of instruction the Y axis is the IPC value, what is the number of instruction that I could complete.

Similarly, I wanted to know what is the percentage mix of load instruction, the percentage mix of ALU instruction branch instruction like that architecture or some command over architectural simulators will help you in finding out this profile for example, we can see 40 percent of them are load instruction 20 percent of them are store instruction like that for, what is this split up of the of the instruction.

Similarly for each benchmark it may vary. So, once you know the split up of the instruction, it will help you in designing architectures this is basically called workload characterization. Similarly I can find out the number of cache misses that each of these benchmarks are encountered, similarly the number of misprediction that the branch predictor is doing. So, we are going to find out what is the significance of these numbers in terms of designing an architecture. Now, what is going to be SPEC ratio, SPEC ratio is yet another important performance parameter or performance metric, which will help us to assess how faster and architecture is.

(Refer Slide Time: 22:18)



SPEC Ratio

$$\text{SPECRatio}_A = \frac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}$$

Reference for SPEC 2006:
Sun Ultra Enterprise 2 workstation with a 296-MHz UltraSPARC II processor

$$\frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_B}} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

$$GeometricMean(a_1, a_2, a_3, \ldots, a_N) = \sqrt[N]{\prod_{i}^{N} a_i}$$

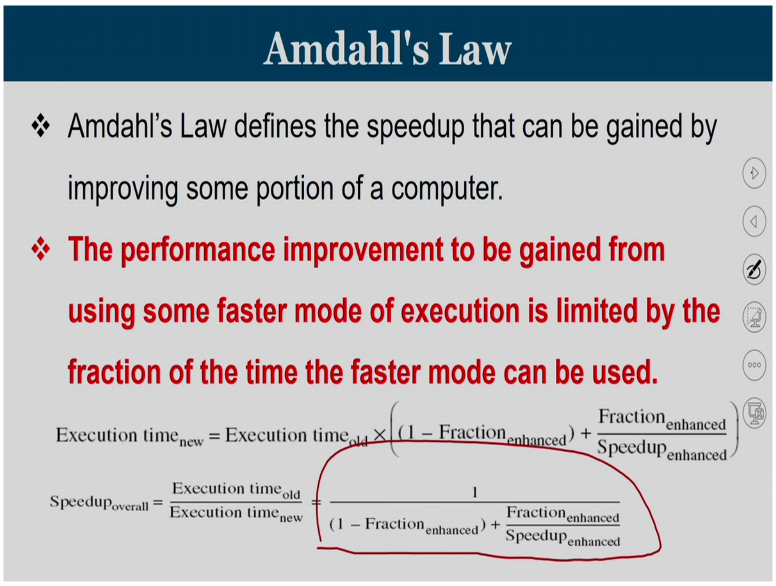
Now, SPEC ratio is defined as the execution time of a program in a reference machine divided by execution time of a program in your machine A. So, for example, we wanted to know what is a SPEC ratio of a given computer architecture with so and so, processor clock cycle with so and so, cache memory specifications and all.

Now, what is the reference machine for SPECs 2006 the reference machine is sun ultra enterprise 2 workstation with a 209 megahertz ultra spark 2 processor. Similarly, every benchmark will be having its own reference. So, once you run you have to run in a program on this machine get the number that is called execution time on the reference machine and, then you have to execute the same thing in your machine and that is going to be the difference.

Now, if you wanted to know what is the difference when I run on machine A and on machine B, then we have to find out SPEC ratio of machine A and SPEC ratio of machine B its nothing, but the execution time on reference divided by execution time on A the whole divided by execution time on reference divided by execution time on B. The execution time on reference will get cancelled, essentially you get execution time on B by execution time on A, it is also known as performance of A divided by performance of B.

This will help us to understand whether a program will be faster on machine A, or it is faster on machine B. And when you have multiple benchmarks for example, A B C D and E for program A architecture A will be good or for example, R 1 will be good, for program B R 2 may be good for program C, again R 1 maybe good where R 1 R 2 R 3 are different architectures. So, when you have multiple programs and the SPEC ratio is varying for these programs, then how will you summarize what is the performance, typically we use geometric mean find out the SPEC ratios of all these across different benchmarks and find out the geometric mean of this.

(Refer Slide Time: 24:35)



Amdahl's Law

- ❖ Amdahl's Law defines the speedup that can be gained by improving some portion of a computer.
- ❖ **The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.**

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

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

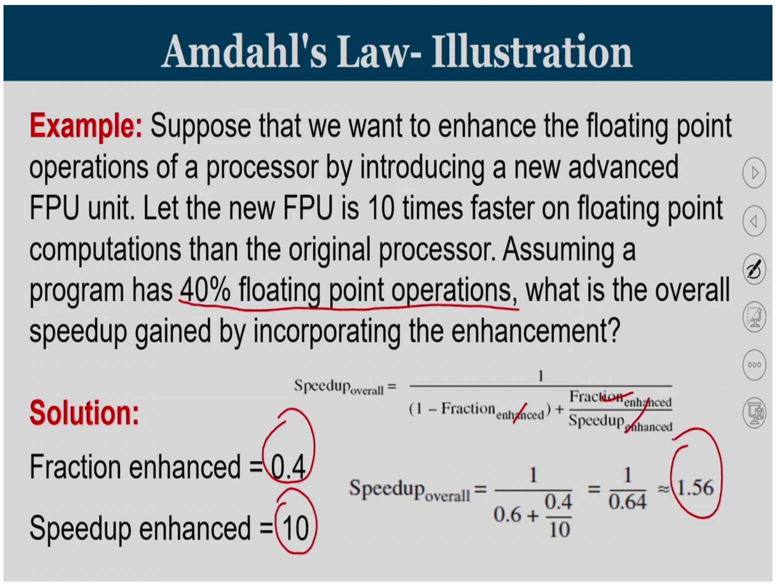
Now, we will see yet another important parameter, or important law that gives intuition about how much speed up that you can obtain, if you increase the units if you improve the performance of a unit and this is abbreviated as Amdahl's law.

Amdahl's law defines the speed up that can be gained by improving some portion of a computer or an architecture. The performance improvement to be gained from using some faster mode of execution is limited by the fraction of time; the faster mode can be used. For example, consider the case that you are having a unicore processor you are going to run a program. Now, I have bought another processor which is having two pores two processing units, we migrating y program from an architecture which is having a single processor to an architecture which is having dual processor, will it improve performance.

It strictly depends on how many instructions of a given program can be a run parallelly in these two, if two processors or this dual processor. To get a deeper intuition on this if every instruction of a given program is dependent on the previous instruction, then we cannot run them parallelly. So, even though we use a dual core processor, or a quad core processor or a chip with 100 processors you cannot improve performance. So, the amount of performance gain that you get, when you migrate from a single core processor to a multi core processor is restricted to the amount of parallelism that is available in a program. So, certain programs maybe very fast, because the program has inherently parallelism that is available.

Certain other programs may not be getting the same kind of speed up, because it may be having more of a sequential nature between its instructions. So, the execution time if you use an enhanced mode is given by execution time in the older mode into 1 minus fraction enhanced plus fraction enhanced by speed up enhanced. Or the overall speed up that you are going to gain is the ratio of these two things, execution time of old by execution time of new it is 1 by 1 minus fraction, enhanced plus fraction enhanced by speed up enhanced. Amdahl's law is driven by this equation.

(Refer Slide Time: 27:22)



Amdahl's Law- Illustration

Example: Suppose that we want to enhance the floating point operations of a processor by introducing a new advanced FPU unit. Let the new FPU is 10 times faster on floating point computations than the original processor. Assuming a program has 40% floating point operations, what is the overall speedup gained by incorporating the enhancement?

Solution:

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

Fraction enhanced = 0.4

Speedup enhanced = 10

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$$

Now, we will have a small illustration to understand Amdahl's law it will more deeper consider, this example suppose that we wanted to enhance the floating point operation of a processor by introducing a new advanced floating point unit. Let the new floating point unit is 10 times faster on floating point computations than original processor.

Assuming a program has 40 percent floating point operation, what is a overall speedup gained by incorporating this enhancement to summarize this problem, we are going to take the case of a processor that has a floating point unit in that and the floating point unit is responsible for execution of floating point instructions.

Now, whether we use a sophisticated floating point unit, or a normal floating point unit, it is not going to have any impact, if the program that you are using is not having any floating point instructions at all. Your floating point unit will be used only when there is a floating point instruction, if you are using or if you are executing a program which is not having not even a single floating point instruction, then the floating when the efficiency of the floating point unit is not go into bother at all.

Suppose out of 100 instructions, if 10 instructions of floating point instruction, then if you use a better floating point unit this 10 instructions will get benefit. If 100 percent of your instructions are floating point unit, then you will get more benefit, because the percentage of instruction that are going to be executed on the enhanced unit, in this case it is a floating point unit is going to have a significant say Amdahl's law is basically summarizing around this factor. The amount of performance improvement you are going to head is restricted to the percentage of instructions that are going to use the enhanced unit.

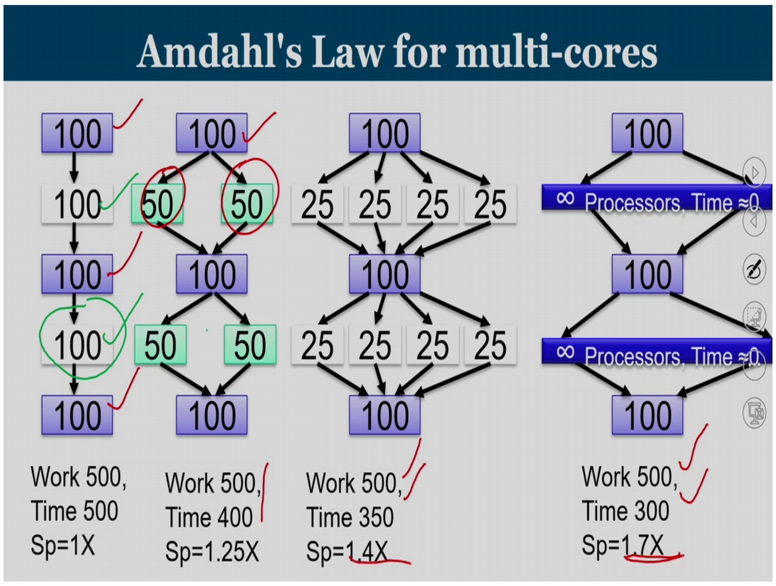
Now, in this case you have to identify certain things solution is this is the equation that, we are going to use the overall speed up that you are going to get is defined by 1 by 1 minus fraction enhanced plus fraction enhanced by speed up enhanced, what is the fraction enhanced. Now, the fraction enhanced portion is how will you find out the fraction enhanced portion from the question, how much percentage of my program are floating point instruction only floating point instruction is going to get benefit with the new floating point unit we are going to use.

So, we have forty percent of our instructions are floating point operation, that is why fraction enhanced is 0.4. Now, how much speed up you get the floating point unit alone will improve performance by 10 times on what, on floating point instructions. So, if you use an advanced floating point unit every floating point instruction will get the speed up of 10, that does not means your overall speed up is 10 overall speed up is much lower than that Amdahl’s law is going to correlate.

Now, we have to substitute this value we have fraction enhanced that is available, we have speed up enhanced that is also available and when you substitute, we will get this value 1 by 1 minus fraction enhanced that is 0.6 plus fraction enhanced by speed up enhanced that is 0.4 divided by 10. So, you will get a overall speed up of only 1.56 times assume that rather than this 40 percent. If you have 90 percent of your instructions are floating point unit, then this number is going to significantly increase, this is the illustration of an Amdahl’s law example.

Now, since our course is all about multi core computer architecture, the question of are we going to get an improved performance, if you use a multi core machine, how will we answer that only if the programs that you are going to run, if the programs have inherent parallelism in it then it can be a parallelly executed across multiple machines.

(Refer Slide Time: 32:04)



Consider the case that you have 500 units of work, you are going to work execute this task on a unicore machine, there are five different phases the first 100 can be done only by 1 processor.

The next 100 can be parallelized, but in this case we have only 1 processor that is available. So, wherever you can see this blue shade these are the instructions that can be run only on a single processor; that means, all this instructions are dependent one after another whereas, whatever you see in this green color these can be parallelized. So, I have my first one 100 instructions that can be run only on one machine, next 100 instructions can be parallelized, next 100 instruction again it can be run only on a single machine this one will be only that can be parallelized and the last 100 have to be run in the same machine.

So, here we have total of 500 instructions and you assume that each instruction will take one unit of time, then 500 units of time will be taken the speed up is 1 X. Now, consider the scenario where you are going to use two processors, when you use two processor still the first 100 of them, how to surely rum sequential even though you have two cores, one core has to be idle when this first 100 instructions are executed.

The next 100 instructions can be parallelized 50 will go to one processor and, next 50 will go to another processor again. So, here you can have parallelism even though you have 100 instruction this get over by 50 units of time. Again the sequential portion will come this 100, again we have a parallel portion that can be splitted up. So, now, if you see there are 500 instructions, but the total time that is taken is 400, we have a 100 here and we have 250 splitters. So, 300 instructions will be running only on a single core machine the remaining 200, you are able to get the benefit of parallelism.

Consider the next case you are going to have four processors now, when you have four processors then whatever 100 instructions you have that will be splitted at across, for processors each will take 25 instructions. So, the execution time is also 25 whereas, you can see that the sequential portion still remain intact it can be executed only on one processor.

Still total of 500 instructions are executed total time is 350 so, the speed up is 1.4 X. Now, you assume let us say you have infinite number of processors, then this portion will become 0, it is a theoretical observation still you have total of 500 units of work, that is completing in 300 units of time the speed up that you get is 1.7.

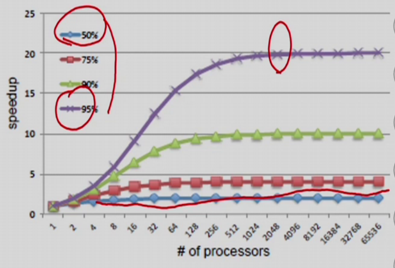
So, for this particular program which has 60 percent of it is code, that can run only on a sequential processor, or like that can be run only on a single processor the maximum speed up that, you can get that you are permitted to use any number of processors its only up to 1.7 X.

(Refer Slide Time: 35:45)

How much Speed up you can achieve ?

Amdahl's Law:

$$Speedup = \frac{1}{(1-\alpha)+\frac{\alpha}{n}}$$

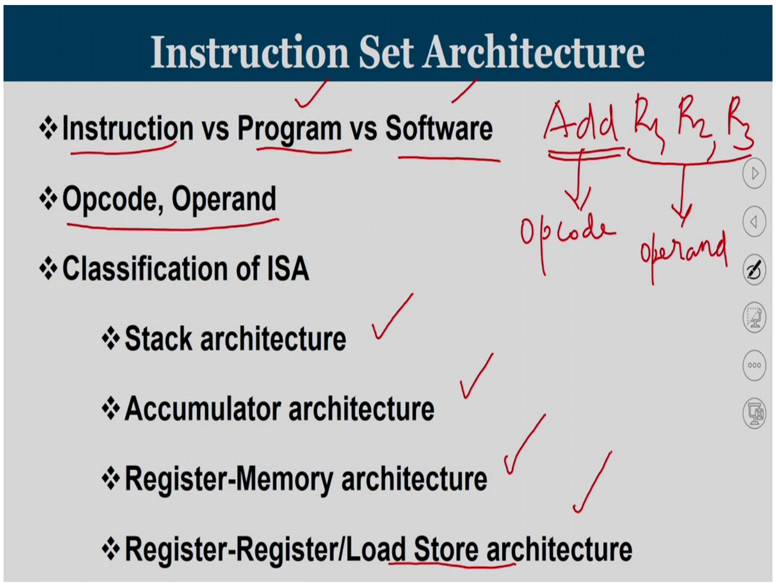


So, how much speed up that you can achieve the Amdahl's law say that speed up is equal to 1 by 1 minus alpha plus alpha by n, where alpha is the fraction then n is the speed up or the number of processors that you are going to use.

This graph shows the y axis shows the speed up the x axis shows the number of processors, and this shows what is the percentage of parallelism that is available. So, if you have 95 percent of parallelism available, you are able to get better speed up that is being shown by this violet graph. If you have only 50 percent of parallelism that is available in your program you can see that, even if you increase the number of processing cores you are not going to get much benefit in terms of speed up.

So, the conclusion is the amount of parallelism that is available in the task, or the software's that you are going to run is an important parameter that will tell you how much performance improvement that you get, if you migrate from unicore processors to multi core processors. So, the rest of the study of multi core computer architecture should be built on top of this law. Now, we will see some fundamentals about instruction set architecture what is the difference between an instruction program and a software.

(Refer Slide Time: 36:59)



Instruction Set Architecture

- ❖ Instruction vs Program vs Software
- ❖ Opcode, Operand
- ❖ Classification of ISA
    - ❖ Stack architecture
    - ❖ Accumulator architecture
    - ❖ Register-Memory architecture
    - ❖ Register-Register/Load Store architecture

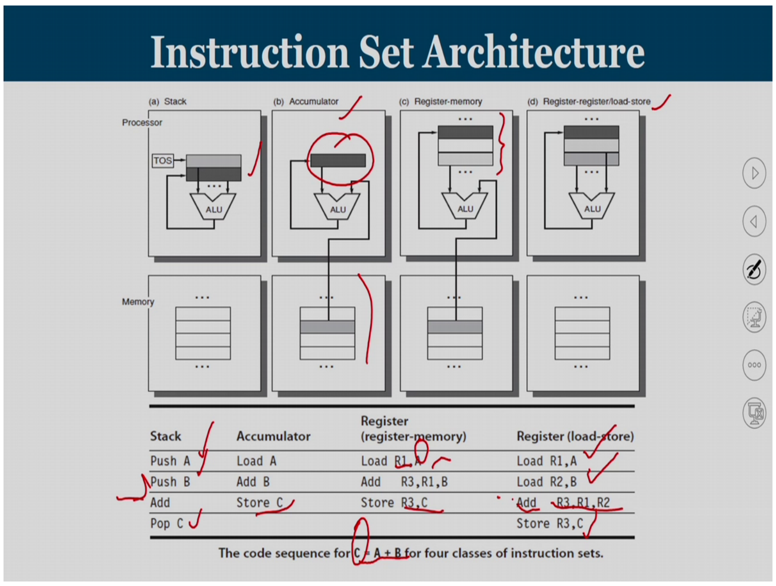
Add R1 R2 R3
Opcode
Operand

So, instruction is a basic machine task which consists of an opcode and an operand. Multiple instructions combine together perform a program and collection of programs is what we known as a software. Now, what is opcode and operand an instruction consists of an opcode and operand, opcode means it refers to the operation to be done.

So, when you look at there instruction what is to be done that is called the opcode portion. Consider this example add R 1 R 2 and R 3 here add will specifies, what is the operation to be done. So, add is basically your opcode and where the operations have to be done, this have to be done in these registers.

So, this registers are the operand sometimes the operand will be in memory, we can tell memory address sometimes it can be the registers. Now, different microprocessors have different instruction set architecture, we can broadly classify instruction set architectures into four, stack architecture, accumulator architecture register memory architecture and register, register architecture also known as load store architecture.

(Refer Slide Time: 38:25)



Instruction Set Architecture

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

The code sequence for C = A + B for four classes of instruction sets.

This diagram will show you what is the difference between these four architectures, in the case of as a stack architecture ALU can take operands only from the stack. So, it will take the data from the stack process it and push the result back to the stack, consider the case of a code C equal to A plus B. Let us try to see how this c equal to A plus B is been operated on a stack architecture.

First you push the value to the stack push the second operand to stack, then you call the operation add. Once you call an operation add you know that for adding it requires minimum of two operands, it will take the operand from the top of the stack and, once you take the element then the top will be adjusted go to the next top. So, top and top minus one will be your operands that is what is been seen top and top minus 1 is been taken there and you store the result back that is call pop C. So, you will get the result back from this stack.

When you go for accumulator architecture 1 operand has to be there in accumulator, there is no concept of stack we have a special register that is called accumulator one of the operand has to be there in the accumulator. And the second operand, you can directly take from memory, what do you see here is memory. So, load the first operand this will go to the accumulator and, then you have to perform the add operations. So, add b means 1 operand is always by default accumulator and second operand is taken from memory. So, B is directly taken from memory.

Once the result is added the result will be available in the accumulator take back the result. So, by storing C you will get back the result, I will summarize the accumulator again in the case of an accumulator architecture, one of the operand in the case of an ALU operation one of the operand has to be present inside the accumulator. And second operand can be directly taken from in the memory, where you specify the operation like add the operand that you specify is the memory location the second operand is in is already the implicit it is available inside accumulator.

So, you load the first value the first operand into accumulator keep it ready, then you perform the add operation by specifying the second operand which is a memory operand. So, add B means contents of accumulator is added with contents of memory location B. And the result is now inside accumulator. Now, if you want to get the result back to location C, then you have to perform a store operation. Now the third category is register memory operation, then in this case you have a set of registers general purpose registers, you can load your data into the registers upon requirement and then you can transfer it across memory.

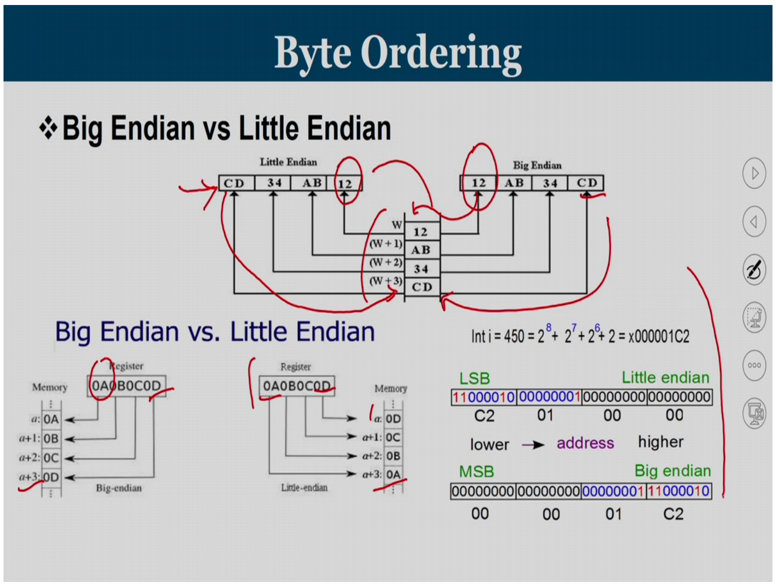
So, load R 1 A the contents of A is copied to the register R 1, then you can say add R 3 R 1 B; that means, contents of R 1 is added with contents of memory location B and the result is available in register R 3 since you want the final result in C you can store it back to C, yet another category is called load store architecture, here the peculiarity is any ALU operation can be performed only on the registers.

So, prior to performing an ALU operation you have to make sure that the data that is already available inside the memory, has to be copied into the registers. So, load the first operand into register R 1, load the second operand into register R 2, add them together such that you get the result in R 3 and then you copy the result back into C.

So, you have a stack architecture where operands has to be in the stack prior to the ALU operation, you have to push the operands into stack. Once the result is obtained you have to pop the result back from the stack, when it comes to accumulator architecture one of the operand has to be there in the accumulator second operand can be there from memory, in the case of register memory architecture, you have a set of general purpose registers move your data from memory into this registers, when you perform an ALU operation one of the operand has to be there in register the other operand can be there in memory.

But both the operands in memory is not generally possible in register memory architectures. And the last category is load store architecture, where whenever you perform an operation, then all the operands has to be there in registers.

(Refer Slide Time: 43:23)



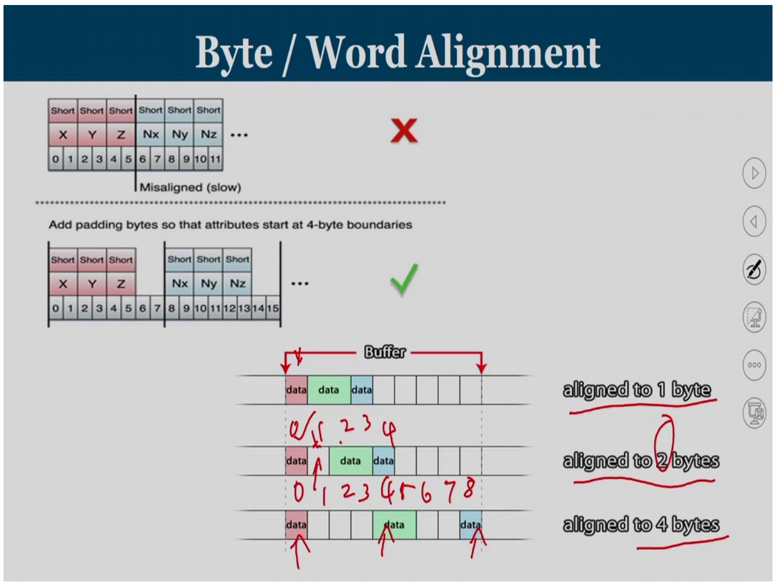
Another important area which we need to focus is about the way how you store multi word data, if your memory is in terms of bytes. Let us say you wanted to store a 4 byte information, how will you store it, there are basically two organizations in which we can store a long data in a memory which consists of sequence of words, one is called Little Endian format, the other one is called Big Endian format.

Consider the case you have to store this hexadecimal value, this is a 32 bit value CD is 8 bit, 3 4 is another 8 bit A B is 8 bit and 1 2 is 8 bit, when I store this in little endian format what do you store is the 12 in this case, that is the least significant bits will be stored in the lower address. Let us say to store this we require 4 bytes and let us called address of this 4 bytes as W, W plus 1 w plus 2 and W plus 3. The least significant bits will be stored in the lower address. So, 12 is coming here and the highest significant bits will be stored in the higher address and the proportion is being maintained.

When it comes to big endian there is only one reverse, the most significant bits are actually stored in lowest address. And the least significant bits are stored in the higher address. So, how will you store let us say this is my data 0 A 0 B 0 C and 0 D, if I am going to store this 32 bit data in four consecutive words in memory, if it is a big endian storage then MSB should be stored in lower address in this case my lower address is a and LSB is 0 D should be stored in the higher address there is A plus 3.

The same data when I am going to store it in a little endian format, then the LS by least significant byte will be stored in the lower address, and the most significant byte will be stored in the higher address, this is also an illustration for the same.

(Refer Slide Time: 45:56)



Byte / Word Alignment

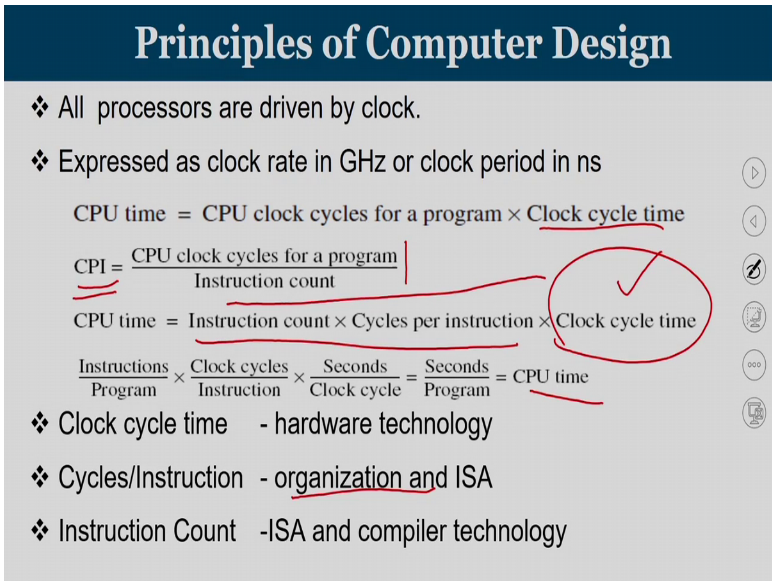
Next concept is byte or word alignment, sometimes your data maybe one byte sometimes your data may be 2 bytes some will be 4 bytes. So, when you store your data consecutively in memory let us say you are going to store a 1 byte data followed by a 2 byte data followed by the next 1 byte data, then there are alignment issues in memory.

Consider the case of a data like this if you are going to align, your memory in terms of 1 byte your 1 byte data is stored, then you have a 2 byte data that will be stored immediately after that and then you have the next 1 byte data. This looks like normal the difference is there, when you are going to align in terms of 2 bytes. When your memory is aligned in terms of 2 bytes no data can start in an address which is a multiple of 2 which is not a multiple of 2. So, you have your 1 byte data a new data cannot start, here new data can always this is like 0 1 2 3 4 like that. So, a new data can start either at 0 or a 2 or at 4 like that that is called aligning in terms of 2 bytes.

So, even though my first data is 1 byte, I have a vacancy that is available, on the next byte since I am aligning it to 2 bytes my data would not get stored here, this is call two alignment. Now what about 4 byte alignment data can always start at multiples of 4, you can see that first data it is a 1 byte data, it starts at 0. The next data is 2 byte I have enough space, because of my alignment is 4 bytes I can start the new data only at the 4th byte. Similarly next data I can start only at the 8 byte. So, this will lead to empty spaces inside memory that is one disadvantage of this.

But this will be always having merits that every new data searching, that is the memory is aligned, it is much more faster in the subsequent memory sections, we will learn deeper about the advantages and disadvantages of this.

(Refer Slide Time: 48:30)



Principles of Computer Design

- All processors are driven by clock.
- Expressed as clock rate in GHz or clock period in ns

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

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

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

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

- Clock cycle time - hardware technology
- Cycles/Instruction - organization and ISA
- Instruction Count -ISA and compiler technology

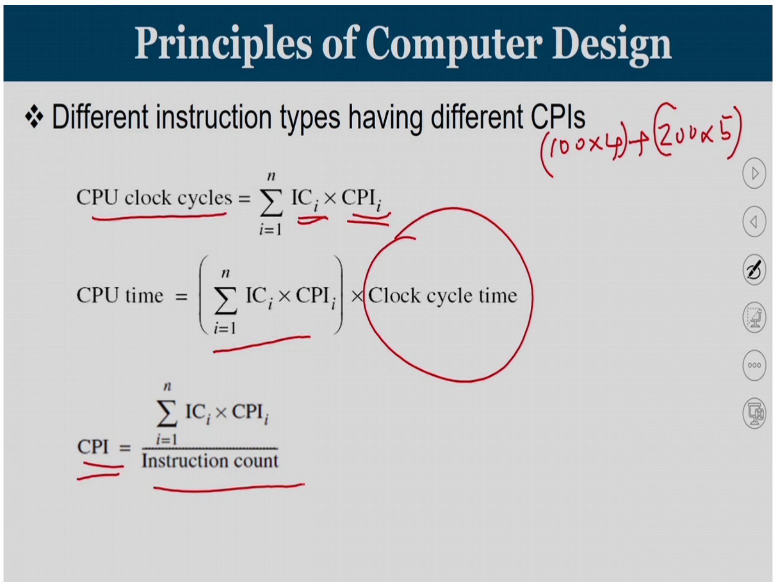
Now, coming to the principles of computer design all processors are driven by clock. And typically we express clock in terms of hertz gigahertz or clock period in nanosecond. So, the execution time of a given task is specified by CPU clock cycles of a program, into how much is one clock cycle that is called clock cycle time.

Now CPI cycles per instruction it is one important metric which will tell, how fast a processor can execute a task CPU clock cycles of a program divided by instruction count this is called CPI. Now, CPU time is defined as instruction count the total number of instructions, into what is the average cycles required to complete one instruction into. So, this will give you in terms of clock cycles and into what is the clock cycle time. So, this section is about in terms of cycles and this will you are going to multiply with what is the clock cycle time.

That means instructions of a program that is called instruction count, in the clock cycles per instruction into seconds per clock cycle that is called seconds per program and that is what is called your CPU time. Now, clock cycle time its dependent on the hardware technology sometimes the processor maybe 1 gigahertz some maybe 1.2 gigahertz it is a hardware technology that is used, now what are the factors that will affect cycles per instruction your second term cycles per instruction, it is basically from the instruction set architecture whether it is going to be a register architecture or load store architecture, it is register memory architecture or accumulator architecture the organization of the instructions and ISA will governed that.

And what will governed your instruction count that also is governed by instruction set architecture and the compiler technology that, you are using. If you are having the risk instructions if you are going to use CISK instruction, these things will may vary. So, if you are using a highly optimizing compiler, compiler may compact your number of instruction which will reduce the instruction count, which is inturn going to affect your CPU time.

(Refer Slide Time: 50:41)



Principles of Computer Design

❖ Different instruction types having different CPIs

$(100 \times 4) + (200 \times 5)$

$$\text{CPU clock cycles} = \sum_{i=1}^{n} IC_i \times CPI_i$$

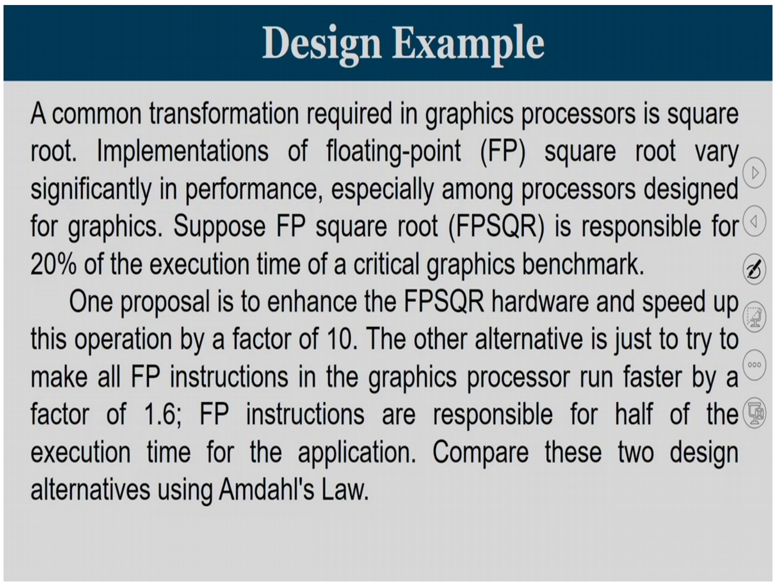
$$\text{CPU time} = \left(\sum_{i=1}^{n} IC_i \times CPI_i\right) \times \text{Clock cycle time}$$

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\text{Instruction count}}$$

Now, when you work on a program your program may have different types of instruction, some instruction may have 4 CPI some may have 10 CPI some may have 2 CPI so, how will you compare this to.

So, the CPU clock cycles is governed by sigma i equal to 1 to n IC i that is the first type of instruction into its CPI. So, some are 4 so, if you have 100 4 cycle instruction plus 200 5 cycle instruction, then it will be 100 into 4 100 4 cycle instruction plus 200 5 cycle instruction that is going to be your total CPU cycles. So, once you get this multiplied with clock cycle time, we will get the total CPU time of execution and what is CPI divide with the total instruction count.

(Refer Slide Time: 51:42)



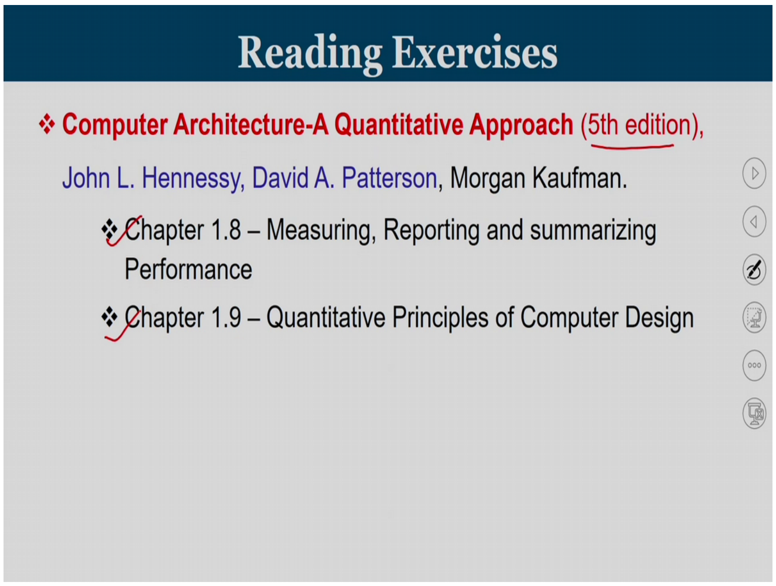
Design Example

A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark.

One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. Compare these two design alternatives using Amdahl's Law.

So, this is one question which I wanted you to explore, over this weekend we will post the solutions of this, once this week is going to end probably by Friday or Saturday, I want all of you to go through this question and apply Amdahl's law on it and find out what is the speed up that you are going to get.

(Refer Slide Time: 52:08)



Reading Exercises

- Computer Architecture-A Quantitative Approach (5th edition), John L. Hennessy, David A. Patterson, Morgan Kaufman.
  - Chapter 1.8 – Measuring, Reporting and summarizing Performance
  - Chapter 1.9 – Quantitative Principles of Computer Design

To understand deeper about the topics that we have covered today, this are the reading exercise this topic is been taken from the computer architecture text book by Hennessey and Patterson, I am referring to 5th edition go to chapters 1.8 and 1.9 and that will give you more study materials for the topics covered today. And if there is any doubt regarding this topic feel free to post your queries in the discussion, forums we will the respond back to you so, that your doubts will be cleared.

So, thank you the lecture is ending today and I hope that you enjoyed the session, we giving and some maybe having background in this topic, we will find this topic CC those were relatively new to computer architecture background, just go through this slides that is being given, we will post the slides as well and try to get back to us if there is any query we will be very happy ah to clarify your doubts.

Thank you.