High Performance Computing

Prof. Matthew Jacob

Department of Computer Science and Automation

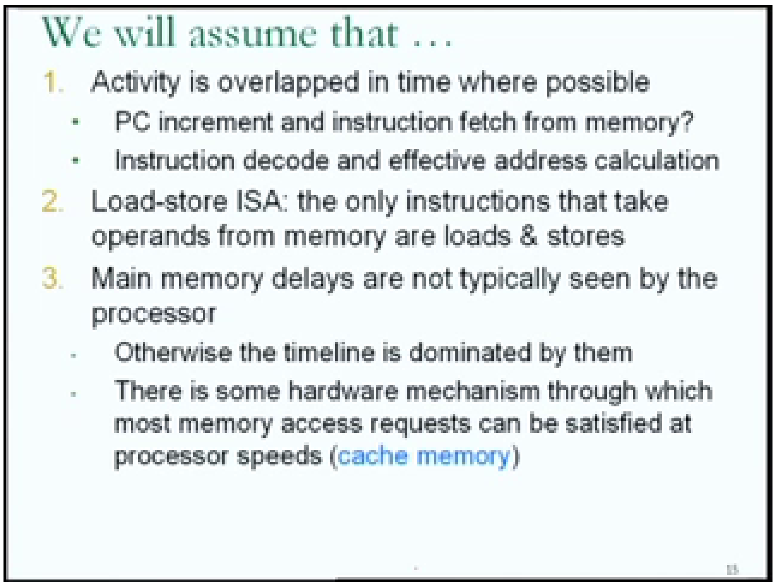
Indian Institute of Science, Bangalore

Module No # 02

Lecture No # 10

Welcome to lecture 10, of our course on, “High Performance Computing”. In the previous lecture, we had taken a look at a few more MIPS one instruction set features, and started looking at the kind of hardware that would be involved, in executing or processing, the instructions as a program executes and just referring you back to the last slide of the previous lecture.

(Refer Slide Time: 00:42)



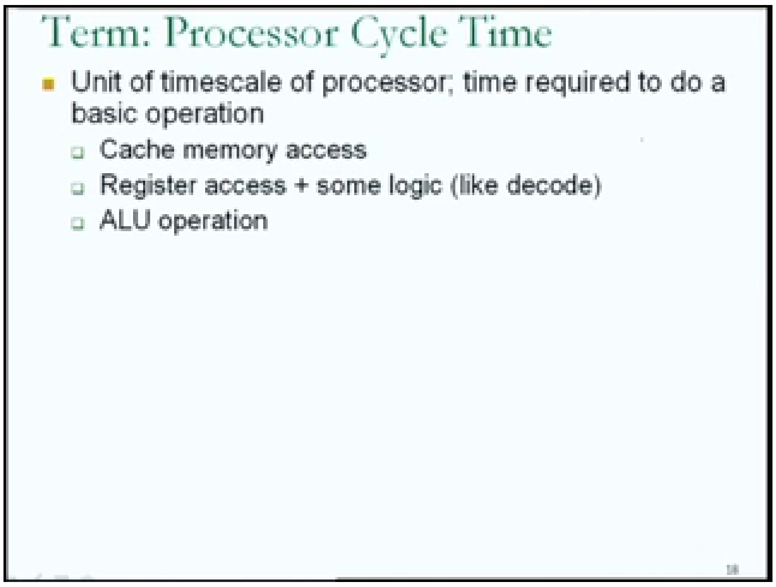
We will assume that …

1. Activity is overlapped in time where possible
   - PC increment and instruction fetch from memory?
   - Instruction decode and effective address calculation
2. Load-store ISA: the only instructions that take operands from memory are loads & stores
3. Main memory delays are not typically seen by the processor
   - Otherwise the timeline is dominated by them
   - There is some hardware mechanism through which most memory access requests can be satisfied at processor speeds (cache memory)

We were going to make three important assumptions in trying to understand the hardware. The first assumption is that, we will assume that in the decisions regarding the hardware activity can be overlapped in time or done simultaneously. So, if there are two different steps in instruction execution which can be done at the same time. And the hardware for those two steps would be designed, so they could operate at the same time.

We will assume that we are concerned only with load, store, risk instruction set, and finally, a big assumption, we will assume that even though main memory is about two orders of magnitude slower than the processor, we will assume that delays due to main memory are not typically seen, during the execution of instructions by the processor. Because otherwise, that would dominate the execution of that the amount of time for the execution of a program, and in fact, this is going to be, though initially, we thought that this was an unrealistic assumption. We are told that if a processor contains something called the cache memory, then this is not an unrealistic assumption. So, we will go with these three assumptions.

(Refer Slide Time: 01:46)



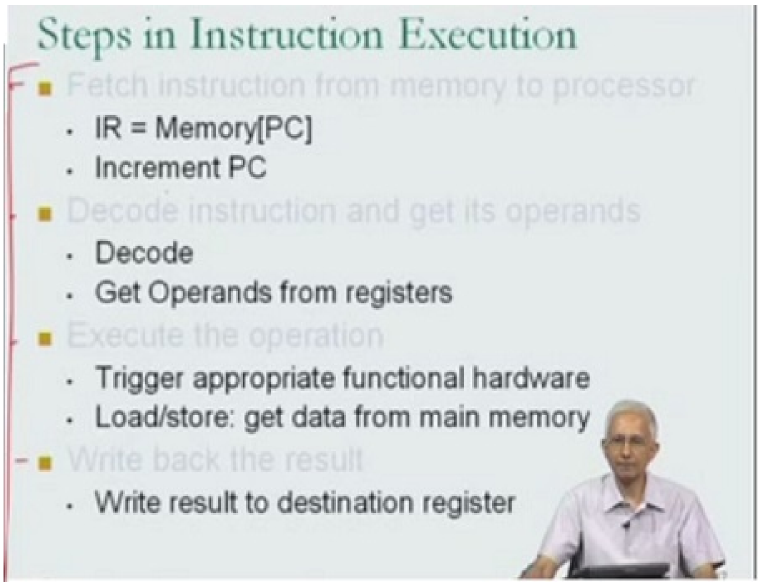
Term: Processor Cycle Time

- Unit of timescale of processor; time required to do a basic operation
  - Cache memory access
  - Register access + some logic (like decode)
  - ALU operation

Now, one of the important concepts for us to have in mind, before we can start looking at simple hardware to execute an instruction, is a notion of, “Processor Cycle Time”. I had use this term earlier. In general, I had used that the idea of the time scale of a processor, when I have said that the time scale, of a typical processor today, is a couple of order magnitudes finer than the time scale of current main memories, and I will refer to the processor cycle time as a unit of that time scale.

So, for example, when we talked about some slower processor today, we talked about it us having a unit of time scale of about one nanosecond. So, that was the ballpark kind of number that we were talking about, in other words, 10 to the power of minus 9 seconds. And more realistically, since now, we are talking about the steps involved in executing an instruction, I will refer to a processor cycle time, as the amount of time required to do a very simple operation. A simple operation can be done in one cycle and therefore, if many steps or simple operations must be done to execute an instruction, the amount of time to execute a single instruction will be multiple clock cycles.

(Refer Slide Time: 03:03)



Steps in Instruction Execution

- Fetch instruction from memory to processor
  - IR = Memory[PC]
  - Increment PC
- Decode instruction and get its operands
  - Decode
  - Get Operands from registers
- Execute the operation
  - Trigger appropriate functional hardware
  - Load/store: get data from main memory
- Write back the result
  - Write result to destination register

So, we have to have some idea of what kind of simple operation can be done in a single processor cycle? Now, let us look back at our sequence of steps involved in the execution of an instruction. So, there are the four steps described at the higher level; you need to fetch the instruction, you need to decode it, you need to do the operation involved and then you need to right back the result. Now, in each of these, we had put some sub steps. These are the more basic operations, which are involved in executing the instruction. The entire sequence of steps is required to do the execution of one instruction, but let us look at these sub steps. So, first there is IR equals the memory of program counter, this is the step where the instruction was read out of memory.

Now, with our 3rd assumption that the most of time memory delays, are not seen, because of the cache memory present in the processor, due to which, the instructions or data that are required, by the processor are much of the time, available at processor speed. So, rather than viewing that first step, IR equals memory of program counter as being a multiple cycle operation, I can say that this is going to take the amount of time that it takes to do a cache memory access. Cache memory is something inside the processor, therefore, the amount of time for the first step, is the amount of time to do a cache memory access. It is inside the processor (Refer Slide Time: 04:24).

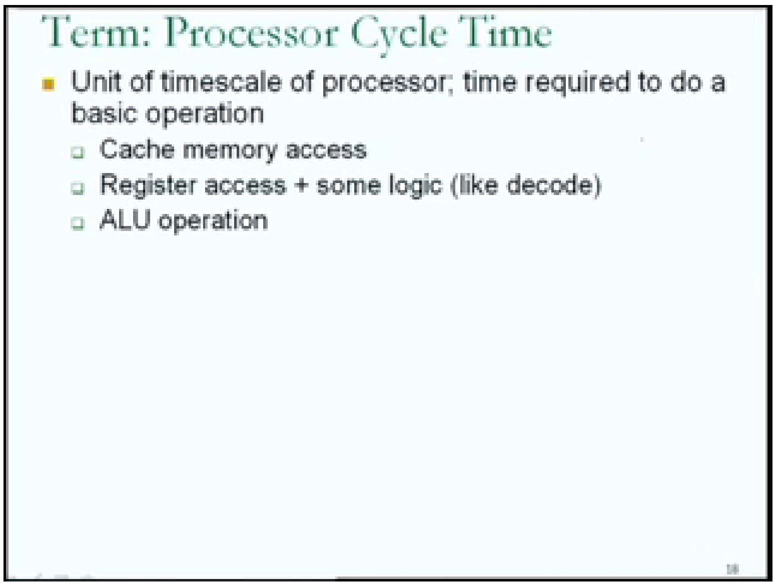
What is the amount of time to increment the program counter? Incrementing the program counter involves adding reading the program counter, adding 4 to it, for our MIPS one style risk instruction set and storing the sum back into the program counter. So, that is our simple ALU operation (Refer Slide Time: 04:42). I write it as a simple ALU operation, because it nearly involves addition by 4. It is much less complicated by a general signed ALU operation. When you come to decode, what is involved in decode? In the decode step, the fields of the instruction has a size in the instruction register are examined by the hardware to determine the operation is for the operands are and so on. And this is going to implemented using some very simple circuitry, well even if it is not simple, still some amount of circuitry.

Now, the way that I am describing this over here, is using the term logic circuitry (Refer Slide Time: 05:18), and that is somewhat, in some cases, people may have even describe it as simple logic or just logic. The term logic, is coming from the fact that the electronic circuits that I use, for this offer purposes like this are may not very, at the lowest level one can give them as implementing functions of logic gates, And gate or gate, inverter etcetera, therefore, the term logic circuitry has come to be use..

So, in order to do the decode some small amount of or some amount of logic circuitry is going to be involved. To get the operands from registers, it just takes the amount of time to access the registers. There are about 32 registers in the processor as based on our current assumption. What about the next step in executing the operation? The appropriate functional hardware has to be trigged and the operation has to be done. So, for example, if this is an add instruction then the ALU has to be trigged and the add operation has to be done (Refer Slide Time: 06:15).

So, that is the time involved in that is the amount of time for the ALU operation. On the other hand, if it is a load or stored then the data has to be, either read from in the case of a load or within two in the case of a store memory. Again with our third assumption, we no longer are talking about the memory delay, but about a cache memory access time. And finally, writing the result back to the destination register takes the register access time. So, now looking at these sub steps in the execution of instruction, we realize that each of them involves a memory access none of them involves the memory access. Each of them involves an operation inside the processor, and we could possibly try to simplistically view these as the determinants of what a cycle; the amount of a time for a processor cycle might be.

(Refer Slide Time: 07:00)



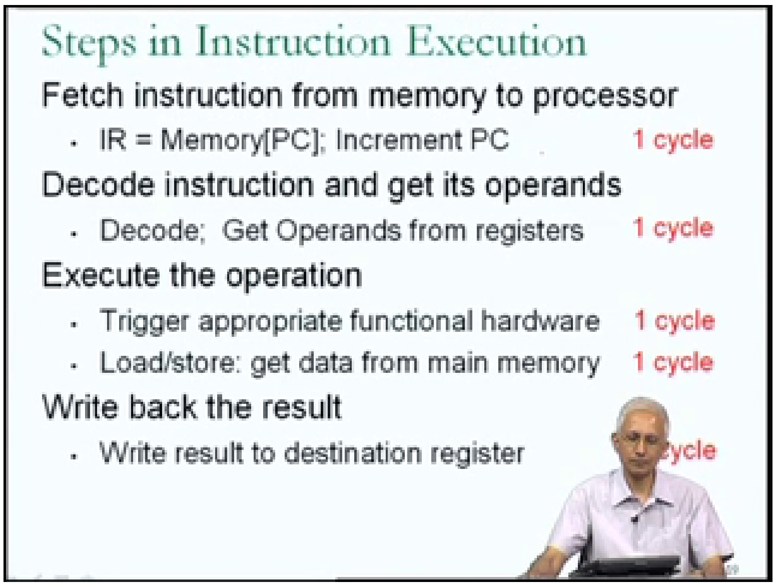
Term: Processor Cycle Time

- Unit of timescale of processor; time required to do a basic operation
  - Cache memory access
  - Register access + some logic (like decode)
  - ALU operation

In other words, we might say that a processor cycle is the amount of time, in which any one of these primitive steps, the simple operations in the execution of instruction might take place. In other words, the amount of time for a cache memory access, or the amount of time for an ALU operation, or the amount of time for a register access may be along with some simple logic like decode.

Since, the amount of time for a register access is likely to be much smaller, much less than the amount of time for an ALU operation, or a definitely much less than the time for a cache memory access. So, with this kind of assumption of what it means, what processor cycle time means, we can look back at the basic steps, and try to count the amount of time there we take to execute any one side, one instruction, in terms of processor cycles, and we think of a processor cycle as being like that one nanosecond. So, you can actually count or associate time intervals or nanosecond counts with execution of instructions.

(Refer Slide Time: 07:55)



Steps in Instruction Execution

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

Write back the result

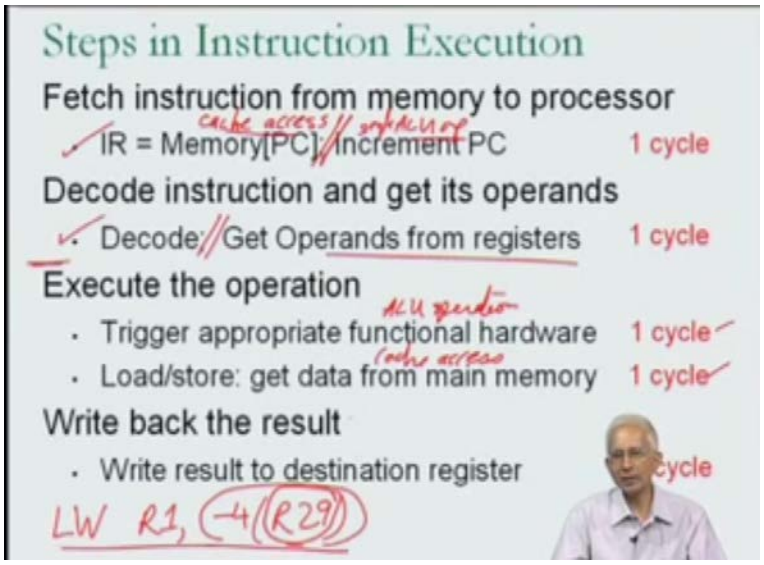
- Write result to destination register ycle

So, let us go back to the slide with the steps in the execution of instruction. So, we were going to assume that IR memory of program counter would take one cache access time. An incrementing the program counter was a simple ALU operation, and we saw that these two could be done in parallel. Therefore, the amount of time to do the instruction fetch, we will view as being one cycle. The amount of time to do the cache access would have been one cycle, the amount of time to do the simple single simple ALU would have been one more cycle, which is why we had showed it as two lines in our previous listing of the operations, but, bearing in mind, our first assumption, which was that things which can be done in, overlapped in time, will be done overlapped in time.

So, I am indicating over here that these two operations could actually be done in parallel, as per our earlier discussion, and therefore, we account one cycle against those two operations. Similarly, we had also discussed and I think sort of realize that the decoding of the instruction and the fetching of the operands from registers might also be done in parallel and could therefore, we account it as being just one cycle. However, in the third major step of instruction execution, there was triggering the appropriate functional hardware, such as an ALU operation, and an addition to that the cache access. And unfortunately, these two operations cannot be overlapped.

Why they cannot be overlapped? The reason they cannot be overlapped is, consider the load store instruction. Let us say a load instruction. So, a load instruction could look something like this (Refer Slide Time: 09:36). So, you will notice that as for as, a load instruction is concerned, it gets fetched in that one cycle, it gets decoded and its operands get fetched in one cycle. What does it means for operands to get fetched? That basically means that the value in R29 will be available and that is what its operands fetch means. Subsequently they are true appropriate functional hardware has to be triggered.

(Refer Slide Time: 09:55)



Steps in Instruction Execution

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

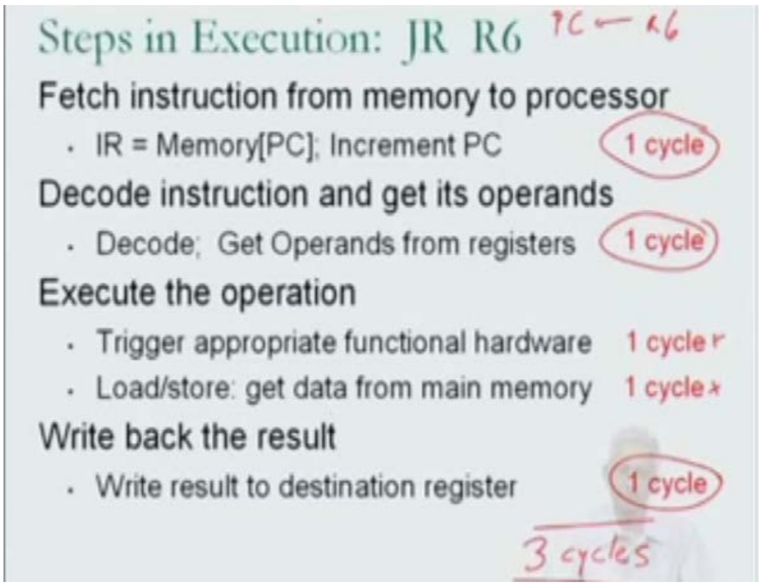
Write back the result

- Write result to destination register cycle

Now, we do not normally think of this load instruction, as requiring an ALU or floating point, multiplier or anything. But, when we realize that there is a need for an addition in order to implement the effective address calculation, in other words, adding the contents of R 29 to minus 4, then we do realize that something like an ALU operation is required. In order to calculate the address, which is supposed to be sent to the cache memory in order for the cache access to take place, therefore, we do need to account one cycle for the appropriate functional hardware and one cycle for the cache access, even in the case of a load instruction, which is why, I show one cycle for each, in this more detailed numeration of the amount of time for execution of an instruction.

The final step is writing the result with destination register, which is equivalent to the amount of time for a register access, which we had accounted as part of the definition of one cycle. So, now let us look at very few different examples of MIPS one instruction, and try to see how much time that particular instruction will take.

(Refer Slide Time: 11:05)



Steps in Execution: JR R6 $PC \leftarrow R6$

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

Write back the result

- Write result to destination register 1 cycle

3 cycles

Let us start, by looking at a jump register instruction, and for each step of the sequence, we are going to ask, does the jump register R6 instruction need that functionality to happen. So, for example, we ask, does the jump register R6 instruction need to be fetched and the answer is yes.

So, I will circle the one cycle that one cycle will have to, the hardware associate with that one cycle will have to execute for the jump register R 6 instruction to the executed. Does it have to be decoded and its operands fetched? The answer once again is yes. It has to be decoded and certainly the value of R 6 must be fetched from the register file. Is there any functional hardware to be trigger or is a need for any load or store to be done from the cache memory? The answer is no, and therefore, we do not account either this cycle or this cycle against the execution of a jump register R 6. Finally, does a destination register have to be written? The answer is yes, because we realize that as for this register has jump register R 6 is concerned, the primary effect of this register is to change the program counter to the contents of R 6, and that is therefore, an integral, that is basically very important as for as this instruction is concerned.

So, looking at the accounting that has been done, we realize that the jump register R 6 could execute in 3 cycles. It does not require all of the steps to take place, but it does take this required steps amounting to 3 cycles to take place in order for it to be executed. Just look at another example. Consider the instruction ADD R 1, R 2, R 3, one of a more popular frequently occurring example. So, does it have to be fetched? The answer is yes. So, that is one cycle. Does it have to be decoded and its operands fetched? Yes, certainly. These operands are the values in registers R 2 and R 3. So, this we account as another cycle. Does any functional hardware have to be triggered? As for as this instruction is concerned, yes! Certainly, because it is an add instruction therefore, the ALU has to do the addition, so that cycle has to be accounted.

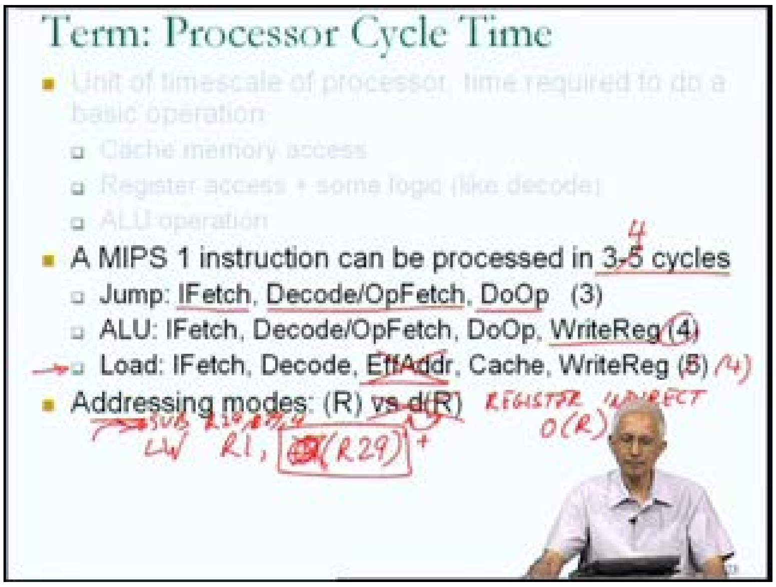
Does data have to fetched or return into the cache memory? The answer is no therefore, we do not account this cycle. Does the destination register have to be written? The answer is yes, R 1 is the destination register. So, we account this cycle and when we do the accounting we see that the ADD R 1, R 2, R 3, instruction will take 4 cycles to execute. It does not require the fifth cycle, which was necessary only for this load or store instruction.

So, just to remind you the jump register R 6 instruction, we accounted as taking 3 cycles. The ADD R 1, R 2, R 3, instruction we account, is taking 4 cycles. One more example, consider the load word R 1, minus eight R 29 instructions. Does it have to be fetched here, as this one cycle is going to account against all instructions, as is the second cycle. Whatever the instruction is it does have to fetched from the cache and it certain we has to be decoded and its operands fetched.

Now, is there any functional hardware that has to be triggered as far as this instruction is concerned? We have just seen in one or two slides ago that there is a need for an A L U operation as part of calculating the effective address. It contains of R 29 add it to the sign displacement and therefore, we do have to account that one cycle. Next, is there a need to account of cycle against the load or store in terms of access to the cache memory? This is the load instruction therefore, certainly need for accounting that one cycle and finally, is there a destination register to be return? Yes there is, R 1 has to be return and therefore, when we do the accounting we see this particular instruction; the load word instruction of the MIPS one instruction set, based on R set of assumptions regard the 3 assumptions as well as are assumptions about accounting of what can be done in cycle, will take 5 cycles to execute.

So, the jump register R 6 instruction to 3 cycles, the ADD R 1, R 2, R 3 instruction into 4 cycles and the load word instruction is accounted as taking 5 cycles.

(Refer Slide Time: 15:11)



Term: Processor Cycle Time

- Unit of timescale of processor: time required to do a basic operation
  - Cache memory access
  - Register access + some logic (like decode)
  - ALU operation
- A MIPS 1 instruction can be processed in 3-~~5~~ 4 cycles
  - Jump: IFetch, Decode/OpFetch, DoOp (3)
  - ALU: IFetch, Decode/OpFetch, DoOp, WriteReg (4)
  - Load: IFetch, Decode, ~~EffAddr~~, Cache, WriteReg (~~5~~) (4)
- Addressing modes: (R) ~~vs d(R)~~

So, in general, you could go through this exercise for all the MIPS one instruction that we have seen, and you would find that based on our 3 assumptions and the assumptions regarding the definition of a cycle, any of the instructions will take something between 3 to 5 cycles. Jump register R 6 takes the minimum, only 3 cycles, and the load word instruction takes the maximum 5 cycles; all the other instructions are 3, 4 or 5.

Now, over here, this is a listing of the 3 examples that we have seen. The kind of terminology we might use, in referring to the fetching of the instruction, I might use the term I Fetch standing for instruction fetch, one cycle for that; in terms of decoding the instruction and fetching its operands in parallel, I might use the term Decode/Operand fetch and in terms of doing the operation, I might use the abbreviation DoOp. This are one of standard abbreviations and are (( )) abbreviations that I will use. In addition to that, I might talk about WriteReg for writing the destination register, and then this step of effective address calculation, which was necessary only for the load and store instructions in our particular MIPS one instruction set, the addition of the base register value to the sign displacement.

So, the bottom line is any of the MIPS one instructions can be executed in somewhere between 3 to 5 cycles. Now, as a little bit of aside, the question that arises now is we had may certain assumptions, not we, but the people who design the MIPS instruction set that made certain assumptions regarding for example, the choice of addressing modes and how critical were those assumptions looking back.

Now, let just make hypothetical assumption. Let us suppose that the people who design the MIPS instruction set had decided that it was not worth having the base displacement addressing mode, but had decided instead to only have the register indirect addressing mode. You will remember that the register indirect addressing mode is in addressing mode where the address of the memory operand is contained inside the register. It was superficially similar to the base displacement addressing mode and is in fact, a special case of the base-displacement addressing mode where the value of the displacement is 0. In other words, nothing has to be added to the destination register I am sorry to the base register.

Now, the people who design the MIPS instruction set architecture could easily have opted to have only the register indirect addressing mode and what is they had made this assumption. Now, if they had made this assumption looking at the sequence of cycles that we are associated with a different operation in our instruction set, we realize that the impact would have been only on the load and store instructions, and specifically they would not have been the need for effective address calculation. They would no longer be the need to add the displacement to the value in the base register, since the displacement is 0 and therefore, they would not be the need for, the cycle for effective address calculation, the A L U operation of calculating the address of the memory operand.

And therefore, the impact on our table on the screen would have been that they would have been known instruction which required 5 cycles; instructions would require either 3 cycles or 4 cycles. In other words, you might say, programs would have executed faster, if they had not had the, if instead of having the base displacement addressing mode they have just add the simpler register in direct addressing mode.

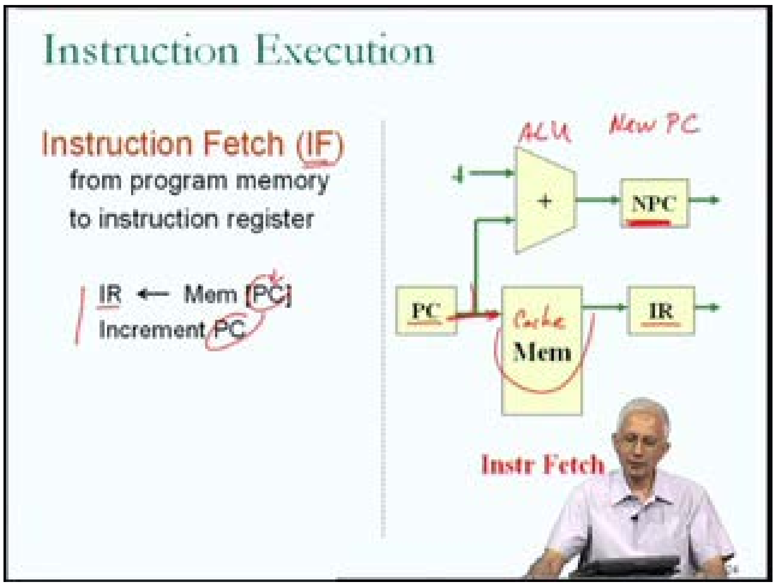
Now, this is of course, is not a complete argument, because you have to also consider what would have happen if they had left out the base displacement addressing mode, and had included only the register indirect addressing mode. Would they have been more instructions required in the program in order to achieve the same effect and you may want to look at this way. In situations where there was a base displacement addressing mode, such as a situation where there was a load word instruction R 1, minus 4 R 29.

This could have been as a result of something like our local variable access inside a function call and that may be where this addressing mode came from. This may have been an instruction, which is loading a local variable x into register R 1.

What if we did not have the base displacement addressing mode? Then in order to generate the address of a local variable x, we would have to have an instruction which would modify the value of R 29. What you that instruction be? That instruction would be a subtract instruction, subtract 4 from R 29 and then the load word instruction could just use R 29 with the displacement of 0.

So, we were had to have to have an additional instruction in our program, if the base displacement addressing mode is not available. and therefore, even though the amount of time to execute the load instruction would have gone down by one cycle, we would have one more instruction, which would take in additional 4 cycles, which is why the decision about whether having the base displacement addressing mode or not, cannot just be taken into account based on the cycle count that results for a single instruction, but must take into account the impact on entire programs and that is why so we will stick with MIPS one designers idea of having the base displacement addressing mode. But, this discussion was interesting in that it reveal some of the kind of thinking that might have to go into account in designing an instruction set, architecture. Now, moving right ahead. We can get into the specifics of what happens when instruction executes.

(Refer Slide Time: 20:43)



Instruction Execution

Instruction Fetch (IF)
from program memory
to instruction register

IR ← Mem [PC]
Increment PC

Now, let us first go through the first step of instruction execution. So, the objective is, and I will refer to this as instruction fetch and instead of abbreviating it, is only as IFetch. I will abbreviate it now as I F, just to saves place. So, I F stands for instruction fetch. It is the part of the processing of an instruction where the program counter is sent to the memory responds with the instruction. The instruction is return into the instruction register and in addition the program counter gets incremented.

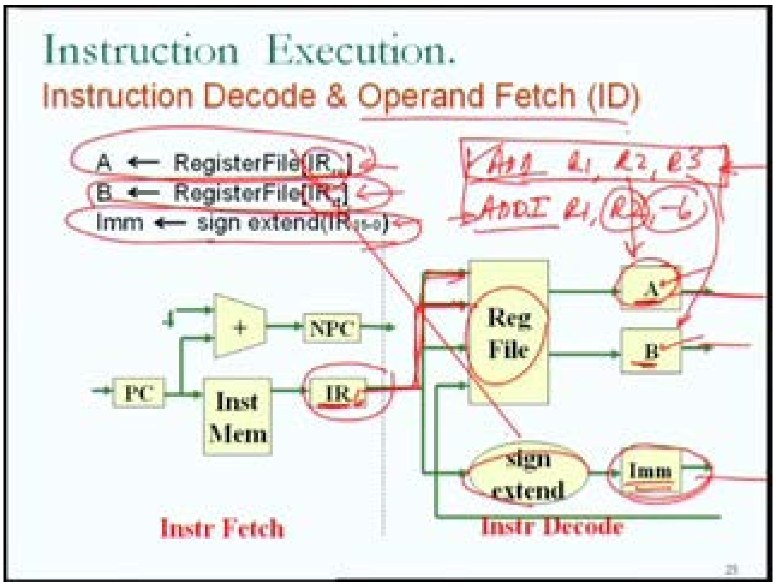
So, what kind of hardware is going to be involved in doing this? Remember, that there is the steps involved are these two send the program counter value to memory or cache is the case may be, memory will respond with the instruction which is store into the instruction register and the incrementing of the program counter.

So, over here, what I am depicting is there is one box, which is the program counter and the arrow which is shown pointing from the program counter to memory indicates that the program, in this particular example, we might be talked about cache memory. So, the value the program counter is rooted to the cache memory. That is the first thing to note. Subsequently, the cache memory operation happens and the instruction goes from the cache memory into the instruction register. In parallel, with this, the program counter value is sent to an adder along with the value 4, since that is the value of the increment and the value is sent to another register call the new program counter.

Now, you will notice here that technically what I have shown over here is a very simplified version of a piece of hardware, is showing you registers, its showing you units of memory. So, P C is a register, I R is register. The adder that I am showing you is something like a very simple A L U. So, it is a piece of hardware. The constant 4 can be generated and sent as one of with inputs. In addition, I am showing you another register, I never talked about it before, it's called the N P C, which we can now understand must be the 'new program counter' and the idea seems to be that rather than modifying the program counter directly, the register call, the program counter directly. In this discussion we are going to assume that the program counter itself may get modify either later or at the same time, but we are assuming that the value of the incremented program counter is available in another register call the new program counter.

So, this is sort of a form of pseudo hardware. We talk about pseudo code to describe software. Here we have a form of pseudo hardware to understand what the hardware is doing and this is basically all the hardware involved with instruction fetch. We can now move on to the second. You will notice that these two activities are shown as happening in parallel because the value in the program counter, simultaneously, goes to at the add of increment and to the cache memory for look up in order to fetch the instruction. So, the parallelism overlap in activity is implicitly being described.

(Refer Slide Time: 23:39)



Instruction Execution.

Instruction Decode & Operand Fetch (ID)

A ← RegisterFile[IR[illegible]]
B ← RegisterFile[IR[illegible]]
Imm ← sign extend(IR[illegible])

So, we now move on to the next step. This was instruction fetch. The next step in instruction execution was the one way the instruction is decoded and its operands are fetched. I am showing you the instruction fetch hardware to the left, because the instruction register, which contains the instruction after it has been fetched is going to be beginning of our second step of the execution, because the decoding of the instruction must examine the instruction register.

So, I am not going to show logic which implements the decoding since it is some kind of circuitry and we do not have any mechanism to talk about what the circuitry does explicitly and therefore, it is no point in showing at a block which does that. But, there is going to be circuitry, which does that in the control hardware. We will concentrate more on the operand fetch and we realize that for the instructions that we are concerned about the operands are being fetched from the register file.

So, I show the register file in terms of this kind of functionality. Now, for the particular kinds of instructions which we have, for example, consider arithmetic and logical instructions, the kinds of operands which may have to be obtained are there is a possibility that they are two source operands, in which case, two values will have to be read out of the register file. There is also the possibility that one of the operands is going to be an immediate value, in this case, there is a possibility of the operand having to come out of the instruction register.

So, in the diagram that is to be shown to the right, we are actually going to show a register file (No audio from 25:14 to 25:26). We are going to show a register file and the input to the register file, so the information from the instruction register gets routed to the register file. What kind of information? Inside the instruction register, you will remember is the instruction and we know the format of the instruction, from the discussion of the instruction set architecture. Therefore, there is a field in the instruction which identifies the first source register identity. There is a field in the instruction which identifies the second source register identity.

Therefore, those fields from the instruction, I send to the register file, and in the register file the hardware is built so that it reads the first source operand, out of the register, appropriate register, and puts that value into a new special purpose register call register A, which is what was indicated by the first step in this sequence. Similarly, the second source register is read from among the registers as identified from its value in the instruction register and read into the register called register B.

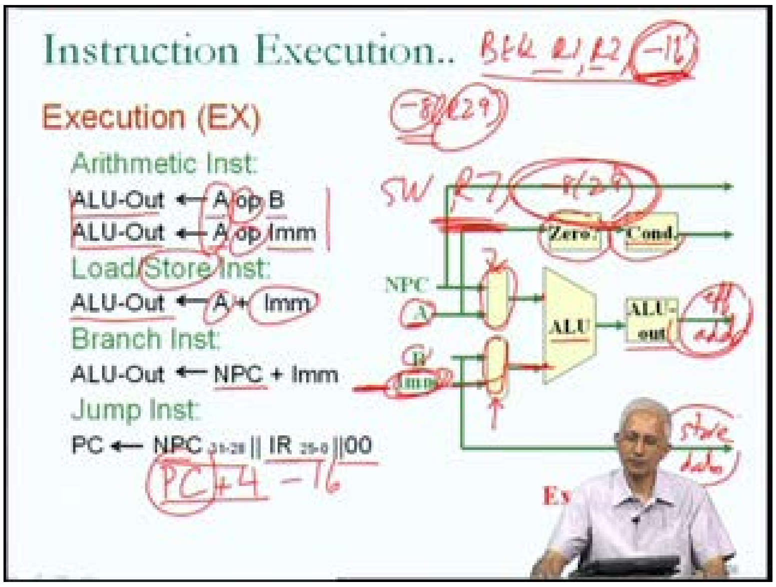
So, A and B are special purpose registers. As a result, of which values, which are operands will be read out of the register file, and that will handle the case of an instruction like ADD R 1, R 2, R 3. At the end of this sequence of events, A will contain the value of R 2 and B will contain the value of R 3. Both have come out of the register file and what about the example of ADD immediate, ADD R 1, R 2, minus six.

The value of R 2 will come as it did in the case of the first example into register A, but what about the value minus 6? The answer is minus 6 is a value, which is present inside the instruction, and there is a path from the instruction to this sign extension hardware into a register called Imm. Another special purpose register and the special purpose register call Imm contains the sign extended version of the bits inside the instruction, which contain the immediate field. The net effect is that whatever kind of instruction it might be, at the end of the second stage of instruction execution A will contain the first source operand if there is any. B will contain the second source operand out of the register if there is any, and the special purpose register call Imm will contain the immediate operand if there is any, as for as this instruction is concerned.

Now, the concern which you may have is, what if the instruction which is currently being decoded and operand fetch is an add instruction? Then, I agree that its first source operand will be available in register special purpose register A and its second source operand will be available in special purpose register B, but something will also be available in the immediate register, as far as this instruction is concerned, and this instruction has no immediate operand.

The answer to that question is even if there is a value in the immediate register, as far as the instruction is concerned as long as we do not use it, in the addition, no harm is done. But, we are here able to do a piece of hardware or implement a piece of hardware, which will do the same thing regardless of whether the instruction is an ADD or an ADDI and therefore, we are simplifying the hardware. There is no necessity to use all 3 of A, B and Imm in the step that follows. Based on the information that came out of decoded the instruction either A and B can be used or A and Imm can be used. In other words for the add instruction A and B will be used and for the add immediate instruction A and Imm will be used in the addition that is to follow. Let us move on to the next instruction and the next stage of the instruction execution.

(Refer Slide Time: 29:12)



Instruction Execution..

Execution (EX)

Arithmetic Inst:

ALU-Out ← A op B

ALU-Out ← A op Imm

Load/Store Inst:

ALU-Out ← A + Imm

Branch Inst:

ALU-Out ← NPC + Imm

Jump Inst:

PC ← NPC $_{31-28}$ || IR $_{25-0}$ ||00

Right now, the next stage of the instruction execution, I am going to label, I am sorry Let me just go back and say that I am going to, from now on, label the second stage of instruction execution as ID, as an abbreviation of instruction decode and operand fetch. Now, in the next stage of execution, we had this requirement to trigger the functionality of the required functional unit and therefore, I will just from now on talk of this as the execute for the ex-stage. This is where the execution of the instruction takes place and now primary example is one where the piece of hardware that has to be triggered is the A L U.

Without loss of generality, we may be talk about triggering a floating point add or floating point multiplier or some other piece of hardware, but the situation will be parallel to a one that we are describing here. So, there are few cases that we have to consider since they are different kinds of instructions, which may require the triggering of the ALU. We have already seen that arithmetic instructions; obviously, require triggering of the A L U, and that even load, store instructions require triggering of the A L U in the MIPS one instruction set for the effective address calculation. In other words, the additions of the base register to the displacement.

So, for the arithmetic instruction, we know that what has to happen is, if it is arithmetic instruction without immediates, then we want the value to be taken from register A and we want second operand to be taken from register B. On the other hand, if this is an arithmetic instruction with an immediate operand, then we want the first operand to be taken out of register A and this for second operand to be taken out of register Imm.

Remember that A,B and Imm are the special purpose registers, into which these operand values had been loaded in the previous step, in our I D step of instruction execution. What are the other possibilities we have to look at we will come to that shortly right. So, we talked about the A, B and the Imm registers, we now seeing that as far as the input to the A L U is concerned, the upper input is always coming out of register A because that is the case for both the add instruction and for the add immediate instruction. But the lower input comes either from B or from Imm, and this piece of hardware over here has to be a piece of hardware, which can be controlled by the decode hardware in the previous stage. So, that the appropriate selection is made between B and Imm and only one of them goes to the A L U.

So, in the previous stage there was logic circuitry which was decoding the instruction, and part of the logic circuitry would trigger this particular piece of hardware to select the correct one from one B and Imm, as the second input of the A L U. The net effect is that the correct input either A and B or A and Imm will go to the A L U. The A L U will be triggered to do the operation and will write its output, in other words the result of the addition or whatever the operation may be. The operation could be subtraction, it could be a logical operation, into another special purpose register called A L U-Out, standing for output from the A L U, and this sort of explains the notation that we are used over here.

A L U-Out is a name of another special purpose register that we are assuming is present inside the hardware and used by the control hardware of the CPU, so much for the arithmetic instruction. What about the load store instruction? Now, in the case of a load store instruction we remember that the need for using the A L U comes from examples like minus R minus 8 R 29. So, we realize that the R 29 value is going to come through the A input, but whereas, the minus eight value going to come from if you go back to the previous slide you will realize that the minus 8 value is going to come from immediate register because for just we will go back to the slide (Refer Slide Time: 32:58) if you recall the load and store instructions use the format in which there is a 16 bit field, the I format. And therefore, in the I format, the 16 bit field will be pulled out, sign extended and is therefore, available in the immediate register, and that will be the immediate value which is the displacement value in the case of the load store instruction. So, for load stores what we want is the value of the a register, which is the base register to be added to the displacement value, which is present in the immediate register, and that is what happens on the appropriate gets return into A L U-Out. And from there you can be sent to the cache memory or the main memory and for accessing the data or storing the data.

In addition, if we are talking about a store instruction, the calculation of this kind will have to be done, but you will recall that for the store instruction things are the other way around. We actually want to store into the memory address, which is being indicated by the base-displacement addressing mode. The value which is inside the register R 7. So, where is the value of R 7 going to come from and the answer is the value R 7 is going to come from the B register, if you look at the format of the instructions and the way that we are doing things.

Which is why, I show a line going from the B register to the next stage of instruction execution, which is the one way the updating of memory is going to take place. So, as far as a load or store instruction is concerned A L U-Out will contain the effective address. In other words, the effect of subtracting 8 from R 29 and the other arrow down here is going to contain the store data coming out of the B register.

Now, there is a third situation that has to be taken into account here, in terms of how the hardware is going to be used by the instructions, and that is the case of the branch instruction. We call that in the MIPS one instruction set, the branch instruction, such as branch if equal R 1, R 2, minus 16, uses the P C relative addressing mode.

And therefore, there is a need for some kind of calculation to take place to calculate the target address as far as the branch instruction is concerned. Now, the value of the minus 16 is going to be available in the immediate register, because if you remember the format of the instruction that is used, minus 16 is a value that will be stored in the 16 bit displacement field or immediate field of the I format. But, in the case of the branch BEQ instruction, what we need to modify is not the register R 1 or the register R 2, but the program counter, and therefore, we need to get into the upper input of the A L U, into the lower input of the A L U, you want to get immediate, which is going to be the equivalent of this P C relative displacement, but as far as the upper input of the A L U is concerned, we want to get something related to the program counter, and that is the reason that we had this additional selection hardware available at the upper input of the A L U. So, what will go on to the upper input of the A L U is the value from the register that we had called N P C in the previous step. So, in the previous step if it why it had been if the decode hardware had understood that this is a branch instruction, then it will trigger the this piece of hardware to send a node A, but the N P C value as the upper input to the A L U and the net effect is going to be that the 16 bit P C relative displacement is going to get added to N P C and you will remember that N P C is P C plus 4.

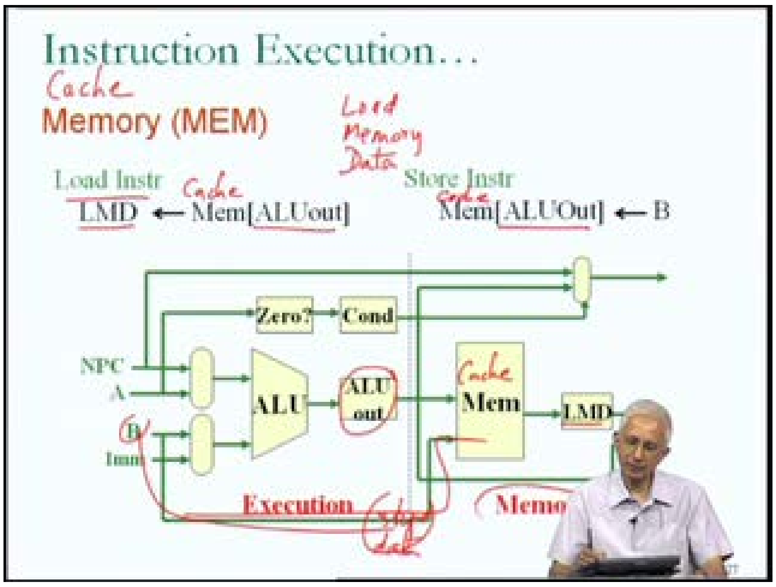
And we now begin to understand why the branch instructions talk about P C plus 4 minus 16, because in our implementation we are going to assume that rather than using the P C value the in the computation of the effect of address or the target address they are going to use the value of P C plus 4, which is available in N P C and we are referring to P C as the value of the address of the instruction which is being executed.

Hence, we show a path from N P C into the adder as well. Now, as far as the branch instruction is concerned, there is a need to also generate during this step of execution. I check of whether the condition that is being evaluated is true or false. So, is R 1 equal to R 2 or is it not and therefore, we show an additional piece of hardware, which can check for comparison with 0 or comparison between two registers and set a register which we will call the condition register.

Therefore at the end of this piece at this step of the instruction execution we have an additional register called COND or condition, which will either be contain the value one for true or the value 0 for false, which will be used in subsequent steps of instruction execution to either change the value of the program counter towards the target address which is available at the output of the A L U or not to change the value of the target address.

Similarly, for the jump instruction, so this is what has to happen in the case of instruction execution. I should comment that in the case of the jump instruction not much activity has to happen other than the concatenation of bits from the instruction register, which will be routed directly along with two zeros, along with bits from the N P C and that is not involve the A L U at all, but separate hardware, which is just concatenating bits from different fields of the N P C and the instruction register together to generate an address. It does not involve the A L U or the other pieces of hardware directly, but would be done in the stage of the instruction execution. Now, moving right ahead in the case of the load and store instructions, in the previous stage of execution, we have talked about generating the effective address. In other words, adding the base register value to the displacement and getting the store data in the case of a store instruction.

(Refer Slide Time: 38:55)



Instruction Execution…

Cache

Memory (MEM)

Load Memory Data

Load Instr

LMD ← Cache Mem[ALUout]

Store Instr

Cache Mem[ALUOut] ← B

There was an additional cycle of wok associated with load and load instructions in particular and will therefore, separate them out into another formal stage in instruction execution, which relates to the memory or cache memory, and I will refer to this stage as the M E M stage. What has to happen in this stage? In this stage, the work which had been done in the previous stage, in other words, that of calculating the effective address and of giving the store data value will have to be rooted to the memory or the cache memory as the case may be.

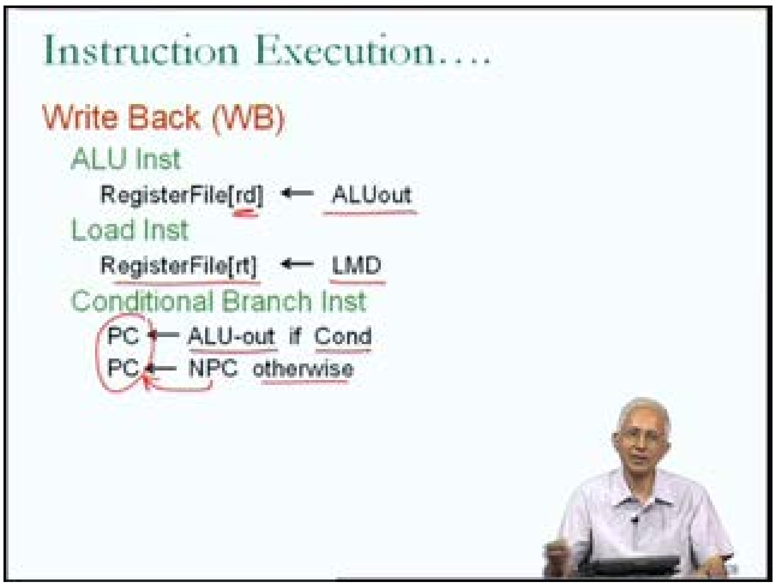
So, the A L U output register contains the address in the cache memory or the memory that has to be modified or has to be read from. So, if we are talking about a load instruction then the A L U output value, which is the address of the identity in cache memory, which is to be loaded into a register. Therefore, the address goes to the cache memory and cache memory response with a value, which is put into a CPU register called L M D or a load, memory, data through its a special purpose register which contains the value which has been read out of memory or cache memory, for subsequent storing into the destination register of the load instruction. So, hence we show that the A

L U output goes to the cache memory and from the cache memory the data comes into a special purpose register called the load, memory, data register.

If it is a store instruction on the other hand, the A L U output will be use to do to identify which entity in the cache memory or memory is to be modified and the B register will contain the identity of the value, which is to be stored into that memory location. And therefore, what has to be done is rooting the value from the B register into the appropriate entity of the cache memory.

So, we show the path from the B register into the cache memory. So, this is essentially something which takes an entire cycle. Just like each of the previous diagrams did and therefore, we show this as one more stage of instruction execution. We have separated this out from the previous stage, which until now we are associated in the same sequence of same sequence of events.

(Refer Slide Time: 41:17)



Instruction Execution....

Write Back (WB)

ALU Inst

RegisterFile[rd] ← ALUout

Load Inst

RegisterFile[rt] ← LMD

Conditional Branch Inst

PC ← ALU-out if Cond

PC ← NPC otherwise

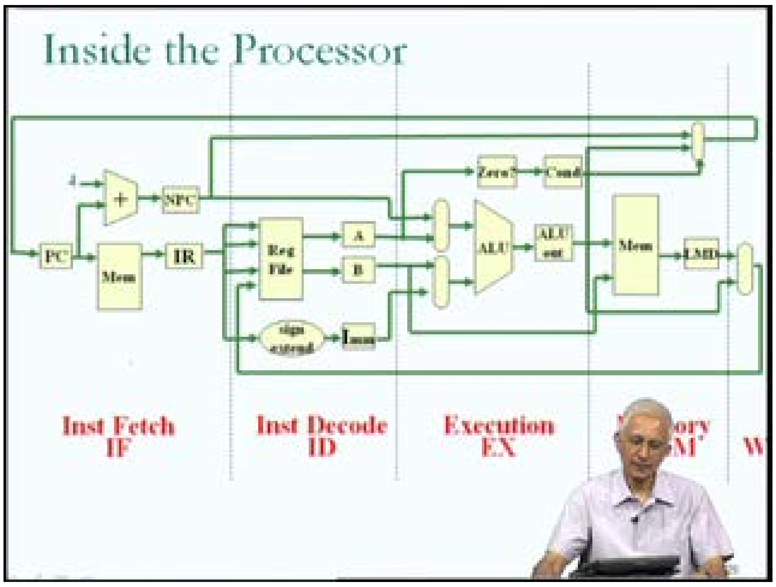
So, this is the activity that is going to happen in the memory stage. And finally, there is the last step, which has to happen. The updating of the destination register, and in the case this is relevant in the case of A L U instructions and involves taking the A L U output and putting it into the appropriate destination register inside the register file.

There is also the need to update the register if this is a load instruction. Do you remember that we have in the case of the previous step, as far as the load instruction is concerned, we have got the data out of memory or cache memory, into a special purpose register called L M D inside the processor.

From there it must finally, be rooted into the actual destination register. So, from the load memory data register into the actual destination register and finally, if we are talking about a conditional branch instruction or a jump, the program counter value will have to be modified appropriately. So, if the condition is true from as read from the condition register then the modified effective address can be rooted at the program counter value.

But, otherwise the new program counter… In other words, if the condition is false then the N P C value, in other words, the instruction after the branch instruction will be the instruction to be executed and therefore, N P C value will be loaded into the program counter value.

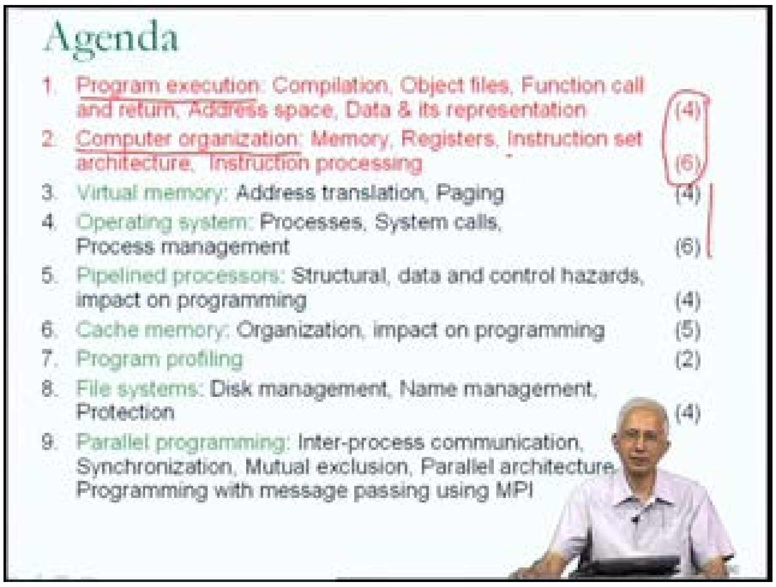
(Refer Slide Time: 42:31)



So, if you put all this together put all this together, this is what we get. We get all those simple pieces of hardware. This is the simple piece of hardware, which was doing instruction fetch. The simple piece of hardware, which was doing instruction decode. We have separated out the A L U functionality or the execute functionality, from the memory access functionality. Here, the two, both of these were listed as two sub steps in one step, but I uniformly separating them out and finally, the write back stage where the register file is, you notice that in the write back stage, a value goes back to modify the register file. We do not show the register file again. We show the register file is being a central entity.

So, from now on, we could refer to the 5 stages of instruction execution: as instruction fetch or I F, instruction decode or I D, instruction execution or E X, memory access or M E M, and W B, which is an abbreviation for write back and so this is a simplified picture putting everything together and you will recall that for some instructions, for example, for the jump register instruction, this piece of hardware is relevant, this piece of register is relevant, this piece of hardware is relevant, this piece of hardware is not, this piece of hardware is not, and this piece of hardware is relevant, which is why the amount of time to execute one instruction is going to be 3 cycles. And they would have to be some mechanism by which if you wanted to the jump register instruction would just execute in those 3 cycles. But for the jump register instruction, this piece of hardware and this piece of hardware do not do anything of relevance.

(Refer Slide Time: 44:11)



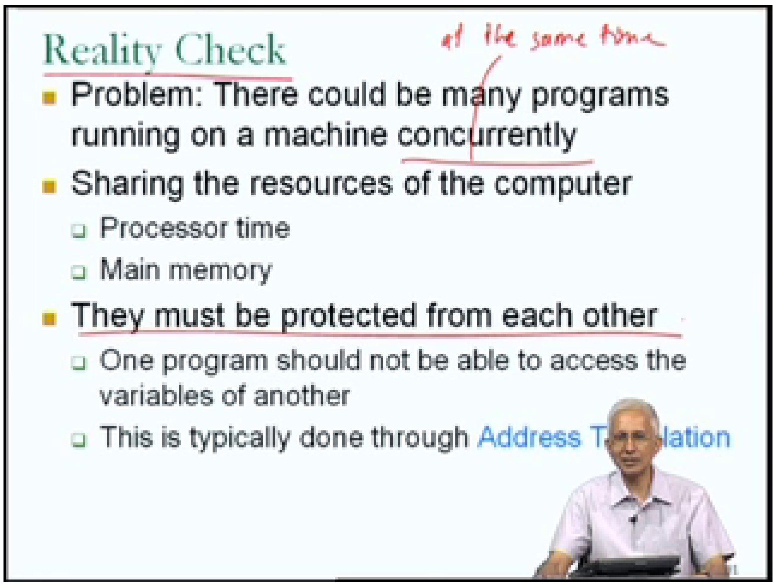
Agenda

1. Program execution: Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. Computer organization: Memory, Registers, Instruction set architecture, Instruction processing (6)
3. Virtual memory: Address translation, Paging (4)
4. Operating system: Processes, System calls, Process management (6)
5. Pipelined processors: Structural, data and control hazards, impact on programming (4)
6. Cache memory: Organization, impact on programming (5)
7. Program profiling (2)
8. File systems: Disk management, Name management, Protection (4)
9. Parallel programming: Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI

So, at this point, we have actually come to a status check point. If you look back at the agenda those we had for the course, they were nine. We had a nine point agenda and by the end of the 10th lecture, we were supposed to have completed the first two points.

The first two points were a description of program execution. Something about compilation, object files, function call and return, address space. In other words, the way that the different entities like text, data, stack and heap associated with a particular program in execution might be arranged in memory. Data and its representation, in other words, the different kinds of data and how they are represented, in addition to that the basics of computer organization and understanding of how memory works, what registers are an example of instruction set architecture and a good understanding of the basic ideas behind how instruction processing hardware is built. And we have achieved that objective in our 10 lectures. In fact, I will proceed to move into the next topic briefly to give a rough introduction to the next set of topics that we are going to get into in the lecture that follow.

(Refer Slide Time: 45:22)



Reality Check

at the same time

- Problem: There could be many programs running on a machine concurrently
- Sharing the resources of the computer
  - Processor time
  - Main memory
- They must be protected from each other
  - One program should not be able to access the variables of another
  - This is typically done through Address T[illegible]lation

Now, let me just do a quick reality check to explain why we do need some collective action before we can say that we are in condition to understand how our programs are execute. Now, if you look back at the hardware that we have, we have a situation where and in this is a clear description of what kind of hardware would be required for the execution of an instruction and the instruction comes from our program.

But, if you think about it a bit, your experience is different from what this kind of hardware might support, because in your experience there could be many programs running on a computer concurrently. What I mean by concurrently is for all practical purposes at the same time there could be many programs running on a machine at the same time. And if there are many programs running on the machine at the same time then we do need to make sure that the programs can use the same hardware at the same time. This is clearly necessary and we should not proceed too much further without understanding how that is possible.

Because if there two programs running and I have to fetch an instruction for each of those programs then, I clearly need more than one program counter, because each of the programs will have to keep track in the hardware. For each of those programs what the instruction of that program there is currently being executed is. Similarly, I would need two instruction registers and so on. So, if they are many machines running at the same time then the picture that we have in mind of the hardware might actually be unrealistic. Hence, the need for a reality check, because as you all know from your experience with computers is that there are many programs running on the machine concurrently.

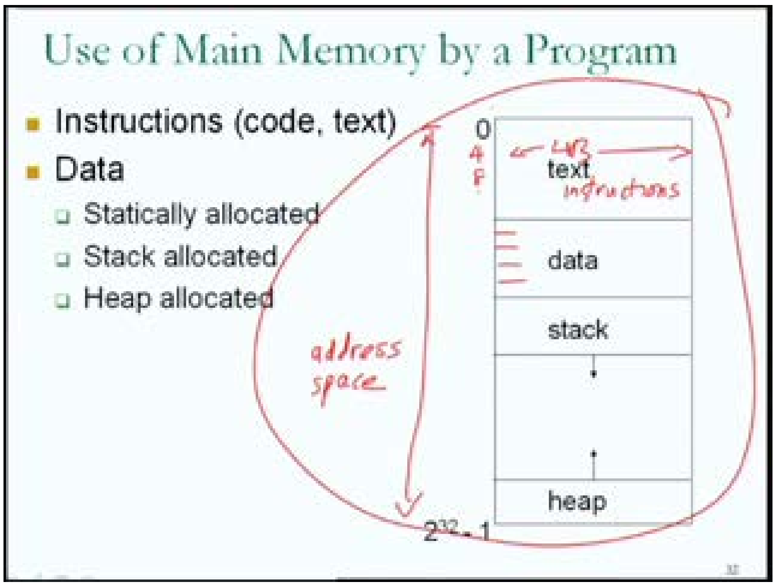
Now, the many programs that are running on the machine can concurrently are sharing the resources of the computer. And what are the resources of the computer that they are sharing? They are sharing the processor; they are also sharing the main memory. In other words, if there are two programs running at the same time concurrently, then both of those programs must be present in main memory. The instructions, the data, the stack, the heap, of each of those programs must be present in main memory. Similarly, the two programs running, then the processor should be in a situation, in a condition to execute the instructions of those two programs.

Now, if there are two programs in main memory at the same time, then they must be protected from each other. This is an interesting concept and think about this way, if I knew that there was a program of yours running on a computer then I could run a program of mine on the same computer with the intent of doing something bad to your program. For example, if I knew exactly what address was associated with each variable of your program, I could write a program which would modify those variables. So, it is clearly going to be necessary to make sure that situations like this do not arise, because that is typically the case that there are many programs running on a machine at the same time concurrently. Hence, the need to make sure those programs is protected from each other.

This must explicitly be done, otherwise no one could safely run a program on a computer unless, he had a guarantee that only his program his or her program was running and that is typically not the case today as you may be aware. So, in specifically we must ensure things like this. We must ensure that one program in execution should not be able to access the variables of another program in execution because we can access the variable x of another program; we can change the value of that variable x, by simply doing a store instruction.

Now, this protection of one program from another, is typically done in terms of not being able to access the variables of the other program, is typically done through a mechanism called address translation. And we will be seen what about address translation later. But, let me quickly give you an idea about address translation before we go ahead.

(Refer Slide Time: 49:28)



Use of Main Memory by a Program

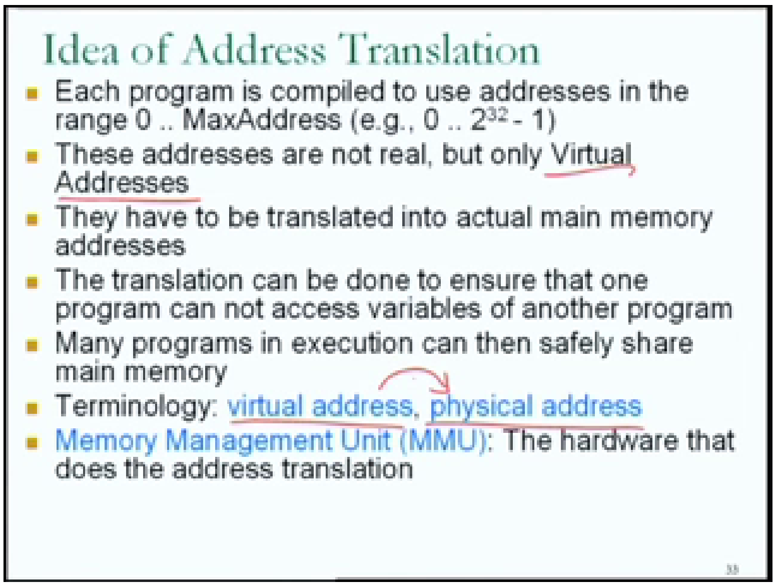
- Instructions (code, text)
- Data
  - Statically allocated
  - Stack allocated
  - Heap allocated

Now, let me remind you that any one of the programs in execution is going to have its text, its data, its stack and its heap. Therefore, as far as its use of memory is concerned and we have this picture in mind where its text or its by text what I mean is its instructions might be the present in the earlier paths of memory.

In other words, the memory address 1 4 8 etcetera, I use I am sorry 0 4 and 8 because the size of each instruction is 4 bytes. We sort of are assuming that these are byte addresses. So, the early paths of memory are the text. This may be followed by the data. This could conceivably be followed by the stack and the heap could be at the other end and what is the other end. We could assume that the other end could be addresses. I mean the highest addressed memory might be 2 to the power 32 minus 1.

If this is a 32 bit machine, the size of the program counter, the size of the various registers is 32 bits. This might be the size of the address space. The address space is the space of addresses is and the address space runs from in this particular example 0 to 2 the power of 32 minus 1. The size the size of the address space depends on the size of an address. Here, I am assuming that an address could be 32 bits in size. Now, the idea of address translation is simply each program is written with this kind of an assumption that it has an address space, which ranges from 0 to 2 to the power 32 minus 1. Let us say and this picture is achieved by the compiler and the compiler generates, pushes the instructions to associate different instructions with these addresses that associates addresses with the data with this kind of a picture in mind.

(Refer Slide Time: 51:18)


Idea of Address Translation

- Each program is compiled to use addresses in the range 0 .. MaxAddress (e.g., 0 .. 2^{32} - 1)
- These addresses are not real, but only Virtual Addresses
- They have to be translated into actual main memory addresses
- The translation can be done to ensure that one program can not access variables of another program
- Many programs in execution can then safely share main memory
- Terminology: virtual address, physical address
- Memory Management Unit (MMU): The hardware that does the address translation

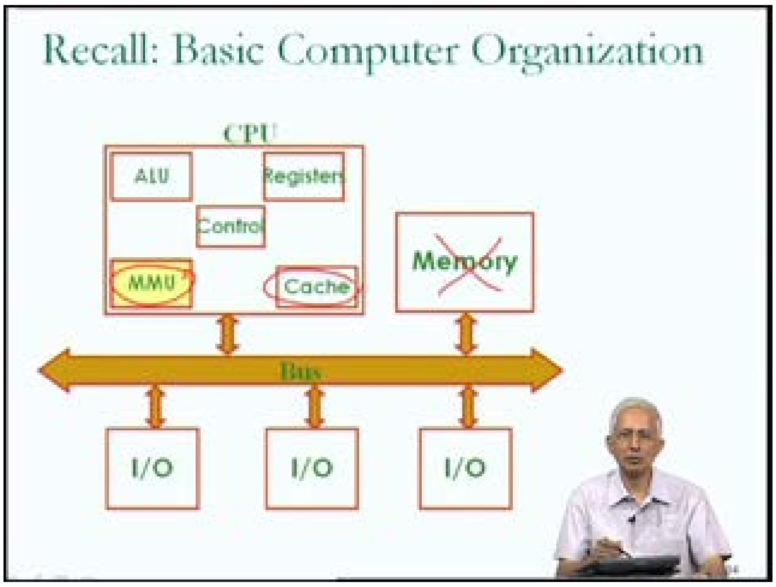
So, each program could be compiled to use addresses which are in the range of 0 up to the maximum address, in our example 0 up to 2 to the power 32 minus 1. But, these addresses need not be considered that is actually being memory addresses. We could actually view them as being fictitious or virtual addresses. Remember, our objective is to try to provide one program protection from another program, and if we have this kind of a situation where the addresses that the program was written or compiled to use are not real, but only pseudo, virtual addresses, then these addresses will have to be translated into real or actual main memory addresses before they are send to the main memory or the cache.

Now, if the translation can be done to ensure that one program cannot access the variables of another program then this translation itself provides the protection that we were talking about earlier. So, as long the translation ensures that one program cannot access variables of another program then we are safe. And then it becomes possible for many programs, which are in execution to safely share main memory. Each of them will be using a separate part of main memory, because of the way the translation is set up.

Now, the only problem here is we have been talked about the piece of hardware, which does the translation, and if the translation is not done by a piece of hardware then it is going to make everything slower, because this address translation will have to be done by through the execution of instructions, which is clearly a bad idea, because it will make programs so much slower. Therefore, it must be the case that this address translation is done by a piece of hardware and in terms of terminology, we refer to the fictitious addresses that are generated by the compiler as virtual addresses, and the actual main memory addresses that will be the result of translation as a physical address.

So, what translation does is it translates of virtual address into a physical address and the physical address can go to the main memory or to the cache memory. As we have seen this translation must be done by a piece of hardware and we will refer to that piece of hardware as a memory management unit and will abbreviate memory management unit as M M U. So, that is the hardware that does the address translation. So, this whole notion of making it possible for more than one program to be executing at pretty much the same time or concurrently makes this necessary; this whole notion of address translation and a piece of hardware to do that.

(Refer Slide Time: 53:50)


Recall: Basic Computer Organization

So, now we understand what that final block in the block diagram of the computer organization that we had seen was? The M M U is the piece of hardware that does the address translation. Just little while back, we had seen that the cache is the piece of hardware which makes it possible for us to make an assumption that most of the steps in instruction execution do not involve the main memory at all, and that only in frequently is main memory going to be involved in the execution of an instruction. Typically, the cache will provide the instructions and the data that are required by the program.

So, this point in time, with this reality check the net effect is that we understand more about why this yellow now yellow box is present. We will be seeing a lot more about what the MMU does and how it does it and with support from what kind of software, but for today we have reach the end of our discussion of instruction, execution. We are finished the first two line items in our agenda and we will proceed to the third and fourth line items of our agenda in the lectures that follow.

Thank You.

.