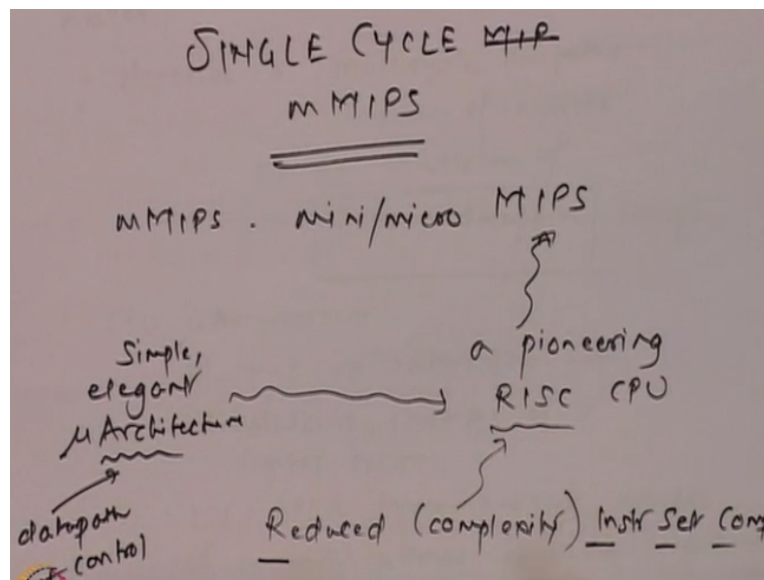
Advanced VLSI Design
Prof. Sachin Patkar
Department of Electrical Engineering
Indian Institute of Technology – Bombay

Lecture - 26
Single cycle MMIPS

Welcome again, in this lecture I will introduce a toy version of very popular pioneering CPU called MIPS.

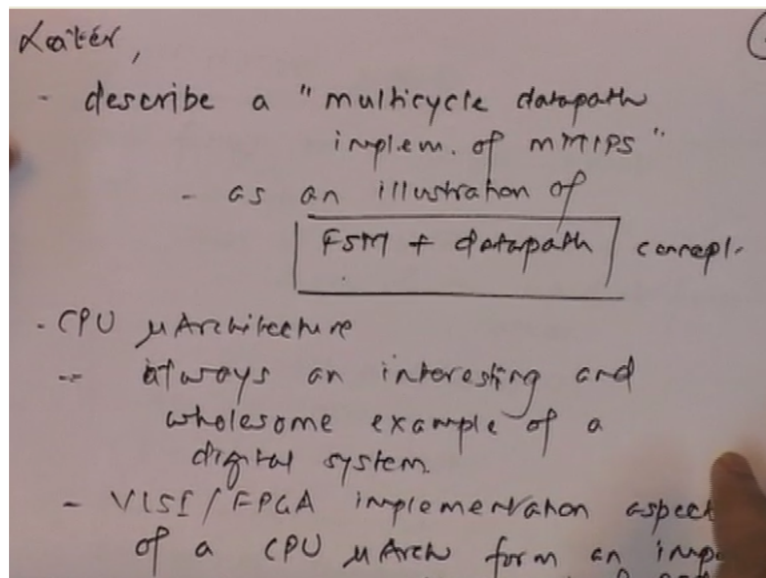
(Refer Slide Time: 00:37)



So, I am going to refer to this toy version of this MIPS as MMIPS, M probably standing for mini MIPS or micro MIPS. So this MIPS is the pioneering RISC CPU, RISC stands for reduced complexity instruction set computer. This complexity is in bracket it is typically read out as reduced instruction set computer. The reduction is really not in terms of number of instructions, reduction is in that sense of the complexity.

The instructions are quite simple and the simplicity of the instruction set leads to simple and elegant micro architecture and micro architecture by that we mean the data path and the controller which kind of define that CPU.

(Refer Slide Time: 01:25)

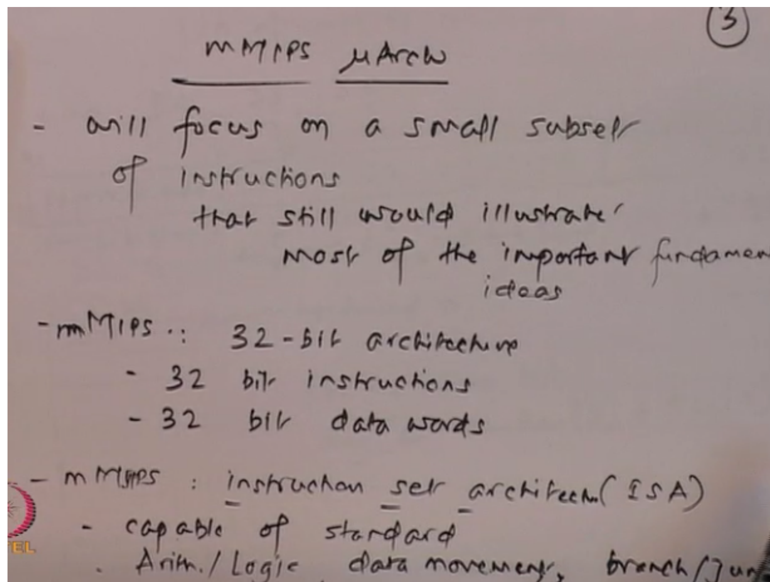


So later on in couple of subsequent lectures, we will describe a multi-cycle data path implementation of micro MIPS as an illustration of the concept of a FSM driving the data path, which we have already illustrated in the other simpler examples of GCD and shift add base multiplication. A CPU micro architecture has always been an interesting and wholesome example of digital system.

So that is what makes it kind of relevant in the context of this course also and VLSI of FPGA implementation aspect of a CPU micro architecture, this is mu architecture micro architecture, this forms an important component of pedagogue of VLSI design. So we will be able to address in this couple of lectures some aspects of implementation issues and comment on it at least and you will get some pointers to go further than this to go towards the more specialized course in processer design or VLS's architecture design.

So there is a good reason for a good example of this kind which is a very standard example in various courses, specifically on Mac computer architecture and digital system design.

(Refer Slide Time: 02:54)

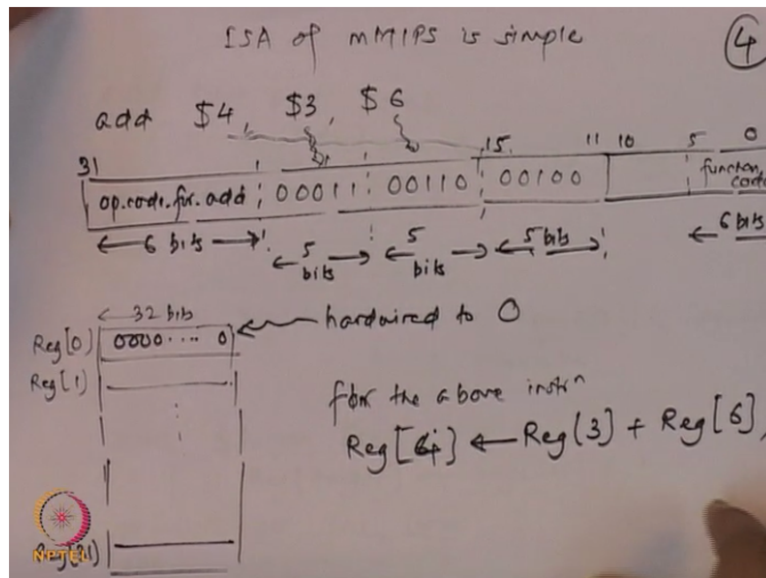


So in micro MIPS, micro architecture, we will be focusing on a very small subset of the instructions, although it is small subset, it would still illustrate most of the important fundamental ideas in the micro architecture and implementation. So this MMIPS, micro MIPS or mini MIPS, is a 32 bit architecture by that I mean the instructions are 32 bit wide and data words are also 32 bit wide, okay.

The good amount of uniformity in this RISC processes which makes things simpler to design, simpler to analyse, simpler to like, you know implement. So at the top level, at the behaviour level a processer is described by its instruction set architecture we can understand what kind of processer is capable of executing. And then the data path and controller are designed to facilitate execution of those particular instructions with the help of components like ALU, register files, multiplexers, shifters and so on so forth.

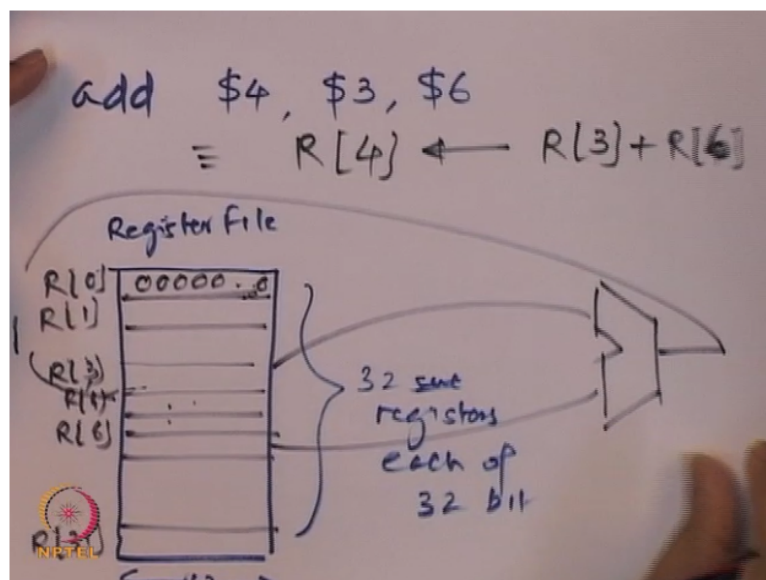
So even the small subset of MIPS which we call MMIPS is capable of very standard arithmetic logic, data movement, and branch and jump kind of instructions. So I am just saying some standard things. There is nothing yet to focus specifically on, so it is a standard toy example of a standard CPU that we are going to consider here, although it is small it will still illustrate most of the concepts.

(Refer Slide Time: 04:41)



So, for example, any standard CPU, I will draw it here again, a standard CPU should be capable of executing an add instruction, arithmetic instruction. So typical example of arithmetic instruction is add.

(Refer Slide Time: 05:02)



So MMIPS in particular has this like an instruction to which the arguments are the destination kind of index which specifies the destination, I will tell you what exactly that $4 means, and a pair of source indexes. And the MMIPS has one, micro architecture of micro MIPS has so called register file which is a collection of registers, every register is 32 bit wide and there are 32 such registers.

This registers are going to be referred to as R0 or X0, R1 and so on R31, and in particular this R0 it is really not a set of memory location, it is all hardware to ground, all zeros. So it would

look as if the register number zero always contains zero. The trick is to hardware it to the ground. And other 31 registers are general purpose, I mean most of them are general purpose and every one of them can be written to.

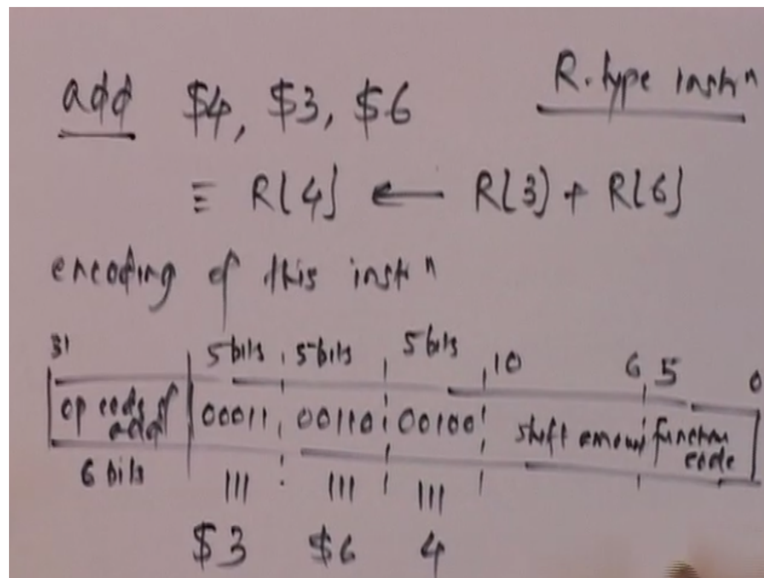
This is you cannot, R0 you cannot write to it always contains a constant. So now, coming back to this instruction this four, three, six, they refer to the indices of this register in the register file. So this particular instruction, the semantics of it is that these two indices specify this source register from where the source operands will be to be read out they will be added and then the result of the addition is going to be kept in the register with the index four.

So R4, register with the index four is going to be loaded with the result of addition on this two operands, which two operands, the contents of registers at index three and register number six, okay, so R3 and R6. So in this particular instruction the contents of R3 and R6 are going to be read out and the result is going to be somehow routed back to R4, something like that.

So the data path is going to definitely have this register file which has 32 registers the data path would also need to support, because the instructions there should be a support for addition instruction. So there must be an adder/subtracter kind of ALU and there should be some set of routers, like multiplexers which will help the things go, which will route a register from appropriate register towards a ALU and result of the ALU back to the register file.

There will be most things of this kind is required, so we will develop, we gradually kind of evolve a picture. We can evolve this complete micro architecture by understanding the need of each and every instruction of this kind. But this is kind of routine exercise, so I will be covering this subject at a fast pace. Now specifically is that how the instructions are represented, so let us get back to the same example.

(Refer Slide Time: 09:27)



An instruction like add, so clearly like you know recall what this means is R4, so this 4 is the index of the destination register and three and six are the indices of the source registers, okay. The encoding is as follows, as a remark every instructions in this CPU is going to be 32 bit wide, okay. And then this 6 MSBs, most significant bit of this 32 bit from zero to 31 all to op code of add, okay. Then we have 5 bits representing these particular 3.

That is the first source index. so this number 00011, so this is equal to 3, okay, that whatever we are specifying one of the source operant. Then the second set of 5 bits is going to encode this number six, which is to be interpreted as this second source index, and then the number 4 which is the index of the destination register for the addition operation and that is 00100. So we are going to use 5 bits for every such index.

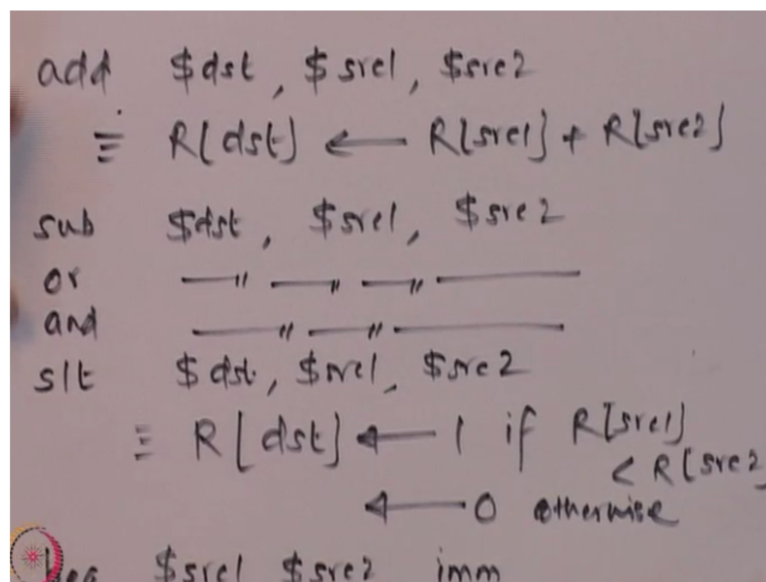
Because all these 3 indices in an instruction like add which operates on the contents on pair of registers and puts the result back in one of the registers in the register file. So this is called as a R-type of instructions, just purely working on the information on the register file, no role of any data memory or anything else here, okay. So we require first, because there is 32 registers we require 5 bits to encode the indices of them.

And this triple of 5 bits is going to contain the index of the source, index of the source another source and index of the destination. So up to here we have used 16 bits, 6 bits for this and 10 bits for this and then this is 5 more bits. In the remaining 11 bits right, bit number 10 to bit number zero we have some extra information. In the context of add it will also, there will be some, op code of add will not be completely describe by this 6 bits.

In fact, this 6bits couple with the, I think it is again 6 more bit. This bits and this bits together is going to be is going to indicate this particular instruction is an add instruction. Working on this pair of indices is describing the source operands and this particular index is describing the destination operator. We will ignore, what this is, this is where some, for some shift instruction the amount of shift is described here, again this requires 5 bits, 5 bits can specify a number up to 32, 31, I think whatever.

In shift instruction this will be used as shift amount. We will not be bothered about that in fact we are not bothered about these details at all. We just need to get an idea about how to in the simple manner these instructions are encoded. So we add and we look at couple of other examples and that would suffice us to get an idea about how the instruction looks like and how the parts of the instruction are to be used for the processing in the micro architecture.

(Refer Slide Time: 14:30)



In general, add has the structure, to specify the index of the destination, to specify the index of a pair of sources, and semantics, R[dst], R[src1] plus R[src2]. Similar to add instruction we have subtract instruction with similar structure where this dst, src1, src2 are again numbers from zero to 31 representing the indices of the registers. And other than these two arithmetic instructions we have the pair of logical instructions for being bit wise OR and bit wise AND.

Again the same triple of indices, and there is one interesting instruction called set less than, SLT, again which is. So this instruction is interesting, it has the semantics that all of, the destination register is to loaded with one provided the content of the register indicated by the

first source index is less than the content of the other register, which is indicated by the second source index in the instruction.

If OR of this, contents of register with this index is less than the contents of register with this index, then the destination register is going to be loaded with one, that is we are setting the destination register to one, setting means setting to one typically, otherwise, we are setting it to zero or we are clearing it, otherwise. So, it looks bit funny or too specialise, but it has a lot of use I means it is going to be very much useful in the comparisons, comparison based branching.

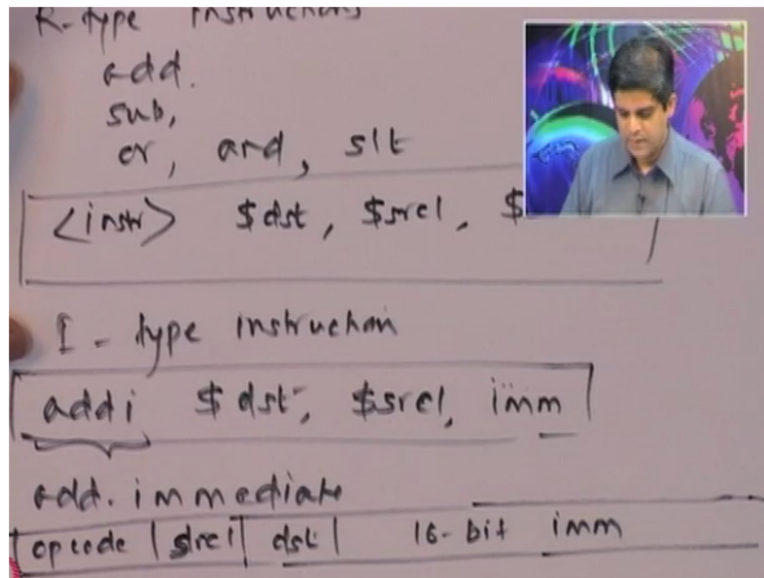
Because we are going to restrict our attention to a very simple small subset of this already reduced complexion instruction set. And that come instruction set should still be sufficient to be able to do any kind of computation. So it is SLT in conjunction with a simple branch instruction, conditional branch instruction called BEQ, which again has very similar but slightly different but mostly register based format.

Branch equitex are a pair of register indices, source one and source two compares the contents of this two register specified by these two indices. And if they are found to be equal, then it would make a relative jump to an address specified by some constant specified here, in the immediate field, this imm stands for immediate, read this as immediate just clarify it soon.

Anyway I just quickly talked about BEQ, thought of mentioning it, because this SLT in conjunction with BEQ is going to be a very powerful kind of instruction. So more about that in any standard text on computer architecture especially many text books which use MIPS as a vehicle to describe the concepts of computer architecture and organisation, standard books one of the best book known is by Patterson Hennessey and so and so.

Many of you are might already familiar with it so I am not going to spend time on this.

(Refer Slide Time: 18:35)



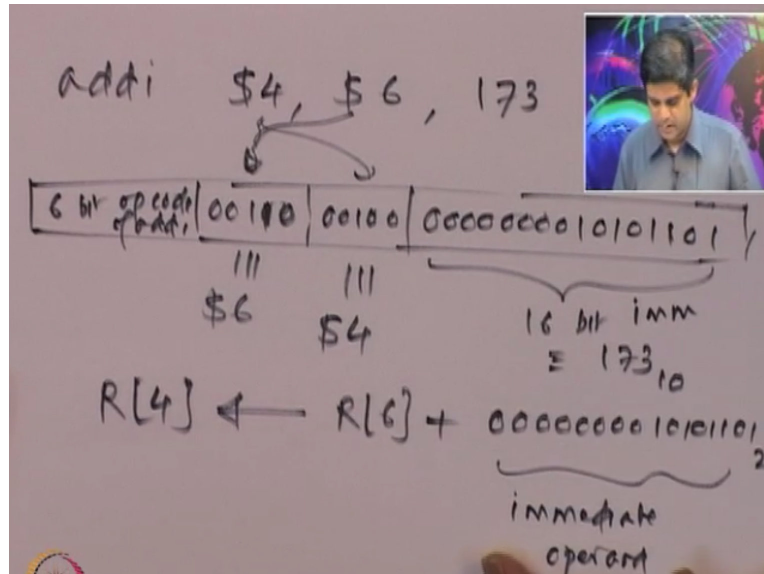
So we have seen some R-type instructions like add, sub, OR, AND, SLT, they are the type instruction the destination register specified and source, pair of source register specifies, that is the format of this such instruction they could be more of this in the standard MIPS. Other than this, we have so called I-type instructions, immediate type instructions, in which not everything is from registers and going to the register.

But an example of that is immediate version of this is add instruction, this is to be read as add immediate. So add immediate to format of that is this destination specify, because finally the result has to go somewhere but what are the source operands, the source operands are not pair of source register indices, but one of them is going to come from register specified by a particular by the source index and the other source operand is going to come from immediate field.

Now where is this immediate field? So again look at the recall the 32 bit instruction format. For I, it was like 6 bits for op code and similarly the most significant 6 bits will be used for op code of add immediate, then there will be 5 bits for the destination, 5 bits for the src1 the first source specifier then the next 5 bits will be for destination. This particular dst information destination index will be stored in the next 5 bits.

That cover 16 bits and remaining 16 bit will store a constant that is to be added to the content of this register specified by src1 and result is to be loaded in to, stored in to the register specified by dst.

(Refer Slide Time: 21:06)

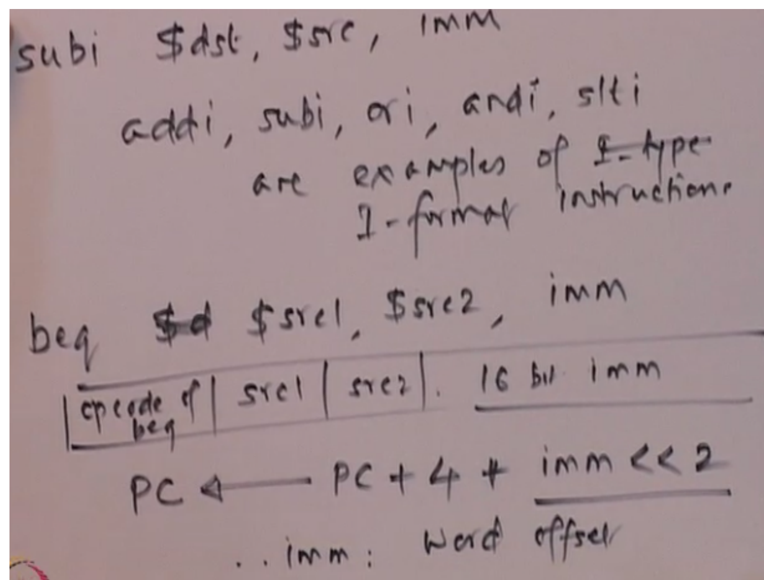


So let us look at an example add immediate it is simple anyway. Add immediate say $4, $6, and say 173 in decimal. So this is going to be 6 bit of op code of add I and then this is the 00100, sorry that is the destination right, so this will be the source and then followed by the index of the destination register that is 00100 and 16 bit, this is an 8 bit number right less than 255, 00001, 128 plus 32, that is 160 plus, this is 13 plus 32, 45 plus 128 right.

So it will be the 16 bits, this is number 4 this is number, $4, $6. So the semantics is that this R, register number 4 is loaded with the addition of contents of register number 6 and this binary number which is this, which stored in this immediate this 16 bits here. Note that compared to, if you compare it with the encoding of the add instruction, add instruction required three register indices, two for source and one for the destination.

Here we require only one register index for source, because the other source operand is going to come from this 16 bits here, and the destination which would have been in the third set of 5 bit indices now it is going to be over here. This 16 bits are going to be free for holding a immediate, I mean holding a constant which is to be treated as a immediate operand, okay, so this is the immediate operand, 16 bit immediate. This is the specifier of direct operand. This is specifier of immediate operand.

(Refer Slide Time: 24:27)



And similar to add I there will be subtract I, with the same format. So this is an add I and subtract I and similarly logical OR, logical AND immediate are examples of I-type instructions, immediate type, I-format instructions, knowing their op code figuring out that its instruction say subtract immediate or add immediate we know that the bits of the instructions are to be interpreted as source index, destination index and 16 bit for immediate.

Unlike in the case of the R-type instructions where we have to look at this, like you know bunches of triple of 5 bits for two source indices and one destination index and the remaining some of the bits can be ignored. There is no rule for immediate operand in the R-type instruction like add, subtract, OR, SLT and so on. So I think SLT immediate is also available in the MIPS instruction set architecture.

Now other than this these are arithmetic and logic kind of instruction that we have seen so far, and I will also mention to you about branches equal to instruction that will have a pair of, there is no role of destination as such, there is a pair of source indices src1, src2 and there is an immediate field. So here the encoding op code of BEQ will be here in this 6 bits and then src1 will be here in this 5 bits src2 will be next 5 bits and this 16 bits will be used for specifying a constant.

What is the meaning of this semantics of this instructions, the program counter which stores the instruction address of the next instruction is going to be updated with current program counter plus four, okay, we will just come to this plus four ignore it for a while, the main the

role of the immediate is this program counter is essentially updated with, this is treated as a relative offset, and this 16 bit number is treated as the word offset.

Remember that I mentioned that this CPU is 32 bit wide, okay. But we are using the addresses at the byte level. Next word of 32 bit is going to be four bytes away and this immediate field is been interpreted as the number of words like, relative offset in terms of number of words. So this is the offset which is to be added to the program counter to get the next value of the program counter that is where the next instruction is supposed to be.

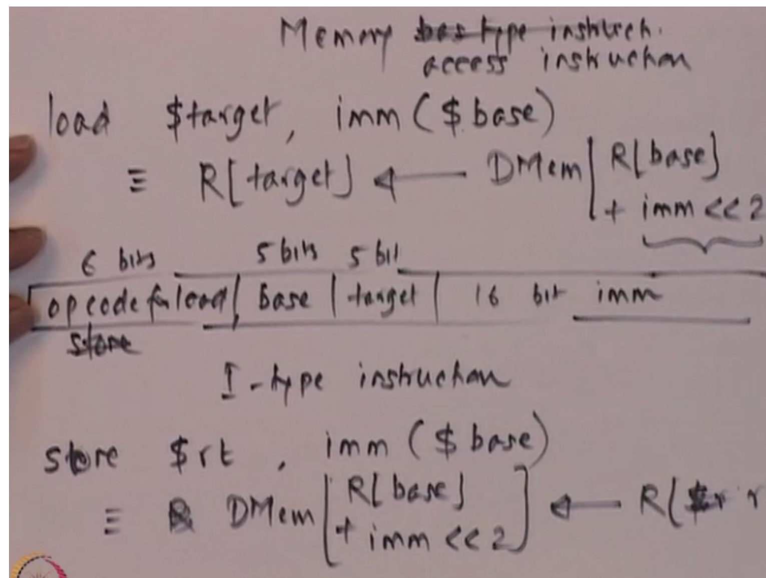
So this is the conditional jump which is of the relative jump kind and but the main thing is that this word offset has to be multiplied by four, so shifted left by two, shifting it left by two will have the effect of multiplying it by four. So this multiplication by four will convert this word offset in to a byte offset, okay, immediate offset in terms of byte address. So assume that the memory can refer to individual byte so the address is referred to different bytes.

So if you want to go to the next word, then you have to change the address by four. This plus four, you ignore it for a while we will talk about it later, it is just one of the subtle features of not too important. MIPS have taken a part of the architecture it was decided that the relative offset would be added to PC plus four. So PC plus four is the default next program counter right, default value of the next program counter.

But so instead of the default is going to updated by the offset, so the compiler or the assembler should make sure that if you try to encode that is of jump address, then it should be just the difference between the address of that place to jump. And correct instruction but rather the difference between the location where to jump minus PC, the address of the next instruction the default next instruction which is PC plus four, that is why this funny thing.

So very quickly we will just windup the couple of other instruction. In fact important instructions that are left are load and store. So for the instruction that we are seen have been of the type arithmetic, logic or the control flow, branch jump is also there, unconditional jump. But let me just ignore it for a while, I mean it is not an very important, we can easily extrapolate by understanding the architecture for this instruction what will be happening for jump, okay.

(Refer Slide Time: 30:37)



So let us look at the instructions which work with the memory, so load and store, these are memory based or other memory access instruction. This is absolutely necessary right, just by providing the ability to do arithmetic and control branch and jump. You are not going to be able to get data or archive data in places, registers are there 32 of them, plus 20 for lots of applications but not in general situations like where you might require lot of data, that will have to be stored in the arrays.

Arrays could be much bigger than the number of registers you have, so you need to make use of data memory and that is why you need to have couple of instructions to provide for that. And one such is load, this will load something from the memory, the syntax is load, specify the target index, index of the register in which you are going to load something. What you are going to load is specified over here, I am just using different kind of names which will suggest the purpose.

So the semantics is the register whose index is given by this target, target index between zero to 31. So this specifies the register from the register file that register is going to be loaded with contents of data memory at appropriate location, which location at appropriate address, that address is calculated by reading out the contents of the register specified by this base and again we have the shifting left by two, okay.

So the encoding of this instruction, load instruction is going to be like this. This is 6 bit op code for load, 6 bits, then you have 5 bit representation of this particular like you know the

base specifier, base register specifier, then 5 bit for specifying target and that leaves us with 16 sixteen bit, okay.

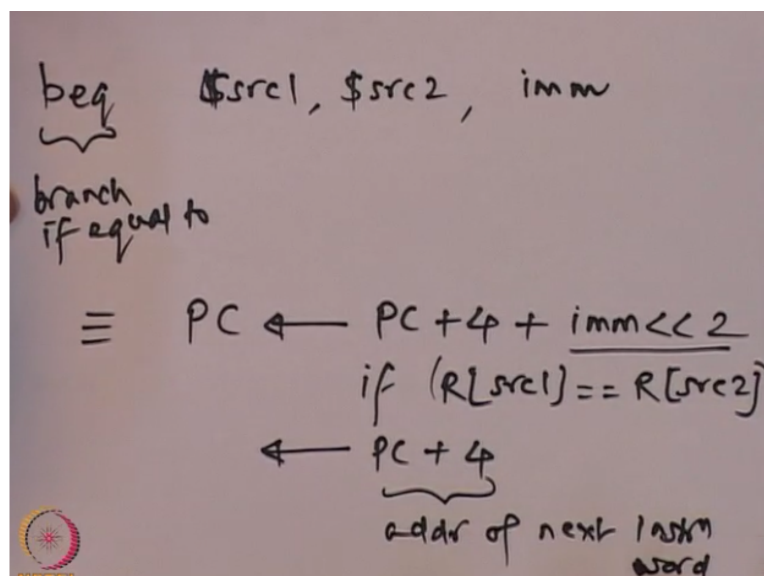
So again now this immediate is going to be regarded as an offset, because this is address calculation right, this is calculation of a address of some location in the data memory and since we have this 32 bit registers, register, if it is to be loaded with something it has to be loaded with 32 bit content. And that has to be like you know a word from memory, 4 byte word. So this immediate, 16 bit immediate field, you treat it as a word offset.

Offset with respect to the base which is specified in this in the register with this particular index, okay. So this is as you can easily imagine that this facility is provided for array kind of indexing, okay. So this is also an I-type instruction, because there is a role of immediate field and complementing load we have a store instruction whose syntax is similar and index specifying the register, and index specifying the another register.

And the immediate field but this semantics is, storing something from a register file in to data memory, so data memory at certain location is going to be updated with the contents of RT. The register with the index RT and which location in data memory, just the way it was done in the case of load. So address calculation is same as that for load instruction. This is going to be the base address that is why we call this register base register plus immediate.

I was just prompted that I missed one point in describing the branch instruction.

(Refer Slide Time: 35:40)



This is branch is equal to, the syntax of this is, it takes a pair of source indices describing the registers, which are to be used as source opponents and specifies an immediate field as an offset for relative branch. So meaning is that PC is to be updated with PC plus 4 plus, conditional right, if the contents of resistance at indices source1 and source2 are equal. So this is the work offset.

We may convert it and to be shifted by 2, left shifted by 2 and this PC plus 4 is the default next program counter that is updated with this offset. So if this is done if the equality holds, you know, the content of these two registers are equal. Otherwise PC is going to be updated with the default, again this plus 4 why, because the instructions are 32 bit wide and address is referred to bytes.

So to refer to the next word, the next instruction we have to add 4. Address of next instruction word. Instruction is word long, right 32-byte long. It is a 32-byte architecture. We can study the variations of this anyway, but right now for simplicity of the presentation, we are assuming data instruction to be 32 bit, but we can explicitly convert the word address to byte addresses because the address is referred or mentioned at the byte level.

Alright so that was just for it. So missed that point while describing BQR instruction in RE. Let us quickly look at the use of this memory instructions, so memory access instructions provides and this immediate field and the source index has to be treated as the base.

(Refer Slide Time: 38:31)

Memory Operand Example 1

- C code:

  g = h + A[8];

  - g in $1, h in $2, base address of A in $3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw   $4, 32($3)     # load word
add $1, $2, $4
```

base register

So here on this slide, you have seen some typical like, you know, kind of C code statement where you have an array A that contains of the 8 location of array A is to be added with the variable h and the result is to be put in variable g. So let us say it as the compiler has a kind of associated g with register number 1, associated variable h with register number 2 and the base address of A is stored in the register number 3, okay.

Now, is this particular c statement is going to be compiled into a MIPS code, then the index A is referring to the eighth word, right, of the array A. So eighth word is going to be at offset of 32 bytes, okay because there are 4 bytes per word. So that statement is going to be converted or compiled into this pair of statements, of course one is for loading something from the data memory.

Because the array A is going to be stored in memory, okay. Only the base address of the array A is stored in one of the registers, specifically the register number 3. Now here this lw stands for the load instruction, sometimes you might have variation called load single byte, but default load instruction is load word, lw stands for load word. We are going to focus only on load word instruction in this particular lecture.

So look at the syntax. It is saying that $4 is specified as a target address that is index of the register in which the data memory word has to be loaded into. So why number 4? That seems to be some temporary location. So you load something from data memory, which is at offset of 32 bytes from the base address stored in register number 3 that is 32$3, 32 is going to be stored in the immediate field of the instruction.

$3 refers to the base register that means #3 is going to be stored in the SRC one field of the first bunch of 5 bits in the load instruction and coding and this number 4 refers to some register, which we are going to use it as a temporary location storage for the eighth word of array A, which we have loaded into, which we arrayed from the memory, okay.

Now for the next instruction what we need to do is we just need to add to h, this memory contents that we arrayed out, h is bound to register number 2, right that we are assuming. So we need to add the contents of register number 2 and register number 4 because that is where we have just, previous instruction, the data has been loaded into. So the add instruction is

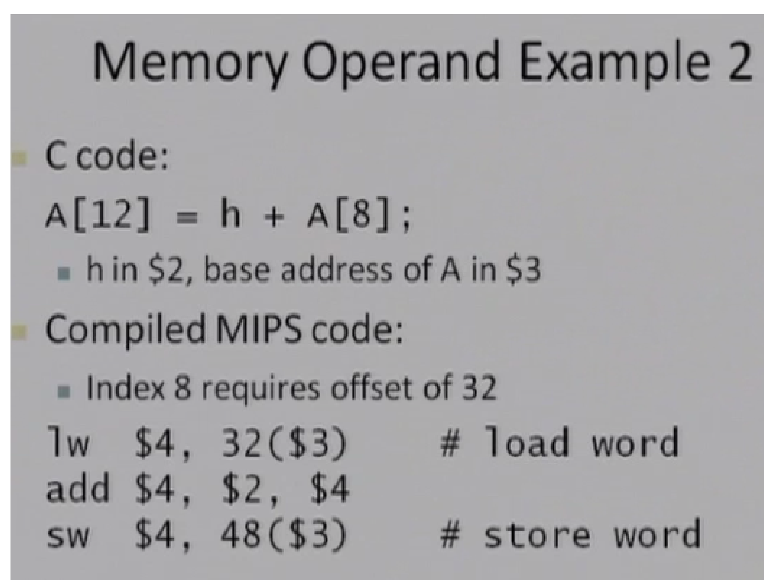
specifying that in updated destination register number 1 with the addition of register number 2 and register number 4, okay.

These two pair of instructions together is equivalent to the statement g equal to h plus 8, okay. This is a simple illustration of how the immediate field is used, the offset is calculated by shifting by 2 and the base of this stored in one of the registers and offsets are specified in the immediate field.

And so like, you know, this shows that such a simple small instruction can also take care of your need of working with arrays, which is the most elementary data structure, but powerful enough to mimic any kind of advanced data structure. So in principle you can write any kind of program with this kind of simple instructions in the sense to complete without need to go into that.

But it will suffice not only for our explanation of fundamentals, but also it is a complete CPU by itself, although tedious to program. We will have to execute a lot of instructions of the simple kind to do something routing, okay. So this was the base instruction 32 is the offset.

(Refer Slide Time: 42:58)



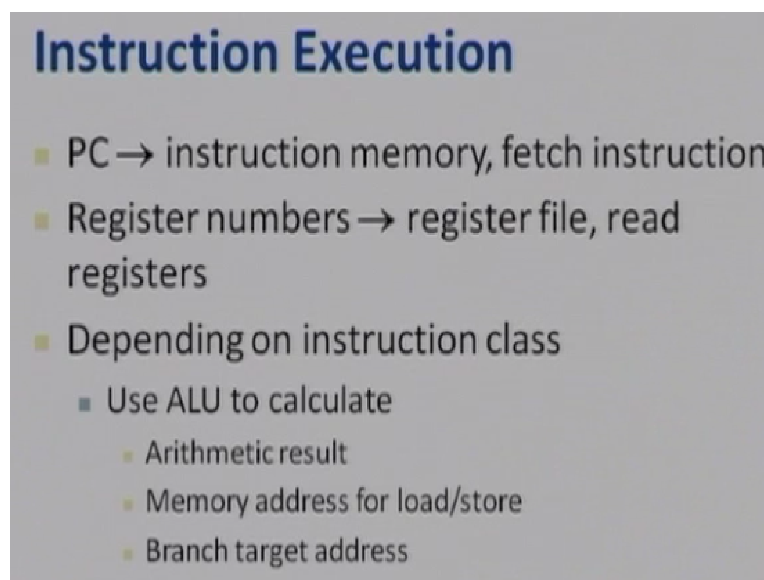
And similarly we can look at a couple of other examples, but I leave it to you to study it on your own. Registers are used compared to the data memory, because they are much faster for access than memory, typically in the single clock side, you can access a register, but for accessing a S-RAM or D-RAM, we require longer time, so we would not go into that again, but just a small point is if you want to operate on data that is stored in memory.

Then first you have to load it into registers, operate on it using an arithmetic or logic instruction and put the result back in register and then store into memory using the store instruction, so load and store would be required other than the arithmetic instructions or logical instructions, so if you want to operate directly on memory, if you want to operate on data and memory locations, then you have use a complimentary pair of load and store also.

So more instructions need to be executed, so it is quite important that comparatively make sure that much of the computations were arithmetic, happens on data, like, you know, most of the data that is required repeatedly, frequently is bound to registers rather than stored in some arbitrary locations in the memory. Arrays obviously have to be stored in memory because arrays are typically large and you do not have large enough state of registers to store big arrays.

But local variables they are to be like, used frequently. It makes better sense to use them in good bind them to registers. Similarly, there is a role of immediate opponent and so on. We have already discussed that

(Refer Slide Time: 44:52)



Steps in instruction execution, what are they? Like you know during a single clock cycle, in which one instruction executes. We will be assuming that our CPU is simple enough that in one single clock cycle and single instruction will completely execute. Next clock cycle, the next instruction will execute, which would have been fetched by using the address in the program counter and so on so forth.

So in the beginning of the instruction execution, program counter will supply its contents as an address to instruction memory. Now there is something called instruction memory and there is something called data memory. So why these two things are to be separate, we will remark on that a bit later. So once the program counter supplies an address to the instruction memory, sometime during the same clock cycle after a bit of delay.

The memory is going to supply its content at that particular location and that would be the instruction, which is to be now processed. So by this time, we can say that we have faced the instruction, okay. Next, looking at the 32-bit instruction, we identify depending on different types of formats of instruction, we identify which of this parts of instruction refer to register indices, source of destination, look at source indices.

Supply them to register file and some mechanism there be like, you know, locate and read out the appropriate registers and bring them on at output of the register file, okay. Now depending on the instruction class, we will use the ALU to calculate either the arithmetic result of what we have just read from the register file, or we will treat the information that we have read from the register file and some part of the information from the instruction itself.

Namely the immediate field and use it to calculate the memory address that is required for load or store. We have seen that immediate field and the source index they together cannot specify the address of the memory location in the data memory, okay. So one part of the address, which is the base address, has to be read, found the register file and the offset is to be obtained from the instruction itself for low as 16 bits of instruction, which are left shifted by 2 bits.

Of course with the sign extension like keeping the polarity of the offset same. so either it could be used for arithmetic result or for calculating the address of memory location for load store, also as best seen in the case of branch instruction, branch equal to target of the branch that address has to be calculated, again by this same ALU, okay. Fortunately, like one main ALU is going to be used in one clock cycle depending on whether the instruction is arithmetic or whether the instruction is load stored or whether the instruction is branch for one of this purposes.

So we do not need three separate ALUs for these three. At any given clock cycle, only one of this kind of activity will be happening. Of course we require a couple of other ALUs that will. I am sorry. This is completely wrong what I said. In fact, we are going to begin with a single cycle CPU and we will be requiring multiple ALUs, just the way we just entered at something called instruction memory and something called data memory.

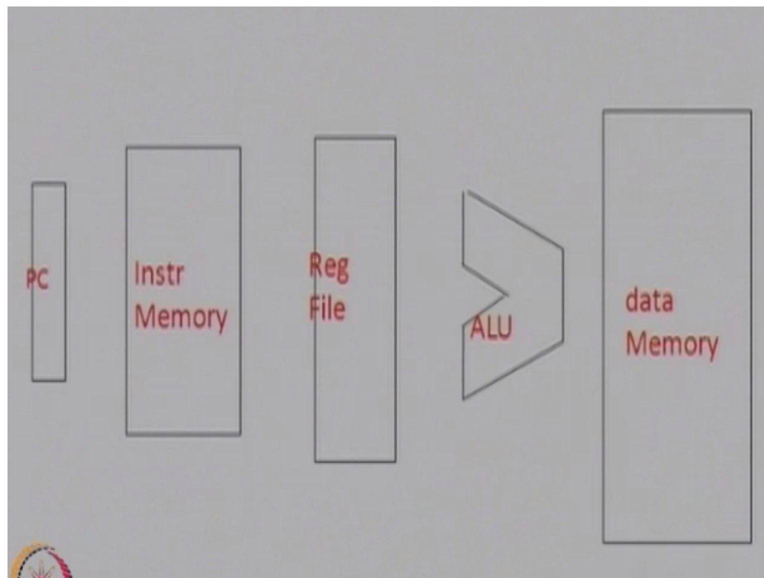
Two separate blocks of memory. This will require multiple ALUs, okay and the reason is that in any given instruction, even if it is arithmetic kind of instruction, we will require to do ALU to do arithmetic at the same time, we will require some other ALU to do calculation of the next instruction PC plus 4. If it were a branch instruction, then we will require one more ALU to add in the same clock cycle, will require one more ALU to add to PC plus 4 the offset.

Offset left shifted by 2. So we will require multiple ALUs. We will soon get a clear picture of that, okay. After we have calculated the address in case of certain memory instructions like load and store, we will actually access the data memory by supplying that address and either taking the data from the data memory from the load instruction or storing some data into the data memory that is on behalf of the store instruction, okay.

In the meanwhile, during the clock cycle, one of the ALUs would have computed PC plus 4 that is incremented PC, this will be of simple ALU, which will essentially be adding constant. It might be optimized added and the result of that is going to be kept ready to be loaded at the end of the clock cycle in to the program counter. So PC is going to be updated with the target address or PC plus 4.

PC plus 4 is a default and in case of branch address, if the branch condition is successful, then another ALU would have added offset to PC plus 4 and that result would be ready to be loaded into PC, okay.

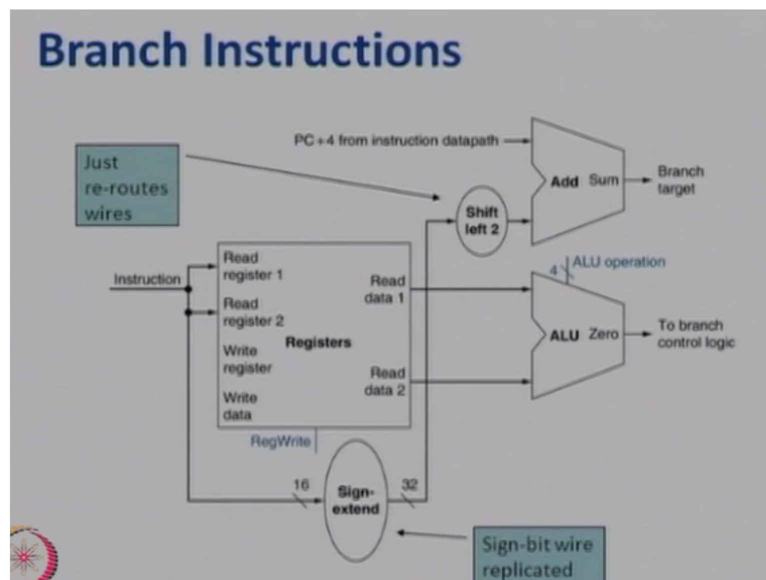
(Refer Slide Time: 50:28)



We will soon see what kind of components will require in the data path, will require a program counter or register, which will be updated at the end of every clock cycle either with PC plus 4 or with a target address or destination address in case of unconditional jump. Definitely, we require memory, but here we see that we require instruction memory as well as data memory, separate blocks. We will see the result for that.

More or less obvious, I will just mention it in a minute or so, then we require a collection of registers, organized in array of registers, which in a standard terminology, we call it reg file, then in ALU we will require more ALUs as we will see this soon.

(Refer Slide Time: 51:13)



We can mention some more things about individual components a bit later. This is how abstract a simple picture of the micro architecture will look like. Note that this is only data

path. The controller will describe shortly a bit later. Again you see the role of program counter, instruction memory, register file, ALU data memory. In addition, you see two more ALUs, which are more specific.

One of them is a very specialized add up, which is adding 4, the one on the top left portion of this slide, and evidently that computes PC plus 4, which is the default next program counter address, but in case the instruction is branched, then there should be a facility of updating this PC plus 4 by adding the part of instruction, the immediate field shifted left by 2 that by another adder and that should be routed back to PC.

This is just a picture. We are not yet looking at a data flow. We will soon look at it over the next couple of slides, but you see the wiring here, the contents of PC will be going to the specialized adder, which is adding 4. PC is going to be at the input of program counter, we have either the contents of the specialized adder, which is adding 4 or the next ALU, which would be adding a part of the immediate field offset.

At the output of instruction memory, we have a couple of wires, a few wires going to the register file, which are basically 32 lines coming out of instruction memory because we are reading our instruction, which is 32 bit long, a few of the bits going to the register file, specifically, those three lines that you see are like 32 bits of instruction are coming out over here.

Evidently these three sets of wires are basically bunches of 5-bits specifying the source indices, source 1 and source 2 and third optional cases like register type indices another bunch of 5 bits, which we have seen in certain instruction, there is a role for it. The green ovals are multiplexors drawn in this funny way. This is some standard convention used in a book by Patterson Hennessy.

We will have to get used to it. Just to keep this diagram less cluttered and more abstract, so we will come to that in a way. This blue blocks are left shift by 2 because we know that we have to take some portion of this instruction, 16 bits, namely the lowest 16 bits, they have to be left shifted in certain situations when we have to use them as word offsets in case of memory at this calculation for load store or the branch target calculation in case of the instruction like branch equal to.

Then, the output of register file corresponding to the two source indices, the two of the registers are read out and they typically are used as inputs to this ALU, okay. So this one is used as a first input and this one is typically used as a second input, but there is a multiples are here and that tells us that the second input to the ALU can come from where. It can come from the left shifted version of part of the instruction, which is for the purpose of shifting the immediate field by 2 bits and using it as a second opponent to ALU.

This will be apt in case of you can see load and store instructions, okay. Similarly, this left shifter will be used for calculation of the branch target arrays adding the offset immediate field shifted by 2 to PC plus 4 that has already been completed by this particular ALU, okay. When you see that disk multiplexer will optionally late either the PC plus 4 or this branch target address calculated with the help of immediate offset.

So this multiplexer is going to be controlled by this situation in the instruction, whether the instruction is branch, whether the branch condition has been found to be successful or not, depending on that disk multiplexer, we will choose whether this one to be set through or this one is to be set through, okay. Coming back to the ALU, this is the multiplexer.

At the second input of the ALU, I have not drawn the complete picture, there are couple of other sources. In fact, this picture is a bit incomplete because it shows that the immediate field has to be left shifted by 2, but that is for the load and store instructions, for address calculation, but for add image instruction, 16 bits of this instruction have to be directly sent over to the second port of ALU. They should not be shifter left by 2.

So another alternate possibility to reach the second input of ALU, okay. In this manner, we can describe the wiring of the data path components. For example, ALU output is to be either routed back as data input to the register files with that data, which is the result of the ALU computation, say on behalf of add instruction or subtract instruction or all instruction are brought in here and it is going to be stored in the register specified by this 5 bits, okay and so on so forth.

(Refer Slide Time: 57:26)

Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

State element 1 → Combinational logic → State element 2

Clock cycle

Anyway instead of showing vaguely, we can look at more specific pictures for different instructions. For example, here in this slide we have marked thick red, flow of data on behalf of some instruction and you should be able to guess for which kind of instruction, this kind of flow of data occurs. So see what is happening over here, what is being depicted is that from program counter, the address is going to the instruction memory.

The program contents also going to this adder, the constant 4 is going to this adder and the result of this adder, that is the one which is going to be routed by this multiplexer, back to program counters. That means at the end of this clock cycle, the program counter is going to be updated with PC plus 4. Apparently in this particular scenario, there is no role of this particular adder. What it does is of no interest to program counter.

Let us look at other part. Program counter is going as an address to instruction memory. The contents of instruction memory are coming out here and 5 bits of them fed over here, 5 bit are fed over here, there is an interesting multiplexer here. We will talk about it a bit later and what we see that corresponding to this 2 bits, this pair of 5 bits, the register file realises, which pair of registers are to be read out on this 32 lines and on these 32 lines.

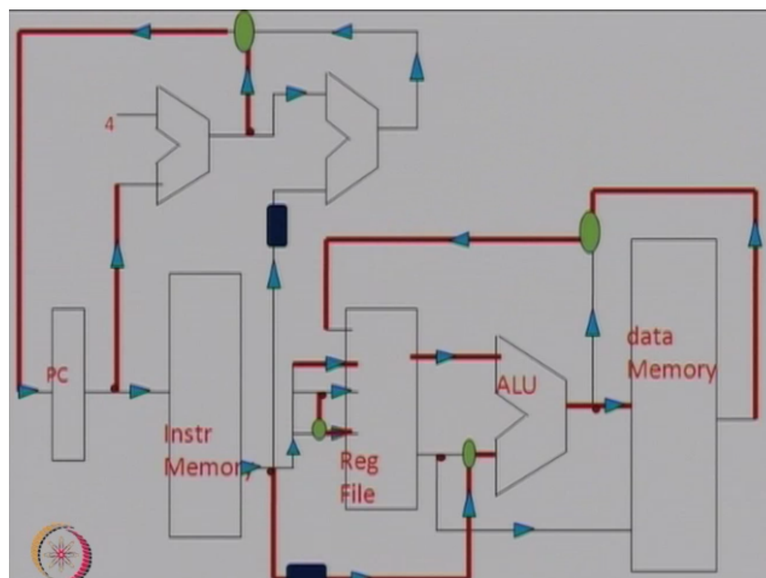
These two pair of 32 lines act as source opponents to this ALU. ALU will work on the contents of registers, which have been laid out here on this lines. The result of the ALU is going to be sent back through this multiplexer back to the register file. Again, you see that there is no role of anything coming out of data memory. It is not going to be routed by this multiplexer to some. So what do you guess?

This is some kind of data flow that is happening on some data processing that is happening in the ALU, a flow of data through these multiplexers from appropriate sources. Many of this data lines are inactive in a sense that they do not seem to matter, so what could be the instruction, which is causing this data movement and data processing. There could be multiple options.

In fact, clearly, it looks like definitely not a branch instruction because they otherwise would have been a role of this. It is a registered type instruction because you see that all these three sets of 5 bits are of use, okay. So this pair of 5 bits are indicating the pair of source registers, the contents are being available at this pair of 32 bit outputs and the result is coming back in to the register files. So it is a R-type instruction where the pair of source operands and result are all like specified with respect to the register file, okay.

The destination is in the register file, the sources are in the register file. So it is the R-type instruction and it could be add or subtract or OR or like you know AND, depending on how the ALU is configured, okay. There are some control signals to be ALU, which will set an ALU in an addition mode or subtraction mode or logical operation mode, okay. So it could be an add instruction or subtract instruction or logical OR, logical AND or set less than for example, okay.

(Refer Slide Time: 1:01:09)



So, next we can look at this example, there is a bit of slightly different data flow. So again you can guess, you see that, it may not be complete in this picture but here this thick red line

should be carried over to this. So this means the PC plus four is computed and is being routed back to and kept ready at the input of program counter. So program counter being a synchronise like you know clock register.

So at the end of the clock cycle program countablity updated and in the beginning of the next clock cycle we will be effectively using this new address as the address of the instruction memory. Next instruction will be fetched out and it will be processed. So this is what is happening over here, again since there is no role of this thing, this cannot be a branch instruction.

But you see that the instruction that has been read out, fetched 5 bits of this instructions are going over here, though I have not show it too clearly, the next 5 bits are going over here. So this is the destination this you can guess, I deliberately not shown this labels because I am treating this as exercise for you to a lot of guess work and get more familiar with the data path of this and it is basically quite simple, quite easy to work out from scratch and assimilate the understanding of that.

So there is a 5 bit index for specifying one of the source operands, the next 5 bit is going to be used as destination address. Now look at the ALU, ALU receives the, ALU processes as one of the source operands the result from the register file, which is basically the first source operand specified by this, the index specified by this particular 5 bits, okay. The second set of 32 bits from the register file is of no interest.

What the other input, the ALU is using is coming through this multiplexer, this multiplexer is allowing this input, which is clearly the 16 bits of the instruction that has been read out. So its immediate field shifted left by 2 bits right, it is passing through this left shift combinational block it could be a barrel shifter. The result of the ALU is clearly in this case, it must be the address right.

Address is being sent out to the data memory and the data memory is accessed at that address and the contents of data memory are routed by this multiplexer unlike in the previous case the disc multiplexer is taking the contents from data memory, whereas in the previous case for the add or subtract instruction, this multiplexer was taking the result of the ALU and it was been sent to this data input port of the register files.

So this is the 32 bit data that is going to be latched in to a register specified by this target information, okay. Over here, this multiplexer is the one, which is going to take care and not let this 32 bit data which is irrelevant and instead let this appropriately shifted immediate field in to the second port and compute load at the memory address, computer memory address and so on so forth.

So this must be, since it is reading the data memory like it must be a load instruction, if it were writing in to the data memory it would be a store instruction. So this data flow must be for this configuration of data path must be for the load instruction, load word instruction. Now what about this one as you can guess, there is a role of data memory, something is being provided here the address has been provided and here the data is been provided.

Where is the data coming from, the data is coming from this second output of this register file, which basically is the content of register specified by this 5 bits, the second set of 5 bits over here, okay. There is no role of destination address over here, because register file is not being written in to, where as the two like you know, two registers are being read out from the register file. First of them is being used as base address, that base address is being added to the word offset and that becomes the address information for data memory.

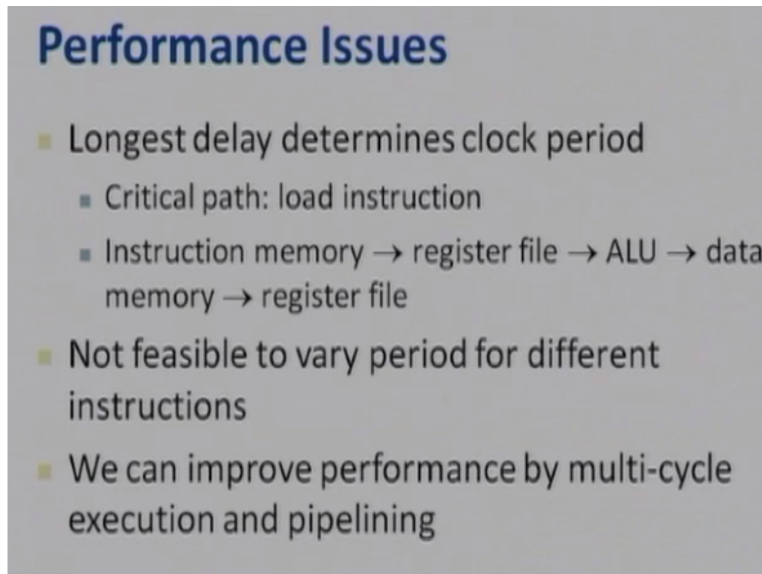
And the data to be stored into the data memory is coming from the second register specified by the set of 5 bits, okay. So again no role up this branch related ALU. This is a store instructions data path, okay. So this way like you know one can study that, one can analyse that this particular data path is more or less adequate of course, there is one or two minor things are missed out here, that is optional.

For example, I mentioned that if we were to show add in the simulation or how the data flows for add immediate instructions, then we would require the 16 of the 32 bits coming from the instruction memory to be routed without shifting into the second port of ALU. This multiplexer will have to be big enough to be able to either send this 32 bits or this 16 bits with sign extension.

I did not remark on that or like the immediate field without shifting. These three possibilities have to be supported by this particular multiplexer, so it has to be at least three to one

multiplexer, this has to be two to one multiplexer, this is another two to one multiplexer, two input one output multiplexer and so on so forth. As an exercise you can sketch out the, trace out the configuration of data path or the data movement for the branch instruction, branch equal to, it is quite simple.

(Refer Slide Time: 01:07:41)



So with this I will stop, the last comment that we need to like understanding the performance issues we notice that the longest delay is the one that determines the clock period. So the longest delay is along critical path, the critical path is the one which causes the longest delay of combinational logic and here intuitively it is clear that load instruction is the one which has the longest delay, because that is where the lot of data processing and data movement is happening.

In particular, for the load instruction, instruction memory is being read out, the contents of parts of the instruction that is being read out are going to the register file, register file warms up, it supplies the like you know supplies one operand to ALU the other operand comes from the instruction itself. ALU completes the address of the memory location from which we have to read the data. So that address is to be sent to data memory, data memory has to kind of take its own time generate the read out the data for you and that data has to be routed to the register file.

So there are some five sub stages in a load instruction, so this seems to be the longest instruction even compared to longer compared to store and other instructions. So evidently the critical path is going to be decided by the way data moves for load instruction and then

we realise that for many other instructions like things are must simpler for branch equal to there is much quicker completion of the work of this processing of that particular instructions.

So the instruction, the clock cycle which is long enough for the longest instruction might be bit of waste for the instructions which would have prepared a result much earlier, you know branch target address would have completed much earlier. By the way there is a role of other ALU in the branch target instruction that you can see when you do the exercise yourself. So there is a performance issue here.

The clock period is bad enough long enough for accommodating the longest instruction. It is not feasible to have varying period for different instructions, sorry for the typo here spelling mistake, that is fine. But we can improve this performance by multi cycle execution or pipe-lining. In the next couple of lectures, we will talk about multi cycle version of the CPU, which is where we see the role of finite state machine and that is what we wanted to discuss mainly.

This was just to background, setting up a background of CPU architecture, micro architecture that there is a data path and similar data path will be used with a bit of changes and it would be adopted to multi-cycle execution with the help of finite state machine which will act as the controller of the data path. Here if you take a closer look the control of the data path means you know control of the multiplexers, control of the ALU which is all combinational.

In a given cycle knowing the instruction, we know completely how the multiplexers have to be controlled, how the ALU has to be controlled and how the memory has to be controlled. So there is no need of a statement, any kind of state information inside the controller itself, controller is purely combinational. I will stop here.