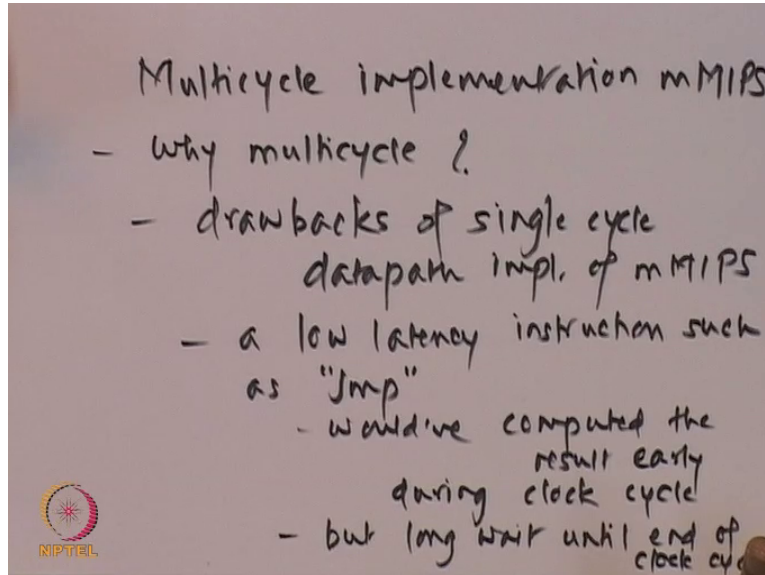


Advanced VLSI Design
Prof. Sachin Patkar
Department of Electrical Engineering
Indian Institute of Technology – Bombay

Lecture - 27
Multicycle MMIPS

(Refer Slide Time: 00:26)



Hello, welcome back. Today's topic is Multicycle implementation of our scale down version of MIPS processor which we call mMIPS think of it as microMIPS or miniMIPS, so why multicycle previously we have seen Single cycle datapath and although it was quite simple to evolve understand and analyze, there are obvious drawbacks of single cycle datapath, single cycle datapath implementation of mMIPS.

The most striking drawback was that even for low latency instructions which would have finished their, which would have finished computing the result very early in the clock cycle though even those instructions would have to wait until the end of the longest clock cycle to commit their results into somewhere some of the state elements memory elements like registers, or memory blocks, so for example a low latency instruction such as jump unconditional jump.

It would have found this decoded the instruction found the target address very quickly, very early in the clock cycle but then it has to wait until the end of the clock cycle, because we are having

an edge triggered synchronous sequential circuits it has to wait until the end of the clock cycle which is quite long wait until it commits the results before which the other instruction cannot start next instruction cannot start.

So a low latency instruction would have computed the result which in this case it simply means it would have simply found decoded the instruction and found the target address where to jump to unconditionally and it would be now ready to load that target address into the program counter. But it has to wait early during the clock cycle result would have been computed but long wait this which is wasteful until end of the clock cycle okay.

So this is quite a waste of time of course how long the cycle has clock cycle has been defined to be or designed to be would have depended on the longest latency instruction. In this in the case of microMIPS we have seen that it would be the instruction load word where which has to kind of go through several phases fetching the instructions, then decoding it.

Then completing the address of the memory from where to load from for that it would use ALU, then accessing the memory block. And then after the memory block returns the data committing the result into the appropriate register in the register files. So this is one this is the longest apparently the longest running instruction and clock cycle would have been designed to be long enough for such a long running instructions.

And but for low latency instructions like jump or conditional branches like most of the CPU cycle is going to go waste or even for little faster instructions then load say store or say register ALU type instructions like add or add immediate the work will be done fairly early in the clock cycle fed little before quite early enough before the end of the clock cycle. But there will be a wasteful wait until the end of the clock cycle for committing the results committing up by that I mean putting the results back into appropriately chosen like registers or memory block.

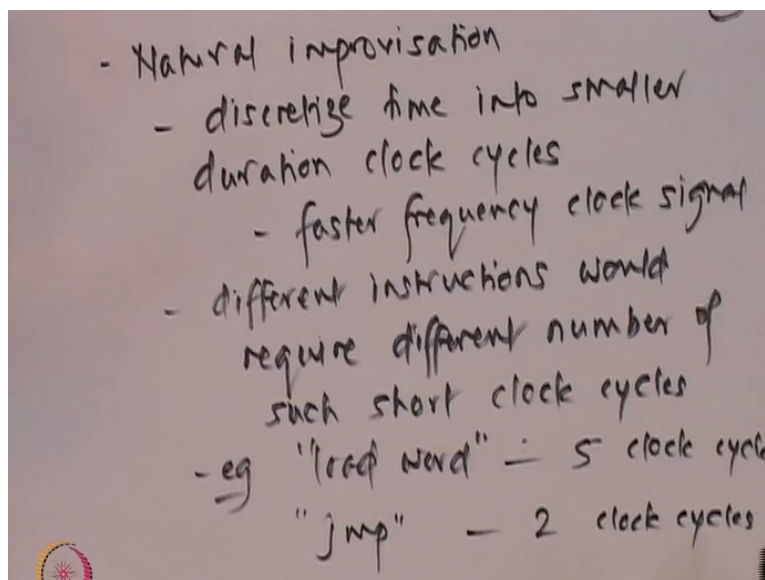
Maybe, for example, store instruction would be committing the result into one of the specify specified locations in the memory block data memory. So this is a drawback like there is a wastage of time okay and the reason was that this the length of the clock cycle duration of the

clock cycle of chosen to be long enough to accommodate the execution or processing of the longest running instructions like load word.

Other instructions had plenty of time but had just to wait so what will the way to think about it is that the continuous time has been discretized or quantized into very coarse very big chunks of clock cycle chunkiest clock cycle of specified duration, the duration is bit too long okay for many instructions, so this discretization of time into clock cycle has been a bit too coarse bit too wasteful in the in case of single cycle implementation, so that is what we want to overcome.

So the natural way to overcome like you know to improvise on this would be to use smaller chunks you know to discretize time quantized time into smaller duration clock cycle that means faster clock cycle at a faster frequency.

(Refer Slide Time: 06:42)

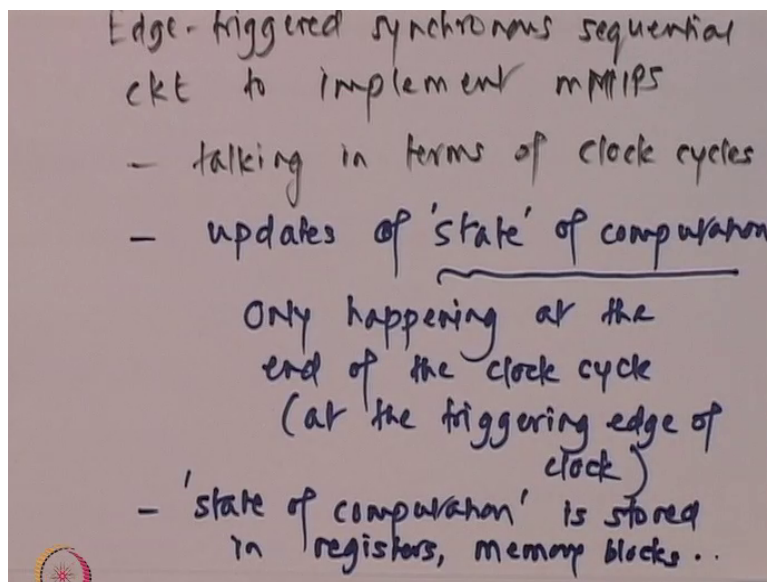


So natural approach Natural improvisation could be discretized time into smaller duration clock cycles that is faster frequency clock signal, so now definitely like because say we are now going to plan to use 4 times faster clock or 5 times faster clock like these long running instructions like load word would not be able to finish in single clock cycle every different kind of instructions are going to require different amount of time.

So load word would require 5 clock cycles let us say unconditional jump can finish execution into of 2 such short clock cycles and so on, so different instructions now, different instructions would require different number of clock cycles may require different number of clock cycles such short clock cycles. For example, as I mentioned we hope that we can get load word we will see that shortly.

It will require 5 clock cycles we just describe how exactly will get this unconditional jump to finish in 2 clock cycles, so obviously number of clock cycles is to be more but then this duration of each clock cycle is proportionately shorter maybe 4 times, 5 times whatever there will be little bit of overhead. But so you know but what we are saving on me is a lot of waste that would have occurred in the low latency instructions like jump or branch and so on, so let us say more details about this.

(Refer Slide Time: 09:24)



Edge-triggered synchronous sequential
ckt to implement mMIPS

- talking in terms of clock cycles
- updates of 'state' of computation

Only happening at the
end of the clock cycle
(at the triggering edge of
clock)

- 'state of computation' is stored
in registers, memory blocks..

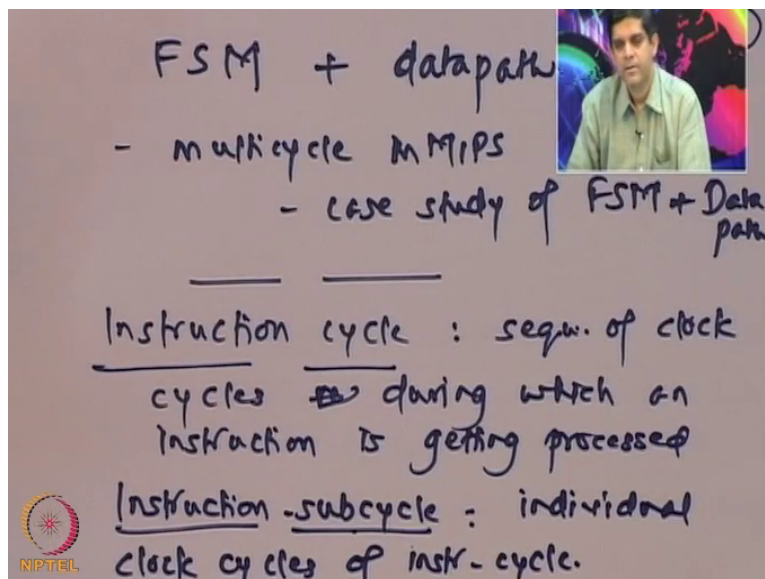
So first on a side note just note that we have been talking about edge triggered synchronous steady single clock sequential circuit to implement mMIPS okay, this the methodology of clocking methodology of using edge trigger synchronous sequential circuit is something that makes a lot of design process, implementation process, extremely algorithmic and synthesis process, efficient and easily analyzable, easily verifiable.

So there a lot of advantages of this over the asynchronous methodology, so we have kind of without any further questions we are going to assume that we are using edge triggered synchronous sequential circuits and that is the reason we are talking in terms of clock cycles okay, the time is discretized into multiple clock cycles okay, so updates of this so called state of computation so this is a very I am using this notion very vaguely the state of computation.

The state of computation which is stored in the registers and memory blocks that state of computation is getting updated only at end of the clock cycles only happening at the end of the clock cycle because that is where the triggering edge is there, that is at the of clock synchronizing clock so it could be if you are using the negative edge triggered discipline it could be at the falling edge or it could be at the rising edge if you are using the positive edge triggered discipline, okay.

So this is and as a remark before this state of computation that MIPS is carried out is stored in is held in registers, memory blocks and so on register file, okay.

(Refer Slide Time: 12:56)



The slide contains handwritten text on a dark background. At the top right, there is a small video inset showing a man in a light green shirt. The main text is as follows:

FSM + datapath

- multicycle MIPS
- case study of FSM + Data path

Instruction cycle : seqw. of clock cycles ~~to~~ during which an instruction is getting processed

Instruction-subcycle : individual clock cycles of instr.-cycle.

In the bottom left corner, there is a small circular logo with the text 'NPTEL' below it.

Next, so one of the things that we want to highlight through this case study of microMIPS and multicycle implementation is that it is a very important example of the concept called FSM + datapath, FSM with datapath FSMD our multicycle implementation is going to be the case study of this FSM + Datapath, so we will not very rigorously very formally like highlight the FSM

aspect of it but we will very clearly see there is an FSM controller controlling the so-called datapath.

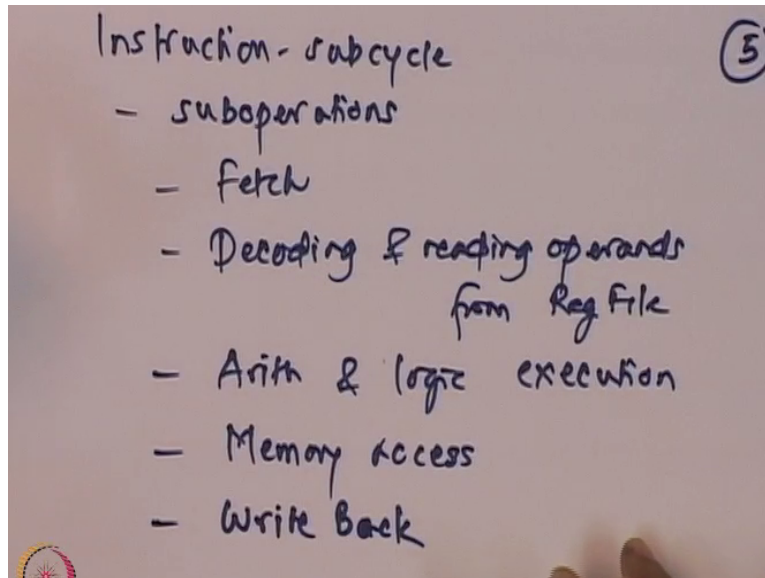
Datapath as we said remarked on datapath is a collection of datapath components which process or route the data and with how which how the data is going to be process, how the data is going to be routed or stored or latched is controlled by a controller and that controller is a typical good interesting example of FSM. Overall the whole computer itself is an FSM of course but then the concept of state means all that has been stored in the memory which might be relevant for future computation that becomes notion of state which is much bulkier notion of state.

And so one typically likes to think of the core notion of the state and core state machine core FSM and the data processing part like you know left to the datapath part of the sequential circuit. So the whole FSM + datapath can be regarded as a huge FSM but that is impractical to study or impractical to like analyze, so this is the typical partition of a sequential circuit in to a core controller and a core datapath that is what we will be emphasizing through this case study.

So there is I will be using this notion or terminology instruction cycle and instruction sub cycle, instruction cycle is the sequence of clock cycles required or during which an instruction is getting processed and you know in case of the multicycle implementation the instruction cycle will consist of number of clock cycles and each one of these clock cycle I will refer to as instructions sub cycle.

Instructions sub cycle are those individual clock cycles of instruction cycle, in this individual clock cycle in this instruction cycle there will be something specific happening you know so that we have to clearly plan for clearly defined and then what are the results of those sub operations, where are the archived, who is going to use them that is what will be that architecture will be designing in this lecture.

(Refer Slide Time: 16:51)



So what are these what is happening inside instruction during instructions sub cycles let us use our intuitive understanding of CPU architectures from prior courses or something that this you might have seen before but let us recall, so what is happening inside instruction sub cycles sub operations okay, some well-defined sub operations and as your natural guess would be the well sub operation would be like in this context fetch sub operation, decoding of instruction and reading the operands.

Operands from register file that will be a sub operation which will have concurrency within that the next will be typically the Arithmetic or logic execution for whatever reason if it is an add instruction then the in this stage the operands which have been read in the previous stage would be added up when the result will be prepared and archived somewhere in case of memory or load store instructions.

This particular stage will be calculating the memory address following this if needed there will be a stage of memory access sub operation here memory is access for the data whereas in the fetch case also memory is access but that access for the instruction following this optionally like in some cases of few instruction cycles there will be a stage sub operation called write back.

So this is the stage where like certain kind of results are being stored back into registers okay, so it is a kind of commit stage it is not that this is the only stage in which results get committed the

results might be getting committed in earlier stages also for different operations but this is the most typical commitment stage committing these results, so clearly not every instruction cycle would go through all the sub operations clearly.

For example, if it is just unconditional jump instructions then it would go through fetch sub operation it would also go through decoding sub operation it may not make any sense to read anything out of the register file for that instruction but yes it will have to kind of break up the instruction and look at the target address do some kind of you know the processing of that get the like extract the target address.

And then arrange to load it back into the program counter that is the commitment part of unconditional jump which must be happening in the second clock cycle second sub operation itself. So since first clock cycle of unconditional jump will be doing the fetch some operation in the second clock cycle of the instruction cycle of unconditional jump we will be doing the decoding which will essentially be understanding that it is a jump instruction.

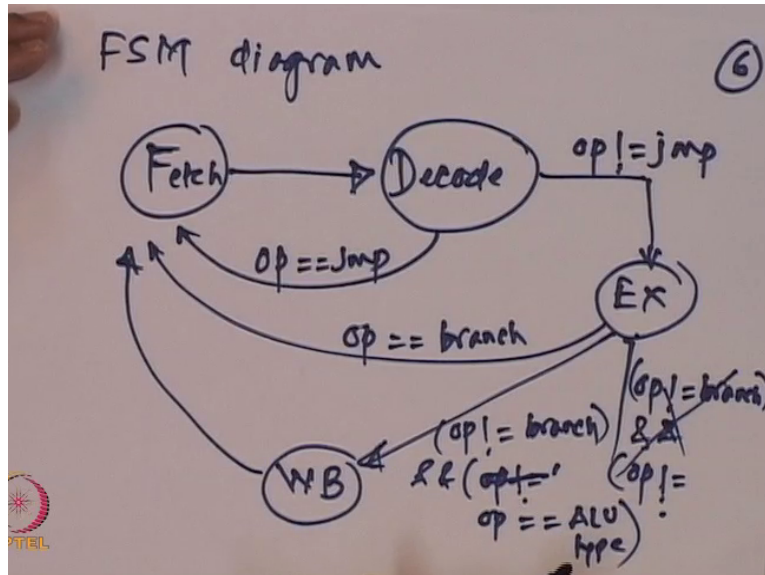
And it also extracting the target address and that target address would be registered into the program counter. So that is the commitment part for which it may not need to wait for the write back special stage in the second clock cycle itself while decoding extracting target address at the end of that clock cycle program counter can be updated, so depends so this is our intuitive view of what is sub operations are and like you know every one of them is going require typically uniform similar amount of time.

So we have we will be allocating 1 clock cycle for every such sub operation, so in case of like I said in case of unconditional jump the instruction cycle will consist of 2 clock cycles 2 sub cycles for this 2 sub operations fetch and decoding and while decoding at end of decoding immediately committing in the same clock cycle in case of some longer latency instructions like store word the instruction cycle will go through all the 5 sub operations.

And there will be 5 clock cycles required for that. That is how, that is why we remarked earlier that in the multicycle we will hope to or we will aim at doing something in the 5 clock cycle,

something in the 2 clock cycles appropriately making more efficient use of time which is discretized in smaller quanta, so that was an intuitive picture how we had already planned to break up instruction cycle into sub cycles and each sub cycle will be doing approximately what.

(Refer Slide Time: 22:11)



So we can now think in terms of an FSM, FSM diagram let us see how to capture what we have been thinking about what we have been analyzing about this, so it is like we can record this sub cycle sub operations happening in different clock cycles as like you know control by different states of the controller, so this is a state called fetch or I will abbreviate by F this is the state that CPU goes into when it starts execution.

Because it has to start fetching an instruction and then decoding it, executing it, optionally doing memory access, write back and so on, now after doing its work in the fetch state of the controller this state machine the controller would go into a state called in which it will control the decode related operations decode or reading the operands or in some cases immediately after decoding putting the target address into the program counter all this will be all such control signals will be generated in this state depending on the situation.

So what is the we are assuming that you already have some exposure to FSM so we know that you have an idea what we are designs FSM for not just for abstractly capturing the behaviour of this the controller but also to demarcate which control signals will be generated which will be

asserted which will be deasserted in which state of this controller, so during decode some control signals to the datapath will be activated, some control signals will be deasserted.

And based on that the data will flow appropriately or the data will be archived in the appropriate place okay, after decode okay supposing it is - if it were the jump instruction then after decode the operation where jump then after decode it will go back to the fetch state for getting the next instruction, okay. So if the input condition whichever which is defined by like you know the value of these signals that has been generated during the decode process.

If that happens to be that have indicated that it was jump instruction, then ident after decode state the instruction cycle would be over and we will go back into the fetch state to start the next instruction cycle. If operation or the instruction were not jump okay then you definitely go into the state called EX of the controller.

The controller now will think of itself being in the execute stage when it will try to control how the ALU is to be used for arithmetic operations on behalf of say add instruction or add immediate instruction or use ALU for like generating the memory addresses using the base register contents and the immediate offset. So controller would be in the EX state for most of the instructions except in jump in the third clock cycle.

If it goes in the third clock cycle that is what I am depicting here, after EX certain instructions can certain kind of instructions would have finished the instruction cycle for example a conditional branch. Conditional branch we can arrange the datapath and control signals in such a way that we will be able to we know that the conditional branch all that we need to do is done and we can commit the results and go back to the go to the next instruction.

So if it is some kind of branch instruction which is conditional branch, jump is the unconditional jump. Then at the end of EX execute stage we would arrange to commit the results what would be the kind of results in case of conditional branch instruction the result that we have computed is just the target address where to jump to which then address of the next instruction to fetch that is the all that result is and where it does need to be committed.

It would get committed into the program counter, no other register is going to be updated, no other memory location is going to be updated when the operation was branch conditional branch instruction, okay. Because in the execute stage it would like try to see check whether the condition is true or not whether the operands were equal to each other or not in case of branch if equal to, in case of branch if not equal or branch negative.

It will check operand values and do comparison or whatever, so that is what it would have done in the prior clock cycle in the decode phase itself it would have prepared what should be the tentative target address. So at the end of this 3 clock cycles of work it would have got all the information to decide precisely where to jump to, either to jump to the next instruction itself or to jump to an instruction specified by the target address, okay.

So this is how like state transition would take place under some situations. So now if the opcode where not branch we are in an instruction execute stage of an instruction if the opcode were not branch and also let us say it will also not and opcode were not load no okay after EX stage where do we go I think if the opcode were not branch and if the opcode where sorry ALU type like you know add instruction or like add immediate and so on.

If this is - this where the situation that we found ourselves in the controller found itself in then it would go to what like you know after completing the execution of ALU type instruction which is done inside the ALU, the ALU results are out where do we go we go to the stage called write back the state called write back okay, this is where we will go and in this state the results of ALU would be picked up and put in appropriate register which is chosen by the instructions some bits of the instructions okay.

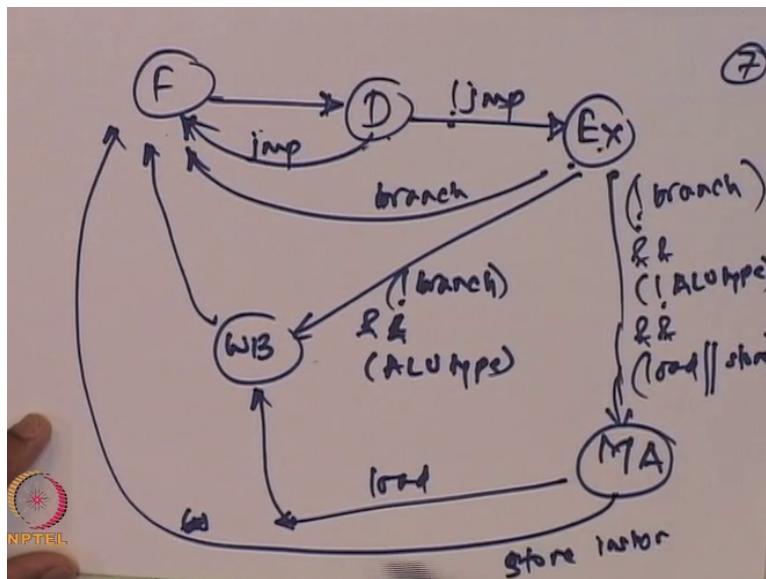
And after write back it will go to start the next instruction okay, so this will happen if at the end of the EX state we have the opcode was we have the controller founded it was not a branch instruction but it was an ALU kind of instruction add instruction or say less than or which were instruction which would have completed the results in and now which we will be ready to write back the result in to register file an appropriate register then the next state would be WB, okay.

So this is how the state diagram is going to evolve, I am not giving the complete state diagram it will be slightly more complicated than this we still have to consider other situation like we have looked at two possibilities out of EX state if the opcode were branch, first of all will be EX state only if opcode were not jump, were not unconditional jump, okay.

And if we are in the EX state then there are possibilities like is it a branch instruction in which case at the end of EX state we know that we will be committing the program counter with appropriate like address of the next instruction to execute or and we will be and will be done the instruction cycle will be over if this opcode were not branch but it were ALU then we just have one stage to one more sub operation to take care of that is write back, okay.

There is no memory access required in this kind of situation, so we will have to modify this state machine further to handle other possibilities out of EX state those possibilities are if it is a if it were a load or a store memory kind of instruction let us see just to as an exercise how this FSM evolves.

(Refer Slide Time: 32:02)



So like you know we had this fetch state unconditionally like you know just on the when next clock cycle starts we go into the decode state we meaning the controller then if it were jump instruction and then we know that the instruction cycle is over and we go back to the beginning

of the next instruction cycle that is starting with fetch instruction fetch sub operation then if it were jump instruction.

And then we know that the instruction cycle is over and we go back to the beginning of the next instruction cycle that is starting with fetch instruction fetch sub operation, so otherwise we go to EX stage right that is not jump I am just using very loose notation like assuming there are similarity with designing finite state machines from earlier course or from computer architecture whatever you might have seen.

And then there are under some condition that is if it were a branch instruction conditional branch the instruction cycle is over if it were not branch but okay but ALU type instruction then all that remains to be done is write back and then the instruction cycle is over, but now we have situation like if neither branch nor ALU type but load or store okay something like that load or store, so if it is a load or store instructions then we this next sub operation would be something that accesses memory.

So the controller will go in a state and in this state controller we generate some control signals which will control appropriate operation to happen inside the memory, so in case of load instruction the memory is to be read, in case of store instruction the memory is to be returned with appropriate data okay, so now if it were a store instruction memory access itself is a commitment like sub operation.

Because in case of store instruction once memory is returned the instruction processing can be regarded to be over okay, everything that instruction had to do is done, but if it whereas load instruction opcode were load then we have to go to sorry we have to go to this write back stage or write back sub operation, because what we have read out from the memory in case of load instruction that has to be now committed into a chosen specified register in the register file.

So that would happen for that to happen the controller will go into this state write back state from within which it will generate appropriate clock signals to like you know to instruct the register file too large or to register the data that is coming from memory into a specified register, a

specified register would be enabled other registers would be disabled and the data that is coming into from memory data memory would be put in that enable register one so on.

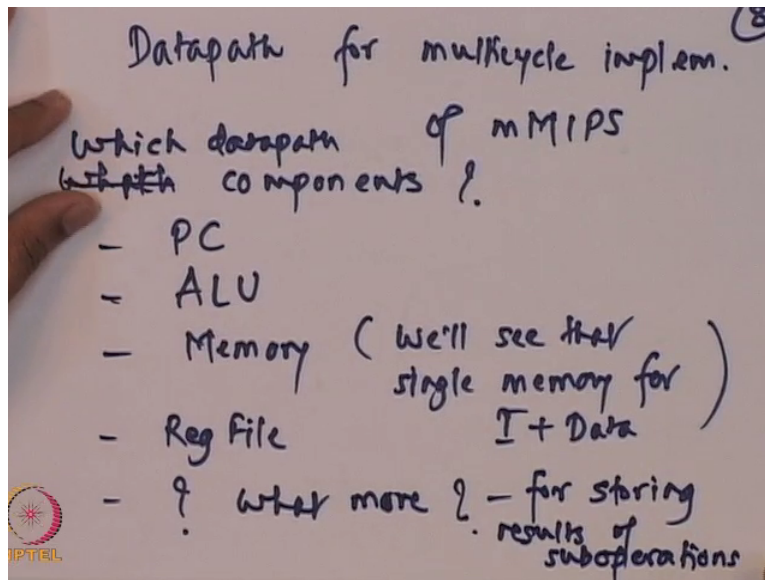
So now you believe that we have a rough picture of what is happening inside this FSM okay to complete the picture we need to really specify how all the control signals are asserted or deasserted and when whether they are more type outputs which depends purely on the state or they are also based on the current inputs many type outputs based on that further notation of this FSM diagram will be done.

And we will get a complete picture when we work out something in detail in one of the later classes for like synthesis oriented implement description implementation oriental description which we can test it out on FPGA implemented test on FPGA, we will make a at that time we will do the exercise completely and describe this FSM in a full in the full glory okay, here this is only for the sake of getting a picture of it, okay.

Because we need to go and now like revisit datapath aspects of this design okay, so anyway like you also recall that while talking about single cycle we did not have to talk about such an FSM which is not too complex but still it had so many states of course even there in a single cycle implementation of CPU during every instruction processing there was sub operation fetch, decode, execute or whatever.

But they were not happening on clock boundaries they were all happening all of them were happening together inside a single clock cycle okay, so they must have been some price to pay for that of course the clock, clock cycle that means lower design plus there was one more one more price we paid which we did not realize at that time but now we will see that using this multicycle approach we will be able to save on resources will see how.

(Refer Slide Time: 37:45)



So you know to understand what is a related datapath for our multicycle implementation or design an implementation of microMIPS or miniMIPS this is slide number 8, we will take a stock, we will try to see what we need in our inventory to prepare the datapath.

What are which components of which kind of components required of course we require register for storing program counter which sorry which datapath components we require program counter definitely, we require like before ALU in absolutely indispensable, then we require memory to store instruction and data, note that in case of a single cycle we required separate instruction memory and separate data memory.

Because we assume that the memory has only single port and because in case of single cycle datapath we had to access instruction memory for both instruction and data in the same clock cycle which is not possible in a single ported memories, we had to have separate blocks for instruction memory and separate block for data memory.

But here we will be able to show that we will be able to time multiplex a single block of memory for accessing instructions as well as accessing reading writing data okay, so we will see that single memory for I and data okay that is one going to be one of the resource optimization which would be feasible because of multicycle approach you probably already have an intuition about why, but just we will go through it anyway.

What about the other things like data instruction memory ALU, PC, register file of course which has which hosts 32 registers for our like to facilitate our instruction set architecture, where we could use as many as 32 registers a file or a block of so many registers that stored in this so called register file which is a standard datapath component, what do we need more okay.

We might like in the single cycle datapath we require several ALUs to do different things like you know to compute $PC + 4$ program counter value + 4 to compute target address of branch instructions or like you know typically we require ALU for computing the memory addresses or the address is of the ALU type or the result of the ALU type instructions.

So multiple ALUs were used some of them were a weak simple ALUs just uses the adders, some of them will just maybe swift maybe simple simply more simplified adders, customized adders but they were ALU we can regard them as like some datapath components of the ALU type, here we will be able to show that again we can optimizing the resources.

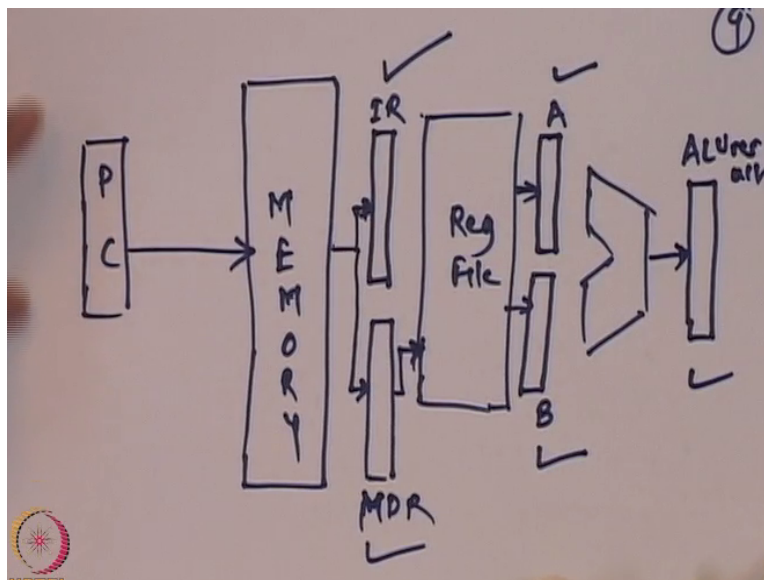
And use a single ALU for instead of using three different such arithmetic components single ALU will suffice in case of multicycle implementation that is again big evidence of big plus for multicycle implementation the resource optimization, so other than that do we need more so they must be some typically cost to pay and what we might need something more and that more that we did not require in case of single cycle datapath this will require for storing or registering results of sub operations, okay.

Let us understand what we mean by that we require something more, something more in the datapath for storing the result of sub operations you know this what we the datapath or registers or memory that we used in single cycle implementation was used for archiving the results only at the end of the instruction cycle the when the instruction is fully executed, whereas now we are going to instruct execute instructions in sub cycles in more than typically more than 1, 2, 3, 4, 5 clock cycles.

And so sub operation will be happening in a given clock cycle, so the sub operations results also have to be archived have to be stored for later use in some registers can its fair amount of common sense intuition to see why we need to do like you know store the results in registers even like for this intermediate sub operations.

Because otherwise if we get to leave them on the signals without registering them they might get overwritten by some changes to it at a later time during the instruction cycle, so it is always a good practice like whenever you have a result of sub operation ready latch it or register it into some registers and for that we will require some more registers other than PC and set of registers in register file, okay. Hopefully that cost will not be too much we will see exactly how much but whatever we will go ahead with it.

(Refer Slide Time: 44:09)



So we are saying that our datapath would have a program counter ready which can supply address to the memory for fetching an instruction okay then and this memory will be able to kind of will establish or will convince you that I will convince this memory can hold both data and instruction we do not need separate instruction memory and separate data memory, these are register file which holds 32 registers each one of the 32 bits on the datapath.

There will be ALU on the datapath as before as in the single cycle there will be required multiple smaller one area and couple of other smaller adders anything else no, this seems to be the thing

that we need from our experience with single cycle datapath but I am saying that now we need something more.

So what that more is we will need something this memory is being used for both instruction and data so in certain clock cycle these memories output is going to be okay the input one of the input to the memory as an address is the contents of the program counter, there is another possibility of supplying an address to memory but this is one possibility and when address comes from program counter the interpretation of what comes out of memory is something that is an instruction and that we will like to store it into a register called IR, okay.

So in the fetch sub cycle when program counter contents are used as address to the memory the output of memory the data output of memory is at the end of that fetch instruction sub cycle the clock cycle the contents of memory are going to be loaded into IR, so we are going to use a new register called IR okay, similarly at in some other situations like in some other sub operations.

For example, in the sub operations load sub operation memory access sub operation of the load instruction where in the memory will be read out in to we need to read out memory for some data which has to be subsequently latched or registered into one of the chosen registers of the registered file okay, so you we will have register called memory data register in addition to other the registers other datapath components, okay.

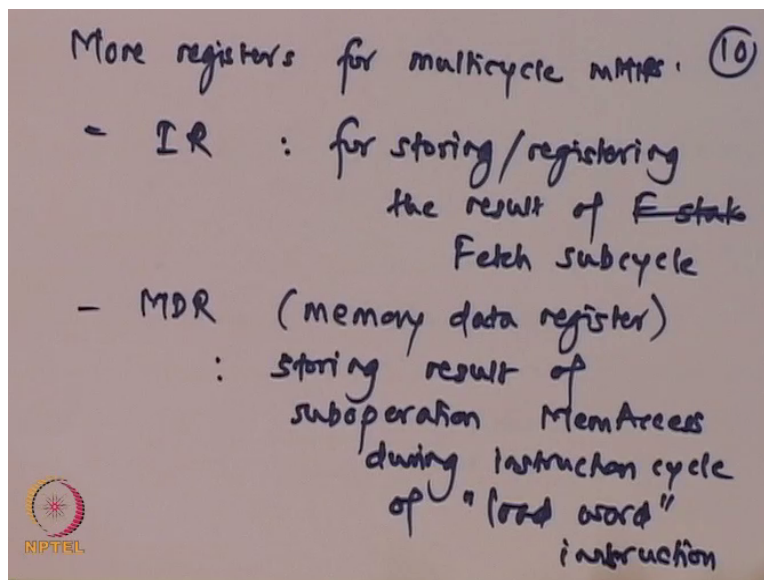
Now you see that common memory is being used but at some time the output of memory is stored in instruction register it is in some other at some other appropriate time the output of memory is going to be stored in memory data register which we know is later required to be fed in to register file, I am not drawing the diagram like you know I am not aiming at a final diagram I am just like as the idea is evolved I am going to sketch the connections okay that is a normal way of designing such architectures okay.

What else so we have seen the need for this two extra registers, similarly you might kind of perceive that you would requires this pair of 32 bit numbers which come out of register file which are the two operands for say addition instruction they are the results of the decode sub

cycle by the register is being register file is being read for operands and for that also we will make use of to store the contents being read out of the register file will use a pair of registers A and B called A and B.

So that the results of the decode sub operation are archived in this okay, similarly to archive to store the result of an ALU or execute sub operation will take the output of ALU and put it in a register called ALU result, ALU result okay so this is another one that we will use, okay.

(Refer Slide Time: 49:21)

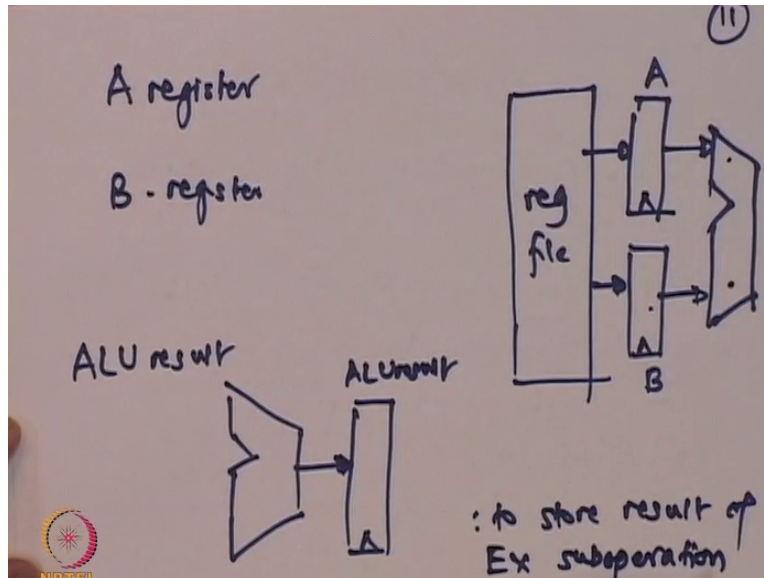


I mean just let summaries it again so more registers why and which ones, so we have actually saved on something we are saved on a block of memory we are saved on a couple of addition that is we are going to prove that or other convince you about that, but we are going to pay a bit of extra cost and that is in terms of registers so which ones IR why do we require IR for storing or registering the result of which fetch sub cycle, okay.

We talked about MDR, MDR stands for memory data register why do we need this to store result of some other sub operation which sub operation, sub operation of instruction sub cycle whatever sub operation which one memory access during instruction cycle of it is not that every register is required for every kind of operation MDR is going to require only for load word kind of instruction, of load word instruction, okay.

Similarly, similar to IR and MGR we will argue the purpose of A and B and ALU result which are the three more like registers that we have kind of decided to use.

(Refer Slide Time: 51:49)



Register, A register and B register where were they coming they were getting the data or input from the register file okay and to the main purpose of them was to feed this values to the ALU's inputs not always, not always it is going to be required but a typically that is the purpose in fact ALU is going to be used for something else at some other time.

But whenever A and B have some meaningful values on behalf of I mean intuitional values on behalf of instruction like add which is an ALU type instruction the in during the execute sub cycle sub operation this will the contents of A and B will become input to the ALU for doing the Arithmetic, okay. So that is how the connections will be so it is not only connections I am just they just we are figuring out which kind of signals.

How the signals might flow, the paths of the signals, so eventually in the final implementation there will not be a single wire for this it might be like broken by multiplexer it might be a path which might have multiplexers, routers along the way, okay. But this is this arrows or whatever I am drawing are indicating this the data can flow or it would be required to flow in some situations from register A to the first input of ALU in some other situation some situations that data would be required to flow from register B to the second input of ALU, okay.

In some other situations some other datapath would be there from somewhere else to the first input of ALU and some other situations there could be some other datapath coming from some other source to the second input of ALU okay, so all that can happen and so for that naturally we will be using multiplexers to do this routing, selection of data sources and so on.

The last register that we plan to use was is kind of kept at the output of the ALU to archive the result of to store result of EX execute sub operation okay, all these registers are clocked so in fact I could kind of have this triangle indicating that they are clocked on the rising edge of the clock provided we are using positive edge or rising edge triggered discipline, so we just made an inventory of the extra registers that would require.

Because we are now taking the multicycle approach IR, MDR. IR for instruction register, MDR memory data register, A and B registers, and ALU result and we justified I mean why we need them at what time they will be of use they will be used to store the results and we are given some rough picture and this was kind of based on the rough like finite state machine that we had designed rough picture of the final state machine that is the basis heart of the controller by the I just realized that, okay.

I did not claim it to be complete, but one of the things that I will missed out here was considered the memory access for the store instructions if it is a store instructions after the memory access after writing into the memory we are done the store instruction is finished right, so after that it will go here that is if the instruction were store instruction then after the memory access that is memory write the instruction cycle is over and we the controller would go to the fetch sub cycle or fetch sub operation for the next instruction cycle, okay.

Whereas if after memory access if it were a load instruction then memory access must have been doing the job of reading something from the memory and then you would have to like go to a state where what has been read out from the memory is now return in to register appropriately chosen register of the register file that is in the write back state and load instruction would only

then be over and instruction cycle finish and then the controller will go to the fetch sub state of the next instruction cycle, okay.

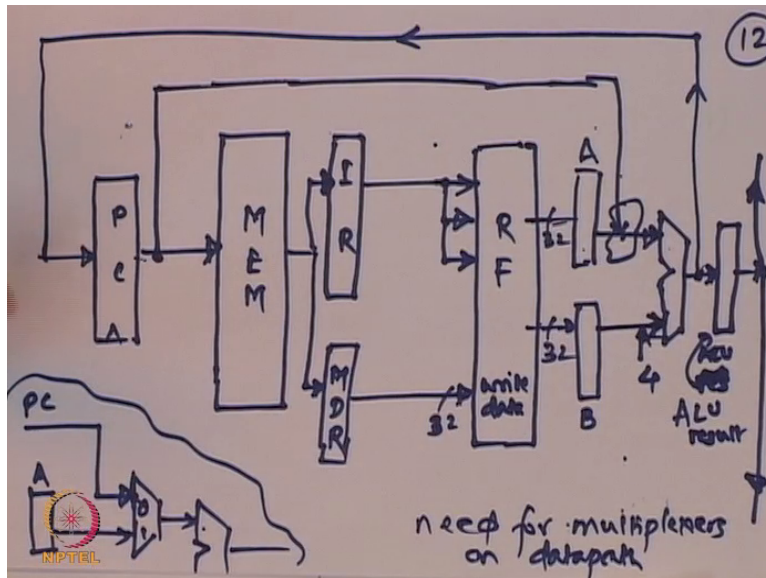
So just looking at the load instruction cycle, load instruction cycle starts in the state fetch, then it does the decoding work when it also its reads the operands from the register file, then it like in the execute state the fetch the load instruction would compute the memory address then it would go to the memory access state in which it will give itself time, give the datapath time to read out the contents of the memory.

And then the contents which have been read out from the memory are going to be registered in appropriate place in the register file that is inside WB state and then the instruction cycle is over and the controller will go back to the fetch state of the next will resume fetch state of the next instruction cycle, okay.

In case of jump sequence of states transitions is that from fetch you go to decode independent of an instruction and if it were a jump conditional - unconditional jump instruction in the decode state itself the program counter would be modified at the end of decode state the program counter would have been modified with the target address that have been extracted out of the instruction which has been decoded so the jump instructions, instruction cycle is over.

Once these two states are finished two sub cycles two clock cycles are over, for branch they will be F, D, EX for store there will be F, D, EX, MA and this and so on. So we can trace the behaviour of different instruction cycles for different kinds of instruction using this FSM I believe it is complete so anyway we will see a very detailed picture when we arrive at a stage where we are going to implement like this and writes a Verilog code or target FPGA for implementation of this.

(Refer Slide Time: 59:22)



So let us like you know try to get more details of the datapath FSM we just had a relook at the datapath this program counter, the program counter has the access address can provide access to memory okay, from memory we can fill up this contents of memory can be used to fill up this IR register you know in which sub operation or which state this IR will be updated that would be in the fetch state.

Whereas in some other state for some other instruction this MDR okay memory data register is going to be used for registering the contents of memory which state it would be the memory access state of a load instruction and for all other instructions which are not of that kind load or its variants this will be of no use if it will not be having any meaningful purpose but it has to be there, because in some situations it is required so cannot help it.

Then there is a register file and its sizes are not proportionate just do not worry about that, what is coming as input to register file RF I call it, the input to the register file or like you know as we seen in the single cycle datapath we require some bits of instructions to arrive here and okay this pair of 5 bits do you recall where used as indices of the registers which have - which are to be read out whose contents to be read out on this 32 bit wires, okay.

So this 5 bits and this 5 bits are going to specify two pair of 5 bits indices to indicate which registers to be read out on this output and on this output okay, and in some situations some of

these bits are to be interpreted as the index of the destination register where like in case of load instruction the data which has been read out from memory has to be archived in okay, so one more set of bits will indicate the destination register index.

I am not giving a detailed picture here it will be too clumsy to, too cluttered to bring it out just the overall concept and the details will be clear later on anyway, what is the purpose of this MDR where will it be used it would be used to provide data input to say right data I need not write this I think you I am going to leave it to you to guess what is where the signals are going from that is clear this signal is going from MDR but to what which port of RF register file it goes to here.

I have kind of given you the answer explicitly it will go to write data port this are 32 bit lines okay, so when will this happen this transfer happen, this will happen this will be arranged to happen inside a write back stage of load word instruction cycle alright, in some other case this write data port of register file would be used would get fed from would get data from some other source possibly, but this is one possible source of data for this.

Then we have this we are decided to use A and register B okay and then its falling short of space I will just draw very narrow ALU okay and this is one possibility of data flow into the two inputs of ALU, the ALU's output goes to the ALU out register ALU result sorry okay this is ALU result okay and we will wondering where the output of this register go we will come to, that just think of its a hint it is to go somewhere else in more than one place maybe.

So PC has to get input from somewhere right from where what is the input to PC the updated program counter value and it is always the result of like it is, it can come from couple of sources for example it can come out from ALU's output or it can come out from instruction register itself okay, so can we try and work out the picture of where this come from at least one possibility.

So typically what is getting loaded into PC program counter the $PC + 4$ and where is that $PC + 4$ available it is going to be available at the output of ALU okay and how is that $PC + 4$ generated, to generate $PC + 4$ through the ALU we will have to feed this program counter's output we will have to arrange to read it as a one possible source of ALU's first input.

And we will have to feed like number 4 here at the second input for $PC + 4$ under some situation this will become the output to the first port of ALU this will become the sorry this will become the input to the first port of ALU this will become the input to the second port of ALU, $PC + 4$ will be computed will be available here and in the same clock cycle we want to arrange to take this output and to keep it ready to be loaded at the end of clock cycle into the program counter.

This program counter is edge triggered okay so in that means in the fetch sub cycle what would be happening is that PC contents are read out in the I am talking about any typical fetch cycle - fetch sub operation okay, in the fetch sub operation the contents of PC are being provided as registered to memory element memory block and the output of memory block are fed to this instruction register at end of clock cycle for fetch this instruction register is going to be going to get updated with the contents of memory.

So instruction the register will henceforth content that particular instruction which has which was at this address, further more during the fetch clock cycle this program counter will be also fed to ALU as the first input and second input to the ALU would be fed as this number 4 constant 4 will be fed to it, so $PC + 4$ will be computed and ALU will be configured as an adder so $PC + 4$ will be available here and that is brought back here.

Note that I am not bringing this back from the ALU result register it is before it gets latched into the register I am taking it directly, so this is all happening in the single clock cycle from this register through the combinational ALU back to this register or and also like through this from this register to the memory and asynchronous read and writing into this register, so there are this two flip-flop to flip-flop kind of paths, state element to state element paths okay here and this.

So this kind of data flow is happening during the fetch sub cycle of any instruction cycle okay, so please make a note that this is going $PC + 4$ is coming from here not from here that would be available only in the next clock cycle, so but whereas PC has to be updated in the fetch clock cycle itself, okay.

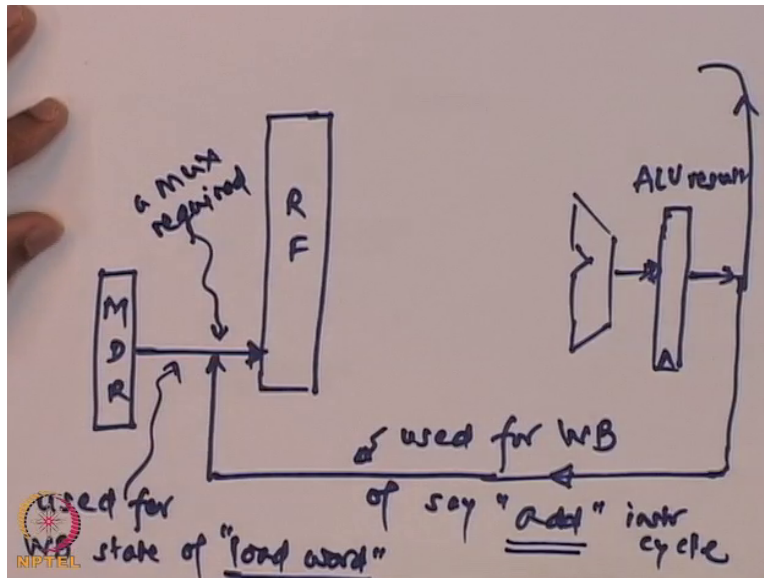
So now we see that we need some multiplexers here we have already seen the need we just in the need that the first input to ALU can typically would typically come from the A register or could come from PC in fact early in the instruction cycle the first input to ALU always comes from PC because we have to compute $PC + 4$ and to keep it ready to be latched into this PC registered into the PC at end of that clock cycle very earlier is right.

And second input to the ALU at that kind of time is required to be this constant 4, on the other hand most typically second input to the ALU comes from the B register do not worry about this clutter we will redraw it for different situations and you have this diagrams very easily available in text and so on, so what I am emphasizing here is how do we evolve this from scratch alright, so the point I am making was making is that we will need multiplexes here.

we will need something like a multiplexer to feed into the first input of ALU okay and typically the contents of A will be fed will be routed through the multiplexer to the first input of ALU or in some cases it is a PC which will be contents of PC will be routed to the first input of ALU okay, so this is the this what I have drawn here is the kind of blown up version of this picture okay it is clear just stare at it for a while, okay.

So need for multiplexers on datapath so that is now clear we need more and more of this multiplexers we need a multiplexer here because there are two possibilities either the second input will get either the data from B register or will get that constant 4 in some other situation in some other clock cycle right, so provision has to be there in appropriate clock cycle is the multiplexer will route appropriate input to its output that is the router okay, what else can we add to this diagram.

(Refer Slide Time: 01:12:14)



Let us look at some other kind of signals we are curious about what is happening, this is that ALU result and we have kind of tentatively drawn that the output of this ALU result register is going to be required somewhere else where, ALU result would be required in case of instruction like add this is reg file and this is the right port to which typical sometimes not typically sometimes MDR will be feeding to it.

But at some other in some other situations it will be the result of ALU that is to be returned into the appropriate register of this register file okay, so this data flow required for WB of which instruction write back of say add instruction cycle okay add instruction cycle okay, so whereas this data flow this signal flow would be used for used in the WB state or WB state of load, load word instruction whatever the mnemonic for that is okay load the instruction cycle of that alright.

So again a MUX required okay, so that shows they need for something like this, so we I mean you know we have been kind of trying to evolve the datapath architecture of multicycle CPU and we have we made an inventory of what datapath components we require, we realize that this is it is going to be done in terms of sub operations at different clock cycles, we are going to require more some more registers like IR, MDR, A, B.

And there will be signals flowing like in certain from certain sources to certain destinations that we have also tried to capture, most of it most of the datapath we have kind of figured out in this

figure as well and in addition with this figure, there are still a couple of details left couple of you know data flow, signal flow wires left here, you can think of it on your own as an exercise, we are anyway complete it in the next when I resume.

So fortunately this multicycle datapath is not fact it is slightly tidier than the single cycle datapath that you would see okay, so just that it is not fitting in too well here we will get a well design figure and show you all of it but what we have done so far is to show you the evolution of it how with the thought process behind designing such a multicycle datapath I hope it has been, okay.