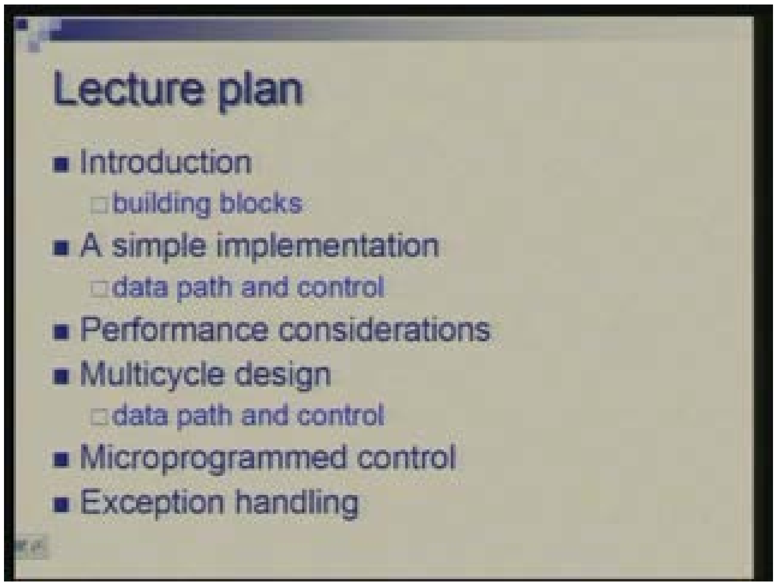
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 17
Processor Design - Introduction

In the last few lectures we have focused on design of very specific component of the processor namely ALU or Arithmetic Logic Unit. Now we are going to look at the overall processor design where this would be the key component but we need to add more pieces of hardware to make the whole processor complete. So, first of all we will talk about the building blocks; what are the other blocks including ALU which I require to build the processor and we will start with a very simple design where we will simply put these building blocks together in the simplest possible manner and see how the data would flow, what is the data path and how it is controlled. Then we will see the performance of this particular design and we will realize that we need do something more to get a better performance.

(Refer Slide Time: 01:52 min)



Lecture plan

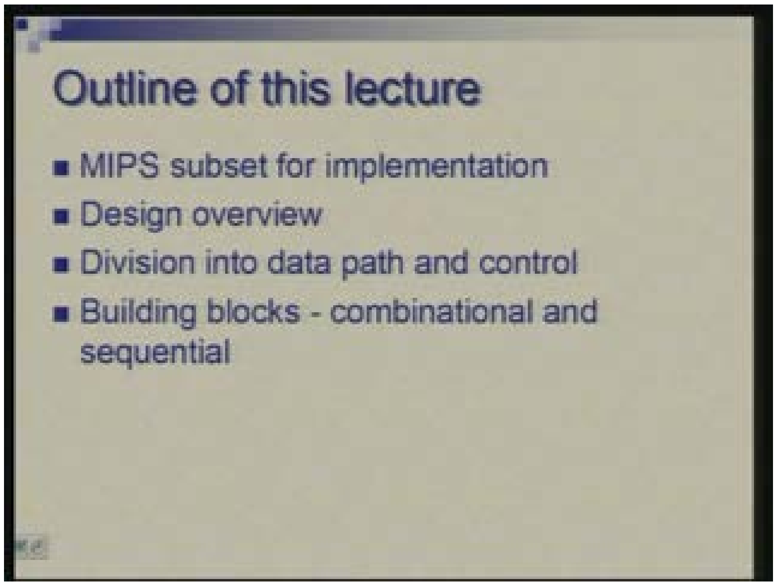
- Introduction
  - building blocks
- A simple implementation
  - data path and control
- Performance considerations
- Multicycle design
  - data path and control
- Microprogrammed control
- Exception handling

We will therefore move over to a little more involved design, more sophisticated design which is called a multicycle design. this term multi cycle will be explained as I go long and again we will see the two major parts data path and control then we will see a very specific style of designing and control called Mac program control where essentially one very small low level program is trying to control the overall processor and finally we will enhance our design to handle what is called exceptions.

While we were talking of instructions we talked of things like overflow and there are many other situation which are exceptional situations which are not normal and then the design has to take care of these situations. So first we will ignore these and then enhance the design to accommodate these.

In particular, today we will first begin with taking a set of instructions from MIPS architecture which we are going to implement so that the design we talk of is a very simple manageable design we can understand discuss and understand in the class and look at the overview of the design; what is the outline, what is the basic idea, see how it is divided into data path and the control, then go on to description of the building blocks.

(Refer Slide Time: 3:30)



Outline of this lecture

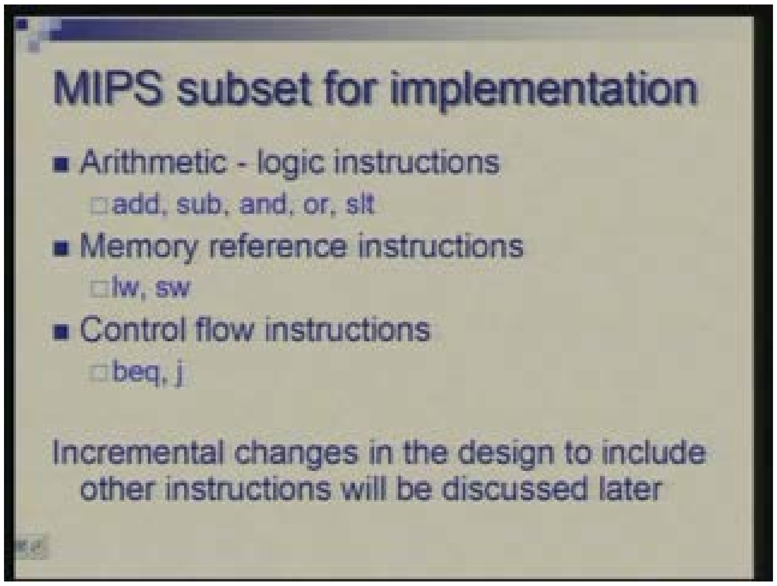
- MIPS subset for implementation
- Design overview
- Division into data path and control
- Building blocks - combinational and sequential

We will see that there are two types of building blocks: combinational and sequential. We will look at the issue of clock which times the whole design and see what are the timing constraints a particular clock frequency puts in the design. Or, given a design what kind of clock frequency you can expect to have. So finally we will see the components which are very specific to MIPS.

initially The initial discussion would be components in a generic sense and then we will talk of something which is required to specifically build MIPS which has to carry out these particular instructions.

So, starting with what instruction we want to begin with, these are the instructions we learnt right in the beginning. So we will take those few instructions and try to work out the design. These are now we have to very very specific because we need to build a circuit which will actually do those particular instructions. So let us take add, subtract, AND, OR and slt. So it is two arithmetic instructions, two logical instructions and one comparison instruction. Then we will take two memory reference instructions: load word and store word; two control flow instructions: branch if equal and jp. So one is a conditional branch which has the simple comparison; slt has little more complex comparison. As now you would realize that after having discussed comparison and ALU circuits that comp[arity] comparison for equality is simpler as compared to comparison for less than or greater than. one simple thing is that when you are comparing for equality there is no overflow information across the bits; you look at each bit separately and check if the corresponding bits of two operand let us say A and B are equal and then individual results could be combine. Whereas if you are doing comparison for less than then either you do by subtraction where a carry flows through or you do direct comparison in which case also there was some information flowing. so you We had written recursive equation describing the comparison operation.

(Refer Slide Time: 6:15)



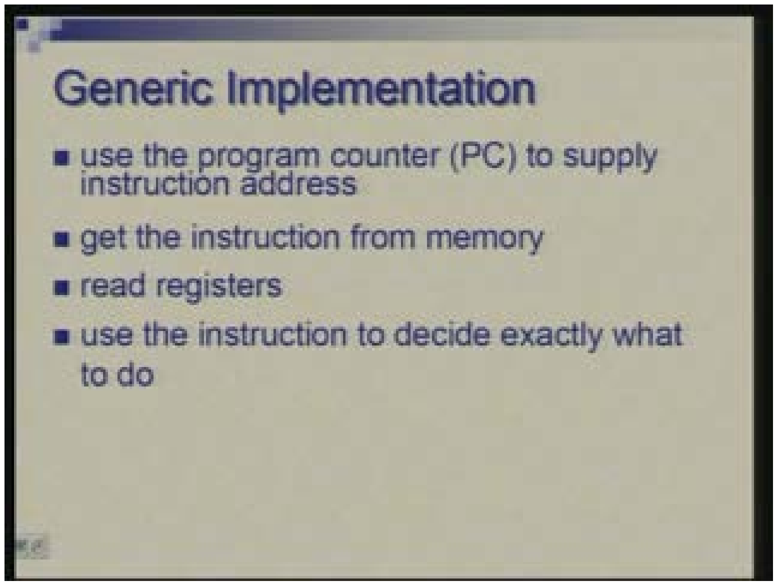
MIPS subset for implementation

- Arithmetic - logic instructions
  - add, sub, and, or, slt
- Memory reference instructions
  - lw, sw
- Control flow instructions
  - beq, j

Incremental changes in the design to include other instructions will be discussed later

So a simple comparison like equality test is combined possibly with branch as in beq. So, after looking at the design which cater for these let us say for five plus two plus two nine instructions so this nine instructions processor would have a simple design but we will notice that additional instructions can be added to this design by making small incremental changes here. In many cases the change will be indeed small, in some of course, depending upon the nature of instruction if you want to add an instruction which is too diverse from what we have the change may be a little larger but once you have the overall structure overall outline of the design then adding other things is comparatively easier. So, probably that discussion will be in tutorials.

What is the overall approach; how do we intend to implement a processor design?
We use a register or a counter which we call PC which will supply the address of the instruction to be executed. So the whole story begins here. You take contents of PC that decides where in the memory instruction is located though you get the instruction from the memory this is called technically fetching the instructions. Then, for instructions like, let us say, add, you read the registers which will give you the operands. So, from register file you read the values the instruction then tells you what is to be done whether addition is to be done, subtraction is to be done, comparison is to be done so you carry out that operation.
(Refer Slide Time: 07:51 min)

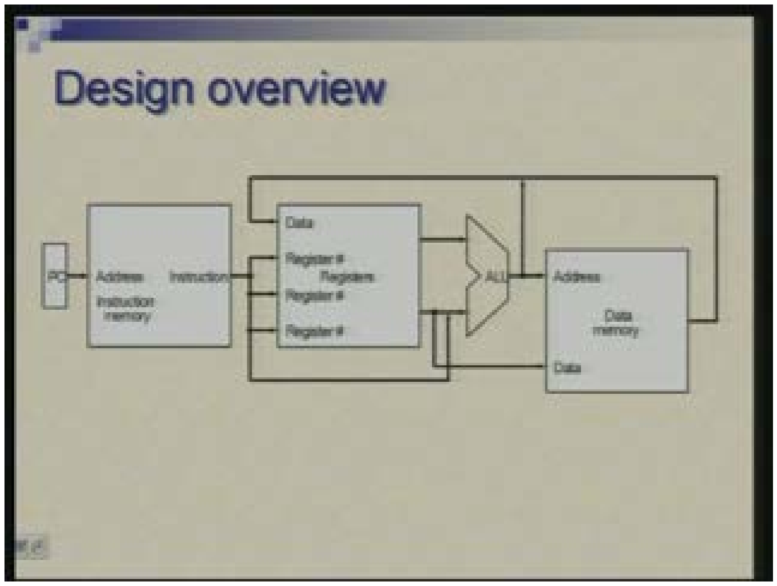


Generic Implementation

- use the program counter (PC) to supply instruction address
- get the instruction from memory
- read registers
- use the instruction to decide exactly what to do

Then basically you have these components.

(Refer Slide Time: 8:07)



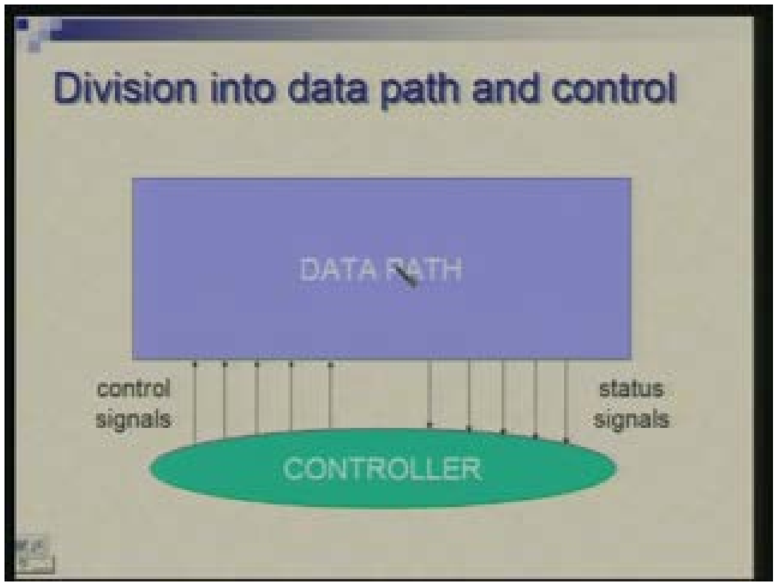
Design overview

I have shown connections roughly. So you have PC which supplies the address for a program which is stored in this, if you are calling as instruction memory, from this we get the instructions. Instruction has various fields, some fields specify registers, some fields specify operation to be performed and so on. so, from different field in the instruction we pick up the register addresses, access the values of registers, pass them on to let us say ALU for example performing the operation and the result of ALU may have to be stored back in a destination register. Or in an instruction like load or store we may have to access data memory. Data memory accessing will require again address calculation. You need to add content of register with an offset which comes from the instruction so we can possibly use the same ALU for doing these things.

So a constant may come from the instruction, register may come from register file, this will provide address and you may write in to the data memory if it is store instruction or read from the data memory if it is a load instruction. So these are the key components. We have talked extensively about ALU itself. Again while designing ALU we looked at a few specific instructions for which we made provision in the ALU. And the instructions I am talking of today are all covered in that. The other components like PC, instruction memory or data memory we are going to see now and also the register file.

So I will come back to these components. But on the whole, one thing we need to keep in mind is that the design on the whole has two major parts: one is called the data path and the other is called controller. So data path is the one. In fact what I have drawn was not the full design; I had drawn the skeleton of the data path.

(Refer Slide Time: 10:22)

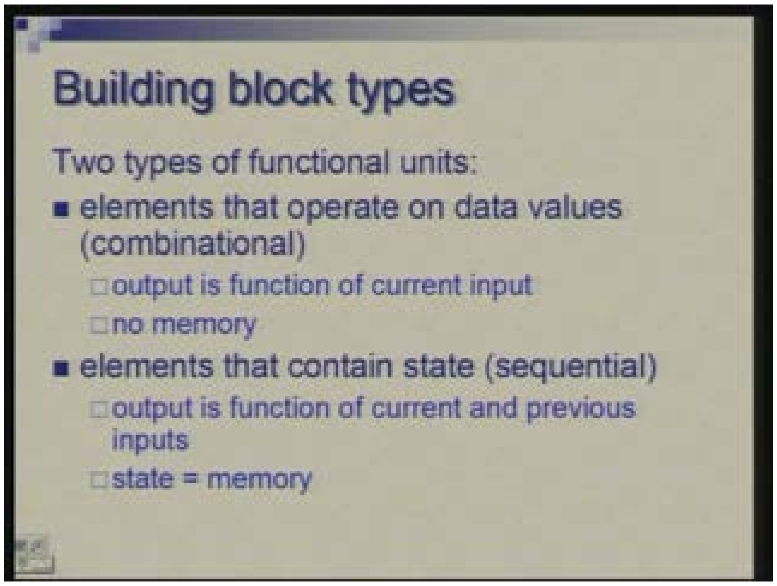

Division into data path and control

So data path is the one where you have values or operands on which you are working. So, data flows here and data is stored here so all calculations get done here but the other part which is the controller is the one which guides this which actually controls this or instructs this. So controller is the one which will tell the ALU to perform addition, subtraction or comparison and controller will instruct memory whether it has to read or write and so on. So, each of the other components, each of the components in data path works under the instructions of the controller but controller for its own decision may look at some information coming from the data path. I have shown these two kinds of signals which flows from controller to data path and data path to controller. There are control signals which go from the controller to data path and the status signal which comes from the data path to controller which help the controller in deciding the action.
Now let us look at the building blocks. There are two types: one is the kind of elements which operate on data values; we call them combinational elements and the other is those which have a state, which contain some information which we call state and this is called sequential elements. So, combinational circuits have output determined by the current input. So combinational circuit

elements will look at the current inputs and respond almost instantly to that; I am saying almost because they might be some delay.

You remember that we talked about gate delays but these have no memory whereas element which contains state which we call sequential elements they have memory, they remember what happened in the past and therefore the output is a function of the current input as well as the previous input. So, in some sense the state contains state represents a summary of what this element has seen in past; a summary of the previous inputs is actually contained in the state. So, strictly speaking, the output of a sequential element is the function of the current input and the state; state encapsulates, state captures the relevant part of the past.

(Refer Slide Time 13:06 min)



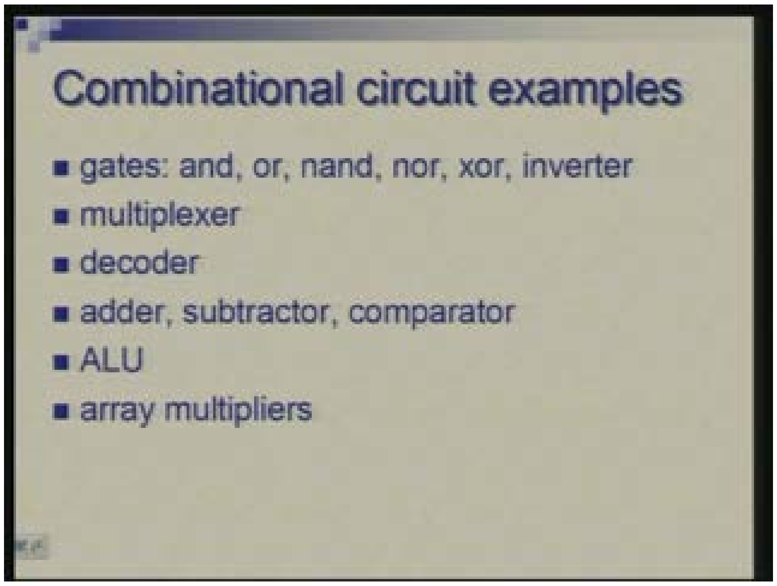
Building block types

Two types of functional units:

- elements that operate on data values (combinational)
  - output is function of current input
  - no memory
- elements that contain state (sequential)
  - output is function of current and previous inputs
  - state = memory

I will give some examples.

(Refer Slide Time: 13:16)



Combinational circuit examples

- gates: and, or, nand, nor, xor, inverter
- multiplexer
- decoder
- adder, subtractor, comparator
- ALU
- array multipliers

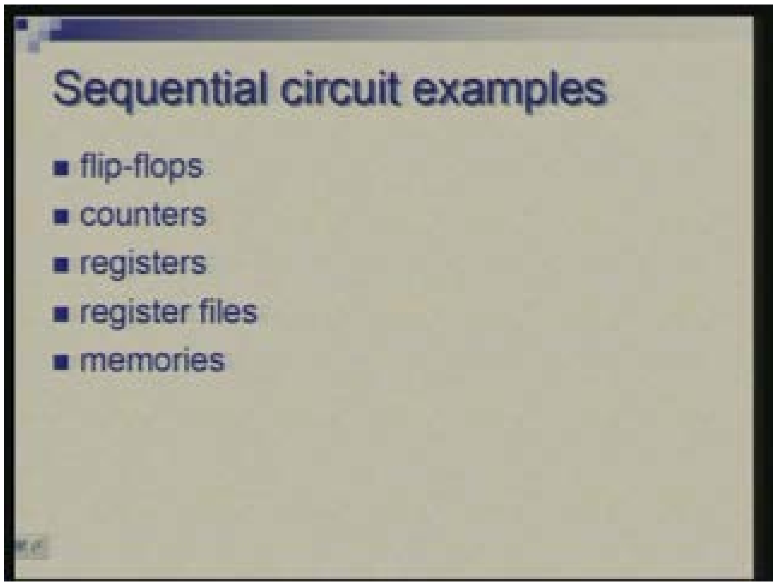
All gates are combinational circuits. If you take AND gate, depending upon what the input is it will almost instantly react to the inputs after some small delay. So, in ideal case we assume zero delay and say that the combinational circuits have output function of the current input. So various gates AND, OR, NAND, NOR, Exclusive OR, inverter and so on all these are combinational circuits.

Multiplexer makes choice between two or more inputs and selects one of these to be available at the output, this is also a combinational circuit.

Decoder: Decoder is a circuit which looks at the bit pattern and the input and identifies the bit patterns. Suppose we talk of a 4 input decoder so it will have 16 outputs and it will activate one of the outputs depending upon which one of the sixteen combinations is available at the input. Then components like adder, subtractor, comparator, ALU all these we have discussed, these are all combinational circuits.

Multiplier we discussed various designs. But the array multiplier which was simply a collection of adders and other logic put together they are combinational circuits. The sequential multiplier which goes through iterations using the same adder is not a combinational circuit and anything which is not a combinational circuit is a sequential circuit. So we will look at examples of these.

(Refer Slide Time 14:50 min)



Sequential circuit examples

- flip-flops
- counters
- registers
- register files
- memories

The most primitive ones are flip-flops which have one bit of memory; they can store a 0 or 1 depending upon what value they contain at any given instant of time depending upon the signals which occurred in the past.

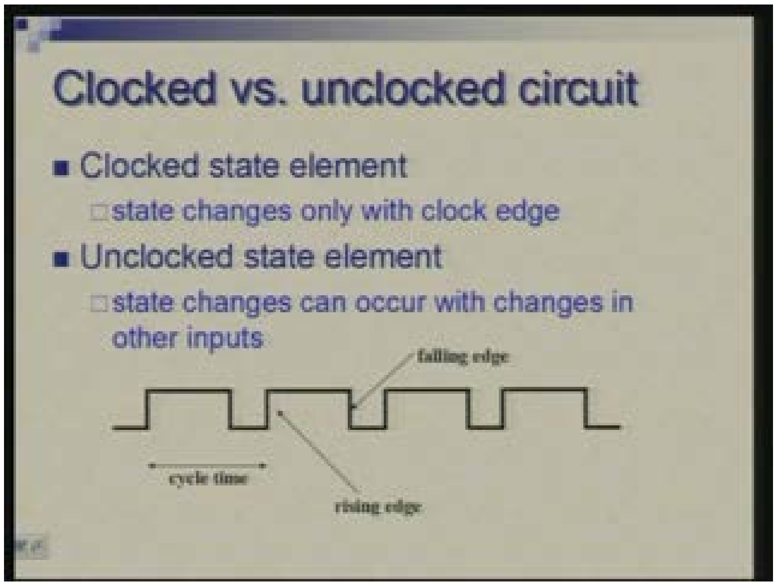
Counters are sequential circuits.

Registers: So registers are extension of essentially flip-flops in one dimension.

Register files and memories are extension of flip-flops in two dimensions.

There is some distinction between some files and memories. I will come to that later when we talk of component specific to MIPS instruction set.

(Refer Slide Time: 15:37)



Clocked vs. unclocked circuit

- Clocked state element
  - state changes only with clock edge
- Unclocked state element
  - state changes can occur with changes in other inputs

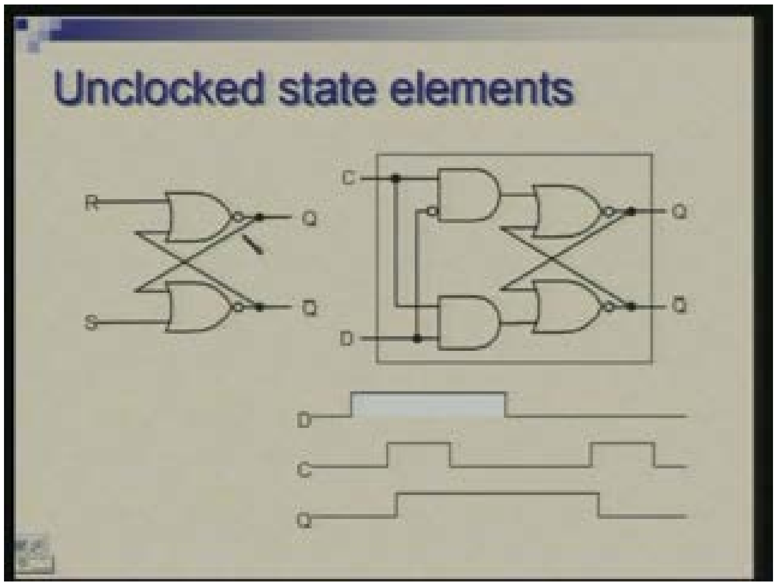
falling edge
cycle time
rising edge

Among the sequential circuits we can have circuits which work with clock and those which do not work with clock so clocked state elements and unclocked state elements. In the clocked case the changes in the state occur with clock whereas in unclocked elements they do not distinguish bet[ween], well, the clocked elements have one of the inputs which is playing a special role that is called clock. Whereas in unclocked elements there is no input which is designated as clock and any change in any input can actually cause change in state. So a clock, as we know, is a periodic signal and the period is called clock cycle time or clock period. Typically one of the edges is assumed to be active edge either the rising edge or the falling edge is considered as an active edge and that is the edge which causes the state transition.

Suppose in a particular design we are going with a convention of keeping the rising edge as active edge then all state changes in the clocked elements will take place with that rising edge.

So let us look at little more details and look at examples of clocked and unclocked circuits. So a simplest unclocked element is a latch. You take two NOR gates or two NAND gates and cross couple them. Suppose you would have studied this in digital electronics we do not need to elaborate on this. This is (Refer Slide Time: 17:37) an unclocked RS latch; R stands for reset, S stands for set. When you activate OR this becomes 0 and this becomes 1 so it is in reset state. When you activate S this becomes 1 and this becomes 0 so there is a feedback loop here through these cross coupled signals and actually it is here the information gets stored. There are two inversions in the path that provides stable storage, so, if this is 0 this inversion brings a 1 here and that further gets inverted to get a 0 back here.

(Refer Slide Time: 18:21)



Unclocked state elements

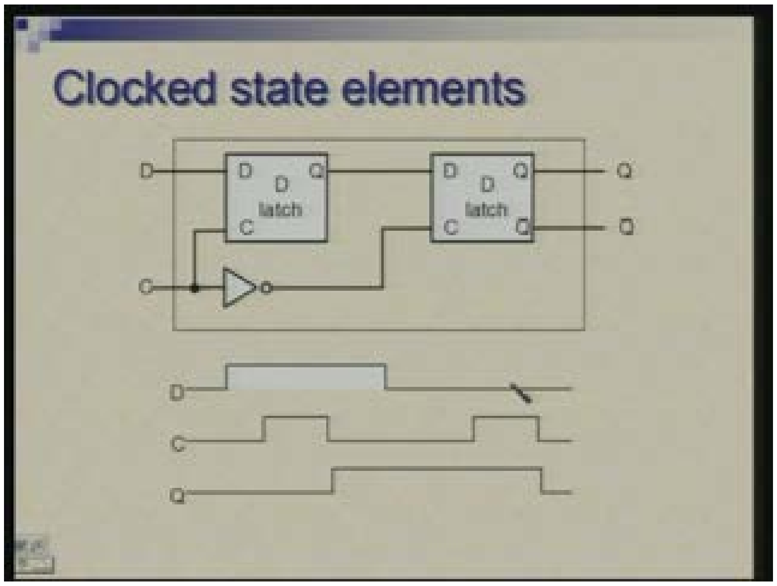
So this is one stable state; similarly, this 1 and that 0 is another stable state. So a circuit like this can stay in this state as long as you want provided there is no change in the input and the changes could occur because of the changes in R or changes in S. An extension of this is what is called a D latch where you put gates at the inputs; this is the D input or the data input (Refer Slide Time: 18:53) and this is the clock input. So although we are calling one signal as a clock but it is not a clock in the sense of a clocked element because the changes can occur because there are changes in D also in this case.

So normally suppose D signal is changing like this and the clock is like this then what this circuit does is that while the clock is in one state the output follows the D input. When clock goes to 0 then whatever was the state of this continues to be there and the next change would possibly occur only when the clock becomes 1 again. So at this point the clock becomes 1 but now the D input is 0 so it becomes 0. But you can analyze this and convince yourself that while the clock is high if there is a change in D here then the output will change. So when clock is high you have 1 here and depending upon this D this let say becomes 0 you will get a 0 here and this becomes a 1 (Refer Slide Time: 20:21) so D becomes 0 or there is an inversion here. So you will get a 1 here which means this will get, the flip-flop will get reset.

While C is 1 again if D changes it can change state. So, during this period Q could change as many times as D changes but while Q is 0 it holds the last value; you can actually spend some time on this and analyze it further.

You can get a clocked D flip-flop by putting two of these together. We take two D latches of the previous diagram and put them in this form. One is connected to C and other is connected to compliment of C. So here, now this can change its state while C is 1 and D varies but those changes will be isolated here because during that interval C will be 0 here and therefore this will keep on holding the last value. So eventually you can see changes here only when C goes from 0 to 1 I am sorry C goes from 1 to 0 because this is connected to C bar. So it is a falling edge triggered D flip-flop. In fact what I said as unclocked elements are sometimes called level triggered circuits because this is active at one level of clock while C is 1 it is active whereas this one is active at least as seen from outside at an edge so it is called edge triggered circuits.

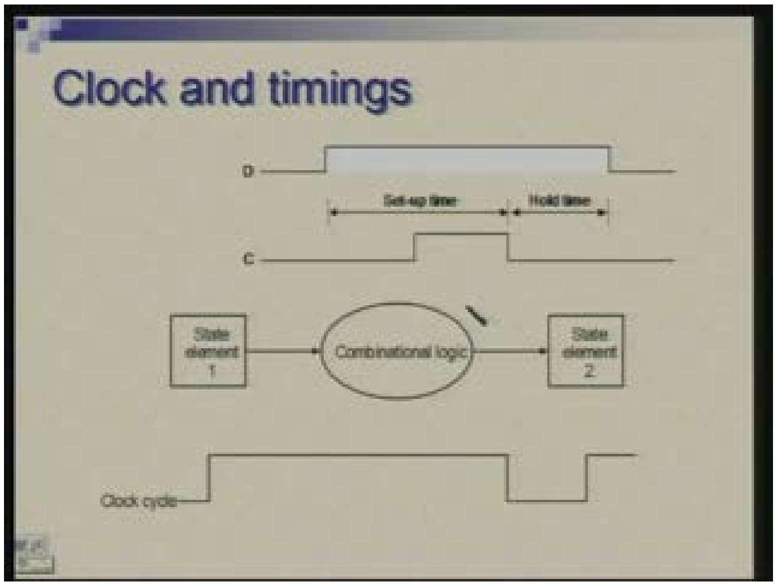
(Refer Slide Time: 22:26)


Clocked state elements

In our design we are going to use edge triggered circuits. Edge triggered circuit, as you see, require more hardware but they give us a nice property that suppose a change occurs at some clock edge here you see that change in Q and suppose through some path that change results in further change in D. But now since the clock edge has just gone that change will not affect it further. this is an important point to be noticed that imagine a D flip-flop which is edge triggered and there is path through some combinational circuits from output of this back to D that means a change here (Refer Slide Time: 23:18) can reflect a change here but now a change at Q will be observed when clock has gone through an active edge so let us say a falling edge in this case.

So, after a clock has gone from 1 to 0 a change appears here although that change may cause a change in D but now since the clock edge had just gone that change will not trigger another change here so that brings this level of convenience here. Whereas in this case (Refer Slide Time: 23:48) if Q causes a change in D here while the clock is still active because clock is active now in an interval then that could change fur[ther] that could cause further changes and you know there could be a period of instability.

(Refer Slide Time: 24:10)



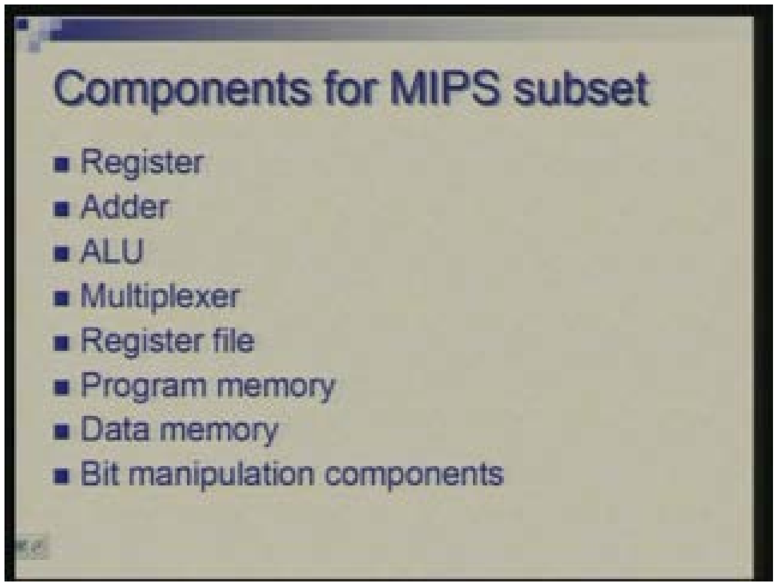
Now how do the clock period relates to the timing of the other portion or circuit?
Suppose you have one state element and there is a path through combinational circuit to another state element; this state element changes its state at one clock edge, let us say, this clock edge (Refer Slide Time: 24:38) and then those changes will result in some changes at the input of this state element; this would notice those changes at the next clock edge because the clock edge for this….. this clock edge would have gone and possibly caused some change but the change here in the element two resulting from the change in the element one here will actually occur here so this interval from this point to this point in time is allowed for signals to propagate through the combinational circuit. So now depending upon what delays these circuits present we can have a fast clock or a slow clock or given a clock we get a constraint on how much delay we can tolerate in these components or in this logic.

There are some timing constraints associated with the sequential elements themselves so let us say this is the active edge of the clock, this is C of a D flip-flop and this is the D input. So now for this value of D to be registered in the flip-flop at this instant we would we want the D flip-flop would want it to be stable for sometime before this instant and sometime after that instant and these requirements are called set up time and hold time requirement.

So, if the change in D has occurred just before the C too close to C the D flip-flop may find it too close to react and it may not be able to see the recent value. So we have to ensure that the change becomes stable, at least this much interval (Refer Slide Time: 26:31) before this instant and continues to be stable for that much period after this instant; so this is called the set up time and that is called the hold time. If D changes within this then the output could become unpredictable; you will not know whether it takes the old value or it takes the new value or if you have a register containing many flip-flops some flip-flops may react fast some may not and therefore you may have a mixed up value so this is an important consideration when we discuss the performance and timings.

Now let us come back to those nine instructions which we want to implement; we want to create a design for implementing those nine instructions and let us carefully see what components we are going to require to build the processor. We will require registers, adder, ALU, multiplexer, register file, program memory, data memory and some additional components for bit manipulation. So let us look at these one by one.

(Refer Slide Time: 27:46 min)

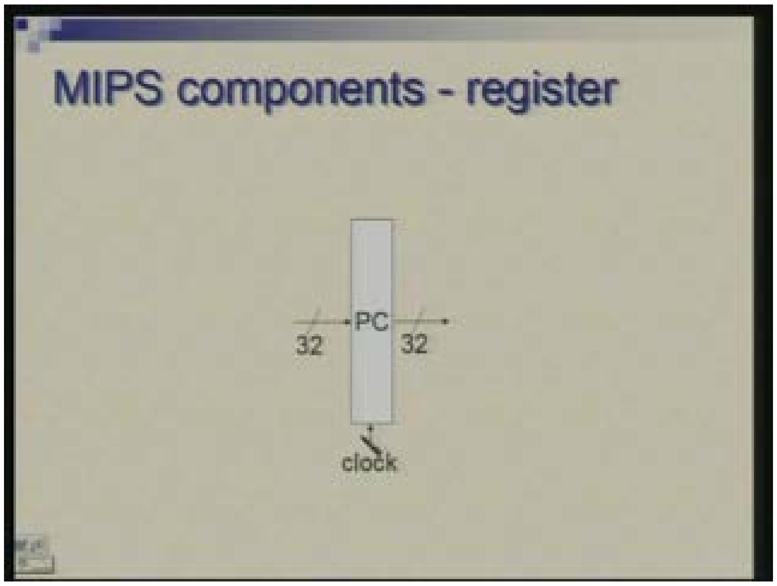


Components for MIPS subset

- Register
- Adder
- ALU
- Multiplexer
- Register file
- Program memory
- Data memory
- Bit manipulation components

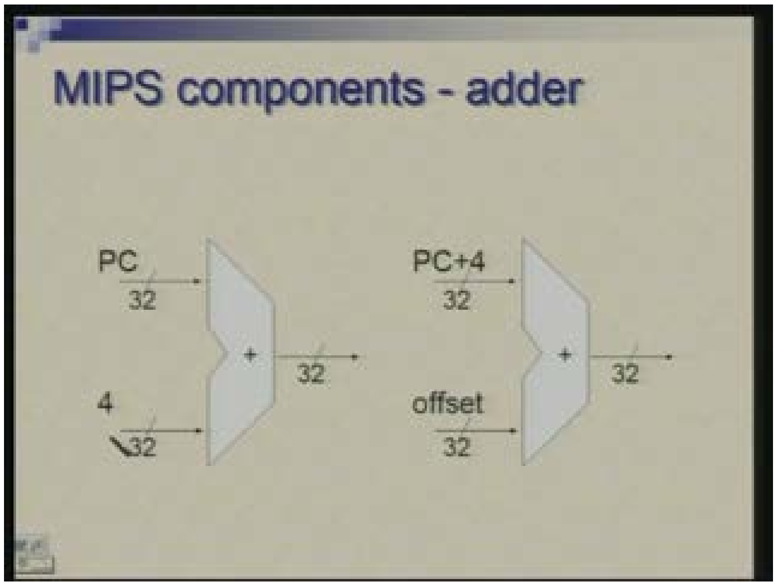
We need a register to have the PC value program counter so this register implements the PC and all this all the sequential elements we will have as clocked elements. So clock will be 1 input, it will have 32 inputs and 32 outputs; 32-bit input, 32-bit output so output will address the program memory and the new value which is going to cause change in the PC will be its input. So now at the active edge of the clock PC will change its value (Refer Slide Time: 28:32). How this value comes and when this clock edge occurs we will see when we put all the components together.

(Refer Slide Time: 28:28 min)


MIPS components - register

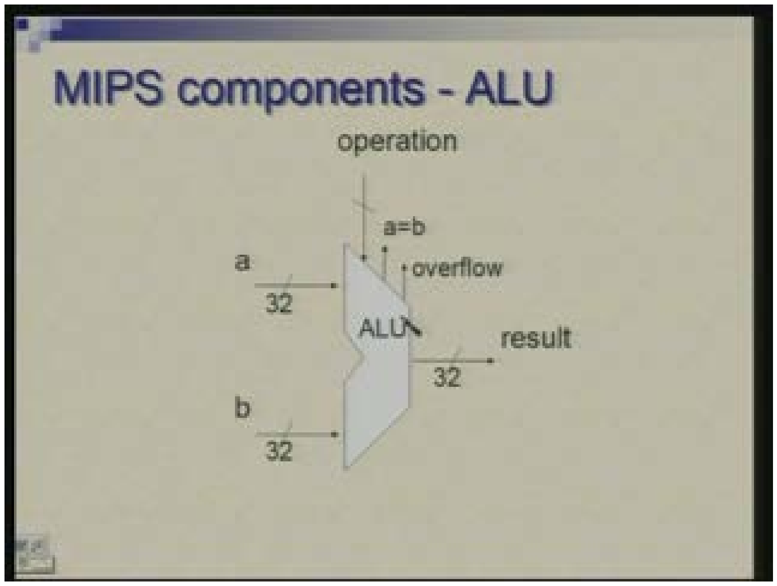
We will require adders and ALUs; ALU design we have seen, ALU will perform main arithmetic and logical operations but we need addition operation in other situations also. So, for example, to prepare PC for the next instructions you need to add 4 to the PC contents so this adder (Refer Slide Time: 29:10) will perform an addition of PC and a constant 4. So this 4 is expressed in 32 bits so there will be lots of zeros. In fact one could simplify the design of this adder noting that it has to add only a constant. But let us no worry about that right now we will see it later.

(Refer Slide Time: 29:37)


MIPS components - adder

We will just put an adder where we need to add PC and 4. Similarly we are likely to have need for adding an offset to PC plus 4 for implementing branch instructions. For branch if the condition is true you are carrying out a relative branch. That means the offset which is specified by the instructions is added to the PC but we would have added 4 to PC by this time.

(Refer Slide Time: 30:14)

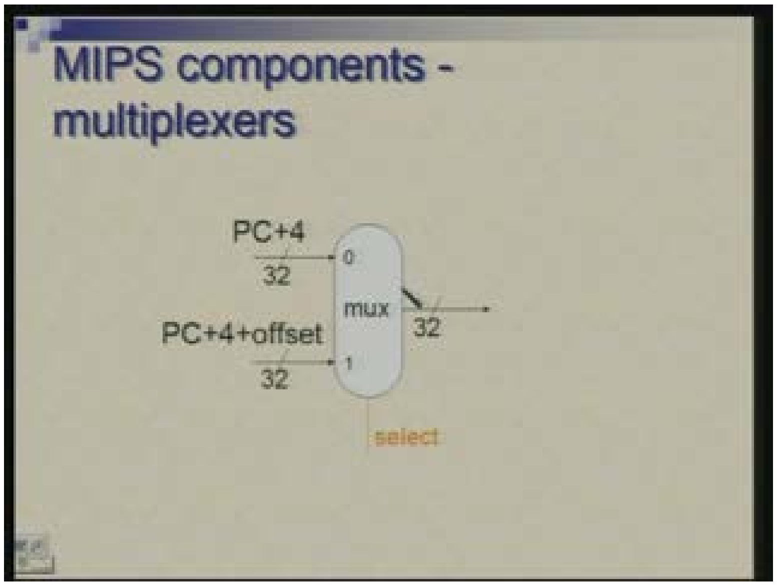


MIPS components - ALU

ALU we have already seen. It will take two 32-bit operand, produce the result and it will need to be told which operation is to be performed because we have designed it as a multi-functional unit depending upon which instruction is being executed, this may do different operations. This is also doing test for equality: a equal to b test is done. The result of slt will come in these 32 bits only but the result of beq will be a single bit which is to be used to decide whether to branch or not and we will not require this immediately (Refer Slide Time: 30:57) but when we build the provision for exception handling will require ALU also you tell us if there is an overflow or not.

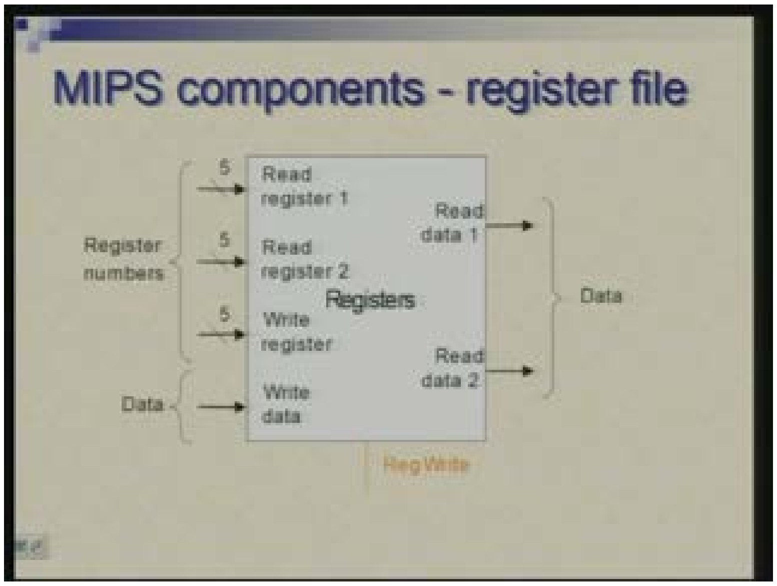
We would need multiplexers perhaps at different points but I am showing just one example here. Multiplexers, for example, may make a choice between PC plus 4 and PC plus 4 plus offset. So this is how for example branch can be implemented by making a choice of one of the two options.

(Refer Slide Time 31:37 min)



MIPS components - multiplexers

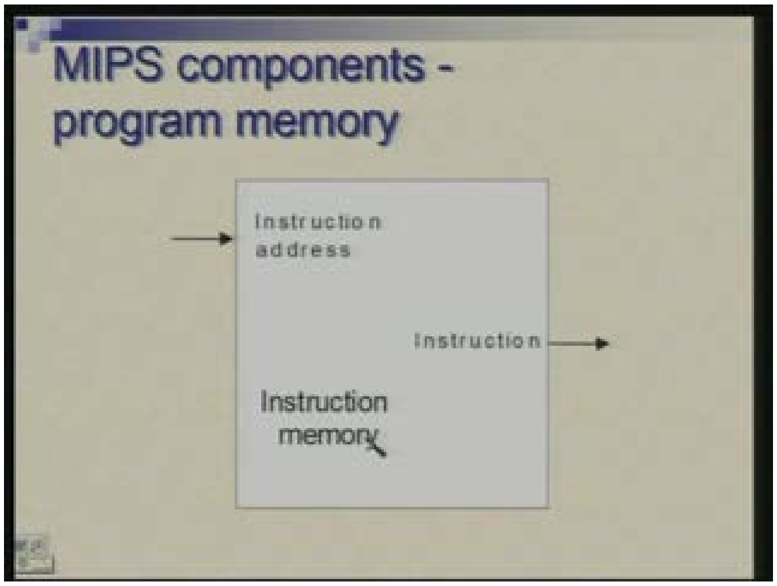
(Refer Slide Time: 31:42)



MIPS components - register file

This is a crucial component; register file. Now, to take care of instructions like add, subtract and so on we need two operands to come from register file and also the result has to go back to the register file. So this register file is nothing but an array of flip-flop; either you can think of it as two dimensional array of flip-flops, a 32 by 32 array of flip-flops or a one dimensional array of registers; each register being 32 bits. So, in any case it has a total of 32 into 32 bits of storage and it has provision for reading two 32-bit values and writing one 32-bit value. So these are the two outputs (Refer Slide Time: 32:32) which have been labeled as read data 1 and read data 2 and 1 input which is labeled as write data. Then the address of the register as which register you want to read or write is specified by these three inputs. So, for the data which is being read out here the address is specified here. Each of the address input is a 5-bit input because you have 32 registers. So it has two addresses for read and one address for write and it is a sequential element so a signal will come here to a clock ….it.

(Refer Slide Time: 33:22)

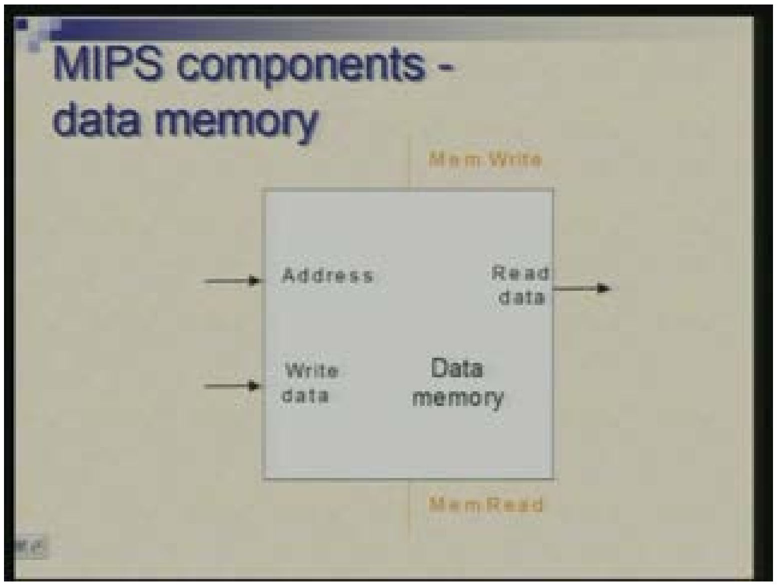


MIPS components - program memory

This is program memory. One could work with a design which has a single memory for data and instruction or one could have separate one. So in the initial design we will assume two separate memories just keep things simple. The instruction memory is something from where we only read instructions we do not modify this. so they it has One might debate whether it is a combinational element or a sequential element. Memories are sequential elements but in this present context we will not be changing the state of it so it will act like a combinational circuit because we will assume that the contents of these are fixed we are not changing it. I am not show any clock input here.

Basically address will be supplied to it, address is an input, the address which comes from program counter and outcome is the instruction. So it responds instantly to the input coming from the program counter much in the same way as a combinational circuit would do.

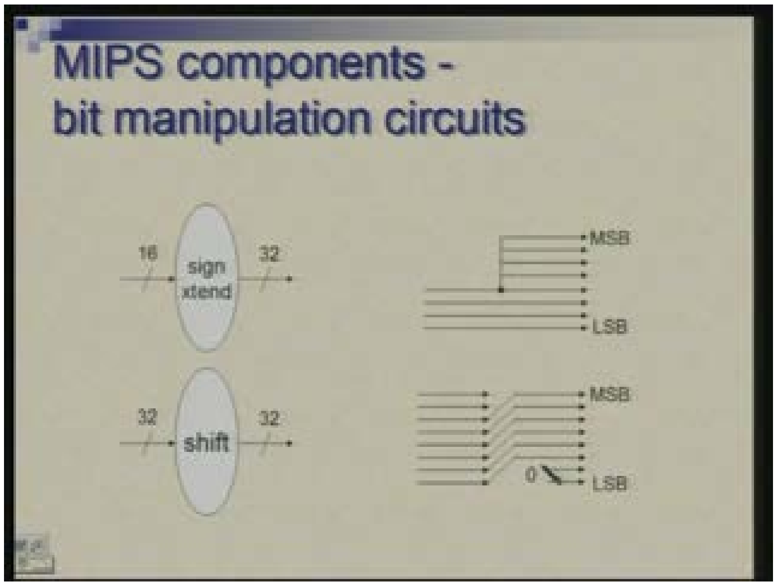
(Refer Slide Time 34:38 min)



MIPS components - data memory

The memory which contains data will have data input and data output, read data and write data and of course address. So here we assume that unlike register file where we would be doing read and write probably simultaneously within the same instruction as far as the data memory is concerned we notice that there is a load instruction which we read from memory and there is a store instruction which writes into the memory so we do not do both of these together and therefore we will have a single address input, this will take care of reading as well as writing operation and there are of course external input which specify what is to be done; whether read is to be done or write is to be done.

(Refer Slide Time: 35:34)



MIPS components - bit manipulation circuits

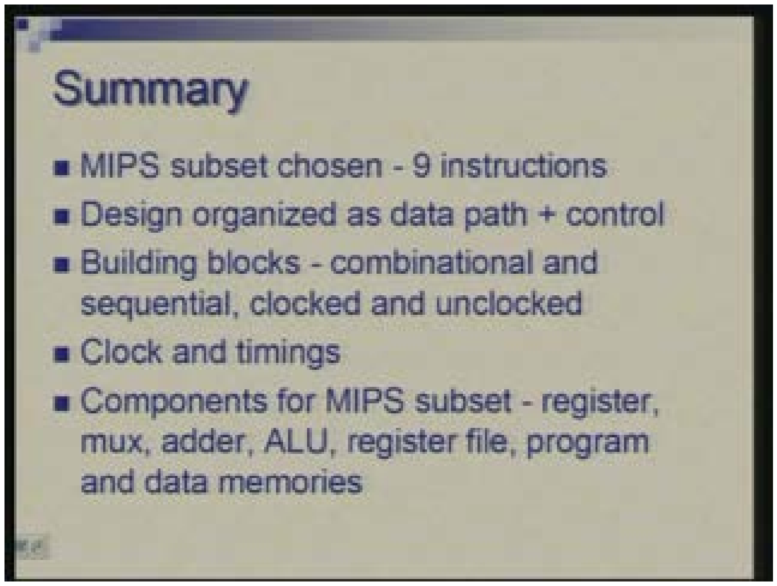
Then we have some miscellaneous kind of circuit elements which would be required to put this together and complete the picture. For example, we require sign instruction or many instruction so, imagine, the instructions which perform load operation; so the address calculation requires a 16-bit offset to be added to the contents of a register. So now the 16-bit number could be positive or negative so before we add it to a 32-bit number we need to do sign extension so that its polarity is preserved. So we need a circuit element which I am denoting by this 16 input and 32 outputs. It does not require any active components; all it needs is a particular way of wiring the inputs and outputs.

So, as an illustration, for 4 bits suppose you are extending from 4 bits to 8 bits all you need to do is that the MSB of the 4 bits needs to be repeated so many times. So, these 4 bits are input and 8 bits are output (Refer Slide Time: 36:52) and you would notice that this particular bit which was MSB earlier gets now replicated and is available as the upper 4 bits apart from that itself.

Shifting (Refer Slide Time: 37:07). So, in particular, shifting by 2 or multiplication by 4 is required when you take offsets which are for branch addresses. So, in beq instructions if you recall that 16-bit offset which you have is actually a word offset, so to get the equivalent byte offset you need to multiply by 4. So, multiplying by 4 is nothing but shifting by 2 and this also does not require any active element any gate; all you need to do is wire it in a particular manner. Suppose you have 8 bits and you want to shift it by 2 the output will have 6 of the bits connected to the 6 of the bits here (Refer Slide Time: 37:52) and these two bits will be hardwired to 0.

So now we are ready, we have all these components to be put together and in the next lecture we will take up this task of putting these together in a manner that the instructions can be correctly executed.

(Refer Slide Time: 38:16)



Summary

- MIPS subset chosen - 9 instructions
- Design organized as data path + control
- Building blocks - combinational and sequential, clocked and unclocked
- Clock and timings
- Components for MIPS subset - register, mux, adder, ALU, register file, program and data memories

So, to summarize we began by looking at the simple subset of instructions for which we want to design the processor. We took nine instructions and we looked at the concept of designing the system in terms of data path and control, the whole thing is divided into data path and control. We looked at the building blocks how they are distinguished, how combinational circuits are distinguished from sequential circuits, how sequential circuits are classified as clocked or unclocked and we looked at generic examples of a combinational sequential circuit.

We looked at the clock and timing issues, what are the time constraints posed by clock period or what are the constraints posed by the circuit delay on the clock period and then we looked at specific circuits which are required to build MIPS processor namely register, multiplexer, adder, ALU, register file, program memory and data memory.

I will stop with this.