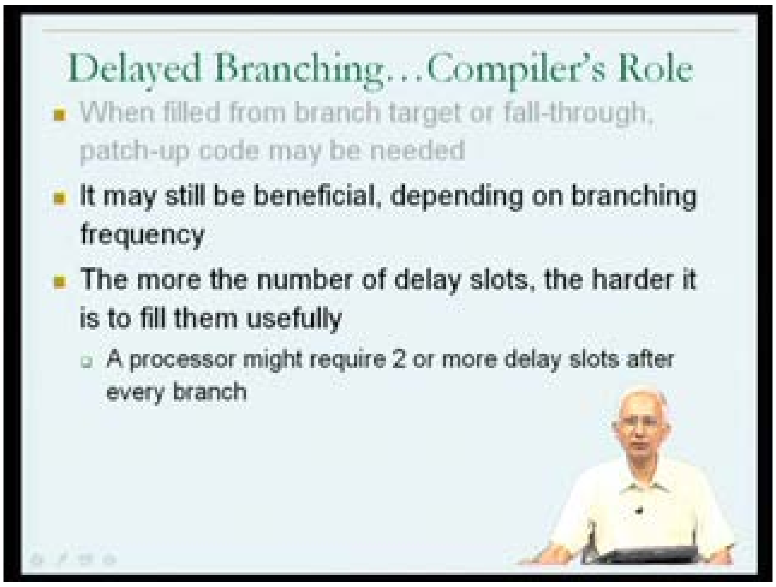
High Performance Computing
Prof. Matthew Jacob
Department of Computer Sc ience and Automation
Indian Institute of Science, Bangalore

Module No. # 06
Lecture No. # 25

Welcome to lecture twenty five of our course on high performance computing. We are currently looking at the different kinds of hazards that a programmer should be aware about in connection with pipeline processors and in talking about the control hazards - hazards which arise due to a control transfer instructions.

(Refer Slide Time: 00:39)



Delayed Branching…Compiler's Role

- When filled from branch target or fall-through, patch-up code may be needed
- It may still be beneficial, depending on branching frequency
- The more the number of delay slots, the harder it is to fill them usefully
  - A processor might require 2 or more delay slots after every branch

We were looking at the role that the compiler may have to play and this compiler's role arises through the, I mean, the existence of what are called delayed branches in instruction set architectures. Delayed branch is a branch in which the effect that controls transfer of the branch happens only after; in some of the following instructions in the program.

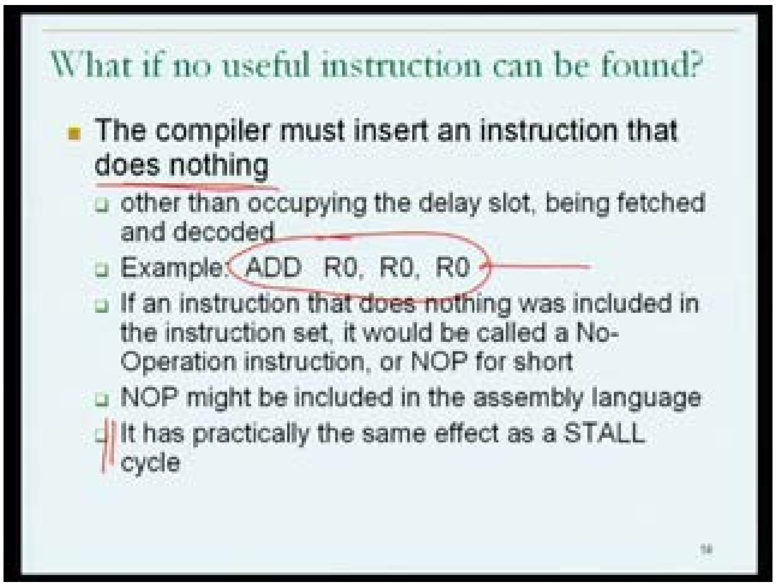
So, the compiler may have to actually identify instructions which could be used to fill the resultant branch delay slots in the example which we used but, there is one branch delay slot. notice that there is branch if equal to 0 instruction branch delay slot following it and the fall-through path, this particular example we are seeing how the first instruction from the branch target could be moved into the branch delay slot.

In this particular example, the add I instruction is at the target. Therefore, if I move the add I instruction into the branch delay slot, the target will now be the target address; will now be that of the load word instruction and in order to correct the code in terms of what happens if the branch is not taken, there has to be patch up code which will suitably undo the work that was done by the branch delay slot instruction which had been moved from the target address. So, in general compilers may have to generate code along lines like this in order to find good instructions to move into the branch delay slots.

Now, the possibility that I had outlined after that was that they could be pipelines in which there is more than one branch delay slot you may be told that the two instructions following a conditional branch instruction will be executed whether or not the branch is taken and therefore, it may end up being a situation that in practice the compiler may not always be able to find useful instructions to put into the branch delay slot.

We do have to bear in mind that whenever the instruction compiler moves some instruction in to the branch delay slot, there may be the need to put patch-up code corrective code and if one takes the corrective code into account the penalties of using this kind of idea may get more and more. Therefore, this question too does have to be addressed.

(Refer Slide Time: 02:40)


What if no useful instruction can be found?

- The compiler must insert an instruction that does nothing
  - other than occupying the delay slot, being fetched and decoded
  - Example: ADD R0, R0, R0
  - If an instruction that does nothing was included in the instruction set, it would be called a No-Operation instruction, or NOP for short
  - NOP might be included in the assembly language
  - It has practically the same effect as a STALL cycle

What if the compiler cannot find useful instructions to put into the branch delay slot? The compiler may consider an instruction from the target address, could then consider an instruction from the fall-through address, could then consider instructions before the branch instruction and may end up not being able to find useful instruction. They can be moved without guarantee of low overhead in terms of the resultant patch-up code.

So, the question is, what could the compiler do in such a situation if no useful instruction can be found? Now, if this is the case, the compiler does not have the luxury of stalling the pipeline for one cycle; that is the hardware activity. But, the compiler must do the next best thing and that might be to insert an instruction that does nothing if the compiler inserts an instruction; that does nothing into the branch delay slot and the net effect is going to be something like that over stall cycle. But, it will have no incorrect action as far as the program is concerned ok.

So, the instruction will do nothing other than occupying the branch delay slot Of course,, being an instruction the instruction will have to be fetched and decoded but, other than this the instruction will do nothing. Then, let me give you a simple example of such an instruction consider the MIPS 1 instruction add R 0 R 0 R 0 - this instruction has as its two source operands R 0; it adds R 0 to R 0 and puts the result in R 0.

You will remember from our first lecture where we talked about the MIPS 1 instruction set that R 0 is the special register it has a property that it always contains the value 0 and cannot be modified cannot be modified to contain any other value therefore, in effect the add R 0 R 0 R 0 instruction reads 0s from the register file and does not modify the register file in any way. And, in effect has no impact on the register file and in fact other than being an instruction has no impact on the rest of the processor state at all.

Of course, it is an instruction it therefore, it does cause one more the program to be one instruction larger that in would have been otherwise but, has no other impact on the program and could therefore, be viewed as an instruction that does nothing.

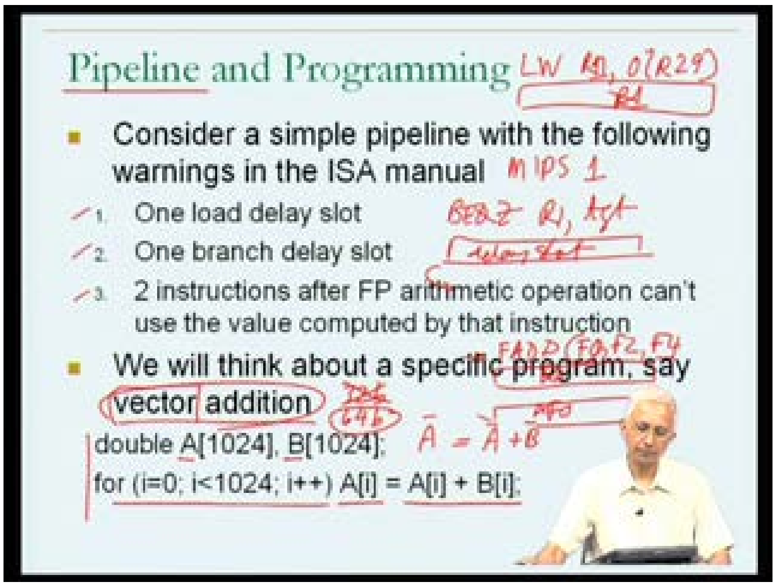
Now, in our discussion of the MIPS 1 instruction set architecture they we did not find an instruction included explicitly for the purpose of doing nothing and that is why we had to invent an instruction which did practically nothing and by the way you could think of many other examples in the MIPS 1 instruction set of instructions which have similar characteristics to the add R 0 R 0 R 0 instruction.

But the thing to note is that if this requirement of the compiler was known to the person designing the instruction set architecture then, they could actually have included an instruction in the instruction set which did nothing and if they had to make up a name for that instruction they may call it a no operation instruction or it might be known as no-op or NOP for short and there are some instruction sets which provide a no op instruction for use in the cases such as the one that we encounter here. In other instruction sets where it is very easy to generate an instruction they does practically nothing it was not felt necessary to have explicit hardware no op instruction.

However, since we suspect that the compilers may frequently be using no op instructions of this kind and I will refer to this as practically being a no op, it may be the case that assemblers like the MIPS 1 assembler may provide a notation by which rather than typing add 0 R 0 R 0 one could just type no op and that no op might be included in the assembly language, if not in the machine language because, they could well be other situations, in addition to the one that we are in right now which desirous to have an instruction that does nothing. Now, with this brief comment about the compilers role in branch prediction, I should also point out that the no op does practically nothing and in that sense it really has the same effect as a stall cycle and that the stall cycle cause that to be one cycle in which no instruction is completed.

The no op in some sense is not a useful instruction of the program and therefore, the cycle in which it completes should not be counted as a cycle in which a useful instruction of the program completes and that sense one could view the no op as having the same effect as a stall cycle. But, it is a software generated event whereas, the stall cycle would have been something generated by an inter lock a piece of hardware which may not be available in the processors that we are talking about.

(Refer Slide Time: 07:17)



Pipeline and Programming

LW R1, 0(R29)

- Consider a simple pipeline with the following warnings in the ISA manual MIPS 1
  1. One load delay slot
  2. One branch delay slot
  3. 2 instructions after FP arithmetic operation can't use the value computed by that instruction
- We will think about a specific program, say vector addition

```
double A[1024], B[1024];
for (i=0; i<1024; i++) A[i] = A[i] + B[i];
```

Ok now, with this general discussion about pipelines and the discussion about hazards which I think that the programmer or compiler should be aware about, the question then arises of, how can a, how can this knowledge change the way the programs are written? How can pipelines - our knowledge of pipelines affect our perspective on programming? So, I thought at least one example of knowledge of pipelines helping in changing the way on programs may be beneficial. First of all, I need to tell you the frame work in which we will be talking about the processor and the instruction set and now that we have instruction set that we are comfortable with. In other words, the MIPS 1 instruction set I will use a MIPS 1 instruction set as the instruction set for the example.

We have additional complications in that; now that we know so much about pipelining, I also have to specify properties of the pipeline because they are going to be relevant to us as programmers in analyzing the code that we are going to look at and therefore, I am going to additionally tell you that be this is a MIPS 1 processor; the one for which we are going to analyze the code fragments that follow. It is a MIPS 1 processor in which there is one load delay slot and there is one branch delay slot and you will remember that what it means to be a load delay slot is that whenever there is a load instruction, the instruction following the load cannot safely use the register which is being loaded. Therefore, I should not use the register which is being loaded as the source register and the one branch delay slot from which, by which we will understand that if there is a conditional transfer instruction or a branch instruction then the conditional transfer occurs not after the branch instruction but, after the instruction following the branch instruction. In other words, the instruction in the branch delay slot is executed whether or not the branch is taken; so we now understand what these two terms mean.

In addition to these two complications from the perspective of the pipeline, I am going to add one more you realize that when we were talking about the instruction set architecture the MIPS 1 instruction set architecture. I showed you extracts of the contents of the instruction set architecture manual which indicated that they was a load delay slot and there was a branch delay slot from this. you should, you will realize that they could be other pages in the instruction set architecture manual which contain other warnings about complications which are raised which might be of interested to the programmer what which arise from properties of the pipeline.

So, I am going to artificially just add one more and this relates to floating point arithmetic instructions. The warning in is as written: the two instructions that follow a floating point arithmetic operation cannot use the value computed by that instruction. In other words, if there is a floating point add instruction then the two instruction and the floating point add instruction writes a result into let say register F 0

You will remember that when we talked about the MIPS floating point instructions I was using a separate set of registers for the floating point values and they were not called R 0 through R 31 they were called F 0 through F 31.

So, this particular warning tells, at this particular warning indicates that if there is a floating point arithmetic instruction such as the F add floating point add which adds the contents of F 2 to the contents of F 4 and puts the result into the register F 0 then the two instructions that follow with cannot use the value computed by that instruction in other words, the two instructions that follow should not use F 0 which is the value computed by the floating point add instruction.

I have just artificially added this third complication to make the analysis a little bit more interesting. So, we are talking about a pipeline implementation of the MIPS 1 instruction set architecture which has both integer and floating point capabilities which has one load delay slot, one branch delay and two slots. After any floating point arithmetic operation in which the value computed by the floating point, arithmetic operation cannot be used ok. Now, we need to think of a specific program to analyze. So, I will use a specific program that I had talked about earlier in passing and it is going to be just a very small piece of code to all of us to do this discussion in short period of time.

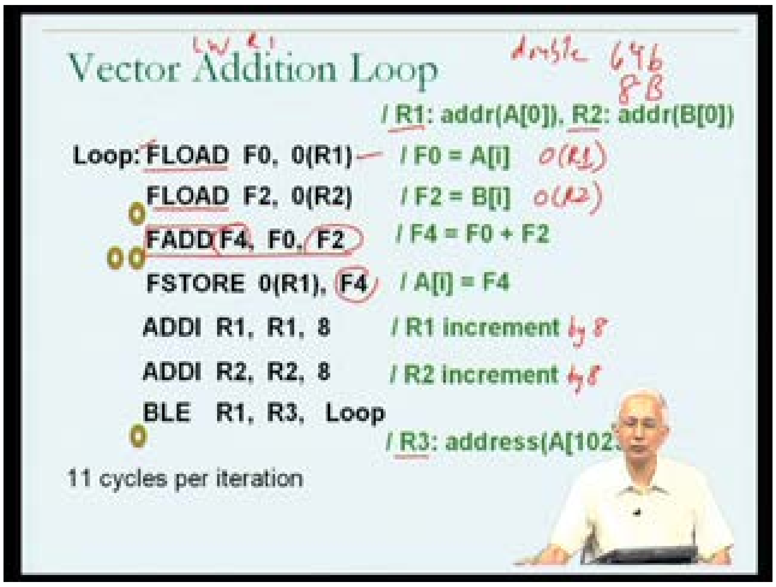
Now, that is what the specific program is code fragment is going to do is vector addition and basically, this C code fragment tells us, tells you what I am talking about. So, in this C code fragment, I have two double procession floating point arrays declared. One of them is called A and one of them is called B; both of them are of size 1024 and these one dimensional arrays are what you could think of is vectors what the code fragment is going to do is it just going to add an element of A to the corresponding element of B in other words, a of 0 added to B of 0 and this becoming the new value of a of 0 and this is done for all of the 1024 elements of the array.

So, essentially what is happening is the vector a is being added to the vector B and the result is the new value of the vector a if one is using vector notation that is what this code fragment is doing which is why I describe this as a vector addition program or a vector addition loop ok.

Now, that the first question that will arise, what will this C code fragment look like in the MIPS 1 instruction set architecture? So, we need to do that compilation, first let me just remind you that we are dealing with two arrays there of double procession. What we mean by double procession is that unlike the single procession the values could you if under I triple E single procession each floating point value was represented in 32 bits; in

I triple E double procession each floating point value is represented in 64 bits right; that is what we mean by double and each of the vectors is of size 1024.

(Refer Slide Time: 13:30)



Vector Addition Loop

double 64b 8B

```
                                   / R1: addr(A[0]), R2: addr(B[0])
Loop: FLOAD  F0, 0(R1)            / F0 = A[i]   0(R1)
      FLOAD  F2, 0(R2)            / F2 = B[i]   0(R2)
   0
      FADD F4, F0, F2             / F4 = F0 + F2
  00
      FSTORE 0(R1), F4            / A[i] = F4
      ADDI   R1, R1, 8            / R1 increment by 8
      ADDI   R2, R2, 8            / R2 increment by 8
      BLE    R1, R3, Loop
   0
                                   / R3: address(A[102
```

11 cycles per iteration

So, if we have to consider writing the vector addition loop in the MIPS instruction set architecture the MIPS instruction set then, what I will do is I will first write it in a pseudo code commented form and then we will get the MIPS instructions to do the same.

So, I will start by assuming that the address of a of 0 remember the loop starts by adding A of 0 with I equal to 0 a of 0 is added to B of 0 the result goes to a of 0 then A of 1 added to B of one the result goes into B of one and so on. So, I will start of by assuming that register R 1 contains the address of a of 0 and register R 2 contains the address of B of 0 and they could be preceding instructions which cause this to happen but, let me assume that this is what we have to start with in the code fragment that we are going to deal with.

Now, in order to add A of 0 to B of 0 I first have to load the value of A of 0 into a floating point register. So, I will assume that that is done so I load the value of A of I. In general, A of 0 in this particular case, into F of 0. After this I must load the value of B of 0 into a floating point register. Subsequently, I can add F of 0 to F of 2 and may be put the result into F of 4. At this point in time, I need to store this result back into A of 0 for which I need a store word instruction but, let me show it as in the pseudo code notation whatever the value in F of 4 is written into A of A of I which is going to require a store instruction after this what has to be done. Now, I am basically writing something which is going to be a loop before I can loop back to the beginning of the loop. Obviously, I will need to increment the value in R of 1; I will also have to increment the value in R of 2.

So, that the second time that this sequence of instructions is executed I do not once again fetch A of 0 but, I fetch A of 1 so the question is, by how much to I have to increment R 1 in order to make R 1 contain the address of A of 1? If you remember that we are dealing with double procession where the size of each vector element is 64 bits in other words, 8 bytes then since the MIPS 1 instruction set architecture talks about byte addressability I will actually need to increment R of 1 by 8 in order to make a point at the address of A of 1.

So, increment R 1 by eight similarly, I will have to increment R 2 also by eight because R 2 at any given point in time should contain the address of the next element of the vector b. So, I increment R 2 again by eight subsequently I need to check whether I have finished doing this operation 1024 times or not and if I have not finished doing it 1024 times I need to loop back to the first instruction in the sequence that is going to be the loop creation.

So, the question is how do I setup this check to see if I have finished incrementing I have finished the vectors every single element of the vector and the way that I am going to set it up for this particular code segment is I am going to assume that I have another register which I called R 3 which contains an address which is just beyond the I'm sorry contains the address of a of 1000 and 23 in other words, it contains the address of the last element of the vector a

So, just like I loaded R 1 to initially contain address of A 0 and I loaded R 2 to initially contain the address of B 0 I load R 3 to contain the address of the last element of the array a the vector a therefore, in order to see whether the termination condition has reached or not I just need to check whether the current value inside R 1 is equal to the current value is less than or equal to the current value inside R 3 and if that is the case I need to loop back.

So, I am using this as a quick way to setup the condition; check for whether to loop back or not from here to be quite easy for us to write the equivalent MIPS 1 instructions. I have to start by writing the instruction for loading into F of 0 from A of I from register from A of I which is going to mean loading into F of 0 from the address which is displacement of 0 from the address contained in R 1 and that is going to be a simple load instruction.

Now, since we are talking about the floating point registers rather than using the l w instruction which was useful for the integer registers, I need to have a separate set of instructions and that notation which I am using is to refer to the floating point load instruction as F load. So, I load into F of 0 out of 0 R 1; similarly, I load into F 2 out of 0 R 2 as we had seen. Then, I do the addition using in F add instruction F add R 4 F 0 F 2 after this I need to store the contents of F of 4 into the memory location associated with A of I which in this case is once again going to be specified by 0 R 1.

So, I will call the floating point store instruction F store; so F store into 0 R 1 from F of 4; then I need to increment R 1 by 8. I can use the integer incremen5it instruction for this; so add I R 1 R 1 8 and increment R 2 by 8 add I R 2 R 2 8. After this I need to have a check to see whether R 1 has gone past the end of R 3 which I can do by checking if branch if less than or equal R 1 R 3 loop, so if the contents of R 1 in other words, I have not gone beyond the last element of address of the vector a I loop back to loop.

So, we have this sequence of 7 instructions which corresponds to an iteration of the vector addition loop and if I analyze this sequence of 7 instructions and let me just go back and remind you what are base assumptions are a base assumptions are that this piece of code is going to execute on a pipeline where there is one branch delay slot one load delay slot and the specific requirement that two instructions following a floating point arithmetic operation cannot use the value computed by that instruction.

So, if I analyze that this piece of code in that light. Then, I realize that consider the first load instruction there is one load delay slot. So, I have to make sure that the instruction following that this load does not use the value being loaded and I find out that is perfectly ok, because, the F load F 2 instruction does not use F 0 I analyze the second F load instruction it loads into F 2 and unfortunately I notice that the instruction following the F load F 2 instruction uses F 2 which means that there is actually a need to separate these two instructions by one instruction potentially by adding a no op between them.

So, I am going to indicate that there is in potentially a need to separate the F load instruction from F add instruction by one no op by putting, I do not know what you call that but, you can view that as being something like a bubble. But, in general I am using that notation in the slide to avoid having to write NOP or no op in fullness.

So, I need to insert something between the F load and the F add to serve the purpose of the no op, so that is one situation that is as it is are there any other situations if has it is we I have taken care of the two load delay slots. I still have to worry about the branch delay slot and the requirement that the two instructions after a floating point arithmetic operation cannot use the value being loaded I am sorry cannot use the value computed by the arithmetic instruction.

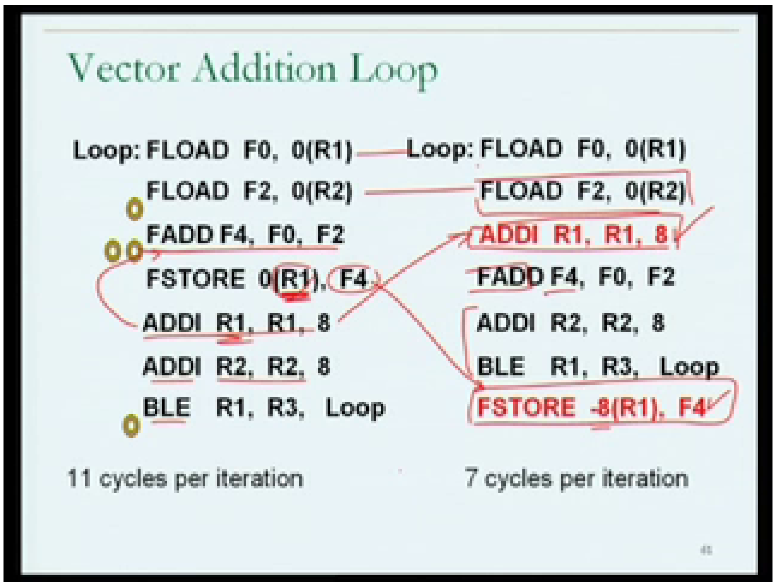
So, immediately I handle the branch delay slot by noticing that there is a need to put a no op or some other instruction into the slot after the branch. Then, I look at the only arithmetic instruction floating point arithmetic instruction in my program and I note that it computes a value; it goes into F 4 and unfortunately, F 4 is used by the next instruction. Therefore, from the third warning about my pipeline I realize that there is a need to put two no ops between the F add and the F store.

Therefore, in analyzing this particular implementation of the vector addition loop, I note that there are 7 instructions which are useful and in addition to that potentially 4 instructions no ops which would have to be inserted in order to cause this vector addition loop to run correctly on the pipeline that we are talking about.

Therefore, I would talk about this particular implementation of the vector addition loop has taking 1 cycles per iteration. What I mean by an iteration of the loop is for the du loop; they constitute iteration one pass through the du loop is what I refer to as an iteration. So, the particular vector addition loop that we are talking about has to execute 1024 iterations and each iteration takes 11 cycles as per the current implementation, 7 useful instructions and possibly 4 no ops.

Now, in our understanding of the pipeline, we realize that we should be able to improve this loop by taking into account the fact that the 11 cycles includes 4 no op's you could try to do something may be some kind of reordering of the instructions to reduce the number of no op's that are required.

(Refer Slide Time: 23:14)



Vector Addition Loop

```
Loop: FLOAD  F0, 0(R1)          Loop: FLOAD  F0, 0(R1)
      FLOAD  F2, 0(R2)                FLOAD  F2, 0(R2)
    0                                 ADDI   R1, R1, 8
   00 FADD   F4, F0, F2               FADD   F4, F0, F2
      FSTORE 0(R1), F4                ADDI   R2, R2, 8
      ADDI   R1, R1, 8                BLE    R1, R3, Loop
      ADDI   R2, R2, 8                FSTORE -8(R1), F4
    0 BLE    R1, R3, Loop

11 cycles per iteration           7 cycles per iteration
```

So, that is what we will try to do next we will try to reason about whether we can reorganize the instructions in this code fragment in some way to eliminate some of these no op's so the basically what we talked about as static instruction scheduling we will go through that exercise just to see to what extent we can improve this code.

So, what I have done over here is I have come up with a particular ordering of the instructions and we will just checked whether this ordering of the instructions is correct to start off with so the F load is we have to use B this F load is we have to use B what I have done is I have taken the add immediate instruction and moved it up the F add is the F add and the add immediate and the B l e are in the order they use to b.

However I have taken the F store and I have moved it down so essentially I have moved two instructions I have moved the add I instruction up by two instructions and I have moved the F store instruction down to the branch delay slot.

Now, the first thing I do not need to check is whether this program is correct whether I have the change the meaning of the program so let us think first about the first change that I made in other words, moving the add I instruction from where it was into the third locations.

So, the first thing that I have to check is I have move the add I instruction above the F store instruction and when I move the add I instruction above the F store instruction I note that the F store instruction is using R 1 and the add I instruction is modifying R 1 and that therefore, this is not a safe move unless I suitably adjust for the use of R 1 inside the F store instruction.

Now, if I move the add I instruction above the F store instruction that would mean that the incrementing of R 1 will happen by eight before the F store instruction is executed and that by moving the add I instruction above the F store I am artificially causing the F store instruction to use the wrong address unless I use a different displacement for the F store instruction correcting for the early addition by the add I instruction hence the minus eight in other words, by moving the add I instruction up and by changing the F store instruction to have a displacement of minus eight I have annulled the effect on the add I instruction from the perspective of the F store instruction.

Now, I have also move the add I instruction above the F add instruction but, you will notice that they are independent of each other and that therefore, that is a safe move therefore, from these arguments we argue the we realize that moving the add I instruction is a safe move.

Now, the next question I seem to have move the F store instruction all the way down to the branch delay slot another is I have moved it past the add immediate instruction and pass the branch instruction

Now, the F store instruction uses F 4 and reads from R 1 I notice that the add I instruction does not affect either and the branch instruction does not affect either does not modify either and that therefore, moving the F store instruction to the branch delay slot is once again safe it is basically does not affect the correctness of the program.

So, my observation is that by making these two by moving these two instructions from the original version of the program program fragment I have not effected the correctness of the program the program is still correct it does on the right does the same thing as a loop on the left.

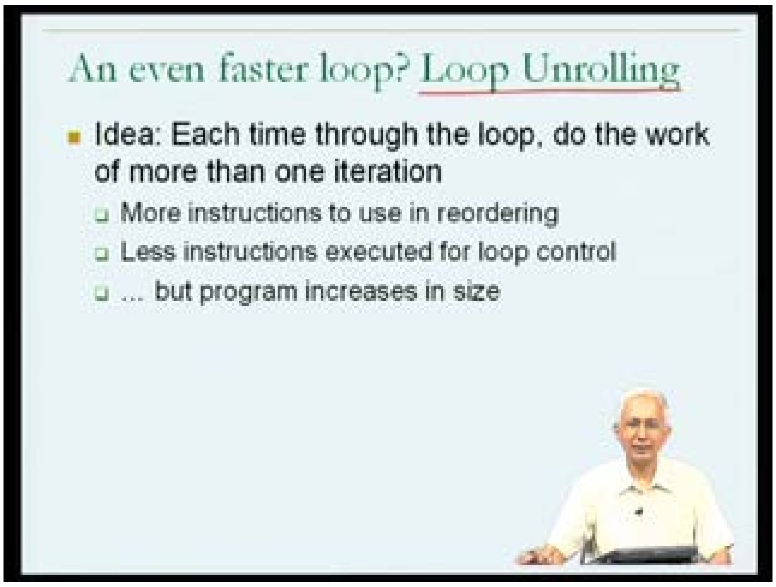
Now, what remains to be done is to try to count how many cycles the new loop will take per iteration so once again we go through the analysis we worry about the load delay slot of the F load and we notice that it is filled by an instruction that is not use F of 0 we worry about the load delay slot of the F load F 2 and we notice that it is fill by an instruction that is not used F of 2.

We worry about the two instructions following the floating point add instruction and we notice that neither of them uses F 4 and finally, we notice that there is a useful instruction in the branch delay slot and hence when we analyze the newly or the reordered the reschedule statically restructured program we come to the conclusion that we do not require any no op's at all in other words, the code on the right will run at 7 cycles per iteration.

So, there has been a substantial improvement in the execution time of the resultant program is come from 11 cycles to 7 cycles that is an improvement by 4 cycles which is better than a thirty three percent improvement in the run time of the program ok

So, that was just a simple example of static instruction scheduling we have seen this before in a simpler example and we realize now that even with more complicated pipeline semantics one could do something similar and improve the quality of instruction sequences. But there is still a question of whether we can do even better than this

(Refer Slide Time: 28:10)



An even faster loop? Loop Unrolling

- Idea: Each time through the loop, do the work of more than one iteration
  - More instructions to use in reordering
  - Less instructions executed for loop control
  - ... but program increases in size

and I am going to introduce one more idea which is often useful and which if one has sufficient understanding of pipelines one could try in cases where the the loop is of great importance to your program. So, if you want an even faster loop there is a concept which we could try which is known as loop unrolling let me talk a little bit about loop unrolling.

Now, the idea of loop unrolling is that we know that we have to execute a particular loop a certain number of times. So, for example, in our particular vector addition loop we know that we have to iterate through the body of the loop one 102 4 times now the idea of loop unrolling is that each iteration or each time we go through the loop rather than doing one iteration of the vector addition we could try to do two iterations of the vector addition. In other words, each time through the loop the sequence of instructions we try to do more than one iteration of the vector addition.

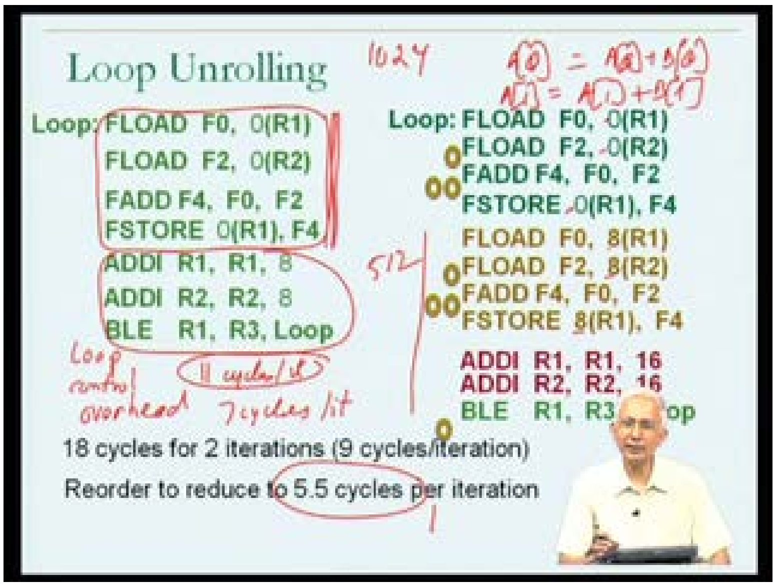
Now, that the question is why would this be useful and there actually several reasons that it might be useful the first is if I actually do two iterations of vector addition each time through the loop this would mean that the number instructions within the loop will increase if there are more instructions in the loop than that gives me more opportunities to try to reorder instructions in order to avoid no op’s.

Therefore, this is a useful property and just for this reason it might be useful for a programmer or a compiler to try to unroll a loop if the loop with the initially was very small it may not contain enough iterations if it is not unrolled it may not contain enough iterations for aggressive statics instruction reordering to allow for improvement in performance of the loop by unrolling the loop by doing 2 or 3 or 4 iterations of vector addition in each pass through the loop one is creating a lot more instructions for potential reordering ok.

Now, another benefit of the idea of loop unrolling is if I do 2 or 3 or 4 iterations of vector addition each time through the loop before branching back that would mean that the number of times that I have to increment the loop variables and the number of times that I have to branch is going to come down in other words, less instructions will be executed for loop control by loop control I mean the incrementing of the loop variables they checking to see whether the loop it is time for the loop to terminate etcetera.

The number of instructions executed for the loop control will reduce which is going to mean that the program can run a little bit faster so this is also positive there is Of course,, possible negative and that is if I unroll the loop and do the work of let say three iterations every time through the loop than the loop is going to become bigger has a consequence the program is going to become bigger in size the number of bytes occupied by the instructions of the program is going to become bigger and if I unroll if I unroll the loop a larger number of times this may end up being a problem because, the size of the program may become substantial but, as long as we do not make the program too large the negative may not be that much of a concerned ok.

(Refer Slide Time: 31:17)



Loop Unrolling

```
Loop: FLOAD  F0, 0(R1)
      FLOAD  F2, 0(R2)
      FADD F4, F0, F2
      FSTORE 0(R1), F4
      ADDI  R1, R1, 8
      ADDI  R2, R2, 8
      BLE   R1, R3, Loop
```

```
Loop: FLOAD  F0, 0(R1)
      FLOAD  F2, 0(R2)
      FADD F4, F0, F2
      FSTORE 0(R1), F4
      FLOAD  F0, 8(R1)
      FLOAD  F2, 8(R2)
      FADD F4, F0, F2
      FSTORE 8(R1), F4
      ADDI  R1, R1, 16
      ADDI  R2, R2, 16
      BLE   R1, R3, Loop
```

18 cycles for 2 iterations (9 cycles/iteration)

Reorder to reduce to 5.5 cycles per iteration

Now, let us look at an example for the code fragment that we have we are going to try to unroll the loop now what I'm using over here in the unrolling example is the original version of R loop this is not the version of R loop that I had rescheduled in order to reduce its execution time from 11 cycles to 7 cycles rather I'm using the original un statistically schedule version of the loop ok.

Now, when I look at his version of the loop I notice that if I look at the first 4 instructions they are the instructions of the loop which are using which are doing the useful work of one iteration of vector addition and if I look at the last three instructions they are the instructions which I would call the loop over head or the loop what it I refer to it in the previous slide loop control

So, this is the loop control over head now my objective in doing loop unrolling is I have each time through the loop I want to do more than one iteration of vector addition which means that I'm going to have more than one copy of the first 4 instructions. So, here I have one copy I have copied it exactly as it is this might be a sequence of code which is doing a of 0 plus B of 0 new value of a F 0.

Now, I also want to do one more iteration in the same loop and that is going to be A of 1 because a of 1 plus B of one now the question is how can I have a sequence of instructions which does similar operations but, with A of 1 rather than a of 0 B of one rather than B of 0 and the answer is I can actually use the same sequence of code but, rather than using a displacement of 0 with each memory access I will use the displacement of 8; 8 being the size of each element of this of these vectors. That is how I come up with the second sequence of 4 instructions which are the brown instructions as you will see. So, they too are copies of the code that was the useful instructions in my original loop.

However, whenever there was A 0 R 1 or A 0 R 2 I replace it by an 8 R 1 or an 8t R 2 thereby actually doing the computation on the next element of the array a or B ok. Now, at the end of this I need to have the loop over head and obviously I need to increment R of one the question is how much to I need to increment R of 1 by you will note that previously I was incrementing R of 1 by 8 because I was in the next iteration I was going to deal with a next element whereas, in the unknown loop I am going to need to increment R of 1 by 16 since I am need to move to not A of 1 but, to a of 2 which is sixteen bytes of a. Similarly, I need to increment R oF 2 by eight by sixteen and finally, I can have the loop terminating instruction.

So, I have a sequence of what is now 11 instructions however remember that I had started with the sequence of code from the unscheduled version of vector addition and therefore, I need to do the accounting of no op's that have to be added to handle load delay slots branch delay slots and the problem with the floating point add instructions

Floating point arithmetic instructions that analysis will be very similar to what we did before they will be the need for one no op after the F load F 2 in both of the iterations and they will be the need for two no op's after the F add in both and finally, one more no op for the branch delay slot.

So, if I add the total number of instructions we will notice that there are 11 useful instructions along 7 no op's which is a total of eighteen cycles but, in this case it is eighteen cycles per a loop per pass to the loop which in which I am actually doing two iterations which means that I am actually if I use this piece of code as it is I am able to execute the vector additional loop at nine cycles per iteration.

Let me just remind you that for the additionally scheduled code we were actually talking about 7 plus 4 or 11 cycles per iteration which we had brought down to 7 cycles per iteration with reordering of this instructions. We now see that by just doing loop unrolling we are able to get nine cycles per iteration ok

Now, in this unroll loop you will notice that there are several no op’s and there also several instructions and as you will imagine is going to be quite easy to reorder the instructions to eliminate all of these no op’s in fact I will leave that as an exercise but, it is possible to eliminate all of those no op’s and end up with 11 cycles for two iterations which is equivalent of 5 point 5 cycles per iteration 5e and half cycles per iteration.

So, we have in effect by doing this loop unrolling improve the performance of this loop by 50 percent it has come from 11 cycles per iteration to 5 point 5 cycles per iteration. Now, you will note that I could conceivably improve the performance of this loop even more if I did more unrolling for example, I could have done 4 iterations of the loop each time 4 iterations of vector addition each time through this loop and in fact I could have carry this through up to a point where I did one 1024 cycles one 1024 iterations of vector addition each time through the loop however that would have been an example of a case where the size of the program may end up being large and a problem to the execution time of the resultant program.

Therefore, we do not consider the extreme case of unrolling the loop completely remember if I had I had this loop which is suppose to iterate 1024 times by unrolling it at this way; this loop now execute 5000 and 11 times.

If I had unrolled it 1024 times then the resultant piece of code would have executed only once. It would have contained an unrolled version of all the 1024 vector element additions but, the program would have ended up being very large. So, somewhere between this mode is attempted loop unrolling and the aggressive attempt of unrolling at all the way there is going to be some point at which there is a very good version of the program which has improved performance without much of the problems of the size of the program.

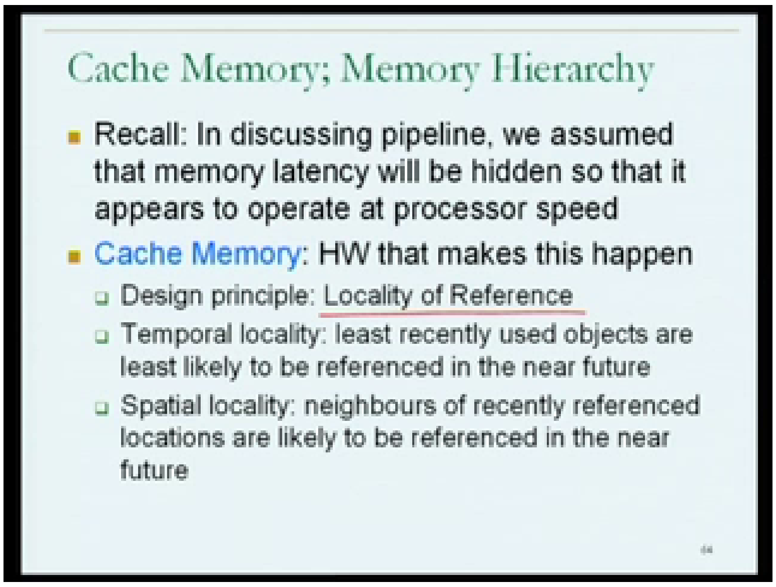
Now, one question which will have a reason in your mind is, in this particular example we have done the loop unrolling while taking the machine code and actually replicating or modifying parts of the machine code or the assembly code and you will realize that you could have done pretty much the same thing by going back to the C code that you had started with. I will roll you back to the C code that we have started with over here and now, if we try to think about unrolling this loop you will realize that I could of unrolled it in C by having each times to the loop not only computations on A of I but, also on A of I plus 1. So, each time through this loop I compute A of I equals A of I plus B of I semicolon as well as A of I plus 1 equals A of I plus 1 plus B of I plus 1.

Then, of course, I will have to make sure that I do not go through this loop too many times and that each time through the loop I increment or each time through the loop I increment by 2. So, I could change this 2 I plus equals 2 thereby reducing the number of times the loop is executed from 1024 to 5012 rather than incrementing by 1 9 increment by 2 each time through the loop.

So, this is one way that one could achieve loop unrolling by manipulation of the C program in addition as it happens we will find out that the compilers like GCC will actually make it possible for you to ask them to do the loop unrolling on your behalf and if you look carefully at the manual entry for GCC you will find out that there is an option for requesting the loop unrolling be done and one could experiment with trying to unroll loops with the help of GCC trying to unroll loops on one zone or trying to unroll the loop in terms of the C language, the C version of the loop and seeing which one results in a better performance for the resultant program and along the way some knowledge of the pipeline will end up being useful.

Now, with this example we have a fairly good idea that some knowledge of pipeline is going to be important on the perspective of understanding what happens when our programs execute and conceivably in improving the quality of those programs may be in terms of execution time and I am going to wrap up the discussion of pipelining at this point.

(Refer Slide Time: 40:10)



Cache Memory; Memory Hierarchy

- Recall: In discussing pipeline, we assumed that memory latency will be hidden so that it appears to operate at processor speed
- Cache Memory: HW that makes this happen
  - Design principle: Locality of Reference
  - Temporal locality: least recently used objects are least likely to be referenced in the near future
  - Spatial locality: neighbours of recently referenced locations are likely to be referenced in the near future

Now, we will now move on to another important part of the processor; you will remember that when we started talking about the execution of instructions we made it the important assumption that the processor the memory system are organized such that there is something called cache memory and that it was necessary for us to be able to ignore memory latencies. You will recall that the one of the big problems in our earlier discussion of computer organization was that we were talking about a processor which updated on a time scale of nano second and a main memory which operated on a time scale of may be 100 nanoseconds 100 times slower.

That therefore, it is very difficult to talk about the activity that happens in one processor cycle when whenever a memory operation happens there was a delay of 100 cycles and therefore, we just made the simplifying assumption that there was something called the cache memory which removes that necessity and made it appear that memories could operate a processor speeds. Therefore, this was important assumption that we made and ran through our discussion of pipelining and made it possible to talk about pipelines with throughputs and speedups related to the number of stages with in the pipeline.

Now, we need to understand more about this very clearly. The cache memory itself is an integral and very important part of the computer system and therefore, I will firmly get into that discussion with a some introductory comments in today's lecture.

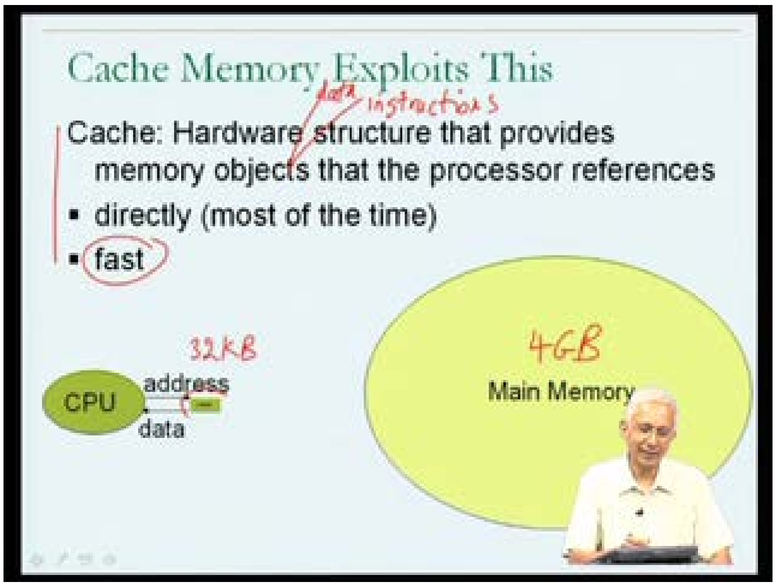
So, basically the cache memory is hardware that makes this assumption that we had made happen or makes the assumption that we had made possible. The piece of hardware that makes reasonable to assume that most of the time memory latencies will be hidden and that the processor can be designed to assume that it operates at processor speed for all it is different activities.

Now, the primary design principle behind the organization of cache memories is something that we have fortunately already seen and that is the principle of locality of reference you will remember that we saw the principle of locality of reference in connection with virtual memory. When we were trying to reason how an operating system could model the behavior of programs and at that time it was important to for the operating system to have some model about how programs behave in order to get some perspective on which pages are likely to be used by the program in the future.

So, the basic concepts of locality of reference that we saw again I will remind about this were two one was temporal locality which suggested that least recently used objects at least slightly to be referenced in the near future. So, this was essentially what temporal locality was telling us and you will note that this was the basis for using page replacement policies such as LRU - least recently used. Although unfortunately, we found that LRU, it was not feasible to implement LRU exactly as a page replacement policy but, in terms of a principle temporal locality tells us something about the least recently used objects being the one that are least likely to be referenced in the near future. Hence, the ones that are best candidates for replacement on the other hand, there was the spatial locality which suggested that if a particular memory location is referenced now then both it and its neighbors a likely to be referenced in the near future.

So, both the principles of locality temporal and spatial are suggesting how programs behave in terms of their behavior in the future and this is the kind of thing which can be used in designing either a page replacement policy in the case of an operating system or in designing as we are now shortly going to find out the operation of the cache memory ok.

(Refer Slide Time: 44:07)



Cache Memory Exploits This

Cache: Hardware structure that provides memory objects that the processor references

- directly (most of the time)
- fast

data
instructions
32KB
CPU
address
data
4GB
Main Memory

Now, as you guess cache memory is exploit the principle of locality of reference and we have to understand that cache is when we use the word cache, we are referring to a hardware structure and that what the hardware does is to provide the memory objects. They could be instructions, they could be data; that is why I refer to them as objects that the processor references.

Now, it provides them directly most of the time; in other words, it does not refer to main memory in order to get them that would have taken a 100 nanoseconds so that would not have made sense and further, that the cache hardware is so fast that it is almost at processor speed which is why we could talk about the amount of time that it takes to access an instruction cache or the amount of time that it takes to access a data cache as being similar to the amount of time that it takes to do an ALU operation when we are talking about definition of cycle time.

So, that was a more less what we were assuming at that time; so this trend is some kind of a definition of what the cache will do for us. It's hardware that provides the data or instruction that the processor references directly in other words, not getting it out of main memory each time and at very low time overhead in other words, very fast.

So, the picture that you should have in mind as far as cache memory is concerned, is there was the CPU which we have learned a lot about. There is a main memory which we have learned enough about the one thing that we know about the main memory is that it is quite large. We know that the main memory could contain gigabytes in processors today 4 gigabytes of main memory is not uncommon for computers of today but, that it is unfortunately slow.

So, we know that whenever the processor generates an address this could be in order to fetch an instruction. On the other hand, it could be in order to load a piece of data from memory into a register. Subsequently, it gets the data back but, we are now being told that rather than the address going all the way to memory and the data coming from memory, the activity may actually be happening out of this thing called the cache memory and I show the cache memory has this very small box at a two common are like to make about the notation which I have used. You will notice that I have shown the cache memory in green rather than in yellow in this slide. I am using green as a color associated with the processor and I am using yellow as a color associated with the main memory. Therefore, by showing you the cache block has being in green; I am clearly saying that view the cache as being part of the processor not as part of the memory.

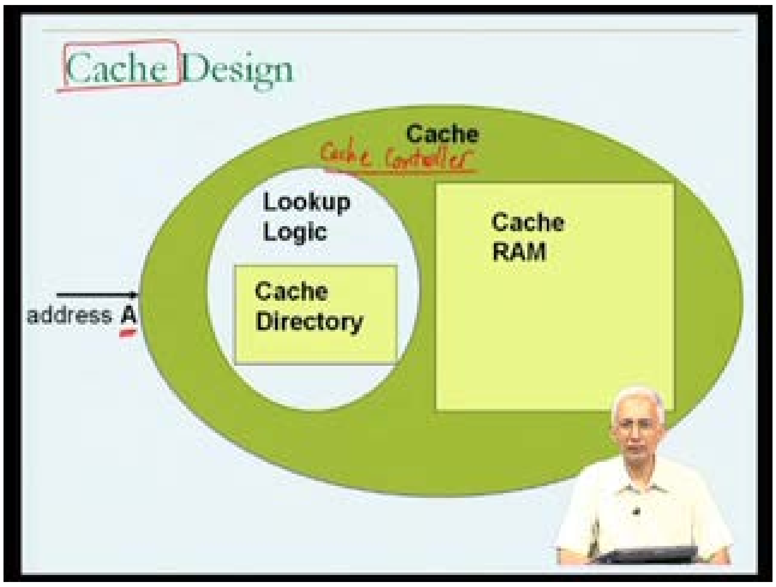
So, if one look at the packaging one might find the cache being incorporated with the processor possibly on a on a chip as opposed to the memory which would be on several chips; other chips the second thing I like to point out is that in comparison to the size of the CPU or in the size of the main memory, I am purposely showing the cache as being extremely small.

In fact, I am showing it so small that you cannot even read the word cache in the label which I have attached there. But, that is just meant to make the point that in practice the cache is so small that it is almost insignificant in size compared to the size of the main memory when we talked about the main memory is potentially being gigabytes in size. Let me just point out for the movement that the size of the cache is actually going to be may be a few tens of kilobytes in size.

So, just note that we are talking about something which is several 1000 times smaller than the main memory giga is in this perspective is be viewed as being billions and kilo is only thousands. So, there is a substantial difference in the size and technically I should have drawn the green blob even smaller. What I have shown it over here is much too large if you look at this size comparison.

Now, this small size is going to be part of the reason for its speed by being small it is going to be possible for cache to be accessed at high speed. Let me I give you some idea about the general principles of operations about the cache through pictorial means.

(Refer Slide Time: 48:17)



So, now I am expanding that tiny cache blob so what we are going to see on the next slide is an expansion of this tiny cache blob because, we have to understand what is happening inside the cache. So, as far as the cache is concerned remember, it is a piece of hardware and the operation of the cache starts with an address coming to the cache from the processor so this is a line some information coming to the cache or request to access the instruction or the data at the address a. Now, obviously within the cache they has to be some amount of very fast memory and as I have suggest that the amount of fast memory may not be too much. It might just be about 32 kilo bytes but, they has to be some fast memory as part of the cache and this is where some of the data which is actually resident in main memory is going to be remembered.

Now, when the address a comes to the cache from the processor that also has to be some part of the cache which is going to be capable of determining whether the address the contents of memory address a are currently present in this fast memory. Note that since the cache fast memory is so much smaller than the actual main memory at any given point in time only a very small fraction of the contents of main memory can be present in the cache memory.

Therefore, obviously the cache has to keep track the cache hardware has to keep track of what the current contains of its fast memory are and when the processor sends an address a to the cache with a request to give a that particular instruction or that particular piece of data. The cache must have some logic some hardware which can determine whether or not that particular entity instruction or piece of data is currently present in the fast memory. Therefore, some logic, some hardware, some circuitry which can answer the question, do I currently have this particular object instruction or piece of data?

Now, in order to answer the question, do I currently have it in other words, is the instruction at address a currently inside this fast memory the cache hardware is going to have to have a table which keeps track of the various things which are currently inside the fast memory right.

So, a table of the addresses that the cache currently has so in order to do the look up, in order to do to answer the question, do I have the object at address a, it has to keep track of the different things that it has in its fast memory and that would come in the form of a table of addresses that are currently inside the fast memory. Now, I have presented the design of the cache in a somewhat non-technical sense I have talked about a piece of hardware which is labeled, do I have it? A table of addresses I have, this is obviously not the technical terms that I use for these entities.

Let me just tell you at the technical terms used for these three entities are, the three entities being the fast memory within the cache which is use to store very small fraction of what is inside the main memory the piece of hardware circuitry which determines whether or not the address a is present inside the fast memory and the table of addresses which are which is used by that piece of logic in order to make the determination.

Now, these three things are actually known as the cache RAM the look up logic or the cache look up logic and the cache directory. So, in general when we talk about the operations of the cache we will talk about a part of the cache which is called the cache controller just as we talked about the control hardware of the CPU that is going to be some hardware within the cache which actually manages the operation of the cache and the cache controller is going to manage the activity.

So, when in address a comes to the cache from the main memory from the processor the cache controller initiates the look up logic with the address a the look up logic looks up in the cache directory to see if the address a is currently there. If the address a is found within the cache directory, the cache directory will contain a link to which particular location in the cache RAM contains that data and the data will be returned to the processor. On the other hand, if the look up logic determines that address a is not currently represented in the cache directory then the situation will have to be handled and we may be talking about the problem similar to the problem that we saw when we are talking about page faulting and we may find out that some similar criteria may come into play in how the cache controller handles the various scenarios that could arise. We will look into these details about the operation of the cache in the few lectures to come but, for the moment that may close by just summarizing that in you know that we have move to the next major topic of our course - on high performance computing in which we are looking at the parts of the computer system which relate to memory.

We are not going to spend too much time looking at main memory itself. We are going to spend reasonable amount of time looking at cache memory, the different possible organizations of cache memory and more specifically, how a programmer must take into account cache design in modifying the way that programs are written; may also make some comments about the role the cache plays in what is called the hierarchy of difference, forms of memory inside a computer system and I will close at this point today; Thank you.