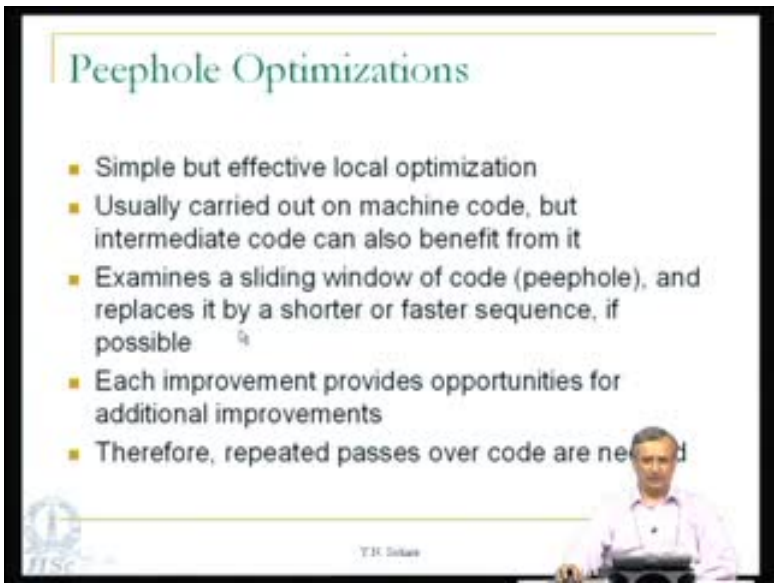
Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 04
Lecture No. # 11
Code Generation-Part 3 and Global Register Allocation

Welcome to part 3 of the lecture on code generation. Today, we will discuss Peephole Optimizations and then continue with global register location.

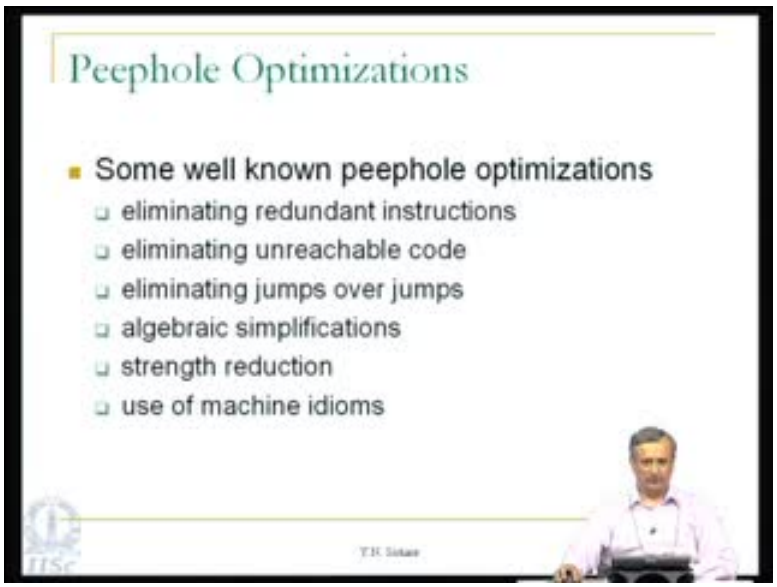
(Refer Slide Time: 00:50)



So, peephole optimization is a very simple, but effective local optimization technique. You know it is usually carried out on machine code, but sometimes it is possible to carry it out on intermediate code and there is benefit by doing it, but the best effects are always on machine code.

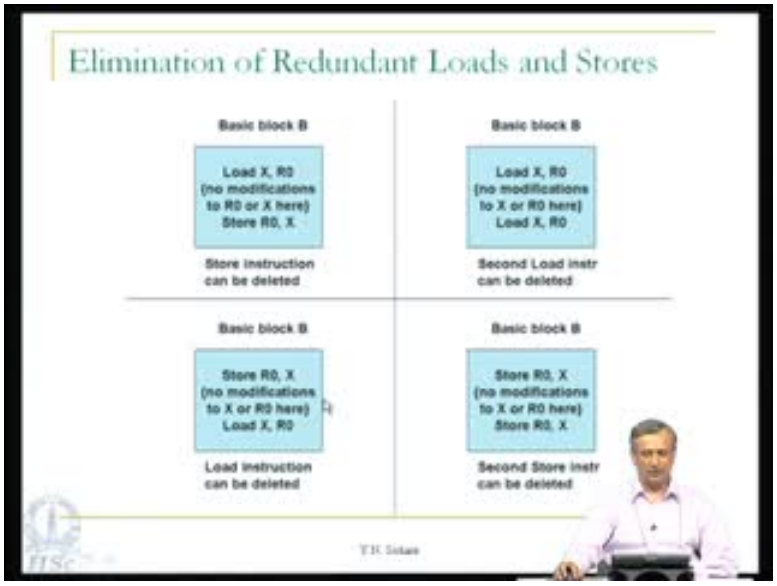
Why it is called a peephole optimization technique? Well, the optimization examines a small sliding window of machine code and this is known as a peephole. You look at the code through this peephole, which is sliding throughout the body of the program. When we find a pattern, which is already known to us and we know it is not very efficient, we can replace it by shorter or faster sequence of code. So, whenever this is possible, we do it, but every time we do this, there may be more opportunities for additional improvements. Therefore, we need to make repeated passes over the code until there are no more advantages possible.

(Refer Slide Time: 02:09)



There are a few well-known peephole optimization techniques that we are going to consider. First one is eliminating redundant instructions, eliminating unreachable code, eliminating jumps over jumps, algebraic simplifications; strengthen reduction and use of machine idioms. So, let me give you examples of what are these in the following few minutes.

(Refer Slide Time: 02:46)



So, Elimination of Redundant Loads and Stores: the picture shows several patterns that can be detected and replaced with better patterns and better code. So, for example, if you consider this particular pattern, there is a load X, R0 and there is a store R0, X.

So, it is quite obvious that if there are no modifications in either R0 or X in this region of the program, then it is not necessary to keep this store R0, X. So, we do all this within

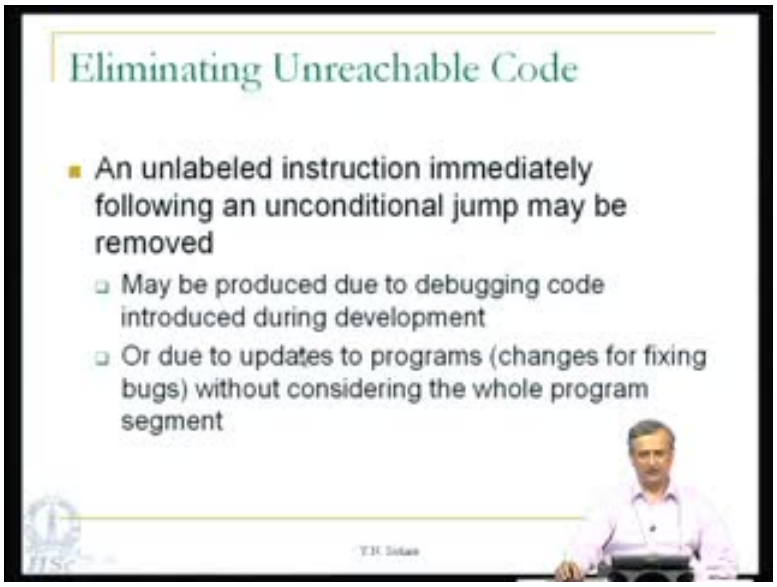
the basic block B because the peepholes are not much larger than a basic block. So, this stores the R0, X can be deleted and only the load X, R0 needs to be retained, provided there are no modifications to R0 and X in this region of code. There is no possibility for control to reach this instruction in any other way because this is a single basic block.

The second pattern is load X, R0 and then load X, R0. So, like in the previous case, you know if there are no modifications to X or R0 in this piece of code in between, there is no need to load X into R0 twice, just one of them will do and so the second one can be deleted.

Similarly, the third pattern store R0, X and then load X, R0. Again, if there are no modifications to X and R0 in between, it is not necessary to load X into R0 once again. It is already in R0 and so the second load instruction can be deleted. The fourth pattern is store R0, X and store R0, X and with no modifications to X and R0 in between. So, in such a case, why should we store R0 into X twice? It is just enough to store it once and the second instruction can be deleted.

So, these are examples with elimination of redundant loads and stores, which can be carried out without trivial, but what happens? These patterns actually creep into the machine code because of machine code generation strategies. It is not easy to foresee that these will happen and take steps to eliminate them during no code generation.

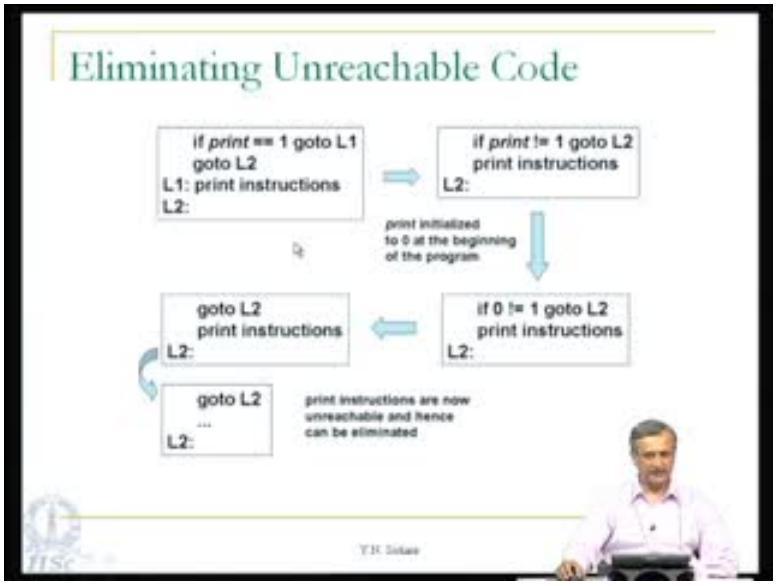
(Refer Slide Time: 05:36)



Now, Eliminating Unreachable Code: suppose an unlabeled instruction immediately follows a conditional jump. In such a case, it is not at all possible to reach this particular unlabeled instruction. I will show you some examples and so what happens is the unconditional jump takes the control elsewhere. Since, the instruction follows the

unconditional jump, it is unlabeled and there is no way we can return to that particular instruction. So, we can remove it and how such sequences are produced? Perhaps, they may be produced due to debugging code introduced during development. One example will be shown later or perhaps due to updates to programs. You know changes for fixing bugs. So, without considering the whole program segment itself, the programmer who was debugging code, would simply introduce some update to the program. Some instruction now becomes unreachable because of this effect.

(Refer Slide Time: 06:52)

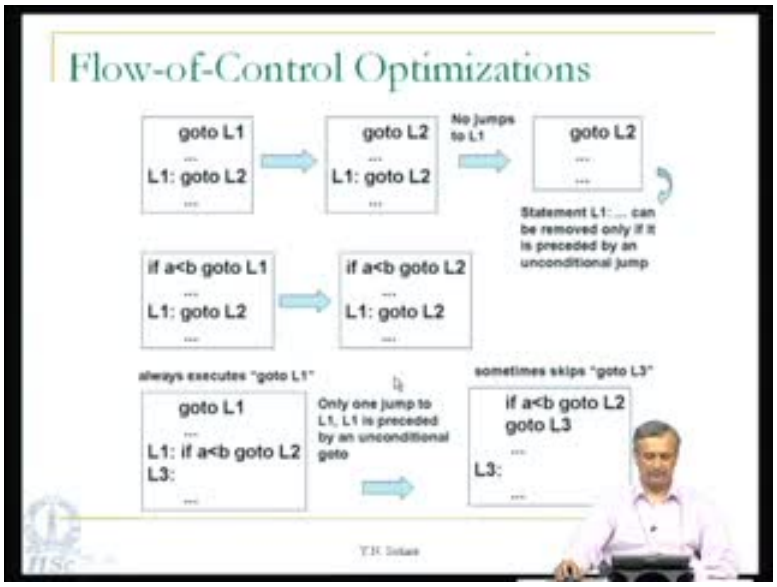


So, here is the scenario. We have the code, if print equal to 1, goto L1 followed by goto L2 followed by L1: print instructions and then L2: Now, here is an unconditional goto and there are instructions following it. So, let us see how transformations change this sequence with unreachable code. So, this print equal to 1 was inserted as 0 or 1 to enable some debugging. Of course, the first step is to rewrite this as: if print not equal to 1, goto L2, then have print instructions. So that means there is a little bit of reduction in the code size and usually this is possible. If print was initialized to 0 at the beginning of the program, this is a variable which is initialized to 0 or 1.

Now, the code changes, if 0 not equal to 1, goto L2 followed by print instructions. So, the compiler will easily know to evaluate this predicate to see that 0 is definitely not equal to 1 and so it jumps to L2. So, the code now becomes 0 not equal to 1 and becomes false. So, the code changes to just goto L2 followed by print instructions and therefore you know 0 not equal to 1 is true. So, this predicate can be evaluated as true and so the code changes to goto L2 followed by print instructions.

Now, you can easily see these print instructions can never be executed. So, these print instructions are now unreachable and hence can be eliminated. So, you just remove them and they are not needed anymore. These were introduced because we need some debugging facility in the code.

(Refer Slide Time: 09:14)



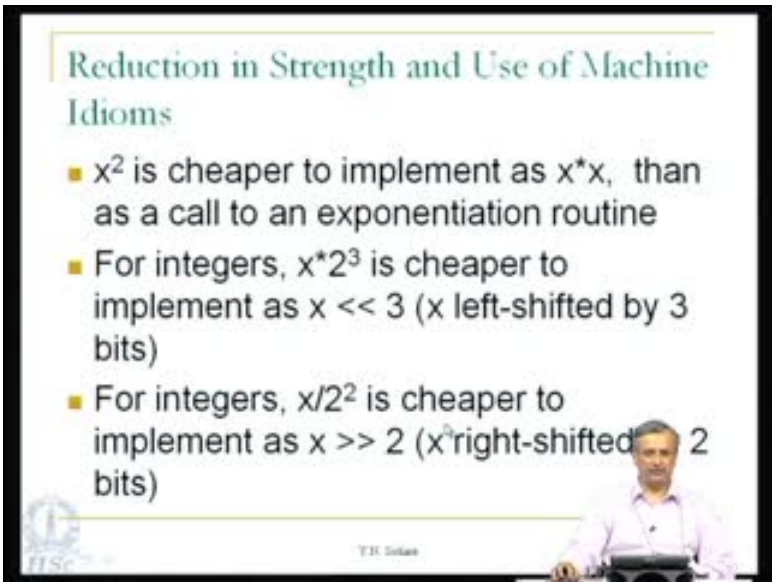
Here is some Flow of Control Optimizations: jump to jumps can be eliminated and let us see how. Here is a piece of code, goto L1 followed by some code. L1 has- goto L2. What we really do now is jump to the L1, which in turn takes you somewhere else and it is L2. Obviously, we are not doing anything, but jumping to L2 and so goto L1 and then L2. Again, we might either change goto L1 to the code goto L2. So, if that is done, there are no jumps to L1 from elsewhere. So, we have no jumps to L1 from anywhere else. It was only here and now this has been changed to goto L2. So, statement L1: can be removed, if it is preceded by an unconditional jump.

Suppose, we had some labels somewhere here, we could jump to this point and then execute. This is the scenario we are looking in this particular statement. So, if this was preceded by an unconditional goto in the previous case, this will not be executed. So, we can remove it and just make it goto L2 and this is how jump to jumps can be improved with the help of other peephole optimizations.

Take this scenario: if a less than b, goto L1 and L1 is goto L2. So, we can change like the previous case. In this case, we change it to- if a less than b, goto L2 and L1 remains as goto L2. Now, this particular code and this is the third one. So, this is the second one and one more. Now, there is a goto L1 and here that is in the third pattern. L1 has the code- if a less than b, goto L2 followed by L3. So, in this case, we jump to L1, check this

predicate and then goto L2. So, only 1 jump to L1, if L1 is preceded by an unconditional goto. So, in that case, we can change this (Refer Slide Time: 11:47) to this particular piece of code. Now, suppose we change it to- if a less than b, goto L2 followed by goto L3. So, in this case, it is possible that goto L3 is not executed sometimes. It skips and goto L3, whereas in this, we have goto L1. We always goto L1 and check this, then jump to L2. So, the number of jumps here is 2, whereas, the number of jumps are quite often. This is possible only if you know 1 jump to L1 and so L1 is preceded by an unconditional goto. In that case, we can perform this particular optimization otherwise, (Refer Slide Time: 12:30) this code may have some jumps into it and then this may be executed. So, we cannot remove this without this condition.

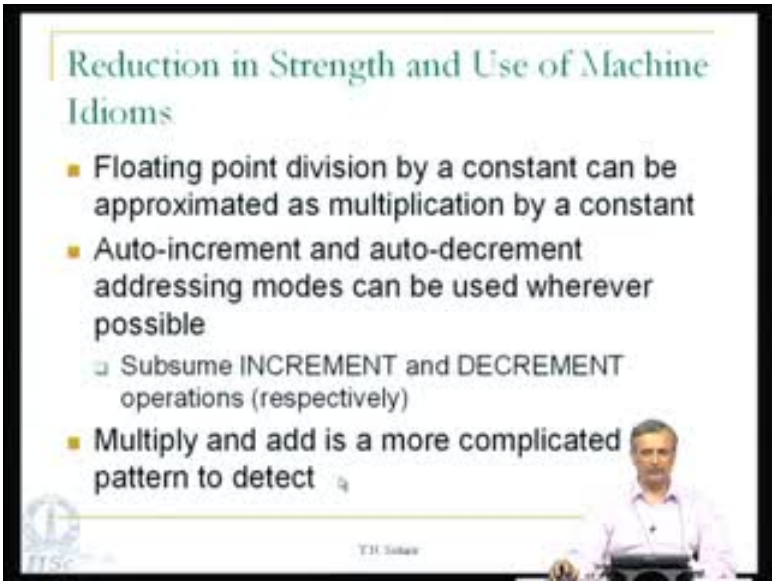
(Refer Slide Time: 12:40)



Reduction in Strength and Use of Machine Idioms: so, for example, it is cheaper to implement x square as x star x and it is very well known, than as a call to an exponentiation routine. So, if you find an exponentiation routine, then we can replace it by x star x. For integers, you know x into 2 cube is cheaper to implement as x left shifted 3 times. So, x left shift once will actually multiply it by 2 and by 3 times it multiplies by 2 cube.

The other way is x divided by 2 squared and it is cheaper to implement as x right shift 2. So, you right shift x by 2 bits, then you have implemented x by 2 square and the second option is always cheaper. So, if we find these instructions in the code, they can be replaced by these left shifts and right shifts appropriately.

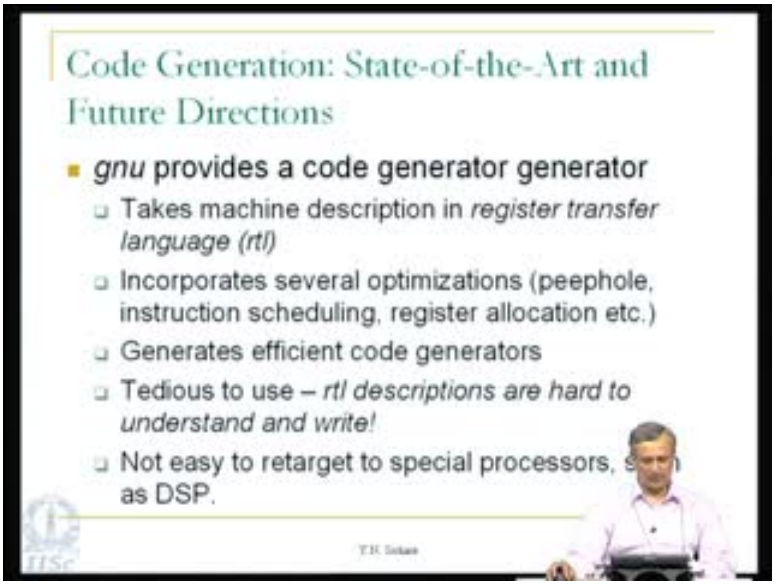
(Refer Slide Time: 13:51)



Reduction in Strength and Use of Machine Idioms: so, if you have a floating point division by a constant, then it can be approximated as multiplication by a constant. So, 1 by a constant can be evaluated by the compiler and instead of doing a division, we can use this reciprocal as a fraction and multiply the variable by that constant.

So, auto increment and auto decrement addressing modes can be used wherever possible. So, what we really try doing here is the increment and decrement operations on the registers, which are performed as a part of auto increment and auto decrement addressing modes. They are actually subsumed within these addressing modes and thereby we save some instructions. However, there are complicated patterns such as multiply and add, which are not so easy to detect. Perhaps, these require much more effort at the time of code generation.

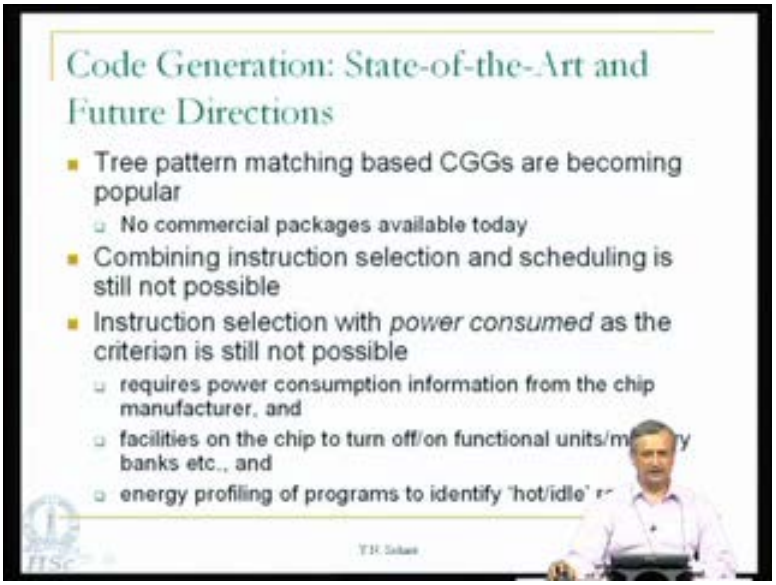
(Refer Slide Time: 15:06)



Now, let us take a very brief look at the State- of- the- Art and Future Direction in Code Generation: the reason is this is a very important topic and we must understand the current status and see where it is going. The most important code generator is the one provided by gnu, which is from free software foundation. It takes machine description in register transfer language- rtl. It incorporates several optimizations, peephole, instruction scheduling, register allocation etc. It generates very efficient code generators and the code generators are produced have excellent code, but they are very tedious to use. rtl descriptions are very hard to understand and even harder to write debug etc.

Normally, people do not write the new specifications starting fresh and they always try to change the old specification and make it new. So, in this process, there will be some optimization may even be lost. It is not easy to re target to special processors, such as DSP. So, these are some of the difficulties with gnu, however there are lots of people who are addressing such issues. Hopefully, in the long run, we will have a much better gnu provided cgg.

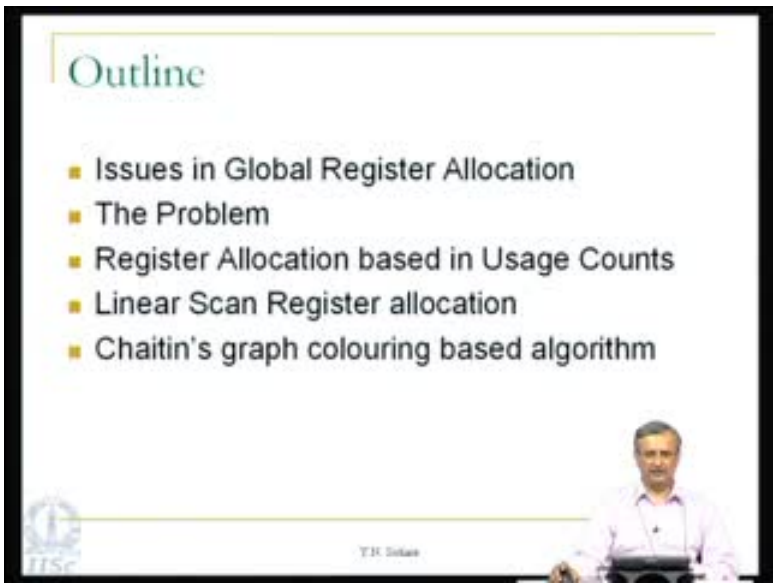
(Refer Slide Time: 16:45)



What is the future direction? Tree pattern matching based code generation is picking up. It is becoming very popular; however, there are still no commercial packages available today. You still have to find out how you know instruction selection and instruction scheduling, which is nothing but reordering of instructions to take care of pipeline stalls can be done.

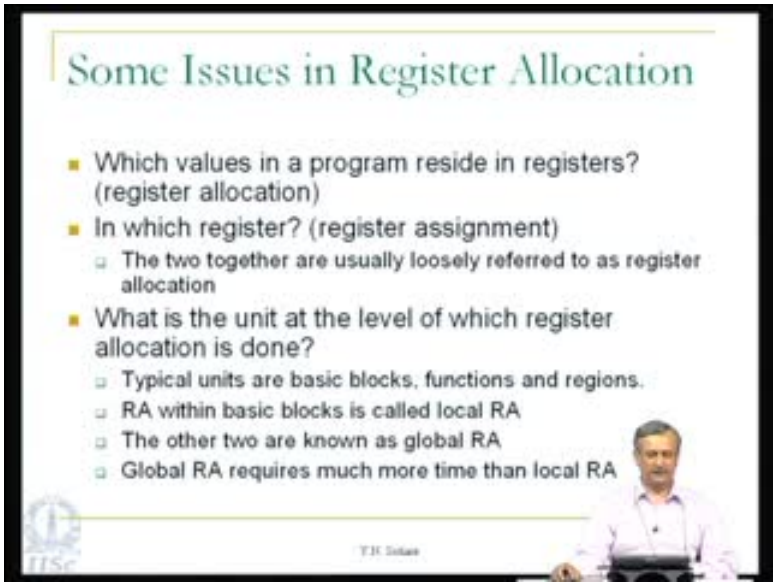
We still do not know how to combine instruction selection with power consumed as the criterion. So far we have seen how to use memory space or the time of execution to make it more efficient etc. However, power consumed is not yet a criterion and the reason for doing this is it requires power consumption information from the chip manufacturer. They cannot be easily found out by experimentation. Facilities on the chip to turn off functional units, memory banks, cache lines etc are necessary. Energy profiling of programs to identify hot and idle regions are also essential. Now, this is the end of code generation and we will start a new lecture on register allocation.

(Refer Slide Time: 18:16)



We will begin with the lecture on global register allocation. First of all, what is the outline of this particular lecture? We are going to consider the issues in global register allocation. Why it is important and so on. Look at the definition of the problem, what is that state? It precisely looks at two simpler methods of register allocation. One is based on usage counts and the other is based on linear scan. We will discuss the most celebrated register allocation algorithm, which is Chaitin's graph colouring based algorithm.

(Refer Slide Time: 19:02)



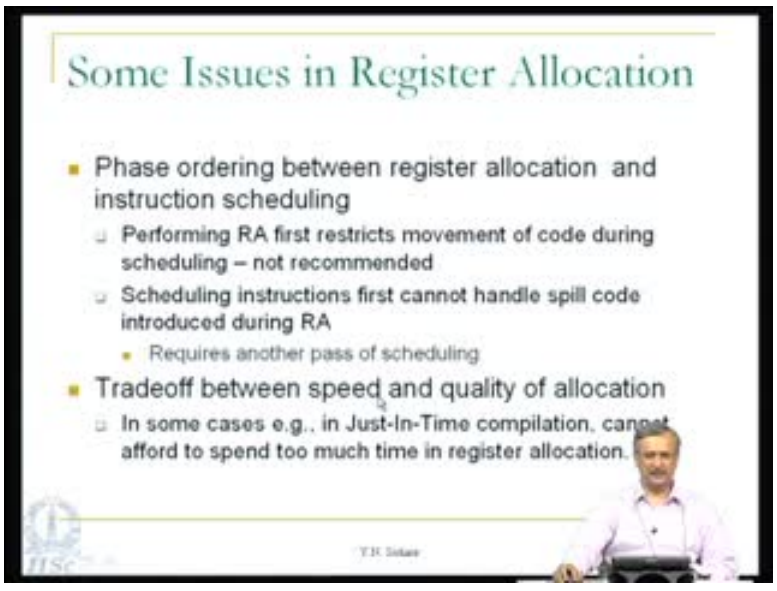
Here are some issues in register allocation, whose values in a program reside in registers. So, this is an issue, which is called as register allocation and whose variables have to be put into registers and what variables should not be put into registers is a difficult issue.

So, this is a slightly easier problem and once we know that x will be residing in a register, the question now is which will be the available registers used for x? It is called as register assignment problem. In general, register allocation and register assignment together are very loosely referred to as register location. So, we will also use the same terminology and we will not deal with these two problems separately.

What is the unit at the level at which register allocation is done? We have seen one simple register allocation strategy; getreg in the code generation lecture, which was confined to basic blocks. So, getreg looked at the next use information of various variables and then using some heuristics it released registers as and when necessary. So, typical units are basic blocks, functions and regions. In this particular global register allocation, we will look at functions. Basic blocks will be a part of these control flow graphs corresponding to the functions and something more than functions is also possible, but this lecture will not consider them.

So, register allocation within basic blocks is called local register allocation and within functions and within regions it is referred to as global register allocation. Obviously, functions and regions are much bigger than basic blocks. So, global register allocation requires much more time than local register allocation and also it is more complex.

(Refer Slide Time: 21:24)



There are two very important issues, which will bug us. The first one is phase ordering between register allocation and instruction scheduling. We have not discussed register and you know instruction scheduling so far. It suffices to tell you now that instruction scheduling is nothing but reordering of instructions, so that pipelines are not interlocked and there is a problem. Suppose, we perform register allocation first, then perform

instruction scheduling. Now, instruction scheduling is supposed to move instructions in the end, change the order compared to the one that is provided.

Once we fix the registers and these are the registers used by variables. Such a movement becomes very restricted and therefore this scheme is not recommended. Whereas, scheduling instructions first creates a different kind of problem. So, we have scheduled instructions and the instructions have been reordered, so that interlocks are minimized. In register allocation, as you will see very soon or we have already seen it in the basic block scheme. It introduces spill code, if there is no register available, then the number of instructions, the timing of instructions etc has all changed.

The instruction schedule, which was produced earlier is not valid anymore. So, this is the problem and what we really do is we do instruction scheduling first, then do register allocation. After all the spill code is introduced, just do instruction scheduling all over again. So, this will solve the correctness problem, which was introduced by spill code. So, this is not exactly the best or optimal strategy, but these problems are all NP complete. So, we will not be able to do better than providing a heuristic for this problem.

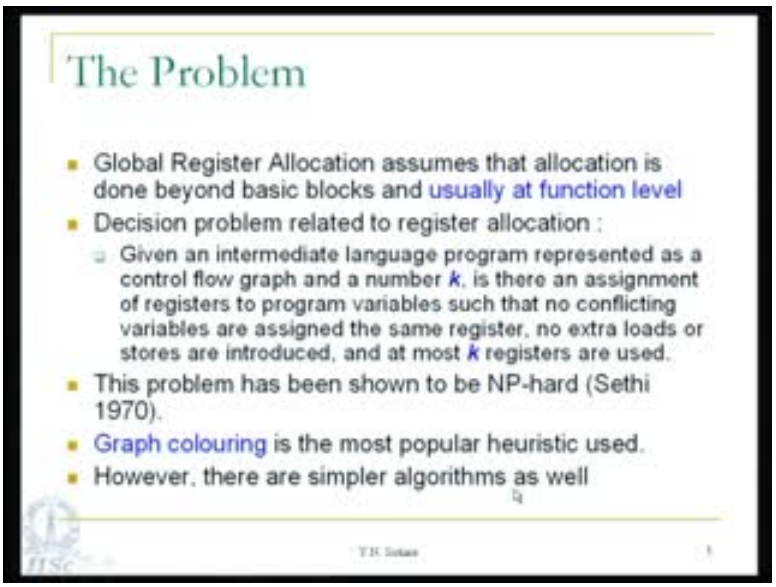
The second important issue is the tradeoff between speed and quality of allocation.

Suppose, we are considering about using register location in a just in time compiler, just in time compilers, for example, for Java, they take a piece of the Java program in the form of Java byte code, then they compile it. When the program is running, it produces machine code. From the next execution onwards that particular machine code will be utilized. So, in other words, let us say a program is running, we have come to a loop and the loop body is quite large. It takes a lot of time to execute the first time and the Java byte code is interpreted as usual.

In the second time, the body of the loop consists of Java byte code and it will be subjected to just in time compilation. Now, this JIT compilation process produces native machine code. So, from the second or third iteration onwards, this machine code is used instead of the Java byte code. Therefore, the speed of execution of the loop goes up. Obviously, executing machine code is faster than interpreting Java byte code. Now, the problem is when we are producing machine code, we require register allocation. If we use a very expensive method of register location, register allocation itself takes a lot of time. So, the program slows down considerably and this is not allowed. So, we actually need to take a very light weight register allocator and implement it during just in time compilation.

Whereas, if it is offline compilations like gnuc or cpcc, there is no problem. We can use a very expensive register allocator and still you know there will be no issues of taking too much time etc.

(Refer Slide Time: 26:12)

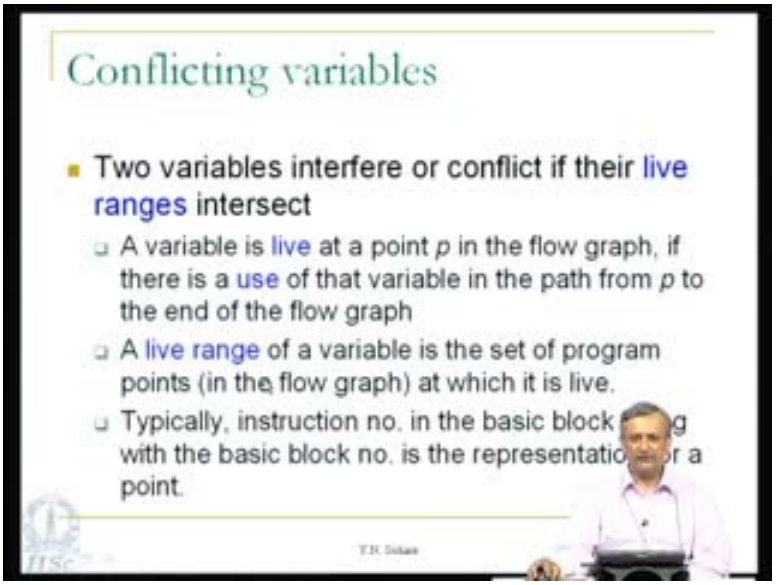


What exactly is the problem of register allocation? Global register allocation assumes that allocation is done beyond basic block and usually at function level. So, the decision problem related to register allocation can be stated as follows: Let us go through it slowly.

Given an intermediate language program, this is represented as a control flow graph. We are also given a number k. We want to find out, if there is an assignment of registers to program variables, such that no conflicting variables are assigned to the same register. So, in other words, if you have 2 variables, both of them are actually active at the same time. We cannot give both of them the same register and that is very obvious. Further, we do not want to introduce any extra loads or stores unnecessarily and at most k registers are used. So, you are given only k registers, how do you use this in the most optimal manner and that is the problem.

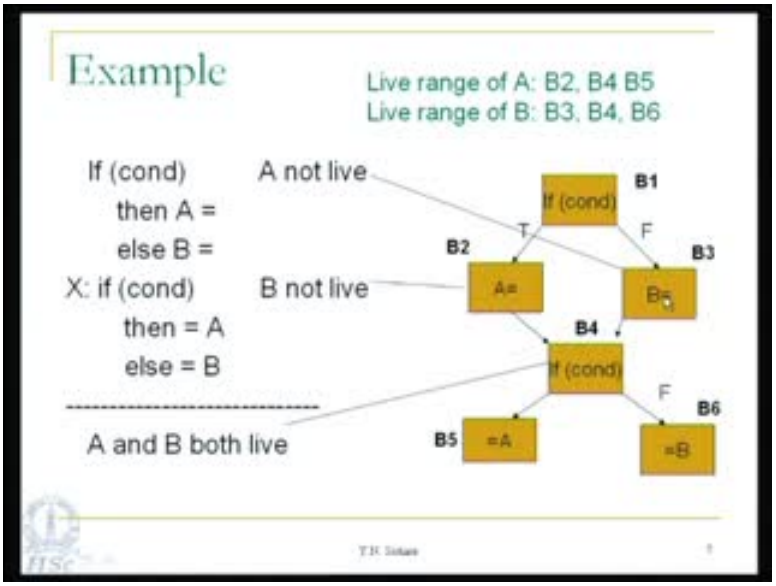
This has been shown to be NP complete by Sethi in his landmark paper in 1970, The Solution, which is a landmark is using graph colouring. This is the most popular heuristic to date; however, there are few simple algorithms. So, we will consider the simpler algorithms first and then consider the graph colouring algorithm, otherwise the charm of the simpler algorithms may be lost.

(Refer Slide Time: 28:10)



Let us go through a few definitions. Two variables interfere or conflict, if their live ranges intersect, so what is a live range? We need to define a live range and to do that we need to define what is live. So, a variable is live at a point p in the flow graph, if there is a use of that variable in the path from p to the end of the flow graph, it is very simple. Take a point p and consider the rest of the flow graph till the end of the control flow graph, if there is any use of that particular variable, then the variable is live at that point p and it is useful later on at that point.

(Refer Slide Time: 29:47)



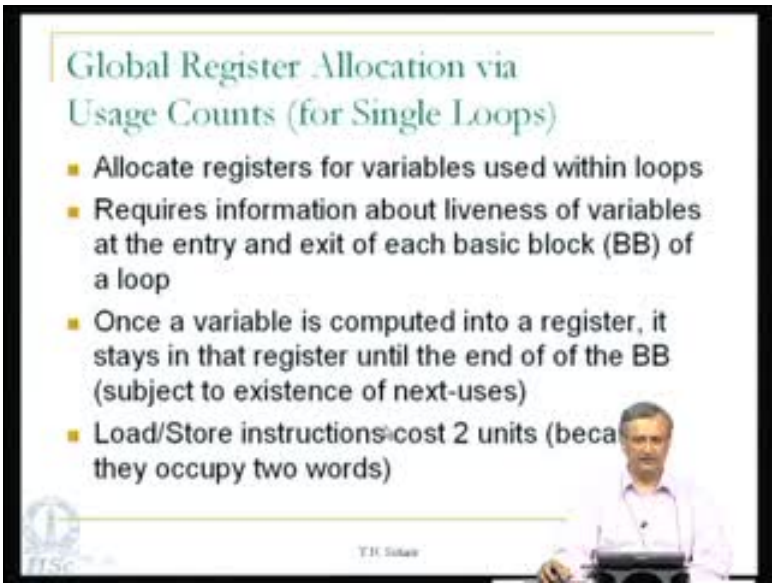
What is a live range? Live range of the variable is the set of program points in the flow graph at which it is live. So, from the first point, where the variable is defined until the last usage of that variable and after that the variable is not used anymore. So, all the

points, where it is live in the program is called as a live range. So, we will see a simple example. Typically, instruction number in the basic block along with the basic block number is the representation for a point. So, these are the coordinates of a point.

So, look at this example. Here, (Refer Slide Time: 29:43) is a program and here is the flowchart, if condition then assignment to A and if it is true, if condition is fault, it is assigned to B. There is a condition here and there is a usage of A in B5 and a usage of B in the block B6. So, if you look at this particular block, B is not used here and so B is not live. If you look at this block, A is not used here and so A is not live; whereas, if you look at that within this block, A and B are both live. Therefore, if you consider the live range of A, so when I said live, we are looking at just that block. Whereas, if you consider this entire thing until the end of the basic block, you know A B everything is live. If you take this path, then A is not live. Look at it; if you take this path, A is live along this path and so live range of A is considered as B2, B4 and B6 sorry it is B2, B4 and B5.

The live range of B is considered as B3, B4 and B6. This set of blocks and this set of blocks constitute the live range for the variables A and B. So, observe that these 2 blocks, actually these 2 live ranges intersect have a common block B4. So, it is not possible to actually assign the same variable, same register to both A and B. You can only assign the same register 1 register to A, but then you have to assign a different register to B.

(Refer Slide Time: 31:49)

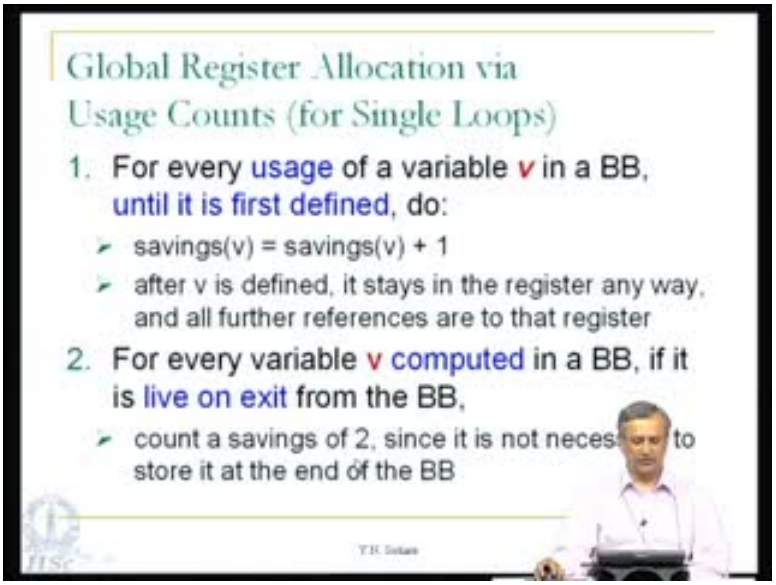


The first register location scheme is using usage counts. Let us first consider single loops and not nesting. We have a loop and we have some variables used within the loops. So,

this particular scheme allocates registers for variables within loops only. So, it does not look beyond loops. It is better than a basic block, but it is not a full function or procedure yet. It requires information about liveliness of variables at the entry and exit of each basic block of a loop.

Once a variable is computed into a register, it stays in that register until the end of the basic block; we are not going to kick it out - subject to of course, existence of next-uses. If it is not used at all, then we can release that register and not otherwise. Load or store instructions are assumed to cost 2 units of time because normally they occupy 2 words. The instruction occupies 1 word and the address occupies another word.

(Refer Slide Time: 33:11)

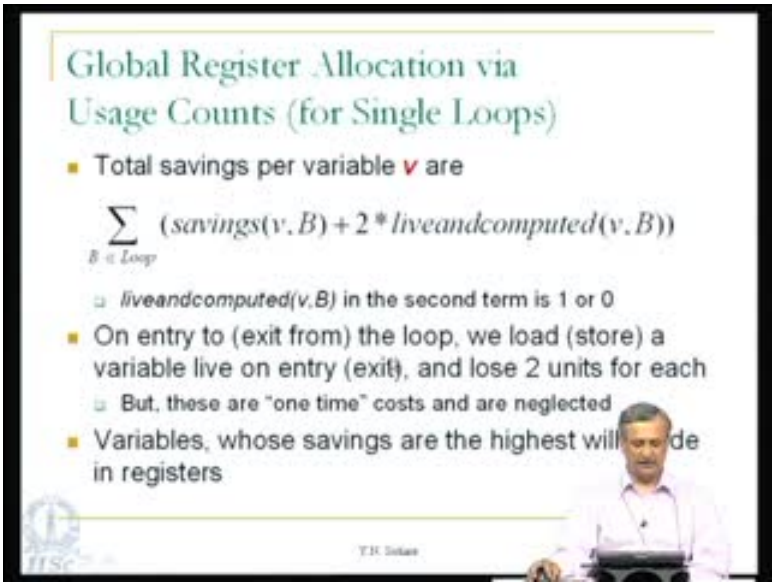


Let us define the usage counts and the savings that occur by doing certain things for every usage of a variable v in a basic block, until it is first defined, do: savings v. It is savings of the variable v and it is savings v plus 1. So, we increment the savings of the variable v. Why for every usage until it is first defined? After v is defined, it stays in the same register until all further references are now to that particular register. So, we do not gain anything after the variable is computed because whatever is computed, it is always into a particular register. This is why, if we had put the variable into a register right at the beginning of the basic block, then even these usages until it is defined to that register. We would have saved 1 unit of time. So, why are we looking at only 1 unit? You know once it is loaded, we are not going to throw it and that is why it is 1 unit.

For every variable computed in a basic block, if it is live on exit from the basic block, you count a savings of 2. Since, it is not necessary to store it at the end of the basic block. So, the variable is computed in a basic block and then it is live at the end of the

basic block. So, we are not going to store it into memory at the end of the basic block. So, there is a savings of 2 for that store instruction.

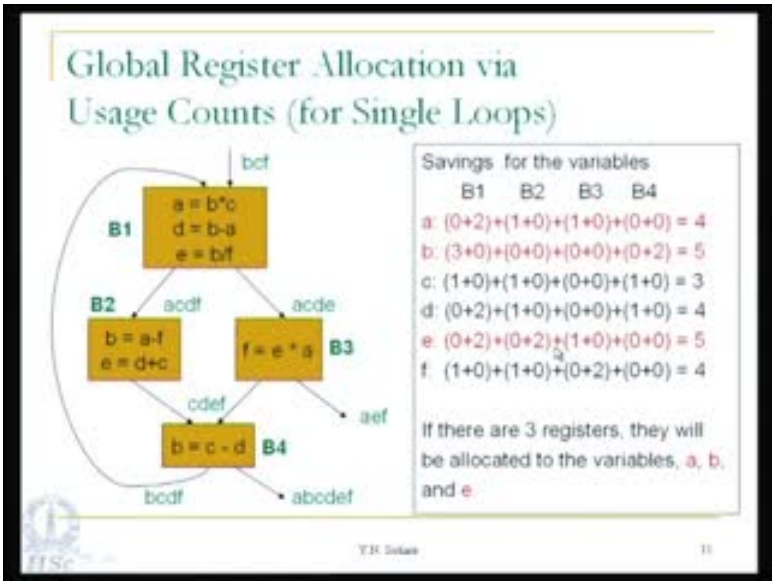
(Refer Slide Time: 35:17)



Now, the total savings of v,B is computed over all the basic blocks, 2 into live and computed of v,B. This particular live and computed v,B is a Boolean variable 1 or 0. So, this corresponds to the variable, which is computed and it is live at the end of the basic block. The condition 2 is that we saw before and this is the saving for each variable, which is used and these are the number of times we incremented before. It is defined, then on entry and exit from the loop; we load a variable live on entry and lose 2 units for each.

So, in other words, when we enter the loop first time, we need to load all the variables which are live on entry to the loop. So, for each variable, we need 2 time units to load it. At the end of the loop, once we are exiting, we need to store it into the memory. So, we are losing 2 units for each variable, but these are one time cost and we are neglecting them. So, variables whose savings are the highest will reside in register.

(Refer Slide Time: 36:56)



Let us take an example; here is a very simple control flow graph. On entry, bcf are live, then there are three instructions on this side. We have acdf live on this side and we have acde live. On entry to this basic block, we have cdef live on exit from the basic block. Through this arc, we have bcdf as live and on exit from the basic block, on this side we have all abcdef live on exit from this basic block and we have aef live. So, here are the savings for the variables.

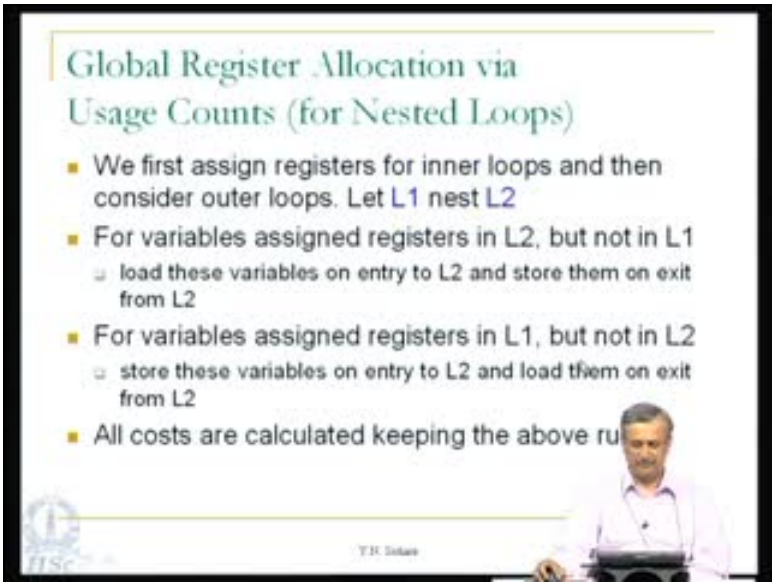
Let us take variable a and see how other contributions are. So, a is used only after a definition, it is not used before a definition, so the savings from usages is 0 and you know the second component, so that you know live and computed part. It is computed and live at the end of the basic block. So, we do not have to store it back into memory and savings is 2, whereas for B2, a is used and it is not defined yet. So, there is a saving of 1 and then a is not defined. So, the live end is computed as 0 and this is a 0.

In B3, again a is used and it is 1, it is not computed and it is 0. Second component is 0. B4 is neither computed nor used and both are 0, so the total savings is 4. Similarly, if you consider f for B1, in the basic block B1, f is used and it is not defined anywhere. So, the savings is 1 and the other part is live and computed as 0 in B2, f is used once. So, it is 1 here and the second part is not defined. So, it is 0 and third one is B3, f is defined and it is not used. So, it is 0 plus 2 and then fourth one is neither used nor computed as 0. So, the savings is 4 and others are also similar.

For abcdef, we get 4, 5, 3, 4, 5, 4. If there are 3 registers, they will be allocated to the 3 variables a b and e because b and d have five as the usage count, savings. a has 4 and it could have been a or it could have been d arbitrarily. Let us say, if we have chosen a,

others will all remain in memory. They will be loaded into memory and stored into memory as soon as the usage is over.

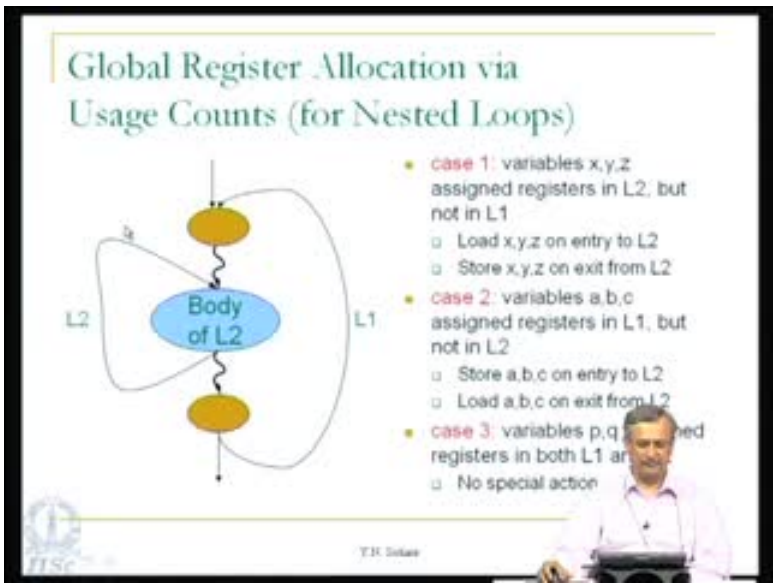
(Refer Slide Time: 40:07)



Suppose, we have nested loops and what do we do? So, we first assign registers for inner loops and then consider the outer loops. So, let L1 nest L2 and we have 2 cases here. For variables assigned registers in L2, but not in L1. We need to load these variables on entry to L2 and that is very obvious. So, we have given them registers in L2, but the outer nesting level L1 does not give them registers. So, in L1, when they were executing, those registers were used for some other purpose.

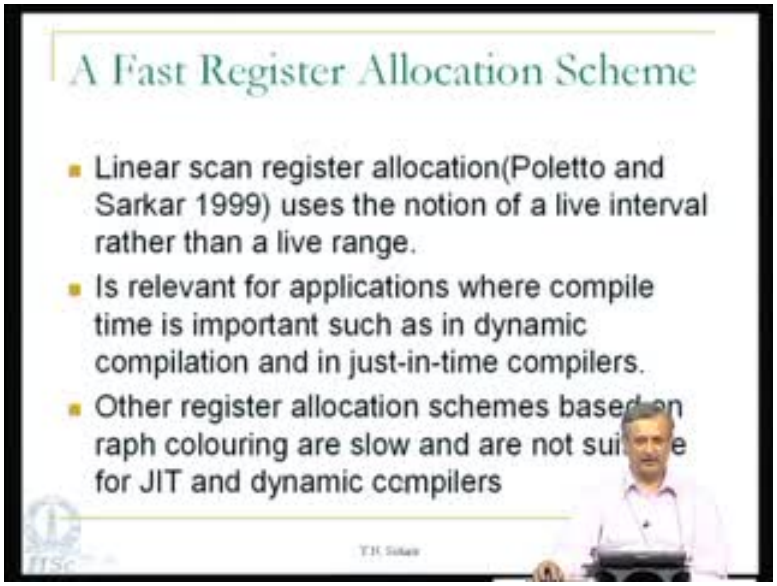
Now, we load these variables into L2 and then store them on exit from L2. So, this is the cost and actually the registers will be free at the end of L2. For variables, which are assigned in a registers in L1, but not in L2 and so this is the other complement. We store these variables on entry to L2, so that the registers can be used in L2 for other variables of L2. Load them on exit from L2 and that is the restoration process. So, these costs must be calculated while keeping the above rules in mind.

(Refer Slide Time: 41:23)



Here is a simple example. This entire thing is L1 and this is L2 and the body of L2. Case 1 is variables x, y, z are assigned to registers in L2, but not in L1. So, we need to load x, y, z on entry to L2 and then store them back to memory at this point in the exit. Case 2 is variables a, b, c are assigned to registers here, but not here. So, before we enter L2, we need to store them into memory. After we exit L2, we again store them, load them from memory and use them in registers. So, if p and q are assigned in registers both L1 and L2, it is a happy situation nothing special has to be done.

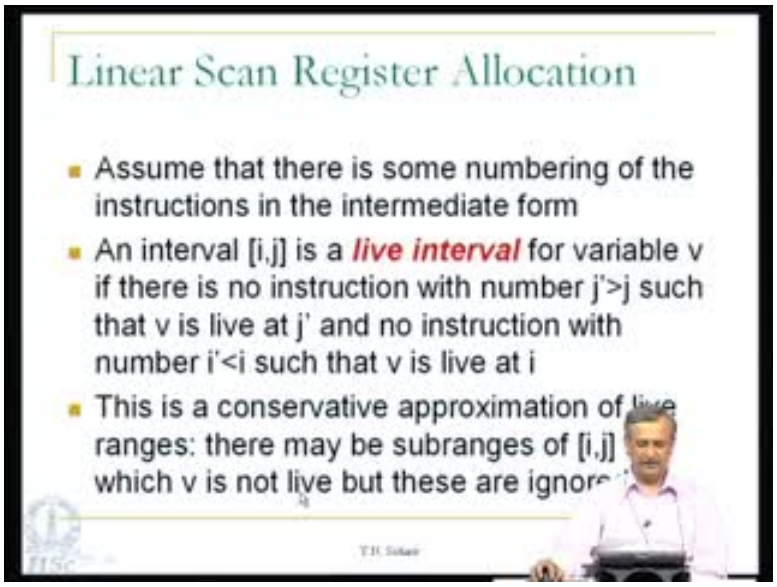
(Refer Slide Time: 42:13)



Let us look at linear scan register allocation scheme. So, we saw the usage count based scheme so far. This is a fast register allocation scheme and it uses the notion of a live interval rather than a live range and why this is simpler? It is relevant for applications,

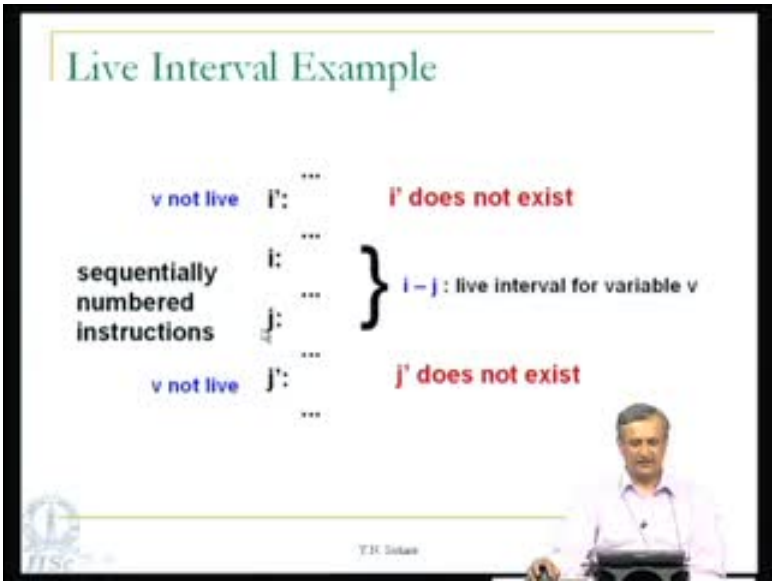
where compile time is more important, such as dynamic compilation, just in time compilers and I have mentioned this already. Other resistor allocation schemes are based on graph coloring, this is a small mistake here, it is not raph and it is graph. They are slow and are not suitable for JIT and dynamic compilers and so I already said the reason.

(Refer Slide Time: 43:02)



Assume that there is some numbering of the instructions in the intermediate form. So the quadruples or byte codes have been numbered 1, 2, 3, 4 etc. Now, what is a live interval? An interval of instructions from i to j is a live interval. I will give an example of this very soon. The intervals are always for the variable v. If there is no instruction with number j prime greater than j and beyond j, such that v is live at j prime and no instruction with number i prime less than and that is before i, such that v is live at i. This is a conservative approximation of live ranges, but there can be sub ranges of i, j in which v is not live, but these are ignored. So, let me give you an example.

(Refer Slide Time: 44:04)

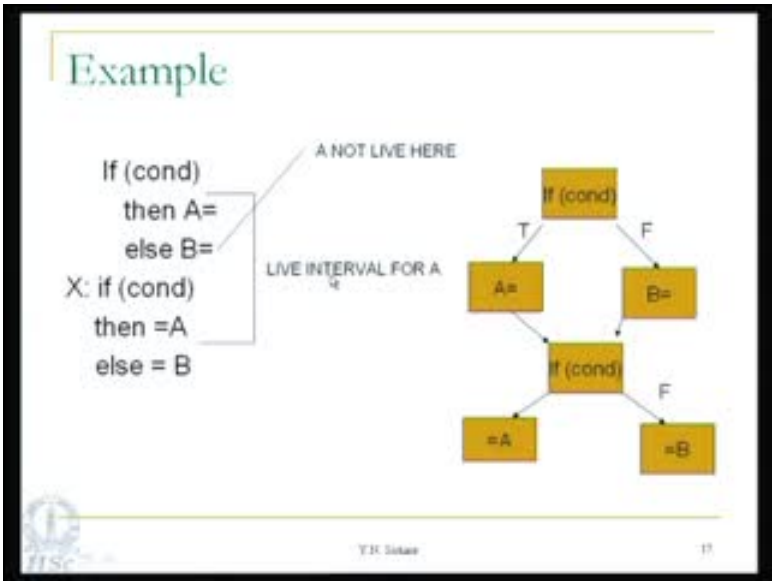


So, we have a linearly numbered sequence of instructions here. So, here i prime, i, j, j prime etc. This is the interval we are considering for the variable v. So, in the previous explanation, we said i prime and j prime do not exist; whereas between i and j, a live interval for variable v. So, in other words, there is no i prime and there is no j prime, such that v is live here and that cannot happen. This is the first definition and (Refer Slide Time: 44:50) this is the last use of j and that is what I am trying to say. Let us go through it again (Refer Slide Time: 44:55) and so live interval for variable v. If there is no instruction with number j prime, j greater than j, such that v is live at j prime. So, there is no j prime, such that v is live at j prime and there is no i prime, such that v is live at i prime. So, this cannot happen and v is not live and i prime does not exist, j prime does not exist.

So, this is the first instruction, where you know the liveliness of the variable v begins. This is the last instruction, where liveliness of j ends. So, in between, there could be loops and if then else conditions, there could be any other code also. We are not bothered about the exact live range in the case of graph colouring. Here, we are going to just consider the number of i, number of j, from i to j and all the instructions throughout will be considered as the live interval.

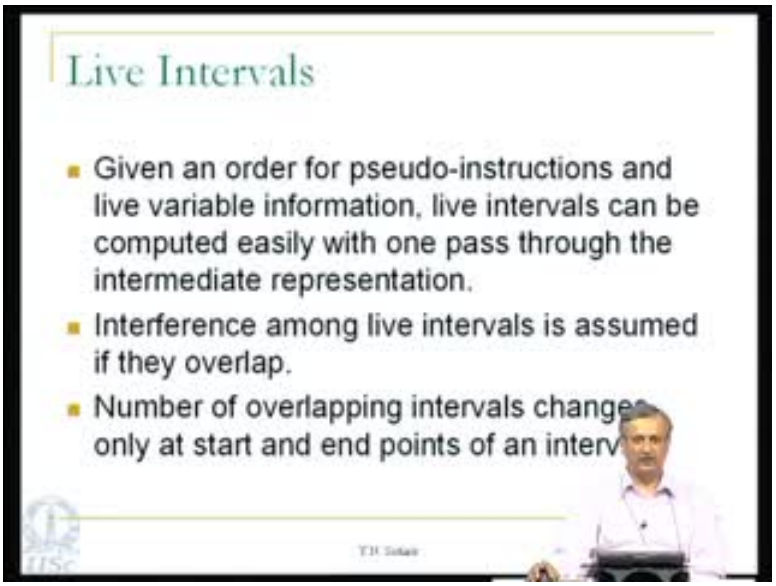
Why is this so easy to find? We do not have to do any serious kind of data flow analysis needed for live variable analysis. We just scan the instructions in some order and we note the first definition of a variable. Go on looking at the instructions and look at the last usage of that particular variable, whatever is in between is the live interval.

(Refer Slide Time: 46:22)



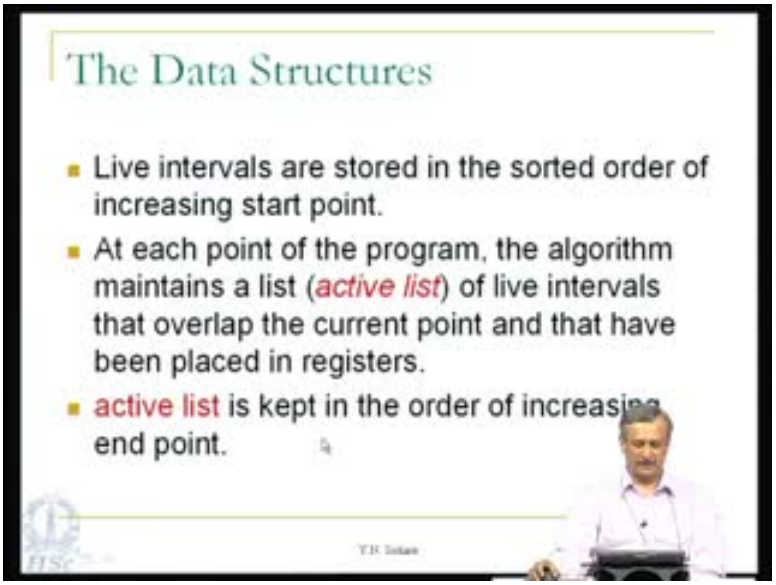
For the previous example, you know from this point, this (Refer Slide Time: 46:30) is a linearly ordered code. A equal to B equal to equal to A equal to B even though A is not live here, strictly it is not used here and B is not used here and we ignore all that. From this point to this point, this entire thing is considered as the live interval for A. So, whatever is the live interval for B will be here. So that is how they are. Whereas, we had considered (Refer Slide Time: 46:56) this this and this as the live range of A this this and this as the live range of B, whereas, now everything are linearly ordered code and this entire thing including the definition of B. Since, this A equal to comes earlier is considered as the live interval for A and so it is much easier to find.

(Refer Slide Time: 47:15)



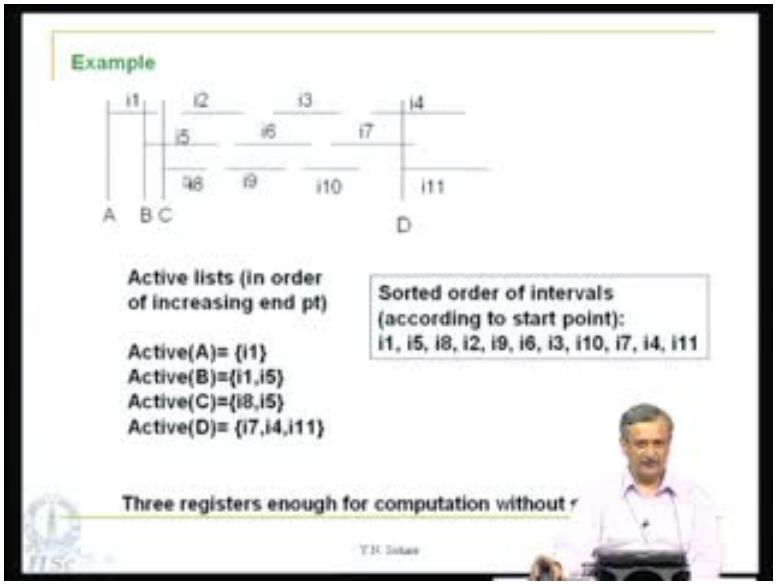
Given an order for pseudo instructions and live variable information, live intervals can be computed very easily with 1 pass through the intermediate representation. This is what I explained just now. Interference among live intervals is assumed if they overlap like in the previous picture A and B, live intervals of A and B overlap and they interfere. The number of overlapping intervals changes only at the start and end points of an interval. It is very obvious because we consider the first definition and the last use. so, there is nothing before or after that particular point other than the interval itself. So, the number of overlapping intervals obviously changes only at the starting and ending points of the interval.

(Refer Slide Time: 48:02)



Now, the Data Structures: which are used for this particular register location. Live intervals are stored in the sorted order of increasing start point. So, this is important and sorted order of increasing the start point. At the end of each point of the program, the algorithm maintains a list called the active list of live intervals that overlap the current point. I will show you a simple example very soon that has been placed in registers. So, we have given registers to some of the live intervals. So, these are all the live intervals, which overlap that particular point and they have been given different registers. So, they are all kept in the active list in the order of increasing end point.

(Refer Slide Time: 49:04)



So, let us look at an example. We have many live intervals i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11. The sorted order of these intervals are according to start point. So, i1 begins here, it is the first one, i5 begins and this is the second one, i8 begins and this is the third one, i4 begins and it is the fourth etc. Finally, the last one to begin is i11. So that is the last one. This is the sorted order for these intervals. What are active lists here? For example, if you consider this point, there is only i1, which is active. So, active list at the point A has only i1. Suppose, you consider point B and this is the starting point of i5, i1 overlaps with i5.

You can see very clearly, we cannot actually give i1 and i5 to the same register. These are active at the same time and so the active list at B contains i1 and i5. Let us look at C and at C, i8 has begun and i5 is still running. It is live, but i1 has finished. So, the active list will correctly contain i8 and i5. So, these are the ones which will be given registers. Let us look at D, we have i7 and i4, which are yet to finish and i11, which has just begun to start. So, these are the three, which will be live. What is important here? It is kept in increasing order of the end point and that is why for this last one, i7 finishes first, i4 finishes next and i11 finishes last. So that is how this is ordered.

There is also a statement here, which says 3 registers are enough for the computation without spills. So, let me explain the allocation algorithm informally with this example. Now, we start with the point A, which is the beginning of i1. There is only i1, which is active and so we give it a register. There are 3 registers available, then the next point we need to consider is only B because this is the starting point of the next. We are considering in this particular order- y1, i5. So, at i5 in at point B, i1 and i5 are both live.

They are active and so we can give i5 also another register, then it is put into active B. So, we do not put it into active list, unless we give it a register. If it is not given a register for some reason, we will see later and we do not put it in the active list.

Then the third one is i8, third interval is i8 and that is point C. So, at this point, we see that in the active list of B, i1 has finished. So, it need not hold up any register. The register given to i1 can be freed now. Only i5 is holding a register and there are two more registers free. So, we can give 1 register very liberally to i8 and put that in the active list. So. at C you put that into active list.

Let us continue with the next point in i2. Here, in the beginning of i2, you see here (Refer Slide Time: 53:09) for these two, we gave both of them registers. So, i5 and i8 both will be in the active list at this point, but you still have a third register free. No problem, give a register to i2 and therefore all three will go into the active list at this point. The next point after i2 is i9. So, at i9, i5 has finished and so we can remove it from the active list. i2 is still active and introduce i9 into the active list. So, give it another register, then we go to i6. So, at i6, both this i2 and i9 are still active. i6 also becomes active, but we have the third register free. So, we can give it one more register.

We go to i3. So, at the point i3, i2 has finished. So, it can be removed from the active list and its register can be freed. So, two registers are given to i6 and i9. Third one is freed and now we can give it to i3, then we take up i10. The same situation in i9 has finished. i6 and i3 are active and third register is free. So, we can give it to i10, then we go to i7. Again, you know i6 has finished i3 and i10. So, we can give the free register to i7, then at i4, i10 has finished. i7 is still active and so we have used only one register. i10's register and i3's register are also finished.

They have been released and so we can give a register to i4, then we go to i11. Then all the three are active, but the third i10 has finished, i3 has also finished. So, we have one register free. We can give it to i11 this way. The entire register location is possible without any spills with three registers.

So, let us just look at the scenario and we will consider the detailed algorithm later. So, for example, we give one register here and this point we give a register to i5 and then at this point, we can give a register to this. i1 is raised and we can give it a register, but once we come to i2, both registers are occupied. So, i2 does not get a free register. Now, the decision has to be made as to which one of these will remain in memory and that is the spilling decision. So, we will look at the spilling decision and the algorithm for spilling etc in detail in the next lecture. This is the end of the lecture. Thank you.