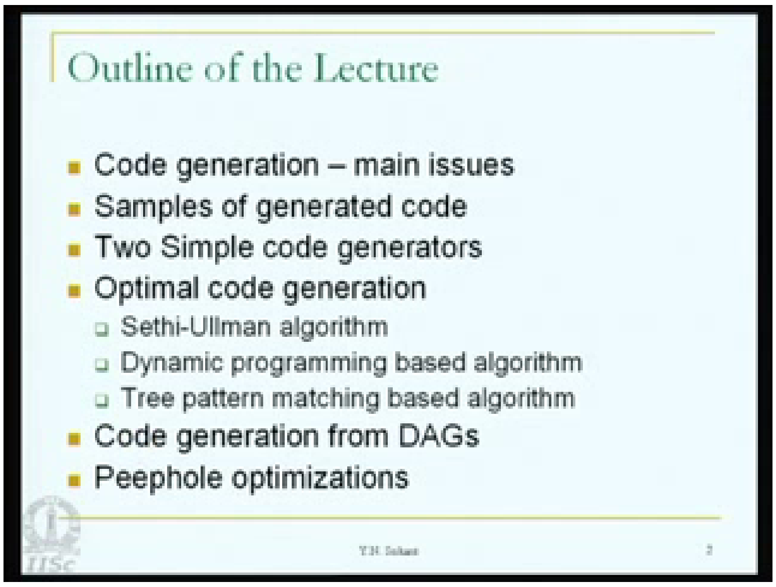
Compiler Design

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

Module No. # 04
Lecture No. # 06
Code Generation

(Refer Slide Time: 00:28)



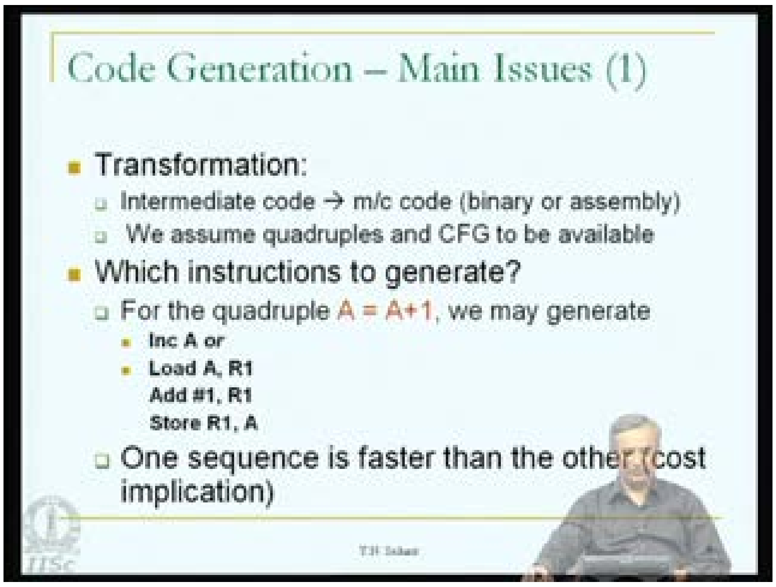
Outline of the Lecture

- Code generation – main issues
- Samples of generated code
- Two Simple code generators
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

Y.N. Srikant 2

Welcome to the lecture on code generation. We covered a little bit of this material in the last class.

(Refer Slide Time: 00:28)

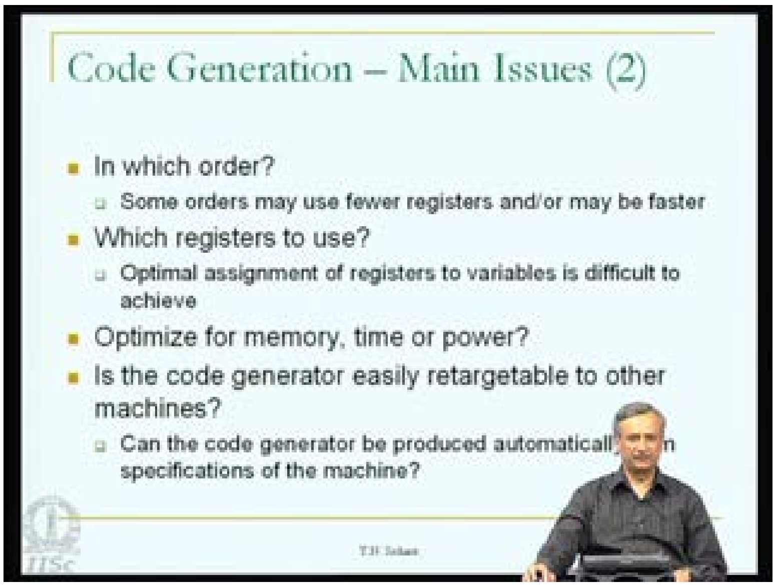


Code Generation – Main Issues (1)

- Transformation:
  - Intermediate code → m/c code (binary or assembly)
  - We assume quadruples and CFG to be available
- Which instructions to generate?
  - For the quadruple A = A+1, we may generate
    - Inc A or
    - Load A, R1
      Add #1, R1
      Store R1, A
  - One sequence is faster than the other (cost implication)

T.H. Srikant

(Refer Slide Time: 00:40)



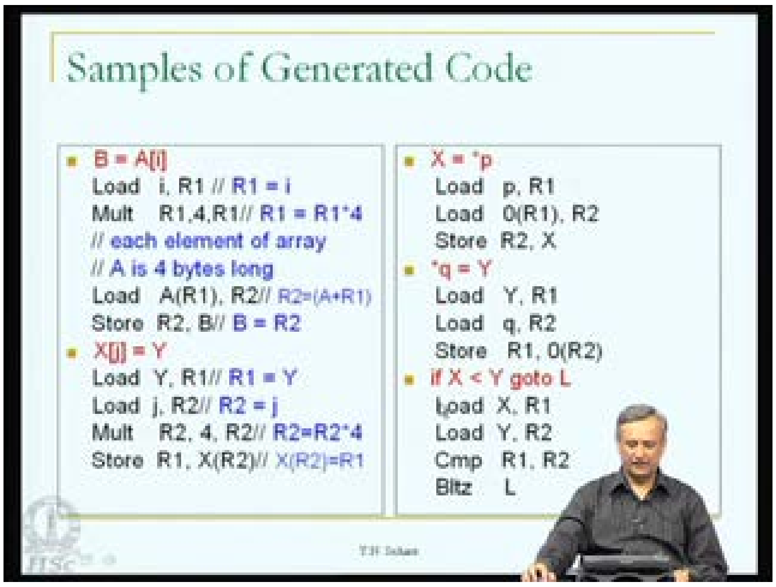
Code Generation – Main Issues (2)

- In which order?
  - Some orders may use fewer registers and/or may be faster
- Which registers to use?
  - Optimal assignment of registers to variables is difficult to achieve
- Optimize for memory, time or power?
- Is the code generator easily retargetable to other machines?
  - Can the code generator be produced automatically from specifications of the machine?

T.H. Srikant

We looked at the main issues involved in code generation - for example, transformation, what exactly is code generation, which instructions to generate, what are the consequences of that, in which order should we generate the code, which registers to use, should we optimize for memory time or power, is the code generator retargetable to other machines etcetera; these are some of the issues that we looked at.

(Refer Slide Time: 00:55)



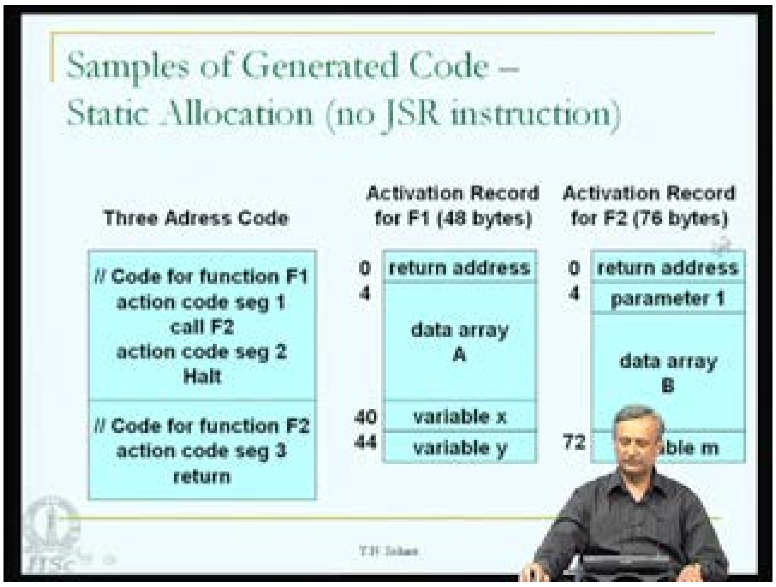
Samples of Generated Code

```
■ B = A[i]
  Load   i, R1 // R1 = i
  Mult   R1,4,R1// R1 = R1*4
  // each element of array
  // A is 4 bytes long
  Load   A(R1), R2// R2=(A+R1)
  Store  R2, B// B = R2
■ X[j] = Y
  Load  Y, R1// R1 = Y
  Load  j, R2// R2 = j
  Mult   R2, 4, R2// R2=R2*4
  Store  R1, X(R2)// X(R2)=R1
```

```
■ X = *p
  Load   p, R1
  Load   0(R1), R2
  Store  R2, X
■ *q = Y
  Load   Y, R1
  Load   q, R2
  Store   R1, 0(R2)
■ if X < Y goto L
  Load  X, R1
  Load  Y, R2
  Cmp   R1, R2
  Bltz    L
```

We also saw some samples of generated code. For example, for the assignment B equal to A i, X j equal to Y, X equal to star p, star q equal to Y, then branch instruction, etcetera.

(Refer Slide Time: 01:17)



Samples of Generated Code – Static Allocation (no JSR instruction)

Three Adress Code

// Code for function F1 action code seg 1 call F2 action code seg 2 Halt
// Code for function F2 action code seg 3 return

Activation Record for F1 (48 bytes)

0	return address
4	data array A
40	variable x
44	variable y

Activation Record for F2 (76 bytes)

0	return address
4	parameter 1
	data array B
72	[illegible]ble m

We will continue that discussion today. Let us look at the generated code for function calls etcetera - first of all, in the case where we have static allocation and then we look at the case where there is dynamic allocation.

Even with static allocation, let us assume that there is no jump subroutine instruction, to begin with and then we will see the minor modification needed, when there is a jump subroutine instruction available.

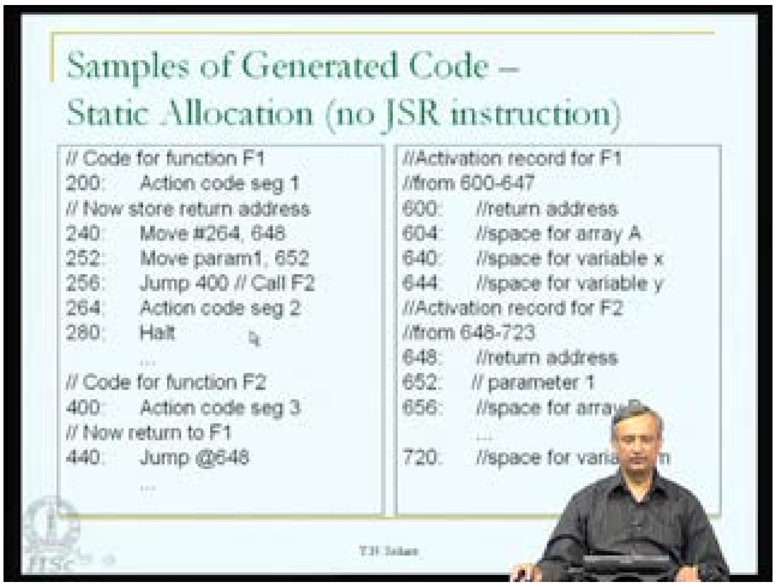
The 3 address code for the function F1 and function F2 will probably look like this. Let us assume that function F1 is the main program itself and function F2 is the subroutine. There is action code segment 1 and then there is a call to the function F2 and then action code segment 2 and then there is halt. These action code segments are some code computations within the function F1 to achieve the meaningful line and result.

In code for function F2 we just have action code segment 3 and then there is a return. Let us first look at the activation record for the function F1; its size is 48 bytes. To begin with, the activation record will have, for example, the static link, dynamic link information, which we already have discussed. So, I am omitting the static link, dynamic link information in this particular activation record to make the presentation simpler. There is a return address field; then there are local parameters. Function F1 does not have any parameters within local variable as I said sorry not local parameters. Function F1 does not have parameters passed to it whereas function F2 has a parameter passed to it.

There are local variables, for example, data array A, then there is a variable x, another variable y. The total size of this activation record is 48 bytes and since this is static allocation, the activation record is allocated just once and there is no recursion possible.

Activation record for the function F2 is 76 bytes in size. To begin with, there is return address then there is space for parameter 1, then there is an array B and another variable m.

(Refer Slide Time: 04:14)



Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move param1, 652
256:    Jump 400 // Call F2
264:    Action code seg 2
280:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1
440:    Jump @648
        ...
```

```
//Activation record for F1
//from 600-647
600:    //return address
604:    //space for array A
640:    //space for variable x
644:    //space for variable y
//Activation record for F2
//from 648-723
648:    //return address
652:    // parameter 1
656:    //space for array B
        ...
720:    //space for varia[illegible]m
```

Y.N. Srikant

What would be the code that is generated for such a set of functions? Here is a code for function F1; then we have code for function F2; and finally, we have the activation record format for F1 and the activation record format for F2. In the activation record, let us look at this first, for F1 it runs from address 600 to address 647; and then the first location, as we saw previously, contains the return address. Then the second location onwards, there is a space for the local variable A, which is an array, then at 640 we have the space for variable x and at 644 we have the variable y.

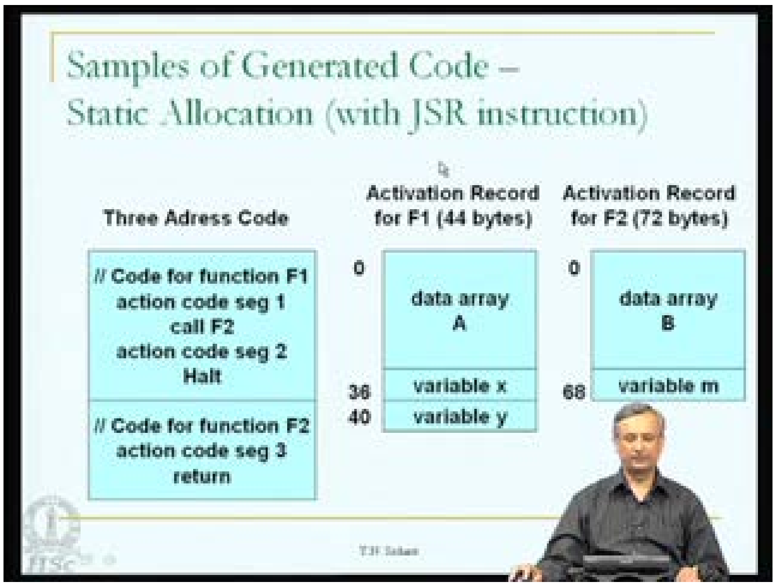
The activation record for F2 runs from 648 to 723, at address 648 we have the return address stored in it; at 652 we have the parameter 1; and then at 656 onwards, we have the array B; and at 720, we have the variable m. So, this is the layout of activation records of F1 and F2.

Codes for function F1 starts at 200 and code for function F2 starts at 400. Action code segment 1 starts at 200 and it runs till 240 - some computation which is meaningful. Now at the address 240, we have a move instruction - Move hash 264 to address 648. This instruction achieves storing the return address onto this particular slot.

The reason we are doing it is because there is no jump subroutine instruction. We need to store the return address explicitly and then jump to it, once the function F2 is over. There is a parameter to the call F2. So, at 252 we have Move param 1, 652 which moves some parameter - variable, let us say, 1 to 652 - and then at 256 we have Jump 400, which is actually a jump to the address 400. Note that this hash 264 is nothing but the address of action code segment 2; 264 is the address of this; so, that is the return address, which is being stored at 648. At location 400, which corresponds to function F2, we have action code segment 3; and then the code is actually till 440; and then jump at 648 is an indirect jump instruction, which takes the contents of 648 and treats that as an address and jumps to that particular address.

648 correctly contains 264 and therefore, once the jump is completed the action code segment 2 continues and then finally, function F1 halts. So, this is a way the code would be for function call with a parameter and no jumps subroutine instruction.

(Refer Slide Time: 07:47)



Samples of Generated Code – Static Allocation (with JSR instruction)

Three Adress Code

```
// Code for function F1
action code seg 1
call F2
action code seg 2
Halt

// Code for function F2
action code seg 3
return
```

Activation Record for F1 (44 bytes)

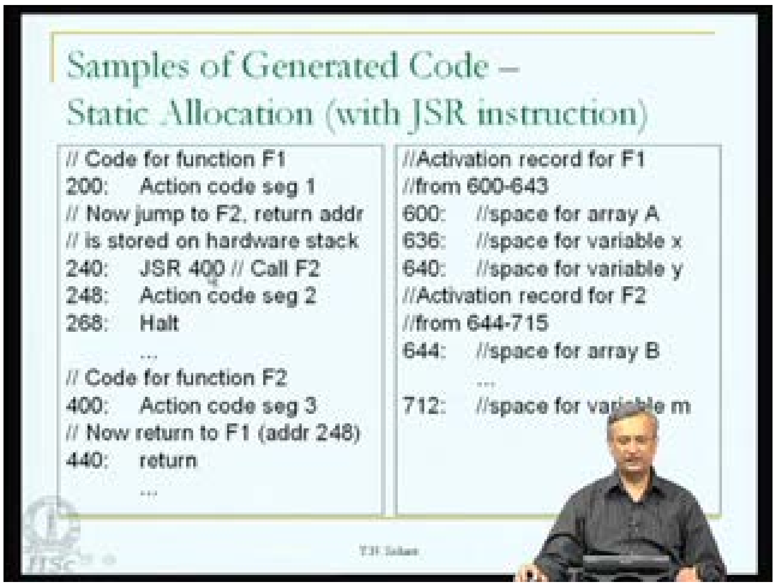
0	data array A
36	variable x
40	variable y

Activation Record for F2 (72 bytes)

0	data array B
68	variable m

Here is the slight change in the activation record format when we have a jump subroutine instruction. For simplicity, since we have already studied how to pass parameters, I have omitted the parameter here as well. Please observe that there is no return address here and here. We do not have that; we had it in the previous activation record format. Rest of it remains the same. The size of the activation record has slightly reduced. Let us see what happens in the code.

(Refer Slide Time: 08:31)



Samples of Generated Code – Static Allocation (with JSR instruction)

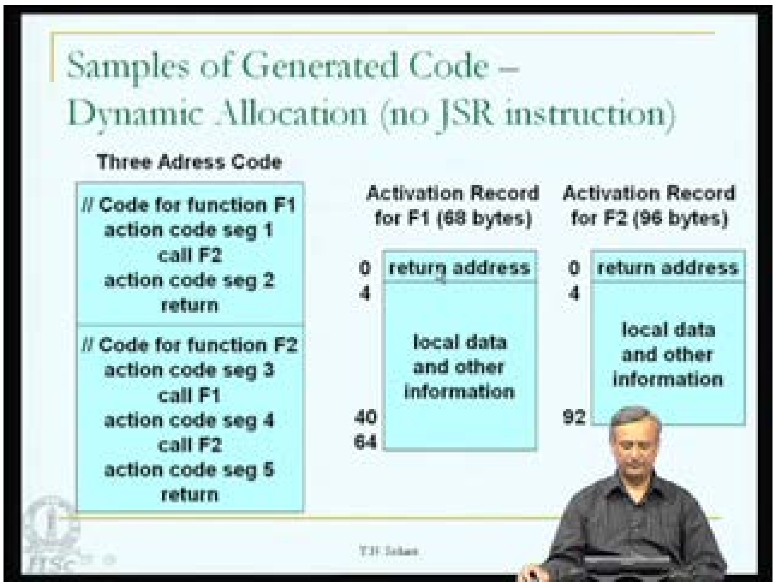
```
// Code for function F1
200:    Action code seg 1
// Now jump to F2, return addr
// is stored on hardware stack
240:    JSR 400 // Call F2
248:    Action code seg 2
268:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1 (addr 248)
440:    return
        ...
```

```
//Activation record for F1
//from 600-643
600:    //space for array A
636:    //space for variable x
640:    //space for variable y
//Activation record for F2
//from 644-715
644:    //space for array B
        ...
712:    //space for variable m
```

The activation record for F1 now runs from 600 to 643. We do not have the return address otherwise everything else remains the same. We do not have the parameter either in this particular case. So, there is no space for the parameter either. Here we execute the action code segment 1 in function F1. Now, there is a jump subroutine instruction. The jump subroutine instruction uses a separate hardware stack to store the return address.

There is no need to store the return address explicitly. JSR 400 automatically stores the return address on its stack, hardware stack and jumps to the address 400. Here we execute action code segment 3 in function F2 and return instruction automatically takes the return address from the hardware stack and returns to the address 248, which is the return address; then it continues. So, this is the minor difference we would have when there is a jump subroutine instruction.

(Refer Slide Time: 09:38)


Samples of Generated Code – Dynamic Allocation (no JSR instruction)

Three Adress Code

```
// Code for function F1
action code seg 1
call F2
action code seg 2
return
// Code for function F2
action code seg 3
call F1
action code seg 4
call F2
action code seg 5
return
```

Activation Record for F1 (68 bytes)

0	return address
4	local data and other information
40	
64	

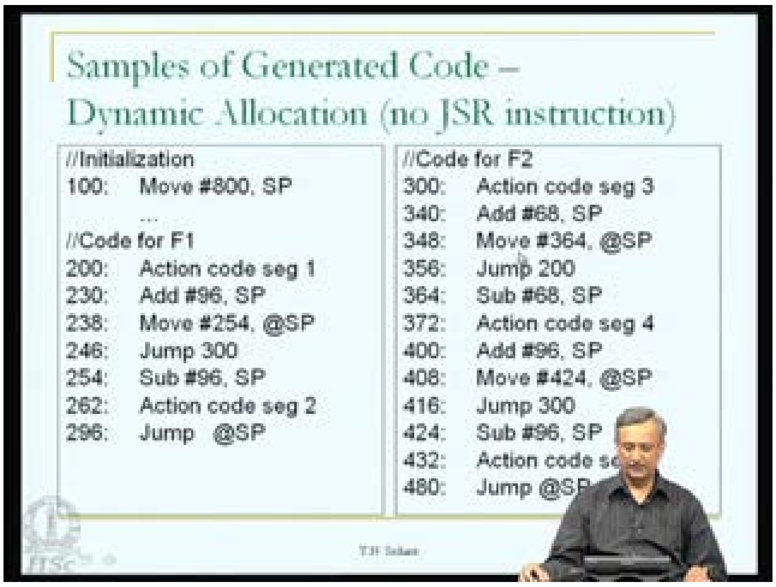
Activation Record for F2 (96 bytes)

0	return address
4	local data and other information
92	

Let us see how the code changes when we have dynamic allocation and again with no JSR. In this case again, we have omitted the parameter because there is not too much difference when we have parameters; we have seen what happens already, but here the allocation is dynamic so, we can have recursion. Function F1 has action code segment 1 and then a call to the function F2, action code segment 2 and then let us says a return. We have called this from the main program. Code for function F2 contains action code segment 3, then a recursive call to function F1, see call F2, F2 call to F1 again, action code segment 4 and then a recursive call to F2 itself again and action code segment 5, finally return.

Let us see here, there is a return address and then local data and other information, return address and local data and information. Please observe that the size of the activation record is 68 bytes here and 96 bytes here. There is no jump subroutine instruction.

(Refer Slide Time: 10:46)



Samples of Generated Code – Dynamic Allocation (no JSR instruction)

```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #96, SP
238:    Move #254, @SP
246:    Jump 300
254:    Sub #96, SP
262:    Action code seg 2
296:    Jump  @SP
```

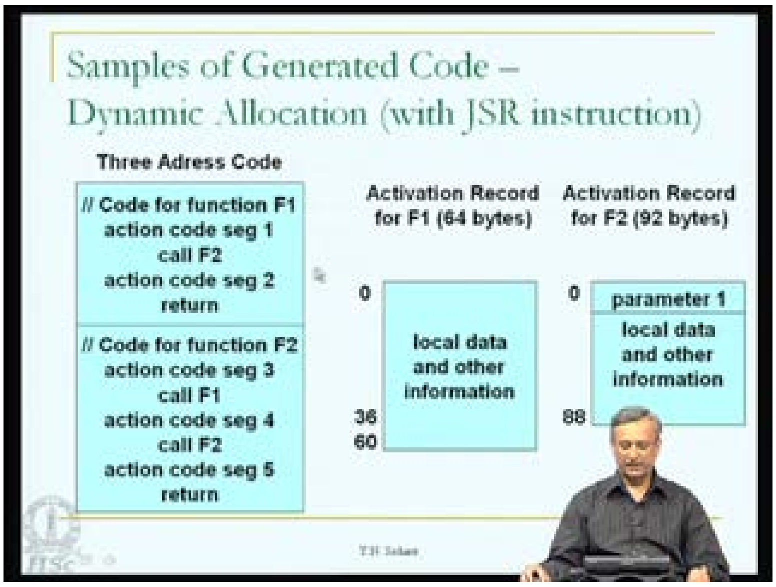
```
//Code for F2
300:    Action code seg 3
340:    Add #68, SP
348:    Move #364, @SP
356:    Jump 200
364:    Sub #68, SP
372:    Action code seg 4
400:    Add #96, SP
408:    Move #424, @SP
416:    Jump 300
424:    Sub #96, SP
432:    Action code s
480:    Jump @SP
```

There is some initialization for the stack pointer saying that from 800 onwards the stack begins and then let us look at the code for the function F1. There is action code segment 1 from 200 to 230 and then we have to create the activation record. We are assuming that the activation record is created on the stack itself; so, it is a stack managed activation record and not a heap managed activation record. 96 - The size of the activation record is added to SP, thereby allocating the space for this particular activation record for the call to the function - function F2, that is.

Then we move the return address which is 254 at SP indirectly. The first location of the activation record now contains a return address, after this instruction is executed; then jump 300; we come here; this is the code for F2. Within this, we execute action code segment 3. Now, there is a recursive call to function F1. We add the size of the activation record to SP thereby creating another activation record for the function call F1 and then the return address is moved to the first location of this particular activation record and finally, we jump to 200. (Refer Slide Time: 12:37) So, we come here. This executes again and again; it keeps going. Finally, once we return from the function, let us see how that is done, we subtract 68 from the stack pointer there by the activation record is destroyed. Action code 4 is executed. Now, there is another recursive call. We add 96 to SP, return address is stored and jump to 300. This is another recursive call; once that recursive call is completed, we come out and we subtract the size of the activation record thereby destroying it and execute the action code segment 5. Jump at SP automatically returns by taking the return address from the first location of the activation record on the stack.

(Refer Slide Time: 13:33) The same thing happens here. Once we return here, we come here. Subtract 96 which is the size of the activation record, execute the action code segment 2 and jump at SP takes the return address and returns to the main program.

(Refer Slide Time: 13:50)



Samples of Generated Code – Dynamic Allocation (with JSR instruction)

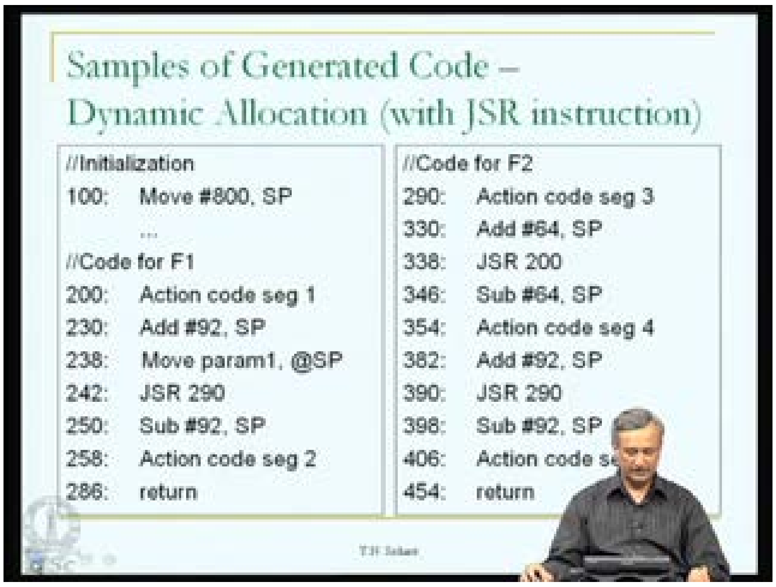
Three Adress Code

```
// Code for function F1
action code seg 1
call F2
action code seg 2
return
// Code for function F2
action code seg 3
call F1
action code seg 4
call F2
action code seg 5
return
```

Activation Record for F1 (64 bytes)
0
local data and other information
36
60
Activation Record for F2 (92 bytes)
0
parameter 1
local data and other information
88
Y.N. Srikant

With jump subroutine instruction, the only change is there is no return address here; so, activation records are smaller.

(Refer Slide Time: 13:58)



Samples of Generated Code – Dynamic Allocation (with JSR instruction)

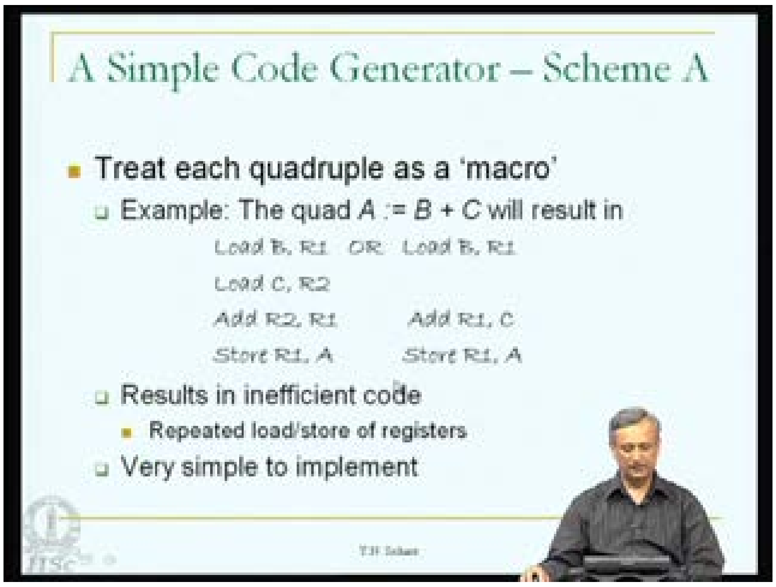
```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #92, SP
238:    Move param1, @SP
242:    JSR 290
250:    Sub #92, SP
258:    Action code seg 2
286:    return
```

```
//Code for F2
290:    Action code seg 3
330:    Add #64, SP
338:    JSR 200
346:    Sub #64, SP
354:    Action code seg 4
382:    Add #92, SP
390:    JSR 290
398:    Sub #92, SP
406:    Action code se
454:    return
```

The code is also much simpler; the initialization remains and the activation record is created. There is a parameter which is permitted here along with jump subroutine. We move the parameter to that particular location and execute a JSR which stores the return address on the hardware stack. Then once it is completed, we come out; deduct the size of the activation records. This destroys the activation record; execute the code segment 2 and return. This is very similar; there is not much change; every time we add the size of the activation record and jump subroutine.

Once we come out of the subroutine, we subtract the size of the activation record, thereby we destroy the activation record; this is a fairly straight forward job.

(Refer Slide Time: 15:00)



A Simple Code Generator – Scheme A

- Treat each quadruple as a 'macro'
  - Example: The quad *A* := *B* + *C* will result in

```
Load B, R1    OR   Load B, R1
Load C, R2
Add R2, R1         Add R1, C
Store R1, A        Store R1, A
```

  - Results in inefficient code
    - Repeated load/store of registers
  - Very simple to implement

Y.N. Srikant

You have seen many examples of what type of code is generated for function calls and other statements. Now, it is time to discuss how exactly code is generated. What this says is, treat each quadruple as a macro; this is the first thing.

An example is here. Let us say the quadruple is A equal to B plus C. We generate extremely simple code. We load this B to R1. (Refer Slide Time: 15:40) This is one possible code sequence; this is the other possible code sequence. Load C to the register R2, add R2 to R1 and finally, store R1 to A.

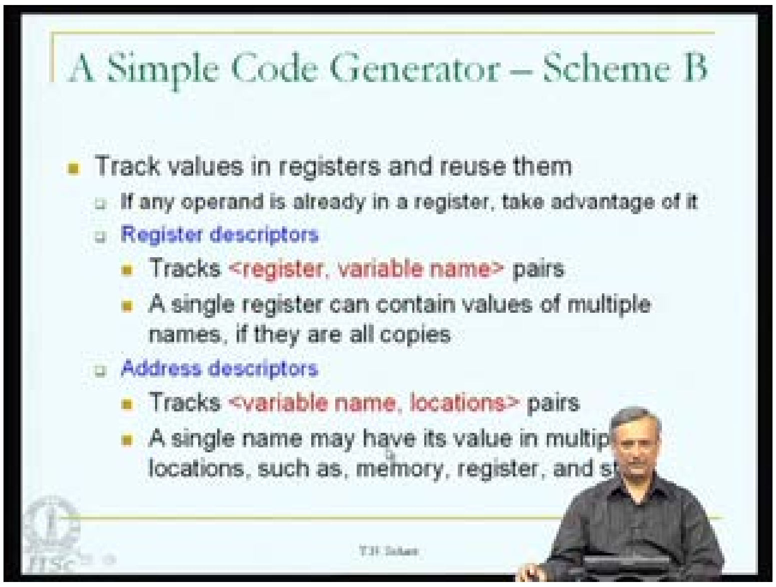
Otherwise, we say load B to R1; and this should have been add C comma R1 and then finally, store R1 to A. This is add C comma R1.

The purpose for such a macro is to make the code generation macros self-contained - complete. Observe that the registers which are used for storing B and C here will be released at the end of this particular macro. They are not going to contain any meaningful values later on, that is the assumption.

If we do this load - store sequence for every quadruple, automatically the code will be very inefficient, but it is very easy to implement. For each type of instruction, we write a macro and expand that macro for the sequence of quadruples. That is all. Nothing more than that, is needed.

So, repeated load - store of registers takes place here. For example, we are storing R1 in A and we are not using the value of A, which is in R1 later on. We load A into R1 all over again, even if the next instruction is like x equal to A plus y. This is very simple to implement, but definitely there is a lot more chance to improve it.

(Refer Slide Time: 17:31)



A Simple Code Generator – Scheme B

- Track values in registers and reuse them
  - If any operand is already in a register, take advantage of it
  - Register descriptors
    - Tracks <register, variable name> pairs
    - A single register can contain values of multiple names, if they are all copies
  - Address descriptors
    - Tracks <variable name, locations> pairs
    - A single name may have its value in multip locations, such as, memory, register, and s

Scheme B: Scheme B is slightly more complicated. It tries to track values which are available in registers and then it tries to reuse them. If any operand is already in a register then take advantage of it.

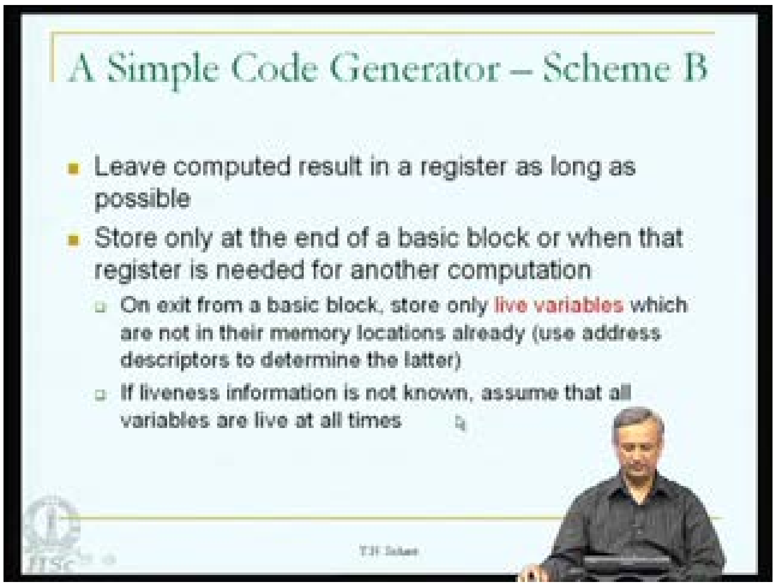
That is the basic idea, but how to take advantage of it. It is done using 2 data structures called register descriptors and address descriptors - two of them. The register descriptor tracks register variable name pairs.

In other words, for each register, it tells you which are the variables which correspond to that value in the register, but then the question will be how a register can hold the values of many names.

This is possible because a single register can contain multiple names, if they are all copies of each other. Let us say we have computed A equal to x plus 5; the value of A is now available in some register R1.

Then we write B equal to A, or C equal to A, or M equal to A and so on. So, B, C and M, all these variables, now contain the same value of A, but we do not have to allocate different registers for them. We just track that they are all corresponding to the same value using this register descriptor. What is an address descriptor? It tracks variable name and location pairs. For each variable name, it stores the locations where that value is present. For example, a single name may have its value in multiple locations such as memory, register and say, if it is stack machine, then on the stack also. In such a case depending on the occasion we may actually want to avoid storing rather loading a particular value from memory to register, if it is available both in memory and register. We will see this a little latter.

(Refer Slide Time: 19:50)



A Simple Code Generator – Scheme B

- Leave computed result in a register as long as possible
- Store only at the end of a basic block or when that register is needed for another computation
  - On exit from a basic block, store only live variables which are not in their memory locations already (use address descriptors to determine the latter)
  - If liveness information is not known, assume that all variables are live at all times

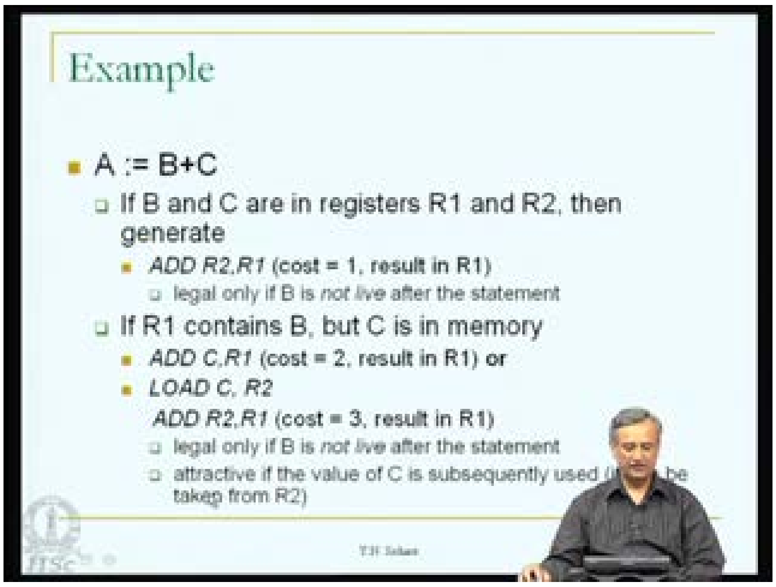
Y.N. Srikant

And then the computed result is left in a register as long as possible. Unless the register is useful for some other work, we are not going to vacate the register thereby we probably avoid repeated loads. We store only at the end of a basic block unless we are compelled to do otherwise or when that register is needed for another computation.

On exit from a basic block, store only live variables which are not in their memory locations already. This as I said can be found out very easily using the address descriptor. Live variables are those which will be useful even after the basic block is completed.

In the next basic block, programmer defined variables are normally live, even after the basic block is completed, whereas temporary variables have the scope within the basic block. If liveness information is, for example, not available at all what do we do? We cannot stop code generation. We assume that all the variables are live at all times thereby, at the end of a basic block we need to store every variable into their home memory location.

(Refer Slide Time: 21:18)



Example

- A := B+C
  - If B and C are in registers R1 and R2, then generate
    - *ADD R2,R1* (cost = 1, result in R1)
      - legal only if B is *not live* after the statement
  - If R1 contains B, but C is in memory
    - *ADD C,R1* (cost = 2, result in R1) or
    - *LOAD C, R2*
      ADD R2,R1 (cost = 3, result in R1)
      - legal only if B is *not live* after the statement
      - attractive if the value of C is subsequently used (it can be taken from R2)

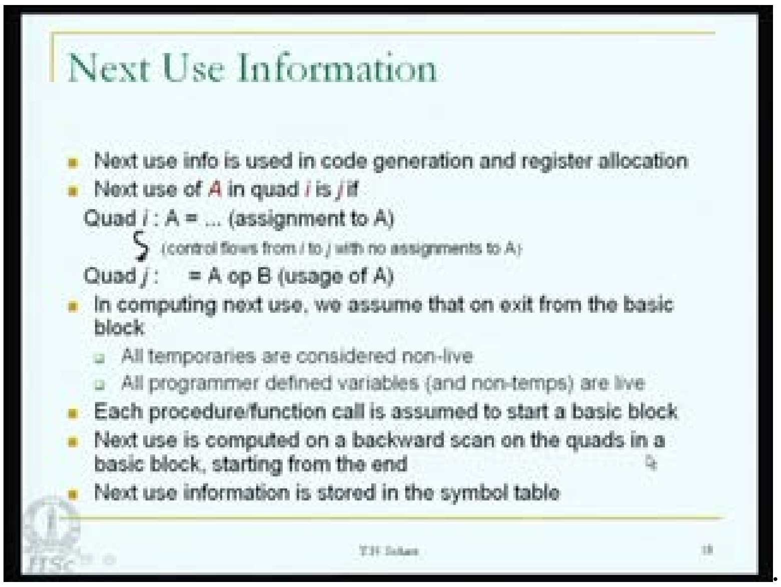
Let us look at an example and understand how this scheme B works and then go on to the scheme itself.

Say, the quadruple is A equal to B plus C. If B and C are in registers R1 and R2, then generate add R2 comma R1; the cost of this is 1 and the result is in R1, but this is legal only if B is not live after the statement because we are destroying B. B is in R1 and after this instruction is executed B will not be in R1.

It is the value B plus C - that is, A - which will be in R1. This is one possible alternative. The other alternative is, if R1 contains B, but C is in memory, then generate add C comma R1; cost is 2 because you need to fetch from the memory location, C also; and the result is in R1.

One more possibility is we load C into R2 and then add R2 to R1. (Refer Slide Time: 22:29) Cost for this is 3 because you are loading this also into R2 and then adding it; the result is in R1. This is legal only if B is not live after the statement. Again the same issue; both these are the same. B is destroyed and it is attractive if the value of C is subsequently used. In both cases, if we are going to use C later on then we can take it from R2 and we do not have to load it into R2 all over again.

(Refer Slide Time: 23:02)



Next Use Information

- Next use info is used in code generation and register allocation
- Next use of *A* in quad *i* is *j* if

  Quad *i* : A = ... (assignment to A)

  (control flows from *i* to *j* with no assignments to A)

  Quad *j* : = A op B (usage of A)
- In computing next use, we assume that on exit from the basic block
  - All temporaries are considered non-live
  - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table

Y.N. Srikant 18

That is how the many options available during code generation regarding availability or otherwise of a variable in memory or in register is tackled. To do that and to usefully retain only those values which are used later in registers, we use some information called next use information.

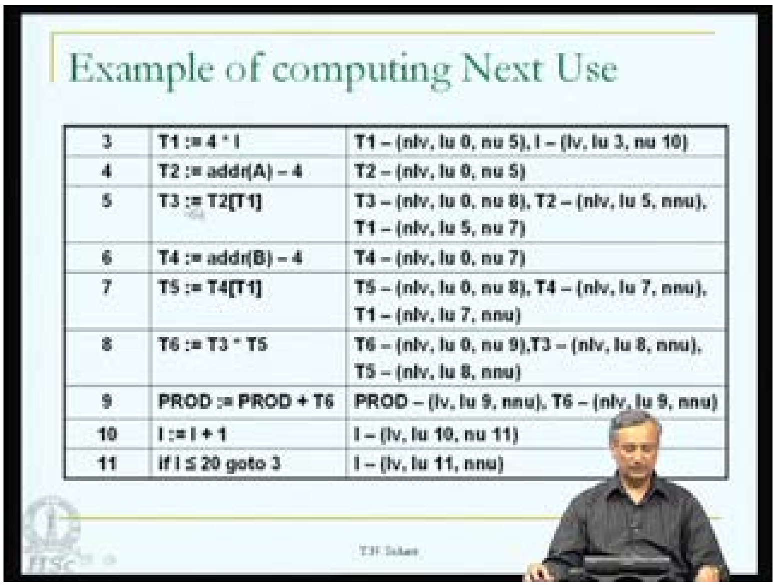
It is used in code generation and also register allocation in scheme B. What is next use? For example, consider quadruple i and quadruple j; the control flows from i to j with no assignments to the variable A, which is defined here.

There is an assignment to A and then there is usage of A; in between there are no other assignments to A. So, the value of A here, is valid here also. (Refer Slide Time: 24:01) The next use of this A in quad i is j, if this is true. In computing the next use, we assume that on exit from the basic block all temporaries are considered non-live, so they are not useful anymore.

All programmer defined variables and non-temporaries are live. This is the assumption that we make. If this is not true then we will have to assume that all temporaries are, all variables are non-live.

Each procedure function call is assumed to start a new basic block. There are no interactions from the procedure call inside the basic block itself. It becomes a separate basic block and it can be handled in a much better fashion. Next use is computed on a backward scan on the quadruples in a basic block, starting from the end. We will see an example of this now and next use information is stored in the symbol table itself.

(Refer Slide Time: 25:05)



Example of computing Next Use

3	T1 := 4 * I	T1 – (nlv, lu 0, nu 5), I – (lv, lu 3, nu 10)
4	T2 := addr(A) – 4	T2 – (nlv, lu 0, nu 5)
5	T3 := T2[T1]	T3 – (nlv, lu 0, nu 8), T2 – (nlv, lu 5, nnu), T1 – (nlv, lu 5, nu 7)
6	T4 := addr(B) – 4	T4 – (nlv, lu 0, nu 7)
7	T5 := T4[T1]	T5 – (nlv, lu 0, nu 8), T4 – (nlv, lu 7, nnu), T1 – (nlv, lu 7, nnu)
8	T6 := T3 * T5	T6 – (nlv, lu 0, nu 9),T3 – (nlv, lu 8, nnu), T5 – (nlv, lu 8, nnu)
9	PROD := PROD + T6	PROD – (lv, lu 9, nnu), T6 – (nlv, lu 9, nnu)
10	I := I + 1	I – (lv, lu 10, nu 11)
11	if I ≤ 20 goto 3	I – (lv, lu 11, nnu)

Y.N. Srikant

Here is an example of how to compute next use. Here nlv means not live, lv means live, lu means last use. nu means next use. Let us start from the bottom and see how the algorithm computes next use information. This is the last quadruple of the basic block.

So, there is no more usage of the variable I or any other variable for that matter after this basic block. For I, it is a programmer defined variable. So, it is recorded as live and then the current quadruple number in which it is used is recorded as last use - that is eleven. There is no next use because this is a last quadruple after which there are no usages of any variables. With this information, we go to quadruple of ten. Here is I is equal to I plus 1 again.

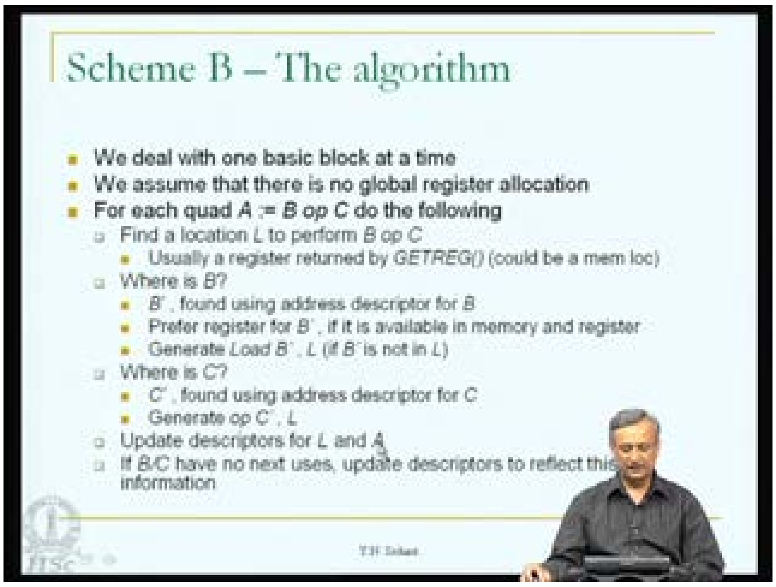
I is live always. Remember that I and PROD are programmer defined variable. So, they are always live. Last use of this I is 10 because I is used here and this is the quadruple. Then next use of I is in 11 because the value defined here is used in 11.

What we really have done is use this last use information to be recorded as next use information for the previous quadruple. That is how, the last use is used. PROD is a different variable; none of this holds now; it is always live. Its last use is nine, the current quadruple number and there is no next use because there is no usage of PROD after this and this was the initialization to the variable and it remains as it is. T6 is a temporary. So, it is always not live after the basic block. Its last use is 9 and there is no next use for it because it is not referred to anywhere here. As far as the initialization is concerned, it would have been initialized to nnu and continues to be nnu.

When you come to quadruple number 8, for T3 and T5, there is no next use because this is the first time that we have used it and the current quadruple numbers are recorded as 8 and 8. For T6, the last use is recorded as 0 indicating that this is a definition. A definition always terminates that live range. Even if there were to be any usages of T6 here, all those are not going to be the same T6. It is redefined here. So, it is apt to say that the there is no last use; that is last use is 0.

Once we go up to T5, this information about T5 which is already in the symbol table, for example, it had recorded not live, last use 8 and nnu. (Refer Slide Time: 28:22) This lu 8 is copied to this nu. For example, T5 becomes next use 8. So, whatever was here is copied here, lu is 0 because it is a definition and as far as T4 is concerned, it is not used again. We still do not have anything here which is useful. For T4 in quadruple number 6, we are copying lu 7 to nu and this continues. For T3 if for T2 T1 For example, T2 and T1 have nothing. Let us look at this T1 itself. We had T1 here which recorded nu as 7 and then the last use as 5; lu 5 here is copied to next use 5 here and I is always recorded as live. (Refer Slide Time: 29:21) Finally, the last use is 3 and nu is 10; from this quadruple we had recorded lu as 10 and now this is recorded as 10. So, in a backward scan you can compute this next use information using the last use information.

(Refer Slide Time: 29:39)



Scheme B – The algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad *A* := *B op C* do the following
  - Find a location *L* to perform *B op C*
    - Usually a register returned by *GETREG()* (could be a mem loc)
  - Where is *B*?
    - *B'* , found using address descriptor for *B*
    - Prefer register for *B'* , if it is available in memory and register
    - Generate *Load B'* , *L* (if *B'* is not in *L*)
  - Where is *C*?
    - *C'* , found using address descriptor for *C*
    - Generate *op C'* , *L*
  - Update descriptors for *L* and *A*
  - If *B/C* have no next uses, update descriptors to reflect this information

Y.N. Srikant

How do we use this? We now begin the discussion of the algorithm itself. This algorithm deals with one basic block at one time and we assume that there is no global register allocation in this; there is a local register allocation which is the GETREG function which we are going to discuss very soon.

For each quadruple A equal to B op C, we do the following. Find a location L to perform the computation B op C. It is not necessarily always a register, but most often a register is returned by GETREG.

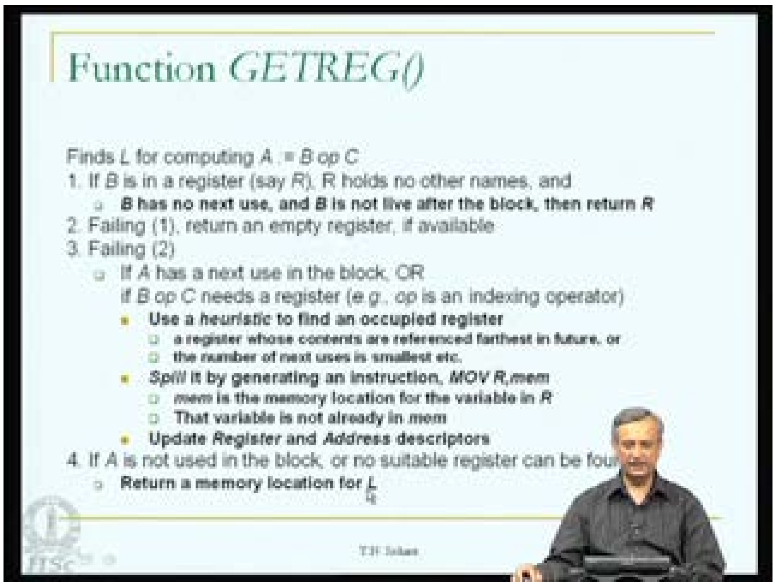
So this L is obtained via GETREG. We will see very soon how? It could be a memory location; normally it is not. Where is B? Let us say the location of the B prime B of B is B prime and it is found using the address descriptor for B that is what it is meant for.

Address descriptors tell you for each variable, where exactly it is located. If it is available in register and memory, let us prefer the register; say B prime is a register now.

If B prime and L are not the same, then we generate an instruction called load B prime comma L to move the value from B prime to L. If B prime and L are the same, there is no need for this particular instruction.

Where is C located? It is in C prime and again we find it using the address descriptor. Now generate the instruction op C prime comma L. Once all this is done we have to update the descriptors for L and A. Now, L contains the result and it is supposed to be contained in A. L is a register; we need to modify the register descriptor for L. Similarly, whatever A is, the address descriptor for A will have to be modified, mention where exactly its value is contained whether a register or a memory location etcetera. If B and C have no next uses then update the descriptor to reflect this information. This is one place where next use is used. That means from now onwards B and C need not be in the registers; their registers can be used for some other purpose that is what this really means.

(Refer Slide Time: 32:28)



Function *GETREG()*

Finds *L* for computing *A* := *B op C*

1. If *B* is in a register (say *R*), R holds no other names, and
   - **B has no next use, and B is not live after the block, then return R**
2. Failing (1), return an empty register, if available
3. Failing (2)
   - If *A* has a next use in the block, OR
     if *B op C* needs a register (*e.g., op* is an indexing operator)
     - **Use a *heuristic* to find an occupied register**
       - **a register whose contents are referenced farthest in future, or**
       - **the number of next uses is smallest etc.**
     - ***Spill* it by generating an instruction, *MOV R,mem***
       - ***mem* is the memory location for the variable in *R***
       - **That variable is not already in *mem***
     - **Update *Register* and *Address* descriptors**
4. If *A* is not used in the block, or no suitable register can be fou
   - **Return a memory location for *L***

Y.N. Srikant

What does the function GETREG do? It finds a location L for computing the B op C; in other words, computing A. There are 4 steps here. First one says if B is in a register, say R, and R holds no other name, the address descriptor tells you what the other names corresponding to this R are and then B also has no next use; so, it is not useful anymore. This is the last place where B is used; B is not live after the block. So, we do not have to bother storing.

If it is live, then we have to store it into its memory location. We cannot destroy the value now, and then return R. If all these conditions are satisfied, B is in a register, it corresponds to only one name, it has no next use and it is not live after the basic block, then return R.

If this is not possible, failing one, return an empty register if it is available. If empty registers are also not available then if A has a next use in the block, that is A is useful later on or if B op C needs a register compulsorily, for example, in indexing operations a register is essential if op is an indexing operator, then somehow we will have to free a register.

How do we free registers? We must use a heuristic to find an occupied register and then empty the contents of that register into its home location and release that register.

How do you find an occupied register? There are many heuristics possible. For example, a register whose contents are referenced farthest in future. If it is used immediately in the next instruction, it is not a good idea to empty that register and release it.

But if it is used 5 or 6 instructions away perhaps by the time we reach there, some other register would be free and therefore, we can load that into the register - the value can be loaded into a register all over again.

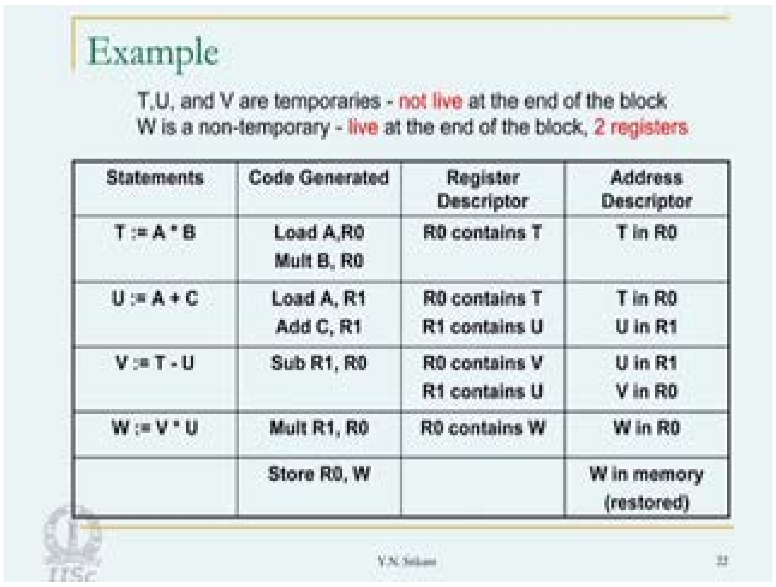
Or the number of next uses is the smallest. Look at the next uses of each register, the value in the register. Take one which is used very few times that may be cost wise the best because if it is used very small number of times, we have to really load that value into a register only a few times, but if you choose a register which is used very large number of times or the value is used very large number of times, we have to load it into a register every time.

So this is the issue behind the heuristic. Now we have a register. We have to split it by the generating an instruction MOV R comma mem. mem is the memory location for the variable in the register R. We can get this information from the register descriptor and that variable is not already in mem.

So this address descriptor will tell you whether mem already contains the updated value or it does not. Once it is done, we probably generate the instruction and update the register and address descriptors because now situation has changed. The registers do not hold the value they are supposed to hold; they have been released. If A is not used in the block or no suitable register can be found, only then return a memory location for L.

So, this is what I was saying it is not compulsory for L to be a register. It can be a memory location, but that is the last resort.

(Refer Slide Time: 36:40)



Example

T,U, and V are temporaries - not live at the end of the block
W is a non-temporary - live at the end of the block, 2 registers

Statements	Code Generated	Register Descriptor	Address Descriptor
T := A * B	Load A,R0 Mult B, R0	R0 contains T	T in R0
U := A + C	Load A, R1 Add C, R1	R0 contains T R1 contains U	T in R0 U in R1
V := T - U	Sub R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W := V * U	Mult R1, R0	R0 contains W	W in R0
	Store R0, W		W in memory (restored)

Y.N. Srikant 22

So let us look at a simple example to understand the code generation process. T, U and V are temporaries they are not live at the end of the block. So, there is no need to store them at the end of the block. W is a non-temporary and it is live at the end of the block. We have to store it into its home location after the block is completed and let us assume there are 2 registers.

First statement is T equal to A star B. To begin with, the first instruction is this. All the registers are free; none of them contain any values. GETREG will return R0 for A. So, we load A into R0. We do not need another register for B because we do not call GETREG, the second time. GETREG would have returned location for A which is R0.

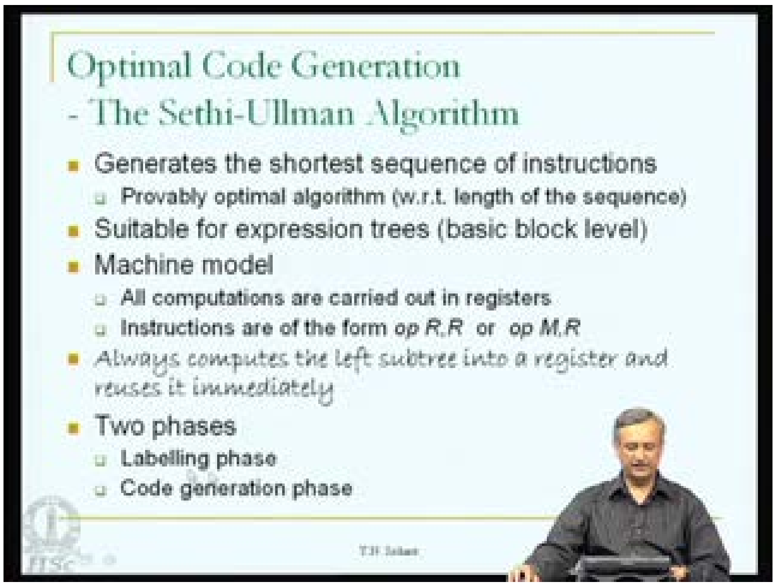
Now, Op B comma R0 is Mult B comma R0 and the result now goes into R0. R0 now contains T that is the register descriptor content and address descriptor says T is in R0.

Now, U equal to A plus C, remember that the R0 which is actually the value of A is now destroyed. It now contains the value of T that is A star B. This is because our code generation scheme is a very simple scheme. It does not see whether there are many uses of the same variable A on the right hand side. It only checks if this T is used later.

U equal to A plus C, we load A into another register R1 which is free; add c to it. Now, R0 contains T, R1 contains U are the contents of register descriptor; T in R0, U in R1 are the contents of address descriptor.

When we look at V equal to T minus U, immediately the GETREG function knows that T is contained in the register using these descriptors. It is in R0, U is in R1; this is immediately found out and it also sees that T is not used later on. So, it is okay to destroy it. It simply emits the single instruction Sub R1 comma R0. From now on, R0 contains V R1 contains U. For the instruction W equal to V star U, it emits the instruction R1 comma R0 because it is okay to destroy V after this instruction and finally, once the basic block is completed the value of W which is in R0 is stored back into its memory location W. Memory is restored and code generation is complete for this particular block.

(Refer Slide Time: 39:44)



Optimal Code Generation - The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form *op R,R* or *op M,R*
- Always computes the left subtree into a register and reuses it immediately
- Two phases
  - Labelling phase
  - Code generation phase

Now we start looking at another scheme of code generation. So far, we saw a model in which code generation is not guaranteed to be optimal or otherwise. There was no result which said anything about the code generation strategy.

Sethi-Ullman algorithm is one of the algorithms which is guaranteed to generate optimal code for a certain type of machine model. For example, look at the cost model; the cost model says it generates the shortest sequence of instructions. The cost is the length of that particular sequence itself and it is possible to prove with respect to the length of the sequence that this is an optimum algorithm.

In other words, the code which is generated by this algorithm is the shortest in length. No other code can be shorter than this, no scheme can give you shorter than this and it is useful for only expression trees. This is a slight disadvantage because you cannot generate optimal code for directed acyclic graph.

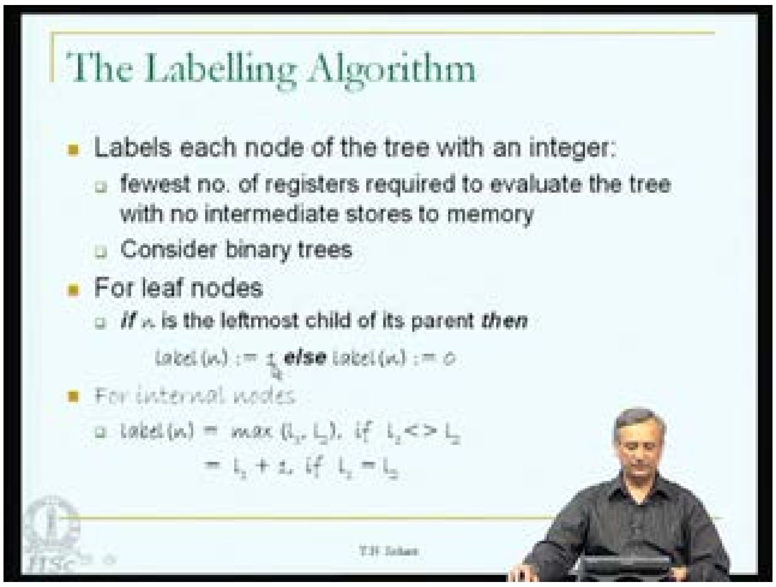
Unfortunately that problem is (( )). So, there is nothing we can do. We have to actually represent basic blocks as trees instead of directed acyclic graph and that is possible. We will see that later.

The machine model which is used here assumes that all computations are carried out in registers. There is no computation which is directly carried out in memory and instructions are of the form, Op R comma R or Op M comma R. So register and register; it does not mean the same register, could be Op Ri comma Rj and op M comma R.

Whatever is the operation, it takes one of the operand as M but, the result is always in R. It always computes the left subtree into a register and reuses it immediately.

This is a very important feature of this particular algorithm. The left subtree is computed into a register and the result of that left subtree is used immediately. There are 2 phases in this particular algorithm: the first phase is the labeling phase and the second phase is the code generation phase.

(Refer Slide Time: 42:35)



The Labelling Algorithm

- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - *if* n is the leftmost child of its parent *then*

    label(n) := 1 ***else*** label(n) := 0
- For internal nodes
  - label(n) = max(l_1, l_2), if $l_1 <> l_2$

    $= l_1 + 1$, if $l_1 = l_2$

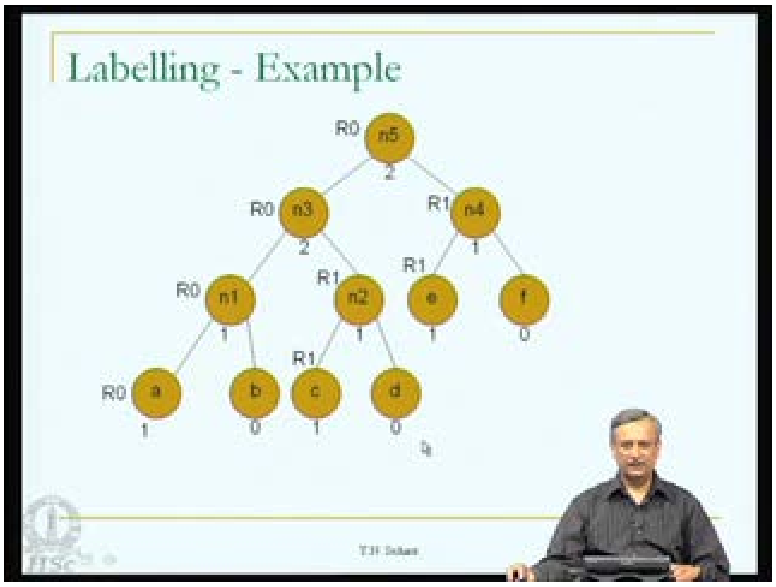
The labeling phase labels each node of the tree with an integer. So, the number that we compute tells you what exactly is the fewest number of registers required to evaluate the tree with no intermediate stores to memory.

So this is what the number tells you. How do we compute such a number? We will see that very soon and we are going to consider only binary trees because all over operators are either unary or binary.

Suppose the node is a leaf node, in that case, if n is the leftmost child of its parent then label of n is 1, otherwise label of n is 0; very simple, leftmost child 1, otherwise 0.

For internal nodes, label is computed as the maximum of L1 and L2, if L1 not equal to L2 and it is L1 plus 1, if L1 equal to L2.

(Refer Slide Time: 43:51)



Labelling - Example

Let us look at an example and understand this process better. We start with the leaves. This is the leftmost child of its parent; so this is 1. This is the second one, the right one; so this is a 0.

Similarly this is a 1 and this is a 0; this is a 1 and this is a 0. For the internal node n1, this value and this value are not the same. L1 not equal L2; so, the maximum is taken and that is 1 here, here also the maximum is 1 and here also the maximum is 1.

Now, all these 3 nodes have 1. Let us look at this node; it has 2 children and its numbering has been 1 and 1. These 2 values are equal; so, this is numbered as 1 plus 1 that is 2 - L1 plus 1.

Now n2, n3 and n4 are the children of n5. This is 2 and this is 1; so they are unequal. We just take the maximum, which is 2.

What does this really mean? This means that this entire expression tree can be evaluated with no stores into memory, with just 2 registers. The same is true for this. It requires 2 registers; this requires 1 register, this requires 1 register and this also requires 1 register.

Let us see how this is done. If you have just 1 register, then load the value of A into that register and perform this operation with b as the other operand. The result is also going to be in the same register. So, 1 register is sufficient for us.

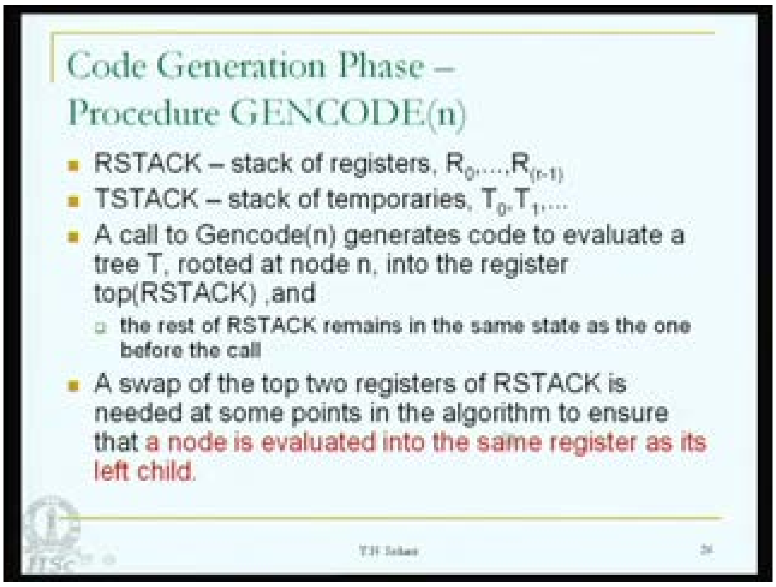
(Refer Slide Time: 45:40) If you do not have another register, this cannot be evaluated. That is the reason, to compute this particular tree, we require 2 registers. We evaluate this tree into 1 register, we already saw how to do that and then we evaluate this tree into another register R1. This is very similar to that. R0 and R1 contain the 2 values of the 2 children and now this can be computed into the same register R0 which is the register of its left child and now R1 becomes free. R0 is held; it holds the entire tree value, but R1 is free. That R1 is used for the computation here.

This is just loading it into R1 and this does the operation with R1 and another operand from here. R1 contains the result of n4, R0 contains the result of this entire tree and finally, we can perform this operation with R0, store the result also in R0 and release the register R1. That is the reason why this entire tree can be evaluated in 2 registers.

Suppose we had performed this particular operation first, we had loaded into R1, then we performed n4 and now the result is available in R1.

We just have one more register left and that is not sufficient for the evaluation of this particular tree. Our code generation strategy is going to be straight forward. It always sees first we compute the numbers, these MINREG values and at every step, we see which subtree requires more registers then generate code for that particular subtree and then go to the other subtree. When we generate code for a particular subtree say k registers are needed for that; we use all the k registers, compute the result, hold it in one of the registers and release the other k minus one registers. This is our strategy.

(Refer Slide Time: 47:59)



Code Generation Phase – Procedure GENCODE(n)

- RSTACK – stack of registers, $R_0, \ldots, R_{(r-1)}$
- TSTACK – stack of temporaries, $T_0, T_1, \ldots$
- A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top(RSTACK) ,and
  - the rest of RSTACK remains in the same state as the one before the call
- A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.

Y.N. Srikant 26

Procedure GENCODE - with a node n which is a root of that particular tree. Here we use 2 data structures: one is called the register stack which is stack of registers R 0 to R r-1. So, there are r number of registers.

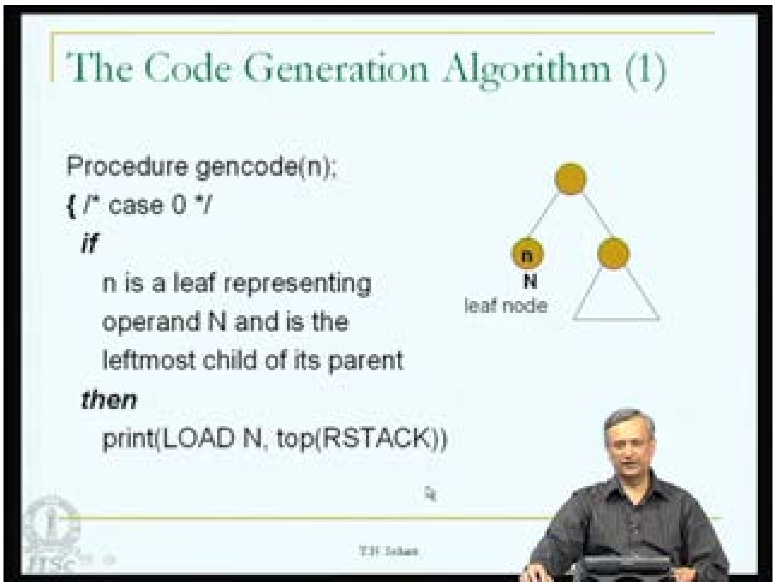
The stack of temporaries is called is called TSTACK and there is no limit on its length as many temporary as we need are available in this particular stack. When we call Gencode n, it generates code to evaluate a tree T rooted at node n, into the register which is on the top of RSTACK. That is a very important thing.

Whenever we call the function Gencode, we give it a stack of registers and a stack of temporaries. The stack of registers will contain the result of this particular tree in the top most registers.

And all other registers are supposed to be free. The rest of RSTACK remains in the same state as one before the call - that is free.

We also require a swap operation of the top 2 registers of RSTACK sometimes to ensure that a node is evaluated into the same register as its left child. These are some of the features. This last feature is needed to prove the optimality of the code. That is why, this is also equally important.

(Refer Slide Time: 49:37)



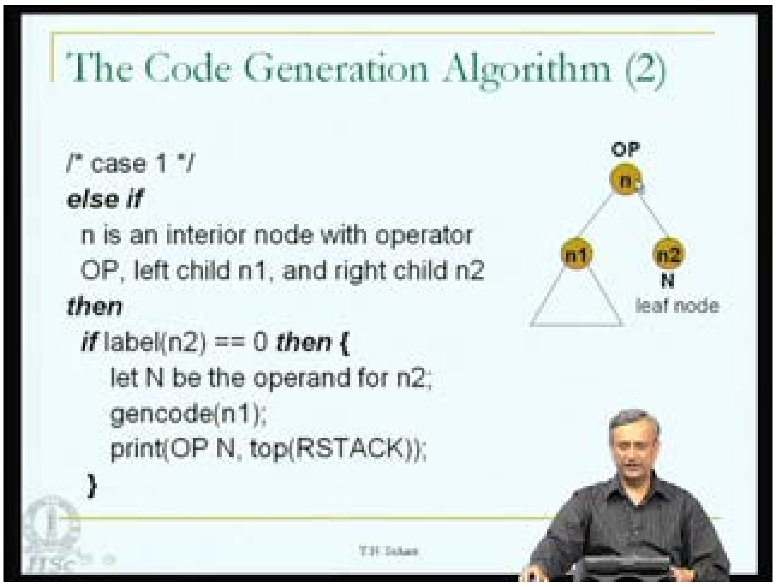
The Code Generation Algorithm (1)

```
Procedure gencode(n);
{ /* case 0 */
  if
    n is a leaf representing
    operand N and is the
    leftmost child of its parent
  then
    print(LOAD N, top(RSTACK))
```

n
N
leaf node

Let us look at the various cases of this particular algorithm. case 0, there is a node n which is a leaf node. This is case 0. If n is a leaf representing an operand N and is the left most child of its parent, we generate the code LOAD N comma top of RSTACK. This will be some register at run time; it may be some R0 or R1, whatever it is, the instruction which should be emitted would be LOAD N comma that particular register. This is the case 0.

(Refer Slide Time: 50:23)



The Code Generation Algorithm (2)

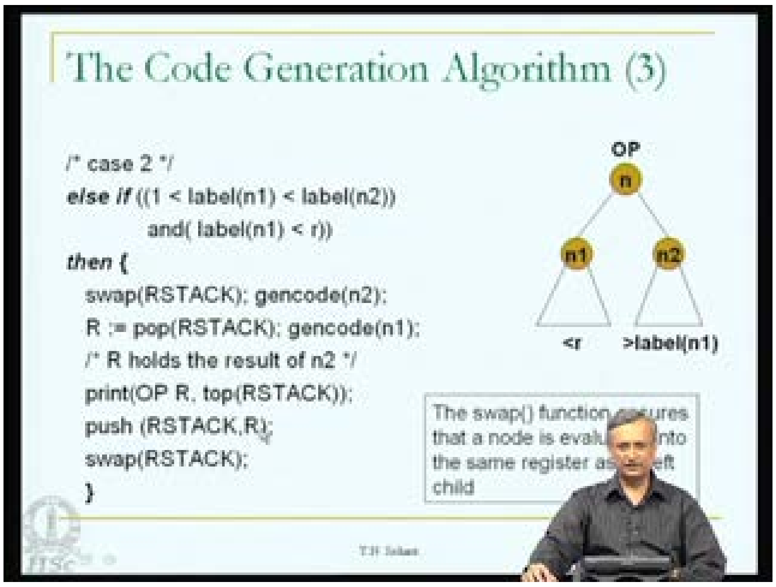
```
/* case 1 */
else if
  n is an interior node with operator
  OP, left child n1, and right child n2
then
  if label(n2) == 0 then {
     let N be the operand for n2;
     gencode(n1);
     print(OP N, top(RSTACK));
  }
```

OP
n
n1
n2
N
leaf node

Case 1: the right operand is a leaf and the left operand is a subtree. n is an integer node with operator OP, left child n1 and right child n2. Right child is a leaf node. That is if label n2 equal to 0, that is it is a leaf node, let N be the operand for n2; we generate code for n1 by calling the rooting gencode recursively on this particular subtree.

Once this is completed, the top of RSTACK will contain the result of n1. We generate code OP N comma top RSTACK. Now again the top of RSTACK will contain the result of this entire tree as well. So, we have restored the stack the way it was.

(Refer Slide Time: 51:18)



The Code Generation Algorithm (3)

```
/* case 2 */
else if ((1 < label(n1) < label(n2))
          and( label(n1) < r))
then {
   swap(RSTACK); gencode(n2);
   R := pop(RSTACK); gencode(n1);
   /* R holds the result of n2 */
   print(OP R, top(RSTACK));
   push (RSTACK,R);
   swap(RSTACK);
   }
```

The swap() function ensures that a node is evalu into the same register as left child

OP
n
n1
n2
<r
>label(n1)
T.H. Srikant

This is case 2. You have a subtree n1; you have another subtree n2 for this particular node n. This subtree n1 requires less than r number of registers; this subtree n2 requires more than what n1 requires. 1 less than label n1 less than label n2, that is the order; the other order is considered in the next slide and label n1 is less than r. In other words, we require less than r number of registers to evaluate n1.

(Refer Slide Time: 51:52) This requires a little more than what this requires. Let us say r is 10, this requires 5, this could require 8 or it could require 11 that does not matter. In this case, we do what is known as swap operation on the RSTACK. Why? The swap function ensures that a node is evaluated into the same register as its left child.

This will become very clear now. So, swap the RSTACK. If let us say the top two are R0 and R1; now it becomes R1 and R0. gencode n2, R1 is at the top of stack. (Refer Slide

Time: 52:28) The result of this entire thing, because this requires more registers, the recursive routine is called on n2 first. This I have explained already.

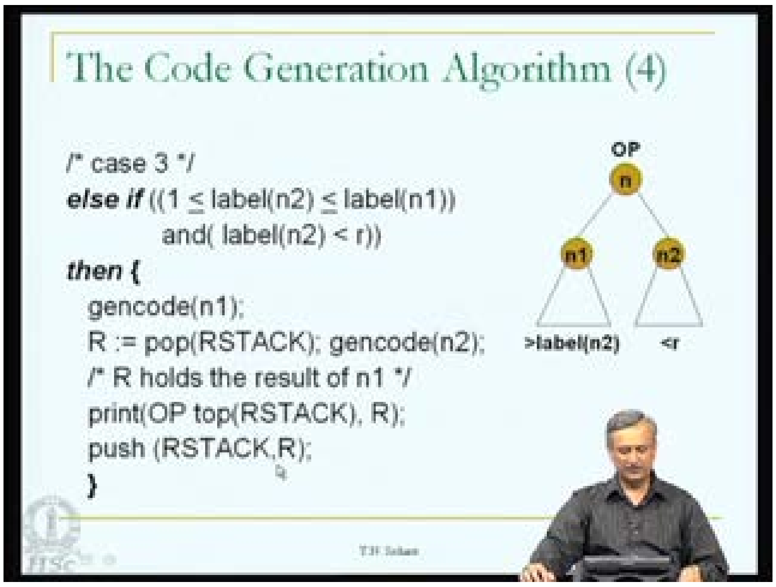
This tree will evaluate into the top of RSTACK which is R1 after the swap. Pop that R1 into the variable R at compile time, variable R. R0 is on the top of the stack, below that is R2 etcetera. Call this routine on n1.

When we do that, R0 will now contain the result of this particular subtree n1. Now, generate the code for this entire thing OP R, which is this and top of RSTACK which is R0, which is corresponding to this.

So this entire tree is now available in top of RSTACK which is R0, but R is outside the stack. So, we push the R into the RSTACK.

But then now the order is reversed; R1 is at the top and R0 is below it. So, we swap the RSTACK, the top 2 entries. R0 comes to the top and it also holds the value of this entire tree, thereby we have restored the configuration the way it was.

(Refer Slide Time: 53:53)



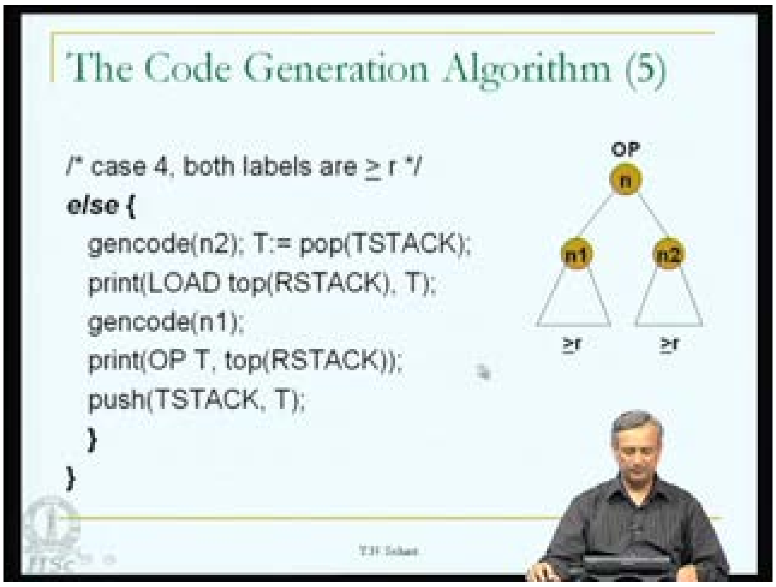
The Code Generation Algorithm (4)

```
/* case 3 */
else if ((1 ≤ label(n2) ≤ label(n1))
          and( label(n2) < r))
then {
  gencode(n1);
  R := pop(RSTACK); gencode(n2);
  /* R holds the result of n1 */
  print(OP top(RSTACK), R);
  push (RSTACK,R);
  }
```

Now the case number 3, this right child requires less than r; this requires more than what n2 requires - so the other case. 1 less than equal to label n2 less than equal to label n1 and label n2 less than r. Obviously, in this case we do not require any swap. We just generate code for n1 and pop it.

So, R0, now, is in the variable R. It is removed from the RSTACK. Call gencode on n2; that will be evaluated into the present top of stack say R1. Issue the instruction OP top of RSTACK comma R and finally, push. This should have been OP R comma top of RSTACK, there is a slight error here. The assumption is The result is the second operand. Then we push the removed register into the stack again, thereby R0 now is at the top of the stack and the configuration is restored.

(Refer Slide Time: 55:03)



The Code Generation Algorithm (5)

```
/* case 4, both labels are ≥ r */
else {
   gencode(n2); T:= pop(TSTACK);
   print(LOAD top(RSTACK), T);
   gencode(n1);
   print(OP T, top(RSTACK));
   push(TSTACK, T);
   }
}
```

The last case is case 4; both require more than r number of registers, more than equal to r number of registers.

We need a temporary, in order to hold this particular result. Let us say arbitrarily, we gencode for n2, take a temporary from the stack, push that result from top of RSTACK to t, the temporary. All the registers are available for n1, generate code for n1, print the instruction OP T comma top of RSTACK, thereby the result of this entire tree is available in the top of RSTACK and replace the temporary into the temporary stack.

So, thereby we have actually use the entire stack of registers which is available to us, but because both these children - the subtrees, require more than r number of registers, it is not possible to generate code without storing this particular result. Gencode n2 generates a result in the top of RSTACK; it cannot be retained there. We have to place it in a temporary and then continue with the operations. We will stop here. This lecture will stop here and the next time, we will continue with examples of the Sethi-Ullman algorithm and also the dynamic programming algorithm. Thank you.