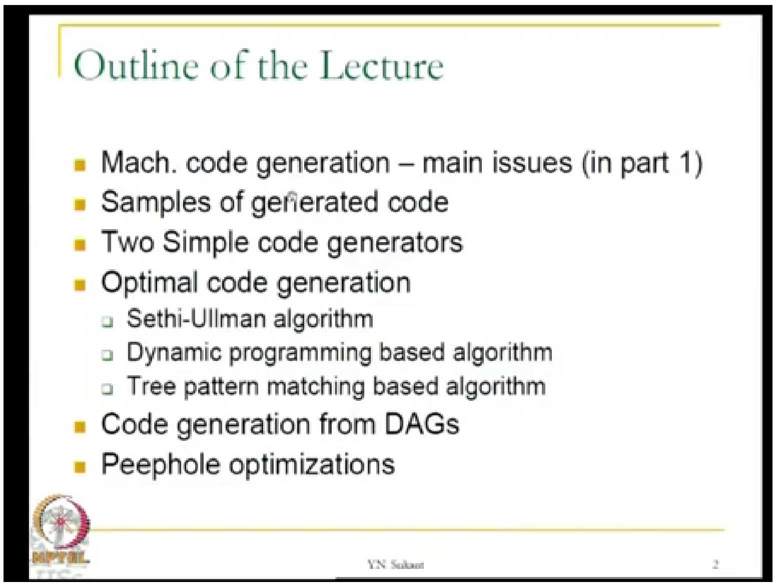
# Principles of Compiler Design
**Prof. Y. N. Srikant**
**Department of Computer Science and Automation**
**Indian Institute of Science, Bangalore**

**Lecture - 25**
**Machine code generation Part – 2**

(Refer Slide Time: 00:21)



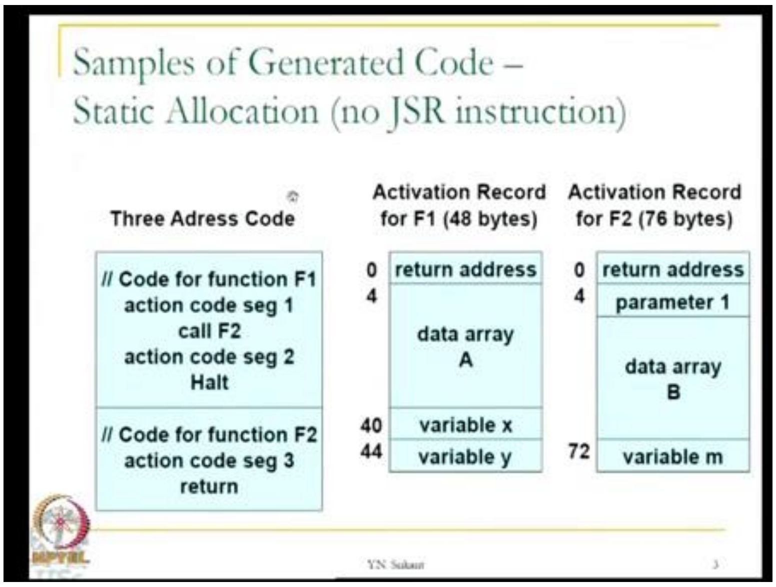
## Outline of the Lecture

- Mach. code generation – main issues (in part 1)
- 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 part two of machine code generation. So, in the part one of the lecture we considered the main issues in a machine generation and I also gave you a sample of generated code.

(Refer Slide Time: 00:34)



## 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 |

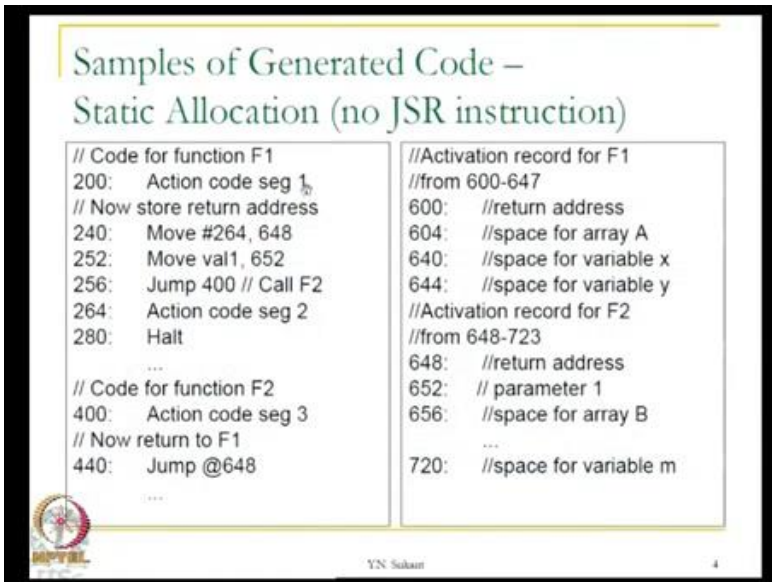
Activation Record for F2 (76 bytes)

| | |
|---|---|
| 0 | return address |
| 4 | parameter 1 |
| | data array B |
| 72 | variable m |

YN Srikant 3

So, today we will continue looking at the samples and other material, so this is the sample, a code that we saw last time. There is a function F 1 which has action code segment 1 and then a call to another function F 2, another piece of code segment 2 and then halt, whereas function F 2 has just a piece of code and then it returns. The activation record for both the functions F 1 and F 2 are shown here. So, the activation record takes 48 bytes for F 1 and 76 bytes for F 2, because we do not have any jump sub routine instruction. We really store the return address and then you know jump to the sub routine itself. So, because of that we require a slot in the activation record of the collie to store the return address as well.

(Refer Slide Time: 01:41)



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

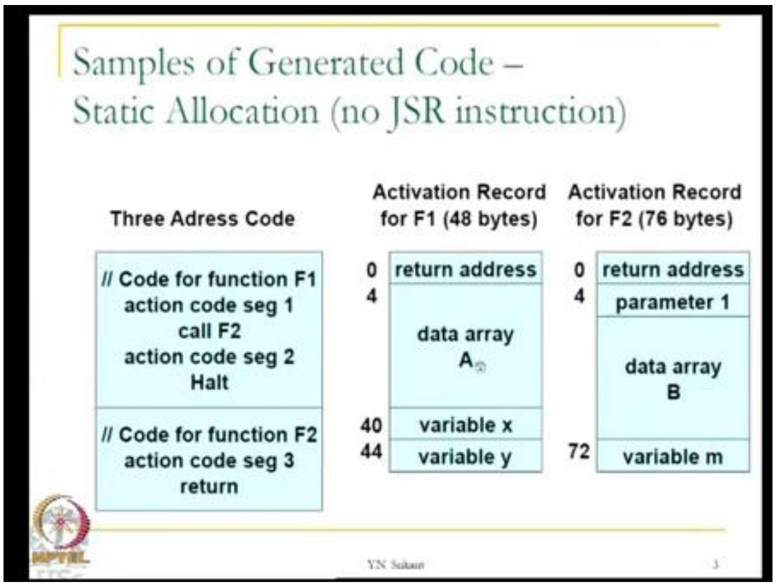
```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move val1, 652
256:    Jump 400 // Call F2
264:    Action code seg 2
280:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1
440:    Jump @648
        ...
```

```
//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 variable m
```

YN Srikant 4

Here is the code that is generated with the machine code for function F 1, so the code for action code segment 1. So, that is here, now it is time to call the function F 2, so to begin with we store the return address, so remember there is no need to you know allocate activation records it is static allocation. So, the activation records have already been set up in memory and they are permanent. So, the activation record for F 1 spans the addresses 600 to 647 and the activation record for F 2 spans the addresses from 648 to 723.

(Refer Slide Time: 02:31)

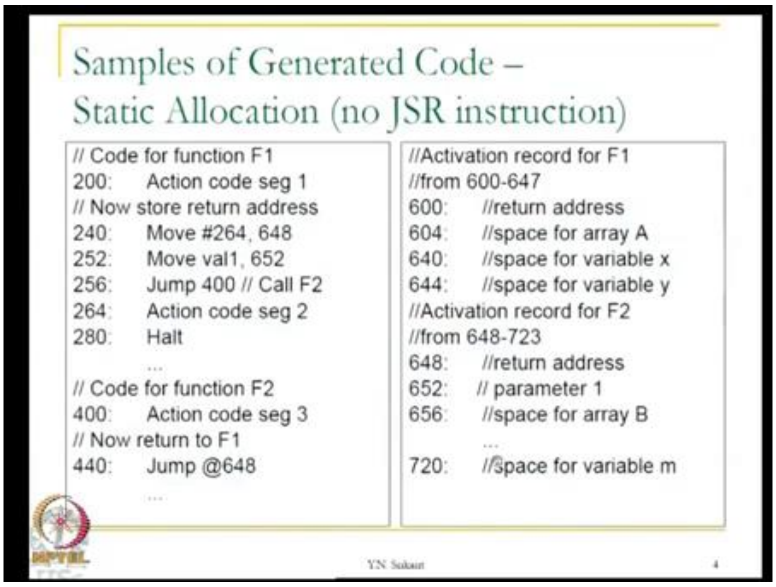


## Samples of Generated Code – Static Allocation (no JSR instruction)

YN Srikant 3

So, this is just a replica of this picture return address data array a variable x variable y.

(Refer Slide Time: 02:37)



## 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 val1, 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
        ...
```

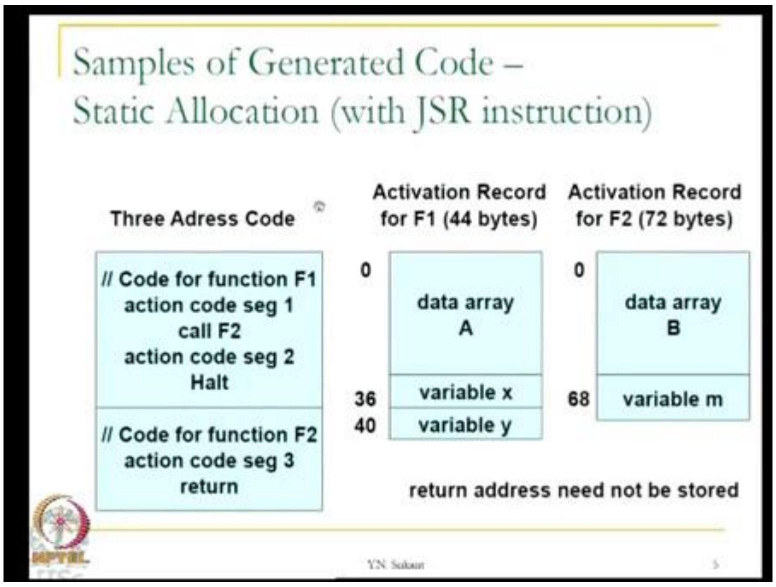
```
//Activation record for F1
//from 600-647
600:    //return address
604:    //space for array A
640:    //space for variable x
644:    //space for variable y
//Activation record for F2
//from 648-723
648:    //return address
652:   // parameter 1
656:    //space for array B
        ...
720:    //space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

Y.N. Srikant 4

So, you can see that and similarly, this as well, so before we jump to the sub routine the return address which is this particular address. The address of action code segment 2 that is moved actually it is moved into location 648 which is the slot to store the return address in the activation record of F 2. Then the parameter is actually pushed or moved into location 652, so that is corresponding to parameter 1 here and then we execute a simple jump instruction.

So, when we go to function F 2, we execute the code and now use the return address, which is given on the activation record at location 648. So, indirect jump will take us back to this particular address 264, so here we execute action code segment 2 and then halt. So, this is the scheme when we do not have any jump subroutine instruction and it is static allocation.

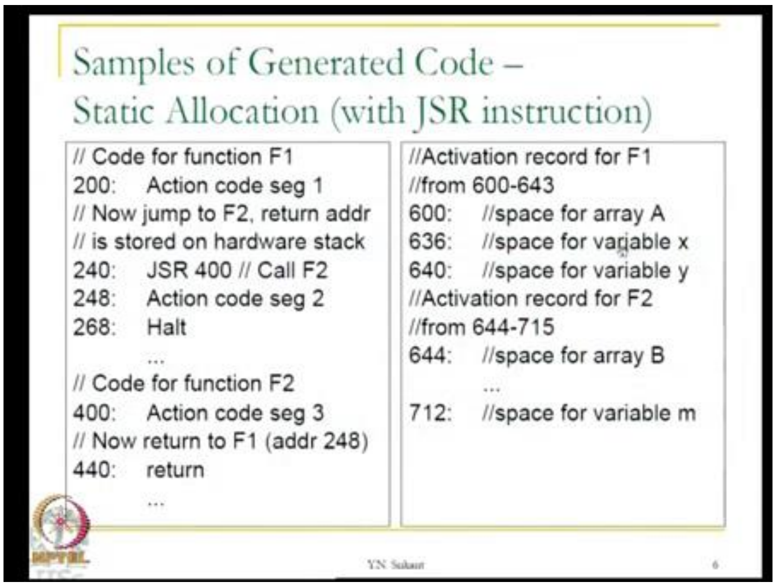
(Refer Slide Time: 03:47)



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

So, let us see what happens with jump subroutine instruction, the address code is three address code is the same. It is just that this activation record does not have a slot for return address here, so the number of bytes required is 44 here and 72 here.

(Refer Slide Time: 04:12)



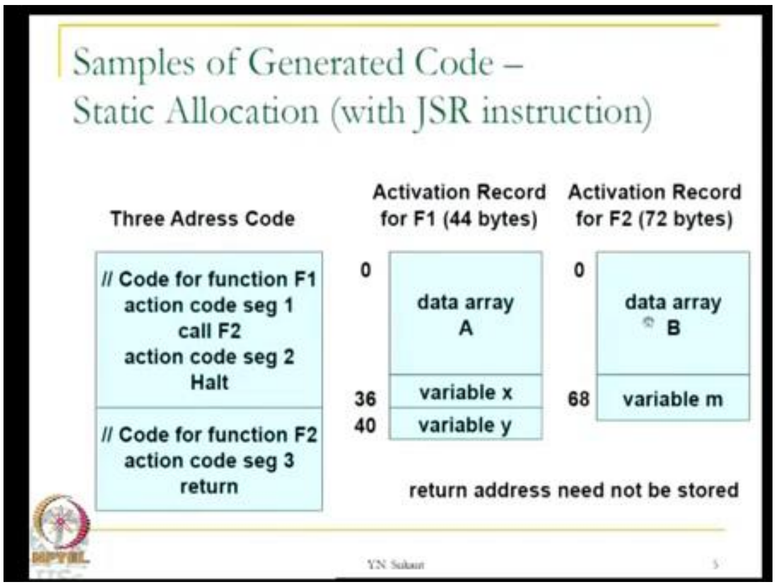
## Samples of Generated Code – Static Allocation (with JSR instruction)

```
// 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
```

What about the code itself, the machine code, now it is slightly simpler, so this would be you know action code segment 1.

(Refer Slide Time: 04:24)



## 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)

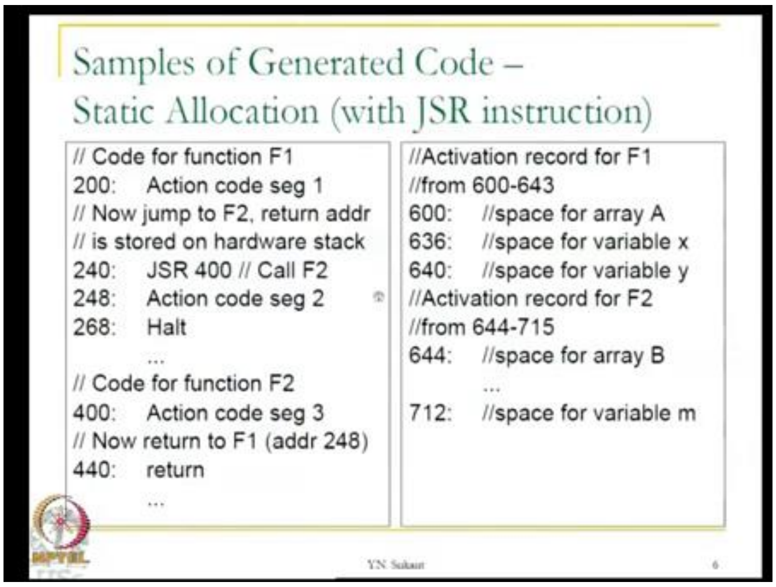
| | |
|---|---|
| 0 | data array B |
| 68 | variable m |

return address need not be stored

YN Srikant 5

So, here suppose we and I have also change a little bit in the F 2, so there is no parameter either just to simplify the code.

(Refer Slide Time: 04:36)



## 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
        ...
```

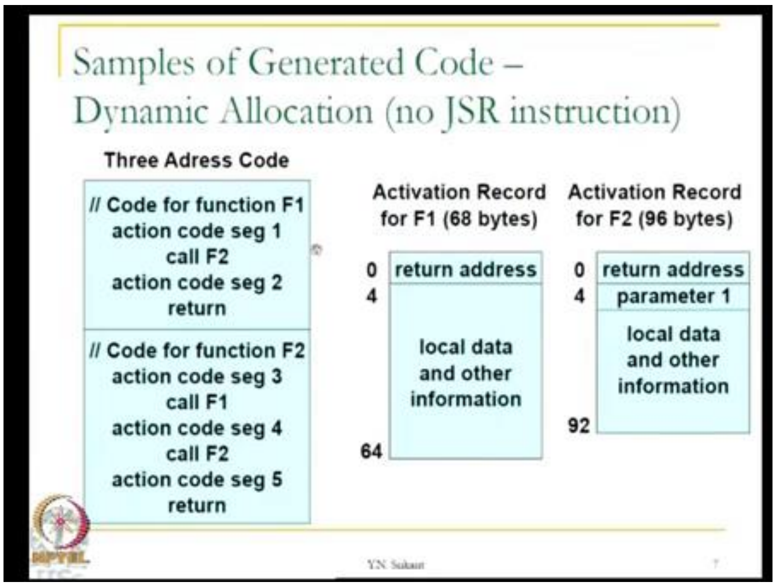
```
//Activation record for F1
//from 600-643
600:    //space for array A
636:    //space for variable x
640:    //space for variable y
//Activation record for F2
//from 644-715
644:    //space for array B
        ...
712:    //space for variable m
```

YN Srikant 6

So, here we execute action code segment 1 and now since the return address is automatically stored on the hardware stack when we execute the jump subroutine instruction, there is no reason to store it. So, we execute JSR 400, which automatically stores the address 248 on the hardware stack.

So, we go to this particular address a start executing it and then we say return. It automatically picks up the return address from the stack and returns to this particular address 248, activation record formats remains the same except that there is no space for the return address otherwise there is not much change.

(Refer Slide Time: 05:18)



## 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 |
| 64 | |

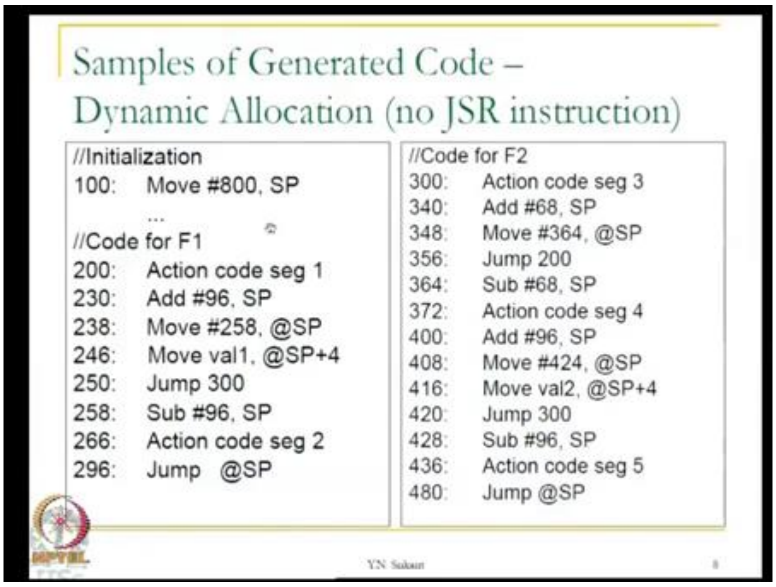
Activation Record for F2 (96 bytes)

| | |
|---|---|
| 0 | return address |
| 4 | parameter 1 |
| | local data and other information |
| 92 | |

YN Srikant 7

Now, the other varieties dynamic allocation and no jump subroutine instruction, so there is code for function F 1, then action code segment 1 call F 2 action code segment 2 and return and F 2 has action code segment 3. Then there is a recursive call to F 1 action code segment 4 another recursive call to F 2 itself and then action code segment 5 and return. So, the return address is stored here because we do not have a jump subroutine instruction and there is local data and other information all the other a temporary sets etcetera stored here. So, we require 68 bytes here and 96 bytes here, so there is also a parameter in the function F 2.

(Refer Slide Time: 06:10)



## 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 #258, @SP
246:   Move val1, @SP+4
250:   Jump 300
258:   Sub #96, SP
266:   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:   Move val2, @SP+4
420:   Jump 300
428:   Sub #96, SP
436:   Action code seg 5
480:   Jump @SP
```

Y.N. Srikant 8

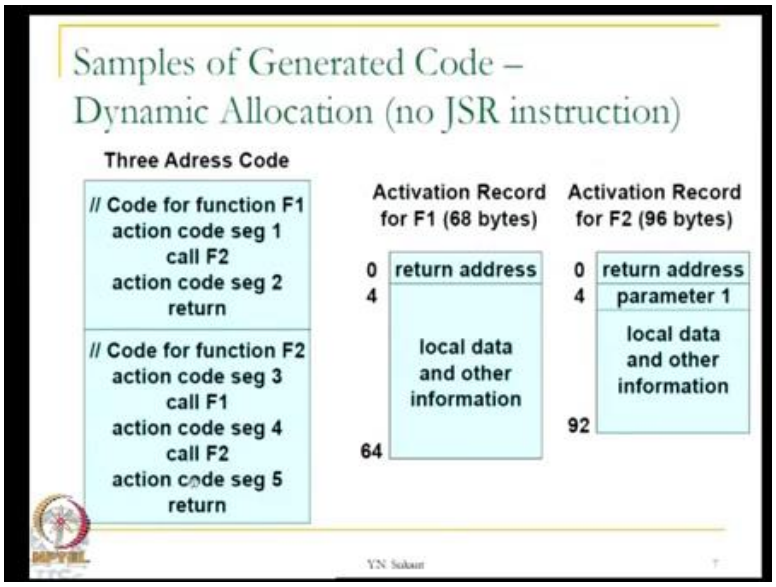
Now, we need to set up activation records when we call functions, so we must initialize the stack pointer say some value 800 that is where the stack may begin. So, let us move that into the stack pointer so that it is a valid address. So, this is a address at which the stack begins, now the code for F 1, it has code for the action code segment 1. Now, this part is the allocation of the activation record add 96 to SP. So, that means we increment the stack pointer by 96, so thereby allocating you know 96 bytes for the activation record of F 2.

Now, we move 258, which is the return address to the first location on the stack pointer move the parameter to the next location and then execute a jump. So, except for this part the rest of it is the same, of course, there is a minor difference, we use the stack pointer with indirect addressing in order to access the information on the activation record or put something into the activation record.

So, all the information is access to SP, so that is why the indirect addressing scheme is required once we jump to 300 that is the code for F 2. So, this has code for the action code segment 3, now there is a recursive call, so the size of the activation record here the size of the activation record for F 2 was 96, the size of the activation record here for F 1 is 68. So, 68 is added to SP thereby allocating the activation record for F 1 on the stack the rest of it is similar the return address and there is no parameter.

So, jump to 200, so again you know the action would come to this point when the code executes, so once the code completes it there is a return using the SP. So, indirect SP, so that is precisely what brings us to you know the return point, so that is 258. So, that is the this is the point, so we when we execute, sorry not this point here, so from here we the return address would be 364, so 364 is the place where we return after we call to F 1 is completed. So, at this point we must de allocate the activation record, so subtract 68 from SP, so that brings us back to the original position. Now, the code for segment four is you know present.

(Refer Slide Time: 09:14)



## 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 |
| 64 | |

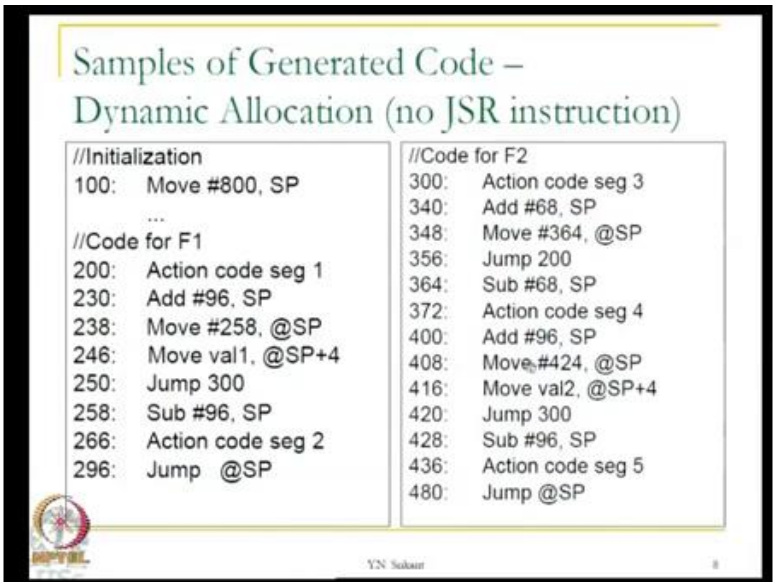
Activation Record for F2 (96 bytes)

| | |
|---|---|
| 0 | return address |
| 4 | parameter 1 |
| | local data and other information |
| 92 | |

YN Srikant

Then, there is another recursive call, so please see that there is recursive call F 2 again.

(Refer Slide Time: 09:18)



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

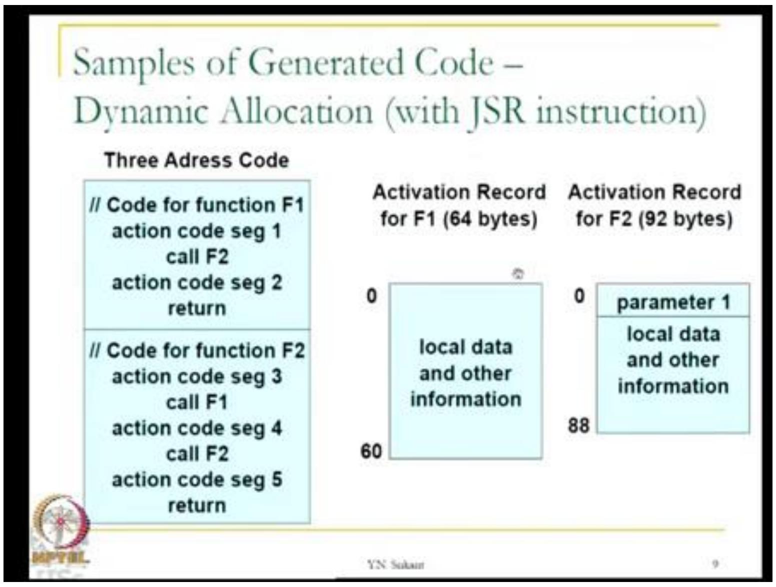
```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #96, SP
238:    Move #258, @SP
246:    Move val1, @SP+4
250:    Jump 300
258:    Sub #96, SP
266:    Action code seg 2
296:    Jump  @SP
```

```
//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:    Move val2, @SP+4
420:    Jump 300
428:    Sub #96, SP
436:    Action code seg 5
480:    Jump @SP
```

YN Srikant 8

So, again the stack allocation of the activation record happens return address is pushed on to the stack the parameter is also pushed on to the activation record and a jump to 300 that is this code itself happens. So, this is how the recursive call you know takes place, so the point time trying to make is that there is no difference between the recursive call and non recursive call. The activation records are created in both cases dynamically rest of it is simple you know whenever there is a return from sub routine it comes back and so on and so forth.

(Refer Slide Time: 09:57)



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

Three Adress Code

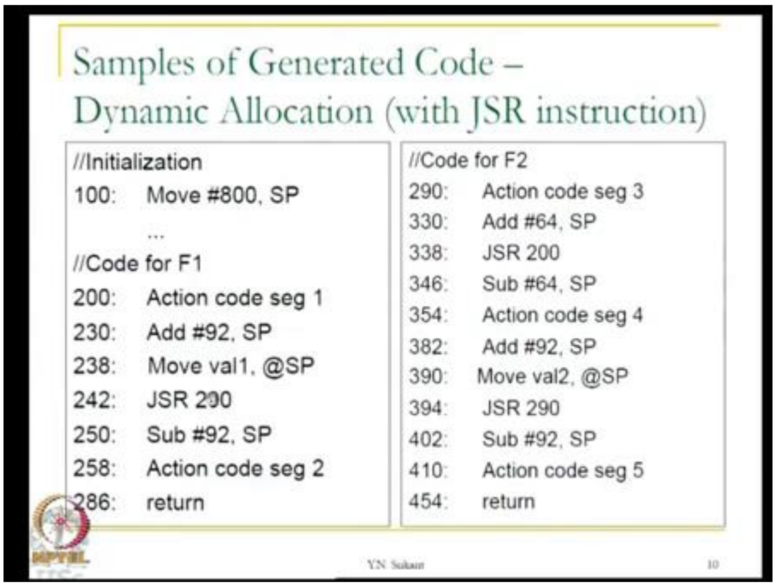
```
// Code for function F1
action code seg 1
call F2
action code seg 2
return

// 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
60
Activation Record for F2 (92 bytes)
0
parameter 1
local data and other information
88

YN Srikant 9

Finally, dynamic allocation with jump sub routine instruction the only difference is there is no return you know slot, there is no slot to store the return a address.

(Refer Slide Time: 10:11)



## 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 val1, @SP
242:    JSR 290
250:    Sub #92, SP
258:    Action code seg 2
286:    return
```

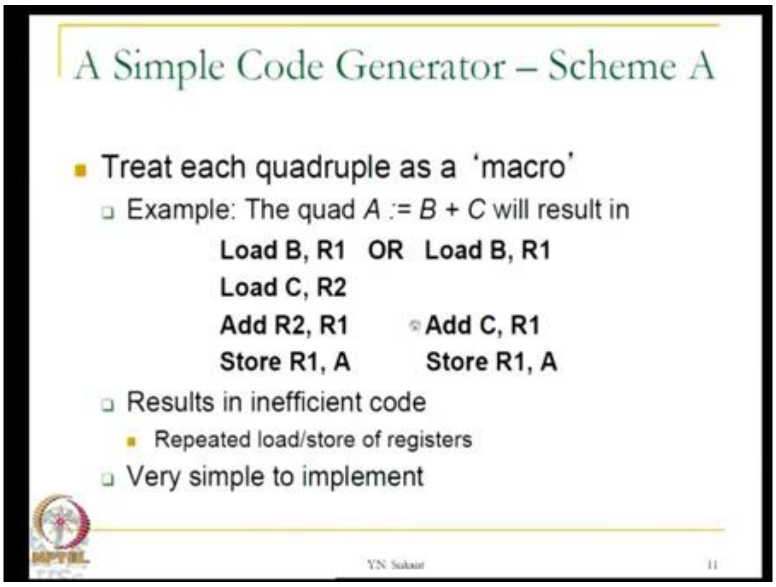
```
//Code for F2
290:    Action code seg 3
330:    Add #64, SP
338:    JSR 200
346:    Sub #64, SP
354:    Action code seg 4
382:    Add #92, SP
390:    Move val2, @SP
394:    JSR 290
402:    Sub #92, SP
410:    Action code seg 5
454:    return
```

Y.N. Srikant 10

Rest of it is the same, so we have to allocate the activation record now the size is slightly less and then we jump sub routine to 290 that is the address of F 2. So, this implies the return address is stored on the hardware stack and then the control goes here. So, even for the you know for example, the recursive call to F 1 it just allocates activation record and jumps to 200 using JSR instruction.

Similarly, when we have a recursive call to F 2, we again allocate another activation record move the parameter and then jump to the beginning of F 2 itself using the JSR instruction. So, JSR return automatically takes care of you know returning to this particular address following the place where we called because the return address is stored on the activation in the hardware stack.

(Refer Slide Time: 11:13)



## 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 C, R1
Store R1, A       Store R1, A
```

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

Y.N. Srikant 11

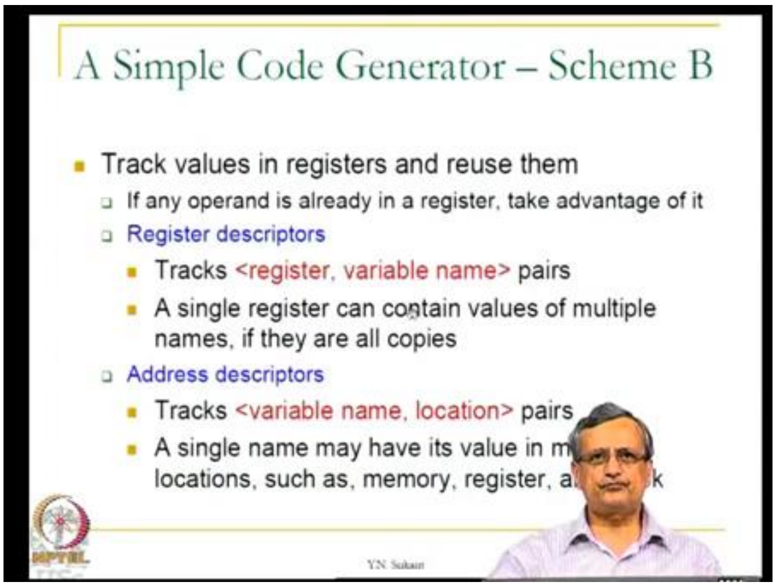
So, that is about the instructions which are really going to be generated for various types of intermediate code. Now, let us begin our discussion on the code generator itself what is the design that we are op for a code generator and there are at least two simple code generators that we are going to discuss the scheme. For example, it treats each quadruple as a macro, so if you take the quadruple A equal to B plus C as a generic quadruple plus need not stand for the plus operator it could be plus minus star slash any one of these.

So, the code generated would be either this or this, so the point is every time we want to do an operation on an operand. We will load it into a register do the operation store it back into that memory location so that no registers are presumed to be news in any particular translation scheme. So, here load a equal to you know A equal to B plus C says load B into R 1 load C into R 2 add R 2, R 1 and store R 1 to A.

So, after the code is executed both R 1 and R 2 are free, similarly if you use this scheme load B comma R 1 add C comma R 1 and store R 1 comma A the register R 1 will be free after the entire sequence is executed. So, the advantage of doing this is we do not have to perform any register location and we can simply write a macro for each one of the quadruples. The macro will be expanded by and whenever we instead of you know the code for a equal to b plus c will be just that call to the macro corresponding to the quadruple.

So, the final output would be a sequence of macro calls, it should be expanded by the macro assembler and the machine code would be generated the difficulty, here is inefficient code repeated you know loads store of registers. So, every time since we load the value from the register to the location rather load the register from the location. It implies load and again and again, so we are not reusing the value which is already present in a register in some other quadruple. So, this is very inefficient, but very simple to implement, so if we are looking at a proto type compiler, then the first cut code generator could be based on macro. So, slowly gradually as the A code is the compiler debug, you know we could start designing a better code generator.

(Refer Slide Time: 14:25)



## 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, location> pairs
    - A single name may have its value in m locations, such as, memory, register, a k

YN Srikant

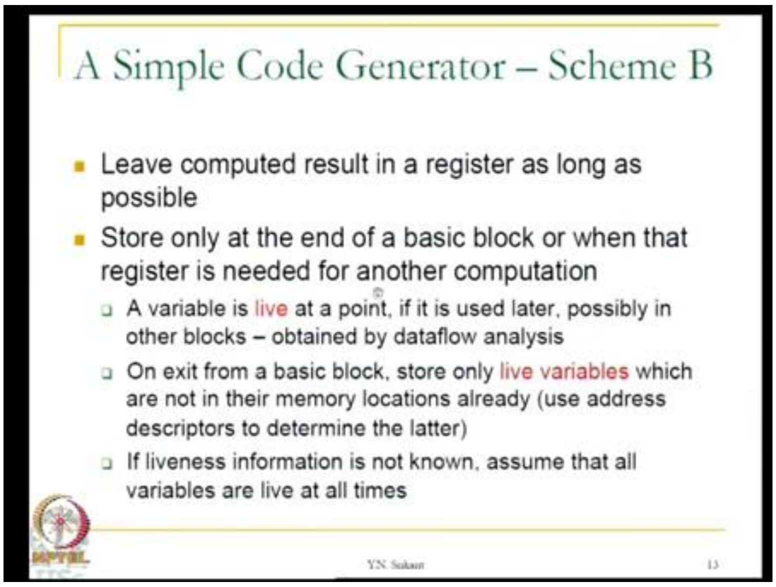
Then, scheme B is slightly more complicated, so here we track the values which are present in registers and try to reuse them. So, the basic rule is if any operand is already in a register take advantage of it. So, this also makes the code generator a little more complicated, we require to track the value at registers when and make sure that we know when it changes values and so on and so forth for that purpose we use register descriptors and address descriptors.

So, these are two data structures that we apply a register descriptor actually tracks register and variable name pairs. So, for each register it says which are you know variables that contain the value at the same time you know. So, if a single variable can be present in different registers, then that is also possible, so many registers can contain the value of a single variable or a single register can contain the values of multiple variables.

We try to avoid the first situation that is many registers containing the same value of the same variable make by making sure that we use the same register. Whenever there is a need to use it, so the other situation is what we need to worry about that is a single register containing a values of multiple names you know if there are all copies.

So, we need to track this and make sure that whenever that variable name occurs the same register is used. So, pairs of this kind or stored in the register descriptor for each register, what are the variable names that it can correspond to the in some way the inverse of that would be with a variable name. What are the locations corresponding to it so it is possible that a single variable name may have its value in multiple locations. It will be definitely in memory, but it may also be in register and if it is a stack machine it may be on the stack as well. So, it is necessary to track such pairs as well and in the process of code generation these two data structures would be repeatedly updated.

(Refer Slide Time: 17:23)



## 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
  - A variable is live at a point, if it is used later, possibly in other blocks – obtained by dataflow analysis
  - 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 13

So, the basic principle is to leave the computed result in a register as long as possible, so what does that mean. So, we try to use a different register whenever we need an extra register and if there are no registers only then we see. Now, it is time to empty a register and store its value back in the memory location corresponding to it. So, every variable has what is known as a home location corresponding to it, so the home location will be updated whenever the register corresponding to it cannot be allow to keep the value anymore the other point. When we need to empty a register would be the end of a basic block, so I already a mention that the register will have to be empty.

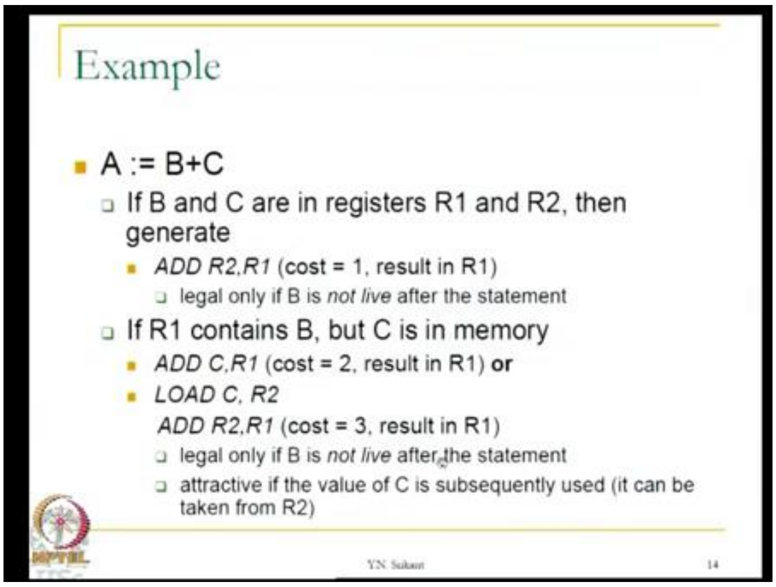
When the register is needed for another computation, but at the end of the basic block also we need to empty the register the reason is our code generation scheme is a basic block oriented scheme. It does not look at any other basic block when it is generating code for a particular basic block. So, at the end of the basic block it is stores all the registers back into their home locations and then you know make sure that the register set is vacant empty for the next basic block. So, at the end of the basic block do we have to store, you know register into its home location in other words do we have to generate instructions to store every register into its home location not necessarily.

For example, we use the concept of livens to help us in this in making this decision we will discuss livens in great detail later during data flow analysis, but for a basic block the definition is very simple a variable is live at a point. If it is used later end of course, if it is used in other basic blocks then it is also live, but we will not bother about such a usage because it requires data flow analysis to track such usage. Suppose, we know that data flow analysis has been performed livens analysis has also be in performed. So, at the end of the basic block, we know which variables are going to be used in other basic blocks and which variables will not be use definitely in other basic blocks.

If this information is known to us I am now talking about something, which we have not yet discussed livens is available by data flow analysis that is the assumption here. Then we can make a sophisticated decision, so on exit from a basic block we need to store only those live variables that are not in their memory locations already. So, if you look at the address descriptor it will tell you where the variable has its value is it in the home location also or is it just in a register. If it is only in a register, then you know when we need to generate a instruction to copy the register value to its home location but if it is also in its home location.

Then, we do not have to generate such an instruction that is one the second is what if the variable is not live at all if the variable is not live. Then it will be not use further, so whether it is in a register or home location it really does not matter. So, we do not have to generate any instruction to store it back into their home location. If the livens information is not known to us, then we must assume that all variables are live at all times. So, this is you know most pessimistic assumption which leads to bad code, but under certain circumstances we may have to leave with this assumption.

(Refer Slide Time: 21:48)



## 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)

YN Srikant 14

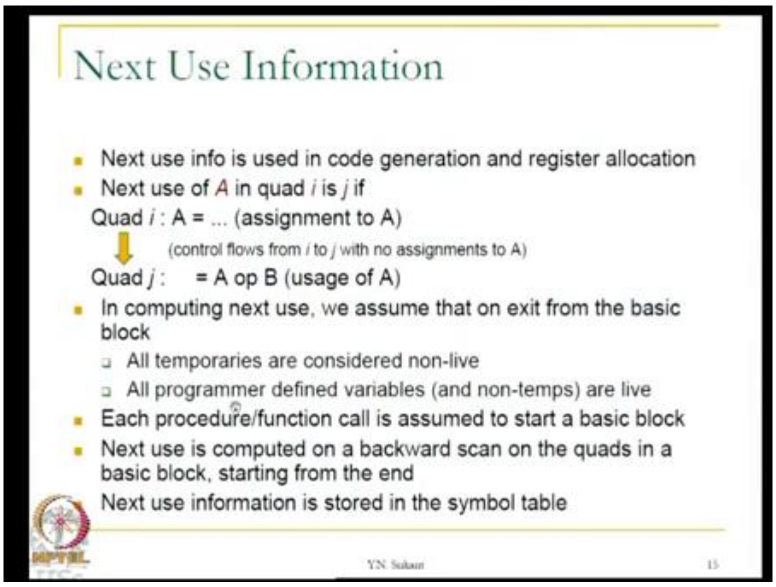
Let us take an example, here is a quadruple A equal to B plus C as I said plus is a generic operator if B and C are in registers and in say R 1 and R 2. So, B is in R 1 c is in R 2. Now, we can possibly generate an instruction add R 2 comma R 1 which brings the result into R 1 cost of course, is one because it uses only registers when is this legal according to the scheme that we are using. We need to keep the value of a variable in a register as long as possible, so this scheme this instruction is legal only if B is not live after the statement, so B is in R 1 and now we are over writing B rather R 1.

So, the value available in R 1 is lost if the value of R 1 is going to be use later, we would possibly have not you know updated its home location. So, generating this instruction when R 1 B is still live rather then would be incorrect this is legal only if B is not live. Otherwise, if we are force to empty the register R 1, we will have to generate instruction to move the contents of for R 1 to the home location of B.

Then, use the a register R 1, so we will see that also a little later, so this is you know that means we need to check where to generate rather where to store the result the second possibility is r one contains B, but C is in memory well you could simply add C comma R 1. So, generate this instruction cost is slightly higher than the previous one that is cost two that is understandable because C is in memory, but the result is in R 1 again this is legal only if B is not live after the statement.

So, that is another thing that we need to keep in mind third possibility is we load C into R 2 then add R 2 comma R 1. So, again the cost is even higher and the result is in R 1 again this is live legal only if B is not live after the statement, but this could be attractive if the value of C is subsequently used, so it can be now taken from R 2 itself because we loaded it into R 2.

(Refer Slide Time: 24:46)



## 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 15

So, the examples are supposed to give you some idea of the complexity involved in code generation that is we need to check the address and the register descriptor at various points in time. We also require some extra information called the next use information to make an inform decision about which register to uses etcetera. So, next use information is used both in code generation, and our local register allocation, which we are going to discuss very soon. So, what is the definition of the next use information, so the next use of a in quadruple i is j.

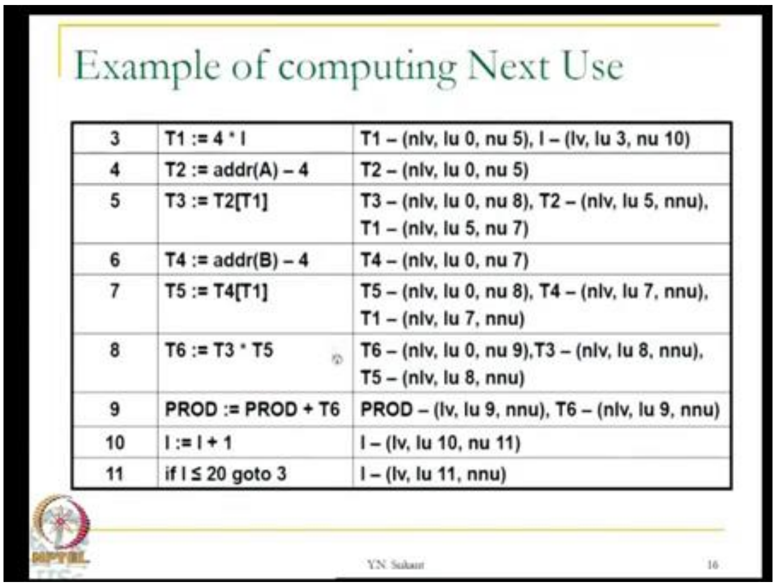
So, in other words we have a quadruple i which defines a value for A, so there is an assignment A equal to something and then we have you know several other quadruples before quadruple j which uses A. The most important point here is control flows from i to j, but there are no more assignments to A. So, the value of a is not changed, then the value computed here is used in this quadruple.

So, we say that next use of a in quadruple i is j, so we have to compute the a next use information, we will see how to use it a little later. Obviously, we need to check for this condition and then attach information about the next use of i to the quadruple to the quadruple i, so in computing, next use we assume that on exit from the basic block all temporaries are considered non live.

So, this is a correct decision because we do not reuse temporaries across basic blocks we generate new temporaries whenever necessary all programmer defined variables and non temporary variables are suppose to be live. So, we did not perform any livens analysis data flow analysis and that is the reason why we are resuming this. So, then each procedure and or function call is suppose to start a new basic block, I already discussed this in the you know basic block discussion lecture on basic blocks, so otherwise we have a problem of killing all the quadruples in the basic block.

So, this is best you know sent to a different basic block next use is computed in a backward scan on the quadruples in a basic block starting from the end. I will give you a example to explain the procedure and of course, the next use information is stored in the symbol table so for that particular variable the next use information is also attached.

(Refer Slide Time: 27:54)



## 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 16

So, here is the example for computing next use, so here this is a simple basic block it is the same dot product example it is just that the quadruple. Some of them are in a different shape prod equal to prod t 6 instead of you know some temporary equal to prod plus t 6 and temporary equal to prod equal to temporary A. Similarly, here so this is slightly already improved version in other words local optimization has been applied and some code rewriting has also been done to take care of such possibilities.

So, we start from here this quadruple it uses i, there is no writing into i it only reads i, so along with the variable i in the symbol table we attach the information the first fields says live or not live. The second fields says what is the last use of that variable and the third field says either next use or the number of the quadruple or says no next use.

So, in this case the variable i is a programmer define variable, so it is live even after exit from the basic block the last use of the variable i is eleven self-quadruple and then there is no next use of the variable because the basic block ends here. So, when then we go to the previous quadruple i equal to i plus 1, so here i is used on left hand side and also the right hand side. So, when we say i is live that means it must be used it is being assigned here and the assigned value must be used later it is indeed being used later. So, the variable is live this can be in ford very easily by looking at the symbol table for i we are updating it.

So, I was actually live and of course, it will be live even now because it is a programmer define variable and the last use was 11. So, if the last use information was 0, then this would have been the first definition of i, but the last use information says 11. So, i value which is defined here will be used in 11, so that is why this is the variable is live the last use is ten so that is the self-quadruple, the right hand side. Then the next use is 11 that is this quarter pal, so we have updated the information of this, so how did we say new 11 that was picked up from l u 11 and copied into this place.

Then this quadruple I will explain just this to show you the procedure prod equal to prod plus t 6. So, there are two variables prod and t 6 for prod, it is live because it is a programmer defined variable which is defined here. Then you know possibly use later the last use of the variable is 9. So, that is right hand side part which is here and there is no next use for the variable within the basic block because this is the last definition and it is not used again the variable t 6 is a temporary.

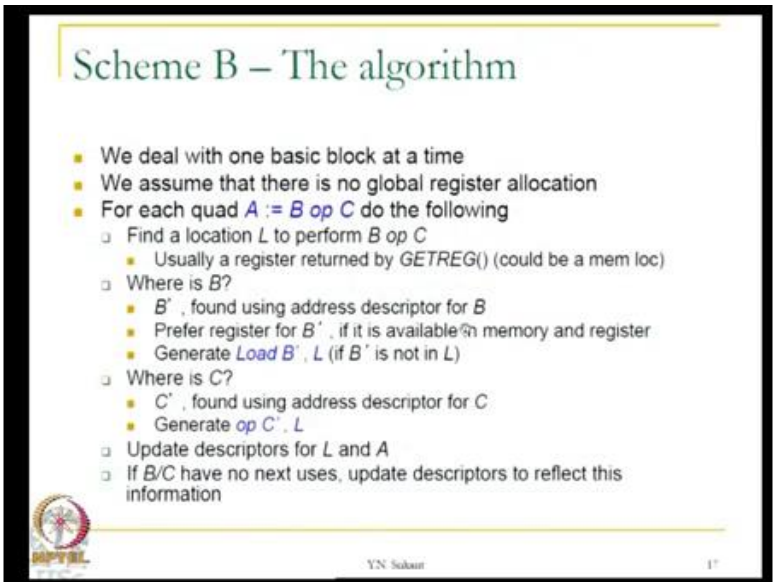
So, this is the first use of that variable and there is no next use again, so this is not live and then from you know when I say first use when we are in the scanning process. It is not live, because there are no more uses here last use nine, that is this basic block and there is no next use of t 6 within the basic block itself.

So, this continues so let us look at this quadruple at address 5 it is t 3 equal to t 2 bracket t 1 indexed assignment. So, t 3 is now you know is defined here, so then you know it is the livens end here right because its being defined here and then the last use is 0 because it is defined here and then the next use is 8 that is because t 3 is being used here. So, from this definition the usage is up to this point so that is the next use 8, sorry that is the next use 8.

So, we say last use is 0 because t 3 is defined here, so the whatever value it had is destroyed where as in this case there was a usage here that is why the last use of set as 9. So, whenever it is a defined definition t 6 equal to t 3 star t 5 l u will be 0, so here also then here also and here also here as well here as well the next use information tells you where that particular variable will be used. So, t 2 and t 1, so t 2 is not live because you know it is not used anymore and then last use was 5. So, that is this itself and then there is no next use for t 2, similarly for t 1 it is not live last use was self-quadruple, but there is a next use in quadruple 7.

So, that is this use, so this can be picked up from the symbol table for t 1 itself, so the symbol table for t 1 here since last use seven, so that would have picked up and copied here. So, this is the way in which we compute the next use information by doing a backward traversal of the quadruple array.

(Refer Slide Time: 34:11)



## 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 17

So, now we have the next use information already computed, now we are ready to discuss the algorithm for code generation. So, we as already mention we deal with one basic block at a time and we assume that there is no global register allocation. There is a local register location, which is called, we are going to discuss that in a few more minutes. So, here is the code generation algorithm for each quadruple A equal to B op C do the following first of all we need to find a location to perform the operation B op C say that is l.

How we find l, this is returned by the function call garter, it could be a register or it could be a memory location, but a preference is always given to a register. We will see details of GETREG later so let us assume that allocation either a register or a memory location has been returned by GETREG and that is where the operation B op C will have to be perform now where is B, B is in a place B prime. If we look up the address descriptor for B, it will tell us where B is to begin with when the generate code for the first quadruple in the basic block.

All these descriptors are empty, so the variable B will not even be found anywhere you know, it will be only in the home location that is the only thing we can say. So, always at the beginning all the variables will be in their home locations and all the register would be empty, but as we go on generating code very soon. We will see that some of the variables will be found in registers, so we find the place B prime or the location B prime where B is present and obviously we prefer the register for B prime.

If it is already available both in memory and register suppose B prime is not in l, so l is the place where we perform B op C. So, we generate the instruction load B prime comma l, so its moves the contents of B prime to the location l and gets it ready to perform the operation op along with another variable C the same thing will have to be done to C as well.

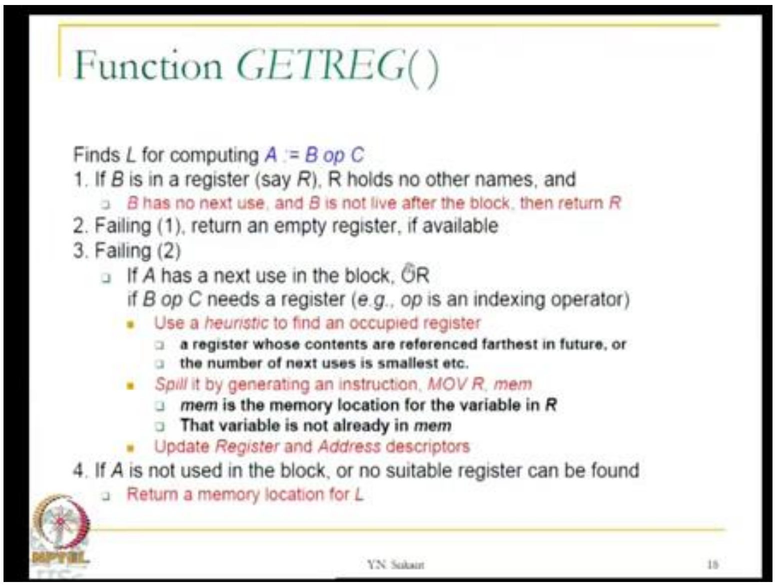
So, we need to answer the question where is C we find the place C prime using the address descriptor for C and the same ruled holds prefer a register if it is available. Now, we are ready to generate the instruction op C prime comma l, so c prime is the place where C is present l is the place where B is present. If it was not present, we have moved it and now the location l will also have the result after the execution of this instruction. So, that is the translation which is generated for A equal to B op C, but after this we have still not you know done anything to the descriptors, so the descriptors for l and a must be updated.

So, if l was a register we have to update the register descriptor if l was allocation we must update the address descriptor the same holds for a as well. So, remember we there a is not in the picture anywhere here we have not generate any instructions to move the result into a not yet that is why this updating the register descriptor for A becomes very important assume that l is a register most of the time it is. So, let us assume that l is a register, now some register are contains the value at execution time after this instruction is executed, so that is the value of A.

So, we are going to associate R with a in both the descriptors in the register descriptor, we will say R and A are associated and in the address descriptor. We will say A and R are associated, so the remember A’s home location is not updated it is now just present in a register.

So, we must keep in the mind we want to empty the register will have to move the generate an instruction to move the value of the register into the home location for a the second updating that we need to do are the book keeping that we need to do. If B or C B and or C have no next uses, so next uses is available in the symbol table. So, if there is no next use that means we must update descriptors to reflect this information so that there is no need to you know keep the values of B and C in the registers that they are supposed to correspond to, so we can discard that value.

(Refer Slide Time: 39:50)



## 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 found
    - Return a memory location for *L*

Y.N. Srikant 16

So, what is left is to discuss the function GETREG, so its finds the location l for computing a equal to A op C first op possibility if B is in a register say R and R holds no other names. This is important if R corresponds to many variables, then we cannot return the register R because we would we destroying more than one variables.

So, we do not do that secondly B has no next use and B is not live after the basic blocks, so that means the value of the B will be used in this instruction and then it is useless so in such a case if b is available in a register. We can very happily used that register to store the final result also this is the best possible situations. So, return R, suppose this is not possible failing one return an empty register if available, so we will have a stack of rather a set of registers available. So, return one of the empty registers, but if we have used up all the registers then two is also not possible. So, failing two if A has a next use in the block that means a will be used again or if B op C needs a register.

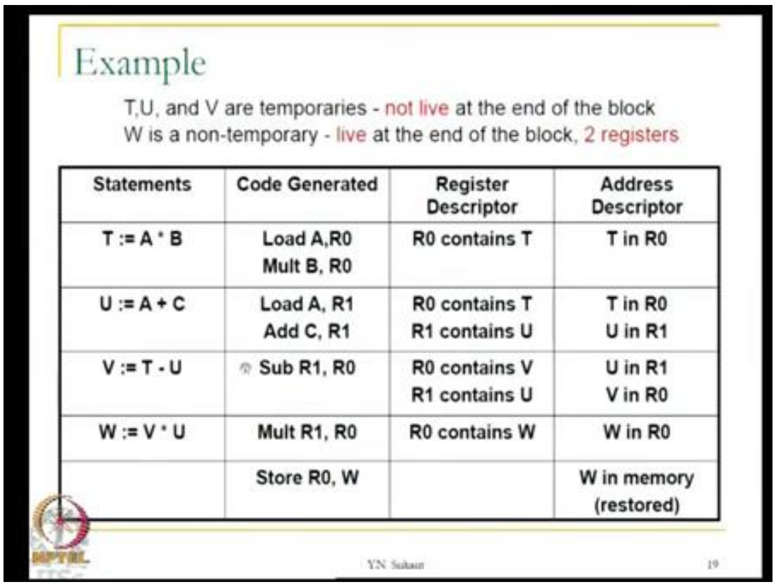
So, you cannot do without a this operation op cannot be done without a register for example, op is an indexing operators so we have a equal to B square bracket C. So, in such a case the op is an a indexing operator which can be executed only using a register. So, in both cases we must compulsorily assign a register to l and there are no registers which are free. So, we must use a heuristic to find an occupied register and then empty it, so what possibilities exist in empty you know in finding an occupied register one possibility is a register whose contains are referenced furthest in a futures.

So, the basic block has many instructions and A, at particular point, you know the register that we want to free may be used only much later. So, we can say may be there will be a free register at that time, so let us not worry too much and free that registers, so this is the heuristic to find an occupied register another possible heuristic is the number of next uses is the smallest. So, look at all the registers see which one has the smallest number of next uses smallest number of next uses. So, then you know we can realize that particular register which has a smallest number of next uses.

So, that means if we realize that register which has a smallest number of next uses the number of loads for that particular variable will be kind of minimum. So, these are the two popular heuristics which are used in local register location, so we are found a register now, but it contains some valid value. So, we must spill it by generating an instruction move R comma MEM, where MEM is the home location for the variable in R, so and that variable obviously should not be already available in MEM. Then there is no need to generate this instruction, then obviously we must update the register and address descriptors to say that this value which was corresponding to this R is you know now will hold a different variable and so on and so forth.

If A is not used in the block or no suitable register can be found then return a memory location for l. So, this is possible provided the architecture permits, you know memories operands also as instructions in the case of risk architectures. You cannot have any other instruction accept load and store with memory operands, so we must compulsorily use a register, then we use when we whatever op operation is to be performed, if there is not the case, then we return a memory location.

(Refer Slide Time: 44:53)



## 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<br>Mult B, R0 | R0 contains T | T in R0 |
| U := A + C | Load A, R1<br>Add C, R1 | R0 contains T<br>R1 contains U | T in R0<br>U in R1 |
| V := T - U | Sub R1, R0 | R0 contains V<br>R1 contains U | U in R1<br>V in R0 |
| W := V * U | Mult R1, R0 | R0 contains W | W in R0 |
|  | Store R0, W |  | W in memory<br>(restored) |

Y.N. Srikant 19

Here, is a simple example of how to generate code for a small basic block the variables t U and V are temporaries they are not live at the end of the basic block w is a programmer defined variable or non temporary and it is live at the end of the basic blocks. We have two registers available to us, so at the end of the basic block, we must store w back into its home location. So, the end we can ignore the values of t U and V which are present in registers at the end of the basic block.

The first quadruple is A T equal to A star B the basic block, you know we are just at the beginning of the basic block. So, both the registers are free, they do not contain any value so obviously the only possibility is to generate a return the register R 0 that is an empty register because neither A or nor B or in registers. So, because we returned R 0 and it does not contain a we generate A instruction load A comma R 0. Then mult B comma R 0 assuming that you know memory instructions R possible, now the register and address descriptors operandly read R 0 contains T and T in R 0.

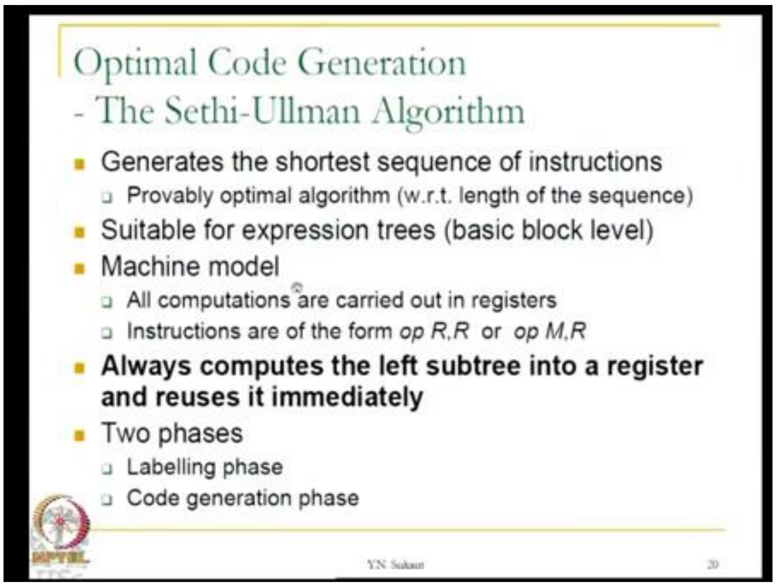
So, T is in R 0 and R 0 contains t the next instruction is U equal to a plus C, now we destroyed A. So, here a was loaded into register R 0, but then we destroyed R 0, sorry not A. Now, we need to load it again from the memory location corresponding to A to the register R 1 because R 0 which contain t has a next use, it will be use later. So, we do not want to realize R 0 and there is an empty register, we will realize R 1 A was not present in any register neither was C.

So, we realized R 1 since a was not present in R 1, we moved A into R 1 by this instruction this was generated and then add C comma R 1 that is the operation. Now, the descriptors read as usual R 0 contains T and T in R 0 from the previous instruction and then now R 1 contains U and U in R 1. So, now we have run out of registers no more let us see what happens v equal to t minus u fortunately t is in a register and t is not live at the end of the basic block it is a temporary.

So, we can use the register of t as the register for the result, so that is R 0, so sub R 1 comma R 0 puts the value of v into the register R 0. So, this is the instruction which is generated, now the descriptors are updated to reflect these R 0 contains V U in R 1 V in R 0 R 1 contains U. So, this is what it iterates the last instruction in the basic block is W equal to V star U, so v is in a register that is R 0, U is in the other register that is R 1, so that information is obtained from the descriptors automatically.

So, this addressed shift tells us that it is, so happened that both V and U can be dispensed it at the end of the basic block. So, we do not need those registers, they are not live at all, so we can the register of we can be used to store the value of W. So, mult r 1 comma R 0 is generated, now R 0 will contain the result and it assign the which is actually going to contain the value of W. So, the descriptors now reflect that as well at the end of the basic block since W is live we must generate in instruction to store R 0 into its home location W. So, that is restored so that ends the goes basic block code generation, so this is how we generate code.

(Refer Slide Time: 49:36)


## 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

Y.N. Srikant 20

So, let us see better schemes of code generation, now the basic block code generation that we discussed does not guarantee any optimality. So, it simply says whenever something is available in a register let us try to keep it in the register as long as possible so there is some reuse, but the literature also explain you know carries descriptions of what are known as optimal algorithms. So, we are going to discuss two of these one is the Sethi Ulman algorithm which is somewhat restricted and the other is the dynamic programming based algorithm which is more general.

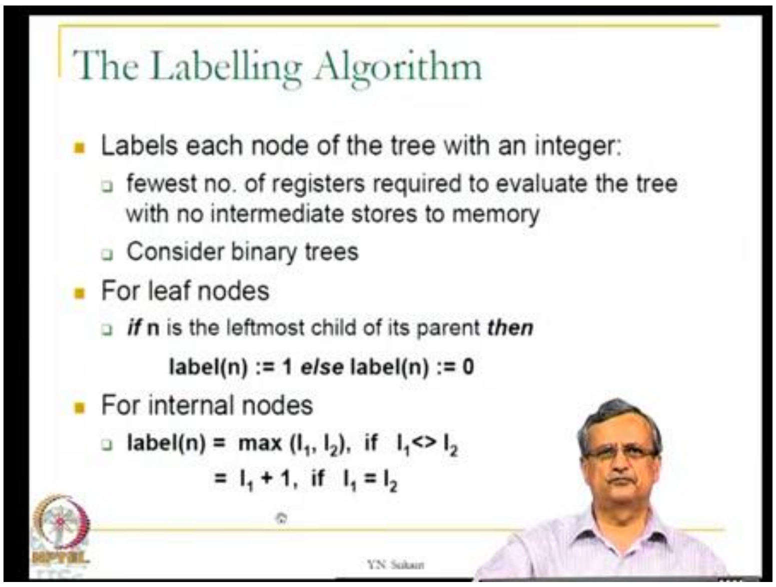
So, let us begin our discussion on this Sethi Ulman algorithm, the Sethi Ulman algorithm generates the shortest sequence of instructions for a particular machine model and it possibly provably optimal. So, with respective to the length of the sequence, so the optimality is with respective to the number of instructions that the algorithm generates for a particular program. Again, this is at the basic block level and the basic block is assume to be an expression tree. So, if it is a dag it does not work, so we need to break the dag into trees and then apply the Sethi Ulman algorithm on each tree separately.

So, what is the machine model that is used here the machine model says all the computations are carried out in registers. So, in other words computation itself cannot use any registers, but there are some exceptions. So, instructions are always of the form op R comma R or op M comma R, so this is the these are the only two possibility as for as the instructions are concerned.

So, but the major most of the major competition are all carried out in registers, so this op can only be a load or store. So, we cannot really have a plus or minus or star as this particular op it has to be this op. So, we will have to do that it always computes the left subtree into a register and then reuses it immediately.

So, this is very important the left sub tree must be computed into a register and then reuses it immediately this is required even for the proof of correctness and optimality etcetera. So, we will see how to you know liberal make the instruction formats a little more liberal later, but in this case they must be of the form of op R comma R op M comma R. There are two phases in this particular algorithm, the first phase is called the labelling phase and the second phase is called as the code generation phase.

(Refer Slide Time: 53:10)



## The Labelling Algorithm

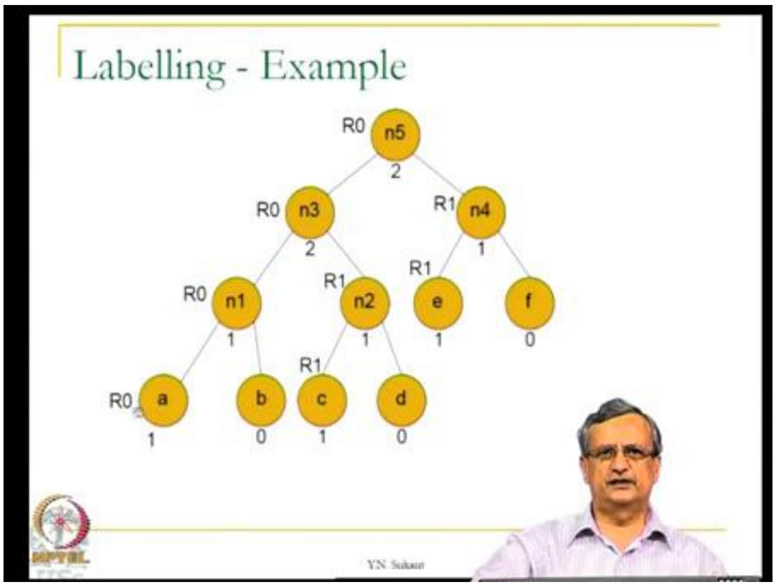
- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - ***if*** **n is the leftmost child of its parent *then***

    **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 $l_1$ = $l_2$**

What is the labelling phase, the labelling phase is very important it tries to it computes the minimum number of registers required to evaluate the tree with no intermediate stores to memory.

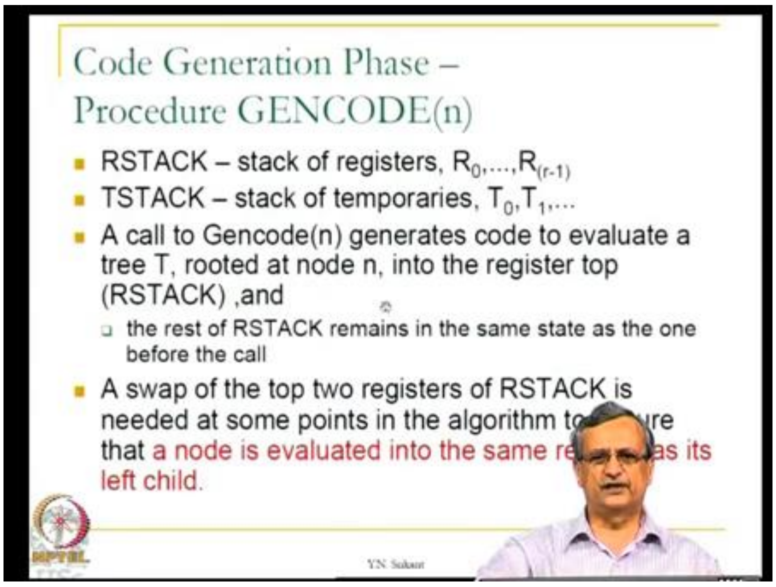
(Refer Slide Time: 53:24)



## Labelling - Example

So, let me show you a picture, so if you take this particular tree expression tree, the labelling algorithm computes a number called you with value two as the min reg value of this particular tree. The implication is you require two and definitely not more or not less than two registers to evaluate this tree and how there will be no stores into memory locations at any one of the descend ends of this particular tree. Every result will be stored in a register itself.

(Refer Slide Time: 54:06)



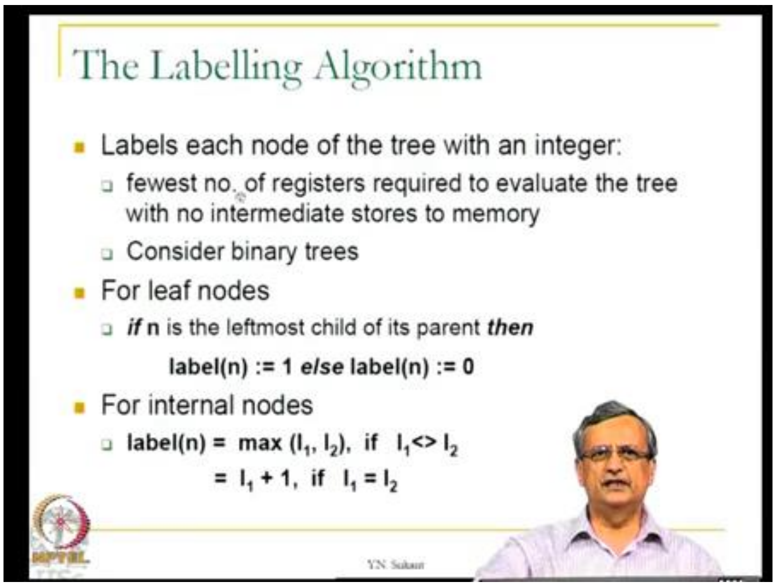
## Code Generation Phase – Procedure GENCODE(n)

- 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 [illegible]re that a node is evaluated into the same re[illegible]as its left child.

Y.N. Srikant

So, this is the significance of this particular labelling algorithm.

(Refer Slide Time: 54:12)



## 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***

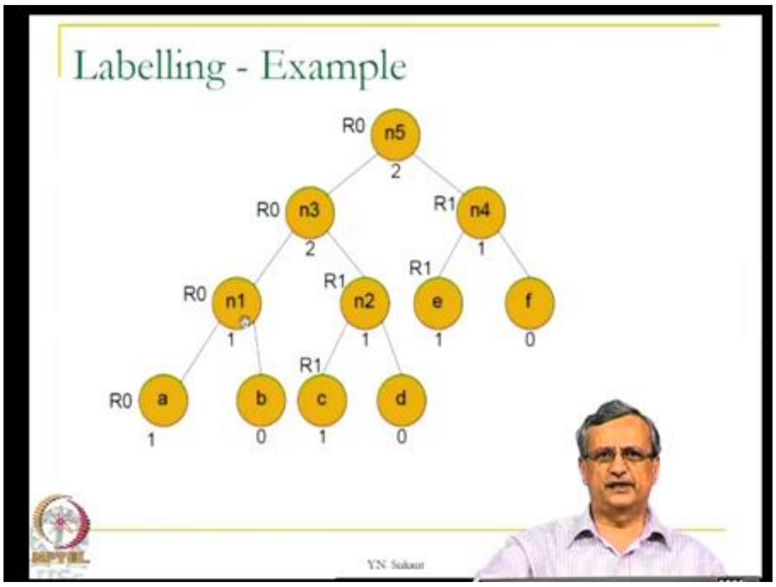
    **label(n) := 1** ***else*** **label(n) := 0**
- For internal nodes
  - **label(n) = max ($l_1$, $l_2$), if $l_1$<> $l_2$**

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

YN Srikant

So, what does it say it says a if n is the leftmost child of its parent then the label of that node is 1, otherwise label is 0 and for internal nodes. It takes if i 1 is not equal to l 1 is not equal to l 2, then label of n is the max of l 1 comma l 2 and if they are equal its simply increments by its 1, so l 1 plus 1 if l 1 equal to l 2.

(Refer Slide Time: 54:45)



## Labelling - Example

YN Srikant

Let me explain the algorithm with this example. So, we have this tree so for this parent this is the left most leaf, so this becomes A 1 this becomes A 0, similarly this is the left most child, so this becomes A 1 this becomes A 0.

Similarly this is the left most child this becomes a one this becomes a zero again this becomes A 1 this becomes A 0 this are the leaves for this parent this l 1 is not equal to l 2. So, the max is one similarly, for this it is one and for this also it is one for this parent l one equal to l 2, so this is l 1 plus, so this is 2 and for this parent again l 1 not equal to l 2. So, this is max, so this is 2, so 2 is the min reg value, let see how the code really evaluates this tree. So, this is very simple I can load this value into a registers, so it value goes into A goes into R 0, that is instructions and then this can be in memory.

So, I can say op of this R 0 comma B, then the result will be in R 0, similarly this result will be computed into R 1. Now, I have 0 result in R 0 and R 1 and the final result of n three will be in R 0, I can compute this in R 0. Now, the register R 1 is free, so I take that here, I will compute E into R 1, then n 4 into R 1 by op R 1 comma F and then finally, R 1 and R 0 contain these two operands. So, this result can be computed into R 0 by op R 0 comma R 1. So, this the way I can compute the tree into a register without intermediates stores into memory locations with just two registers will stop here and continue the lecture next time.

Thank you.