

## **UNIT 5 : CODE GENERATION**

### **Introduction:**

- This is the final phase in the compiler model that takes as input the Intermediate Representation (IR) produced by the front end of the compiler along with the relevant information in the symbol table and produces the semantically equivalent target program.
- The target program must preserve the semantic meaning of the source program and be of high quality in terms of space and execution.
- The compiler that need to produce target programs must include optimization phase prior to code generation or during code generation.
- The code generation has to perform three major tasks that are identified as Instruction selection, register allocation and assignment and instruction ordering.

### **Difficulties and issues in Code Generation:**

- Deciding what target machine instruction to generate for a particular computation that implements the IR statements i.e., *Instruction Selection*.
- Deciding which order of computation should be selected to schedule the execution of statements. i.e., *Instruction ordering*. ( order of computation that uses fewer registers to hold the intermediate results).
- Deciding what value to keep in which registers and optimal assignment of registers to variables. i.e., *Register allocation and assignment*.

### **Target language and machine:**

- Familiarity with the target machine and its instruction set is a prerequisite for generating a code for the source language.
- We shall assume the Target language as the assembly language for a simple computer DEC-PDP –II.
- The target computer models a three address machine with load and store operations, computation operations, jump operations and conditional jumps.
- The underlying computer is byte addressable machine with a general purpose registers R0,R1.....Rn-1.
- We shall use a limited set of instructions and assume all operands are of types integer.
- Instruction consists of an operator, followed by a target, and a list of operands i.e., OP result,op1,op2

We assume the following kinds of instructions.

➤ Load operations : LD dst, addr

Example: LD r, z and LD r1,r2

➤ Store Operations : ST X, r

➤ Computation operations : OP dst, src1, src2

[Type text]

where OP is a operation code like ADD,SUB...n and dst scr1, src2 are memory locations.

Example: SUB r1,r2,r3

- Unconditional Jump : BR L
- Conditional jump : Bcond r, L

Example : BLTZ r, L

We also assume the standard addressing modes.

Examples to generate code for the three address statements assuming all variables are stored in memory locations :

1.  $x = y - z$

LD R0, y

LD R1, z

SUB R0, R0, R1

ST x, R0

2.  $b = a[i]$  : here a is an array whose elements are 8 byte values and  
elements of array a are indexed starting at 0

LD R1, 1 // R1 =1

MUL R1, R1, 8 // R1=R1+8

LD R2, a(R1) // R2 = contents( a + contents(R1))

ST b, R2 // b = R2

3.  $a[j] = c$

LD R1, C

LD R2, j

MUL R2,R2, 8

ST a(R2), R1

4. If  $x < y$  goto L

LD R1, x

[Type text]

LD R2, y

SUB R1, R1, R2

BLTZ R1, M

### **Program and Instruction cost:**

- Determining the actual cost of compiling and running a program is a complex problem and finding the optimal target program for a given source program is a undecidable problem.
- In general we use heuristic technique that produces good but not necessarily a optimal target program and assume the followings for the target language instructions and their associated cost
  - Instruction LD R0,R1 has a cost one because no additional memory reference
  - Instruction LD R0, M has a cost two because memory reference.
  - Instruction LD R1, \*100(R2) has a cost three because two memory references.

### **Code optimization Principles:**

#### **What is code optimization ?**

- Code optimization refers to some technique of transformation employed by the compiler in order to get the optimal object program which is most obvious for a given source program.
- Optimal program refers to efficient code measured in terms time (Faster) and space (less)

#### **How to measure the quality of the object program?**

- The quality of the object program is basically measured in terms of size and running time. Also the machine on which the object program is executed.
- For large computers running time is important and for small computers memory size is particularly important.

### **Possible Sources of optimization:**

1. Detecting patterns in the program and replacing these patterns by equivalent and more efficient construct.

Example could be multiplication operation may be replaced by addition and division operation may be replaced by subtraction

2. The richest source of optimization is in the efficient utilization of registers and instructions from the instruction set of the machine.

This is achieved during the code generation by maintaining the register descriptor and register descriptor.

3. An identification of common sub expression and replacement of run time computation by compile time computation.

[Type text]

DAG is used to catch the common sub-expression.

For example  $A[i+1] = B[i+1]$  statement could be replaced by  $T=i+1$  and  $A[T]=B[T]$

or  $a = b * c$

$x = b * c + 5$

could be replaced by  $t1 = b * c$

$a = t1$

$t2 = t1 + 5$

Examples on Finding local Common sub expression and elimination

Example -1

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

Note : variable  $b$  is not live on exit of the block

Example – 2

$d = b * c$

$a = a + b$

$b = b * c$

$a = e - d$

Note : 1. Only the 'a' is live on exit of the block

2.  $a$ ,  $b$ , and  $c$  are live on exit of the block

#### 4. Compilation time computation or folding

It is a process of executing the source program at compile time where the operand values are known before runtime. It is also called as folding :

Example :

$l = l + 1$

$l = 3$

[Type text]

$A = 4 * I$

→  $T1 = I + 1$

$I = T1$

$I = 3$

$T2 = 4 * I$

$A = T2$

→  $T1 = I + 1$   
 $I = T1$  } Dead code  
 $I = 3$   
 $A = T2$

Note: I and A are live on exit. Also all temporaries are dead on exit

### 5. Loop Unrolling:

if number of iterations are constant then we can reduce the replicating the body of the loop to reduce the number of iterations.

For example :

$I=1$

While( $I < 100$ )

```
{  
    x[I]=0;  
    I=I+1;  
}
```

The above segment of code could be replaced by

$I=1$

While( $I < 50$ )

```
{  
    x[I]=0;  
    I=I+1;
```

[Type text]

```
x[l]=0;  
l=l+1;  
}
```

## 6. Code optimization by Copy propagation and dead code elimination

Copy propagation technique could be applied for assignment statement of the form  $f=g$ . Though it is not a direct optimization but it gives an opportunity to remove dead code and identification of common sub-expression.

Example 1:

$x = t1$	$x = t1$
$a[t2] = t3$	$a[t2] = t3$
$a[t3] = x$	$a[t3] = t1$

Here  $x=t1$  is a dead code and could be later eliminated.

Example 2:

$c=d$	$c=d$
$x = c+e$	$x=d+e$
$z = d+e-10.5$	$z=d+e -10.5$

Here the statement  $x = c+e$  could be modified as  $x = d+e$  by propagating the variable 'd' to it.

This creates the possibility of identifying the common sub-expression

## 7. Loop optimization:

Begin

```
prod=0;  
l=1;  
do  
Begin  
    prod=prod + a[i] *b[i]  
    l=l+1  
end Until l <= 20
```

[Type text]

End

Assumption : Consider the machine with four bytes per  
word. i.e., word length

Three address code statement for the source construct

1.  $prod = 0$
2.  $I = 1$
3.  $T1 = 4 * I$
4.  $T2 = add(a) - 4$
5.  $T3 = T2[T1]$
6.  $T4 = add(b) - 4$
7.  $T5 = T4[T1]$
8.  $T6 = T3 * T5$
9.  $Prod = Prod + T6$
10.  $I = I + 1$
11. If  $I \leq 20$  Goto 3
12. ...

First step towards loop optimization is to break or partition the TAC's into one more basic blocks.

- **What is Basic Block?**

It is a sequence of consecutive TAC statements which may be entered only at the beginning and when entered, all the statements are executed in sequence without halt or possibility of branch. i.e., if one statement is executed then all the statements of that block are executed in sequence without halt or there is possibility of branch

- **Algorithm for partitioning the TAC's into Basic Blocks**

1. Determine the set of leaders i.e., the first statements of Basic Blocks.

Rules for Leader:

- a. The First TAC statement in the Intermediate code is the leader.
  - b. Any statement which is target of a conditional statement or unconditional goto statement is a leader.
  - c. Any statement TAC statement which immediately follows a conditional goto is a leader.
2. For each leader construct its Basic Block which consists of the leader and all the TAC statements up to but not including the next leader or the end of the program.

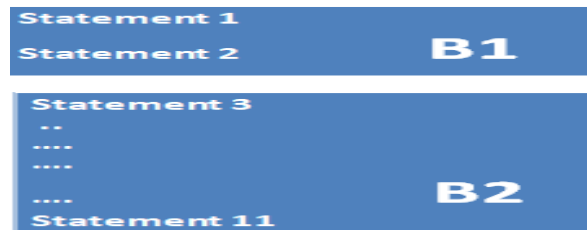
In the above example,

[Type text]

Statement 1 is a leader – By rule 1

statement 3 is a leader - By rule 2

statement 12 is a leader – By rule 3



### Construction of flow graph from basic block:

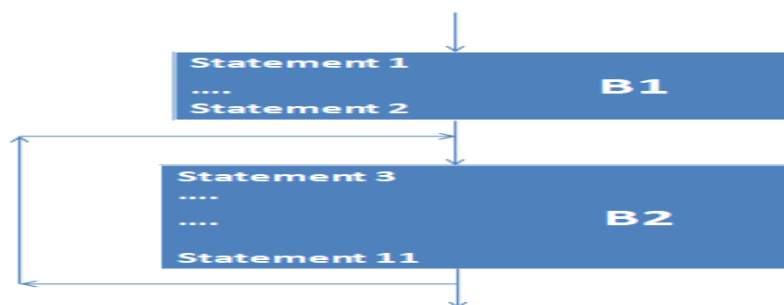
- **What is Flow graph ?**

Flow graph is a directed graph that determines the successive relationship between the basic blocks. Here basic blocks represent the nodes for the flow graph and one node is distinguished as the initial node where it is the basic block whose leader is the first statement. The edges are obtained as follows,

1. There is direct edge from the Block B1 to B2 if B2 could immediately follow B1 during the execution OR

2. There is jump from the last statement of B1 to the first statement of B2

### Flow Graph:



In the above flow graph we determine the loop. This is done by checking the loop to be strongly connected .i.e., loop is a collection of nodes in flow graph which is strongly connected i.e., from any node in the loop to any other node there is a path of length one or more wholly within the loop.

### 1. Applying Code Motion :

#### Code motion :

- The running time of the program is improved by decreasing the length of one of its loop. By doing this we may increase the length program but the number of iteration may be reduced.
- Here we assume that the loop is executed at least once.

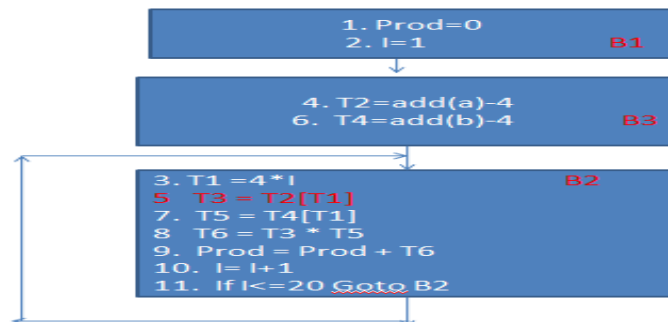


[Type text]

- We apply code motion technique where the important modification is done in the loop i.e., in this process we determine the loop invariant computations and these computations are placed before the loop
- A computations that yields same result, during the number of iterations of the Loop is called Loop invariant computations.

In the example, statement 4 and 6 are loop invariant computations and these are placed before the loop by adding new basic block.

Flow graph after applying Code motion:



## 2. Removal of Induction Variable

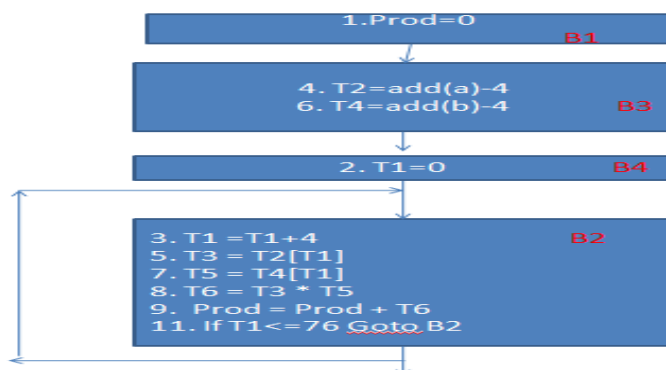
Another important optimization which may be applied to the flow graph which actually decreases the number of instructions and speeds up and reduces the loop iteration is Removal of Induction variable and Reduction in strength.

### What is induction variable ?

A variables that forms the arithmetic progression is called Induction variable.

In the example I and T1 are Induction variable since they form arithmetic progression i.e whenever  $i=1,2,\dots,20$  T1 is  $T1*I = 4,8,\dots,80$ . So the relation is determined and new statements ( when  $i=i+1$  the relationship  $T1=4*I-4$  must hold )  $T1=0$  and  $T1=T1+4$  is added with new Basic Block

Flow graph after Removing Induction variable:



[Type text]

### 3. Reduction in strength:

Last transformation in loop optimization is to apply reduction in strength where we can replace expensive operations by an equivalent cheaper operation of the target machine.

We can think of the following transformations on the loop.

$X^2$  is invariably replaced by  $X * X$  where multiplications is replaced by call operations to exponentiations routine .

Fixed point multiplication or division by power of 2 is replaced by Shift operation.

Fixed point constant Multiplication may be replaced by constant addition.

#### Sample Example:

```
for(i=1;i<=100; i++)
```

```
{  ----
```

```
  ----
```

```
    k= i*5
```

```
    printf(" K is %d ",k);
```

```
}
```

->

```
Sum=5;
```

```
for(i=1;i<100; i++)
```

```
{  ----
```

```
  ----
```

```
    k= sum
```

```
    printf(" K is %d ",k);
```

```
    Sum= sum + 5;
```

```
}
```

**Example - 2 :**For i =1 to 10 do

For j= 1 to 10 do

a[i, j]=1.0;

[Type text]

### Three Address Code statements:

1.  $i = 1 / (*(\text{base}_a - ((i * n_2) + j) * \text{width}))$   $n_2 =$  no of elements in each row , for  $i=1$  and  $j=1$  the value is 88 for width 8 bytes.
2.  $J=1$
3.  $t1 = 10 * I$
4.  $t2 = t1 + j$
5.  $t3 = 8 * t2$
6.  $t4 = t3 - 88$
7.  $a[t4] = 1.0$
8.  $J = j + 1$
9. If  $j \leq 10$  goto (3)
10.  $I = I + 1$
11. If  $i \leq 10$  goto (2)
12.  $i = 1$
13.  $t5 = i - 1$
14.  $t6 = 88 * t5$
15.  $a[t6] = 1.0$
16.  $i = i + 1$
17. If  $i \leq 10$  goto (13)

### Code generation Algorithm:

- Here we use straight forward strategy to generate assembly code from a quadruples.
- The operands in the quadruple are currently available in the register and for each operator there is a corresponding Assembly code available.
- We also assume that the computed results are available in register as long as possible.
- Symbol table initially shows all non temporary in block B as being live on exit.
- To make more informed decision about register allocation we compute the *next use* information and liveness of each name in a quadruple. Where USE is defined as follows.
- Suppose quadruple I assigns a value to 'A', if quadruple j has 'A' as an operand and control can flow from quadruple I to j along a path that has no intervening assignments to A, Then we say that that quadruple j uses the value of A computed at I
- We wish to find for each quadruple  $A = B \text{ op } C$  the next uses of A, B, and C, the algorithm must first scan the stream of quadruple to find the end of basic block . Then scan backwards to the beginning, recording for each name whether A has next a next use in the block and if not whether it is live on exit from that block.
- Suppose in the backward scan we reach quadruple  
I :  $A = B \text{ op } C$ . we then do the following.
  1. Attach to quadruple I the information currently found in the symbol table regarding the 'next USE' and 'liveness' of A, B and C
  2. In the symbol table set 'A' to 'live' and no 'next use' and set B and C to 'live' and 'next use' for quadruple i.

### Register descriptor and Address descriptor:

- **Register descriptor:**
  - To perform register allocation we use register descriptor that keeps track of what is currently in each register . Whenever we need a register we consult register descriptor.
  - Initially register descriptor shows all registers are empty and as the code generation for blocks progresses each register may hold the value of names used in the program.
- **Address descriptor:**
  - For each name in the block we shall maintain the address descriptor that keeps track of the location where the current value of the name can found at runtime. Here the location may be reg , stack or memory location
  - The algorithm uses getreg(l) function which selects registers for each memory location associated with the TAC l. This function has an access to register and address descriptor for all the variables of the basic block.

### Code generation Algorithm:

For a three address code such as  $x=y+z$  do the following

Use  $\text{getreg}(x=y+z)$  to select registers for x, y and z. Call these Rx, Ry and Rz

If y is not in Ry ( according to the register descriptor), Then generate an instruction LD Ry, y' where y' is one of the memory location for y (according to the address descriptor for y).

Similarly if z is not in Rz then generate an instruction LD Rz,z' where z' is memory location for z

Generate the instruction ADD Rx, Ry, Rz

Getreg(l) function works on the following rules:

1. If y is currently in a register, pick a register already containing y as Ry. Do not issue the machine instruction to load this register.

2.If y is not in a register, but there is a register that is currently empty, pick one such register as Ry.

3. If y is not in a register and there is no register empty, we need to pick up one of the allowable registers and we need to make it safe to use.

- Getreg(l) function works on the following rules.

1. If y is currently in a register, pick a register already containing y as Ry. Do not issue the machine instruction to load this register.

2.If y is not in a register, but there is a register that is currently empty, pick one such register as Ry.

[Type text]

3. If  $y$  is not in a register and there is no register empty, we need to pick up one of the allowable registers and we need to make it safe to use.

> Example and Trace of Code generation Algorithm :

$t = a - b$

$u = a - c$

$v = t + u$

$a = d$

$d = v + u$

Note :  $t$ ,  $u$  and  $v$  are temporaries , local to the block and  $a$ ,  $b$ ,  $c$  and  $d$  are variables that are live on exit.

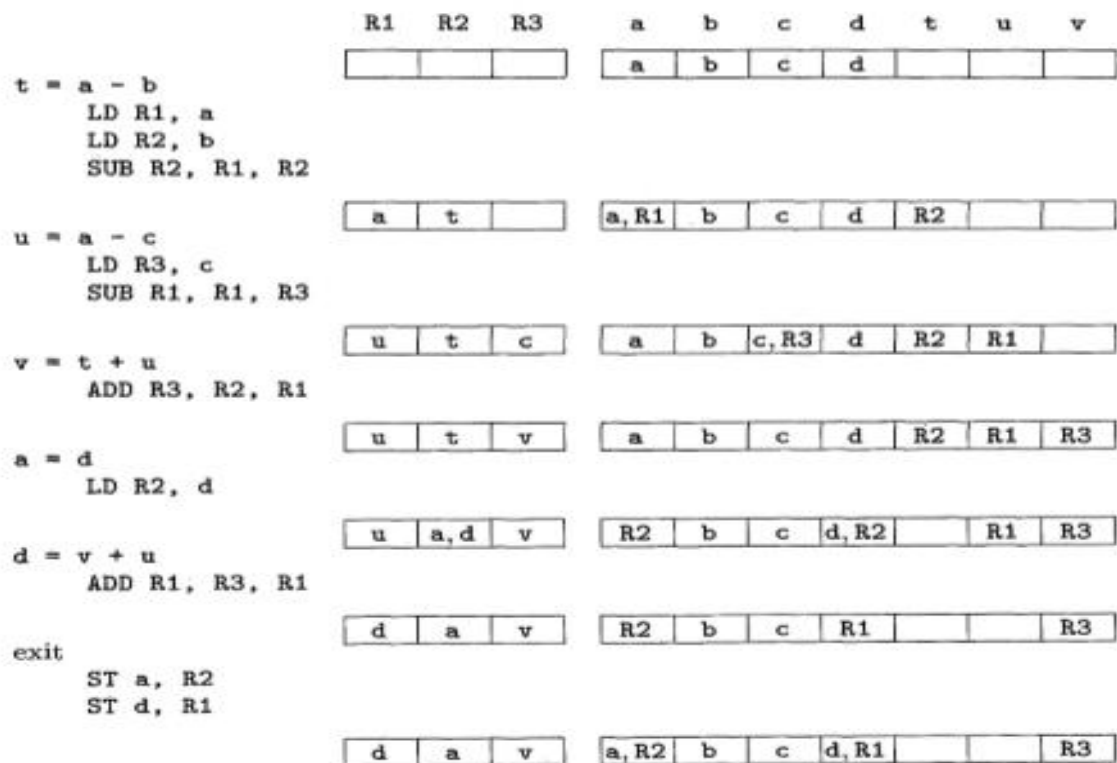


Figure 8.16: Instructions generated and the changes in the register and address descriptors

\*\*\*