UNIT-IV Code Generation.

- Issues in the design of code generator
- The target machine - Run time storage management
- Basic blocks and flow graphs
- Next-use information - A simple code generator
- DAG representation of Basic blocks
- Peephole Optimization
- Cross compiler - T diagrams.

Code generation:

⇒ Code generation is the final phase in the process of Compilation.

⇒ It takes intermediate code as an input and generates target machine code as output.

Issues in Design of code generator:-

1. Input to the code generator

⇒ The intermediate Code generated by the front end should be such that the target machine can easily manipulate it, in order to generate appropriate target machine code.

⇒ In the front end of compiler necessary type checking and type conversion Code to be done.

⇒ The detection of semantic errors should be done before submitting the input to the code generator.

⇒ The code generation phase requires the complete error free intermediate code as input.

2. Target programs

⇒ Output of code generation is target code. Typically, the target code comes in three forms such as,

i) Absolute machine language

ii) Relocatable machine language

iii) Assembly language.

Advantages:

⇒ Advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediatly.

⇒ The benefit of such target code is small programs can be quickly compiled.

⇒ Advantages of producing the relocatable machine code as output is that the subroutines can be compiled separately.

⇒ Advantage of producing assembly code as output makes the code generation process easier. It occupies small memory.

⇒ However, out of these three forms of output target code it is always preferable to have relocatable machine code as target code.

3. Memory management:

⇒ Both the front end and code generator performs the task of mapping the names in the source program to addresses to the data objects in run time memory

⇒ The names used in the three address code refer to the entries in the symbol table. The type in a declarative statement determines the amount of storage (memory) needed to store the declared identifier.

⇒ Thus using the symbol table information about memory requirements code generator determines the addresses in the target code.

4. Instruction selection:

⇒ The uniformity and completeness of instruction set is an important factor for the code generator.

⇒ The selection of instruction depends upon the instruction set of target machine.

⇒ The speed of instruction and machine idioms are two important factors in selection of instructions.

⇒ The quality of generated code is decided by its speed and size. Simply line by line translation of three address code into target code leads to correct code but it can generate unacceptably non efficient target code.

5. Register Allocation.

⇒ If the instruction contain register operands then such a use becomes shorter and faster than that of using operands in the memory.

⇒ Hence while generating a good code efficient utilization of register is an important factor.

Activities:

i) Register allocation ⇒ During register allocation, select appropriate set of variables that will resides in registers.

2) Register assignment ⇒ During register assignment, pick up the specific register in which corresponding variable will reside.

Obtaining the optimal (minimum) assignment of registers to variable is difficult.

6. Choice of evaluation order

⇒ The evaluation order is an important factor in generating an efficient target code.

⇒ Some orders requires less number of registers to hold the intermediate results than the others.

⇒ Picking up the best order is one of the difficulty in code generation.

7. Approaches to code generation

⇒ The most important factor for a code generation is that it should produce the correct code.

⇒ With this approach of code generation various algorithms for generating code are designed.

Target Machine Description

⇒ For designing the good code generator it is necessary to have prior knowledge of target machine and instruction set used for this target machine.

⇒ Specifically following assumptions are made for code generation.

i) Target computer addresses are given in bytes and four bytes form a word.

ii) There are n-general purpose registers $R_0, R_1, \ldots, R_{n-1}$

iii) The two address instruction is of the form,

$$\underset{\uparrow \text{ Opcode}}{\text{op}} \quad \underset{\uparrow \text{ Source and destination are data fields.}}{\underbrace{\text{Source} \quad \text{destination}}}$$

For instance,

Mov – Moves from souce to destination

ADD – Add souce to destination.

Source and destination are specified by register and memory locations.

Addressing modes used are as follows,

Addressing mode	Form	Address	Added Cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c+ contents(R)	1
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents(c+contents(R))	1
literal	#c	c	1

Cost of instruction:

The instruction cost can be computed as one plus cost associated with the source and destination addressing modes by added cost.

Instruction	Cost	Interpretation
Mov R_0, R_1	1	Cost of register mode +1 = 0+1 = 1
Mov R_1, M	2	Use of memory Variable +1 = 1+1 = 2
SUB 5(R_0), *10(R_1)	3	use of first constant + use of second constant +1 = 3

Example: Compute the cost of following set of instructions,

MOV a, R_0
ADD b, R_0
MOV R_0, c

Solution:

Mov a, R_0	2
ADD b, R_0	2
MOV R_0, c	2
Total cost =	6

Basic Blocks and Flow Graphs.

⇒ Basic block is a sequence of consecutive statements in which flow of control enters at the begining and leaves at the end without halt or possibility of branching.

Example:

```
t1:= a*5
t2:= t1+7
t3:= t2-5
t4:= t1+t3
t5:= t2+b
```

⇒ Define and use: Three address statement a=b+c said to define a and to use b and c.

⇒ Live and dead: The name in the basic block is said to be live at a given point if its value is used after that point in the program. Name in the basic block is said to be dead at a given point if its values is never used after that points in the program.

Algorithm for partitioning into Blocks

1. First determine the leaders by using following rules,
   - (a) The first statement is a leader
   - (b) Any target statement of conditional or unconditional goto is a leader.
   - (c) Any statement that immediately follow a goto or unconditional goto is a leader.
2. The basic block is formed starting at the leader statement and ending just before the next leader statement appearing.

Example: consider the following program code for computing dot product of two vectors a and b of length 10 and partition into basic blocks.

```
prod=0;
  i=1;
do
 {
   prod = prod + a[i] * b[i];
     i = i+1;
  } while(i<=10);
```

Solution: Three address code for the program,

```
1. prod=0
2. i=1
3. t1=4*i
4. t2=a[t1]
5. t3=4*i
6. t4=b[t3]
7. t5=t2*t4
8. t6=prod+t5
9. prod=t6
10. t7=i+1
11. i=t7
12. if i<=10 goto(3)
```

Block 1

```
1. prod=0
2. i=1
```

Block2

```
3. t1=4*i
4. t2=a[t1]
5. t3=4*i
6. t4=b[t3]
7. t5=t2*t4
8. t6=prod+t5
9. prod=t6
10. t7=i+1
11. i=t7
12. if i<=10 goto(3)
```

Flow Graph:-

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

⇒ The nodes to the flow graph are represented by basic blocks.

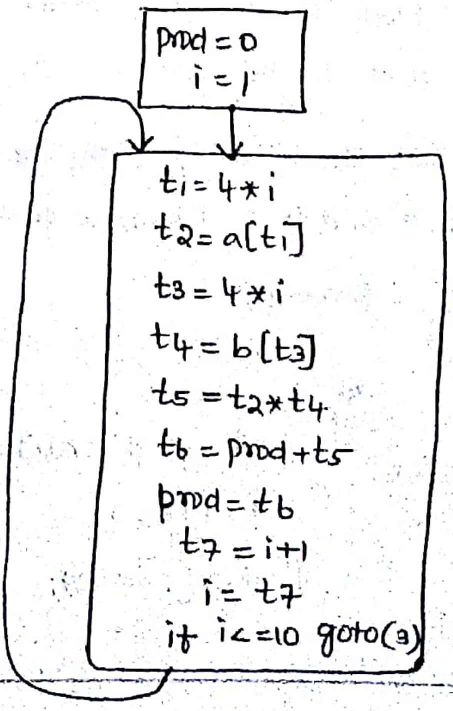
⇒ The block whose leader is the first statement is called initial block.

⇒ There is a directed edge from block B_1 to Block B_2 if B_2 immediately follows B_1 in the given sequence.

Example:

consider the three address code,

```
1. prod=0
2. i=1
3. t1=4*i
4. t2=a[t1]
5. t3=4*i
6. t4=b[t3]
7. t5=t2*t4
8. t6=prod+t5
9. prod=t6
10. t7=i+1
11. i=t7
12. if i<=10 goto (3)
```



```
prod=0
i=1
```

```
t1=4*i
t2=a[t1]
t3=4*i
t4=b[t3]
t5=t2*t4
t6=prod+t5
prod=t6
t7=i+1
i=t7
if i<=10 goto(3)
```

Loop: Loop is a collection of nodes in the flow graph such that,

i) All such nodes are strongly connected. That means always there is a path from any node to any other node within that loop.

ii) The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.

iii) The loop that contains no other loop is called inner loop.

DAG representation of basic block

The directed acyclic graph is used to apply transformations on the basic block. To apply the transformations on basic block a DAG is constructed from three address statement.

A DAG can be constructed for the following type of labels on nodes.

1) Leaf nodes are labeled by identifiers or variable names or constants. Generally leaves represent r_values

2) Interior node store operator value.

Example: consider,

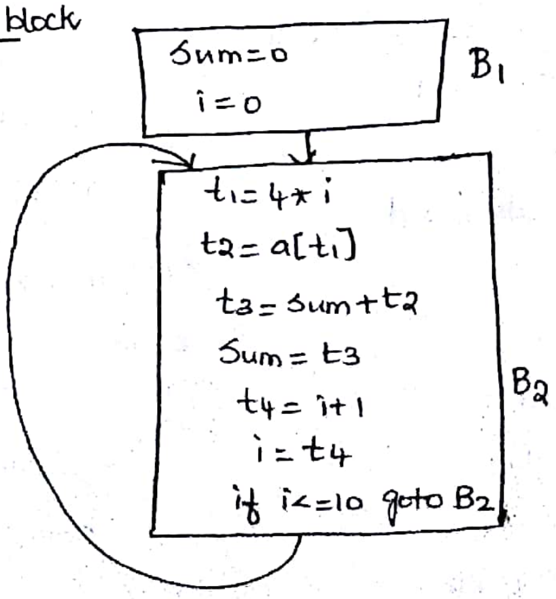
```
sum=0;
for(i=0;i<=10;i++)
Sum= sum+a[i];
```

Solution:

(i) Three address code,

```
(1) sum=0
(2) i=0
(3) t1=4*i
(4) t2=a[t1]
(5) t3=sum+t2
(6) Sum=t3
(7) t4=i+1;
(8) i=t4
(9) if i<=10 goto(3)
```

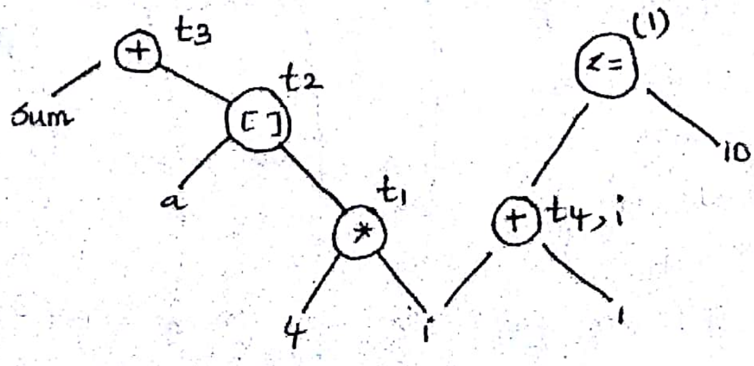
(ii) Basic block



```
Sum=0
i=0
```

```
t1=4*i
t2=a[t1]
t3=sum+t2
Sum=t3
t4=i+1
i=t4
if i<=10 goto B2
```

DAG for B2



Algorithm for construction of DAG.

We assume the three address statement could of following types.

(i) $x = y \text{ op } z$ => If y is undefined then create node(y). Similarly if z is undefined create a node(z).

(ii) $x = \text{op } y$ =>

(iii) $x = y$

Application of DAG:

1. Determining the common sub-expressions.
2. Determining which means are used inside the block and computed outside the block.
3. Determining which statements of the block could have their computed value outside the block.

Pbm: Generate DAG representation of the following code and list out the applications of DAG representation.

```
i=1, s=0
While (i<10)
s= s+a[i][i]
 i = i+1
```

Solution:

Three address code

(1) $i=1$

(2) $s=0$

(3) $t_1 = i*20$

(4) $t_1 = t_1+i$

(5) $t_2 = c$

(6) $t_3 = t_1*4$

(7) $t_4 = t_2[t_3]$

(8) $t_5 = s+t_4$

(9) $s = t_5$

(10) $t_6 = i+1$

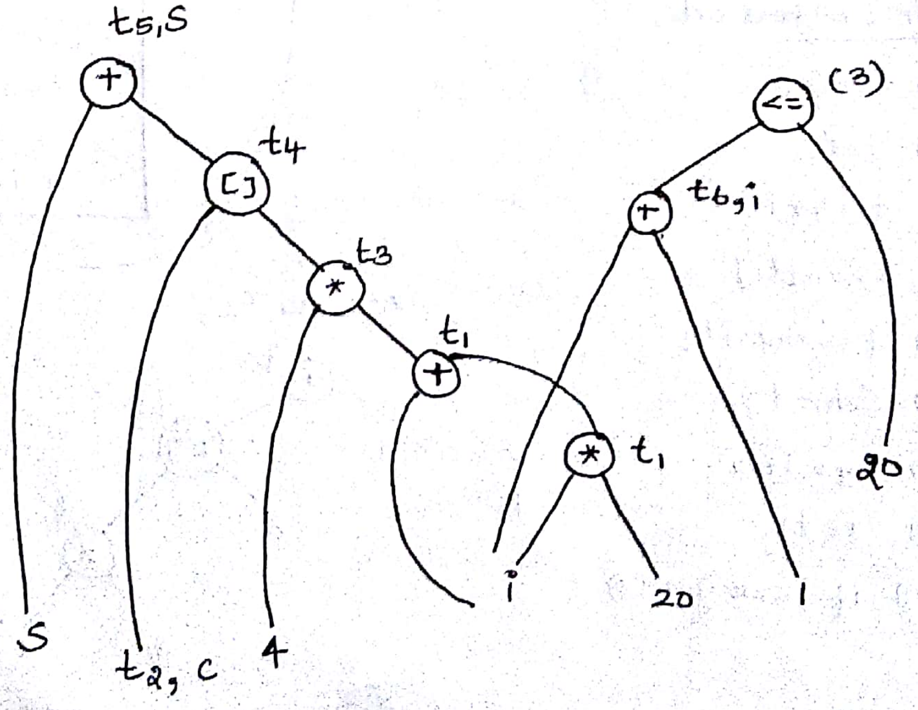
(11) $i = t_6$

(12) if i<10 goto(3)

Here a[i][i] can be computed by,

$a[i][i] = 4*(20i+i)+(base-84)$

DAG can be constructed as,



Peephole optimization

⇒ peephole optimization is a simple and effective technique for locally improving target code.

⇒ This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter and faster sequence.

Characteristics of peephole optimization:

1. Redundant instruction elimination

⇒ Especially the redundant loads and stores can be eliminated in this type of transformations.

example:

```
MOV  R0, X
MOV  X, R0
```

⇒ we can eliminate the 2nd instruction since X is already in R0

```
Sum = 0
if (Sum)
Printf ("%d", Sum)
```

⇒ Eliminate the unreachable instructions. Here if statement will never get executed hence we can eliminate such a unreachable code.

2. Flow of control optimization

using peephole optimization unnecessary jumps on jumps can be eliminated.

```
if a<b goto test              if a<b goto done
....                          ....
test: goto done       ⇒       test: goto done
....                          ....
done;                         done;
```

3. Algebric Simplification

Peephole optimization is an effective technique for algebric simplification.

$x = x + 0$ (or) $x = x * 1$, can be eliminated by peephole optimization.

4. Reduction in strength.

⇒ In order to improve the performance of the intermediate code, replace the instructions by equivalent cheaper instruction.

Example: x^2 is cheaper than $x * x$.

5. Machine idioms:

⇒ The target instructions have equivalent machine instructions for performing some operations.

⇒ Replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Example: Auto-increment. or auto-decrement addressing modes that are used to perform add or subtract operations.

Cross compiler - T diagrams.

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross compiler.