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. The position of code generator in compilation process is illustrated by following fig.

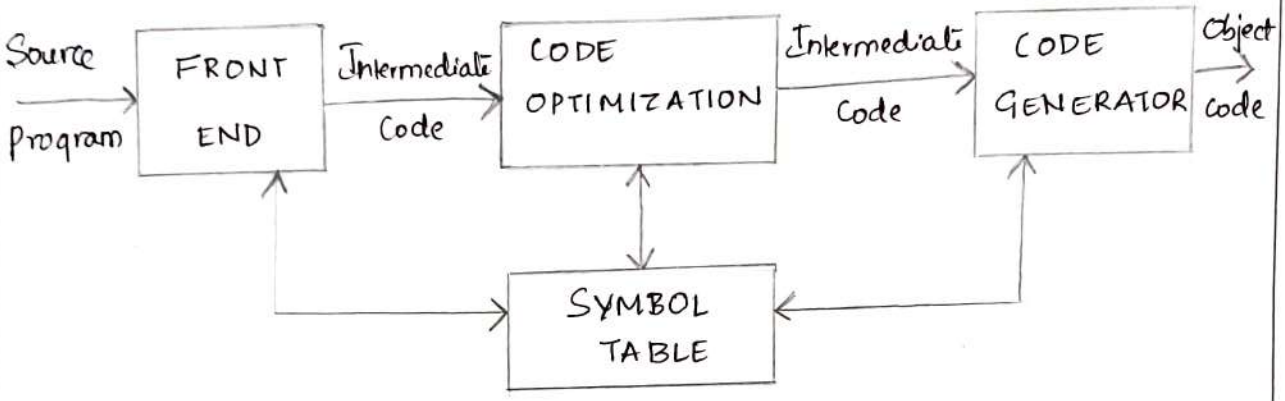


Fig:- Position of code Generator in Compiler.

Basically code generation is a process of creating assembly language / machine language statements which will perform the operations specified by the source program when they run.

Various properties desired by an object code generation phase are :-

*] Correctness :- It should produce a correct code and not alter the purpose of source code.

*] High Quality :- It should produce a high quality object code.

*] Efficient use of Resources of the Target Machine :-

while generating the code it is necessary to know the target machine on which it is going to get generated.

By this the code generation phase can make an efficient use of resource of the target machine. For instance memory utilization while allocating the registers or utilization of arithmetic logic unit while performing the arithmetic operations.

*] Quick Code Generation :- This is the most desired feature of code generation phase. It is necessary that the code generation phase should produce the code Quickly after compiling the source program.

Issues in the Design of a code generator.

The most important criterian for a code generator is that it produce correct code.

Input to the code Generator.

The input to the Code generator is the intermediate representation of the source program produced by the front end along the information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The Many Choices for IR include

*] Three-address representations such as quadruples triples and indirect Triples.

*] Virtual machine representation such as byte codes and stack machine code.

*] Linear representations such as postfix notation.

*] Graphical representations such as Syntax trees and DAG.

that all the syntatic and static semantic errors have been detected, that the necessary type checking has taken place, and that type conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that input is free of three kinds of errors.

The Target Program

The instruction set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code. The most common target-machine architectures are RISC (Reduced Instruction set Computer) and stack based.

*] A RISC machine typically has many registers, three-address instructions, simple addressing modes and a Relatively simple instruction-set architecture.

*] A CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register class, variable-length instructions and instructions with side effects

*] In a stack based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.

Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.

Instruction Selection:-

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For Example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling on some machine for Example, floating point operations are done using seperate registers.

Instruction speeds and machine idioms are other other important factors. If we do not care about the efficiency of the target program, instruction selections is straight forward.

For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.

For example, every three-address statement of the form $x = y + z$, where x, y and z are statically allocated, can be translated into the code sequence.

```
LD    R0, Y       // R0 = Y     (load Y into Register R0)
ADD   R0, R0, Z   // R0 = R0+Z  (load Z to R0)
ST    X, R0       // X = R0     (Store R0 into X)
```

This strategy often produce redundant loads and stores.

For Example, the Sequence of 3-Address statements.

$$a = b + c$$

$$d = a + e$$

could be translated into

```
LD    R0, b       // R0 = b
ADD   R0, R0, c   // R0 = R0 + c
ST    a, R0       // a = R0
LD    R0, a       // R0 = a
ADD   R0, R0, e   // R0 = R0 + e
ST    d, R0       // d = R0
```

Here the fourth statement is redundant since it loads a value that has just been stored.

The Quality of the generated code is usually determined by its speed and size.

For example, if the target machine has an "increment" instruction (INC), then the three-address statement a=a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into register, adds one to the register and then stores the result back into a.

```
LD    R0, a       // R0 = a
ADD   R0, R0, #1  // R0 = R0 + 1
ST    a, R0       // a = R0.
```

Register Allocation :-

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory.

Instructions involving registers operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into 2 subproblems.

1) Register Allocation :- During Register Allocations, select appropriate set of variables that will reside in registers.

2) Register Assignment :- During Register Assignment, pick up the specific register in which corresponding variable will reside.

Certain machine require register pairs such as even odd numbered registered for some operands and results.

Example :- The multiplication instruction is of the form

```
MUL   X, Y
```

where X, the multiplicand, is the even register of an even/odd register pair and Y, the multiplier, is the odd register. The product occupies the entire even/odd register pair.

The Division instruction is of the form

```
DIV   X, Y
```

Where the dividend occupies an even/odd register pair whose even register is X, the divisor is Y. After division, the even register holds the remainder and the odd register quotient.

For example in IBM systems integer multiplication requires register pair.

Consider the address code

$t_1 = a + b$

$t_2 = t_1 * c$

$t_3 = t_2 / d$

The efficient machine code sequence.

```
LD    R1, a
ADD   R1, b
MUL   R1, c
DIV   R1, d
ST    t, R1
```

5] Evalution Order :-

The evalution order is an important factor in generating an efficient target code. Some order 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. Mostly, we can avoid this problem by refering the order in which the three address code is generated by semantic action.

Address in Target Code :-

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

In this chapter we will assume that the target machine code is a register machine like mini computers. specifically following assumptions are made of code generation.

*] we will assume that in the target computer addresses are given in bytes and four bytes form a word.

*] There are n general purpose registers $R_0, R_1 \ldots R_{n-1}$.

*] The two address instruction is of the form

OP Source Destination.

where OP is an opcode and source and destinations are data fields.

For instance :

MOV - Moves from source to destination.

ADD - Add source to destination.

SUB - substracts source from distination.

The source and destination are specified by registers and memory locations.

The 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	*R	Contents (R)	0
Register Indirect	*C (R)	Contents (C + Contents (R))	1
Literal	#C	C	1

If we have absolute or register addressing mode we can use M or register for source or destination.

For Example :- the instruction MOV R1, M stores the contents of Registers R1 into memory location M.

For the indexed addrusing mode the address offset C from the value of Registers Ro can be written as

MOV 7(R1), M

Means it stores the value

Contents (7 + contents (R1)) to the memory location M.

The Indirect address indicates by *

For the instruction MOV *7(R0), M

It store the value contents (contents (7+ contents (R0)))
to the memory location M.

In the Literal addressing mode the source becomes constant.

For Eg:- MOV #5, R0.

By this instruction we can store the constant 5 into register R0.

Cost of Instruction :-

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

Instruction	cost	Interpretation
MOV R0, R1	1	cost of register mode = 1+0 = 1
MOV R1, M	2	use of Memory variable 1+1 = 2
SUB 5(R0), *10(R1)	3	1 + use of First constant + use of second constant = 3

The Instruction MOV R0, R1 moves the constant of Register R0 into register R1. This instruction has a cost of one because no additional memory words are required.

The instruction MOV R1, M moves the content of register R1 into the memory allocation M. The cost is two since the address of memory location M is in the word following the instruction.

The instruction SUB 5(R0), *10(R1) substracts the content (5+R0) with the contents (contents (10 + contents (R1))). The cost is three the constant 5 and 10 stored in the word following the instruction.

1] Compute the cost of following set of instructions.

```
MOV   a, R0
ADD   R0, b
MOV   R0, c
```

For instruction

```
MOV      a    ,   R0
 ↓       ↓        ↓
 1       1        0
```

Cost of Instruction | Cost of Memory location | Cost of Register

∴ MOV a, R0 → Cost = 1 + 1 + 0 = 2.

Similarly

```
ADD   R0, b
```

Cost = 1 + 0 + 1 = 2

III'ly

```
MOV   R0, c
```

Cost = 1 + 0 + 1 = 2

The cost of this instruction set is 6 because

MOV	a, R0	2
ADD	R0, b	2
MOV	R0, C	2
Total cost =		6

2] Compute the cost of following set of instructions

```
MOV   *R1, *R0
ADD   *R2, *R0
```

For the instruction

```
MOV   *R1, *R0
```

Cost = 1 + 0 + 0 = 1

```
ADD   *R2, *R0
```

Cost = 1 + 0 + 0 = 1

The cost of this set is 2 because

MOV *R1, *R0	1
ADD *R2, *R0	1

Total cost = 2

3] Consider the following code sequence

i) MOV B, R0
ADD R0, C
MOV R0, A

for instruction MOV B, R0

↓ ↓ ↓
1 1 0

Cost of Instruction — Cost of Memory location — cost of register

∴ MOV B, R0 → cost = 1 + 1 + 0 = 2

iii) for instruction ADD R0, C

↓ ↓ ↓
1 0 1

Cost of Instruction — Cost of Register — Cost of Memory Location

∴ ADD R0, C → Cost = 1 + 0 + 1 = 2

The cost of instruction set is 6 because

MOV B, A	3
ADD A, C	3
Total cost =	6

[iii] LD R0, Y
LD R1, Z
ADD R0, R1
ST X, R0

for 0 instruction LD R0, Y
↓ ↓ ↓
1 0 1
Cost of instruction, cost of Register, cost of memory location

∴ LD R0, Y → Cost = 1+0+1 = 2

for 0 instruction LD R1, Z
↓ ↓ ↓
1 0 1
Cost of instruction, cost of Register, cost of memory location

∴ LD R1, Z → Cost = 1+0+1 = 2

for 0 instruction ADD R0, R1
↓ ↓ ↓
1 0 0

∴ ADD R0, R1 → Cost = 1+0+0 = 1

for instruction MOV R0, A

↓ ↓ ↓

1 0 1

Cost of instruction Register | Cost of memory location

∴ MOV R0, A → cost = 1 + 0 + 1

The cost of the instruction set is 6 because

MOV	R, R0	2
ADD	R0, C	2
MOV	R0, A	2
Total cost	=	6

[iii] MOV B, A

ADD A, C

for instruction MOV B, A

↓ ↓ ↓

1 1 1

Cost of instruction | Cost of memory location | Cost of memory location

∴ MOV B, A → cost = 1 + 1 + 1 = 3

IIIrdly for instruction ADD A, C

↓ ↓ ↓

1 1 1.

∴ to ADD A, C → Cost = 1 + 1 + 1 = 3

for instruction ST X, R0

↓	↓	↓
1	1	0
Cost for the instruction	Cost for the memory location	cost for Register

ST X, R0 ⟶ Cost = 1 + 1 + 0 = 2

∴ Total cost of the instruction set is because

LD	R0, Y	2
LD	R1, Z	2
ADD	R0, R1	1
ST	X, R0	2

Total cost = 7

Basic Blocks and Flow Graphs :-

Basic Blocks

Our first job is to partition a sequence of three address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instruction until we meet either a jump, a conditional jump or a label on the following instruction

Algorithm :-

Partition three address instructions into basic blocks.

INPUT: A sequence of three address instructions.

OUTPUT: A list of the basic block for that sequence in which each instruction is assigned to exactly one basic block.

1. The First Three-address instruction in the intermediate code is a leader (The first statement is a Leader).
2. Any Instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Consider the source code

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i,j] = 0.0 ;
for i from 1 to 10 do
    a[i,i] = 1.0 ;
```

Assume : width = 8 , $low_1 = 1$, $low_2 = 1$, base = 0

given : $n_1 = 10$, $n_2 = 10$

$$a[i,j] = ((i * n_2) + j) * w + (base - ((low_1 * n_2) + low_2)) * w$$

$$= ((i * 10) + j) * 8 + (base - ((1 * 10) + 1)) * 8$$

$$= (10i + j) * 8 + base - 88$$

$$= (10i + j) * 8 - 88$$

1] $i = 1$
2] $j = 1$
3] $t_1 = 10 * i$
4] $t_2 = t_1 + j$
5] $t_3 = t_2 * 8$
6] $t_4 = t_3 - 88$
7] $a[t_4] = 0.0$
8] $j = j + 1$
9] if $j <= 10$ go to (3)
10] $i = i + 1$
11] if $i <= 10$ goto (2)
12] $i = 1$
13] $t_5 = i - 1$
14] $t_6 = t_5 * 88$

15] a[t6] = 1.0

16] i = i + 1

17] if i <= 10 goto (13)

Intermediate code to set a 10 * 10 matrix to an Identity matrix.

First, Instruction 1 is a leader by rule (1) of algorithm To find the other leaders, we first need to find the jumps. In this example, there are 3 jumps, all conditions at instructions 9, 11, and 17.

By rule (2) the target of these jumps are leader, there are instructions 3, 2 and 13 respectively. Then By rule (3), each instructions following a jump is a leader, those are instruction 10 and 12.

We conclude that the leaders are instructions 1, 2, 3, 10, 12 and 13.

B1

```
i = 1
```

B2

```
j = 1
```

B3

```
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
j = j + 1
if j <= 10 go to B3
```

B4

```
i = i + 1
if i<=10 goto B2
```

B5

```
i = 1
```

B6

```
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i+1
if i<=10 goto B6
```

Flow Graphs :-

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.

The Nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are 2 ways that such an edge could be justified.

*] There is a conditional or unconditional jump from the end of B to the beginning of C.

*] C immediatly follows B in the original order of the three address instructions and B does not end in a unconditional jump

we say that B is a predecessor of C , and C is a successor of B.

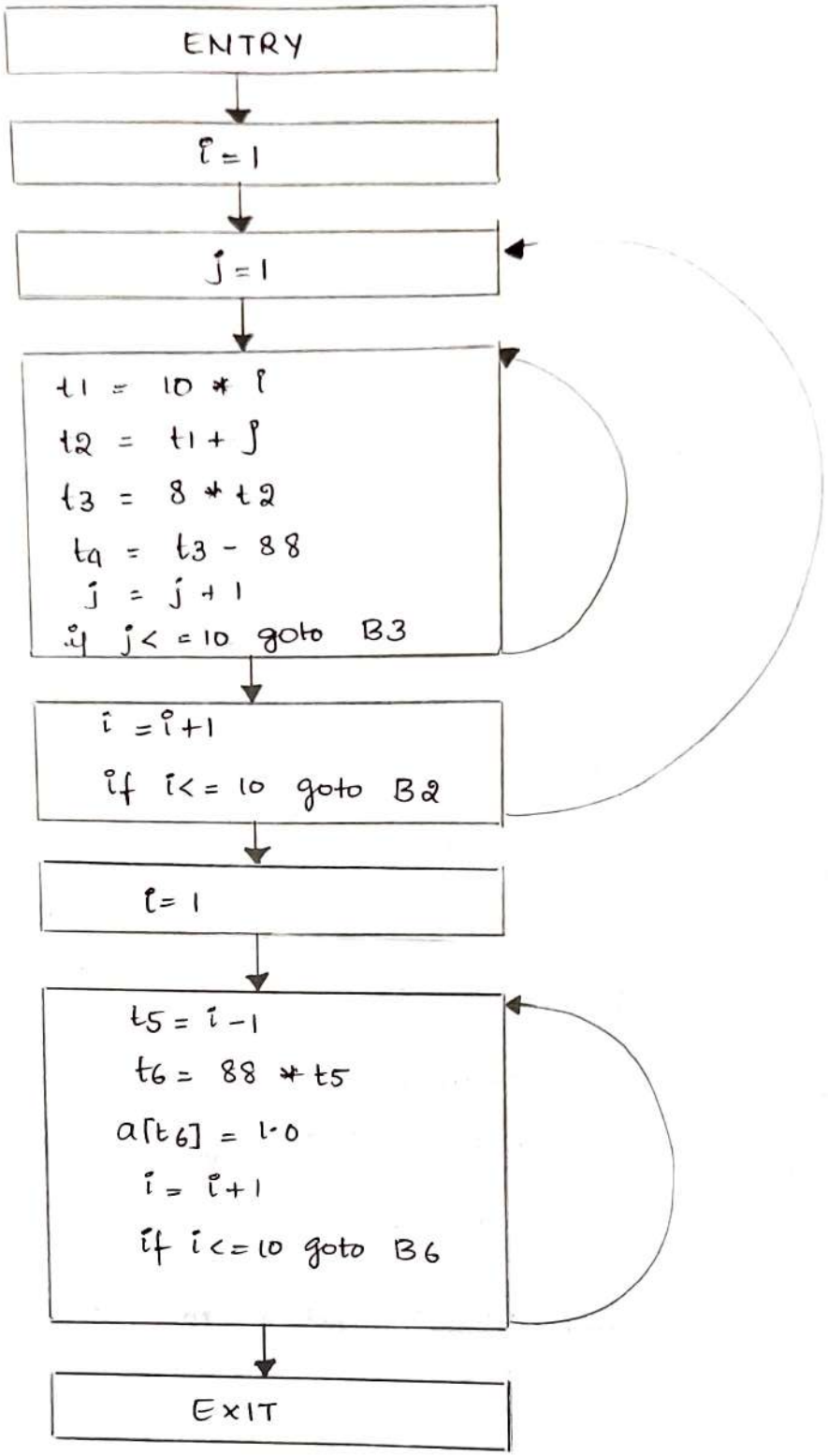


Fig:- FLOW GRAPH

Next use of Information

The next use of information is a collection of all the names that are useful for next subsequent statement in a block.

The use of a name is defined as follows.

Consider a statement

$$x = i$$
$$j = x \text{ op } y$$

that means the statement j uses value of x.

The Next-use information can be collected by making the back-ward scan of the programming code in that specific block. Suppose the three address statement is o given below.

$$i : x = y + z$$

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y and z.

2. In the symbol table, set x to not live and no next use.

3. In the symbol table, set y and z to live and the next uses of y and z to i.

Loops:-

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.

The Flow graph figure has three loops.

1. B3 by itself
2. B6 by itself
3. {B2, B3, B4}.

Q] write the three address code and Construct the basic block for the following program segment.

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

Assume :- $low_1 = 1$, $w = 8$

$$a[i] = i * w + (base - low * w)$$
$$= i * 8 + (base - 8)$$

The Three address code is as follows.

1] sum = 0

2] i = 0

3] $t_1 = i * 8$

4] t_2 = base - 8 /* address of a */

5] $t_3 = t_2[t_1]$

6] sum = sum + t_3

7] i = i + 1

8] if i <= 10 goto 3

9] goto 10

10] .....

First, find the leaders :

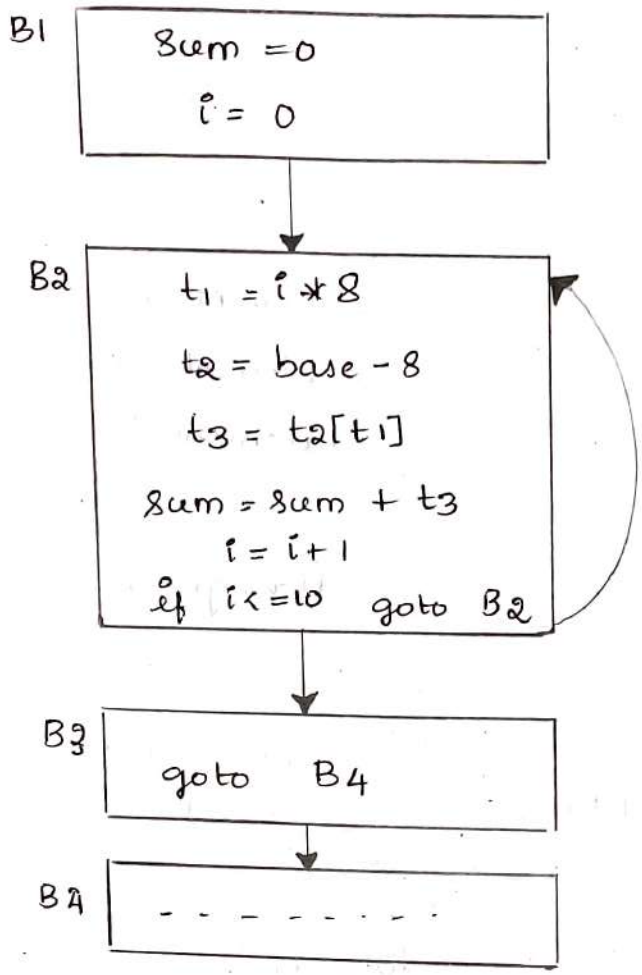
Statement (1) is a Leader

Statement (3) is a Leader

Statement (10) is a leader

Statement (9) is a leader

flow graph of the code is as follows :-



Q] Write the three address code and construct the basic blocks for the following program segment.

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

Assume :- w = 8 low = 1

a[i] = (i * w) + (base - low * w)
= (i * 8) + (base - 8)

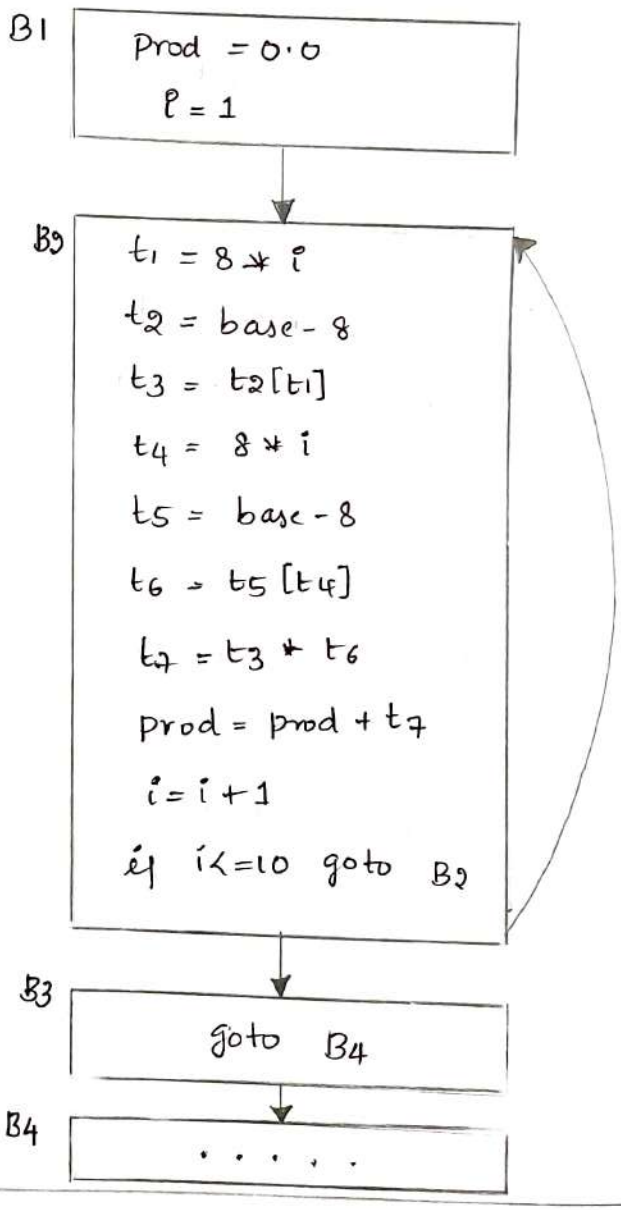
b[i] = (i * w) + (base - low * w)
= (i * 8) + (base - 8)

The Three address code is as follows

1] Prod = 0
2] i = 1
3] t_1 = 8 * i
4] t_2 = base - 8
5] t_3 = t_2[t_1]
6] t_4 = 8 * i
7] t_5 = base - 8
8] t_6 = t_5[t_4]
9] t_7 = t_3 * t_6
10] prod = prod + t_7
11] i = i + 1
12] if i <= 10 goto 3
13] goto 14
14] .....

Statement 1 is a leader
Statement 3 is a leader
Statement 13 is a leader

flow graph :



A Simple Code generator :-

In this section, we shall consider an algorithm that generates code for a single basic block.

It considers each 3-Address instruction in turn and keep track of what values are in what registers so it can avoid generating unnecessary loads and stores.

There are four principal uses of registers.

1] In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.

2] Registers make good temporaries place to hold the result of a sub expression.

3] Registers are used to hold global values that are computed in one basic block and used in other blocks.

For Example, A loop Index that is incremented going around the loop and is used several times within the loop.

4] Registers are often used to help with runtime storage management, for Example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

Register and Address Descriptors.

Our code generation algorithm considers each three address instruction in turn and decide what loads are necessary to get the needed operands into registers. After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store

Register Descriptors :-

For each available register, a register descriptor keeps track of the variable names whose current value in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progress, each register will hold the value of zero or more names.

Address Descriptor :-

For each program variable, an address descriptor keep track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location

Example :-

Three address code	Target code Sequence	Register Descriptor	Address Descriptor.
		R1 R2 R3	a b t

t = a-b	LD R1, a
	LD R2, b
	SUB R2, R1, R2
		a t	a, R1 b R2

The Code - Generation Algorithm.

An Essential part of the algorithm is a function getReg(I), which selects registers for each memory location associated with the three address Instructions I. Function getReg has access to the register and address discriptor for all the variables of the basic block.

Machine Instructions for Operations :-

For a three address instructions such as X = Y+Z, do the following :

1. Use getReg (x = y+z) to select registers for X, Y and Z Call these Rx, Ry and Rz.

2. If Y is not in Ry, then issue an instruction LD Ry, Y' where Y' is one of the memory location of Y.

3. Similarly if z is not in Rz, issue an instruction

LD Rz, z' , where z' is a location for z.

4. Issue the instruction ADD Rx, Ry, Rz

Machine Instruction for Copy statements :-

There is an important special case : a three address Copy statement of the form x = y.

We assume that getReg will always choose the same register for both x and y. If y is not already in that register Ry, then generate the machine instruction

LD Ry, y

Next store the register Ry content into a variable x

ST x, Ry

Managing Register and Address Descriptors :

As the Code generation algorithm issues loads, store and other machine instructions, it need to update the register and address descriptors. The rules are as follows.

1. For the instruction LD R, x

a] Change the register descriptor for register R so it holds only x.

b] Change the address descriptor for x by adding register R as an additional location.

2. For the instruction ST x, R change the address descriptor for x to include its own memory location.

3. For an operation such as ADD Rx, Ry, Rz implementing a three-address instruction x = y + z.

a] Change the register descriptor for Rx so that it holds only x.

b] Change the address descriptor for x so that its only location is Rx.

c] Remove Rx from the address descriptor of any variable other than x.

4. When we process a copy statement x = y, after generating the load for y into register Ry

a] add x to the Register descriptor for Ry.

b] Change the address descriptor for x so that its only location is Ry.

Generate Code for the following :-

t = a - b

u = a - c

v = t + u

a = d

d = v + u

Three Address code	Target code Sequence	Register Descriptor	Address Descriptor
		R1 | R2 | R3	a | b | c | d | t | u | v
		| |	a | b | c | d | | |

t = a - b	LD R1, a
	LD R2, b
	SUB R2, R1, R2
		a | t |	a, R1 | b | c | d | R2 | |

u = a - c	LD R3, c SUB R1, R1, R3	u | t | c	a | b | c, R3 | d | R2 | R1 |
v = t + u	ADD R3, R2, R1	u | t | v	a | b | c | d | R2 | R1 | R3
a = d	LD R2, d	u | a, d | v	R2 | b | c | d, R2 | | R1 | R3
d = v + u	ADD R1, R3, R1	d | a | v	R2 | b | c | R1 | | | R3
exit	ST a, R2 ST d, R1	d | a | v	a, R2 | b | c | d, R1 | | | R3

Generate the Code Sequence using code generation algorithm for the following expression.

$$W = (A-B) + (A-C) + (A-C)$$

write the three address code for given expression.

$t_1 = A - B$

$t_2 = A - C$

$t_3 = t_1 + t_2$

$t_4 = t_3 + t_2$

$W = t_4$

Three Address Code	Target Code Sequence	Register Descriptor	Address descriptor
		R1 R2 R3	A B C W t_1 t_2 t_3 t_4
		[ ] [ ] [ ]	A B C W

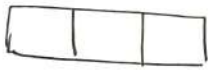
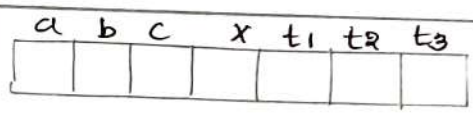
$t_1 = A-B$	LD R1, A
	LD R2, B
	SUB R2, R1, R2
		R1 R2 R3
		A t_1
			A, R1 B C W R2

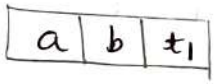
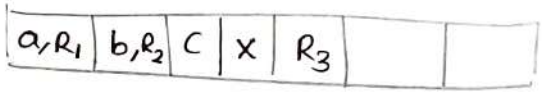
$t_2 = A-C$	LD R3, C SUB R1, R1, R3	t_2 t_1 C	A B C, R3 W R2 R1
$t_3 = t_1 + t_2$	ADD R3, R2, R1	t_2 t_1 t_3	A B C W R2 R1 R3
$t_4 = t_3 + t_2$	ADD R2, R1, R3	t_2 t_4 t_3	A B C W R1 R3 R2
$W = t_4$	ST W, R2	t_2 W t_3	A B C W, R2 R1 R3

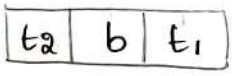
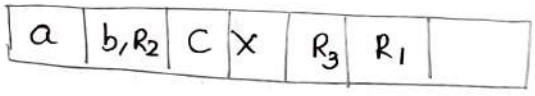
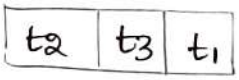
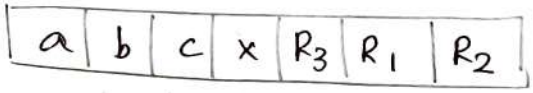
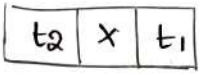
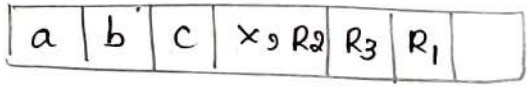
Q] Generate the code sequence using code generation algorithm for the following expression.

$$x = (a \cdot b) + (a + c)$$

$t_1 = a - b$
$t_2 = a + c$
$t_3 = t_1 + t_2$
$x = t_3$

Three Address Code	Target code Sequence	Register Descriptor	Address Descriptor
			

$t_1 = a - b$	LD R1, a
	LD R2, b
	ADD R3, R1, R2
			

$t_2 = a + c$	LD R2, c ADD R1, R1, R2		
$t_3 = t_1 + t_2$	ADD R2, R3, R1		
$x = t_3$	ST x, R2		

Optimization of Basic Blocks.

We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each block by itself.

The DAG Representation of Basic Blocks.

Many important techniques for local optimization begin by transforming a basic block into a DAG (Directed Acyclic Graph). We construct a DAG for a basic blocks as follows.

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic blocks.

2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statement that are the last definitions, prior to s, of the operands used by s.

3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.

4. Certain nodes are designed output nodes. There are the nodes whose variables are live on exist

from the block, that is their values may be used later.

The DAG representation of a basic block let us perform several code improving transformations on the code represented by the block.

a] we can eliminate local common sub expressions, that is instructions that compute a value that has already been compulid.

b] we can eliminate dead code, that is instructions that computes a value that is never used.

c] we can reorder statement that do not depend on another, such reordering may reduce the time a temporary value needs to be preserved in a register.

d] we can apply algebraic laws to re-order operands of three-address instructions, and sometimes thereby simplify the computations.

Finding Local Common Sub-expression.

Common sub expression can be detected by noticing as a new node M is about to be detected, whether there is an existing node N with the same children, in the same order and with the same operator.

For Example Consider

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

if we assume the values of $a=1$, $b=2$, $c=3$ & $d=4$. then the expressions becomes

$a = b+c$

$a = 2+3 = 5$

$b = a-d$

$b = 5 - 4 = 1$

$c = b + c$

$c = 1 + 3 = 4$

$d = a-d$

$d = 5 - 4 = 1$

The DAG for the block is as shown below.

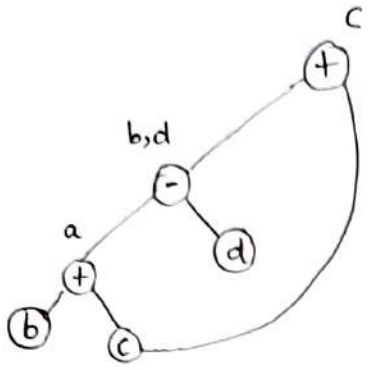


Fig: DAG for basic blocks.

The above sequence of instructions contains two common sub expression such as b+c and a-d. But for the common sub expression b+c the value of n gets changed when it reappears. Hence b+c is not a common subexpression.

But the expression a-d gives the same result in repetitive appearance and values of a and d are consistent each time. Therefore a-d is supported to be the common subexpression.

The common subexpression means the expression that are guaranteed to compute the same value

The block then becomes

$$a = b + c$$
$$d = a - d$$
$$c = b + c$$

Dead Code Elimination :-

The operation on DAG's that corresponds to dead-Code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that as no live variable attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead node.

A variable is said to be live at a point in a program if its value can be used subsequently. otherwise, it is said to be dead at that point.

For Example, Consider a part of program

```
flag = false ;

if ( flag)
  printf ( "..... ");
```

In above program statements flag is always set to false before checking for the condition. since it is always false, the print statement is never executed. Thus, the print statement is dead because it is never reached.

we can eliminate both the test and printing from the object code. Thus, reducing the compile time

The use of Algebraic Identities.

Algebraic Identities represents another important class of optimisations on basic blocks.

For Example, we may apply arithmetic identities, such as $x+0 = 0+x = x$ $\quad x-0 = x$

$x*1 = 1*x = x$ $\quad x/1 = x$

to eliminate computations from a basic blocks.

Another Class of algebraic optimisations includes local reductions in strength, that is replacing a more expensive operator by a cheaper one as in:

Expensive		Cheaper
x^2	=	$x*x$
$2*x$	=	$x+x$
$x/2$	=	$x*0.5$

The use of lower strength operator instead of higher strength operator makes the code efficient.

A Third class of related optimisations is constant folding.

Here we evaluate constant expressions at compile time and replace the constant expressions by their values.

Example :- The expression 2 * 3.14 would be replaced by 6.28.

The DAG construction process can help us apply general algebraic transformations such as commutativity and associativity.

For Example, suppose the language reference manual specifies that * is commutative., that is x*y = y*x. Before we create a new node labeled * with left child M and right child N, we always check whether such a node already exists. However, because * is commutative, we should then check for a node having operator *, left child N, and right child M.

Associative laws might also be applicable to explore common subexpressions.

For Example, if the source code has the assignments.

```
a = b+c ;
e = c+d+b ;
```

The following intermediate code might be generated.

$$a = b + c$$
$$t = c + d$$
$$e = t + b$$

If t is not needed outside this block, we can change this sequence to

$$a = b + c$$
$$e = a + d$$