Code Generation:- Code generation is a process of creating assembly language / machine language statements. It is the final phase of compiler.

Properties of Code generation phase:-

1. **Correctness:-** It should produce a correct code & don't alter the purpose of source code.
2. **High Quality:-** It should produce a high quality object code.
3. **Effecient use of resources of the target Machine:-** By knowing the target machine, the code generation phase can make effecient use of the target machine resources.
4. **Quick Code generation:-** It is most desired feature. Code generation phase should produce code quickly.

Issues in the design of a Code generator

The various issues in the design of code generator are as given:-

1. **Input to Code generator:-** The input to code generator consists of intermediate representation of the source program along with information in the symbol table used to determine runtime address. The code generation phase proceed on the assumption that its input is free of errors.
2. **Target Programs:-** The output of code generator is the target program. This output can be absolute machine language, relocatable machine language or assembly language. The adv. of absolute machine language output is that it can be placed in a fixed location in memory and

implementing executed.

Producing a relocatable machine language output may allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Producing an assembly language program as output makes the process of code generation easier.

(3) Memory Management :- Mapping names in the source program to addresses of data objects in runtime memory is done co-operately by the front end & the code generator.

(4) Instruction Selection :- The nature of the instruction set of the target machine determines the difficulty of the instruction selection.

- The uniformity & completeness of instruction set are important factors.
- Instruction speeds and machine idioms are other important factors.
- The quality of the generated code is determined by its speed and size.
- A target machine with a rich instruction set may provide several ways of implementation of a given operation.

(5) Register Allocation :- Efficient utilization of register is important in generating good code.

- The use of registers is subdivided into two subprograms:
  (a) During register allocation, we select the set of variables that will reside in registers.
  (b) During register assignment, we pick the specific register that a variable will reside in.

⑥ Choice of Evaluation order:- The order in which computations are performed can affect the efficiency of the target code. Some computation order require fewer registers to hold intermediate results than others.

The Target Language:-

The output of code generation is an object code or machine code.

It can be in following forms:-

(a) Absolute Code
(b) Relocatable Machine Code
(c) Assembler Code

(a) Absolute Code:- Absolute code is machine code that contains reference to actual addresses within program's address space. The generated code can be placed directly in the memory and execution starts immediately. Generally small programs can be compiled & executed quickly because of absolute code generation.

(b) Relocatable Code:- Producing a relocatable m/c code as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together & loaded for execution using "linking loader".

(c) Assembler Code:- Producing an assembly code as output makes the process of code generation easier. We can generate symbolic instructions and use the macro facilities of the assembler to help in code generation. But in this, code generation process is slow (because assembling, linking & loading is required)

Unit-5 Compiler Notes

Code optimization:- refers to the techniques used by compiler to improve the execution effeciency of the generated object code. It is an optional phase.

Goals for optimization

(i) optimizing for time effeciency.
(ii) optimizing for memory conservation.

Principal sources of optimization

(i) **Effecient Utilization of Registers:-** The richest source of optimization is in the effecient utilization of registers and the instruction set of a machine.

(ii) **Inner Loops:-** (Loop optimization) After register allocation, the handling of inner loop is the next imp. class of optimization. In a program, most of the execution time is spend in just a small part of program. This is called "90-10" rule. stating that 90% of the time is spent in 10% of the code.
The typical optimization performed in inner loops are
(a) Removal of loop invariant Computation (Code motion)
(b) Elimination of inductional variable
(c) Reduction in strength. (d) Loop unrolling (e) Loop Jamming

(iii) **Common Subexpression elimination:-** The Common subexpressions in a program are also a major source of optimization. The optimization can be performed by eliminating the Common Subexpressions

(iv) Constant Folding:- Constant folding refers to the replacement of runtime computation by compile time computation ie. the substitution of values for names whose values are constant.

(v) Dead code elimination:- A piece of code is dead of data computed is never used elsewhere.

or

Basic Blocks:- A basic block is a sequence of consecative statements in which the flow of control enters at the begining and leaves at the end without halt or possibility of branching.

Algorithm for partitioning Three address code into Basic Blocks

(1) Determine the 'leaders' using
   (a) The first statement of a program is a leader statement.
   (b) for any conditional and unconditional 'goto' the target statement is a 'leader'.
   (c) Any statement that immediately follow a conditional 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.

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 follow B_1.

Loops in flow Graph:- A loop is defined as the collection of nodes, such that

(i) All such nodes are strongly connected. i.e. all nodes in a loop can be reached from the others.
(ii) has a unique entry.

Example for the following program fragment

(i) write the 3-address code
(ii) Find the basic blocks.
(iii) Draw the flow graph
(iv) identify the loops in the flow graph, if any.

```
Begin
    Prod = 0;
    i = 1;
    do begin
        Prod = Prod + A[i] * B[i];
        i = i + 1;
    end
    while i <= 20
End
```

"each array element is of 4 bytes"

(i) 3-address Code

(1) Prod = 0
(2) i = 1
(3) $t_1 = i \times 4$
(4) $t_2 = addr(A) - 4$
(5) $t_3 = t_2[t_1]$
(6) $t_4 = addr(B) - 4$
(7) $t_5 = t_4[t_1]$
(8) $t_6 = t_3 * t_5$
(9) $t_7 = Prod + t_6$
(10) $Prod = t_7$
(11) $t_8 = i + 1$
(12) $i = t_8$
(13) if $i <= 20$ goto (3)

(ii) & (iii) (iv)

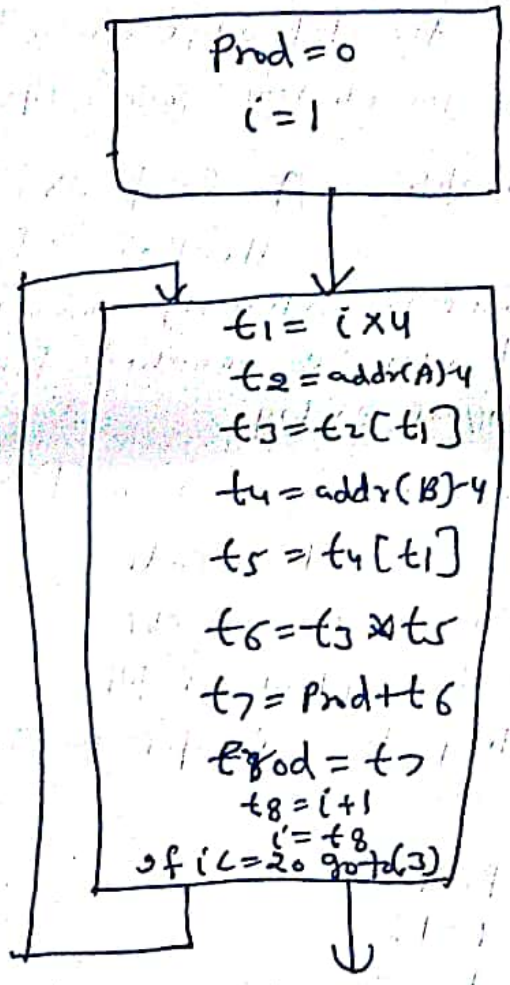


fig. (1) Basic block & flow graph Showing Loops.

Loop optimization:- Loop optimization is a technique in which code optimization is performed on inner loops.

Loop optimization techniques (methods):-

(i) Code Motion (Loop invariant Computation)
(ii) Induction variable Elimination
(iii) Reduction in Strength
(iv) Loop Unrolling
(v) Loop Jamming

(i) Code Motion (Loop invariant Computation) Code motion is a technique which moves the code outside the loop. If there is an expression in the loop whose result remains unchanged even after executing the loop for for several times, then such expression should be placed before the loop.

ex:-1

Before code Motion

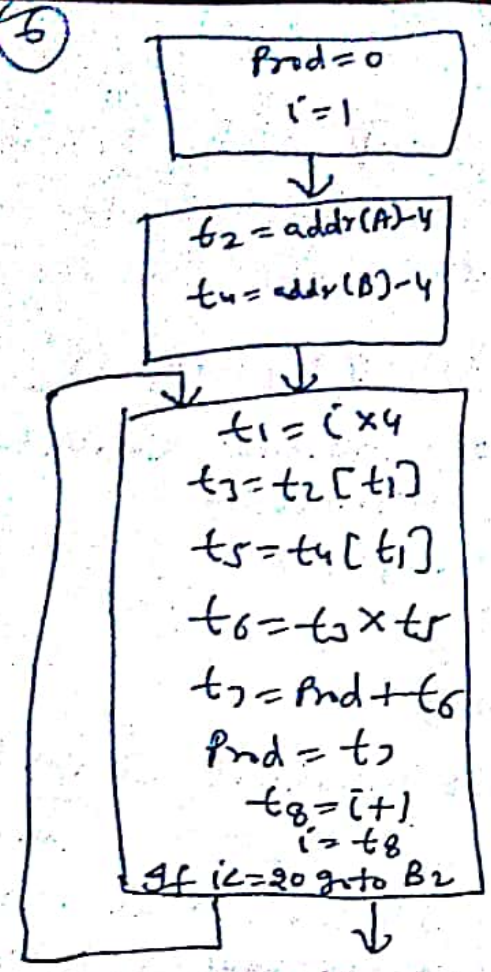
```
while (i<=max-1)
{ sum= sum+a[i];
}
```

After Code Motion

```
n= max-1
while (i<=n)
{ sum= sum+a[i];
}
```

ex:-2 ~~(i) loop invariant elimination~~ from fig. (1)

$t_2 = addr(A) - 4$
and
$t_4 = addr(B) - 4$
are loop invariant expressions
So putting them outside:-



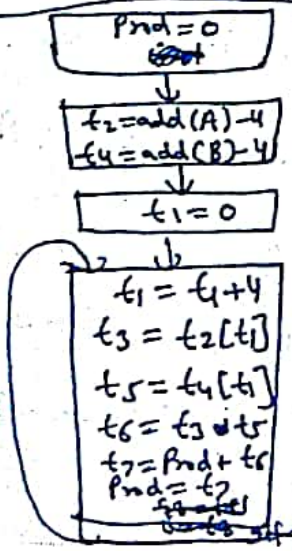
fig(2) Flow Graph after the code Motion

(ii) ~~Loop~~ Induction Variable Elimination:-

A variable is called induction variable, if the value of variable gets changed every time, it is either decremented or incremented by some constant when there are two or more induction variables in a loop, it may be possible to get rid of all but one.

Ex:- In fig(2), when i = 1, 2, 3, 4 ---- 20
then t_1 = 4, 8, 12, ---- 80,

So [i & t_1 are induction variables], so eliminate i,



fig(3) flow graph after induction code elimination.

(iii) Reduction in strength:-

Reduction in strength is another major source of optimization. The reduction in strength replaces a more expensive operation with a less expensive one.

ex:- Before strength reduction

```
for(i=1; i<=50; i++)
{ count = i*7;
}
```

After strength reduction

```
temp = 7
for(i=1; i<=50; i++)
{ count = temp;
  temp = temp+7;
}
```

It replaces a complex multiplication operation by a simple addition operation, which result is speeding up the object code as addition operation takes less time than multiplication.

(iv) Loop Unrolling:- In this method, the no. of jumps and tests can be reduced by writing the code multiple times. It is more efficient in time. But results increased memory load.

ex:

Before Loop Unrolling

```
int i=1;
while(i<=100)
{ a[i]= b[i];
  i++;
}
```

After Loop Unrolling

```
int i=1;
while(i<=100)
{ a[i]= b[i];
  i++;
  a[i]=b[i];
  i++;
}
```

(Loop Fusion)

(v) Loop Jamming:- In loop Jamming method, several loops are merged to one loop. when two adjacent loops would iterate the same no. of times, then their bodies can be merged.

Dead Code Elimination:- A variable is said to be dead at a point in a program if the value contained into it is never been used. The code containing such a variable is said to be dead code. The optimization can be performed by eliminating such a dead code.

example

```
i=0;
if(i=1)
{a=x+5;
}
```

In this, if statement is a dead code, as this condition will never get satisfied, so such statement can be eliminated for optimization.

After elimination

```
{a=x+5;
}
```

Common Sub expression elimination:-

The common sub expression is an expression appearing repeatedly in the program which is computed previously.

my

DAG Representation of Basic blocks

The DAG (Directed Acyclic Graph) is a useful data structure for automatically analyzing the basic blocks.

A DAG is a directed graph with no cycles which gives a picture of how the values computed by each statement in a basic block is used in the subsequent statement in the block. In a DAG

(i) The leaf nodes are labeled by unique identifier (either variable names / constant)

(ii) The interior nodes are labeled by operator

DAG Construction Algorithm:-

(10) (10)

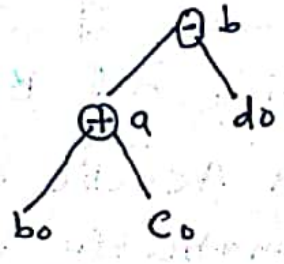
Numerical Based on DAG

(1) Construct the DAG for the following

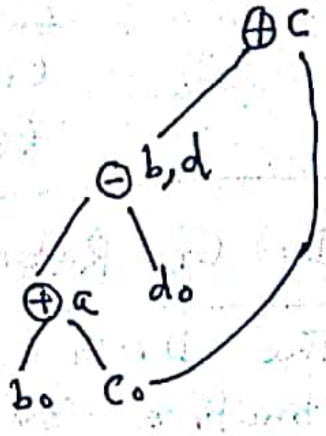
$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

Ans

(a)



(b)

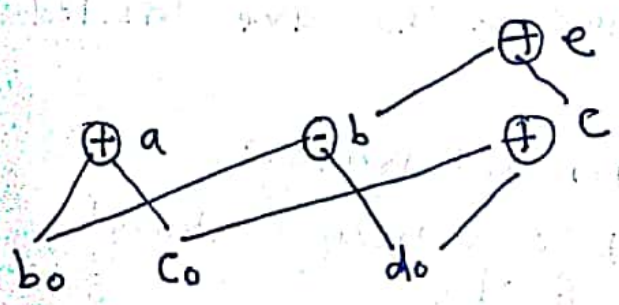


(c)

(2) Construct the DAG for the following

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

Ans



Advantages & Application of DAG

(1) Determine the common sub expression.

(2) Determine which name are used inside block and computed outside block.

(3) Simplify the list of quadruples by the elimination of common subexpression

$x + 0 = 0 + x = x$

$x - 0 = x$

$x * 1 = 1 * x = x$

$x / 1 = x$

(ii) Reduction in strength

$x ** 2 = x * x$

$2 * x = x + x$

$x / 2 = x * 0.5$

(iii) Constant folding

In this, we evaluate constant expression at Compile time and replace the constant expression by their values.

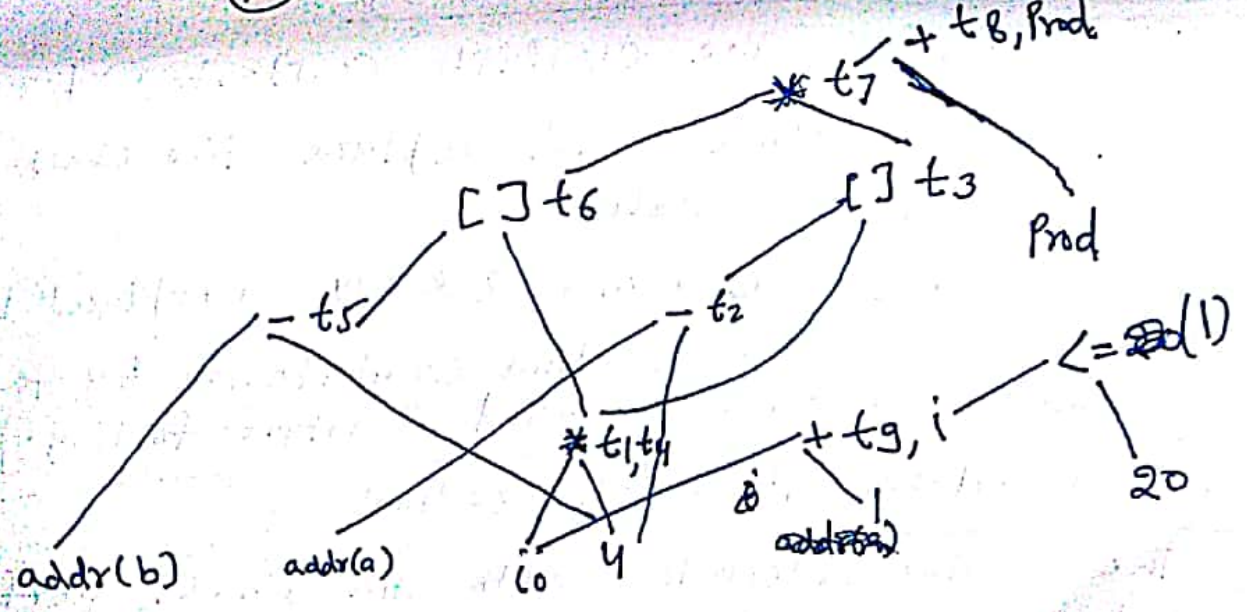
So the expression $2 * 3.14$ is replaced by 6.28.

[Replacement of run-time computation by the compile time computation is called Constant folding.]

Construct the DAG for the following code:

1. $t_1 = i \times 4$
2. $t_2 = addr(a) - 4$
3. $t_3 = t_2[t_1]$
4. $t_4 = i \times 4$
5. $t_5 = addr(b) - 4$
6. $t_6 = t_5[t_4]$
7. $t_7 = t_3 \times t_6$
8. $t_8 = Prod + t_7$
9. $Prod = t_8$
10. $t_9 = i + 1$
11. $i = t_9$
12. If $i <= 20$ goto (1)

Ans



DAG for the following code

ex:- Construct the DAG for the following:-

$a = b * c$

$d = b$

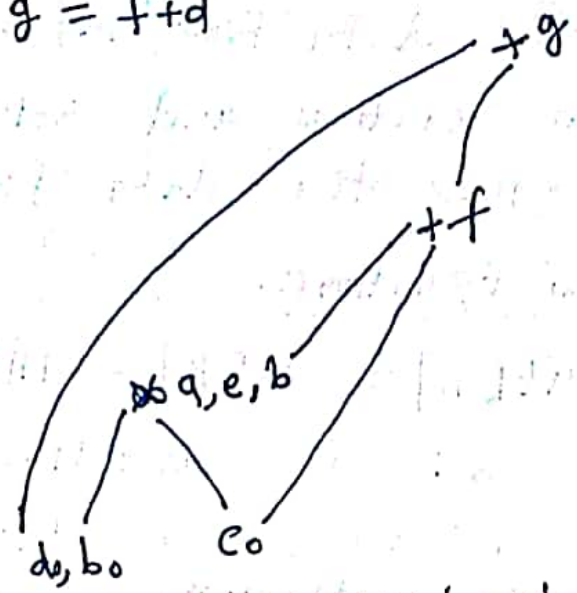
$e = d * c$

$b = e$

$f = b + c$

$g = f + d$

Sol:-



DAG (Directed Acyclic Graph)

ex:- Construct the DAG for the following code

X:=X * 1

or X:=X + 0

are often produced by straight forward intermediate code-generation algorithms, and they can be eliminated.

(v) Multiplication by two could be replaced by an addition or by a shift if the other operand is fixed point.

10.8 GLOBAL DATA FLOW ANALYSIS

In order to optimize the code efficiently, the compiler collects the information about the program as whole and distribute. This information to each block of the flow graph. This process is known as data-flow analysis. In order to perform the global data flow analysis the program is to be represented in the form of a flow graph as discussed in section 10.5.2.

The global data flow analysis helps in obtaining the various pieces of information by the examination of the entire program. This information is useful for achieving a number of optimizations. The particular problem that we discuss here is called **use-definition (ud-) chaining.** The ud-chaining provides the answer to the problem :

"Given that the identifiers A is used at a point p, at what point could the value of A used at p have been defined ?"

This definition requires the understanding of following definitions :

1. Program point. A point in a program is defined as the position before or after any intermediate-langauge statement. The control reach the point just before a statement when that statement is about to be executed. The control reeaches the point after a statement when that statement has just been executed.

2. Use of an identifier. By a use of an identifiers *A*, we mean any occurence of *A* as an operand. For example, in the statement

$$B := A * C$$

as *A* appears on the right hand side of the statement. Therefore, it represents the use of A as an operand.

3. Definition of an identifier. By a definition of an identifier *A*, we mean an assignment to *A*. For example, in the statement

$$B := A * C$$

as an identifier B is assigned the value therefore, in this assignment statement B is defined.

Next we present the process of ud-chaining in detail.

10.8.1 Reaching Definitions

The reaching definitions implies the determination of definitions that apply at a point *p* in a flow graph. This determination follows the steps as given below:

3. For each basic block B, compute the following two sets :
   (a) **GEN[B] :** The set of all the definitions within block B that reach the end of the block B, *i.e.*, the set GEN[B] consists of all the definitions generated in block B.
   (b) **KILL[B] :** The set of all the definitions outside block B that define the same variables having definitions in block B also.
4. Use a bit-vector representation for the sets GEN[B] and KILL[B].
5. For all the basic blocks B, comput the following sets :
   (a) **IN[B] :** The set of all the definitions reaching the point just before the first statement of block B.
   (b) **OUT[B] :** The set of all the definitions reaching the point just after the last statement of basic block B.
6. After computing the sets IN[B] and OUT[B], we can determine the definitons which reach any use of the identifier. A within block B by applying the following rules. Let u be the statement in question using A.
   (a) If there are definitions of A with the block B prior to the statement u, then the last such definition is the only definition of A reaching u *i.e.*, the *ud*-chain for A consist of that definition.
   (b) If there are no definitions of A within block B prior to u, then the definitions of A reaching u are these definitions of A that are in IN[B] *i.e.*, the ud-chain for A consists of a the definition of A in IN[B].

10.8.2 Data-Flow Equations

There are two sets of equations, called data-flow equations, that relate IN's and OUT's. For all blocks B, these equations are defined as given below :

1. OUT[B] = IN[B] – KILL[B] ∪ GEN[B]

This data flow equation implies that a definition a reaches at the end of block B if and only if either.

(a) d is in IN[B], and is not in KILL[B], or

(b) d is generated within B *i.e.*, is in GEN[B].

In this equation we assume that the operations – and ∪ are equal in precedence and left associative. Therefore, the equation can be represented as

OUT[B] = (IN[B] – KILL[B]) ∪ GEN[B]

In this equation, IN[B] – KILL[B] can be implemented by the following steps :

- Take the templement of KILL[B].
- Apply logical **AND** to compute IN[B] ^ ~ KILL[B].

The union operation in OUT[B] can be implemented by logical **OR** operation.

In general, the OUT[B] can be computed mathematical as

OUT[B] = (IN[B] ^ ~ KILL[B]) V GEN[B]

2. $\mathbf{IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]}$

This data equation implies that a definition reaches the beginning of a block B if and only if it reaches the end of one of its predecessors. If a block B has no predecessor, then IN[B] is the empty set.

Algorithm for computing IN and OUT

Input : A flow graph for with KILL[*B*] and GEN[*B*] have been computed for each block *B*.

Output : IN[*B*] and OUT[*B*] for each block *B*.

Method : The method consists of following procedure :

```
For each block B do
   begin
      IN[B]:= Ø
      OUT[B]:=GEN[B]
   end;
   CHANGE:=true :
   while CHANGE DO
      begin
         CHANGE:=false;
         for each block B do
            begin
               NEWIN:=  ⋃  OUT[P];
                     P a processor
                        of B
               If NEWIN ≠ IN[B] then CHANGE;=true;
               IN[B]:=NEWIN;
               OUT[B]:=IN[B]–KILL[B] U GEN[B]
            end
      end
```

Example 10.7. Consider the flow graph of Fig. 10.19

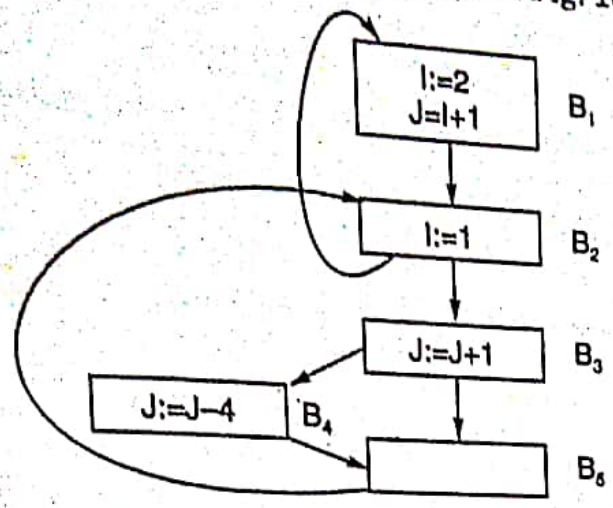


Fig. 10.19. A flow graph

(*a*) Compute GEN and KILL for each block.

(*b*) Compute IN and OUT for reaching definitions.

(*c*) Compute ud-chains.

Solution. First of all we name the definitions appearing in blocks as shown in Fig. 10.20.

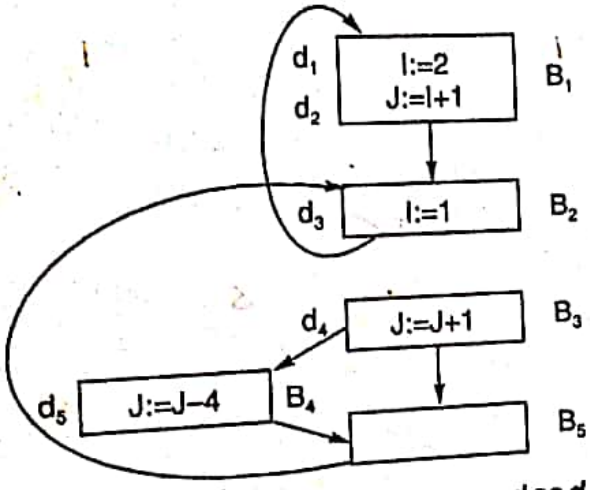


Fig. 10.20. A flow graph in which definitions are named as d_1, d_2, d_3, d_4 and d_5 respectively.

(*a*) GEN and KILL sets (in bit-vector form) for each blocks are computed in Fig. 10.21.

Block B	GEN[B]	bit vector $d_1\ d_2\ d_3\ d_4\ d_5$	KILL[B]	bit vector $d_1\ d_2\ d_3\ d_4\ d_5$
B_1	$\{d_1, d_2\}$	11000	$\{d_3, d_4, d_5\}$	00111
B_2	$\{d_3\}$	00100	$\{d_1\}$	10000
B_3	$\{d_4\}$	00010	$\{d_2, d_5\}$	01001
B_4	$\{d_5\}$	00001	$\{d_2, d_4\}$	01010
B_5	Ø	00000	Ø	00000

Fig. 10.21. Computation of GEN and KILL

(b) We initialize

$$IN[B] = \emptyset$$
$$OUT[B] = GEN[B]$$

Therefore, initially the IN[B] and OUT[B] are shown in Fig. 10.22.

Block B	*IN[B]*	*OUT[B]*
B_1	00000	11000
B_2	00000	00100
B_3	00000	00010
B_4	00000	00001
B_5	00000	00000

Fig. 10.22. The initial values of IN[B] and OUT[B].

Pass 1 :

$$IN[B_1] = OUT[B_2]$$

as the only predecessor of B_1 is block B_2.

$\therefore$ $IN[B_1] = 00100$

$$OUT[B_1] = IN[B_1] - KILL[B_1] \cup GEN[B_1]$$
$$= (00100 \wedge \sim 00111) \vee 11000$$
$$= (00100 \wedge 11000) \vee 11000$$
$$= 00000 \wedge 11000$$
$$= 11000$$

$$IN[B_2] = OUT[B_1] \vee OUT[B_3] = 11000 \vee 00000 = 11000$$

$$OUT[B_2] = IN[B_2] - KILL[B_2] \cup GEN[B_2]$$
$$= 11000 - 10000 \cup 00100$$
$$= 01100$$

$$IN[B_3] = OUT[B_2] = 01100$$

$$OUT[B_3] = IN[B_3] - KILL[B_3] \cup GEN[B_3]$$
$$= 01100 - 01001 \cup 00010$$
$$= 00110$$

$$IN[B_4] = OUT[B_3] = 00110$$

$$OUT[B_4] = IN[B_4] - KILL[B_4] \cup GEN[B_4]$$
$$= 00110 - 01010 \cup 00001$$
$$= 00101$$

$$IN[B_5] = OUT[B_3] \vee OUT[B_4]$$
$$= 00110 \vee 00101 = 00111$$

$$OUT[B_5] = IN[B_5] - KILL[B_5] \cup GEN[B_5]$$
$$= 00111 - 00000 \cup 00000$$
$$= 00111.$$

The results of first pass are shown in Fig. 10.23.

Block B	*IN[B]*	*OUT[B]*
B_1	00100	11000
B_2	11000	01100
B_3	01100	00110
B_4	00110	00101
B_5	00111	00111

Fig. 10.23. The first pass of IN and OUT computation.

Next we compute subsequent passes until the results of Pass k = Pass k – 1. When we continue we find that the pass 4 yields the same results as pass 3. Therefore, we terminate the process at pass 3. The complete computation of In and OUT is given in Fig. 10.24.

Block	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B_1	00000	11000	00100	11000	01100	11000	01111	11000
B_2	00000	00100	11000	01100	11111	01111	11111	01111
B_3	00000	00010	01100	00110	01111	00110	01111	00110
B_4	00000	00001	00110	00101	00110	00101	00110	00101
B_5	00000	00000	00111	00111	00111	00111	00111	00111

Fig. 10.24. Computation of IN[B] and OUT[B]

(c) Computing ud-Chains

The different uses of variable names are

(*i*) Use of I in d_2

(*ii*) Use of J in d_4

(*iii*) Use of J in d_5

(*i*) The use of I at d_2 in block B_1 is preceded by a definition of I in B_1, *i.e.*, d_1. Thus, the ud-chain for I in d_2 consists of d_1.

(*ii*) The use of J at d_4 in B_3 is not preceded by a definition of J in B_3. Therefore, we conside IN[B_3] where [From Fig. 10.24]

$$IN[B_3] = 0111$$
$$= [d_2, d_3, d_4, d_5].$$

Out of these d_2, d_4 and d_5 are the definitions of].

Therefore, the ud-chain for] in d_4 consists of $\{d_2, d_4, d_5\}$.

(*iii*) The use of J at d_5 in block B_4 is not preceded by a definition of J. Therefore, we consider IN[B_4] where [From Fig. 10.24]

$$IN[B_4] = 00110$$
$$= [d_3, d_4]$$

Out of these d_4 is the only definition of J. Therefore, the ud-chain for J in d_5 consists of a only of d_4.