UNIT – VI

Code generation

Objectives

To understand various target code forms and to generate target code from three address code and DAG.

Syllabus

Code Generation- Issues in code generation, generic code generation, code generation from DAG. Machine dependent code optimization: Peephole optimization, register allocation, instruction scheduling, inter procedural optimization.

Learning Outcomes

Students will be able to

- Understand various target code forms such as absolute machine language, relocatable machine language or assembly language.
- Generate target code from three address code.
- Generate target code from DAG.
- Understand machine dependent optimizations-peephole optimization, register allocation, instruction scheduling.
- Understand inter procedural optimization.

Learning Material

6.1 Code generation

- The final phase in compiler model is the code generator.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.
- The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.
- A code generator has three primary tasks: instruction selection, register allocation and assignment and instruction ordering.

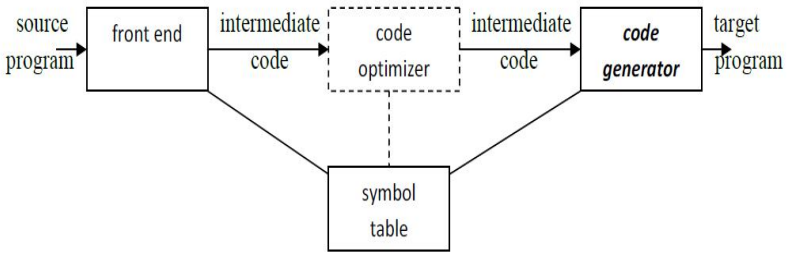


Figure 6.1 Position of code generator

6.1.1 Issues in code generation:

1. **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 with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation.
    - Intermediate representation include three-address representations such as Quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's.
    - We assume that the front end has scanned, parsed, and translated the source program into a relatively low-level

intermediate representation, so that the values of the names appearing in the intermediate representation can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers.

- The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

2. Target program

- The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.
- The output of code generator is the target program. Like the intermediate code, the ouput may take variety of forms: absolute machine languae, relocatable machine language or assembly language.
- 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.
- Producing a relocatable machine-language program (often called an object

  module) as output 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 somewhat easier.

3. Register Allocation

- Instructions involving register 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 two subproblems:
  1. **Register allocation**, during which we select the set of variables that will reside in registers at each point in the program.
  2. **Register assignment**, during which we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete.

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d
```

(a) (b)

Two three-address code sequences

```
L   R1,a           L   R0, a
A   R1,b           A   R0, b
M   R0,c           A   R0, c
D   R0,d           SRDA R0, 32
ST  R1,t           D   R0, d
                   ST  R1, t
```

(a) (b)

Optimal machine-code sequences

4. 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 machines, for example, floating-point operations are done using separate registers.

- Instruction speeds and machine idioms are other important factors.
- For example, the sequence of three-address statements

  a=b+c

  d=a+e

  would 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, and so is the third if a is not subsequently used.
- 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.

5. Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order in the general case is a difficult NP-complete problem.

6.1.2 Approaches to Code Generation:

1. Straight forward code generation algorithm (Generic code generation algorithm)
2. Tree directed code selection technique (Code Generation from DAG)

- The target machine and its instruction set is a pre requisite for designing a good code generator.
- Our target computer is a byte addressable machine with four bytes to a word and n general purpose registers are R_0, R_1,.....R_{n-1}. It has two address instructions of the form op source, destination
- op is an op-code and source and destination are data fields.

  MOV (move source to destination)

  ADD (add source to destination)

  SUB (subtract source from destination)
- The addressing mode with their assembly language forms and associated costs are as follows:

Table: 6.1 addressing mode with their assembly language forms and associated costs

MODE	FORM	ADDRESS	EXAMPLE	ADDED-COST
Absolute	M	M	ADD R_0, R_1	1
Register	R	R	ADD temp, R_1	0
Index	c (R)	c + contents (R)	ADD 100(R_2), R_1	1
Indirect register	*R	contents (R)	ADD * R_2, R_1	0
Indirect Index	*c (R)	contents (c + contents (R)	ADD * 100(R_2), R_1	1
Literal	# c	constant c	ADD # 3, R_1	1

- A memory location M or a register R represents itself when used as a source or destination.

  For example, the instruction

  MOV R0, M stores the contents of register R0 into memory location M.

- An address offset c from the value in register R is written as c(R).

  MOV 4(R0), M stores the value contents(4+contents(R0)) into memory location M.

- Indirect versions of the last two modes are indirected by prefix *.

  MOV *4(R0), M stores the value contents(contents(4+contents(R0))) into memory location M.

- A final address mode allows the source to be a constant:

MODE	FORM	CONSTANT	ADDED COST
literal	#c	c	1

The instruction **MOV #1, R0** loads the constant 1 into register R0.

6.1.3 Instruction Costs

- The cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

i) MOV b, R0
ADD c, R0 cost = 6
MOV R0, a

ii) MOV b, a
ADD c, a cost = 6

iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :

MOV *R1, *R0
ADD *R2, *R0 cost = 2

- In order to generate good code for target machine, we must utilize its addressing
  capabilities efficiently

6.1.4 A SIMPLE CODE GENERATOR (Generic Code Generation)

- A code generator generates target code for a sequence of three-address statements and effectively uses registers to store operands of the statements.
- For example: consider the three-address statement a := b+c
  It can have the following sequence of codes:

```
ADD Rj, Ri   Cost = 1      // if Ri contains b and Rj contains c
             (or)
ADD c, Ri    Cost = 2      // if c is in a memory location
             (or)
MOV c, Rj    Cost = 3      // move c from memory to Rj and add
ADD Rj, Ri
```

Register and Address Descriptors:

- A **register descriptor** is used to keep track of what is currently in each registers. The
  register descriptors show that initially all the registers are empty.
- An **address descriptor** stores the location where the current value of the name can be
  found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three address code sequence:

  t : = a – b

  u : = a – c

  v : = t + u

  d : = v + u

  with d live at the end.

Code sequence for the example is:

Table 6.2 Code sequence

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

6.2 GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

t1 : = a + b

t2 : = c + d

t3 : = e – t2

t4 : = t1 – t3

Generated code sequence for basic block:

MOV a , R0

ADD b , R0

```
MOV c , R1
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

Rearranged basic block:

Now t1 occurs immediately before t4.

```
t2 : = c + d
t3 : = e – t2
t1 : = a + b
t4 : = t1 – t3
```

Revised code sequence:

```
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
```

In this order, two instructions MOV R0 , t1 and MOV t1 , R1 have been saved.

6.2.1 A Heuristic ordering for Dags

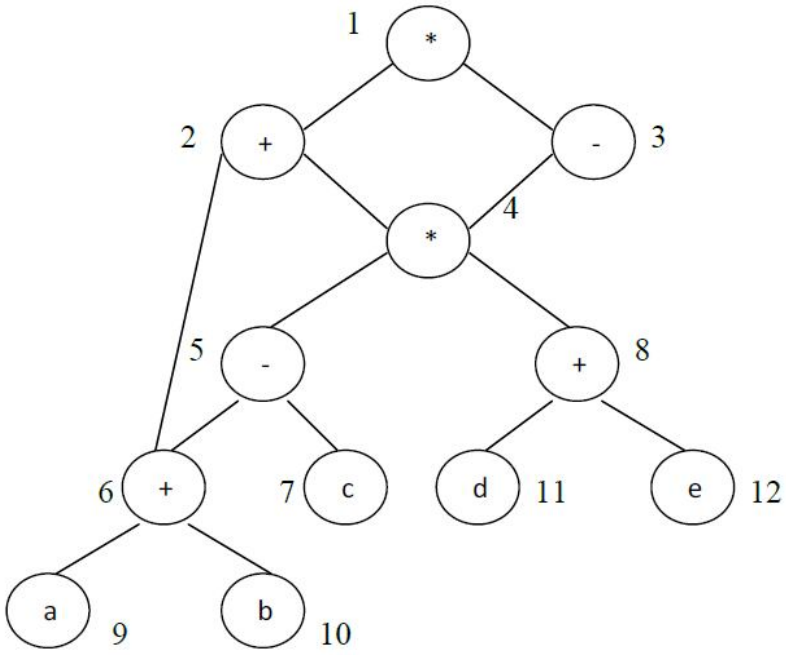
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

6.2.2 Listing Algorithm:

1) **while** unlisted interior nodes remain **do begin**
2) select an unlisted node n, all of whose parents have been listed;
3) list n;
4) **while** the leftmost child m of n has no unlisted parents and is not a leaf **do**
begin
5) list m;
6) n : = m
end
end

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we

select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t8 : = d + e

t6 : = a + b

t5 : = t6 – c

t4 : = t5 * t8

t3 : = t4 – e

t2 : = t6 + t4

t1 : = t2 * t3

This will yield an optimal code for the DAG on machine whatever be the number of registers.

6.3 Optimal ordering for Trees

The algorithm has two parts:

- The first part labels each node of the tree, bottom-up, with an integer that denotes the fewest number of registers required to evaluate the tree with no stores of intermediate results.
- The second of the algorithm is a tree traversal whose order is governed by the computed node labels. The output code is generated during the tree traversal.

6.3.1 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

***(1)if* n is a leaf then**

(2) if n is the* leftmost child of its parent *then

(3) label(n) := 1

(4) else label(n) :=0

else begin /* n is an interior node */

(5) let n1, n2, ...nk be the children of n ordered by label,

so label(n1)>=label(n2)>=……. >=label(nk)

(6) label(n) := max(label(ni)+i-1)

1<i<k

end

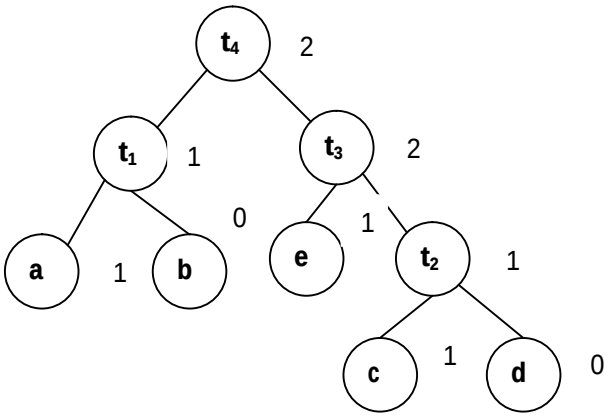


Figure 6.2 Labelled Tree

6.4 Register Allocation and Assignment:

- Instructions involving only register operands are faster than those involving memory operands. Therefore, efficient utilization of registers is vitally important in generating good code. Various strategies for deciding at each point in a program what values should reside in registers (register allocation) and in which register each value should reside (register assignment).
- One approach to register allocation and assignment is to assign specific values in the target program to certain registers. For example, we could decide to assign base addresses to one group of registers,

arithmetic computations to another, the top of the stack to a fixed register, and so on.

- This approach has the advantage that it simplifies the design of a code generator. Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated into the other registers.

6.4.1 Global Register Allocation

- One strategy for global register allocation is to assign ***some fixed number of registers to hold the most active values*** in each inner loop.
- The selected values may be different in different loops. Registers not already allocated may be used to hold values local to one block.
- This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation.
- With C compilers, a programmer could do some register allocation explicitly by using register declarations to keep certain values in registers for the duration of a procedure.

Usage Counts

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$

where use(x, B) is the number of times x is used in B prior to any definition of x; live(x, B) is 1 if x is live on exit from B and is assigned a value in B, and live(x, B) is 0 otherwise.

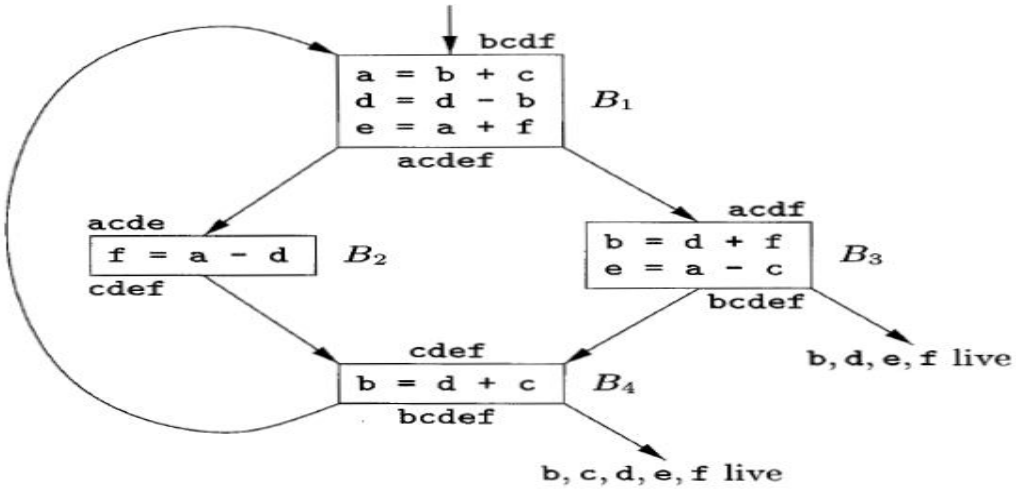


Fig 1: Flow graph of an inner loop

- For x = a, we observe that a is live on exit from B1 and is assigned a value there, but is not live on exit from B2, B3, or B4.
- Thus, $\sum_{B \text{ in } L}$ use (a, B) = 2. Hence the value for x = a is 4. That is, four units of cost can be saved by selecting a for one of the global registers.
- The values of (Fig 1) for b, c, d, e, and f are 5, **3,** 6, 4, and 4, respectively. Thus, we may select a, b, and d for registers RO, R1, and R2, respectively. Using R0 for e or f instead of a would be another choice with the same apparent benefit.

6.4.2 Register Allocation by Graph Coloring

- When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored *(spilled)* into a memory location in order to free up a register. Graph coloring is a simple, systematic technique for allocating registers and managing register spills.
- In the method, two passes are used. In the first, target-machine instructions are selected as though there are an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and the three-address instructions become machine-language instructions.

- Once the instructions have been selected, a second pass assigns physical registers to symbolic ones. The goal is to find an assignment that minimizes the cost of spills.
- In the second pass, for each procedure a register-interference graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.
- An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers. A graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have the same color. A color represents a register, and the color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register. Although the problem of determining whether a graph is k-colorable is NPcomplete

6.4.3 Interprocedural Optimization

- An interprocedural optimiaztion operates across an entire program, flowing information from the caller to its callees and vice versa.
- One relatively simple but useful technique is to *inline* procedures, that is, to replace a procedure invocation by the body of the procedure itself with suitable modifications to account for parameter passing and the return value.
- This method is applicable only if we know the target of the procedure call.

6.5 Call Graphs

A call graph for a program is a set of nodes and edges such that

1. There is one node for each procedure in the program.
2. There is one node for each call site, that is, a place in the program where a procedure is invoked.
3. If call site c may call procedure p, then there is an edge from the node for c to the node for p.

```
          int (*pf)(int);

          int fun1(int x) {
              if (x < 10)
c1:               return (*pf)(x+1);
              else
                  return x;
          }

          int fun2(int y) {
              pf = &fun1;
c2:           return (*pf)(y);
          }

          void main() {
              pf = &fun2;
c3:           (*pf)(5);
          }
```

A program with a function pointer

- A C program that declares pf to be a global pointer to a function whose type is "integer to integer." There are two functions of this type, f unl and f un2, and a main function that is not of the type that pf points to. The figure shows three call sites, denoted cl, c2, and c3; the labels are not part of the program.

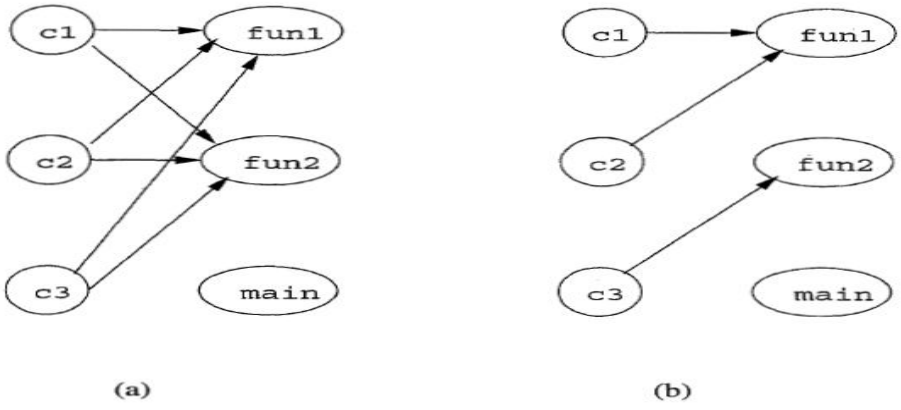


Figure 6.3 Call graphs for the above program

Context Sensitivity

- One simplistic but extremely inaccurate approach to interprocedural analysis, known as context-insensitive analysis, is to treat each call and return statement as "goto" operations.
- We create a *super* control-flow graph where, besides the normal intraprocedural control flow edges, additional edges are created connecting
    1. Each call site to the beginning of the procedure it calls, and
    2. The return statements back to the call sites.

6.6 Peephole Optimization:

- A statement-by-statement code generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for locally improving the target code is ***peephole optimization***, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.
- The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristic of peephole optimizations:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

- If we see the instruction sequence

  LD a, **R0**

  ST R0, a

  in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register **R0.**
- Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction.

Eliminating Unreachable Code

- Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed.
- This operation can be repeated to eliminate a sequence of instructions.

```
# define debug 0
        ...
if(debug)
{
        print debugging information
}
```

- In the intermediate representation, the if-statement may be translated as:

  if debug == 1 goto **L1**

  got0 L2

  L I : print debugging information

  L2:
- One obvious peephole optimization is to eliminate jumps over jumps. The code sequence above can be replaced by

if debug != **1** goto **L2**

print debugging information

L2:

- If debug is set to 0 at the beginning of the program, constant propagation would transform this sequence into

  if *0* != **1** goto **L2**

  print debugging information

  L2:

- Now the argument of the first statement always evaluates to true, so the statement can be replaced by goto **L2.**

6.6.1 Flow-of-Control Optimizations

- Intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.
- These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

  goto **L1**

  …

  Ll: goto **L2**

  by the sequence

  goto L2

  …

  L1: goto L2

- If there are now no jumps to **L1,** then it may be possible to eliminate the statement **L1:** goto **L2** provided it is preceded by an unconditional jump.
- Similarly, the sequence

  if a<b goto L1

  …

  L1: goto L2

  can be replaced by the sequence

if a<b goto L2

...

L1: goto L2

- Finally, suppose there is only one jump to **L1** and **L1** is preceded by an unconditional goto. Then the sequence

    goto L1

    ...

    L1: if a<b goto L2

    L3:

  may be replaced by the sequence

    if a<b goto L2

    goto L3

    ...

    L3:

- While the number of instructions in the two sequences is the same, we sometimes skip the unconditional jump in the second sequence, but never in the first. Thus, the second sequence is superior to the first in execution time.

a) Algebraic Simplification

- The algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as
    - x=x+0
    - or
    - x=x*1

  in the peephole.

b) Reduction in Strength

- Reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

- For example, x^2 is invariably cheaper to implement as **x * x** than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be approximated as multiplication by a constant, which may be cheaper.

c) Use of Machine Idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example, some machines have auto-increment and auto-decrement addressing modes. These modes can be used in code for statements like x=x+l.

UNIT – VI

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. The input of code generation phase is [ ]

a. Polish notation b. Three address code

c. Abstract Syntax tree d. All of the above

2. _____ phase is responsible for generating the target code.

3. _____ code is static and is always placed in same location in memory.

4. Instructions involving only register operands are faster than those memory operands. [True | False]

5. _____ and _____ programs are needed to link the modules and load the programs into the memory for execution.

6. _____is the process of deciding which intermediate representation values to keep in registers.

7. _____is the process of deciding which register should hold a given intermediate representation value.

8. Use of _____ reduces the cost of instruction.

9. _____ maintains a pointer to the list that contains the information about what is currently available in the registers?

10. _____keeps track of locations of each variable.

SECTION-B

Descriptive Questions

1. What are the object code forms? Explain in brief about the issues in code generation?
2. What is peephole optimization? Explain its characteristics.
3. What is address descriptor and register descriptor?
4. Explain the Generic code generation algorithm.
5. Explain code generation from DAG.
6. Explain about inter procedural optimization with an example.

Multiple Choice Question

1. Which of the following machine idioms perform the task equivalent to a=a+1? [ ]

a. INC b. SFT c. Both are valid d. none

2. Register assignment variables is _____ problem [ ]

a. P-hard problem b. P-complete problem

c. NP- hard problem d. NP-complete problem

3. For integer multiplication in IBM system or 370 _____ registers are used [ ]

a. Single b. Pair c. Two pairs d. Any of the above

4. Name the descriptor required to keep track of content of registers

a. Address b. Register c. Both d. Any of the above

5. Name the descriptor used to keep track of the availability of the value for the variable [ ]

a. Address b. Register c. Both d. Any of the above

6. Which of the following intermediate code helps in generating the efficient target code [ ]

a. Postfix notation b. Three address code c. Quadruples d. DAG

7. Code generator is dependent on [ ]

a. Type of input b. Type of output c. Register allocation d. all

8. What is the cost of the instruction Mov r5, r3? [ ]

a. 1 b. 2 c. 3 d. 4

9. What is the cost of the instruction Mov a, r3? [ ]

a. 1 b. 2 c. 3 d. 4

10. What is the cost of the instruction Mov c, a(r3)? [ ]

a. 1 b. 2 c. 3 d. 4

II) Problems

1. Generate code for the following statement

x=1

2. Generate code for the following statement

a=b+1

3. Generate code for the following statement

a = b + c

d = a + e

4. Generate code for the following three address code

D = B - C

E = A + B

A = E – D

5. Draw DAG for the statement a / (b + c) – d * (e + f) and generate the target code.

6.Generate code with and without optimization for the following instructions.

t_1= a+b

t_2 = c+d

t_3 = e=t_2

t_4 = t_1=t_3

7. Determine the costs of the following instruction sequences:

LD R_0, y

LD R_1, z

ADD R_0, R_0, R_1

ST x, R_0

8. Determine the costs of the following instruction sequences:

LD R_0, i

MUL R_0, R_0, 8

LD R_1, a(R_0)

ST b, R_1