UNIT – V

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.

Learning Outcomes

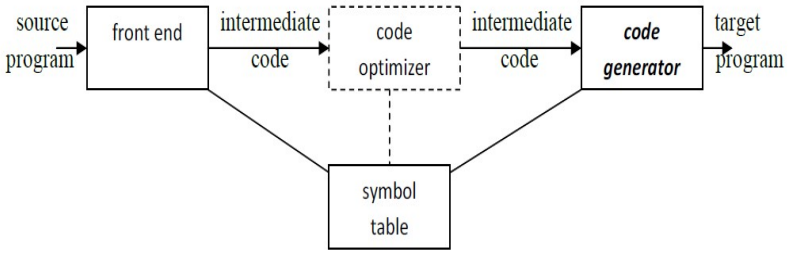
Students will be able to

- 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

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.



Position of code generator

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.
   - 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 sub problems:
     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.

<u>Approaches to Code Generation</u>:

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:

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.

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

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:

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

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.

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.

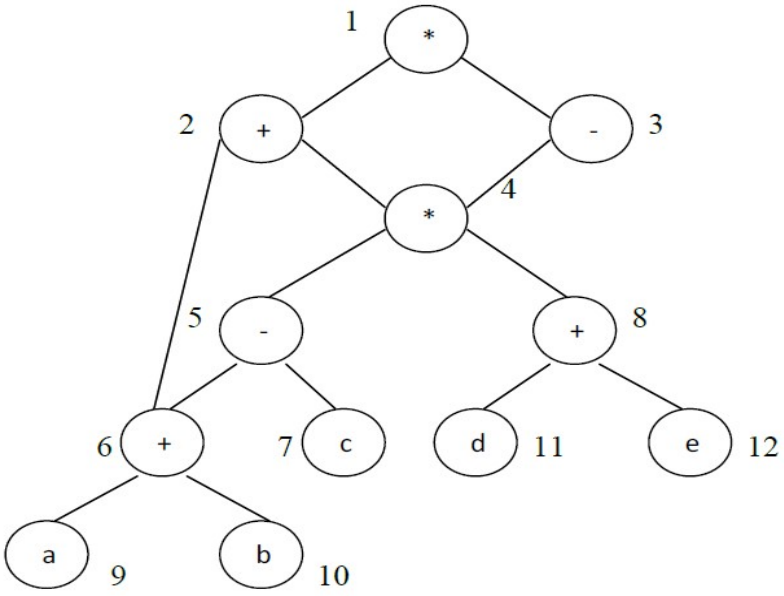
The algorithm shown below produces the ordering in reverse.

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:



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.

Optimal ordering for Trees

The algorithm has two parts:

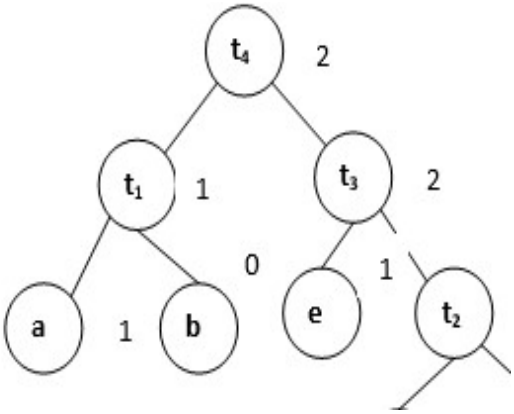
- The first part labels each node of the tree, bottom-up, with an integer that denotes the fewest number of registers required to evaluate the tree with no stores of intermediate results.
- The second of the algorithm is a tree traversal whose order is governed by the computed node labels. The output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

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



Labelled Tree

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.**

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.

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.

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, $\mathbf{x^2}$ 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.

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.