UNIT-5

Code Generation:

The final phase in our compiler model is the code generator. It takes as input form an code optimization or an intermediate representation of the source program and produces as output an equivalent target program.

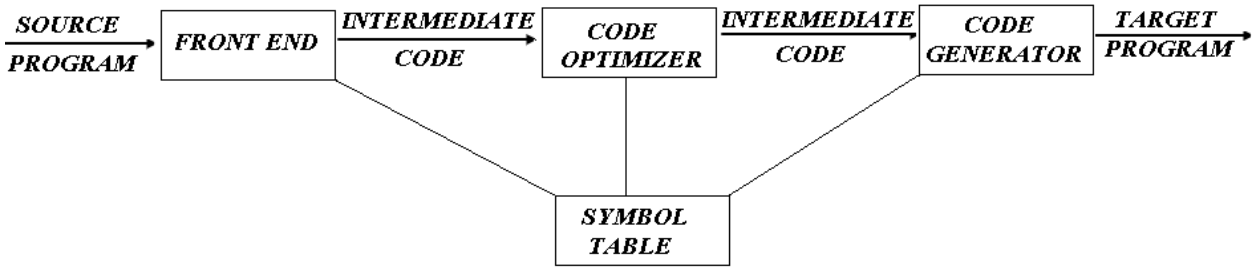


Figure: POSITION OF CODE GENERATION

Code generation process is required for following:

1. Preserve semantic meaning of source program.
2. Make effective use of available resources of target machine.
3. Code generator itself must run efficiently.

Challenges in code generation:

1. Problem of generating optimal target program is un-decidable.
2. Many sub programmable problems encountered in code generation are computationally intractable.

Main Tasks of Code Generator:

1. **Instruction selection:** Choosing appropriate target-machine instructions to implement the IR statements.
2. **Registers allocation and assignment:** Deciding what values to keep in which registers.
3. **Instruction ordering:** Deciding in what order to schedule the execution of instructions.

Design Issues of a Code Generator:

The details of the source code is dependent on the target language (code generation) and the operating system, so the design issues of the code generation process are includes,

1. Input of the code generation,
2. Target program,
3. Memory management,
4. Instruction selection,
5. Register allocation,
6. Evaluation order.

1. Input to the code generation:

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addressed of the data objects denoted by the name in the intermediate representation.
There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and DAGs.

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, real's, pointers, etc.). We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

Input:

1. Three-address presentations (quadruples, triples, etc.)
2. Virtual machine presentations (byte code, stack-machine, etc.)
3. Linear presentation (postfix, etc.)
4. Graphical presentation (syntax trees, DAGs, etc.)

2. Target Programs:

The output of the code generator is the target program. The output may take on a variety of forms; absolute machine language, re-locatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly. A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute code.

Producing a re-locatable machine language program as output allows subprograms to be compiled separately. A set of re-locatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

Producing an assembly language program as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code .The price paid is the assembly step after code generation.

Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

Target program:

1. Instruction set architecture (RISC, CISC).
2. Producing absolute machine-language program.
3. Producing re-locatable machine-language program.
4. Producing assembly language programs.

3. Memory management:

Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the "back patching". Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: *goto i* is encountered, and i is less than j, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i. If, however, the jump is forward, so i exceed j, we must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, we fill in the proper machine location for all instructions that are forward jumps to i.

4. **Instruction selection:**

The nature of the instruction set of the target machine determines the difficulty of instruction selection. 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 hand line.
Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three- address statement we can design a code skeleton that outlines 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.

```
MOV y, R0        /* load y into register R0 */
ADD z, R0        /* add z to R0 */
MOV R0, x        /* store R0 into x */
```

Unfortunately, this kind of statement – by - statement code generation often produces poor code.
For example, the sequence of statements;

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

would be translated into

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

Here the fourth statement is redundant, and so is the third if 'a' is not subsequently used.
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 implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code. 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 a register, add one to the register, and then stores the result back into a.

```
MOV   a, R0
ADD   #1, R0
MOV   R0, a
```

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

Instruction Selection:

The complexity of mapping IR program into code-sequence for target machine depends on:

1. Level of IR (high-level or low-level)
2. Nature of instruction set (data type support)
3. Desired quality of generated code (speed and size)

5. **Register allocation:**

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During **register allocation,** we select the set of variables that will reside in registers at a point in the program.

2. During a subsequent **register assignment** phase, 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 values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

```
M    x, y
```

where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form

```
D    x, y
```

where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c).

Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

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

(a) **(b)**

Fig. 2 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)**

Fig.3 Optimal machine code sequence

Register Allocation:

• Selecting the set of variables that will reside in registers at each point in the program.

Register Assignment:

• Picking 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 orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

Evaluation Order

1. Selecting the order in which computations are performed.
2. Affects the efficiency of the target code.
3. Picking a best order is NP-complete.
4. Some orders require fewer registers than others.

CODE OPTIMIZATION

Code optimization is aimed at obtaining a more efficient code.

Criteria for Code-Improving Transformations:

Simply stated, the best program transformations are those that yield the most benefit for the least effort. The transformations provided by an optimizing compiler should have several properties.

First, a transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program. The influence of this criterion pervades this chapter; at all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

Second, a transformation must, on the average, speed up programs by a measurable amount. Sometimes we are interested in reducing the space taken by the compiled code, although the size of code has less importance than it once had. Of course, not every transformation succeeds in improving every program, and occasionally an "optimization" may slow down a program slightly, as long as on the average it improves things.

Third, a transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. Certain local or "peephole" transformations of the kind are simple enough and beneficial enough to be included in any compiler.

Some transformations can only be applied after detailed, often time-consuming, analysis of the source program, so there is little point in applying them to programs that will be run only a few times. For example, a fast, non-optimizing, compiler is likely to be more helpful during debugging or for "student jobs" that will be run successfully a few times and thrown away. Only when the program in question takes up a significant fraction of the machine's cycles does improved code quality justify the time spent running an optimizing compiler on the program.

Two constraints on the technique used to perform optimizations:

1) They must ensure that the transformed program is semantically equivalent to the original program.

2) The improvement of the program efficiency must be achieved without changing the algorithms which are used in the program.

Optimization may be classified as Machine dependent and Machine independent.

1) Machine dependent optimizations exploit characteristics of the target machine.

2) Machine independent optimizations are based on mathematical properties of a sequence of source statements.

Issues of Machine-dependent optimization:

1. **Input and output formats:** The formats of the intermediate code and the target program.
2. **Memory management:**
    a) Alignment, indirect addressing, paging, segment, . . .
    b) Those you learned from your assembly language class.
3. **Instruction cost:** Special machine instructions to speed up execution.
   Example:
   Increment by 1.
   Multiplying or dividing by 2.
   Bit-wise manipulation.
   Operators applied on a continuous block of memory space.
   Pick a fastest instruction combination for a certain target machine.
4. **Register allocation:** In-between machine dependent and independent issues.
   C language allows the user to management a pool of registers.
   Some language leaves the task to compiler.

Idea: save mostly used intermediate result in a register. However, finding an optimal solution for using a limited set of registers is NP-hard.

Example:

t := a + b

load R0,a	load R0,a
load R1,b	add R0,b
add R0,R1	store R0,T
store R0,T	

Heuristic solutions: similar to the ones used for the swapping problem.

Issues of Machine-independent optimization:

1) Basic blocks and flow graphs.
2) DAG(directed acyclic graph).
3) Structure-preserving transformations.

BASIC BLOCKS AND FLOW GRAPHS:

A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

Basic Blocks:

A **basic block** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

```
t1:= a*a
t2:= a*b
t3:= 2*t2
t4:= t1+t3
t5:= b*b
t6:= t4+t5
```

A three-address statement x: = y + z is said to *define* x and to *use* y or z. A name in a basic block is said to *live* at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition into basic blocks.

Input: A sequence of three-address statements.
Output: A list of basic blocks with each three-address statement in exactly one block.
Method:

1. We first determine the set of **leaders**, the first statements of basic blocks.
   The rules we use are the following:
   a) The first statement is a leader.
   b) Any statement that is the target of a conditional or unconditional goto is a leader.
   c) Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 3: Consider the fragment of source code shown in **(figure: 7)**; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

```
Begin
        prod := 0;
        i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i+1;
    end
    while  i<= 20
end
```

fig 7: program to compute dot product

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic blocks.
Statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

1. prod := 0
2. i := 1
3. t1 := 4*i
4. t2 := a [t1]
5. t3 := 4*i
6. t4 :=b [t3]
7. t5 := t2*t4
8. t6 := prod +t5
9. prod := t6
10. t7 := i+1
11. i := t7
12. if i<=20 goto (3)

Fig 8. Three-address code computing dot product

Transformations on Basic Blocks:

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be *equivalent* if they compute the same set of expressions.
A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

Flow Graph:

A flow graph is a directed graph that is used to portray basic block and their successor relationships. The nodes of a flow graph are the basic blocks. The basic block whose leader is the first statement is known as the initial block. There is a directed edge from block B1 to B2 if B2 could immediately follow B1 during execution.
To determine whether there should be directed edge from B1 to B2, following criteria is applied:

- There is a jump from last statement of B1 to the first statement of B2, OR
- B2 immediately follows B1 in order of the program and B1 does not end in an unconditional jump.

B1 is known as the predecessor of B2 and B2 is a successor of B1.

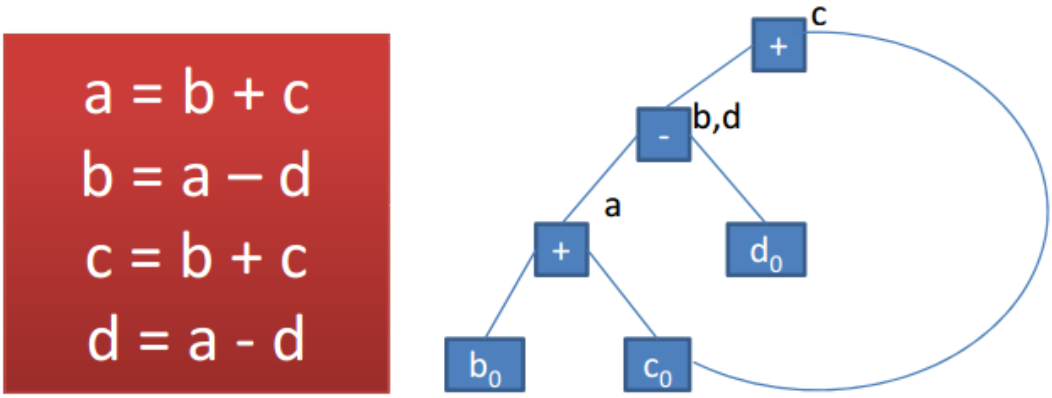
DAG Representation of Basic Blocks:

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 block as follows:

1) There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
2) There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements 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 designated *output nodes.* These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph.

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

a) We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.
b) We can eliminate *dead code,* that is, instructions that compute a value that is never used.
c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes t hereby simplify t he computation.



Applications of DAG:

- We automatically detect common sub-expressions while constructing DAG.
- It is also known as to which identifiers have there values used in side the block; they are exactly those for which a leaf is created in Step (1).
- We can also determine which statements compute values which could be used outside the block; they are exactly those statements S whose node n in step (2) still has NODE(A) = n at the end of DAG construction, where A is the identifier assigned by statement S i. e. A is still an attached identifier for n.

Structure Preserving transformations:

The primary structure-preserving transformations on basic blocks are:

1. Common sub-expression elimination
2. Dead-code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements.
5. Code Motion
6. Induction Variables and Strength Reduction
7. Variable Propagation

We assume basic blocks have no arrays, pointers, or procedure calls.

1) **Common sub-expression elimination:**

Consider the basic block

a:= b+c

b:= a-d

c:= b+c

d:= a-d

The second and fourth statements compute the same expression, namely b+c-d, and hence this basic block may be transformed into the equivalent block

a:= b+c

b:= a-d

c:= b+c

d:= b

Although the 1st and 3rd statements in both cases appear to have the same expression on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

2) **Dead-code elimination:**

Suppose x is dead, that is, never subsequently used, at the point where the statement x: = y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

3) **Renaming temporary variables:**

Suppose we have a statement t: = b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed. In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a *normal-form* block.

4) **Interchange of statements:**

Suppose we have a block with the two adjacent statements

t1:= b+c

t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that are possible.

5) **Code Motion:**

The main aim of optimization is to improve the execution time of the program by reducing the evaluation frequency of expressions. Evaluation of expressions is moved from one part of the program to another in such a way that it is evaluated lesser frequently. Loops are usually executed several times. We can bring the loop-invariant statements out of the loop.

Example:

```
a = 200;
while (a > 0)
{
```

```
        b = x + y;
        if ( a%b == 0)
        printf ("%d", a);
}
```

The statement b = x + y is executed every time with the loop. But because it is loop invariant, we can bring it outside the loop. It will then be executed only once.

```
a = 200;
b = x + y;
while (a > 0)
{
        if ( a%b == 0)
        printf ("%d", a);
}
```

6) Induction Variables and Strength Reduction:

An induction variable may be defined as an integer scalar variable which is used in loop for the following kind of assignments i = i + constant.

Strength Reduction means replacing the high strength operator by a low strength operator. Strength Reduction used on induction variables to achieve a more efficient code.

Example:

```
i = 1;
        while (i < 10)
        {
        .
        .
        .
        y = i * 4;
        .
        .
        .
}
```

This code can be replaced by the following code.

```
i = 1;
t = 4;
while (t < 40)
{
        .
        .
        .
        y = t;
```

```
        .
        .
        .
        t = t + 4;
        .
        .
        .
}
```

7) **Variable Propagation:**

If a variable is assigned to another variable, we use one in place of another. This will be useful to carry out other optimization that were otherwise not possible.

Example:

```
c = a * b;
x = a;
.
.
.
d = x * b;
```

Here, if we replace x by a then a * b and x * b will be identified as common sub expressions.

Global Data Flow Analysis:

Certain optimizations can be achieved by examining the entire program and not just a portion of the program.

- User-defined chaining is one particular problem of this kind.
- Here we try to find out as to which definition of a variable is applicable in a statement using the value of that variable.