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.

Position of code generator

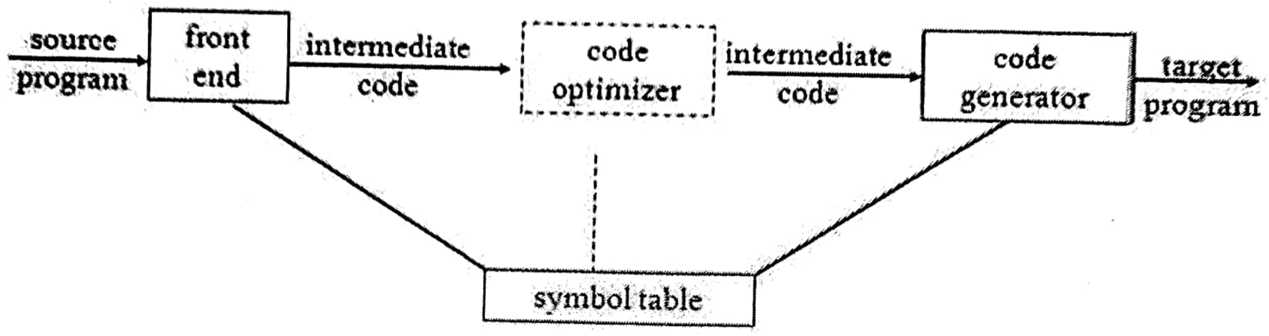


Fig. 4.1 Position of code generator

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :
    a. Linear representation such as postfix notation
    b. Three address representation such as quadruples
    c. Virtual machine representation such as stack machine code

d. Graphical representations such as syntax trees and dags.

- Prior to code generation, the front end must be scanned, parsed andranslated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program.

The output may be :

a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a compiler

c. Assembly language

- Code generation is made easier

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions.

For example,

j : **goto** *i* generates jump instruction as follows :

if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.

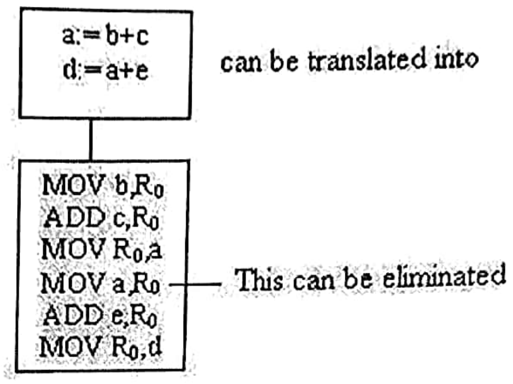
if $i > j$, the jump is forward.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
-
-

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

The former statement can be translated into the latter statement as shown below:



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

can be translated into

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

5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :

Register allocation – the set of variables that will reside in registers at a point in the program is selected.

Register assignment – the specific register that a variable will reside in is picked

- Certain machine requires even-odd *register pairs* for some operands and results.

  For example , consider the division instruction of the form : Div x, y

  where, x – dividend even register in even/odd register pair

  y – divisor even register holds the remainder odd register holds the quotient

6. Evaluation order

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

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word. It has *n*
- general-purpose registers, $R_0, R_1, \ldots, R_{n-1}$.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

MOV	(move *source* to *destination*)
ADD	(add *source* to *destination*)
SUB	(subtract *source* from *destination*)

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	*ADDRESS*	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c+contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ; MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
-
- Address modes involving memory location or literal have cost one.

  Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.

For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.

- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

i) MOV b, R_0

ADD c, R_0 cost = 6

MOV R_0, a

ii) MOV b, a

ADD c, a cost = 6

iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c :

MOV *R_1, *R_0

ADD *R_2, *R_0 cost = 2

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

A SIMPLE CODE GENERATOR

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

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 for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements

a : = b [i] and **a [i] : = b**

Statements	**Code Generated**	**Cost**
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments

a : = *p and ***p : = a**

Statements	**Code Generated**	**Cost**
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

***Generating* Code for Conditional Statements**

Statement	**Code**
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z if x < 0 goto z	MOV y, R_0 ADD z, R_0 MOV R_0,x