Module 30 – Simple code generator and Register allocation

In this module we will try to learn the simple code generator algorithm. We shall also discuss the data structures involved in the simple code generator algorithm. We will conclude this module by understanding the algorithms for register allocation.

30.1 Simple Code Generator

The simple code generator algorithm generates target code for a sequence of three-address statements. The code generator algorithm works by considering individually all the basic blocks. It uses the next-use information to decide on whether to keep the computation in the register or move it to a variable so that the register could be reused. The computation of next-use information is explained in the previous module. We assume that for each operator in the input statement there is a target-language operator. It uses new function *getreg()* to assign registers to variables. The algorithm for code generation initially checks if operands to three-address code are available in registers. After computing the results of two operations, the results are kept in registers till the result is required by another computation or register is kept up to a procedure call or end of block to avoid errors.

For example, consider the following instruction:

- a := b+c

The following sequence would be followed by the code generator algorithm:

- The algorithm initially checks if ‘b’ and ‘c’ are in registers Ri, Rj. If available then the instruction ADD Ri, Rj is generated and this costs 1 and the result is stored in Rj
- If ‘b’ alone is in register and ‘c’ is not in the register, then the instruction ADD c, Ri is generated to a cost of 2
- MOV a, Rj is issued to move the computation to ‘a’.

The issues that need to be resolved are how to get the registers. If the registers are free then there is no issue. If the registers are all occupied, we must free an existing register. Will the instruction generator go for memory based instructions or will necessarily have to go for a register based instruction only is to be considered.

With these issues in consideration, the simple code generator algorithm uses two data structures to resolve, which is discussed in the next section.

30.1.1 Data structures for the Simple code generator algorithm

This algorithm uses two data structures for generating code. The first one is the Register Descriptor which is used to keep track of which variable is currently stored in a register at a

particular point in the code. The second data structure is referred to as address descriptor which is used to keep track of the location where the current value of the variable can be found at run time.

Both these data structures are hash table and have the fields as given in Table 30.1

Table 30.1 Register and Address Descriptor

Register Descriptor		**Address Descriptor**	
Register Name	Contents of the register	Variable Name	Available location

For example consider the following code:

MOV a, R0 after this instruction is executed, the register descriptor of R0 will have its contents column updated as 'a'.

For the following sequence of code, the contents of register descriptor of R0 and R1 will be 'a'. The contents of the address descriptor of the variable 'a' will read **R0** and **R1.**

MOV a, R0
MOV R0, R1

A more simple description for register descriptor would be that, if we query "register name" as input the output will be what variable it contains. On the other hand, if we query the "variable name" to the address descriptor the output will be the location of the variable 'a', which could be address or register.

30.1.2 The Code Generation Algorithm

Algorithm 30.1 SimpleCodeGenerator( )

Input : Sequence of 3-address statements from a basic block.

Output: Assembly language code

For each statement $x := y$ op z

1. Set location $L = getreg(y, z)$ to store the result of y op z
2. If $y \notin L$ then generate
   MOV *y'*,*L*
   where *y'* denotes one of the locations where the value of *y* is available - choose register if possible
3. Generate
   OP *z'*,*L*

where *z'* is one of the locations of *z*;
Update register/address descriptor of *x* to include *L*

4. If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*

The first step in the algorithm is to invoke the getreg() function to get a register to store the result of the computation. The next step is to find the location of the first operand on the LHS. If it is in a register which is found by querying the address descriptor, then the same register is issued. The next step is issue a MOV command to transfer the variable's value into the register. If the variable is already in a register then this instruction could be eliminated. The next instruction is to operate on that register using the other operand and at this point the value of the LHS variable of the input instruction is in the register. So, the address descriptor and register descriptors are updated accordingly. The last step is to find out whether the current variable has a next-use immediately. Depending on the next-use the register content is copied to the variable and the descriptors are updated so that the register could be used for some other instructions.

Algorithm 30.2 getreg ( )

Input: Request for a register

Output: A register or the memory location

1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
   Update address descriptor: value *y* no longer in *R*

2. Else, return a new empty register if available

3. Else, find an occupied register *R*;
   Store contents (register spill) by generating
   MOV *R,M*
   for every *M* in address descriptor of *y*;
   Return register *R*

4. Else Return a memory location

The getreg() function, returns a register if a free register is available. If the value of the variable for which we are trying to issue a MOV instruction is already in a register then the same register is used. If there are no free registers, then an occupied register is identified, it is freed by moving its contents to variable and then is issued. If no such free register could be identified, then the instruction operates on memory location.

Consider the following statement which is part of a high-level language:

d := (a-b) + (a-c) + (a-c)

The corresponding three- address code will be the following, where t, u, v, are temporary variables.

t: = a-b

u := a-c

v := t + u

d := v +u

The code generation sequence is given in Table 30.2.

Table 30.2 Example code generated using SimpleCodeGenerator algorithm

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*	Comments
t := a - b	MOV a,R0 SUB b,R0	Registers empty R0 contains t	t in R0	The assumption is that there are two empty registers R0 and R1. Assuming getreg() issues R0, we generate two instructions. The first MOV copies value of 'a' into R0 and the second instruction computes the subtraction of 'b' from 'a' and the result is in R0. The register and address descriptors are updated to the information that R0 has 't'
u := a - c	MOV a,R1 SUB c,R1	R0 contains t R1 contains v	t in R0 u in R1	'a' and 'c' are not in any register. So, the next empty register R1 is used and the instruction sequence is similar to the previous one. The register descriptor and address descriptor are updated to know about the temporary variable 'u'
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0	't', 'u' are in registers and hence this instruction requires only one instruction ADD. Register R0 is used as 'u' has an immediate next-use. The register descriptor along with the address descriptor are updated to know about the

				variable ‘v’
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory	‘v’ and ‘u’ are in registers and hence ADD is the only instruction and the final result is moved from register to the variable ‘d’. The descriptors are appropriately updated.

To generate target code for instructions involving arrays and pointers Table 30.3 gives an overview of the instruction sequence along with their corresponding cost.

Table 30.3 Sample instructions for arrays and pointers

Statement	**i in Register Ri**		**i in Memory Mi**		**i in Stack**	
	Code	Cost	Code	Cost	Code	Cost
a := b[i]	MOV b[Ri], R	2	MOV Mi, R MOV b[R], R	4	MOV Si(A), R MOV b(R), R	4
a[i] := b	MOV b, a[Ri]	2	MOV Mi, R MOV b, a[R]	5	MOV Si(A), R MOV b, a(R)	5
Statement	**p in Register Rp**		**p in Memory Mp**		**p in Stack**	
	Code	Cost	Code	Cost	Code	Cost
a := *p	MOV *Rp, a	2	MOV Mp, R MOV *R, R	3	MOV Sp(A), R MOV *R, R	3
*p := a	MOV a, *Rp	2	MOV Mp, R MOV a, *R	4	MOV a, R MOV R, *Sp(A)	4

Conditional Statements are part of any programming construct to take an appropriate branch. Conditional jumps are implemented by finding out the value of the register. If the value of a register is negative, zero, positive, non-negative, non-zero, non-positive are the various possibilities to check to branch to a particular situation. The compiler typically uses a set of condition codes to indicate whether the computed quantity of a register is zero, positive or negative

- First case of conditional statement: if x < y goto z - The code that is generated should involve subtracting ‘y’ from ‘x’ which is in register R and then jump to location ‘z’ if R is negative
- Second case of conditional statement: CMP x, y - Sets the condition code to positive if x> y and so on
- CJ < z - Jump to z if value is negative

Consider the following example

- x := y + z
- If x < 0 goto z

The following would be the target code

MOV y, R0

ADD z, R0

MOV R0, x // x is the condition code

CJ < z

30.2 Register Allocation

A primary task of the compiler is register allocation for the variables. The number of registers available in any hardware architecture is very minimal compared to the number of variables that are defined in a particular piece of program. The getreg algorithm is simple but not optimal as the algorithm stores all live variables in registers till the end of a block. The register allocation problem is NP complete. Suppose, if we go in for Global register allocation which involves assigning variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries.

Keeping variables in registers in loops can be beneficial as it avoids register spilling. Suppose loading a variable *x* has a cost of 2 and storing a variable *x* has also a cost of 2, benefit of allocating a register to a variable *x* within a loop *L* is

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

where *use*(*x*, *B*) is the number of times *x* is used in *B* and *live*(*x*, *B*) = true if *x* is live on exit from *B*

Consider the example of basic block and control flow graph of figure 30.1:

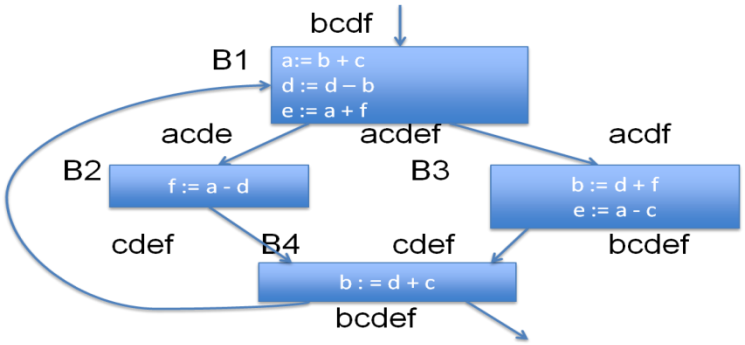

Figure 30.1 Sample graph

Table 30.4 Example global register allocation computation

Block→ ↓ VARIABLE	B1		B2		B3		B4		Total	Comments
	Use	Live	Use	Live	Use	Live	Use	Live		
a	0	1	1	0	1	0	0	0	4	B1 – variable is defined once. Use is 0 as it is defined only here B2 – variable is in the RHS of the expression B3 - variable is in the RHS of the expression B4 – variable is not involved Total is computed using the equation use(x) + 2 * live(x)
b	2	0	0	0	0	1	0	1	6	B1 – variable is defined in two expressions B3 – variable is live as it is in the LHS B4 – same as B3
c	1	0	0	0	1	0	1	0	3	B1 – variable is part of one expression B3 – variable is used is the RHS B4 – variable is in the RHS
d	1	1	1	0	1	0	1	0	6	B1 – defined once and used once B2 – used once B3 – used once B4 – used once
e	0	1	0	0	0	1	0	0	4	B1 – defined once B3 – defined once
f	1	0	0	1	1	0	0	0	4	B1 – used once B2 – defined once B3 – used once

From Table 30.4, we find the maximum cost associated with every variable. A dedicated register is given to the variable that has the maximum cost and is never disturbed during register spilling. A maximum cost indicates that variable is used and defined more.

Global Register Allocation can also be done using a graph coloring algorithm. When a register is needed and all available registers are in use, the content of one of the used registers must be stored to free a register and this is referred to as register spilling. Graph coloring allocates registers and attempts to minimize the cost of spills. An interference graph is built based on how variable interfere with each other. After constructing the graph, the graph coloring algorithm is applied to identify how many colors are at the least required to color this graph and this essentially translates to the number of registers required to compute the sequence of instructions.

Register interference graph is constructed with nodes indicating the variables which indirectly refer to the symbolic registers. An edge between nodes is established such that if one variable is live at a point where other is defined. For the first block B1 of the example in figure 30.1, the interference graph is shown in figure 30.2.

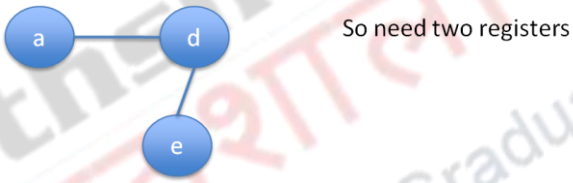

Figure 30.2 Interference graph for Block B1 of figure 30.1

For the graph of figure 30.2, two colors are required. Thus 2 registers are required to compute this basic block B1.

Summary: To summarize, in this module we have discussed the simple code generator algorithm detailing on register descriptors and address descriptors. We also looked at the register allocation algorithm which is based on use and live statistics and also another methodology based on graph coloring.