UNIT-5
CODE GENERATION

Issues involved in code generation – Register allocation – Conversion of three address code to assembly code using code generation algorithm – examples – Procedure for converting assembly code to machine code – Case study

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.

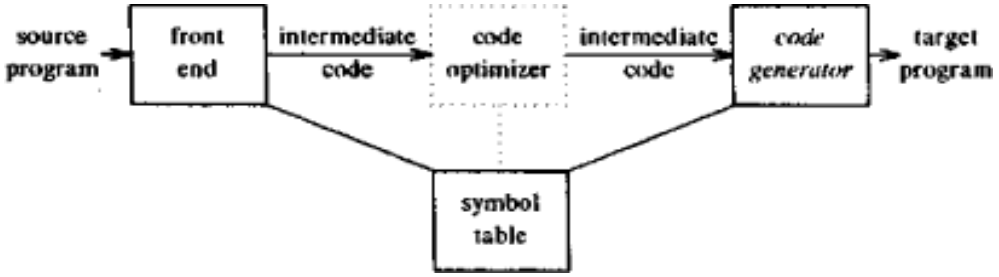


Figure 5.1 Position of code generator

- The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.
- The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:
  - It should carry the exact meaning of the source code.
  - It should be efficient in terms of CPU usage and memory management.

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 :

1)Linear representation such as postfix notation

2)Three address representation such as Quadruples

3)Virtual machine representation such as stack machine code

4)Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated 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.

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. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

Instruction selection:

1. The instructions of target machine should be complete and uniform.

2. Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
3. The quality of the generated code is determined by its speed and size.

For example:

Every three-address statement of the form

x=y+z

where x, y and z are statically allocated.

Code sequence generated is shown as:

```
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,

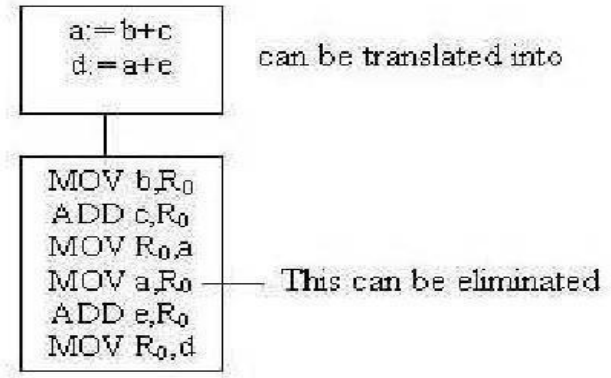


Figure 5.2 Code Translation

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.

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

instead of ,

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

Register allocation:

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

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

Register assignment - the specific register that a variable will reside is selected.

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 in even register in even/odd register pair, y – divisor in even register holds the remainder odd register holds the quotient

Evaluation order:

At last, the code generator decides the order in which the instruction will be executed. 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.

- Picking the best order is a difficult task.
- Initially avoid this problem by generating code for the three address statements in the order in which they have been produced by the intermediate code generator.
- It creates schedules for instructions to execute them.

When instructions are independent, their evaluation order can be changed to utilize registers and save on instruction cost. Consider the following instruction:

a+b-(c+d)*e

The three-address code, the corresponding code and its reordered instruction are given below:

Three-address code	Code	Reordered three-address code	Code	Inference

t1:=a+b
t2:=c+d
t3:=e*t2
t4:=t1-t3
	MOV a,R0
	ADD b,R0
	MOV R0,t1
	MOV c,R1
	ADD d,R1
	MOV e,R0
	MUL R1,R0
	MOV t1,R1
	SUB R0,R1
	MOV R1,t4
		t2:=c+d
		t3:=e*t2
		t1:=a+b
		t4:=t1-t3
			MOV c,R0
			ADD d,R0
			MOV e,R1
			MUL R0,R1
			MOV a,R0
			ADD b,R0
			SUB R1,R0
			MOV R0,t4
				The reordered instructions reduced the number of final code by 2 and thus saved in cost. The three-address code is reordered so that t1 is computed after computing t2 and t3. This reordering has saved in the instruction cost.

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, R0, R1, . . . , Rn-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.

Table 5.1 Mode and address allocation table

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

For example : *contents(a)* denotes the contents of the register or memory address represented by a. A memory location M or a register R represents itself when used as a source or destination.

e.g. MOV R0,M → stores the content of register R0 into memory location 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 :

1. The instruction **MOV R0, R1** copies the contents of register R0 into R1. It has cost

one, since it occupies only one word of memory.

2. The (store) instruction **MOV R5,M** copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
3. The instruction **ADD #1,R3** adds the constant 1 to the contents of register 3,and has cost two, since the constant 1 must appear in the next word following the instruction.
4. The instruction **SUB 4(R0),*12(R1)** stores the value contents(contents(12+contents(R1)))-contents(contents(4+R0)) into the destination *12(R1). Cost of this instruction is three, since the constant 4 and 12 are stored in the next two words following 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 :

```
MOV b, R0
ADD  c, R0
MOV R0, a                    cost = 6

MOV b, a
ADD c, a                     cost = 6
```

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.

Run-Time Storage Management:

Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.

The two standard storage allocation strategies are:

- Static allocation
- Stack allocation

In static allocation, the position of an activation record in memory is fixed at compile time.

In stack allocation, a new activation record is pushed onto the stack for each execution of a

procedure. The record is popped when the activation ends.

The following three-address statements are associated with the run-time allocation and deallocation of activation records:

- Call
- Return
- Halt

We assume that the run-time memory is divided into areas for:

- Code
- Static data
- Stack

Static allocation:

- In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes.
- As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* +20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,

callee.static_area – Address of the activation record

callee.code_area– Address of the first instruction for called procedure

#here +20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :

GOTO * *callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation:

- Using the relative address, static allocation can become stack allocation for storage in activation records.
- The position of the record for an activation of a procedure is not known until run time.
- In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.
- The indexed address mode of target machine is used for this purpose.
- Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

```
MOV #stackstart , SP   /* initializes stack */
Code for the first procedure
HALT /* terminate execution */
```

Implementation of Call statement:

```
ADD #caller.recordsize, SP        /* increment stack pointer */
MOV #here +16, *SP                /*Save return address */
GOTO callee.code_area
```

where,

caller.recordsize – size of the activation record

#here +16 – address of the instruction following the GOTO

Implementation of Return statement:

```
GOTO * (SP ) /*return to the caller */
SUB #caller.recordsize, SP    /* decrement SP and restore to previous value */
```

Basic Blocks and FlowGraph:

In compiler design, a basic block is a straight-line piece of code that has only one entry point and one exit point. Basic block construction is the process of dividing a program's control flow graph into basic blocks.

The task is to partition a sequence of three-address codes into the basic block. The new basic block always begins with the first instruction and continues to add instructions until a jump or a label is reached. If no jumps or labels are identified, control will flow sequentially from one instruction to another.

Task: Partition a sequence of three-address codes into basic blocks.

Input: Sequence of three address statements.

Output: A sequence of basic blocks.

Also See, Top Down Parsing

Algorithm

First, find the set of leaders from intermediate code, the first statements of basic blocks. The following are the steps for finding leaders:
The first instruction of the three-address code is a leader.
Instructions that are the target of conditional/unconditional goto are leaders.
Instructions that immediately follow any conditional/unconditional goto/jump statements are leaders.
For each leader found, its basic block contains itself and all instructions up to the next leader.

Hence following the above algorithm, you can partition a sequence of three-address code into basic blocks.

Example
Consider the source code for converting a 10 x 10 matrix to an identity matrix.

```
for r from 1 to 10 do
    for c from 1 to 10 do
        a [ r, c ] = 0.0;

for r from 1 to 10 do
      a [ r, c ] = 1.0;
```

The following are the three address codes for the above source code:

```
1) r = 1
 2) c = 1
 3) t1 = 10 * r
 4) t2 = t1 + c
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) c = c + 1
 9) if c <= 10 goto (3)
 10) r = r + 1
 11) if r <= 10 goto (2)
 12) r = 1
 13) t5 = c - 1
 14) t6 = 88 * t5
 15) a[t6] = 1.0
 16) r = r + 1
 17) if r <= 10 goto (13)
```

There are six basic blocks for the above-given code, which are:

B1 for statement 1
B2 for statement 2
B3 for statements 3-9
B4 for statements 10-11
B5 for statement 12
B6 for statements 13-17.

Explanation:
According to the definition of leaders given in the above algorithm,

- Instruction 1 is a leader as the first instruction of a three-address code is always a leader.
- Instruction 2 is a leader because it is followed by a goto statement at Instruction 11.
- Instruction 3 and Instruction 13 are also leaders because they are followed by a goto statement at Instruction 9 and 17, respectively.
- Instruction 10 and Instruction 12 are also leaders because they are followed by a conditional goto statement at Instruction 9 and 17, respectively.

Properties of Flow Graphs

The control flow graph is process-oriented.
A control flow graph shows how program control is parsed among the blocks.
The control flow graph depicts all of the paths that can be traversed during the execution of a program.
It can be used in software optimization to find unwanted loops.

Representation of Flow Graphs

Flow graphs are directed graphs. The nodes/bocks of the control flow graph are the basic blocks of the program. There are two designated blocks in Control Flow Graph:

Entry Block: The entry block allows the control to enter in the control flow graph.
Exit Block: Control flow leaves through the exit block.

An edge can flow from one block A to another block B if:

the first instruction of the B's block immediately follows the last instruction of the A's block.
there is a conditional/unconditional jump from A's end to the starting of B.
B follows X in the original order of the three-address code, and A does not end in an unconditional jump.

Let's see an example,

Consider the source code for converting a 10 x 10 matrix to an identity matrix.

```
for r from 1 to 10 do
    for c from 1 to 10 do
        a [ r, c ] = 0.0;

for r from 1 to 10 do
```

```
a [ r, c ] = 1.0;
```

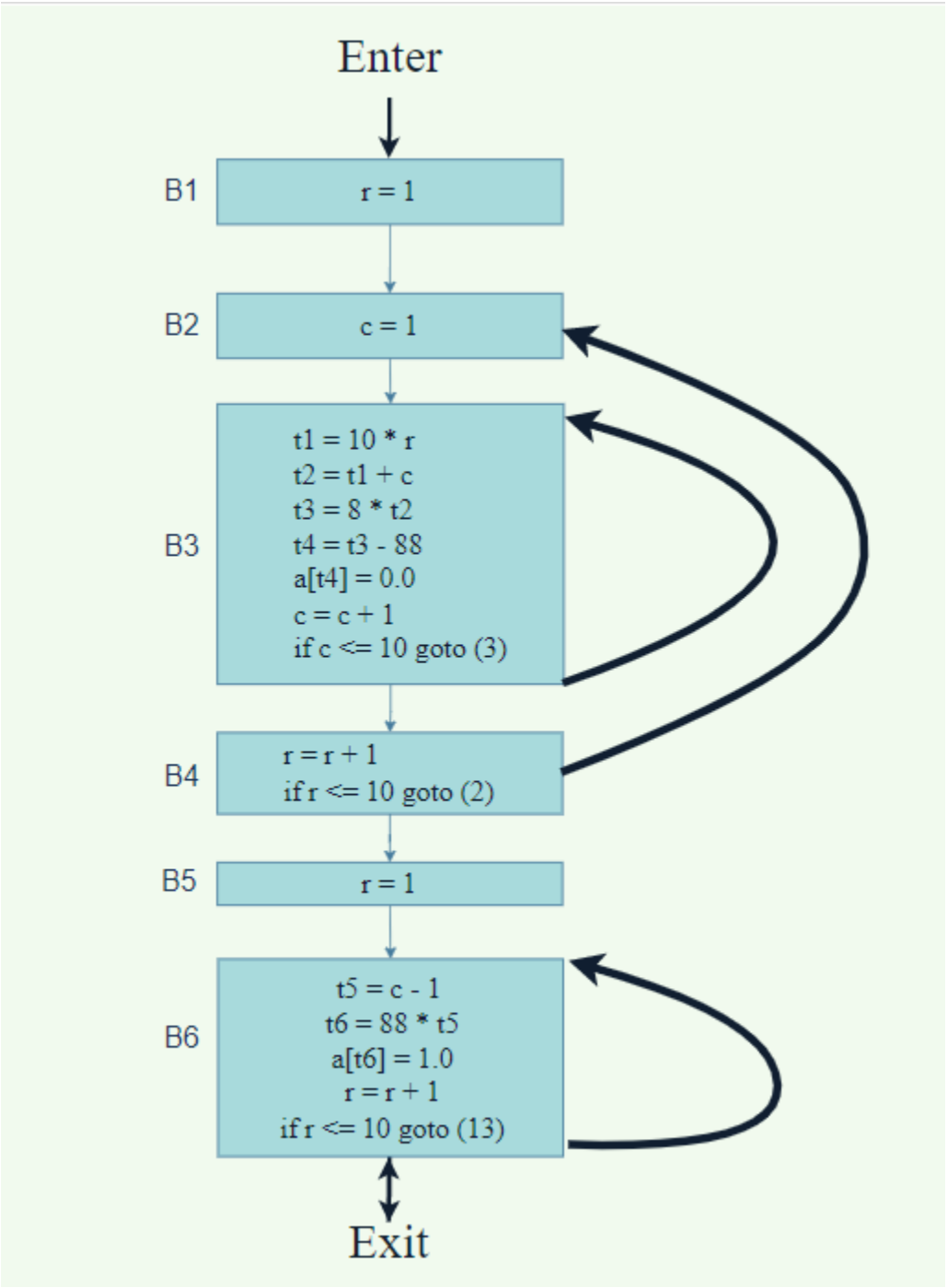
The following are the three address codes for the above source code:

```
1) r = 1
 2) c = 1
 3) t1 = 10 * r
 4) t2 = t1 + c
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) c = c + 1
 9) if c <= 10 goto (3)
 10) r = r + 1
 11) if r <= 10 goto (2)
 12) r = 1
 13) t5 = c - 1
 14) t6 = 88 * t5
 15) a[t6] = 1.0
 16) r = r + 1
 17) if r <= 10 goto (13)
```

There are six basic blocks for the above-given code, which are:

B1 for statement 1
B2 for statement 2
B3 for statements 3-9
B4 for statements 10-11
B5 for statement 12
B6 for statements 13-17.

The control flow graph of the above-given basic blocks is:



Explanation:

- B1 is the start point for the control flow graph because B1 contains the starting instructions.
- Because B1 does not end with unconditional jumps, and the B2 block's leader immediately follows B1's leader, B2 is the only successor of B1.
- There are two successors to the B3 block. The conditional jump in the last instruction of block B3 is targeted at the first instruction of the B3 block; therefore, one is block B3 itself. Another is block B4 due to conditional jump at the end of the B3 block.
- The last block, B6, is the exit point of the control flow graph.

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 can have the following sequence of codes:

MOV b,Ri; cost=2 //b is moved to Ri register

MOV c,Rj; cost=2 //c is moved to Rj register

ADD Rj, Ri Cost = 1 // if Ri contains b and Rj contains c

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:

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

- 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.
- 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.
- If the current values of y or z have no next uses, are not live on exit from the block, and are inregisters, 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

Code sequence for the example is:

Table 5.2 Shows the code sequence and the register allocation

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

Generating code for Indexed Assignments

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

Generating code for Pointer Assignments

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

Generating code for Conditional Statements

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

Register Allocation and Assignment:

Register allocation is only within a basic block. It follows top-down approach.

Local register allocation

- Register allocation is only within a basic block. It follows top-down approach.
- Assign registers to the most heavily used variables
- Traverse the block
- Count uses
- Use count as a priority function
- Assign registers to higher priority variables first

Need of global register allocation:

- Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.
- To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:

- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Usage Counts:

A simple method of determining the savings to be realized by keeping variable x in a register

for the duration of loop L is to recognize that in our machine model we save one unit of cost for each reference to x if x is in a register.

An approximate formula for the benefit to be realized from allocating a register to x within a loop L is:

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

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 0 otherwise.

Example: Consider the basic block in the inner loop in Fig 5.3 where jump and conditional jumps have been omitted. Assume R0, R1 and R2 are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in the figure.

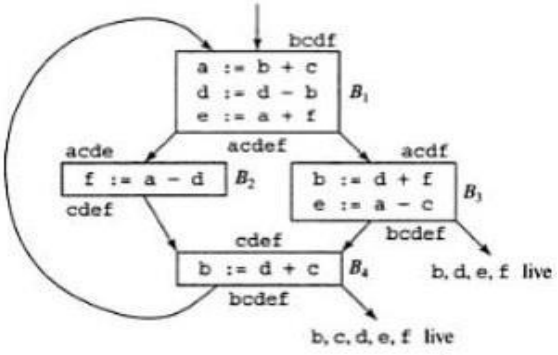


Fig 5.3 Flow graph of an inner loop

Fig 5.4 shows the assembly code generated from Fig 5.3.

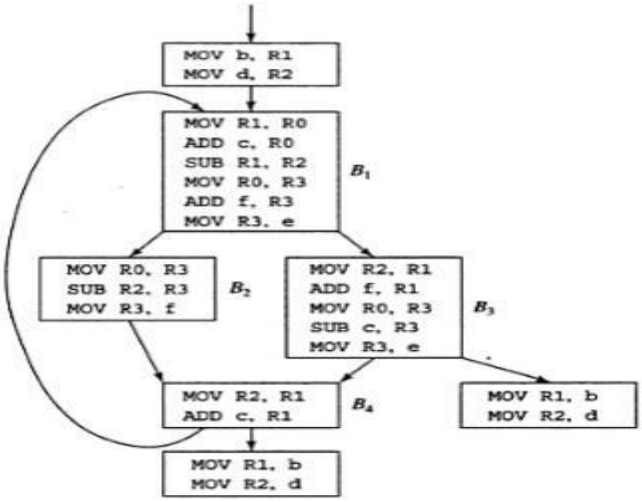


Fig 5.4 Code sequence using global register assignment