Chapter 7. Object Code Generation

1. Introduction
2. Blind Generation
3. Context Sensitive Generation
   a) Live/next-use
   b) Register Management
   c) OC Generation
4. Instruction Sequencing

M. Paran Venkat Virah Kesireddy
Teaching Associate
Mail: pkesireddy@fullerton.edu

Inputs from Prof Doina Bein & Prof James Choi

7.1 Introduction

OC is a phase where the IC is translated into the language of the target machine.

In our case 3AC -> Assembly Code **Types of Assembly Instructions: Memory-to-Register**
Register-to-Register
Memory-to-Memory
IN our case , assume NO M-M instructions
⇒<u>Need at least one register per instruction</u>

Issues are:

- **Which instruction to use**

This refers to the selection of the appropriate assembly instruction to represent the desired operation**.**

- **How to avoid redundant operations**

This issue concerns the optimization of code to minimize redundant or unnecessary operations

- **How to manage the register**

Register management is critical to ensure that registers are effectively used throughout the code generation process. This involves tasks like register allocation (assigning variables to registers), tracking the liveness of variables, and managing register spills to memory when necessary.

7.2 Blind Generation

Idea: For each 3AC instruction, we have a "template" of assembly instructions

E.g.,
3AC: x = y + z

Template:
MOV Rx,
y ADD
Rx,
Z
STO x,
Rx

MOV: Copy
ADD: Perform Addition
STO: Store Data in Memory

Problem is:
We could have redundant instructions:

E.g.;
a = b+c;
d = a+b;

SO, we generate 6 instructions:
MOV Rx, b
ADD Rx, c
STO A, Rx
MOV Rx, a (NOT necessary – not an "intelligent use") ADD Rx, b
STO d, Rx

7.3 Context Sensitive Generation

IDEA: Makes use of values that exist in the register, and If a value in a register and going to be used keep it in the register otherwise clear the register.

Issues: How do we keep track of?

- **Which registers are used and what do they hold?**

The compiler needs to maintain a register allocation table or data structure to track the status of each register.

- **Where the current value of a variable can be found?**

The compiler must determine where the current value of a variable can be found. This is particularly important when values are not kept in registers and need to be loaded from memory.

- **Which variable will be used later and where?**

The compiler needs to anticipate variable usage to decide when to keep values in registers.

- **Which variables in the register must be saved for later purpose, if it has to give up the register**

Solutions:

a)Register Association table hold information about the variables

b)Address table hold all info about the variables (where the value is stored(R or M. And which R)

However, to address (c) and (d) we need to have the concept of live and next-use

Live and Next-Use Analysis:

Live and next-use analysis is a crucial technique used to determine the lifetime and usage of variables within a program. It helps address points (c) and (d) mentioned in the context-sensitive code generation:

Live Analysis: This analysis determines which variables are live (in use) at any point in the program. It helps the compiler decide when it's necessary to keep a variable's value in a register to avoid unnecessary memory accesses. Variables that are live are those that will be used later in the program.

Next-Use Analysis: Next-use analysis predicts when a variable will be used next in the program. This information is valuable for determining when a variable can be safely removed from a register to make space for other variables or operations.

a) Live/Next-Use

Live: A variable is said to be "Live" if it is going to be used again
Next-Use: where it is going to be used

Q: How do we find this information?

One way: By searching the instructions from the top

Ex: source d= (a-b) + (c-a) – (d+b)*(c+1) 3AC:

1. **u = a-b u(L, 3), a(L, 2), b(L, 4)**
2. **v = c-a**
3. **w = u+v**
4. **x = d+b**
5. **y = c+1**
6. **z = x*y**
7. **d = w-z**

However, the operation is O(N*N)
The (L, n) notation shows the line number where each variable is defined

More efficient way : Start from the last instruction backwards and utilize symbol table

Step1. In the symbol table, mark all user-created variables “live” and temporaries with “dead”

Step 2.
For each 3 AC instruction a = b (op) c backwards do

- **look-up the symbol table and attach Live/next-use information**
- **Update the symbol table as follows mark the target “a” to be “dead”**
  mark the operands “b” and “c” “ to be “live” and set next-use to be the current instruction

u = a–b

v = c–a

w = u+v

x = d+b

y = c+1

z = x*y

d = w–z

Ex. source: d = (a-b) + (c-a) – (a+b)*(c+1)

1.	u = a-b	u(L, 3), a(L, 2), b(L, 4)
2.	v = c-a	v(L,3), c(L, 5), a(L,N)
3.	w = u+v	w(L, 7), U(D, N), v(D, N)
4.	x = d+b	x(L, 6), d(D, N), b(L, N) y(L, 6), c(L, N)
5.	y = c+1	z(L, 7), X(D, N), y(D, N)
6.	z = x*y	d(L, N), w(D, N),
7.	d = w-z	z(D,N)

Symbol Table

Variable (initial)	Liveness	Next-use
a	L	N
b	L	N
c	L	N
d	L	N
u	D	N
v	D	N
w	D	N
x	D	N
y	D	N
z	D	N

2.Register Management

Since we need at least one register for operation (RM, RR instruction), we have to manage the limited number of registers.

```
/* Find an available register for a 3AC instruction a = b (op) c  */

function GetReg(b)
{
     If (b is in register R) and (R does not hold another variable) then
          return R
else if there is an unused register R, then return R
     else /* all register are used */
  Select an occupied register R for use
        Store the content of R in memory
}Return R
```

3.Context Sensitive Generation of the Code

Algorithm for Code Generation

```
Procedure GenCode (3ac: a = b (op) c, or  a =b)
{
   R = GetReg(b);                                  load the value of b from
   If R is empty,      then generate "L   R, b"    memory
   If 3ac is "a =b" then, change the tables so that R holds
        also "a" and return;
Find "C" using the address table
   If "C" is in memory then generate "       R, c"
   op else if "C" is in a Register S, then
                {
                Generate "op R, S"
                If  "C" is dead then free the Register
                S
                }
  Update tables
  }
```

Example:

Source code : d = (a-b) + (c-a) – (d+b) * (c+1)

1. u = a-b	u(L,3), a(L,2), b(L,4)
2. v = c-a	v(L,3), c(L,5), a(L,N)
3. w = u+v	w(L,7), U(D, N), V(D,N)
4. x = d+b	x(L,6), d(D,N), b(L,N)
5. y = c+1	y(L,6), C(L,N)
6. z = x*y	z(L,7), x(D,N), y (D, N)
7. d = w – z	d (L, N), w(D, N), Z (D,N)

Register ASSN Table

Address Table

2	a m
3	b m
4	c m
5	d m
6	u -
7	v -
8	w -
9	x -
10	y -
	z -

```
Procedure GenCode(3ac: a = b (op) c, or  a =b)
{
  R = GetReg(b);
  If R is empty,  then generate "L   R, B"
  If 3ac is "a =b" then, change the tables so that R holds also
      "a" and return;
  Find "C" using the address table
  If "C" is in memory then generate " op    R, c"
  Else if "C" is in a Register S, then
          {
          Generate "opR   R, S"
          If  "C" is dead then free the Register S
          }
 Update tables
 }
```

Load
Subtract

(1) u = a –b

GetReg(a) = 2

L 2, a

S 2, b

(2) v = c –a

GetTeg(c) = 3

L 3, c

S 3, a

1.	u = a-b	u(L,3), a(L,2), b(L,4)
2.	v = c-a	v(L,3), c(L,5), a(L,N)
3.	w = u+v	w(L,7), U(D, N), V(D,N)
4.	x = d+b	x(L,6), d(D,N), b(L,N)
5.	y = c+1	y(L,6), C(L,N)
6.	z = x*y	z(L,7), x(D,N), y (D, N)
7.	d = w – z	d (L, N), w(D, N), Z (D,N)

```
Procedure GenCode(3ac: a = b (op) c, or  a =b)
{
   R = GetReg(b);
   If R is empty,  then generate "L   R, B"
   If 3ac is "a =b" then, change the tables so that R holds also
        "a" and return;
   Find "C" using the address table
   If "C" is in memory then generate " op    R, c"
   Else if "C" is in a Register S, then
            {
            Generate "opR   R, S"
            If  "C" is dead then free the Register S
            }
 Update tables
 }
```

1.	u = a-b	u(L,3), a(L,2), b(L,4)
2.	v = c-a	v(L,3), c(L,5), a(L,N)
3.	w = u+v	w(L,7), U(D, N), V(D,N)
4.	x = d+b	x(L,6), d(D,N), b(L,N)
5.	y = c+1	y(L,6), C(L,N)
6.	z = x*y	z(L,7), x(D,N), y (D, N)
7.	d = w – z	d (L, N), w(D, N), Z (D,N)

(3)w = u + v GetReg(v) = 3

```
AR 3,2    /* R2 can be freed */
/* Add Register */
```

(4)x = d + b GetReg(d) =

```
Procedure GenCode(3ac: a = b (op) c, or  a =b)
{
   R = GetReg(b);
   If R is empty,  then generate "L   R, B"
   If 3ac is "a =b" then, change the tables so that R holds also
        "a" and return;
   Find "C" using the address table
   If "C" is in memory then generate " op    R, c"
   Else if "C" is in a Register S, then
            {
            Generate "opR   R, S"
            If  "C" is dead then free the Register S
            }
 Update tables
 }
```

1.	u = a-b	u(L,3), a(L,2), b(L,4)
2.	v = c-a	v(L,3), c(L,5), a(L,N)
3.	w = u+v	w(L,7), U(D, N), V(D,N)
4.	x = d+b	x(L,6), d(D,N), b(L,N)
5.	y = c+1	y(L,6), C(L,N)
6.	z = x*y	z(L,7), x(D,N), y (D, N)
7.	d = w – z	d (L, N), w(D, N), Z (D,N)

(5) y = c+1
Getreg(c) = 5
L 5, c
A 5, = F'1'

(6)z = x * y /* need even-odd registers for multiply */ GetReg(x) = 4
MR 4,5

```
Procedure GenCode(3ac: a = b (op) c, or  a =b)
{
   R = GetReg(b);
   If R is empty,  then generate "L   R, B"
   If 3ac is "a =b" then, change the tables so that R holds also
        "a" and return;
   Find "C" using the address table
   If "C" is in memory then generate " op    R, c"
   Else if "C" is in a Register S, then
            {
            Generate "opR   R, S"
            If  "C" is dead then free the Register S
            }
 Update tables
 }
```

1.	u = a-b	u(L,3), a(L,2), b(L,4)
2.	v = c-a	v(L,3), c(L,5), a(L,N)
3.	w = u+v	w(L,7), U(D, N), V(D,N)
4.	x = d+b	x(L,6), d(D,N), b(L,N)
5.	y = c+1	y(L,6), C(L,N)
6.	z = x*y	z(L,7), x(D,N), y (D, N)
7.	d = w – z	d (L, N), w(D, N), Z (D,N)

(7) d = w-z

GetReg(w) =3

SR 3, 4

In total we get 11 instructions VS. 21 instructions in BLIND generation => About 45 % instruction savings

7.4 Instruction Sequencing

Sethi-Ullman Method or Minimum use of Registers

If we need one register per 3AC instruction, it is important to minimize the usage of the Registers.

Idea: We change the sequence of 3Ac instructions such that the new sequence uses Less number of registers

Assume We have an AST (!!), otherwise we have to create one.

Step1: Label(Give a count to) the nodes in AST as follows:

Leaves: The left node gets a count of 1 and right node gets a count of 0 Roots: If the subtrees have the same count K, then K+1

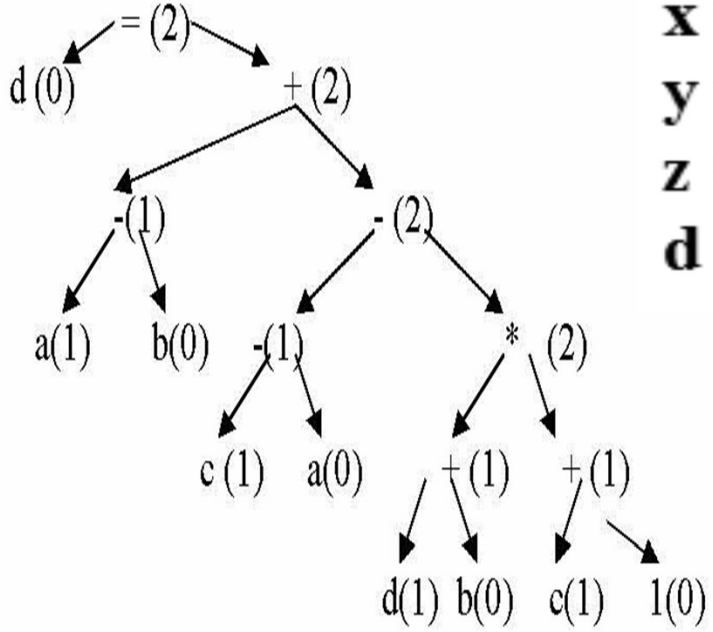
else the greater count of two

Step 2: Generate 3Acs such that more expensive subtree is evaluated first. If the subtrees have the same count, then favor the right subtree (or left) – consistent

Step 3: Generate the OC from the new sequence

EX: d = (a-b) + (c-a) – (d+b)*(c+1)

Step1:



u = a-b
v = c-a
w = u+v
x = d+b
y = c+1
z = x*y
d = w-z

Step2:

1. t1 = c +1
2. t2 = d +b
3. t3 = t2 * t1
4. t4 = c –a
5. t5 = t4 – t3
6. t6 = a –b
7. d = t6 + t5

Step 3:
We can see that we need two registers only

EN
D