Unit – 5 Code Generation

- Introduction
- Difficulties and Issues in the Design of Code generator
- Target Language and Target machine
- Program and instruction Cost.
- Code optimization principles
- Code generation Algorithm

Introduction

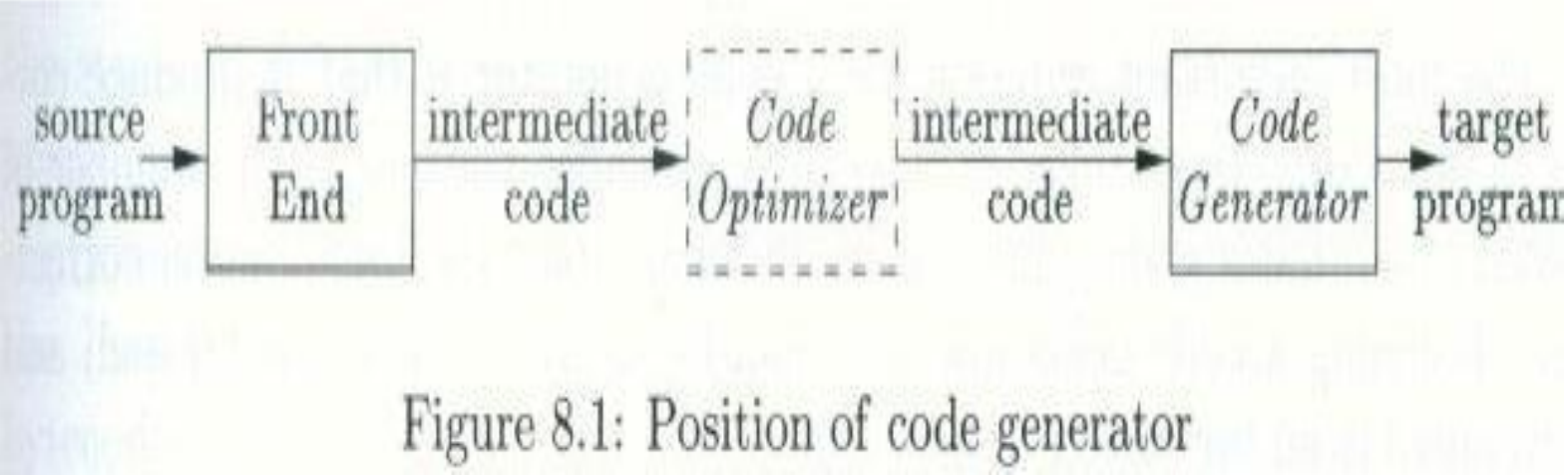


Figure 8.1: Position of code generator

1. This is the final phase in the compiler model that takes as input the Intermediate Representation(IR) produced by the front end of the compiler along with the relevant information in the symbol table and produces the semantically equivalent target program.

Introduction

2. The target program must preserve the semantic meaning of the source program and be of high quality in terms of space and execution.
3. The compiler that need to produce target programs must include optimization phase prior to code generation or during code generation.
4. The code generation has to perform three major tasks that are identified as
    1. **Instruction selection,**
    2. **register allocation and assignment** and
    3. **instruction ordering**.

Difficulties and issues in Code Generation

- Deciding what target machine instruction to generate for a particular computation that implements the IR statements i.e., ***Instruction Selection. (Dynamic Prog..)***

  Example : TAC – A = A + 1

  IR-1 – AOS A or IR-2- LD R A ,
  ADD R, R, #1
  ST R A

- Deciding which order of computation should be selected to schedule the execution of statements. i.e., ***Instruction ordering. (*** order of computation that uses fewer registers to hold the intermediate results).

  (Node Listing Algorithm)

- Deciding what value to keep in which registers and optimal assignment of registers to variables. i.e., ***Register allocation and assignment.***

(Register Descriptor and Address descriptor)

Target language and machine

- Familiarity with the target machine and its instruction set is a prerequisite for generating a code for the source language.
- We shall assume the Target language as the assembly language for a simple computer DEC-PDP –II.
- The target computer models a three address machine with **load and store operations, computation operations, jump operations and conditional jumps**.
- The underlying computer is byte addressable machine with a general purpose registers R0,R1……Rn-1.
- We shall use a limited set of instructions that can be mapped to **IR-intermediate Representation** of a source statement.
- Let us assume all operands are of types integer.
- Instruction consists of an operator, followed by a target, and a list of operands i.e., **OP result, operand1, operand2**

- We assume the following kinds of instructions.
  1. Load operations : **LD dst, addr**

     Example: LD r, z and LD r1,r2

  2. Store Operations : **ST X, r**

  3. Computation operations : OP dst, src1, src2 where OP is a operation code like ADD,SUB…n and dst scr1, src2 are memory locations.

     Example: SUB r1,r2,r3

  4. Unconditional Jump : **BR L**

  5. Conditional jump : **Bcond r, L**

     Example : BLTZ r, L

- We also assume the standard addressing modes.

Examples to generate code for the three address statements assuming all variables are stored in memory locations

1.x= y-z

```
LD R0, y
LD  R1,z
SUB R0, R0, R1
ST x, R0
```

2. b=a[i] : here a is an array whose elements are 8 byte values and elements of array a are indexed starting at 0

```
LD R1, I              // R1 =1
MUL R1, R1, 8     // R1=R1+8
LD R2, a(R1)       // R2 = contents( a + contents(R1))
ST b, R2              // b = R2
```

3. a[j]=c

```
LD R1, C
LD R2, j
MUL R2,R2, 8
ST a(R2), R1
```

4. If x < y goto L

```
LD R1, x
LD R2, y
SUB R1, R1, R2
BLTZ R1, M
```

Program and Instruction cost

- Determining the actual cost of compiling and running a program is a complex problem and finding the optimal target program for a given source program is a un decidable problem.
- In general we use heuristic technique that produces good but not necessarily a optimal target program and assume the followings for the target language instructions and their associated cost

Instruction LD R0,R1 has a cost one because no additional memory reference

Instruction LD R0, M has a cost two because memory reference.

Instruction LD R1, *100(R2) has a cost three because two memory references.

Examples on Code Genaration

- Generate code for the following three-address statements assuming all variables are stored in memory locations.

Examples :

1. x=1
2. x=a
3. x=a+1
4. x = a +b

Answers :

1.
```
LD R0, #1
ST x R0
```
2.
```
LD R0, a
ST x, R0
```
3.
```
LD R0, a
ADD R0, R0, #1
ST x, R0
```
4.
```
LD  R0, a
LD  R1, b
ADD R0, R0, R1
ST x, R0
```

Examples on Code Genaration

- Generate code for the following three-address statements assuming all variables are stored in memory locations.
- a and b arrays whose elements are 8 byte values and elements of an array a and b are indexed starting at 0

Examples :

1. x=b + c
   y=a + x
2. x=a[i]
   Y=b[j]
3. a[i] =y
   b[j]=x

Answers :

1. LD R0, b
   LD R1, c
   ADD R0, R0, R1
   LD R1, a
   ADD R1, R1, R0
   ST x, R0
   ST y, R1.

Answers :

2. LD R0, I
   MUL R0, R0,#8
   LD R1, a(R0)
   ST x, R1
   LD R0, j
   MUL R0, R0,#8
   LD R1, b(R0)
   ST y, R1
3. Homework

Code optimization Principles

- What is code optimization?
- How to measure the quality of the object program?
- Different sources of optimization
- Loop optimization
  - Construction of three address code.
  - Partitioning three address code into Basic block.
  - Representation of Basic block.
  - Construction of flow graph from basic block.
  - Determination of loop
  - Code motion and Removal of Induction variable
  - Reduction in strength.

What is code optimization ?

- Code optimization refers to an important activity where sophisticated techniques employed by the complier to intermediate representation in order to get the optimal object program which is most obvious for a given source program.
- It mainly deals with rearranging the computations in a program to gain the **advantage of execution speed and use of memory** without changing the meaning of the whole program
- Optimal program refers to efficient code measured in terms time (Faster) and space (less)

Quality of the object program

- The quality of the object program is basically measured in terms of size and running time. Also the machine on which the object program is executed.
- For large computers running time is important and for small computers memory size is particularly important.

Possible Sources of optimization

1. Detecting patterns in the program and replacing these patterns by equivalent and more efficient construct.
   - Example could be multiplication operation may be replaced by addition and division operation may be replaced by subtraction
   - X2 is replaced by X * X, 2 * X is replaced by X + X and X/2 is replaced by x * 0.5
2. The richest source of optimization is in the efficient utilization of registers and instructions from the instruction set of the machine.
   - This is achieved during the code generation by maintaining the register descriptor and register descriptor.
3. An identification of common sub expression and eliminating them.
   - DAG is used to catch the common sub-expression.
   - For example A[I+1] = B[i+1] statement could be replaced by T=I+1 and A[T]=B[T]

     or a= b*c

     x=b*c +5

     could be replaced by

     t1= b*c

     a= t1

     t2= t1 + 5

     x=t2

Examples on Finding local Common sub expression and elimination

Example -1

```
a=b + c
b=a – d
c= b + c
d= a – d
```

Note : variable b is not live on exit of the block

Example – 2

```
d=b * c
a=a + b
b= b * c
a= e – d
```

Note : 1. Only the ‘a’ is live on exit of the block

2. a, b, and c are live on exit of the block

4. Compilation time computation or constant folding

- It is a process of executing the source program at compile time where the operand values are known before runtime. It is also called as constant folding :

Example :

```
I= I + 1
I= 3
A= 4 * I
```

→

```
T1 = I + 1
I  = T1
I  = 3
T2  = 4 * I
A   = T2
```

→

```
T1 = I + 1
I = T1
I= 3
A= 12
```

→ Dead code

Note: I and A are live on exit. Also all temporaries are dead on exit

Loop Unrolling

5. In Loop unrolling : If number of iterations are constant then we can reduce the replicating the body of the loop to reduce the number of iterations.

   For example :

- ```
  I=1
  While(I<100)
  {
     x[I]=0;
      I=I+1;
  }
  ```

The above segment of code could be replaced by

- ```
  I=1
  While(I< 50)
   {
     x[I]=0;
      I=I+1;
     x[I]=0;
       I=I+1;
   }
  ```

6. . Code optimization by Copy propagation and dead code elimination

- Copy propagation technique could be applied for assignment statement of the form **f=g. Though it is not a direct optimization but it gives an opportunity to remove dead code and identification of common sub-expression.**
  - **Example 1:**

    x= t1 **x= t1**

    a[t2] = t3 **a[t2]= t3**

    a[t3]= x **a[t3] = t1**

    Here x=t1 is a dead code and could be later eliminated.
  - **Example 2:**

    c=d **c=d**

    x= c+e **x=d+e**

    z= d+e-10.5 **z=d+e -10.5**

    Here the statement x= c+e could be modified as x= d+e by propagating the variable 'd' to it.

    This creates the possibility of identifying the common sub-exprssion

7. Loop optimization

Next important class concerns the handling of loops which removes the loop invariant computation and the induction variable.

- Begin

```
prod=0;
I=1;
do
Begin
    prod=prod + a[i] *b[i]
    I=I+1
end Until I <= 20
```

End

Assumption : Consider the machine with four bytes per word. i.e., word length

Three address code statement for the source construct

1. prod=0
2. I=1
3. T1 =4*I
4. T2 =add(a) -4
5. T3 = T2[T1]
6. T4= add(b) -4
7. T5 = T4[T1]
8. T6 = T3 * T5
9. Prod = Prod + T6
10. I= I+1
11. If I<=20 Goto 3
12. ---

7. Loop Optimization

- First step towards loop optimization is to break or partition the TAC's into one more basic blocks.
- What is Basic Block?

  It is a sequence of consecutive TAC statements which may be entered only at the beginning and when entered, all the statements are executed in sequence without halt or possibility of branch. i.e., if one statement is executed then all the statements of that block are executed in sequence without halt or there is possibility of branch

Algorithm for partitioning the TAC's into Basic Blocks

1. Determine the set of leaders i.e., the first statements of Basic Blocks.

   Rules for Leader

   a. The First TAC statement in the Intermediate code is the leader.

   b. Any statement which is target of a conditional statement or unconditional goto statement is a leader.

   c. Any statement TAC statement which immediately follows a conditional goto is a leader.

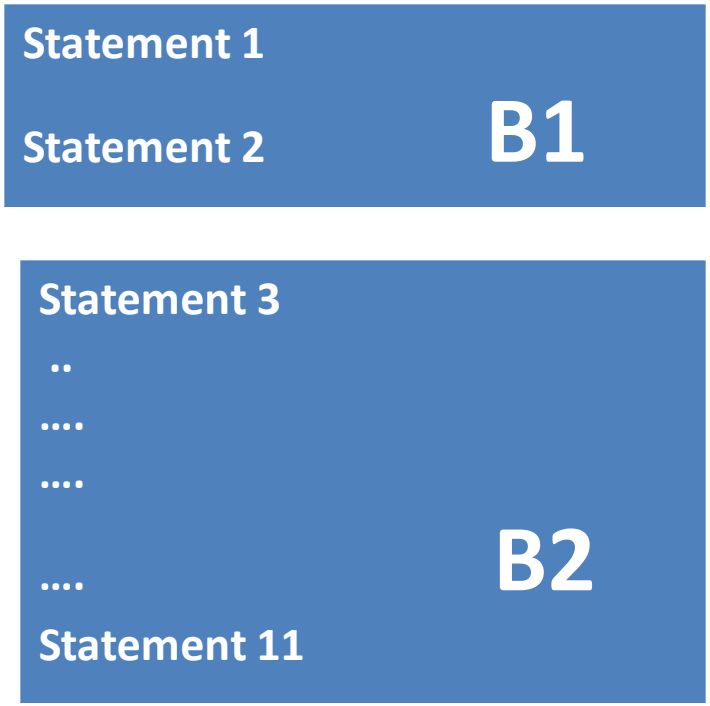
2. For each leader construct its Basic Block which consists of the leader and all the TAC statements up to but not including the next leader or the end of the program.

In the above example,

Statement 1 is a leader – By rule 1

statement 3 is a leader - By rule 2

statement 12 is a leader – By rule 3



Construction of flow graph from basic block.

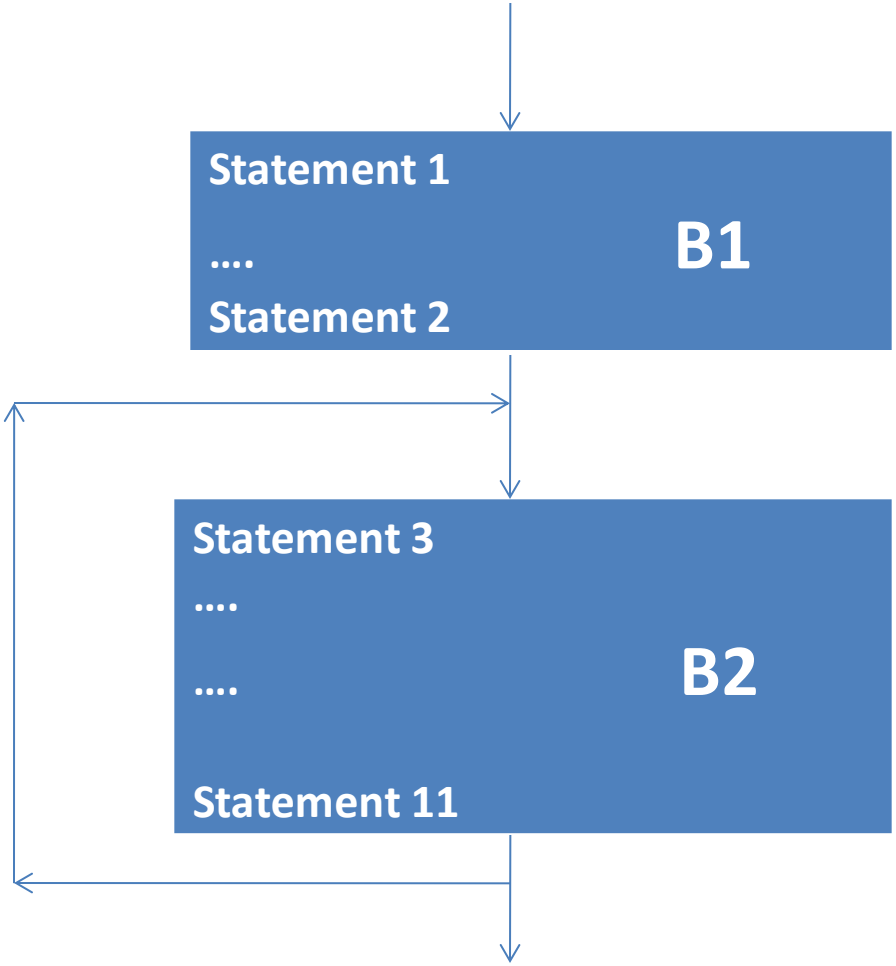
What is Flow graph ?

Flow graph is a directed graph that determines the successive relation ship between the basic blocks. Here basic blocks represents the nodes for the flow graph and one node is distinguished as the initial node where it is the basic block whose leader is the first statement. The edges are obtained as follows,

1. There is direct edge from the Block B1 to B2 if B2 could immediately follow B1 during the execution OR

2. There is jump from the last statement of B1 to the first statement of B2

Flow Graph



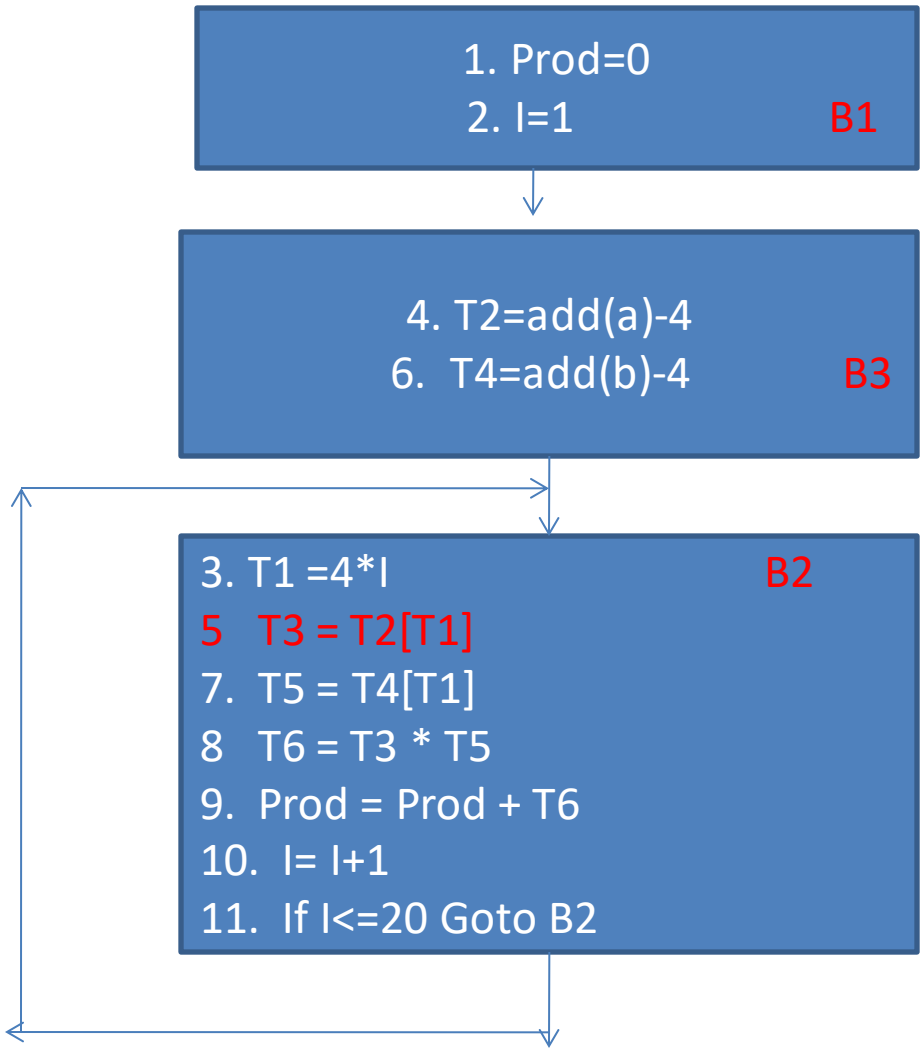
- In the above flow graph we determine the loop. This is done by checking the loop
- to be strongly connected .i.e., loop is a collection of nodes in flow graph which is strongly connected i.e., from any node in the loop to any other node there is a path of length one or more wholly within the loop.

1. Applying Code Motion

- Code motion : The running time of the program is improved by decreasing the length of one of its loop. By doing this we may increase the length program but the number of iteration may be reduced.
- Here we assume that the loop is executed at least once.
- We apply code motion technique where the important modification is done in the loop i.e., in this process we determine the loop invariant computations and these computations are placed before the loop
- A computations that yields same result, during the number of iterations of the Loop is called Loop invariant computations.

In the example, statement 4 and 6 are loop invariant computations and these are placed before the loop by adding new basic block.

Flow graph after applying Code motion



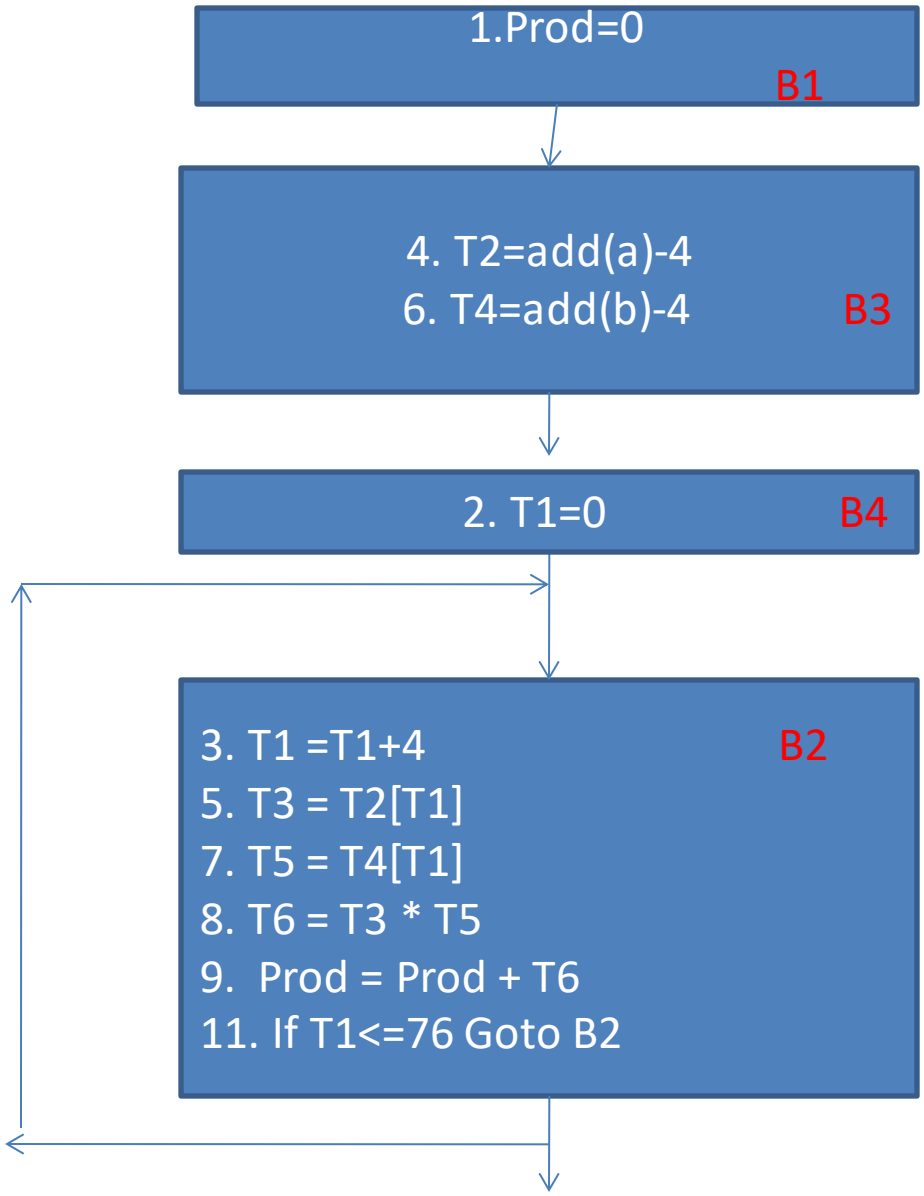
2. Removal of Induction Variable

- Another important optimization which may be applied to the flow graph which actually decreases the number of instructions and speeds up and reduces the loop iteration is Removal of Induction variable and Reduction in strength.
- What is induction variable ?

  A variables that forms the arithmetic progression is called Induction variable.

  In the example I and T1 are Induction variable since they form arithmetic progression i.e whenever i=1,2….20 T1 is T1*I =4,8……80. So the relation is determined and new statements (when i=I +1 the relationship T1=4*I-4 must hold) T1=0 and T1=T1+4 is added with new Basic Block

Flow graph after Removing Induction variable



3. Reduction in strength

- Last transformation in loop optimization is to apply reduction in strength where we can replace expensive operations by an equivalent cheaper operation of the target machine.
- We can think of the following transformations on the loop.
  - X^2 is invariably replaced by X * X where multiplications is replaced by call operations to exponentiations routine .
  - Fixed point multiplication or division by power of 2 is replaced by Shift operation.
  - Fixed point constant Multiplication may be replaced by constant addition.

Sample Example:

```
for(i=1;i<=100; i++)
{
    -----
    -----
    k= i*5
    printf(“ K is %d “,k);
}
```

```
Sum=5;
for(i=1;i<100; i++)
{
    -----
    -----
    k= sum
    printf(“ K is %d “,k);
    Sum= sum + 5;
}
```

Three Address Code statements

1. i=1 /*($base_a$-(($i*n_2$)+j)*width) n2= no of elements in each row , for i=1 and j=1 the value is 88 for width 8 bytes.
2. J=1
3. t1= 10*I
4. t2 =t1 + j
5. t3=8*t2
6. t4=t3-88
7. a[t4]=1.0
8. J=j+1
9. If j<=10 goto (3)
10. I = I +1
11. If i<=10 goto (2)
12. i=1
13. t5=i-1
14. t6=88*t5
15. a[t6]=1.0
16. i=i+1
17. If i<=10 goto (13)

Code generation Algorithm

- Here we use straight forward strategy to generate assembly code from a quadruples.
- The operands in the quadruple are currently available in the register and for each operator there is a corresponding Assembly code available.
- We also assume that the computed results are available in register as long as possible.
- Symbol table initially shows all non temporary in a Basic block as being live on exit.
- To make more informed decision about register allocation we compute the ***next use*** information and liveness of each name in a quadruples of a basic block. Where ***USE*** is defined as follows.

- Suppose quadruple ***I*** assigns a value to '**A',** if quadruple **j** has '**A**' as an operand and control can flow from quadruple **I to j** along a path that **has** no intervening assignments to **A**, Then we say that that **quadruple j *uses*** the value of **A** computed at **I**
- We wish to find for each quadruple **A= B op C** the **next uses of A, B, and C,** the algorithm must first scan the stream of quadruple to find the end of basic block . Then scan backwards to the beginning, recording for each name whether **A** has a next use in the block and if not whether it is **live on exit** from that block.
- Suppose in the backward scan we reach quadruple

  I : **A= B op C**. we then, do the following.

1. Attach to quadruple **I** the information currently found in the symbol table regarding the **'next USE'** and **'liveness' of A, B and C**
2. In the symbol table set **'A'** to **' not live'** and **no 'next use'** and set **B and C to 'live'** and **'next uses'** for quadruple **I**

Register descriptor and Address descriptor:

Register descriptor:

To perform register allocation we use **register descriptor** that keeps track of what is currently in each register . Whenever we need a register we consult **register descriptor.**

Initially **register descriptor** shows all registers are empty and as the code generation for blocks progresses each register may hold the value of names used in the program.

Address descriptor:

For each name in the block we shall maintain the **address descriptor** that keeps track of the location where the current value of the name can found at runtime. Here the location may be reg , stack or memory location

- The algorithm uses **getreg(I)** function which selects registers for each memory location associated with the **TAC I**. This function has an access to **register and address descriptor** for all the variables of the basic block.

Code Generation Algorithm and Trace

Consider a basic Block containing the following TAC statements represented as a quadruple.

Basic Block:

1. t = a – b
2. u =a – c
3. v = t + u
4. a = d
5. d = v + u

Assumptions :

1. t, u and v are temporaries and are local to the Basic block on exit the values are no longer available
2. a, b, c and d are Non Temporaries(variables) that are live on exit of the basic block

Example showing the recording of LIVENESS and NEXTUSE information for Basic Block-B

Quadruple A =B op C	Live-ness	Next Use
5. d = v + u	d is not live. v, u are live	v and u have next use at quadruple 5
4. a = d	a is not live , d is live	d has next use at quadruple 4
3. v = t + u	v is not live t and u are live	t and u have next use at quadruple 3
2. u = a – c	u is not live a and c are live	a and c have next use at quadruple 2
1. t= a – b	t is not live a and b are live	a and b have next use at quadruple 1

Basic Block-B

1. t= a – b
2. u = a – c
3. v = t + u
4. a = d
5. d = v + u

.

Code generation Algorithm

For a three address code such as **x=y OP z** do the following

1. Use **getreg(x=y+z)** to select registers for **x, y and z.** Call these **Rx, Ry and Rz**
2. If **y is not in Ry** (according to the register descriptor), Then generate an instruction **LD Ry, y'** where y' is one of the memory location for y (according to the address descriptor for y).
3. Similarly if **z is not in Rz** then generate an instruction **LD Rz,z'** where z' is memory location for z
4. Generate the instruction **OP-Code Rx, Ry, Rz**

GetReg(I)

- **GetReg(I) function** works on the following rules.

1. If **y is currently** in a **register**, pick a register already containing **y as Ry**. Do not issue the machine instruction to load this register.
2. If **y is not in a register**, but there is a register that is currently empty, **pick one such register as Ry.**
3. If **y is not in a register** and there is no register empty, we need to pick up one of the allowable registers and we need to make it safe to use.
4. The above **steps 1 to 3** can be followed for **z** to select the **Register Rz**
5. The Selection of **register Rx** which is same as above with following differences
   a. If **y** is not used after **ith instruction** then the register **Ry of y** is used to store the result i.e **Rx for x.**
   b. If **z** is not used after **ith instruction** then the register **Rz of z** is used to store the result.

Tracing the Algorithm with example

Register Descriptor			Address Descriptor						
R1	R2	R3	a	b	c	d	t	u	v
-	-	-	a	b	c	d	-	-	-

Let us consider the above status for **Register descriptor** and **Address Descriptor** before the execution of code Generation algorithm.

Assume that we have **registers – R1, R2 and R3,** initially all are **empty** and **variables a, b, c** are in memory location

Consider **Quadruple-No- 1** from basic Block

1. t = a – b

Code generation alg. calls **Getreg(t=a – b)** function to decide the **register usage for the variables t, a and b**

- Since **R1 and R2** are empty , **GetReg()** function identifies **R1 for ‘a’** and **R2 for ‘b’.** Since ‘b’ has no **next-use** information which is in memory it’s **Register R2** is used to store the result of the operation. i.e. **t in R2**
- Code-gen Algorithm checks the availability of operand values **‘a’ and ‘b’** in the registers **R1 and R2**. Since the values are not available in the registers **R1 and R2 LOAD instruction** is generated and **SUB** instruction is generated.

  LD R1, a

  LD R2, b

  SUM R2, R1, R2

Tracing the Algorithm with example

Once the code is generated for quadruple No-1, the Register and Address Descriptors are updated

Register Descriptor			Address Descriptor						
R1	**R2**	**R3**	**a**	**b**	**c**	**d**	**t**	**u**	**v**
a	t	-	a, R1	b	c	d	R2	-	-

Consider **Quadruple-No- 2** from basic Block

2. u = a – c

Code generation alg. calls **Getreg(u = a – c)** function to decide the register usage for the variables **u, a and c**

- Since **operand 'a'** is already in **register R1** – **R1 is decided for 'a' by GetReg() function.**
- **GetReg() function identifies R3 for 'b'** as **R3 is** empty from the **register descriptor**.
- For **'a'** Since **'a'** has no next-use information which is in memory it's **Register R1** is used to store the result of the operation. i.e**. u is in R1.**
- Code-gen Algorithm checks the availability of operand values **'a' and 'c'** in the registers **R1 and R3**. Since the operand **'a'** is available in R1 No LOAD instruction is generated , however the **operand 'c'** value is not available in the registers **R3, LOAD instruction** is genarated to get **'c'** into **register R3** and then **SUB instruction** is generated.

  LD R3, c

  SUB R1, R1, R3

Tracing the Algorithm with example

Once the code is generated for quadruple No-2, the Register and Address Descriptors are updated

Register Descriptor			Address Descriptor						
R1	**R2**	**R3**	**a**	**b**	**c**	**d**	**t**	**u**	**v**
u	t	c	a	b	c, R3	d	R2	R1	-

Consider **Quadruple-No- 3** from basic Block

3. v = t + u

Code generation alg. calls **Getreg(v = t + u)** function to decide the register usage for the variables **v, t and u**

- Since operand **'t' is already in register R2 – R2 is decided for 't' by GetReg() function.**
- Also operand **'u' is already in register R1 – R1 is decided for 'u' by GetReg() function.**
- For 'c' Since **'c' has no next-use information which is in memory and it's Register R3** is used to store the result of the operation. i.e. **v is in R3.**
- Code-gen Algorithm checks the availability of operand values **'t' and 'u' in the registers R2 and R1**. Since the operand **'t' is available in R2** and the operand **'u' is available in R1** , **No LOAD** instruction is generated ,.

  ADD R3, R2, R1

Tracing the Algorithm with example

Once the code is generated for quadruple No-3, the Register and Address Descriptors are updated

Register Descriptor			Address Descriptor						
R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

Consider Quadruple-No- 4 from basic Block

4. a = d

Code generation alg. calls Getreg(a = d) function to decide the register usage for the variables a and d

- It is a copy instruction,

- For ‘t’ Since ‘t’ has no next-use information which is temporary and it’s Register R2 is used to store the result of the operation. i.e. Both a and d is in R2.
- Code-gen Algorithm checks the availability of operand values ‘d’ . Since the operand ‘d’ is not available in R2, so LOAD instruction is generated to “d’ into R2.

  LD, R2, d

Tracing the Algorithm with example

Once the code is generated for quadruple No-4, the Register and Address Descriptors are updated

Register Descriptor			Address Descriptor						
R1	R2	R3	a	b	c	d	t	u	v
u	a, d	v	a, R2	b	c	d, R2		R1	R3

Consider Quadruple-No- 5 from basic Block

5. d = v + u

Code generation alg. calls Getreg(d = v + u) function to decide the register usage for the variables d,v and u

- Since operand ‘v’ is already in register R3 – R3 is decided for ‘v’ by GetReg() function.
- Also operand ‘u’ is already in register R1 – R1 is decided for ‘u’ by GetReg() function.
- For ‘u’ Since ‘u’ has no next-use information which is temprory and it’s Register R1 is used to store the result of the operation. i.e. d is in R1.
- Code-gen Algorithm checks the availability of operand values ‘v’ and ‘u’ in the registers R3 and R1. Since the operand ‘u’ is available in R3 and the operand ‘u’ is available in R1 , No LOAD instruction is generated ,.

  ADD R1, R3, R1

Tracing the Algorithm with example

Once the code is generated for quadruple No-5, the Register and Address Descriptors are updated

Register Descriptor			Address Descriptor						
R1	**R2**	**R3**	**a**	**b**	**c**	**d**	**t**	**u**	**v**
d	a	v	a, R2	b	c	d, R1			R3

Once the code generated the Code generation alg. Refers the Register descriptor and Checks for the availability of values for Non-temporaies, since they are live on exit and if the values are stored in register then STORE instruction is generated to get the values into their respective memory. In the register descriptor R1 and R2 Holds the value of Two non Temporaries d and a. hence Store instruction is generated.

ST a, R2

ST d, R1

Summary of Trace of Code Gen Alg showing Instructions generated and Updating values in register and address descriptor