12th April '17
Lecture-48.

Unit-5.

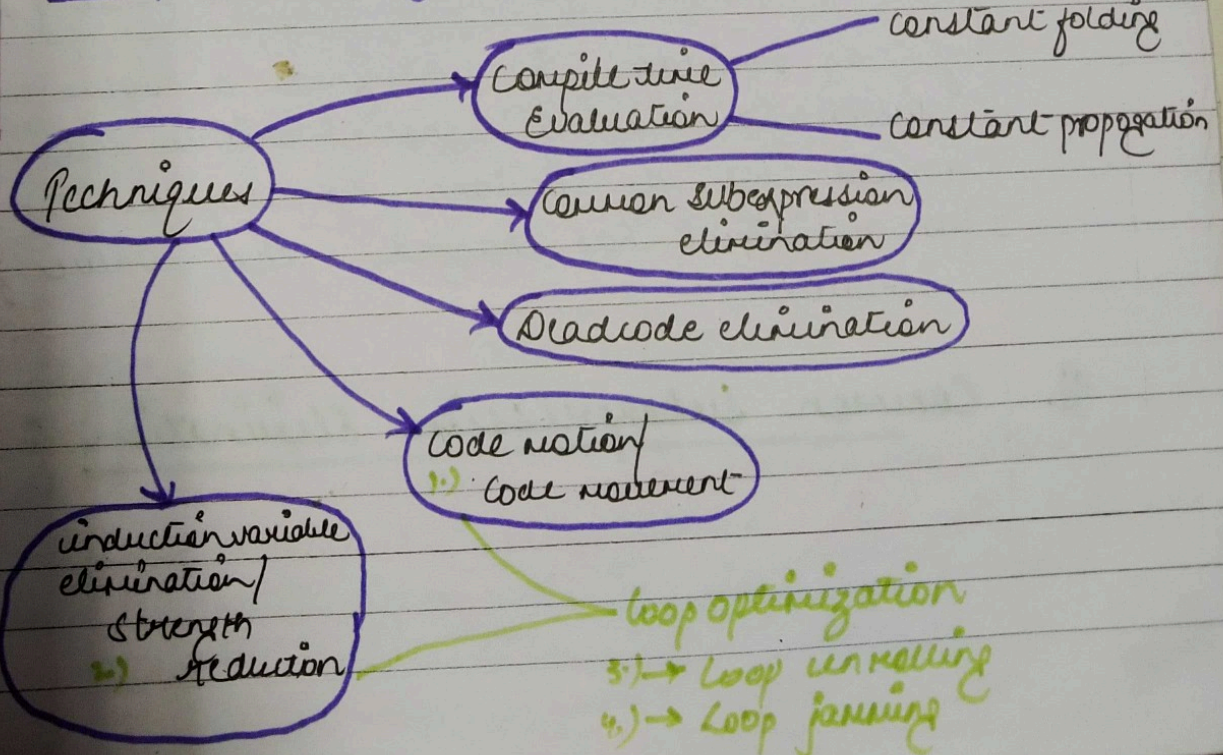
Code Optimization

Code optimization is a technique which tries to improve the code by eliminating the unnecessary code lines & arranging the statements in such a sequence that speed up the program execution without wasting the resources.

* Advantages :

1. Executes better
2. Efficient memory usage
3. ~~Pattern~~ performance.
   better

* Principle Techniques used in code optimization:



Date: __/__/__
Page:

I. Compile time Evaluation :

1.) constant folding :

It refers to the technique of evaluating the expression whose operands are known to be constant at compile time itself.

Example :

length = (22/7) * d (original)

optimised

= (3.14) * d (folded value at compile time)

2.) constant propogation :

Example :

Area of circle (A) = π * r * r ; r=5 10 times evaluated

A = 3.14 * 5 * 5

A = x ← Calculated at compile time as 10 times A is calculated for same value of r (i.e. 5)

If a variable is assigned a value then subsequent use of that variable can be replaced by a constant as long as no in b/w assignment has changed the value of variable.

II. Common Subexpression Elimination :

The common subexpression is an expression i.e. repeatedly appearing in the code which is computed previously.

Ex:

Before elimination

$T_1 = 4 * i$
$T_2 = addr(a) - 4$
$T_3 = T_2[T_1]$
$T_4 = 4 * i$
$T_5 = addr(b) - 4$
$T_6 = T_5[T_4]$

common subexpression

After elimination

$T_1 = 4 * i$
$T_2 = addr(a) - 4$
$T_3 = T_2[T_1]$
$T_5 = addr(b) - 4$
$T_6 = T_5[T_1]$

III Dead code Elimination :

Dead code elimination eliminates those code statements which are never executed or are unreachable or if executed, their o/p is never used.

Ex:

```
i = 0;
if (i = 1)
{
    x = a + 5;
}
```

Dead code: this will never execute as i ≠ 0 ever

```
i = 0;
```

IV Code motion / code movement :

It is a technique of moving a block of code outside the loop if it does not have any difference if it is executed outside or inside the loop.

Ex:

```
for (i=0; i<=10; i++)
{
    x = y + z;
    a[i] = 6 * i;
}
```

This does not depends on "for" loop. It will also execute 10 times if will remain inside the loop. So, optimize it.

```
x = y + z;
for (i=0; i<=10; i++)
{
    a[i] = 6 * i;
}
```

x = y * z;

```
while(i <= (max-1))
{
    sum = sum + a[i] + b[i];
}
```

compiler will every time compute this value then will move inside the loop. so, compute this value & keep it in this place.

```
t = max-1;
while(i<=t)
{
    sum = sum + a[i] + b[i];
}
```

Induction variable elimination / Strength Reduction:

A variable x is called induction variable of a loop if its value is incremented or decremented by some constant each time.

Ex:

```
i=1
T1 = 4 (*) i          =>   T1 = T1 (+) 4
|                          |
if i <= 20 goto       =>   if T1 <= T6 goto
```

"*" strength is higher than "+"
So, "*" replaced by low strength operator.

VII Loop Jamming / Loop Fusion :

Strength of certain operator is higher than others then higher strength operator are replaced by lower strength operator.

14th April 17
Lecture-49.

VI Loop Unrolling Technique :

The idea behind this technique is to save time by reducing number of overhead instructions that the computer has to execute in a loop. To achieve this statements that are called in multiple iterations are combined in a single iteration.

Example :

```
i=1
while (i<=100)
{
a[i]=0; a[i]=0
i++;
}
```

execution loop will be 100 times

at once only 1 value of "i" will be executed

i=1, i=2 & so on

Solution :

```
i=1
while (i<=100)
{
a[i]=0; a[i]=0;
i=2 i++;
a[i]=0 a[i]=0;
i=3 i++;
}
```

loop will now be executed 50 times

at once 2 values of "i" will be executed

VII Loop Jamming / Loop fusion :

It is a technique in which we can merge the bodies of two loops if both the loops have same number of iterations.

Example :

```
for (i=1; i<=100; i++)
{
    a[i] = 0;
}
for (i=1; i<=100; i++)
{
    b[i] = a[i] + 5;
}
```

Same loop.

: Equivalent following code :

```
for (i=1; i<=100; i++)
{
    a[i] = 0;
    b[i] = x[i] + 5;
}
```

Basic Blocks :

The basic block is a sequence of consecutive statements which are always executed in sequence without halt or possibility of branching.

* Algorithms for partitioning the 3-address statements into basic blocks :

Step-1 : Determine the leader.

a.) 1st statement is a leader

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

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

Step-2 : The basic block is formed starting at the leader statement & ending just before the next leader statement appearing.

Example :

```
prod = 0;
i = 1;
do
{
  prod = prod + a[i] * b[i];
  i = i + 1;
}
while (i <= 20);
```

20

4 bytes / word.

B1
1.) prod = 0; // 1(a)
2.) i = 1

B2
3.) $t_1 = i * 4$ // 2 1(b)
4.) $t_2 = addr(a) - 4$
5.) $t_3 = t_2[t_1]$
6.) $t_4 = i * 4$
7.) $t_5 = addr(b) - 4$
8.) $t_6 = t_5[t_4]$
9.) $t_7 = t_3 * t_6$
10.) $t_8 = t_7 + prod$
11.) $prod = t_8$
12.) $t_9 = i + 1$
13.) $i = t_9$
14.) if $i <= 20$ go to (3)

B3
15.) Next Statement // 3 1(c)

No. of statements executing =

$2 + (20 \times 12) \Rightarrow 242$ statements

B_1 B_2

15th April'17
Lecture-50

* Flow Graph :

It is a directed graph in which the flow control information is added to basic blocks.

Rules :

Rule-1 : Basic blocks are the nodes of the flow graph.

Rule-2 : The block whose leader is the 1st statement is called Initial Block.

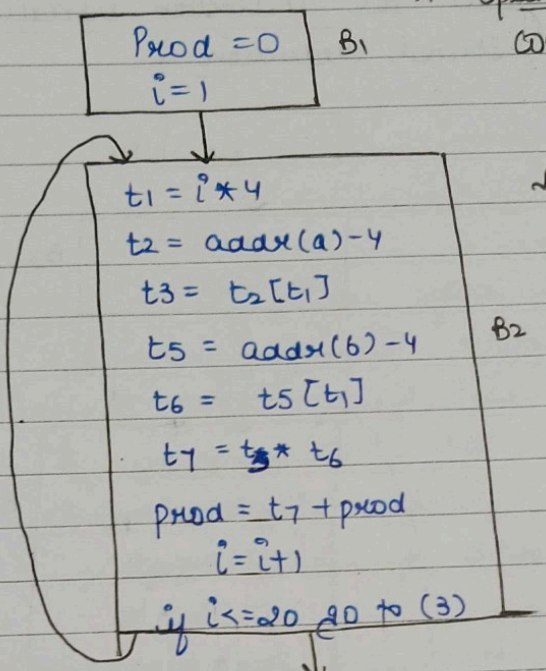
Rule-3 : There is a directed edge from block B_1 to block B_2. If B_2 immediately follows B_1 in the given sequence so we can say B_1 is the predecessor of B_2.

* Optimization of Basic blocks :

For the previous block digram, optimization is applied to that block and common subexpressions are optimized.

Optimised block is :

* Optimization for common subexpression



to block beginning with statement four.

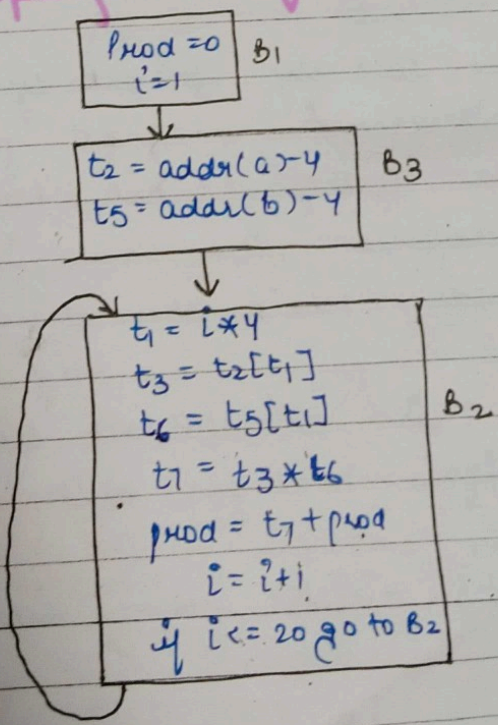
No. of statements executing =

2 + (9 × 20)

↓ B1 ↓ B2

= 182

* Optimization for code motion applied:



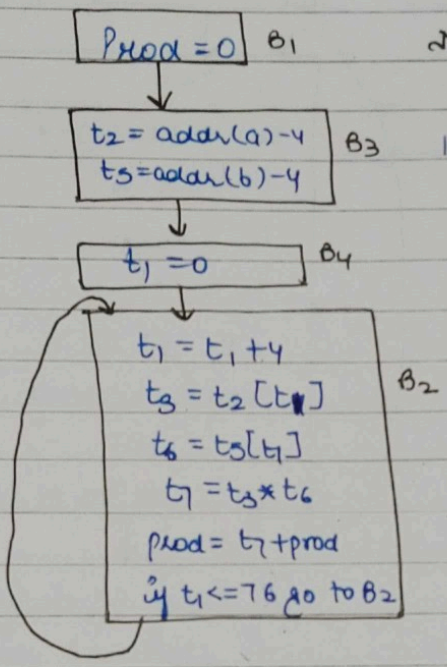
No. of statements executing =

2 + 2 + (7 × 20)

↓ B1 ↓ B3 ↓ B2

= 144

* Optimization for induction variable elimination or strength reduction :



No. of statements executed =
1 + 2 + 1 + (6 x 20)
= 124

Directed Acyclic Graph (DAG) :

* Similar to any abstract syntax tree but the only difference is that for common subexp. the tree is not made again & value is not recomputed. The value is directly taken from the node where the value is stored.

→ A DAG is an abstract syntax tree with a unique node for each value.

→ DAG is a directed graph that contains no cycle.

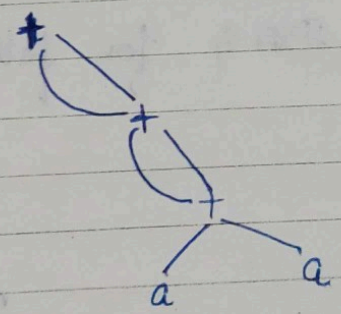
* Rules for construction of DAG:

1. In a DAG, leaf nodes represents identifiers, names or constants and interior nodes represents operations.

2. While constructing DAG, we have to check to find if there is an existing node with the same children. A new node is created only when such a node does not exist. This action allows us to detect common subexpressions & eliminate recomputation of the same.

3. The assignment of the form $x := y$ must not be performed until it is unless it is a must.

Example : $((a+a)+(a+a))+((a+a)+(a+a))$

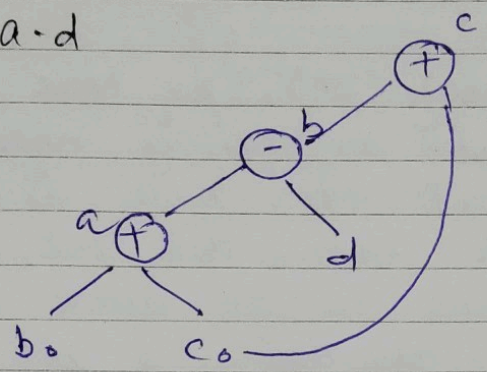


Date
17/4/17

Lecture 5

Ques Construct DAG for the following :-

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

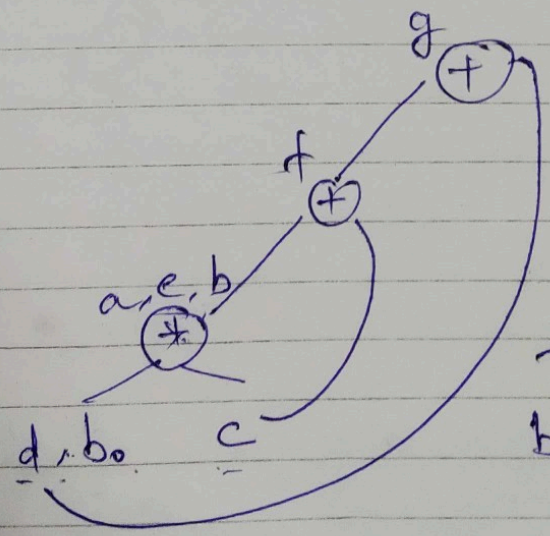


Now by optimising

$a = b + c$
$d = a - d$
$c = d + c$

Ques Construct DAG for following block.

$a = b * c$
$d = b$
$e = d * c$
$b = e$
$f = b + c$
$g = f + d$



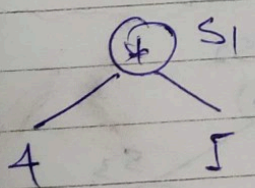
Take initial b as b_0.

```
a = b * c          a = b * c
d = b      ⟹      t = a + c
t = a + c          g = t + b //
g = t + d
```

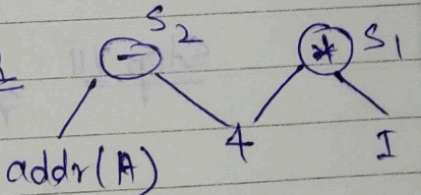
Ques Consider the following basic block. Draw DAG representation of the block and identify local common subexpression. Eliminate the common expression and rewrite the code.

```
L10 :  S1 = 4 * I
       S2 = addr(A) - 4
       S3 = S2[S1]
       S4 = 4 * I
       S5 = addr(B) - 4
       S6 = S5[S4]
       S7 = S3 * S6
       S8 = PROD + S7
       PROD = S8
       S9 = I + 1
       I = S9
       if I <= 20 goto L10
```

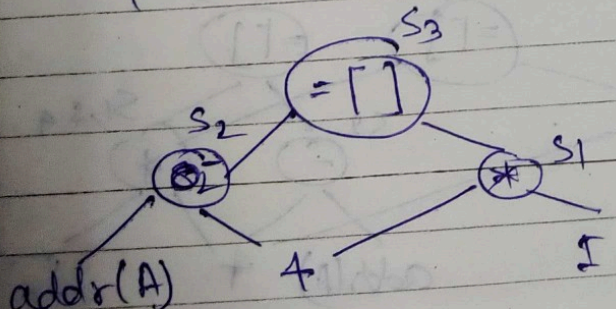
Step 1



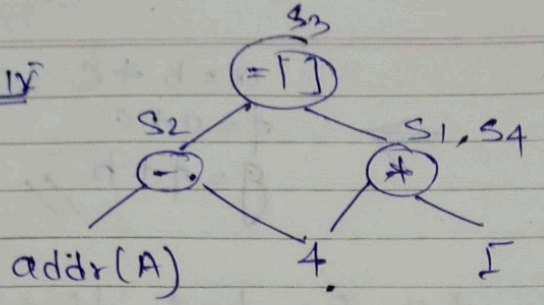
Step 2



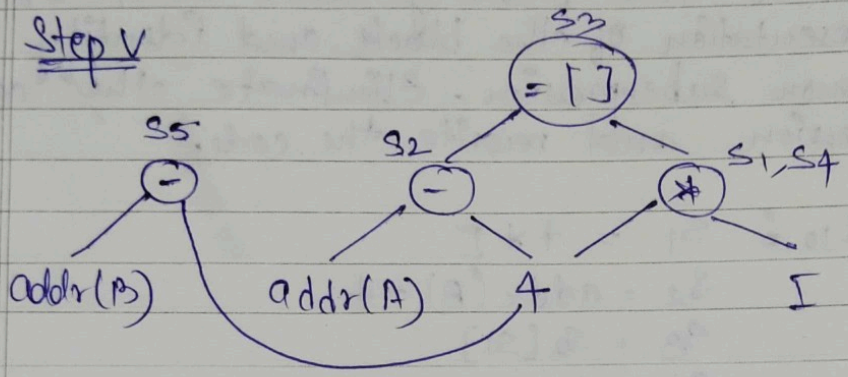
Step 3



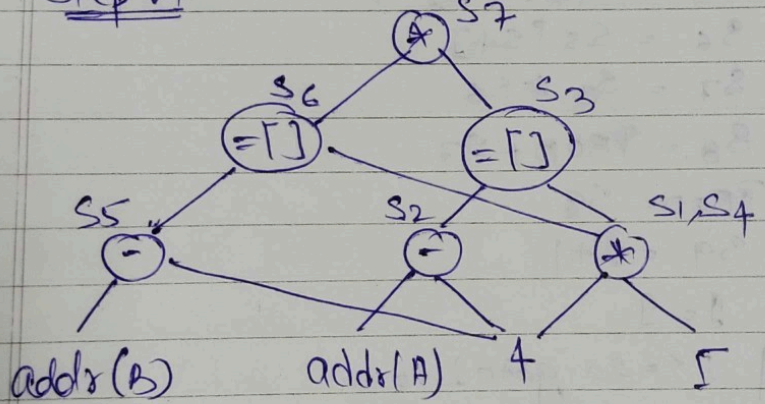
Step IV



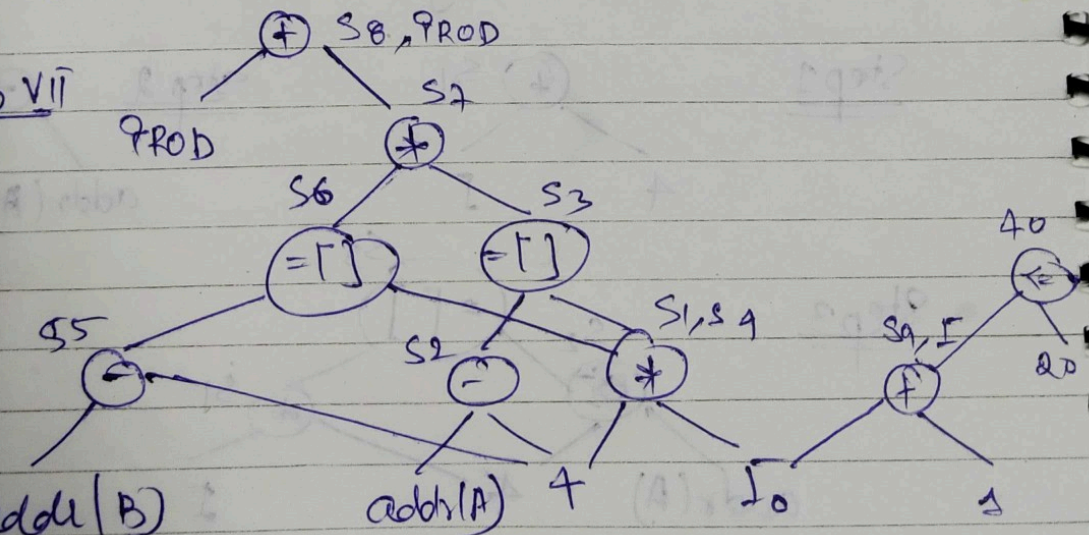
Step V



Step VI



Step VII

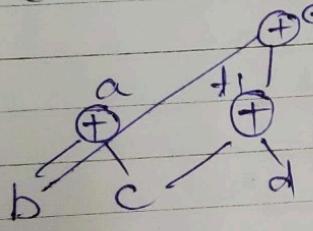


Now by optimising

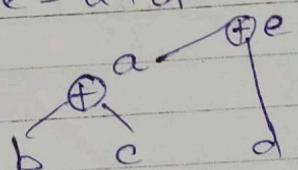
```
L10 : S1 = 4 * I
      S2 = addr (A) - 4
      S3 = S2 [S1]
      S5 = addr (B) - 4
      S6 = S5 [S1]
      S7 = S3 * S6
      PROD = PROD + S7
      I = I + 1
      if I <= 20 goto L10
```

⇒ Algebric Identities.

a = b + c
e = c + d + b



Now by optimising
a = b + c
e = a + d ∵ a = b + c



18th April'17
Lecture-52

1. # Global Data Flow Analysis :

A number of optimisations can be achieved by knowing various pieces of information that

can be obtained only by examining the entire program.

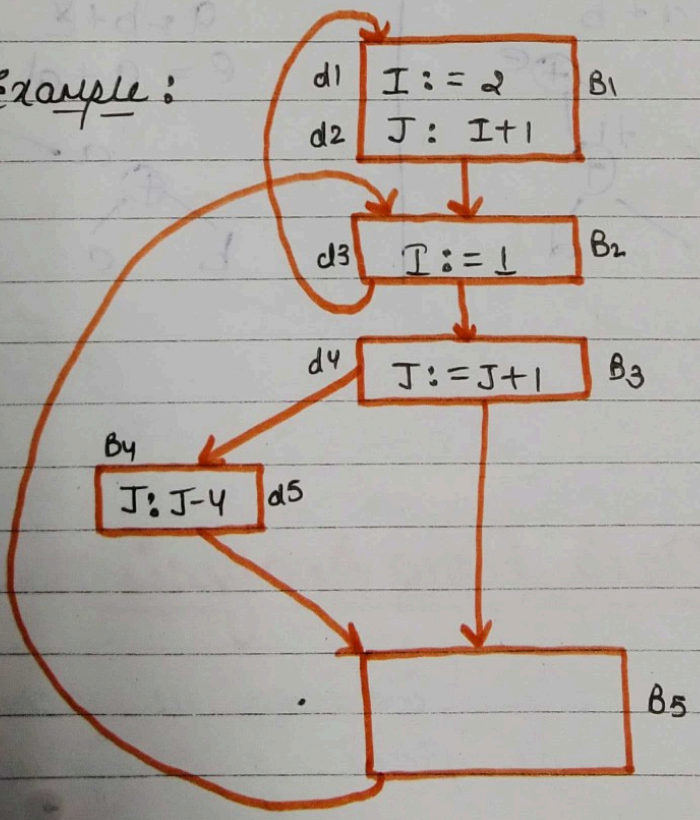
Example: If a variable A has value 3, everytime control reaches a certain point p then we can substitute 3 for each use of A at p.

Reaching Definations :

→ To determine the definations that can reach a given point in a program, we first assign a distinct number to each defination.

Example :

(to find UD chains)



→ The next step in UD-chaining is to compute 2 sets for each basic blocks B. The first GEN[B] is the set of GENERATED definations, those definations within block B that reach the end of the block.

bit vector

GEN[B1] = {d1, d2} 11000
GEN[B2] = {d3} 00100
GEN[B3] = {d4} 00010
GEN[B4] = {d5} 00001
GEN[B5] = {ϕ} 00000

→ The second set KILL[B] which is the set of definations outside of B that define identifiers that also have defination within B.

KILL[B1] = {d3, d4, d5} 00111
KILL[B2] = {d1} 10000
KILL[B3] = {d2, d5} 01001
KILL[B4] = {d2} 01000
KILL[B5] = ϕ 00000

→ The next and most complex step is to compute for all blocks B in the set B i.e. is consisting of all the definations reaching the point just before the 1st statement of block B.

$IN[B_1] = OUT[B_2]$

$IN[B_2] = OUT[B_1] \cup OUT[B_5]$

→ To help compute IN we will also compute OUT[B] for all blocks B.

OUT[B] is the set of definitions reaching the point just after the last statement of B.

Data Flow Equations :

2 sets of equations called data flow equations that relate IN & OUT.

For all blocks B :

1.) $\boxed{OUT[B] = ((IN[B] - KILL[B]) \cup GEN[B])}$

$IN[B] - KILL[B] \Rightarrow IN[B] \wedge \neg KILL[B]$

2.) $\boxed{IN[B] = \cup\ OUT[P]}$

P - predecessor of B

Initially $IN[B] = \phi$ and $OUT[B] = GEN[B]$.

	INITIAL		Pass 1	
	IN[B]	OUT[B]	IN[B]	OUT[B]
B1	00000	11000	00100	11000
B2	00000	00100	11000	01100
B3	00000	00010	00100 ? 01100	00110
B4	00000	00001	00110	00101
B5	00000	00000	00111	00111

21st April '17
Lecture-53

Pass 1 :

Block B1 :

$NEW[IN] = OUT[B_2]$

$NEW[IN] = 00100 \neq 00000 = IN[B]$

$OUT[B_1] = IN[B_1] - KILL[B_1] \cup GEN[B_1]$

$= 00100 - 00111 \cup 11000$

$= (00100 \wedge 11000) \vee 11000$

$= 00000 \vee 11000$

$OUT[B_1] = 11000$

Block B2 :

$NEW[IN] = OUT[B_1] + OUT[B_5]$

$= 11000 \vee 00000$

$NEW[IN] = 11000$

$OUT[B_2] = IN[B_2] - KILL[B_2] \cup GEN[B_1]$

$= 11000 - 10000 + 00100$

$= 11000 \wedge 01111 \vee 00100$

$= 01000 \vee 00100$

$OUT[B_2] = 01100$

Block B_3 :

$NEW[IN] = OUT[B_2]$

$NEW[IN] = 01100$

$OUT[B_3] = IN[B_3] - KILL[B_3] \cup GEN[B_3]$

$= 01100 - 01001 \cup 00010$

$= (01100 \wedge 10110) \vee 00010$

$= 00100 \vee 00010$

$OUT[B_3] = 00110$

Block B_4 :

$NEW[IN] = OUT[B_3]$

$NEW[IN] = 00110$

$OUT[B_4] = IN[B_4] - KILL[B_4] \cup GEN[B_4]$

$= 00110 - 01010 \cup 00001$

$= (00110 \wedge 10101) \vee 00001$

$= 00100 \vee 00001$

$OUT[B_4] = 00101$

Block B_5 :

$NEW[IN] = OUT[B_3] + OUT[B_4]$

$= 00110 + 00101$

$= 00110 \vee 00101$

$NEW[IN] = 00111$

$OUT[B_5] = IN[B_5] - KILL[B_5] \cup GEN[B_5]$

$= 00111 - 00000 \cup 00000$

$= (00111 \wedge 11111) \vee 00000$

$= 00111 \vee 00000$

$OUT[B_5] = 00111$

B1
B2
B3
B4
B5

	Pass 2		Pass 3		Pass 4	
	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B_1	01100	11000	01111	11000	01111	
B_2	11111	01111	11111	01111		
B_3	01111	00110	01111	00110		
B_4	00110	00101	00110	00101		
B_5	00111	00111	00111	00111		

Pass 2 :

Block B_1 :

$$NEW[IN] = OUT[B_2]$$

$$\boxed{NEW[IN] = 01100}$$

$$OUT[B_1] = IN[B_1] - KILL[B_1] \cup GEN[B_1]$$

$$= (01100 - 00111) \cup 11000$$

$$= (01100 \wedge 11000) \vee 11000$$

$$= 01000 \vee 11000$$

$$\boxed{OUT[B_1] = 11000}$$

Block B_2 :

$$NEW[IN] = OUT[B_1] + OUT[B_5]$$

$$= 11000 + 00111$$

$$= 11000 \vee 00111$$

$$\boxed{NEW[IN] = 11111}$$

$$OUT[B_2] = IN[B_2] - KILL[B_2] \cup GEN[B_2]$$

$$= (11111 \wedge 10000) \vee 00100$$

$$= (11111 \wedge 01111) \vee 00100$$

$$= 01111 \vee 00100$$

$$\boxed{OUT[B_2] = 01111}$$

Similarly for Pass 2, 3 & 4

22nd April'17
Lecture-54

Computing UD-chains :

→ If a use of a variable "A" is preceded (predecessor) in its block by a definition of A then only the last definition of A in the block prior to this use reaches the use. Hence, the list or UD chain for this use consists of only this one definition.

- The UD chain for I in d2 consists only of d1.

→ If a use of A is precedent in its block B by no definition of A then, the UD chain for this use consists of all definitions of A in IN[B].

- The UD chain of J in d4 is IN[B3] = 01111 i.e; d2, d3, d4, d5 but in d3 J is not defined so, UD chain of J in d4 is d2, d4, d5.

- UD chain of J in d5 is only d4.

22nd April '17
Lecture-54

Code Generation :

The final phase of compiler model is code generator. It takes input from the intermediate representation with supplementary information in symbol table of source program and produces as o/p an equivalent target program.

* Design Issues in CG :

1.) I/p to CG
2.) Target Program : o/p may take variety of forms :
   a.) Absolute M/c language (Executable code)
   b.) Relocatable M/c code
   c.) Assembly code/language
3.) Memory management :
   If the M/c code is being generated level in 3 address statement have to be converted to addresses of instructions. This process is similar to backpatching technique.
4.) Instruction selection :
   It is important to obtain efficient code.

Example :

$x = y + z$

MOV y, R0
MOV z, R1
ADD R0, R1
STX R1, x

MOV y, R0
ADD z, R0
STX R0, x

$a = a + 1$

MOV a, R0
ADD #1, R0
STX R0, a

Add #1, a

INC a

efficient but time consuming

5.) Register allocation :

6.) Evaluation Order

Peephole Optimization : performed on target program

2 marks

1.) Optimization can be directly assigned on assembly language.

2.) A simple but effective technique for locally improving the target code.

3.) It is done by examining a sliding window of target instruction (peephole) and replacing instruction sequences within a peephole by a shorter & faster sequence.

Example : MOV a, b
MOV b, a
⇒ MOV a, b

DAG
Code optimization techniques
Global Data flow analysis

Unit-V

4.) Repeated passes over the target code is are necessary to get maximum benifit.

I. Redundant Instruction Elimination :

OR

Redundant Load & Store Instruction :

```
LD R0, a        ⇒    LD R0, a
ST a, R0
```

II. Eliminating unreachable code :

```
if debug == 1 go to L1          if debug != 1 go to L2
go to L2                   ⇒    print xy
L1: print xy                    L2: ---
L2: ------
```