14/9/18

Code Optimization

Code optimization is done after code generation phase or before code generation

⇒ It is optimization to do to apply the optimization it is important to do control flow analysis. Data flow analysis & transformation

controlflow Analysis It determines control structure of a pgm & builds control flow graph.

Dataflow analysis It determines flow of scalar values and built dataflow graph.

Transformation: Transformations help in improving the code without changing the code / functionality

Flowgraph: Graphical representation of 3-address code is flowgraph.

Nodes represents in the flow graph are blocks.
Edges represent flow of control

Basic block: Is set of consecutive statements that are executed sequentially.

Procedure to identify basic blocks

0 For given 3-address code identify the leader stmts and group the leader statement with the statements upto the next leader.

To identify the leader stmt use the following rules

① First statement in the given 3-address code is leader stmt.

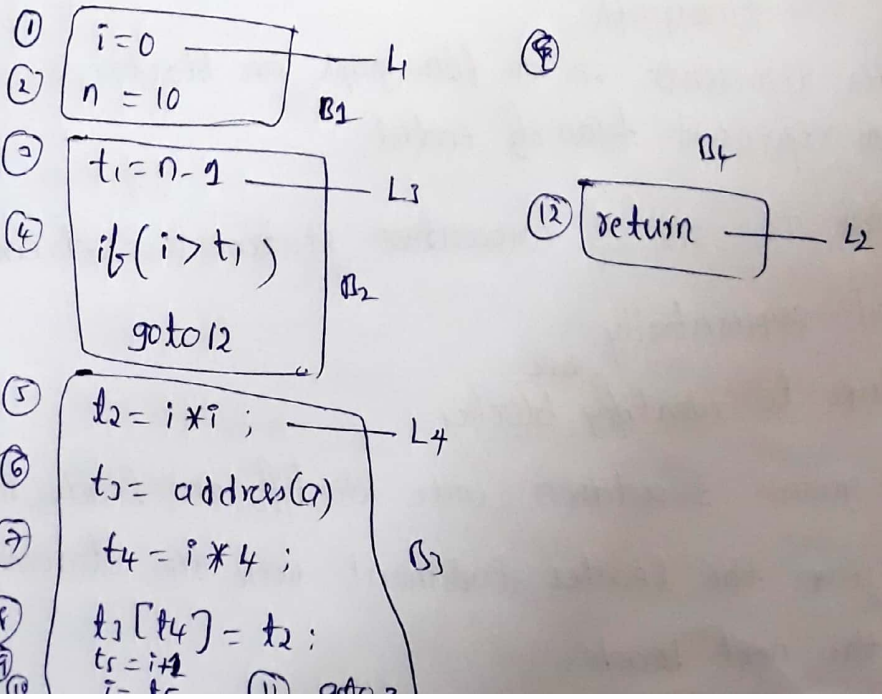
② Any statement that is target of conditional or unconditional stmt is a leader stmt.

③ Any statement that immediately follows conditional or unconditional stmt is a leader stmt.

Following flowgraphs are used in 1st 2 phases

```
main()
{
   int i=0 ; n=10;
   int a[n];
   while (i<=n-1)
   {
       a[i] = i*i;
       i=i+1;
   }
   return;
}
```

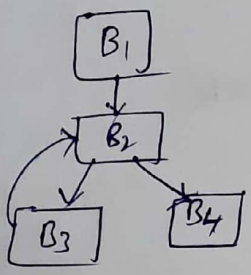
3-address code



An edge is placed from B_1 and B_2 if Block B_2 immediately follows block B_1 during execution or satisfies the following cond^{n}s.

① Last statement in B_1 is either conditional / unconditional jump stmt that is followed by first stmt in B_2. (or)

② First stmt in B_2 follows last stmt in B_1 and is not an unconditional or conditional jump stmt.



Constant Propagation

① If the value of variable is constant then replace the variable by constant.

② Constant Folding

$x = y$ op z

$x =$ op z

if y & z are constants then value of x can be computed at compilation time.

```
# define m 10
  x = 2 * m
x [ if (x == 0)
      goto L1 ]      Noneed of this
```

Strength reduction

* is replaced by '+' or '<<'

Common sub expression elimination

```
a = b+c          a = b+c
c = b+c    ⇒     c = a
d = b+c          d = b+c
```

Dead code elimination

Not at all using code is dead code

```
m = 20
x = 2 * m
if (x == 0)
    goto L1        } Dead code
```

copy propagation

```
y = x
z = y    ⇒    z = x
```

loop optimization

① code motion / loop invariant

```
t1 = z/y
for(i=0; i<1000; i++)
    x = i + t1;
```

not at all changing inside value of loop we can put outside

② loop unrolling

```
for(i=0; i<100; i+=2)
{
    g();
    g();
}
```

All above are machine independent optimization

③ loop fusion

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

} we can combine both.

Machine dependent optimization

Redundant loads & stores

```
a = b          c = a + b          e = a + b
MOV R0, b      MOV R0, a          MOV R0, a
MOV R1, a      add R0, b          MOV R1, b
MOV R1, R0     MOV c, R0          add R0, R1
```

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

MOV R0, b
MOV R1, c
add R0, R1
MOV a, R0
MOV add R0, e
    MOV d, R0
```

Algebraic simplifications

```
x = x + 0
x = x * 1
```

① Dead code elimination

```
if (x == 0)
    goto L1
L1: goto L2
L2: goto L3
```

— we can replace by goto L_3

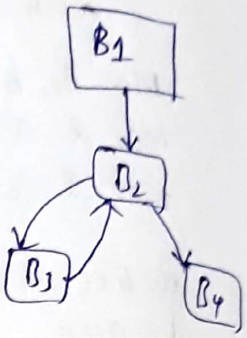
② Machine Idioms:

```
x = x + 1
Inc x
```

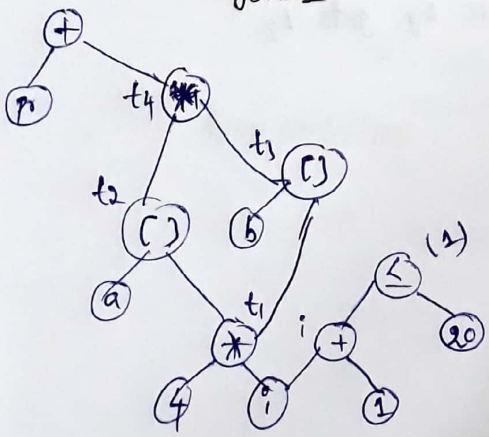
Peephole optimization

5 marks

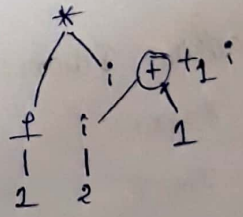
1) f = 1 ; — L_1
2) i = 2 ; — B_1
3) if (i > n) goto 8 — L_3, B_2
4) f = f * i ; — L_4
5) t_1 = i + 1 ; } we can reduce
6) i = t_1 ; — B_3
7) goto 3 ;
8) goto calling program — L_2, B_4



1) $t_1 = 4 * i$
2. $t_2 = a[t_1]$
3. $t_3 = b[t_1]$
4. $t_4 = t_2 * t_3$
5. $p_1 = p_1 + t_4$
6. i = i + 1
7. if (i <= 20) goto 1



DAG for I



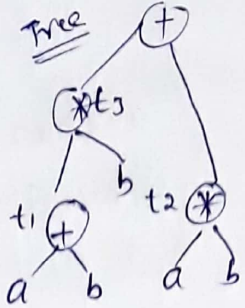
$t_1 = a+b$

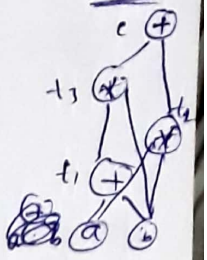
$t_2 = a*b$

$t_3 = t_1 * b$

$c = t_3 + t_2$

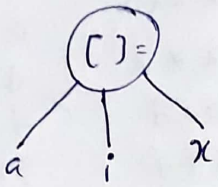
Tree



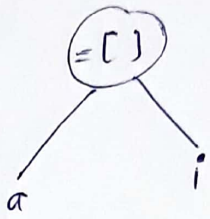


Q6

$a[i] = x$



$x = a[i]$



$a * 2^b \Rightarrow a << b$

$\frac{a}{2^b} \Rightarrow a >> b$

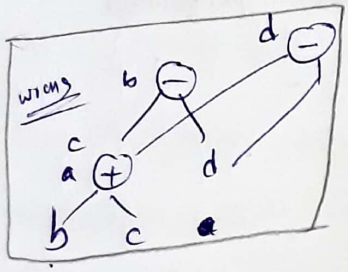
③

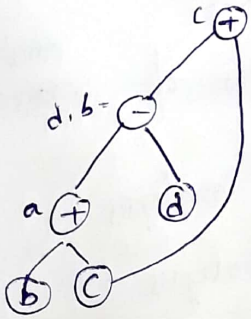
$a = b+c$

$b = a-d$

$c = b+c$

$d = a-d$





④

	Algebraic		$w = x$ copy & constant		Common
$a = b+c$	$a = b+c$		$a = b+c$	$a = b+c$	$a = b+c$
$z = a**2$	$z = a*a$		$z = a*a$	$z = a*a$	$z = a*a$
$x = 0*b$	$x = 0$		$x = 0$	$x = 0$	$x = 0$
$y = b+c$	$y = b+c$		$y = b+c$	$y = b+c$	$y = a$
$w = y*y$	$w = y*y$		$w = y*y$	$w = y*y$	$w = y*y$
$u = x+3$	$u = x+3$		$u = 0+3$	$u = 3$	$u = 3$
$v = u+w$	$v = u+w$		$v = u+w$	$v = u+w$	$v = u+w$

Dead code elimination

⇒

$a = b+c$	$a = b+c$	$a = b+c$		$a = b+c$
$z = a*a$	$z = a*a$	$z = a*a$		$z = a*a$
$x = 0$	$x = 0$	$x = 0$	⇒	$v = 3+z$
$y = a$	$y = a$	$y = a$		
$w = a*a$	$w = z$	$w = z$		
$u = 3$	$u = 3$	$u = 3$		
$v = u+w$	$v = 3+w$	$v = 3+z$		

⑤ $a = x**2$

$b = 3$

$c = x$

$d = c * c$

$e = b * 2$

$f = a + d$

$g = e + f$

f & g are used in the rest of the pgm.

$a = x**2$

$f = a + d$

$g = e + f$

$c = x)$

$d = x**x$

$e = 3 * 2$

$a = x**2$

$d = x**x$

$f = a + d$

$e = 6$

$g = 6 + f$

⇓

$a = x**2$

$d = x * x$

$f = a + d$

$g = 6 + f$

->

$a = x**x$

$d = a$

$f = a + d$

$g = 6 + f$

⇓

$a = x * x$

$f = a + a$

$g = 6 + f$

10/10/18 **Global optimization**

There are 2 types of analysis operations performed

① Controlflow analysis

② Dataflow analysis

Dataflow analysis: Dataflow analysis is a process of computing values of dataflow properties

Dataflow properties

① Available expressions

② Reaching definitions

③ Live variables Analysis

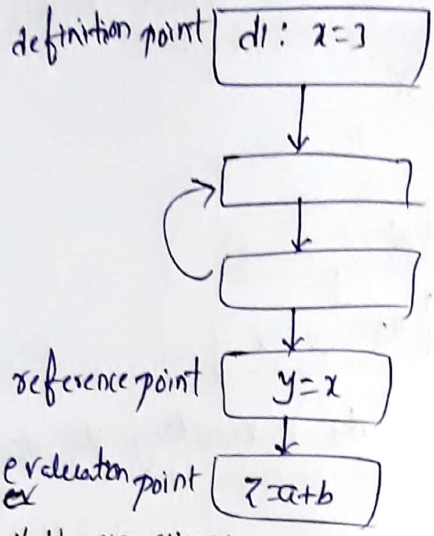
④ Busy variables Analysis

Basic terminology

A program point containing the definition is called definition point.

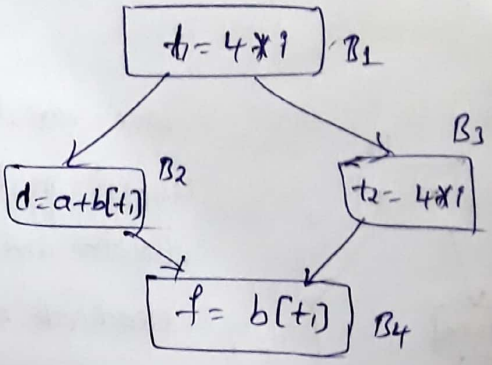
→ A program point at which a reference to a data item is made is called reference point.

→ A program point at which some evaluating expression is given is called evaluation point



Available expressions:-

An expression x+y is available at a program point 'w' if and only if along all paths are reaching to 'w'. The expression x+y is said to be available at its evaluation point. If neither of two operands get modified before their use.



Expression is available at: B_1, B_2, B_3, B_4.

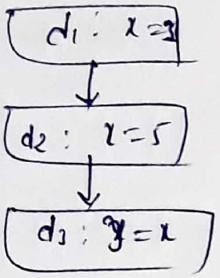
=> used to eliminate common sub expressions. elimination

11/10/18

Reaching Definitions

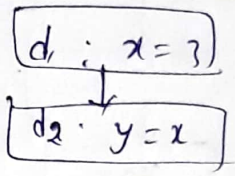
A Definition 'D' reaches at point 'P' if there is a path from D→P along which 'D' is not killed

→ A definition D of variable 'x' is killed when there is a redefinition of 'x'.



d_1 is reaching to d_2

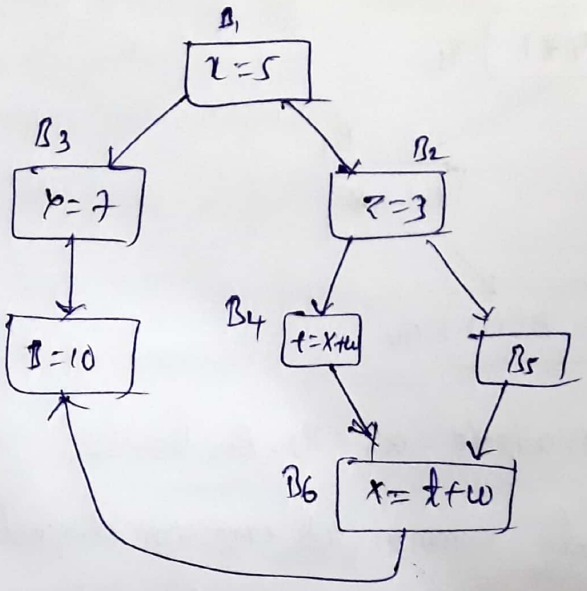
d_1 is not reaching to d_3



d_1 is reaching to d_2

Live variables

a variable 'x' is live at some point 'p' if there is a path from p to exit along which the value of 'x' is used before it is redefined. otherwise the variable is said to be dead at that point.

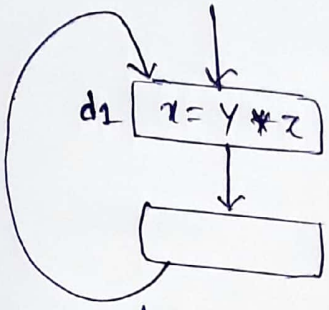


Live variables are useful in Register allocation and also for dead code elimination.

→ X is live at Block B_1, B_2, B_3, B_4 but not available at B_6.

Busy Expressions

an expression 'E' is said to be busy expression along some path P_i to P_j if and only if an evaluation of E' exist along some path P_i to P_j and no definition of any operand exist before its evaluation along the path.



These are useful in code-movement optimization.

Dataflow Analysis

Dataflow information can be collected by setting up and slaving systems of equations that related information at various points in a program

$$out[s] = gen[s] \cup [In[s] - KILL[s]]$$

this is read as information at the end of the statement either generated within the stmt or enters at the begining & is not killed as control flows through the stmts.

→ gen[s] is set of generations generated by s.

→ kill[s] is set of definitions that never reach the end of the s even if they reach begining

d_1 : $I = n-1$

d_2 : $J = m$

d_3 : $a = x_1$

do

d_4 : $I = I+1$

d_5 : $J = J-1$

if E then

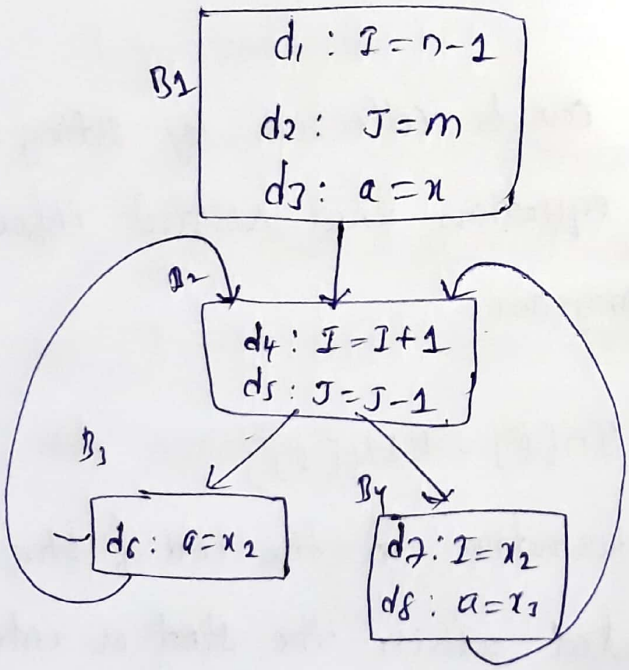
d_6 : $a = x_2$

else

d_7 : $I = x_2$

d_8 : $a = x_1$

while E_2



GEN [B1] = { d_1, d_2, d_3 }

KILL [B1] = { d_1, d_2, d_3, ~~d_4, d_5, d_6, d_7, d_8~~ }

KILL [B1] = 11000111

GEN [B2] = { d_4, d_5 }

KILL [B2] = { d_4, d_5, ~~d_7~~, ~~d_1, d_2~~ }

= 11000010

GEN [B3] = { d_6 }

KILL [B3] = { d_6 } − { d_3, d_8 }

=

Register Discriptor: It is a structure that maintains a pointer to the list that contains information about what is currently available in each of registers.

Address Discriptor: It is a structure that keeps track of the locations for each variable.

```
t = a - b
u = a - c
v = t + u
d = v + u
```

Mov R0, a	R0 contains a	a is in R0 & memory
sub R0, b	R0 contains t	a is in memory
Mov t R0		
Mov R1 a		
SUB R1 c		
Mov u R1		
Add R0 R1		
Add R0 R1		

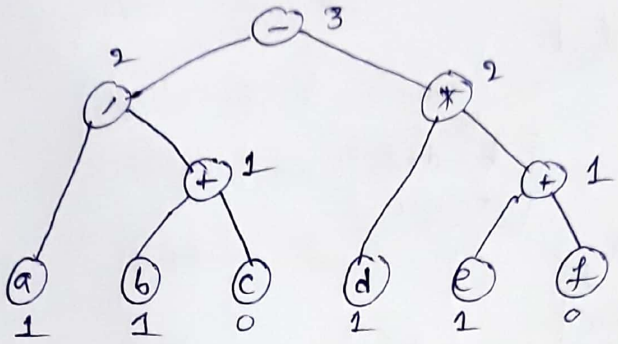
25/10/18

Code Generation Using DAG

① $a / (b+c) - d * (e+f)$

abc+/ eft+-

abc+/deft+*-



no. of reg required

If left is greater than right number allocate max no.

If left & right are equal add +1 to that number.

Available registers are R_1, R_2, R_3

	cost
MOV R_1, b	1+0+1 => 2
add R_1, c	
MOV R_2, a	
div R_2, R_1	1+1+1 => 3
MOV R_3, e	
add R_3, f	
MOV R_1, d	
mul R_1, R_3	
sub R_2, R_1	

Global Register Allocation

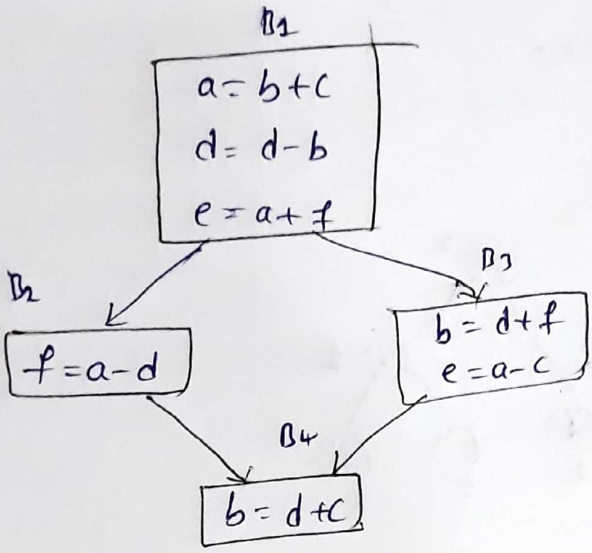
assign registers to the variables that has height usage count.

$$\sum_{\text{block } B} \text{use}(x, B) + 2 * \text{Live}(x, B)$$

use(x, B) – It gives the no. of times the variable x is used in block B.

Live(x, B) – it is true if x is live on exit from B

5



variables	B_1		B_2		B_3		B_4		usage count
	U	L	U	L	U	L	U	L	
a	1	1	1	0	1	0	0	0	5
b	2	0	0	0	0	1	0	1	6
c	1	0	0	0	1	0	1	0	3
d	1	1	1	0	1	0	1	0	6
e	0	1	0	0	0	1	0	0	4
f	1	0	0	1	1	0	0	0	4

Usage count:

$$\sum use(x, B) + 2 * Live(x, B)$$

$a = 1 + 2*(1) + 1 + 2\times(0) + 1 + 0 + 0 + 0$

$= 5$

$b = 2 + 2 + 2 = 6$

$c = 1 + 1 + 1 = 3$

$d = 3 + 1 + 1 + 1 = 6$

$e = 2 + 2 = 4$

$f = 1 + 2 + 1 = 4$

B_1 : MOV R_0, b

Mov R_3, c

add R_2 R_0 R_3

Mov R_2 R_0

Sub R_1 R_1 R_0

Mov R_3 f

add R_3 R_2 R_3

mov e R_3

B_2 : Sub R_3 R_2 R_1

Mov f R_3

B_3 : Mov R_3 f

add R_0 R_1 R_3

Mov R_3 c

Sub R_3 R_2 R_3

Mov e R_3

B_4 : Mov R_3 c

Add R_0 R_1 R_3