Unit -V

Code Optimization

→ principles of code optimization

- platform (or) Machine Independent
  - peephole optimization
  - Instruction level parallelism
  - Data level parallelism
  - Cache Optimization
- platform (or) MachineInDependent
  - common subexpression elimination
  - Compiletime evaluation
    - Constant folding
    - constant propogation
  - code movement (or) code motion
  - Dead Code elimination
  - Induction variable & strength reduction

Common Subexpression Elimination

It is an expression which appears repeatedly in the program which is computed previously, but the values of variables in expression not changes. such expressions are replaced

Eg:

$5 = a = b+c$	$a = b+c$	$a = 1$
$1 = b = a-d$	$b = a-d$	$b = 2$
$4 = c = b+c$	$c = b+c$	$c = 3$
$1 = d = a-d$	$d = b$	$d = 4$
∴ before	∴ after	

Same

so $d = b$ //

Eg2:

$S_1 = 4 * i$	$S_1 = 4 * i$
$S_2 = a[S_1]$	$S_2 = a[S_1]$
$S_3 = 4 * j$	$S_3 = 4 * j$
$S_4 = 4 * i$	$S_5 = n$
Same $S_5 = n$	$S_6 = b[S_1] + S_5$
$S_6 = b[S_4] + S_5$	

↓

instead of S_4 write S_1

Compile time Evaluation

Evaluation is done at compile time instead of runtime

Constant folding

Evaluate 1 expression and submit the result

$$\text{Area} = \pi r^2$$
$$= \frac{22}{7} \times 5 \times r$$
$$= 3.14 \times 5 \times r$$

constant propogation

propogate constant and replaces with a variable

$$r = 5 \quad \text{Area} = \pi r^2$$
$$= \frac{22}{7} \times r \times r$$
$$= 3.14 \times 5 \times 5$$

Code movement (or) Code motion

It moves the code outside of the loop if it does not map any differences, if it is executed inside or outside of the loop

```
for(j=0; j<n; j++)
{
  x=y+z;
a[j]=a[j]+4;
}
```

```
x=y+z
for(j=0; j<n; j++)
{
a[j]=a[j]+4;
}
```

The code which does not effect then it will be removed out of the loop

Dead code elimination

eliminate those statements which are never executed or if executed never used

```
int add(int x, int y)
{
 int z;
 z=x+y;
 return z;
 Pf("%d", n);
}
```

```
int add(int x, int y)
{
 int z;
 z=x+y;
 return z;
}
```

Induction variable & strength reduction

It is used to replaced variable from inner loop

→ temperary variable

strength reduction

replacing high priority operator with low priority operator

Eg: $x=2\times b \longrightarrow x=b+b$

both results in 2b

loop optimization

It is a machine independent optimization and as program inner loops takes bulk amount of time of a programmes.

so if we decrease the no of instructions in an inner loop then running time of a program may be include even if we increase the amount of code outside the loop

methods:-

- code motion or code movement
- loop fusion
- loop invariant computation
- loop unrolling

loop unrolling

loop overhead can be reduced by reducing no of iterations and replacing the body of the loop.

```
for(i=0; i<100; i++)
    add();
```

```
for(i=0; i<50; i++)
{
 add();
 add();
}
```

loop fusion

In this adjacent loops can be merged to reduce loop overhead and improve performance

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

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

loop invarient computation

The statements in the loop whose results of the computation do not change the iteration.

Eg :-

```
a=1, b=2, c=3;
for(i=0;i<n;i++)
{
  a=b+c;
  b=a+c;
  c=a+b;
  a[i]=a[i]*6;
}
```

loop invarient variables

```
a=1, b=2, c=3;
a=b+c;
b=a+c;
c=a+b;
for(i=0;i<n;i++)
{
  a[i]=a[i]*6;
}
```

Copy propagation

It is used in replacing the occurrences of target of direct assignment with their values.

→ A direct assignment is an instruction i.e

$$x = y$$

$$y = x \qquad \Big| \qquad y = x$$

$$z = y + 5. \qquad \Big| \qquad z = x + 5$$

Basic Blocks

- It is a sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt.

Algorithm :-

partition into basic blocks

Input :- A sequence of 3 address statements

output :- A list of blocks with each 3 address statements in exactly one block

method 1

- First determine the set of leaders i.e the first statement

of basic block. The rules are

- The first statement is a leader
- any statement which is target of a unconditional or conditional ~~go to~~ is leader
- any statement which immediately follows a conditional goto is a leader

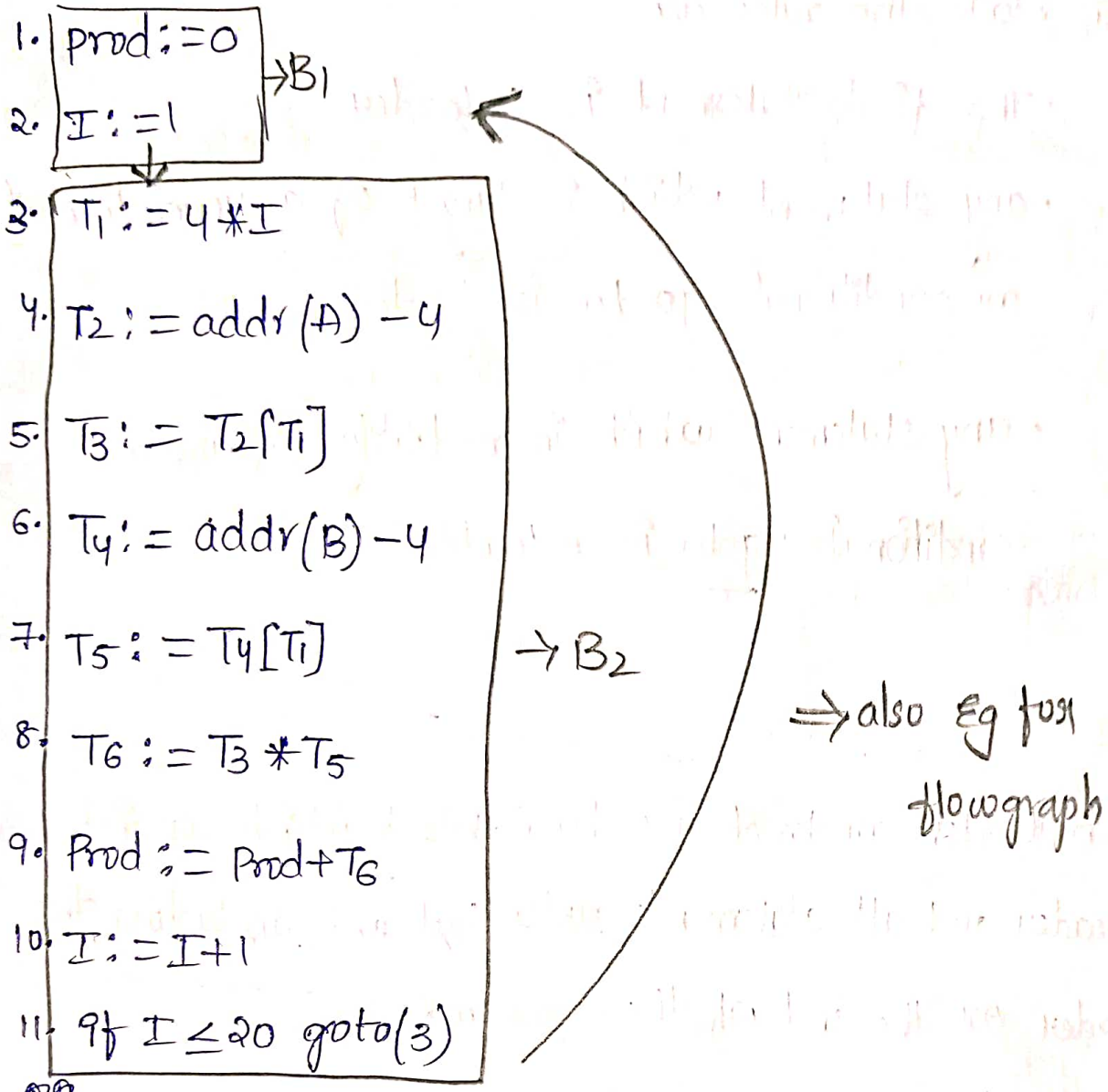
Method-2

→ For each leader construct its basic block which consist of the leader and all statements upto but not including the next leader or the end of the program.

→ Any statements not placed in a block can never be executed and removed

Eg: Three address computing product

```
begin
    prod := 0;
    I := 1;
do
    begin
        Prod := Prod + A[I] * B[I];
    end
while I ≤ 20
end
```



Flowgraph

It is useful to demonstrate the basic block & their successor relationships by their directed graph called flowgraph.

→ The nodes of the flowgraph are the basic graph

→ one node is denoted as initial which is the block whose leader is the first statement.

→ There is the directed edge from ~~top~~ block B_1 to block B_2 if B_2 could immediately follow B_1 during execution

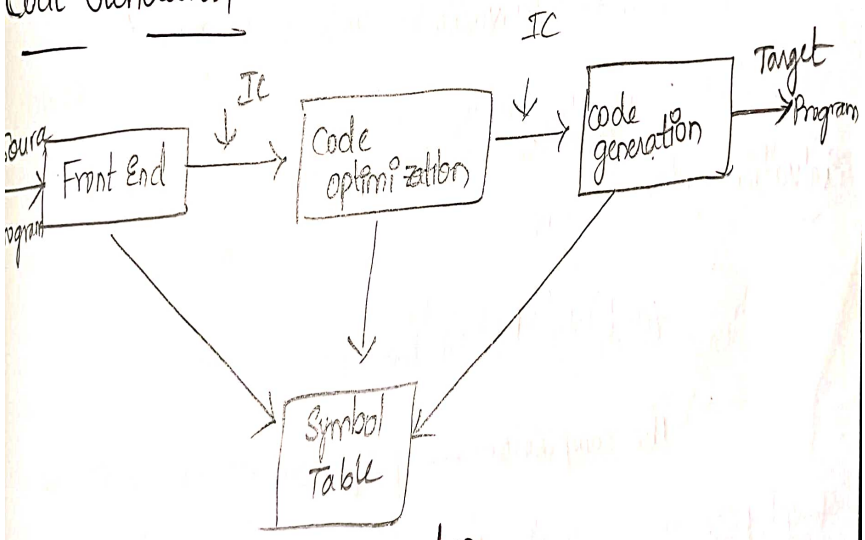
i.e

- There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2
- If B_2 immediately follows B_1 in the order of the program and B_1 does not end in an unconditional jump. In this B_1 is predecessor of B_2 and B_2 is the successor of B_1

Eg:- the above one

Code Generator

- take input from intermediate code then convert to target program



Issues (or) problems of code generator

→ Input to code generator
→ Target program → Absolute code (.exe)
relocatable code (obj file)
Assembly code
→ Memory management → Symbol-table
→ Instruction selection
→ register allocation Issues
→ Evaluation order.

```
X = Y + Z
LD  R0, Y
Add R0, Z
STORE X, R0
```

} selection of Instruction

Register allocation Issues

- register operands — In this we select the set of variables that will resides in registers at a point in a prg
- memory operands — In this we select a specific register that a variable resides in program

Evaluation order

$$(a-b) * (a+c) + d/e$$

The computations are performed effectively to the target program where the Instructions are of Independent. It uses 3 address code

$$T_1 = a - b$$

$$T_2 = a + c$$

$$T_3 = T_1 * T_2$$

$$T_4 = d/e$$

$$T_5 = T_3 + T_4$$

$$X = T_5$$

Machine model

We assume that a machine uses a byte addressable machine with 2^{16} bytes and uses 8 general purpose register R_0 to R_7. Each capable of holiday 16 bit.

Eg:- The binary operation

op S D

opcode uses 4 bit

source (s) and destination uses 6 bit field

The following addressing modes are used to represent assembly language memonic forms

r → register mode, contains operands

*r → indirect register mode, contains addres of operand

X(r) → indexed mode the value x store in the word following instn which is added to register r to produce address of operand

* X(r) L→

#x → immediate mode the world following instruction contains operand x

x → absolute mode

The address of x follows the instruction. the

following opcodes are used to perform operation

MOV
ADD
SUB
ST
LOAD
MUL

Eg :- The quadruple of the form A:=B+C are represented in different code sequence

1) mov B, R0
   ADD C, R0
   MOV R0, A } 6

2) mov B, A
   mov C, A } 6

3) mov *R1 *R0
   ADD *R2 *R0 } 2

4) ADD R2, R1
   mov R1, A } 3

Simple code generator

GETREG()
RD
AD

Code Generation Algorithm

Eg :- A:=B op C

Step1 : Invoke a function GETREG() to determine the location L where the computation BopC should be performed

Step2 : Consult the address descriptor for B to determine B' i.e the current location of B

- Prefer the register for B' if the value of B is currently in memory and register

if the value of B is not in L. Generate the instruction mov B', L to place a copy of B in L

Step3 : Generate the instruction op C', L where C' is the current location of C. update the address descriptor of A to indicate that A is location 'L'

step4: If the current values of B and C has no next uses exit from the block.

Evaluate the following expression into three address code sequence

$$X = (A-B)+(A-C)+(A-C)$$

$t_1 = (A-B)$

$t_2 = (A-C)$

$t_3 = t_1 + t_2$

$t_4 = t_3 + t_2$

Stmt	code generated CG	Register Descriptor	Address Descriptor
	MOV A, R0	R0 contains t_1	t_1 in R0
$t_1 := A-B$	SUB B, R0		
	MOV A, R1	R0 contains t_1	t_1 in R0
$t_2 := A-C$	SUB C, R1	R1 contains t_2	t_2 in R1
		R0 contains t_3	t_3 in R0
$t_3 := t_1 + t_2$	ADD R1, R0	R1 contains t_2	t_2 in R1
	ADD R1 R0	R0 contains t_4	t_4 in R0
$t_4 := t_3 + t_2$	MOV R0, t_4		t_4 in R0 & memory