

CONTENTS :

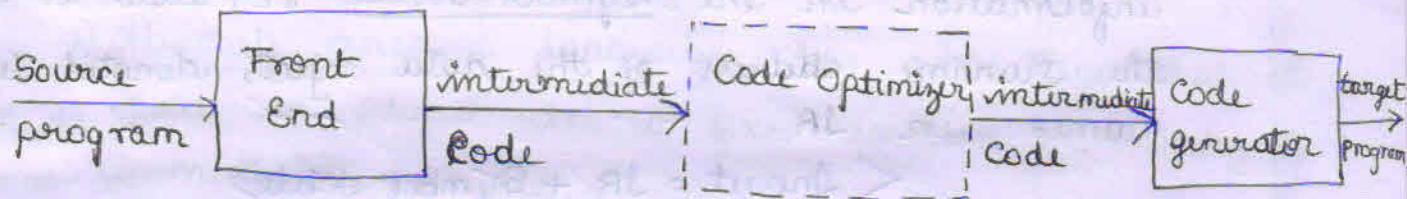
- > Issues in the design of Code generator
- > The target language
- > Addresses in the target code
- > Basic blocks & Flow graphs
- > Optimisation of basic blocks
- > A simple code generator

UNIT - 8

CODE GENERATION

INTRODUCTION :

- * Code generation is the final phase in the compiler design.
- * The code optimizer accepts intermediate code representation which is generated from the front end of the compiler & produces another intermediate code representation which is optimized.
- * Code generator takes intermediate representation produced by code optimizer along with supplementary information in symbol table of the source program & produce as output an equivalent target program.



- * Code generator has 3 main tasks:

- 1) Instruction selection
- 2) Register allocation & assignment
- 3) Instruction Ordering

1) INSTRUCTION SELECTION :

Choose appropriate target machine codes instructions to implement the IR [intermediate representation statements]

2) REGISTER ALLOCATION & ASSIGNMENT :

Decide what values to keep in which registers

3) INSTRUCTION ORDERING

Decide in what order to schedule the execution of instructions.

8.1

ISSUES IN THE DESIGN OF CODE GENERATOR:

1) Input to the code generator

2) The target program

3) Instruction selection

4) Register Allocation

5) Evaluation Order

1) Input to the code generator

* Input to the code generator is the intermediate representation of the source program produced by the front end along with information in the symbol table i.e., used to determine the runtime address of the data objects denoted by the names in IR.

< Input = IR + Symbol table >

* IR has several choices

(a) 3-address representation : quadruples, triples, indirect triples

(b) Virtual machine representation : byte codes of stack machine codes

(c) Linear representation such as postfix notation

(d) Graphical representation such as syntax trees or DAG's

* Assumptions made are

(i) Front end produces low-level IR, i.e., values of names in it can be directly manipulated by the machine instruction.

(ii) Syntactic & semantic errors have been already detected

2) The Target Program:

- * The output of code generator is target program.
- * The instruction set architecture of the target machine has a significant impact on the design of code generator.
- * Most common architectures are:
 - (a) CISC: It has few registers, has maximum of 2 operands & variety of addressing mode, variable length instructions & instruction with side effects.
 - (b) RISC: It has many registers, has maximum of 3 operands with simple addressing modes, & relatively simple instruction set architecture.
- * Output may take variety of forms.
 - a) Absolute machine language [Executable code]
 - b) Relocatable machine language [object files for linker]
 - c) Assembly language [facilitates debugging]
- a) Absolute machine language has advantage that it can be placed in a fixed location in memory & immediately executed.
- b) Relocatable machine language program allows subprograms to be compiled separately.
- c) Producing Assembly language program as output makes the process of code generation somewhat easier.

3) Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine.

* The complexity of performing this mapping is determined by the factors such as:

- (i) the level of the IR
- (ii) the nature of the instruction set architectures.
- (iii) the desired quality of the generated code.

(i) the levels of the IR:

- > If the [IR is high level], use code templates to translate each IR statements into a sequence of machine instruction.
- > produces poor code, needs further optimizatⁿ.
- > If the [IR is low level], use ~~code, this~~ ^{low level} information to generate more efficient code sequence.

(ii) the nature of the instruction set architectures has strong effect on difficulty of instruct^r select^r.

- > Uniformity & completeness of the instruct^r set are imp factors.
- > If we do not care about the efficiency of the target program, instruct^r select^r is straightforward.

> For eg:

| | |
|-------------------------|---|
| $x = y + z \Rightarrow$ | LD R ₀ , y |
| | ADD R ₀ , R ₀ , z |
| | ST x, R ₀ |

∴ produces redundant LD & store

eg2:

| | |
|-------------------------|---|
| $a = b + c \Rightarrow$ | LD R ₀ , b |
| $d = a + b$ | ADD R ₀ , R ₀ , c |

| |
|-----------------------|
| ST a, R ₀ |
| LD R ₀ , a |

| |
|---|
| ADD R ₀ , R ₀ , c |
| ST d, R ₀ |

REDUNDANT

(iii) the quality of the generated code is determined by its speed & size.

> For eg:

$$\begin{array}{l} a = a + 1 \rightarrow LD R_0, a \\ \quad \quad \quad ADD R_0, R_0, \#1 \\ \quad \quad \quad ST a, R_0 \end{array} \quad \begin{array}{l} \text{replaced} \\ \text{by} \end{array} \quad \begin{array}{l} INC a \end{array}$$

4) Register Allocation:

* Instrucⁿ involving register operands are usually shorter & faster than those involving operands in memory.

* 2 subproblems:

(i) Register allocation: Select the set of variables that will reside in registers at each point in the program.

(ii) Register assignment: Select specific register that a variable will reside in.

* Complications imposed by the hardware architecture
Eg: Register pairs for multiplication & division.

* Multiplication instrⁿ is of the form

$$M \boxed{x, y}$$

where x → Multiplicand, is the odd register of an even/odd register pair.

y → Multiplier, is a single register.

⇒ Product → occupies the entire even/odd register pair.

* Division instrⁿ is of the form

$$D \boxed{x, y}$$

where x → dividend, occupies even register

y → divisor, occupies odd/even register

⇒ Quotient → stored in odd register

remainder → stored in even register

Eg: two 3-address code sequences

$$t = a + b$$

$$t = t * c$$

$$t = t / d$$

$$t = a + b$$

$$t = t + c$$

$$t = t / d$$

Optimal machine-Code sequences

L R1, a

A R1, b

M R0, c

D R0, d

ST R1, t

L R0, a

A R0, b

A R0, c

SRDA R0, 3R

D R0, d

ST R1, t

5) Evaluation Order:

- * The order in which computations are performed can effect the efficiency of the target code.
- * When instructions are independent their evaluation order can be changed.
- * Some computation orders require fewer registers to hold intermediate results than others.
- * However picking a best order in the general case is a difficult NP-complete problem.

ADDITIONAL INFORMATION: Eg

$$t_1 = a + b$$
$$a + b - (c + d) * e \Rightarrow t_2 = c + d$$

$$t_3 = e * t_2$$

$$t_4 = t_1 - t_3$$

Reorder↓

$$t_2 = c + d$$

$$t_3 = e * t_2$$

$$t_1 = a + b$$

$$t_4 = t_1 - t_3$$

MOV R0, a

ADD R0, b

MOV R1, R0

MOV R1, c

ADD R1, d

MOV R0, e

MUL R0, R1

MOV R1, t1

SUB R1, R0

MOV B4, t4, R1

MOV R0, c

ADD R0, d

MOV R1, e

MUL R1, R0

MOV R0, a

ADD R0, b

SUB R0, R1

MOV t4, R0

THE

8.2 THE TARGET LANGUAGE:

For designing a good code generator, we need to have familiarity with target machine & its instruction set. Instead of generating code on a specific target machine, a general machine consisting of many registers are considered.

A SIMPLE TARGET MACHINE MODEL:

The characteristics of target machine mode with instruction format & instruction set are shown below:

* Our hypothetical machine:

(i) It is a 3-address machine with the following format

OP destination, Source1, Source2

NOTE:

A 3 address instruction can have 2 operands or 1 operand also but it can have max of 3 operands

(ii) The target machine is byte addressable i.e., it can access 8 bit of info from specific address

(iii) It has n no of Registers denoted by

$R_0, R_1, R_2, \dots, R_{n-1}$

* Various types of instruction that are used by target m/c

(i) Load Instruction

(ii) Store Instruction

(iii) Computational Instruction

(iv) Unconditional Instruction

(v) Conditional Instruction

(i) Load Instruction: Used to copy the data into destination operand which must be a register.

SYNTAX: LD dest, addr

where addr operand \rightarrow register or memory locat

(ii) Store instruction: Used to copy the data into mem locatⁿ specified in the destinatⁿ operand.

SYNTAX: ST dst, or

where dst \rightarrow destination & it is a mem location

or \rightarrow register.

Computational operation.

(iii) Arithmetic instruction: They are performed using these instruction.

SYNTAX: OP dst, Src1, Src2.

where 1st operand, dst \rightarrow destination

2nd & 3rd operand \rightarrow Operands where R values fetched for Operatⁿ to be per

Eg1: ADD R0, R1, RR // $R0 = R1 + RR$

Eg2: SUB R0, R0, R1 // $R0 = R0 - R1$

Eg3: MUL RR, R0, R1 // $RR = R0 * R1$

(iv) Unconditional Jumps: The branch instructⁿ without any conditⁿ are called unconditional jumps.

SYNTAX: BR label

where BR \rightarrow BRanch instructⁿ

(v) Conditional Jumps: Based on the value stored in a register i.e., whether it is true or zero or -ve, if branching takes place, then the branch instⁿ are called conditional jumps.

SYNTAX: Bcond or, label

where B stands from Branch,

Cond can be LT, GT, LTX, GTEX

Less than or equal
Less than
Greater than
Greater than or equal

or → register, contains value such as 0, +ve or -ve.

Eg1: BL RO, T1

// Branch to T1, if RO contains +ve value

Eg2: BLTR RI, TR

// Branch to TR, if RI contains either 0 or -ve value

- * Different addressing modes supported by generalized target machine:

1) Direct addressing mode

2) Indexed —————

3) Integer Indexed —————

4) Indirect —————

5) Immediate —————

(i) Direct A/M:

Address of the data to be accessed is directly present in the instructn, i.e., location is identified by a variable name x.

Eg: LD P LD RI, x

// Load value stored in memory locatⁿ x into RI

(ii) Indexed A/M: The data can be accessed from a memory locatⁿ using index. This addressing mode is useful for accessing arrays, where a is the base address of the array & register holds the index value

Eg: LD RI, a(RR)

// Access the data stored in
RI = contents(a + contents(RR))

(iii) Indexed A/M where memory locatⁿ is integer

It is same as previous one except that a memory locatⁿ is identified as integer.

Eg: LD RI, 100(RR)

// RI = contents(100 + contents(RR))

(iv) Indirect A/M: Contents of the data can be accessed by differencing using * operators as shown below:

LD R1, *(R2)

// R2 contains memory locⁿ
the data stored in that
memory locⁿ is copied in
register R1

LD R1, *100(R2)

// R1 = contents(contents(100+
contents(R

(v) Immediate A/M: The data to be manipulated is directly present in the instruction & preceded by

LD R1, #100

// R1 $\leftarrow 100$

EXERCISE:

code for

1. Generate 3 address statement for $x = y - z$

LD R1, y // R1 = y

LD RR, z // RR = z

ADD R1, R1, RR // R1 = R1 + RR

ST x, R1 // ~~R1~~ x = R1

code for

2. Generate 3 address statement $x = *p$

LD R1, p // R1 $\leftarrow p$

LD RR, 0(R1) // RR = contents(0 + contents(R1))

ST x, RR // x = RR

3. Generate code for 3 address statement $*p = y$

LD R1, p // $R1 = p$
LD RR, y // $RR = y$
ST O(R1), RR // contents(O + contents(R1)) = RR

4. Generate m/c code for 3 address statement $b = a[i]$

LD R1, i // $R1 = i$
MUL R1, R1, 8 // $R1 = R1 * 8$
LD RR, a[R1] // $RR = \text{contents}(a + \text{contents}(R1))$
ST b, RR // $b = RR$

5. Generate m/c code for 3 address statement $a[j] = c$

LD R1, j // $R1 = j$
LD RR, c. // $RR = c$
MUL R1, R1, 8 // $R1 = R1 * c$
ST a[R1], RR // $\text{contents}(a + \text{contents}(R1)) = RR$

6. Generate m/c code for 3 address statement
if $x < y$ goto L

LD R1, x // $R1 = x$
LD RR, y // $RR = y$
SUB R1, R1, RR // $R1 = R1 - RR$
BUTZ R1, M // if $R1 < 0$ jump to M

Program & Instruction Cost

* For simplicity we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.

- * A/M involves registers have zero additional cost.
- * A/M involving memory locatⁿ or constant have additional cost of 1.

* For example:

- a) LD R0, RI \rightarrow cost = 1
- b) LD R0, M \rightarrow cost = 2.
- c) LD RI, *100(RR) \rightarrow cost = 3

* Cost of Addressing mode:

| Mode A/M | Form | Address | Added Cost |
|----------------------------|-------|-------------------------|------------|
| ① Absolute direct A/M | M | M | 1 |
| ② Register direct A/M | R | R | 0 |
| ③ Indexed A/M | C(R) | C+contents(R) | 1 |
| ④ Indirect register A/M | *R | contents(R) | 0 |
| ⑤ Indirect indexed A/M | *C(R) | contents(C+contents(R)) | 1 |
| ⑥ Immediate A/M | #C | N/A | 1 |

NOTE: Cost of each statement = 1 + cost (Addressing mode)

EXERCISES (8.2)

1. Determine the costs of the following instruction sequence

| | |
|----------------|-----------------------|
| LD R0, Y | Cost = 1 + cost(A.M) |
| LD RI, Y | Cost = 1 + 1 = 2 |
| ADD R0, R0, RI | Cost = 1 + 1 = 2 |
| ST x, R0 | Cost = 1 + 0 = 1 |
| | Cost = 1 + 1 = 2 |
| | <u>Total Cost = 7</u> |

2. LD R0, i

MUL R0, R0, 8

LD RI, a(R0)

ST b, RI

| | |
|---------------|------------------------|
| LD R0, i | Cost = Cost (A.M) + 1. |
| MUL R0, R0, 8 | Cost = 1 + 1 = 2 |
| LD RI, a(R0) | Cost = 1 + 1 = 2 |
| ST b, RI | Cost = 1 + 1 = 2 |
| | <u>Total cost = 8</u> |

3. LD R0, C

LD RI, i

MUL RI, RI, 8

ST a(RI), R0

| | |
|---------------|-----------------------|
| LD R0, C | Cost = cost (A.M) + 1 |
| LD RI, i | 1 + 1 = 2 |
| MUL RI, RI, 8 | 1 + 1 = 2 |
| ST a(RI), R0 | 1 + 1 = 2 |
| | <u>Total cost = 8</u> |

4. LD R0, P
 LD RI, O(R0)
 ST x, RI

$$\text{Cost} = \text{Cost(A.M)} + 1$$

~~S = 1~~
 LD R0, P
 LD RI, O(R0)
 ST x, RI

$$1 + 1 = 2$$

$$1 + 1 = 2$$

$$1 + 1 = 2$$

$$\text{Total Cost} = 6$$

~~1 = 0~~ // of constan

5. LD R0, P
 LD RI, x
 ST O(R0), RI

$$\text{Cost} = \text{Cost(A.M)} + 1$$

~~S = 1 + 1 = 2~~
 LD R0, P

$$1 + 1 = 2$$

~~S = 2 + 1 = 3~~
 LD RI, x

$$1 + 1 = 2$$

~~S = 1 + 1 = 2~~
 ST O(RI), RI

$$1 + 1 = 2$$

$$\text{Total Cost} = 6$$

6. LD R0, x
 LD RI, y
 SUB R0, R0, RI
 BLTR *R3, R0

$$\text{Cost} = 1 + \text{Cost(A.M)}$$

~~S = 1 + 1 = 2~~
 LD R0, x

$$1 + 1 = 2$$

~~S = 2 + 1 = 3~~
 LD RI, y

$$1 + 1 = 2$$

~~S = 1 + 1 = 2~~
 SUB R0, R0, RI

$$1 + 0 = 1$$

~~S = 1 + 1 = 2~~
 BLTR *R3, R0

$$1 + 1 = 2$$

<:: indirect A

$$\text{Total Cost} = 7$$

8.3 ADDRESS IN THE TARGET CODE

* We show how names in the IR can be converted into addresses in the target code by looking at code generated for simple procedure calls. & returns using static & stack allocation.

* Each executing program runs in its own logical address space that was partitioned into Code & data areas.

1. A statically determined area Code, that holds the executable target code.

2. A statically determined data area Static, for holding global constants & other data generated by the compiler.

3. A dynamically managed area Heap, for holding data objects that are allocated & freed during prog exec?

4. A dynamically managed area, Stack for holding activation records as they are created & destroyed during procedure calls & returns

* R standard storage allocatⁿ strategies namely:

(a) Static allocation

(b) Stack allocation.

* In static allocation, the position of an activatⁿ record in memory is fixed at compile time.

* In stack allocatⁿ, a new activatⁿ record is pushed onto stack for each executⁿ of the procedure. The record is popped when the activatⁿ ends.

(a) STATIC ALLOCATION

- * The 3 address code for functⁿ call consists of following statements
 - 1. call
 - 2. return
 - 3. halt
 - 4. action
- * Call statement used for functⁿ call, it has to mail the control to the functⁿ along with save the status of current functⁿ.
- * Return statement to give control back to called functⁿ.
- * Action defines other operatⁿ or instructⁿ for assignment or flow control statements.
- * Halt indicates the completⁿ of operatⁿ of called functⁿ.
- * Let us consider code needed to implement the simplest static allocatⁿ case:

Here "call calle" statement can be implemented by sequence of 2 target m/c inst

| | |
|----|---|
| ST | staticArea calle. staticArea, #here + R0 |
| BR | calle. codeArea |

- ST instructⁿ saves return address
- BR transfers control to the target code for called procedure.
- calle. staticArea, calle. codeArea are con referring to the address of activation record of first instructⁿ for the called procedure.
- #here + R0 in the ST instructⁿ is return address
- Cost of 2 instructⁿ is 5.

3 address code.

/* Code for c */

action 1

call p

action 2

halt

/* Code for p */

action 3

return

Target program for above 3 address code will be generated as follows:

100 : ACTION1

/* code for c */

120 : ST 364, #140

// code for action 1

// store return address
140 in locat² 364

130 : BR 200

// call p

140 : ACTIONR

// return to OS

160 : HALT

200 : ACTION3

// return to address
saved in 364

220 : BR *364

300 : // 300-363 hold act. no. for c
304 : // return address
..... // local data for c

364 : // 364-451 holds activatⁿ

// record for p
// return address
368 : // local data for p

.....

(b) STACK ALLOCATION:

- * Static allocatⁿ can become stack allocatⁿ by using relative address for storage in activatⁿ records. The positⁿ of the record for an activatⁿ of a procedure is not known until run time.
- * In stack allocatⁿ, this positⁿ is usually stored in register, so words in the activatⁿ records can be accessed as offsets from the value in this register.
- * When a procedure call occurs, the calling proced increments SP & transfers control to the ca procedure.
- * After control returns to the caller, it decrements SP, thereby deallocating the activatⁿ records of the called procedure.
- * The code for the 1st procedure initializes the stack by setting SP to the start of the stack area in the memory.

```
LD SP, #stack_start //Initialize the stack  
code for the first procedure  
HALT //terminate executn
```

- * A procedure call sequence increments SP, saves the return address, & transfers control to the called procedure

```
ADD SP, SP, #caller_record_size //increment stack pos  
ST O(SP), #here +16 //save reti address
```

```
BR Callic_code Area //jump to the calle
```

- * The attribute `caller.recordsize` represents size of an activatⁿ record, so the ADD instrucⁿ have SP pointing to the beginning of the next activatⁿ record.
- * The source #here+16 in the ST instrucⁿ in the address of the instrⁿ following BR statement, it is saved in the address pointed to by SP.
- * The return sequence consists of 2 parts. The called procedure transfers control to the return address using

$\text{BR } \text{gotō} \quad *O(\text{SP}) \quad //\text{return to caller}$

→ The reason for using $*O(\text{SP})$ in the BR instrucⁿ is that we need 2 levels of indirectⁿ: $O(\text{SP})$ is the address of the first word in the activatⁿ record & $*O(\text{SP})$ is the return addr saved there.

→ The 2nd part of the return sequence is the caller which increments SP, thereby restoring SP to its previous value i.e., after subtractⁿ SP points to the beginning of the activatⁿ record of the caller:

$\text{SUB } \text{SP}, \text{SP}, \# \text{caller. record Size}$
 $//\text{increment Stack pointer}$

- * Following prog is a condensatⁿ of the 3 address code for the pascal prog for reading & sorting integers. procedure q is recursive, so more than one or activatⁿ of q can be alive at the same time

//code for an

action 1

call q

action 2

halt

// code for p
action 3
return

// code for q
action 4
call p
action 5
call q
action 6
call q
return

Target code for stack allocation:

// code for s

100: mov #1

// code for m

100: LD SP, #600 // Initialize the stack.

108 : ACTION1

// code for action 1

1R8 : ADD SP, SP, #mSize // call sequence begin

136 : ST O(SP), #15R // push return address

144 : BR 300 // call q,

15R : SUB SP, SP, #mSize // restore SP

160 : ACTION2

180 : HALT

200 : ACTION3

// code for p

220 : BR *O(SP) // return

300 : ACTION

// code for q

// contains a conditional
to 456

320 : ADD SP, SP, #qSize

328 : ST O(SP), #34 // push return address

| | | |
|---------------|-----------------|-----------------------|
| 336: BR | 300 | //call p |
| 344: SUB | SP, SP, #q.size | push q.size |
| 352: ACTION 5 | | |
| 372: ADD | SP, SP, #q.size | |
| 380: ST | O(SP), #396 | //push return address |
| 388: BR | 300 | //call q |
| 396: SUB | SP, SP, #q.size | |
| 404: ACTION 6 | | |
| 424: ADD | SP, SP, #q.size | |
| 432: ST | O(SP), #HND | //push return address |
| 440: BR | 300 | //call q |
| 448: SUB | SP, SP, #q.size | |
| 456: BR | *O(SP) | //return |
| 600: | | //stack starts here |

RUN TIME ADDRESS FOR NAMES

- * Assumptⁿ: A name in a 3-address statement is really a pointer to a symbol table entry for that name.
- * Note that names must eventually be replaced by code to access storage locations.

* Eg:

$\rightarrow x = 0$

\rightarrow Suppose the symbol table entry for x contains a relative address 1R

\rightarrow x is in statically allocated area begining at address static

\rightarrow the actual runtime address of x is static + 1R

\rightarrow The actual assignment: Static[1R] = 0

\rightarrow For a static area starting at address 100: LD 11R, #0

* This approach makes compiler portable, since front end need not be changed even when compiler moved to a machine with varied machine architecture.

~~8.4~~ BASIC BLOCKS & FLOW GRAPHS

Basic blocks & flow graphs are very useful in optimizing the intermediate code.

BASIC BLOCKS

- * Representation of intermediate code as a graph
 - nodes of the graph are basic block, where the flow of control can only enter at the 1st instr & leave through the last.
 - Edges indicate which blocks can follow other blocks, representing jumps in the instr.
- * Useful for discussing code generation

BASIC BLOCKS

- * Definition:
 - A basic block is a maximal sequence of TAC [Three address code] such that :
 - Flow of control enters at the beginning of block
 - Flow of control leaves at the end of basic block
 - No possibility of halting or branching except the last instruction in the block.

* Algorithm to obtain the basic block : 8.4 (a)

Algorithm : Partitioning 3 address instructions into basic blocks.

Input : A sequence of 3-address instructions.

Output : A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

Method : First, we determine those instructions in the intermediate code that are leaders, rules for finding leaders are :

1. The first 3 address instruction in the intermediate code is a leader
2. Any instruction i.e., the target of the conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader

* Find the leaders for below ~~new~~ Intermediate code to set a 10×10 matrix to an identity matrix.

Eq: 8.7

1) $i = 1$

Solution

leader

2) $j = 1$

leader

3) $t1 = 10 * i$

leader

4) $t2 = t1 + j$

leader

5) $t3 = 8 * t2$

leader

6) $t4 = t3 - 88$

leader

7) $a[t4] = 0.0$

leader

8) $j = j + 1$

leader

9) $\text{if } j \leq 10 \text{ goto (3)}$

leader

10) $i = i + 1$

leader

11) $\text{if } i \leq 10 \text{ goto (2)}$

leader

12) $i = 1$

leader

NOTE

Source code

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i,j] = 0.0;
    for i from 1 to 10 do
        a[i,i] = 1.0;
```

13) $t5 = i - 1$ record all records of leader

14) $t6 = 88 * t5$ ~~pushed to stack~~ : record of leader

15) $a[t6] = 1.0 - 5$ ~~pushed to stack~~ : record

16) $i = i + 1$ record all ~~in~~ to task k : ~~temp~~

17) if $i \leq 10$ goto [13]

Conclusion:

- * Instruction 1, 2, 3, 10, 12 & 13 are leaders.
- * Since basic block of each leader contains all instruction from itself until just before the next leader
- * Thus basic block 1 = just 1
- basic block 2 = just 2
- basic block 3 = 3 to 9
- basic block 4 : 10 to 11
- basic block 5 : 12
- basic block 6 : 13 to 17

NEXT USE INFORMATION

- * The user of a name in a 3 address statement
 - The 3 address statement i assigns a value
 - Statement j has x as an operand
 - Control can flow from statement i to j along a path that has no intervening assignments to x
 - Then statement j uses the value of x computed at i
 - Say that x is live at statement i.

For eg:

i: $x = a$

.....
.....

j: $y = x$

// Statement i assigns a value
to x

} x is never assigned a new
value to x

// statement j uses value of x
computed at i

* Algorithm of determining the liveliness of next use information

Algorithm (for $x = y + z$) : Determining the liveliness of next use information for each statement in a basic block.

Input : A basic block B of three-address statements.
Assume the symbol table initially shows all non-temporary variables in B as being live on exit.

Output : At each statement:
 $i: x = y + z$
in block B, attach to i the liveliness of next use information of x, y & z.

Method : Start at the last statement in B & scan backwards to the beginning of B. At each statement $i: x = y + z$ do the following:

1. Attach to i the information currently found in the symbol table regarding the next use of liveliness of x, y, z.

2. In symbol table, set x to "not live" & "no next use".

3. In symbol table, set y & z to "live" & next uses of

$y \oplus z$ to i .

NOTE:

If the 3-addr statement i is of the form

" $x = +y$ " or " $x = y$ "

the steps is same as above ignoring z . This algo stores the information whether the code is live or dead & whether the variable has next use info or not in symbol table.

FLOW GRAPHS

* Definition:

A flow graph is a graphical depiction of sequence of instruction with control flow edges, which show how the control flow from 1 basic block to other basic block.

* → A node represents basic block

- An edge represents control flow from 1 Basic block

→ A flow graph can be defined at the intermediate ^{code} level or target code level.

* Write flow graph for below source code

for i from 1 to 10 do

 for j from 1 to 10 do

$a[i, j] = 0.0$;

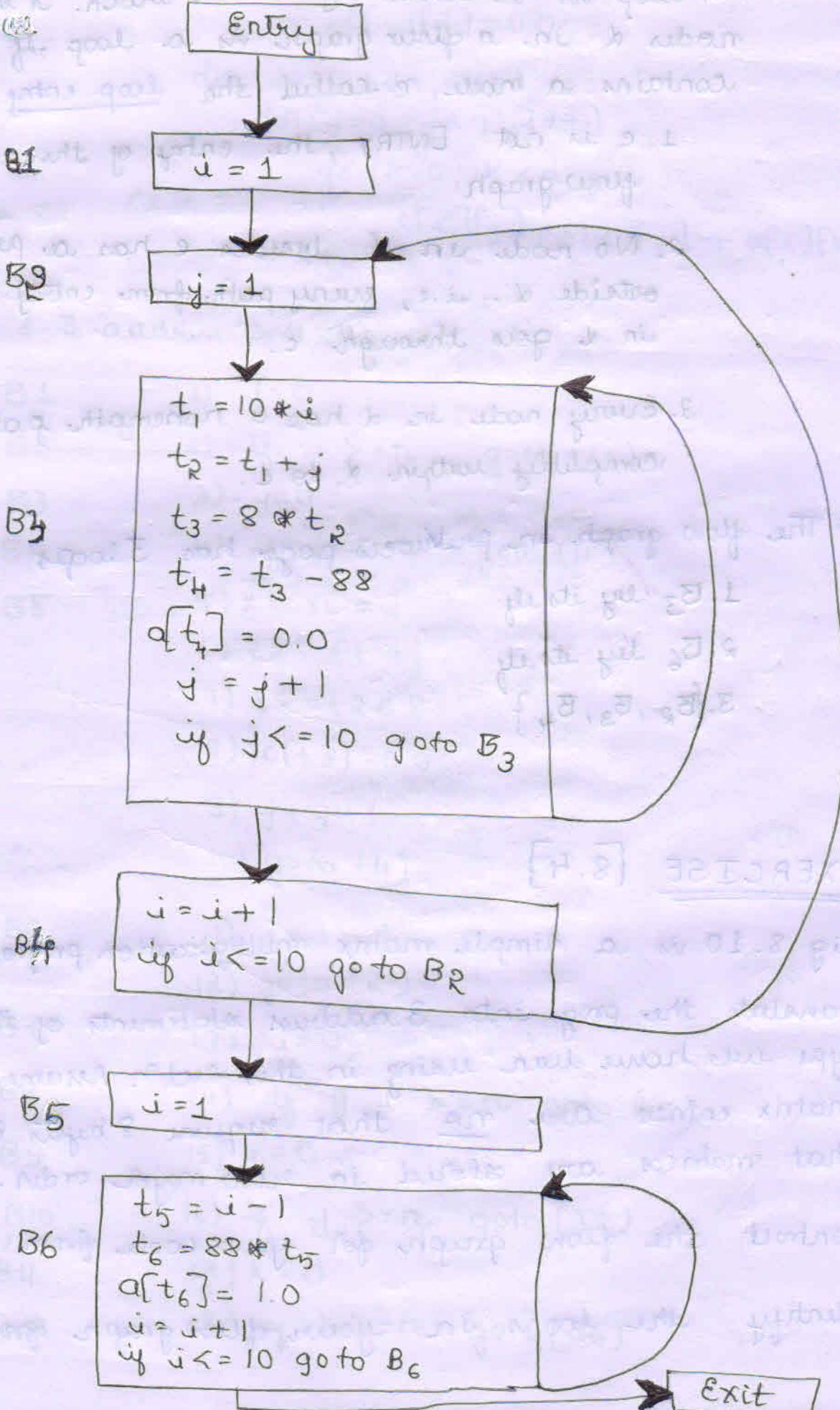
 for i from 1 to 10 do

$a[i, i] = 1.0$;

Soln:

fig 8.9

intermediate code \Rightarrow refer to eq: 8.7



LOOPS

- * A loop is a collectⁿ of basic block. A set of nodes δ in a flow graph is a loop, if δ contains a node e called the loop entry, such that
 1. e is not ENTRY, the entry of the entire flow graph
 2. No node in δ besides e has a predecessor outside δ . i.e., every path from entry to no node in δ goes through e .
 3. Every node in δ has a nonempty path, completely within δ , to e .

* The flow graph in previous page has 3 loops

1. B_3 by itself
2. B_6 by itself
3. $\{B_2, B_3, B_4\}$

EXERCISE [8.4]

Fig 8.10 is a simple matrix multiplication program.

- a) Translate the prog into 3 address statements of the type we have been using in this sect^r. Assume the matrix entries are no that require 8 bytes of that matrix are stored in row major order.
- b) Construct the flow graph for your code from (a)
- c) Identify the loops in your flow graph from (b)

```

for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];

```

Soln:

1. 3 address statements

- B1 1) i = 0
- B2 2) if i >= n goto (13)
- B3 3) j = 0
- B4 4) if j >= n goto (11)
- B5 5) t1 = n * i
- 6) t2 = t1 + j
- 7) t3 = t2 * 8
- 8) c[t3] = 0.0
- 9) j = j + 1
- 10) goto (4)
- B6 11) i = i + 1
- 12) goto (2)
- B7 13) i = 0 stop (BB)
- B8 14) if ! i >= n goto (40)
- B9 15) j = 0
- B10 16) if j >= n goto (38)
- B11 17) k = 0
- B12 18) if k >= n goto (36)

B13

$$19) t_4 = n * i$$

$$20) t_5 = t_4 + j$$

$$21) t_6 = t_5 * 8$$

$$22) t_7 = c[t_6]$$

$$23) t_8 = n * i$$

$$24) t_9 = (\cancel{t_9 * 8} \quad \cancel{t_8 * k}) \quad t_8 + k$$

$$25) t_{10} = \cancel{t_9} * 8$$

$$26) t_{11} = a[t_{10}]$$

$$27) t_{12} = n * k$$

$$28) t_{13} = t_{12} + j$$

$$29) t_{14} = t_{13} * 8$$

$$30) t_{15} = b[t_{14}]$$

$$31) t_{16} = t_{11} * t_{15}$$

$$32) t_{17} = t_7 + t_{16}$$

$$33) c[t_{16}] = t_{17} + ($$

$$34) k = k + 1$$

35) goto (18)

B14

$$36) j = j + 1$$

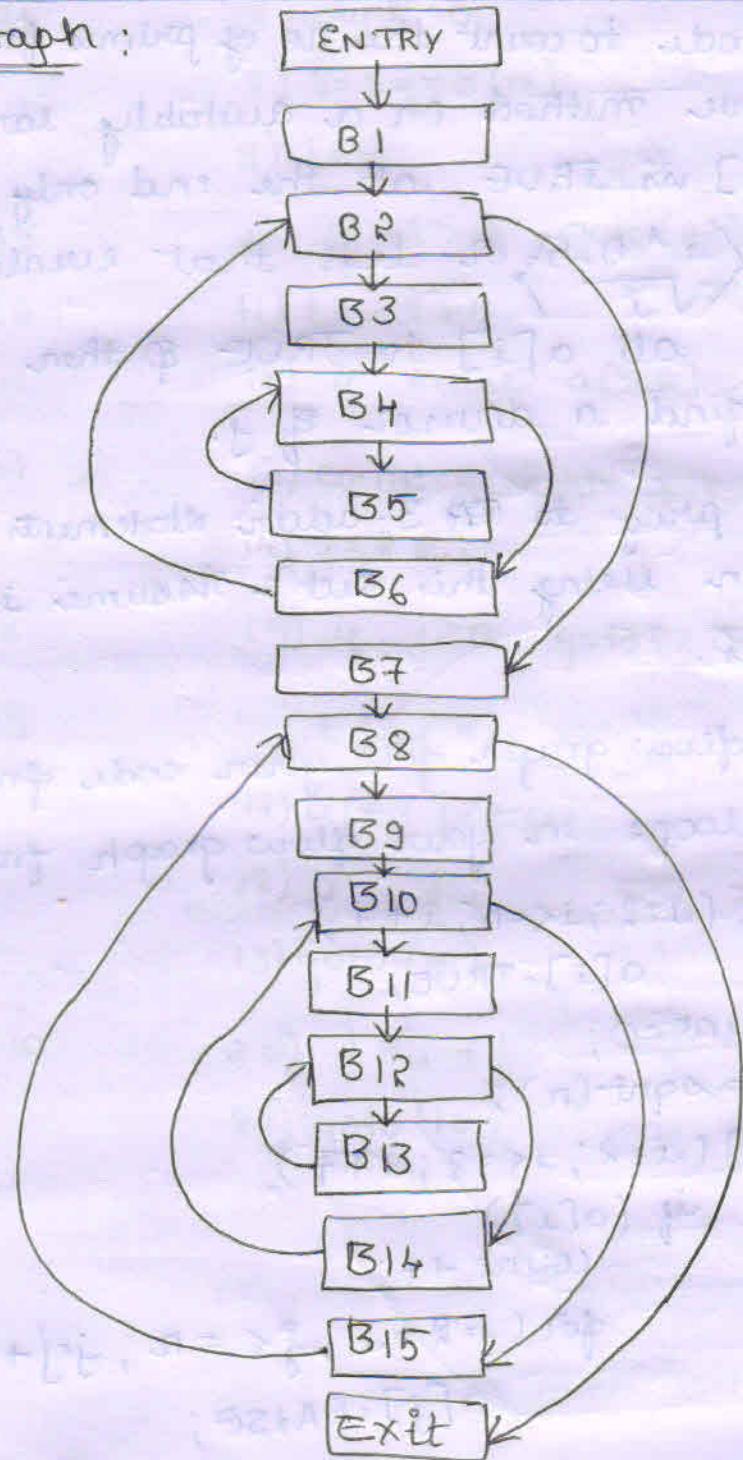
37) goto (16)

B15

$$38) i = i + 1$$

39) goto (14)

b) Flow graph:



c) loops

1. {B2, B3, B4, B6}
2. {B4, B5}
3. {B8, B9, B10, B15}
4. {B10, B11, B12, B13, B14}
5. {B12, B13}

② Fig 8.11 is code to count the no of primes from 2 to \sqrt{n} using the sieve method on a suitably large arr. That is, $a[i]$ is TRUE at the end only if there is no prime $\leq \sqrt{i}$ or less than evenly divides i . We initialize all $a[i]$ to TRUE & then set $a[j]$ FALSE if we find a divisor of j .

a) Translate the prog to 3-addr statements of the type we have been using this sectⁿ. Assume integers require 4 bytes.

- b) Construct the flow graph for your code from (a)
 c) Identify the loops in your flow graph from (b)

```

for (i=2; i<=n; i++)
    a[i]=TRUE;
count=0;
d=sqrt(n);
for (i=2; i<=d; i++)
    if (a[i])
        count++;
        for (j=i*i; j<=n; j=j+i)
            a[j]=FALSE;
}
  
```

Soln:

a) 3-addr statements

B1

1) $i = 2$

B2

2) if $i > n$ goto (7)

B3

3) $t1 = i * 4$

4) $a[t1] = \text{TRUE}$

5) $i = i + 1$

6) goto (2)

34

7) count = 0

$$8) S = \text{Sqr}t(n)$$

g) $i = \mathbb{R}$

B5

10) if $i > 8$ goto(22)

B6

$$11) t_2 = i \neq y$$

12) if False a[tz] goto(zo)

37

13) $\text{count} = \text{count} + 1$

$$14) j = 2 * i$$

B8

(5) if $j > n$ goto (20)

B9

$$16) t_3 = i \approx 4$$

$$17) a[t_3] = \text{FALSE}$$

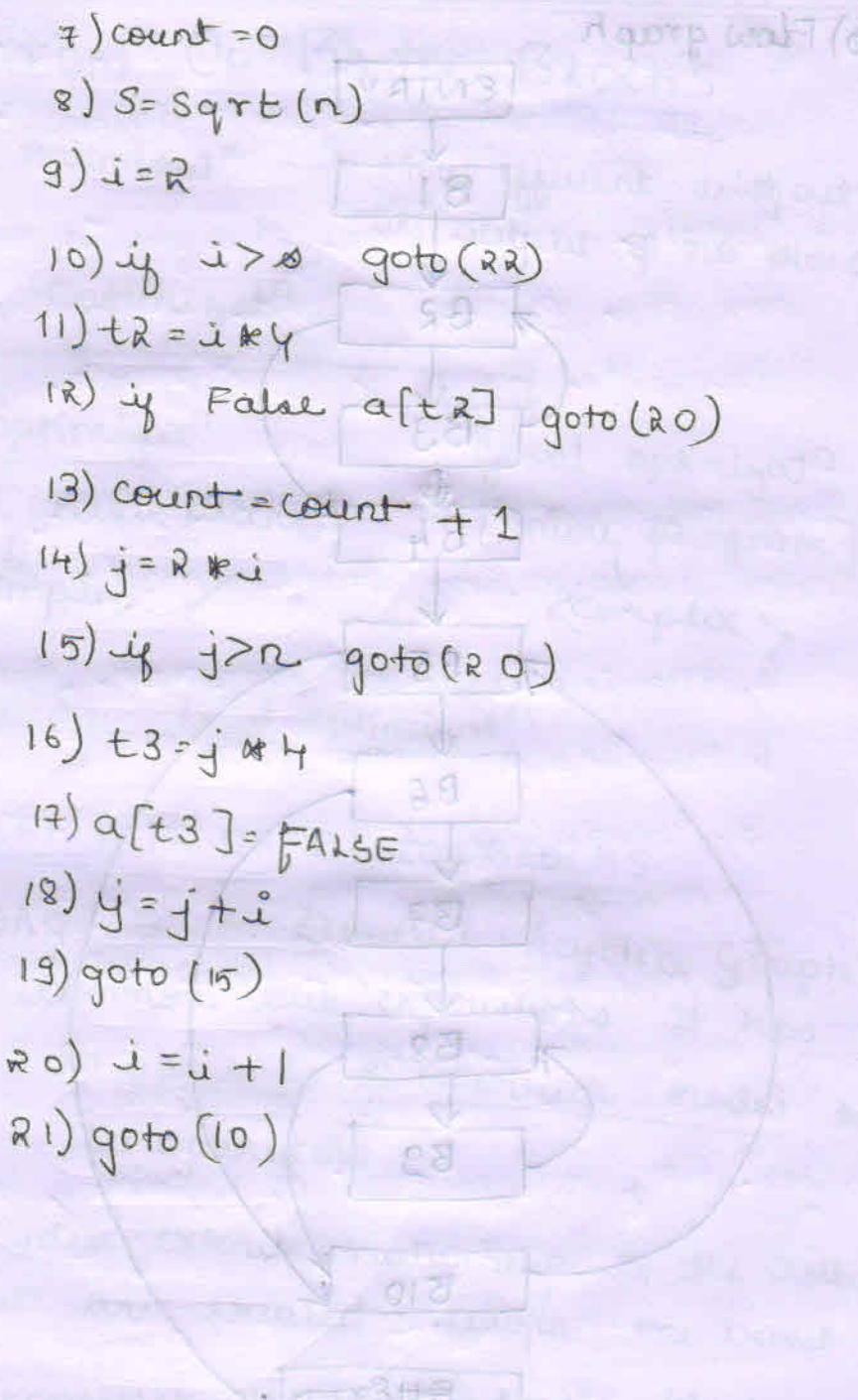
$$18) \quad y = f + i$$

19) goto (15)

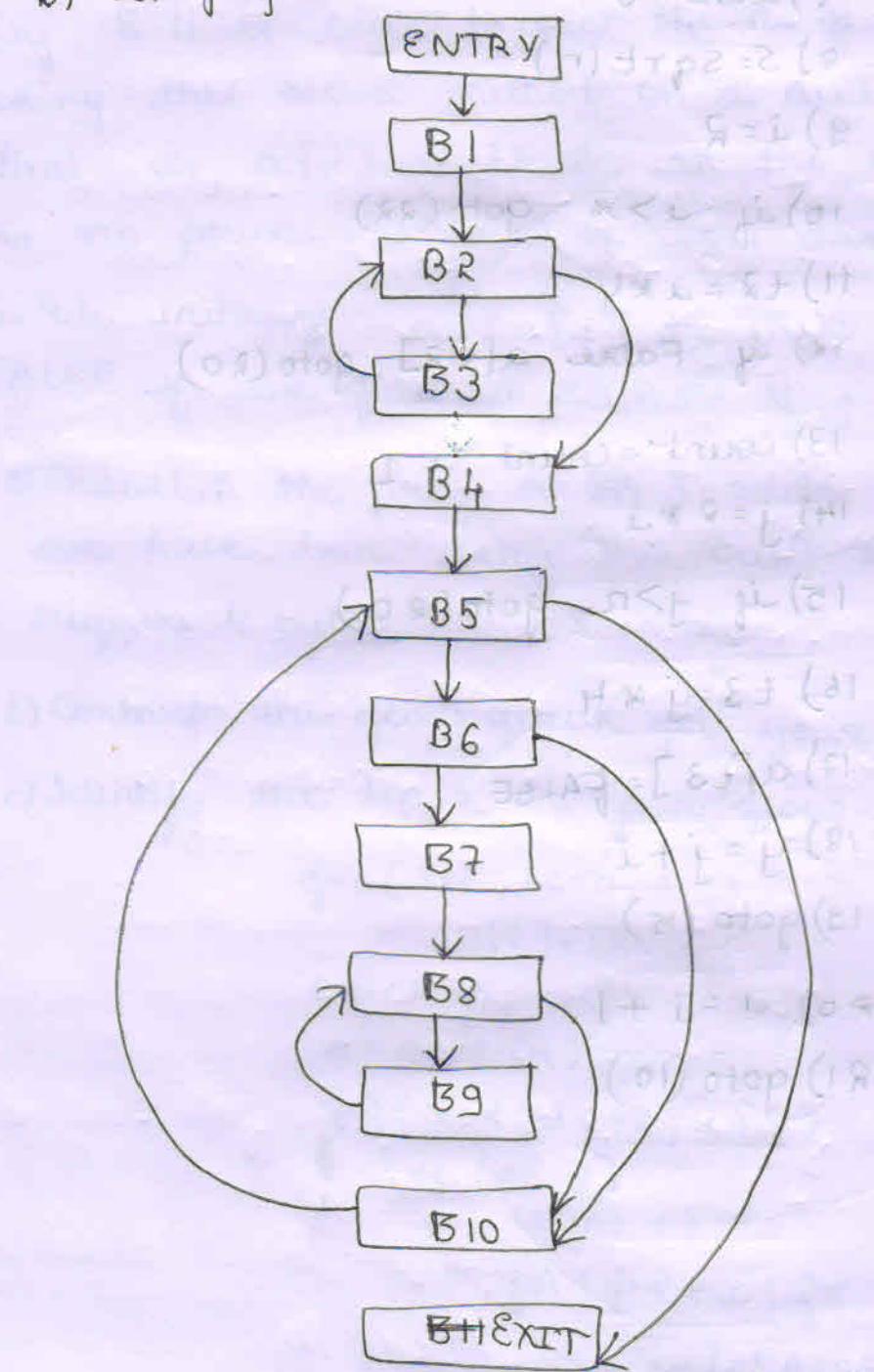
B10

π₀) i = i + 1

21) goto (10)



b) Flow graph



c) Loops

1. {B2, B3}
2. {B5, B6, B7, B8, B10}
3. {B5, B6, B10}
4. {B8, B9}

8.5. OPTIMIZATION OF BASIC BLOCKS

* local optimisatⁿ → More efficient without changing its output & no side effects

* Optimizatⁿ

local optimizatⁿ

[within basic blocks]

<Simple> ✓

Global optimizatⁿ

[entire program]

<Complex>

* we study local optimizatⁿ

DAG REPRESENTAT^N OF BASIC BLOCKS

DAG is a Directed Acyclic Graph used to represent common sub expressions. It has leaves corresponding to operands & interior nodes corresponding to operators.

* The idea extends naturally to the collectⁿ of expressions that are created within one basic block.

* we construct DAG for a basic block as follows:

(i) There is a node in DAG for each of the initial values of variables appearing in the basic block.

(ii) There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last & definatⁿ, prior to s, of the operands used by s.

(iii) Node N is labelled by the operator applied at s

ϕ also attached to N is the list of variables for which it is the last definition within the block.

4. Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block of flow graph.

* DAG representation of a basic block results in several code improving transformations such as:

1. Eliminates local common subexpressions.
2. Eliminates dead code
3. Reordering of statements that do not depend on one another.
4. Apply algebraic laws for grouping operands of 3-addr Instruct.

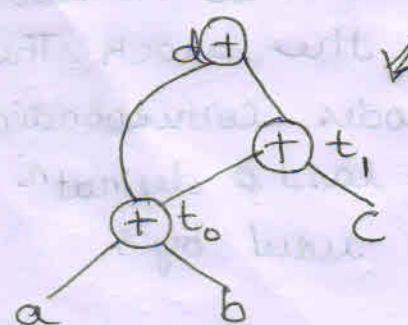
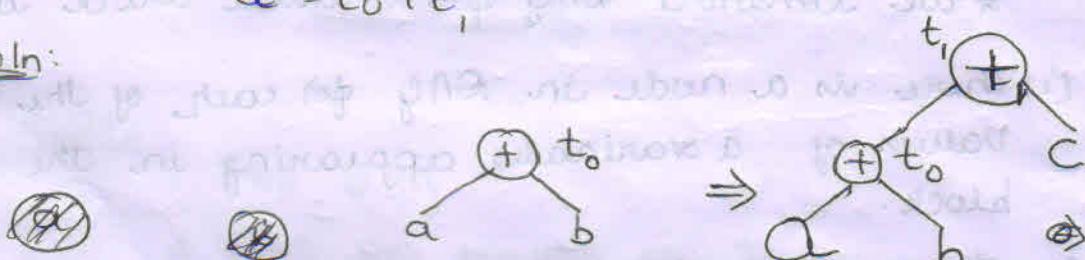
Ex: Give DAG representation of following basic block.

$$t_0 = a + b$$

$$t_1 = t_0 + c$$

$$d = t_0 + b$$

Soln:



b) FINDING LOCAL COMMON SUB EXPRESSIONS

Common subexpression present in 3 address
in a basic block evaluating to same value are
called common sub expression.

- * For eg; consider the following statements in a basic block:

1) $a = b + c$

2) $b = a - d$

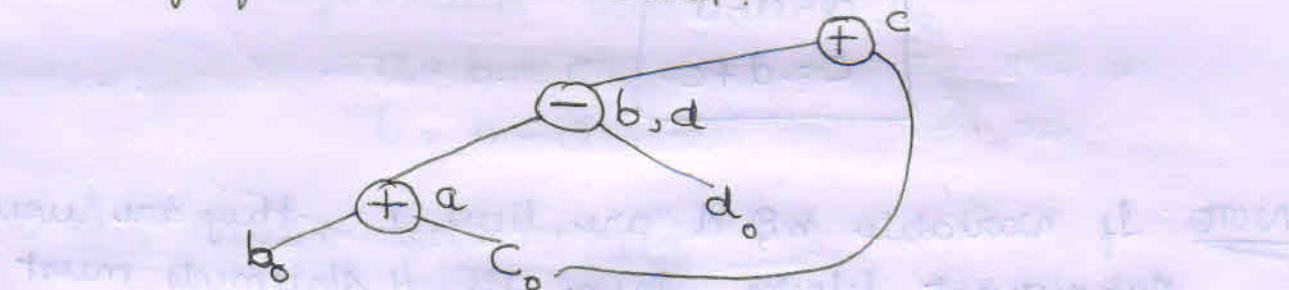
3) $c = b + c$

4) $d = a - d$

" $b + c$ " is not common sub expression.

" $a - d$ " is common sub expression.

- * AAG for the basic block:



- * Expression "a-d" present in statement ② & ④ are common sub expression ∵ both places will be evaluated to same time.

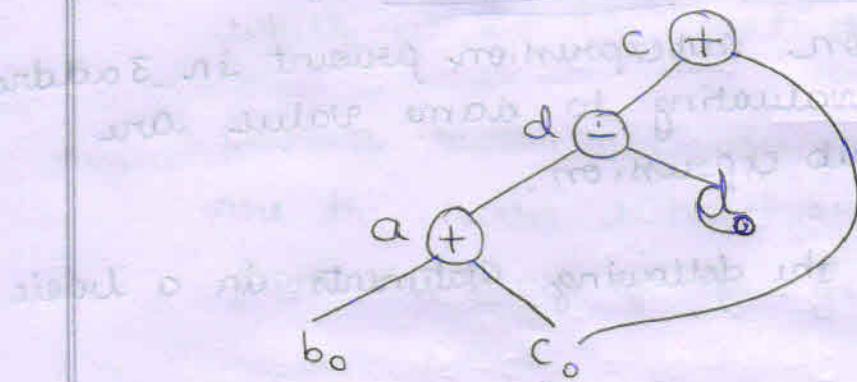
- * Expr "b+c" used in statement ① is different from statement ③ & hence not common sub expression.

- * Observe the following points:

→ If ~~the~~ b is not live on exit from the block,
then we do not need to compute that variable
∴ b can be deleted

→ Once ~~is~~ deleted, it can be used to receive the value resulting from symbol.

→ The resulting DAG after deleting b is shown



→ Since only 3 non leaf nodes in final DAG
⇒ Only 3 statements in basic block.

→ Basic block can be reconstructed as shown

$$\boxed{\begin{array}{l} a = b + c \\ d = a - d \\ c = d + c \end{array}}$$

NOTE: If variable b & d are live i.e., they are used in subsequent blocks then all 4 statements must be there.

c) DEAD CODE ELIMINATION:

* We delete from a DAG any node with no ancestors that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that corresponds to dead code.

* For example, consider following 3 address statements in basic block.

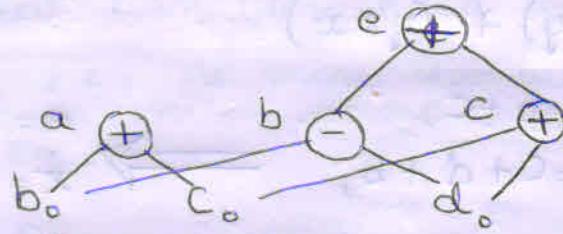
$$a = b + c$$

$$b = b - d$$

$$c = c + d$$

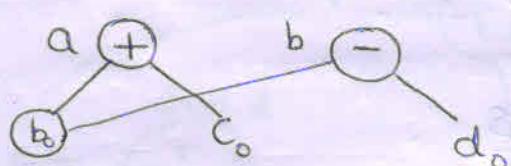
$$e = b + c$$

* DAG:



* In above basic block assume that the variables a & b are live & c & e are not live

Resulting DAG



- we can remove node e & c ∵ dead
- Regenerated code will be

$$a = b + c$$

$$b = b - d$$

d) ALGEBRAIC IDENTITIES

All the below expensive statements can be replaced by their equivalent cheaper ones.

* Algebraic properties

| <u>Expensive</u> | <u>Cheaper</u> |
|----------------------|----------------|
| (i) $x + 0 = 0 + x$ | = x |
| (ii) $x * 1 = 1 * x$ | = x |
| (iii) $x - 0$ | = x |
| (iv) $x / 1$ | = x |
| (v) x^2 | = $x * x$ |
| (vi) $2 * x$ | = $x + x$ |
| (vii) $x / 2$ | = $x * 0.5$ |

* Strength reduction

* Constant folding (viii) $2 * 3.14 = 6.28$

* Commutativity & Associativity

→ DAG construct can help us here

→ Apart from checking left op right, we could also check right op left for commutativity.

$$\text{Eg: } (x * y) + (y * x)$$

$$\begin{array}{l} \text{Eg: } a = b + c; \\ e = c + d + b; \end{array} \Rightarrow \begin{array}{l} a = b + c \\ t = c + d \\ e = t + b \end{array} \Rightarrow \begin{array}{l} a = b \\ e = t \\ e = b \end{array}$$

if t outside
 $a = b$

* Some algebraic laws are not obvious

$$\text{Eg: } a + (b - c) \neq (a + b) - c$$

∴ Careful about the parenthesis.

e) ARRAY REFERENCES

* Array references cannot be treated like usual variables.

* The proper way to represent array occurs in DAG in as follows:

1. An assignment from an array

$$[\text{Eg: } x = a[i]]$$

" is represented by creating a node with $=[]$ & 2 children representing initial of array, a & index i . Variable x becomes label of this node.

2. An assignment to an array

$$[\text{Eg: } a[j] = y]$$

" is represented by a new node with op. $[]=$ & 3 children representing a , j & y . There is no variable labeling this node. Also, creatⁿ of this node kills all currently constructed node whose value depends on

NOTE: A node that has been killed cannot receive any more label i.e., no more for becoming a common subexpression.

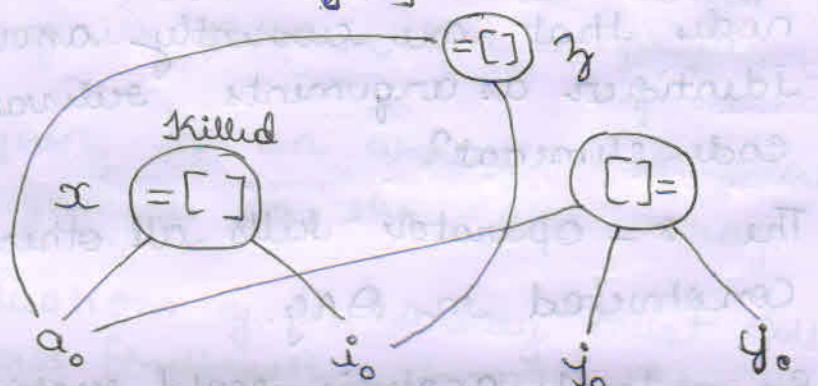
* Eg: Draw DAG for the block

$$x = a[i]$$

$$a[j] = y$$

$$y = a[i]$$

Soln:



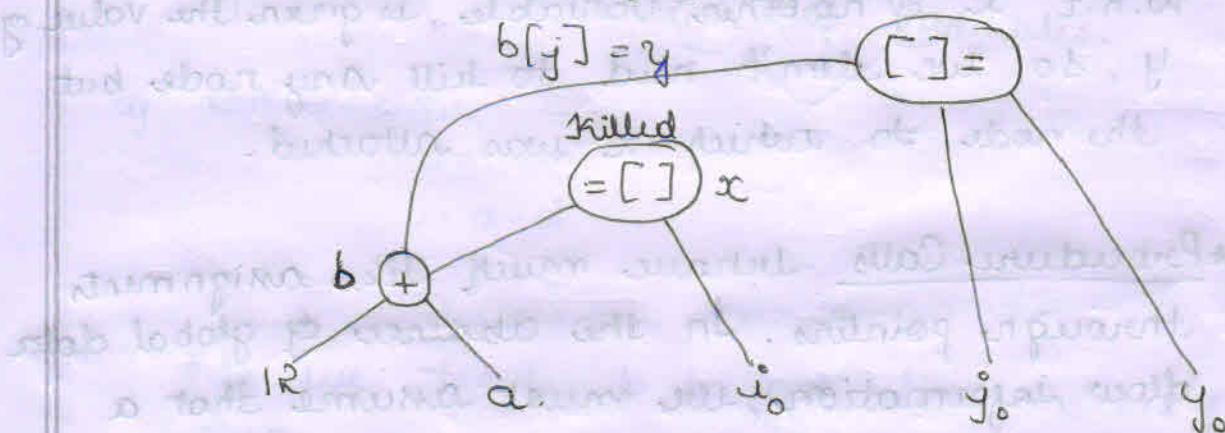
* Eg2: Draw DAG for the block

$$b = 12 + a$$

$$x = b[i]$$

$$b[j] = y$$

$$[] =$$



A node that kills a use of an array need not have that array as a child.

f) POINTER ASSIGNMENTS g) PROCEDURE CALLS

- * when we assign indirectly through a pointer say,

$$\begin{array}{l} \boxed{p = *p} \\ \boxed{*q = y} \end{array}$$

- * we do not know what p or q points to.
- * As a consequence, the operator $=*$ must take a nodes that are currently associated with identifier as arguments, relevant for dead code elimination?
- * The $=*$ operator kills all other nodes constructed in DAG.
- * Even local analysis could restrict the flow of a pointer. For eg:

$$\begin{array}{l} \boxed{p = \&x} \\ \boxed{*p = y} \end{array}$$

w.r.t x , if no other variable, is given the value y , so we don't need to kill any node but the node to which x was attached.

- * Procedure Calls behave much like assignments through pointers. In the absence of global data flow information, we must assume that a procedure uses $\&$ changes any data to which it has access.

- * Thus if procedure P is in the scope of variable x , a call to P both uses the node with attach variable x & kills that node.

g) Reassembling Basic Blocks from DAG's

After converting a basic block into a DAG & applying optimization we must reassemble the basic block.

* Rules:

1. The Order of instructions must respect the order of nodes in the DAG i.e., cannot compute node's value until value for each of its children are computed.
2. Assigning to an array must follow any previous assignments to the same array.
3. Evaluations of an array must follow any previous (indirect) assignments to the same array.
4. Any use of a variable must follow any previous indirect assignments through a pointer.
5. Any indirect assignment must follow all previous evaluation of variables.

Eg: * Consider

$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = a - d$$

* If we decide b is not live on exit from the block then,

$$a = b + c$$

$$d = a - d$$

$$c = d + c$$

* If both b & d are live on exit or not sure whether or not they are live on exit then need to compute b & d.

$$a = b + c$$

$$d = a - d$$

$$b = d$$

$$c = d + c$$

conclusion:

This basic block is more efficient than our original one.

(i) Less expensive since subtraction is replaced by assignment.

(ii) By global analysis, we can eliminate the use of computation of b outside the block by replacing it by uses of d .

EXERCISES [8.5]

1. Construct the DAG for the basic block

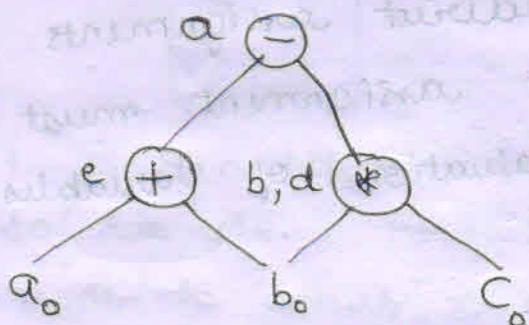
$$d = b * c$$

$$e = a + b$$

$$b = b * c$$

$$a = e - d$$

Soln:



2. Simplify the 3-address code in exercise 8.5.1, assuming

- Only a is live on exit from the block
- $a, b \& c$ are live on exit from the block
- Only a is live

$$e = a + b$$

$$d = b * c$$

$$a = e - d$$

b) $a, b \oplus c$ are live

$$e = a + b$$

$$b = b * c$$

$$a = e - b$$

3. Construct the DAG for the code in block B_6 of fig 8.9. Do not forget to include the comparison $i \leq 10$

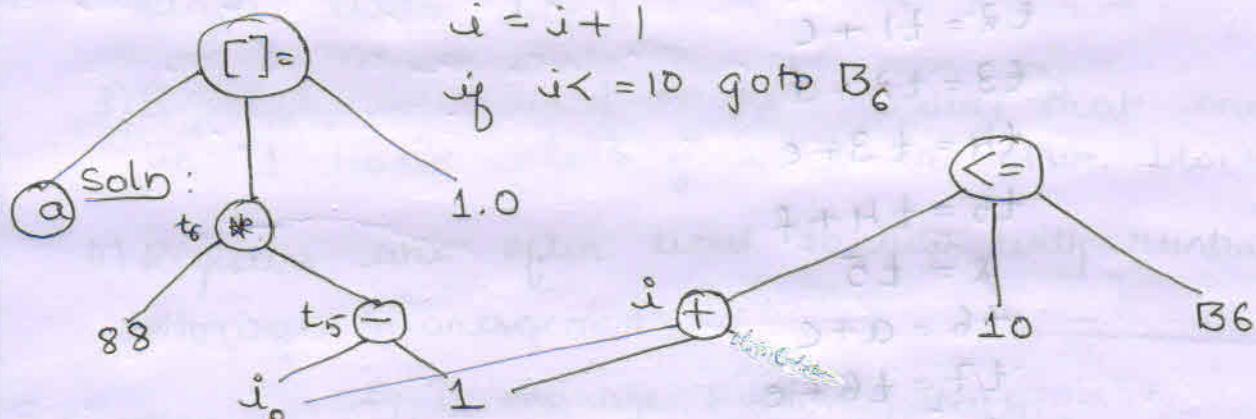
B_6 : $t_5 = i - 1$

$$t_6 = 88 * t_5$$

$$a[t_6] = 1.0$$

$$i = i + 1$$

if $i \leq 10$ goto B_6



4. Construct the DAG for the code in block B_3 of fig 8.9

$$t_1 = 10 * j$$

$$t_2 = t_1 + j$$

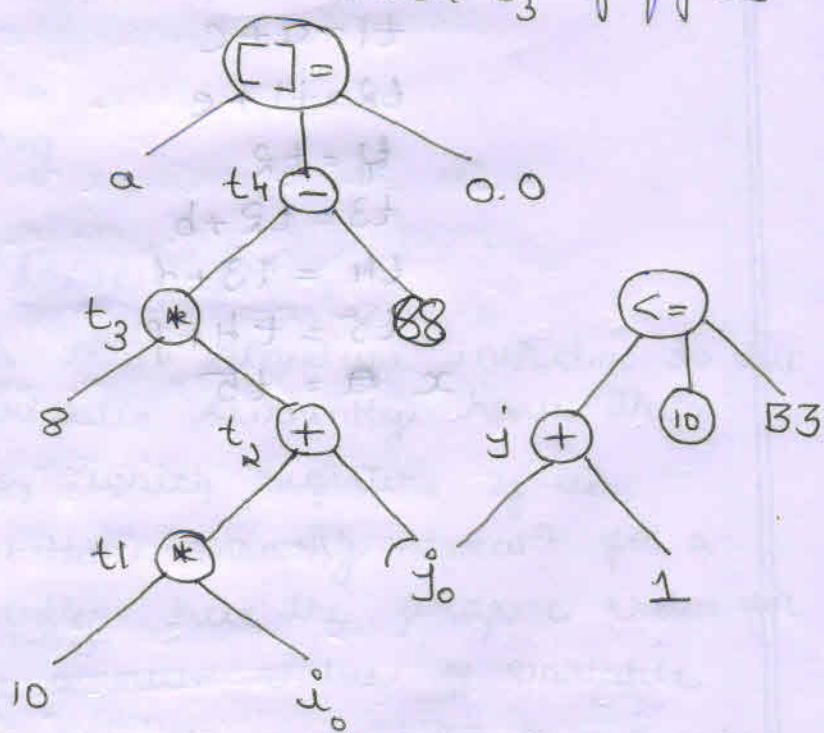
$$t_3 = 8 * t_2$$

$$t_4 = t_3 - 88$$

$$a[t_4] = 0.0$$

$$j = j + 1$$

if $j \leq 10$ goto B_3



b) $a, b \oplus c$ are live

$$e = a + b$$

$$b = b * c$$

$$a = e - b$$

3. Construct the DAG for the code in block B_6 of fig 8.9. Do not forget to include the comparison $i \leq 10$

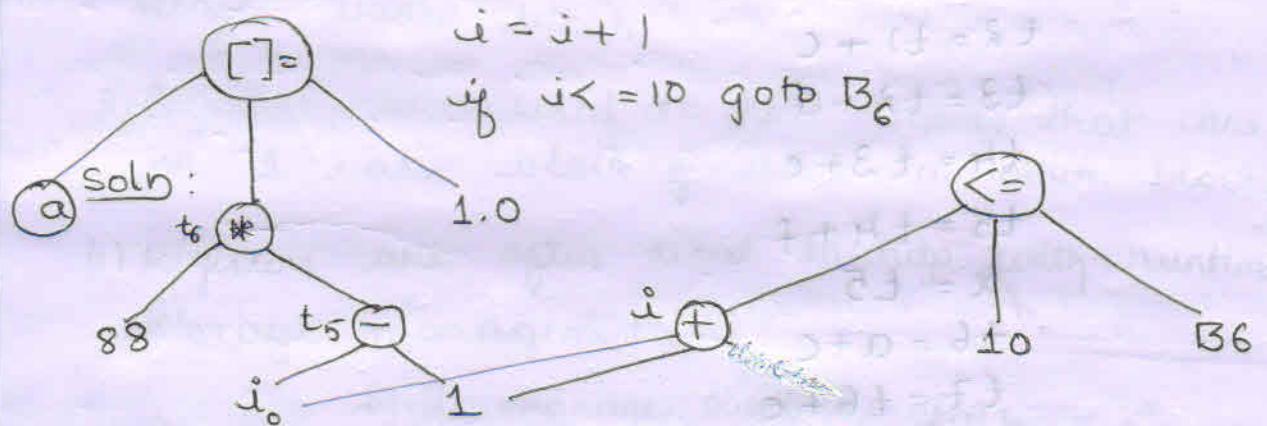
$$\underline{B_6}: \quad t_5 = i - 1$$

$$t_6 = 88 * t_5$$

$$a[t_6] = 1.0$$

$$i = i + 1$$

if $i \leq 10$ goto B_6



4. Construct the DAG for the code in block B_3 of fig 8.9

$$t_1 = 10 * i$$

$$t_2 = t_1 + j$$

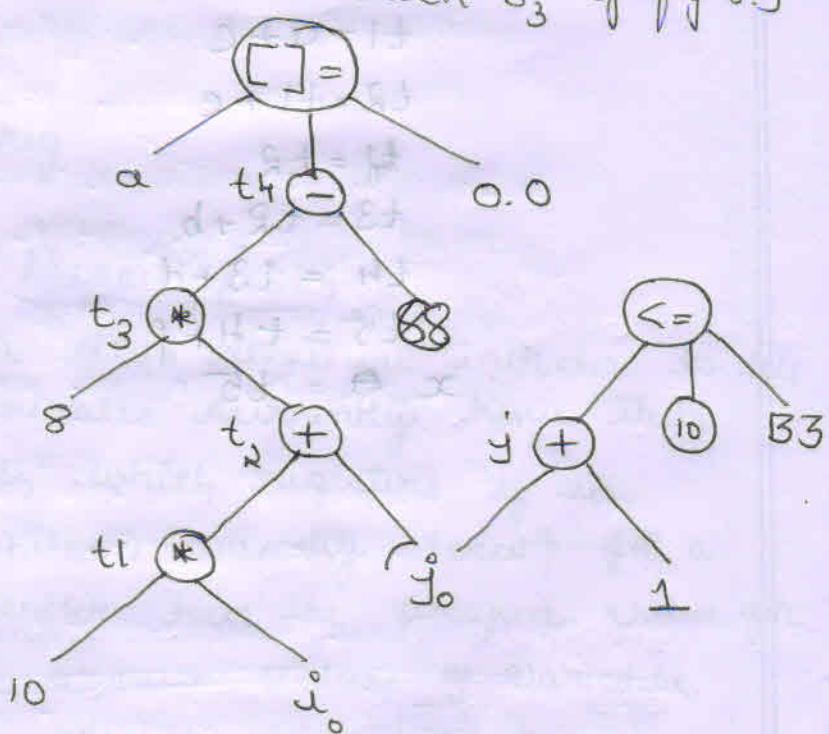
$$t_3 = 8 * t_2$$

$$t_4 = t_3 - 88$$

$$a[t_4] = 0.0$$

$$j = j + 1$$

if $j \leq 10$ goto B_3



5. Suppose a basic block is formed from the C assignment statements

$$x = a + b + c + d + e + f; \quad d + 0 = 3$$

$$y = a + c + e; \quad d - 3 = 0$$

- Give the 3 address statements for this block
- Use the associative & commutative laws to modify the block to use the fewest possible no of instruction, assume both x & y are live on exit from the block.

a) 3 address instⁿ

$$t1 = a + b$$

$$t2 = t1 + c$$

$$t3 = t2 + d$$

$$t4 = t3 + e$$

$$t5 = t4 + f$$

$$x = t5$$

$$t6 = a + c$$

$$t7 = t6 + e$$

$$y = t7$$

b) Optimized statements

$$t1 = a + c$$

$$t2 = t1 + e$$

$$t3 = t2$$

$$t4 = t3 + b$$

$$t5 = t4 + d$$

$$t5 = t4 + f$$

$$x = t5$$

8.6 A Simple Code Generator

* 4 principal uses of registers are:

- 1) In most machine architecture, some or all of the operands of an operatⁿ must be in registers in order to perform the operation.
- 2) Registers make good temporaries - places to hold the result of a subex expression while a larger expression is being evaluated, or generally a place to hold a variable i.e, used only within a single basic block.
- 3) Registers are used to hold values that are computed in 1 basic block & used in other blocks,
- 4) Registers are often used to help with runtime storage management.

Eg: To manage runtime stack,
maintenance of stack pointers, ...

* Machine instructions is of form:

- 1) LD reg, mem
- 2) ST mem, reg
- 3) OP reg, reg, reg

Register & Address Descriptions

We require a data structure in order to tell

- what program variables currently have their values in a register, which registers if so.
- we need to know whether memory locatⁿ for a given variable currently has the proper value for that variable, since a new value of variable may have been computed in a register & not yet stored.

* The desired datastructure has following descriptors.

1) Register descriptors keeps track of the variable names whose current value present in registers. Initially all register descriptors are empty. As the code generator progresses, each registers will hold the value of zero or more names.

2) Address descriptors keeps track of the locations where current value of that variable can be found. The location might be a register, a memory address, a stack location or some set of more than one of these. The information can be stored in the symbol table entry of that variable name.

Code Generation Algorithm

* Function getReg has access to the register & address descriptors for all the variables of the basic block, & may have access to certain useful data flow information such as variables that are live on exit from the block.

* In a 3 address instruction, eg: $x = y + z$, we treat $=$ as a generic (copy) operator & ADD as the equivalent m/c instruction. A possible improvement to the algorithm is to generate code for both $x = y + z$ & $x = z + y$ whenever + is commutative operator & pick the better code sequence.

* Machine instructions for operations

For a 3-address instn such as $x = y + z$ do the following:

1) Use * getReg ($x = y + z$) to select registers for x, y, z .

Call these R_x, R_y & R_z

2. If y is not in R_y , then issue an instructⁿ LD R_y, Y' where Y' is one of the memory locatⁿ for y .
3. Similarly if z is not in R_z , issue an instructⁿ LD R_z, Z' where Z' is a location for z .
4. Issue the instructⁿ ADD R_x, R_y, R_z .

* Machine instructⁿ for copy statements

For a 3 address copy statement of the form $x = y$

- 1) Assume getReg will always choose same register for both x & y
- 2) If y is not already in the Register R_y , then generate the machine instructⁿ LD R_y, Y .
- 3) If y was already in R_y , do nothing.
- 4) It is necessary to adjust the register descriptor for R_y so that it includes x as one of the values found there.

* Ending the basic block

- 1) If variable is a temporary, used only within a block & block ends, we can neglect temporary value & assume its register is empty.
- 2) If variable is live on exit from the block or its status not known in such cases for each variable x whose address descriptor does not say that its value is located in the memory locatⁿ say x , generate the instructⁿ ST x, R where $R \rightarrow$ register

$x \rightarrow$ Value exists @ the end of the block.

* Managing Register & Address description

As the code-generator algorithm issues load, store & other m/c instructⁿ, it needs to update the register & address description.

* The rules are as follows:

1) For instructⁿ LD R, x

- change register descriptor for R so it holds x
- change the addr descriptor for x by adding register R as an additional locatⁿ.

2) For instructⁿ ST x, R

change addr descriptor for x to include its own memory locatⁿ.

3) For operation such as ADD Rx, Ry, Rz implies $x = y + z$

a) change register descriptor for Rx so that it holds only x.

b) change the address descriptor of x so that its only locatⁿ is Rx.

c) Remove Rx from the address descriptor of any variable other than x.

4) When we process a copy statement $x = y$, after generating load for y into register Ry,

a) add x to the register descriptor for Ry.

b) change the address descriptor for x so that its only locatⁿ is Ry.

Eg 8.16 : Translate the basic block consisting of the three address statements

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

Soln:

Assume t, u, v are temp variables while a, b, c, d are global variables.

assume registers are enough, reuse the register whenever possible.

| $t = a - b$ | R1 | R2 | R3 | a | b | c | d | t | u | v |
|-------------|----|----|----|---|---|---|---|---|---|---|
| | | | | a | b | c | d | | | |

$\Rightarrow LD\ R1, a;$

$LD\ R2, b;$

$SUB\ R2, R1, RR$

| | R1 | R2 | R3 | a | b | c | d | t | u | v |
|--|----|----|----|-------|---|---|---|----|---|---|
| | a | t | | a, R1 | b | c | d | RR | | |

$U = a - c$

$\Rightarrow LD\ R3, C$

$SUB\ R1, R1, R3$

| | R1 | R2 | R3 | a | b | c | d | t | u | v |
|--|----|----|----|---|---|-------|---|--------|---|---|
| | U | t | C | a | b | c, R3 | d | R2, R1 | | |

$V = t + u$

$\Rightarrow ADD\ R3, RR, R1$

| | R1 | R2 | R3 | a | b | c | d | t | u | v |
|--|----|----|----|---|---|---|---|--------|----|---|
| | U | t | V | a | b | c | d | RR, R1 | R3 | |

$a = d$

$\Rightarrow LD\ RR, d$

| | R1 | R2 | R3 | a | b | c | d | t | u | v |
|--|----|------|----|----|---|---|-------|--------|----|---|
| | U | a, d | V | RR | b | c | d, RR | RR, R1 | R3 | |

$d = v + u$

$\Rightarrow ADD\ R1, R3, R1$

| | R1 | R2 | R3 | a | b | c | d | t | u | v |
|--|----|----|----|----|---|---|----|---|---|----|
| | d | a | v | RR | b | c | R1 | | | R3 |

exit

ST a, R2

ST d, R1

| R1 | R2 | R3 |
|----|----|----|
| d | a | v |

| a | b | c | d | t | u | v |
|-------|---|---|-------|---|---|----|
| a, R2 | b | c | d, R1 | | | R3 |

Design of the Function getReg

Consider picking Ry for y in $x = y + z$. The rules are as follows:

1) If y is currently in a register, pick a register already containing y as Ry.

2) If y is not in a register & there is empty one pick one such register as Ry.

3) Difficulty arises when y is not in a register & there is no register that is currently empty.

Then we need to pick one of the allowable registers & we need to make it safe for reuse.

Suppose R be a candidate register & v is one of the variables for the register descriptor for R.

We need to make sure that v's value is not really needed or value of v can be obtained by some other means.

Possibilities are:

1) We are OK, if address descriptor for v is somewhere besides R.

2) We are OK, if v is x

3) We are OK, if v is not used later

4) If we are not OK by one of the first 3 cases then perform Spill Operation i.e., generate the

store instructⁿ to place a copy of V in its own memory locatⁿ

ST V, R

NOTE: For select^r of the register R_x . The issues of ~~op_{0,1}~~ options are almost similar to y. The differences are:

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x .

This statement holds even if x is one of y or z , since our m/c instructⁿ allows 2 registers to be the same in one instructⁿ.

2. If y is not used after instruct^r I, if R_y holds only y after being loaded, if necessary then R_y can also be used as R_x . A similar option regarding z & R_z .

UNIT - 8

CODE GENERATION

Assignment questions & answers

Pg 1

1. Explain the different issues involved in the design of code generator.

- 1) Input to the code generator
- 2) The target program
- 3) Instruction selection
- 4) Register allocation
- 5) Evaluation order

↳ Input to the code generator :

* I/p to the code generator is the intermediate representation of the source program produced by the front end along with information in the symbol table i.e., used to determine the runtime address of the data objects denoted by the names in IR.

* IR has several choices.

- (a) 3 address representation: quadruples, triples, indirect triples.
- (b) Virtual machine representation: bytecodes or stack machine codes
- (c) Linear representation such as postfix notation
- (d) Graphical representation such as syntax trees or DAG's

* Assumptions made are:

- (i) Front end produces low-level IR i.e., values of names in it can be directly manipulated by the machine instruction.
- (ii) Syntactic & semantic errors have been already detected.

2) The Target program:

- * The o/p of code generator is target program.
- * The instruction set architecture of the target machine has a significant impact on the design of code generator.
- * Most common architectures are:
 - (a) CISC: It has few registers, has maximum of 2 operands & variety of addressing mode, variable length of instruction with side effects.
 - (b) RISC: It has many registers, has maximum of 3 operands & simple addressing modes, relatively simple instruction set architecture.
- * Output may take variety of forms.
 - (a) Absolute machine language [Executable Code]
 - (b) Relocatable machine language [object files for linker]
 - (c) Assembly language [facilitates debugging]
- a) Absolute machine language has advantage that it can be placed in a fixed locatⁿ in memory & immediately executed.
- b) Relocatable machine language program allows subprograms to be compiled separately.
- c) Producing Assembly language program as o/p makes the process of code generation somewhat easier.

3) Instruction Selection:

The code generator must map the IR program into a code sequence that can be executed by the target machine.

- * The complexity of performing this mapping is determined by the factors such as:

- (i) The level of the IR

- (ii) The nature of the instruction set architecture
 (iii) The desired quantity of the generated code.

(i) The level of the IR:

- > If the [IR is high level], use code template to translate each IR statements into sequence of machine instruction.
- > If the [IR is low level], use low level information to generate more efficient code sequence.

(ii) The nature of the instruction set architecture has a strong effect on difficulty of instruction select.

- > Uniformity & Completeness of the instruction set are imp factors.
- > If we do not care about the efficiency of the target program, instruction select is straight forward.

> For eg:

$$\begin{aligned} x = y + z &\Rightarrow \text{LD } R_0, y \\ &\quad \text{ADD } R_0, R_0, z \\ &\quad \text{ST } x, R_0 \end{aligned}$$

∴ produces redundant LD & ST

> Eg 2:

$$\begin{aligned} a = b + c \\ d = a + b &\Rightarrow \text{LD } R_0, b \\ &\quad \text{ADD } R_0, R_0, C \\ &\quad \boxed{\text{ST } a, R_0} \\ &\quad \boxed{\text{LD } R_0, a} \\ &\quad \text{ADD } R_0, R_0, C \\ &\quad \text{ST } d, R_0 \end{aligned}$$

REDUNDANT!

(iii) The quality of generated code is determined by its speed & size.

> For eg:

$$\begin{aligned} a = a + 1 &\Rightarrow \text{LD } R_0, a \\ &\quad \text{ADD } R_0, R_0, 1 \\ &\quad \text{ST } a, R_0 \end{aligned}$$

can be replaced by INC a

4) Register Allocation:

* Instructⁿ involving register operands are usually shorter & faster than those involving operands in memory.

* 2 subproblems:

(i) Register allocation: Select the set of variables that will reside in registers at each point in the program.

(ii) Register assignment: Select specific register that a variable will reside in.

* Complications imposed by the hardware architecture
Eg: Register pairs for multiplication & division.

* Multiplicatⁿ instructⁿ is of the form

$M \ x, y$

where $x \rightarrow$ Multiplicand, is the odd register of an even/odd register pair

$y \rightarrow$ Multiplier is a single register

product \rightarrow Occupies the entire even/odd register pair.

* Division instructⁿ is of the form

$D \ x, y$

where $x \rightarrow$ dividend, occupies even register

$y \rightarrow$ divisor, occupies odd/even register

quotient \rightarrow stored in odd register

remainder \rightarrow stored in even register

* Eg: 2 Two 3 address code sequences

$$t = a + b$$

$$t = t * c$$

$$t = t / d$$

$$t = a + b$$

$$t = t + c$$

$$t = t / d$$

Optimal m/c code sequences

| | |
|---------|------------|
| L R1,a | L R0,a |
| A R1,b | A R0,b |
| M R0,c | A R0,c |
| D R0,d | SRDA R0,3R |
| ST R1,t | θ R0,d |
| | ST R1,t |

5) Evaluational Order:

- * The order in which computatⁿ are performed can effect the efficiency of the target code.
- * When instruction are independent their evaluatⁿ order can be changed.
- * Some computatⁿ orders require fewer registers to hold intermediate result than others.
- * However picking a best order in the general case is a difficult NP-Complete problem.

* Eg: (Extra Info)

$$a+b-(c+d)*e \Rightarrow t_2 = c+d$$

$$t_3 = e * t_2$$

$$t_4 = t_1 - t_3$$

REORDER

$$t_2 = c+d$$

$$t_3 = e * t_2$$

$$t_1 = a+b$$

$$t_4 = t_1 - t_3$$

```

MOV R0,a
MDD R0,b
MOV t1,R0
MOV R1,c
ADD R1,d
MOV R0,e
MUL R0,R,
MOV R1,t1
SUB R1,R0
MOV t4,R1

```

```

MOV R0,C
ADD R0,d
MOV R1,e
MUL R1,R0
MOV R0,a
ADD R0,b
SUB R0,R1
MOV t4,R0

```

② What is a basic block? How optimization is done in basic blocks? Explain with an example.

A basic block is a maximal consecutive sequence of 3 address code [TAC] such that:

- Flow of control enters at the beginning of basic block.
- Flow of control leaves at the end of basic block.
- No possibility of halting or branching except at the last instruction in the block.

Example of basic block:

Consider source code, for i from 1 to 10 do
 for j from 1 to 10 do
 $a[i, j] = 0.0;$
 for i from 1 to 10 do
 $a[i, i] = 1.0;$

B1 : 1) $i = 1$

BR : 2) $j = 1$

B3 : 3) $t_1 = 10 * i$

4) $t_2 = t_1 + j$

5) $t_3 = t_2 * 8$

6) $t_4 = t_3 - 88$

7) $a[t_4] = 0.0$

8) $j = j + 1$

9) if $j \leq 10$ goto B3

B4 : 10) $i = i + 1$

11) if $i \leq 10$ goto B2

B5 : 12) $i = 1$

B6 : 13) $t_5 = i - 1$

14) $t_6 = 88 * t_5$

15) $a[t_6] = 1.0$

16) $i = i + 1$

17) if $i \leq 10$ goto B6

Optimization Of basic block

- * Optimizatⁿ results in more efficient code without changing its o/p & has no side effects.
- * Optimizatⁿ done within basic blocks is called local optimizatⁿ.

a) DAG representatⁿ of basic blocks:

DAG refers to Directed Acyclic Graph used to represent common sub expressions. It has leaves corresponding to operands & interior nodes corresponding to operators.

* The idea extends naturally to the collectⁿ of expression that are created within one basic block.

* We construct DAG for a basic block as follows:

- (i) There is a node in DAG for each of the initial values of a variables appearing in the basic block.
- (ii) There is a node N associated with each statement σ within the block. The children of N are those nodes corresponding to statements that are the last definition, prior to σ , of the operands used by σ .
- (iii) Node N is labelled by the operator applied at σ & also attached to N is the list of variables for which it is last definatⁿ within the block.
- (iv) Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block of flow graph.

* DAG representatⁿ of basic block results in several code improving transformatⁿ such as:

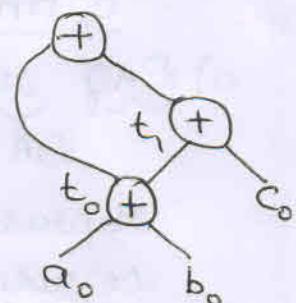
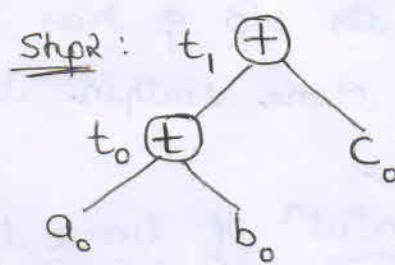
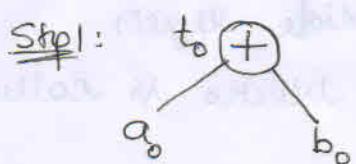
- 1) Eliminates local common subexpressions
- 2) Eliminates dead code
- 3) Reordering of statements that don't depend on one another
- 4) Apply algebraic laws for reordering operands of 3-addr instructⁿ.

* Eg: DAG representation for basic block: $t_0 = a + b$

$$t_1 = t_0 + c$$

$$d = t_0 + t_1$$

Step1: (Initial)



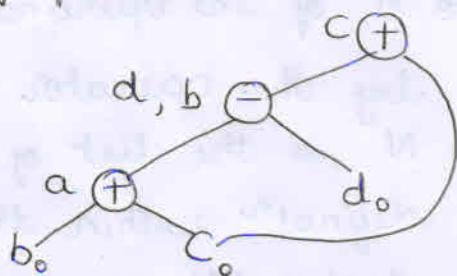
b) Finding local common subexpressions:

* Common subexpressions present in 3 address in a basic block evaluating to same value are called sub expression.

* For eg: Consider following statements in a basic block

- 1) $a = b + c$
 - 2) $b = a - d$
 - 3) $c = b + c$
 - 4) $d = a - d$
- " $b + c$ " is not common subexp.
" $a - d$ " is common subexpression

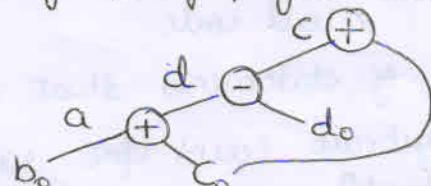
DAG for above basic block:



* If b is not live on exit from the block, then we don't need to compute that variable
 $\therefore b$ can be deleted

* Once b deleted, d can be used to receive the value resulting from the b symbol.

* The resulting DAG, after deleting b is shown below



- * Since, only 3 leaf nodes in modified DAG implies 3 statement in basic block.
- * Basic block can be rewritten as shown below:

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned}$$

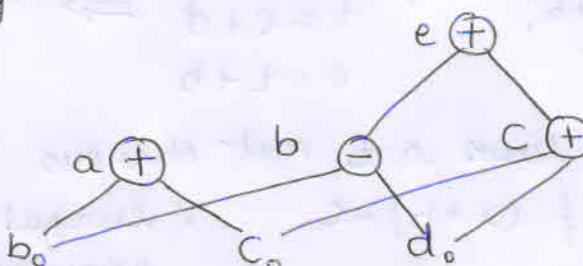
c) Dead code elimination:

* we delete from a DAG any node ~~(#)~~ with no ancestors that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that corresponds to dead code.

* For eg: Consider following 3 address statements in basic block

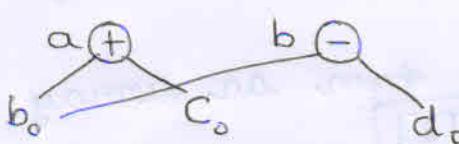
$$\begin{aligned} a &= b + c \\ b &= b - d \\ c &= c + d \\ e &= b + c \end{aligned}$$

* DAG



* In above basic block assume that the variables a & b are live & c & e are not live.

Resulting DAG



* we can remove node e & c ∵ dead

* Regenerated code will be

$$\begin{aligned} a &= b + c \\ b &= b - d \end{aligned}$$

d) Algebraic Identities:

All the below expensive statements can be replaced by their equivalent cheaper ones.

* Algebraic properties

Expensive - cheaper

$$(i) x + 0 = 0 + x = x$$

$$(ii) x * 1 = 1 * x = x$$

$$(iii) x - 0 = x$$

$$(iv) x / 1 = x$$

$$(v) x^R = x * x$$

$$(vi) R * x = x + x$$

$$(vii) x / 2 = x * 0.5$$

* Strength reduction

*

$$(viii) R * x = x + x$$

$$(ix) x / 2 = x * 0.5$$

* Constant folding

$$(x) R \approx 3.14 = 6.28$$

* Commutativity & Associativity

→ DAG construct can help us here

→ Apart from checking [left op right], we could also check [right op left] for commutativity.

$$\text{Eq: } (x * y) + (y * x)$$

$$\begin{array}{l} \text{Ex: } a = b + c; \\ \quad e = c + d + b; \end{array} \Rightarrow \begin{array}{l} a = b + c \\ t = c + d \\ e = t + b \end{array} \Rightarrow \begin{array}{l} \text{"If t not needed} \\ \text{outside the block."} \\ a = b + c \\ e = a + d \end{array}$$

* Some algebraic laws are not obvious

$$\text{Eq: } a + (b - c) \neq (a + b) - c \quad ; \text{ careful about the parenthesis.}$$

c) Array References

- * Array references cannot be treated like casual variables.
- * The proper way to represent array access in DAG is as follows.

1. An assignment from an array

$$\text{Eq: } x = a[i]$$

is represented by creating a node with operator $=[]$ & children representing initial value of array, a_0 & index i . Variable x becomes label of this node.

2. An assignment to an array

$$\text{Eq: } a[j] = y$$

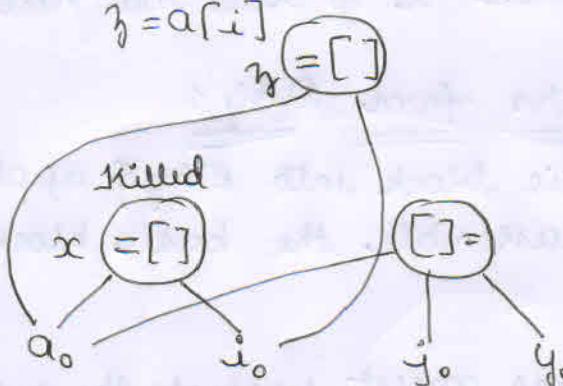
is represented by a new node with operator " $[] =$ " of 3 children representing $a_0, j_0 \& y_0$. There is no variable labeling this node. Also creation of this node kills all currently constructed node whose value depends on a_0 .

* Eg: DAG for the block

$$(i) x = a[i]$$

$$a[j] = y$$

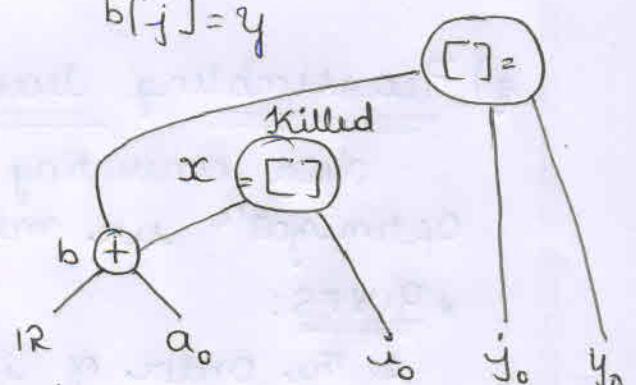
$$y = a[i]$$



$$(ii) b = IR + a$$

$$x = b[i]$$

$$b[j] = y$$



i. A node that kills a use of an array, needn't have that array as a child.

f) Pointer assignments & procedure calls

* When we assign indirectly through a pointer. say,

$$\boxed{x = *p}$$

$$\boxed{*q = y}$$

* We don't know what p or q points to.

* As a consequence, the operator $=*$ must take all nodes that are currently associated with identifier as arguments, relevant for dead code elimination

* The $=*$ operator kills all other nodes so far constructed in DAG.

* Even local analysis could restrict the scope of a pointer

Eg: $\boxed{p = \&x}$
 $\boxed{*p = y}$

w.r.t x & no other variable is given the value of y, so we don't need to kill any node but the node to which x was attached.

- * Procedure calls behave much like assignments through pointers. In the absence of global data flow info, we must assume that a procedure uses & changes any data to which it has access.
- * Thus if procedure P is in the scope of variable x, a call to P both uses the node with attached variable $x \in \emptyset$ kills that node.

g) Reassembling basic blocks from DAG's

After converting a basic block into DAG & applying optimization we must reassemble the basic block.

* RULES:

1. The order of instructions must respect the order of nodes in the DAG.
2. Assigning to an array must follow any previous assignments to the same array.
3. Evaluation of an array must follow any previous assignments to the same array.
4. Any use of a variable must follow any previous indirect assignments through a pointer.
5. Any indirect assignment must follow all previous evaluation of variables.

Eg: Consider $a = b + c$
 $b = a - d$
 $c = b + c$
 $d = a - d$

* If we decide b is not live on exit from the block then

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned}$$

* If both b & d are live on exit & not sure whether or not they are live on exit then need to compute b & d

$$\begin{aligned}a &= b+c \\d &= a-d \\b &= d \\c &= d+c\end{aligned}$$

∴ This basic block is more efficient than original, because it is less expensive since subtract is replaced by assignment.

③ For the following program (exp0) segment

```
for i=1 to 10 do
    for j=1 to 10 do
        a[i, j]=0.0
    for i=1 to 10 do
        a[i, i]=1.0
```

generate intermediate code & identify basic block & draw the flow graph.

Intermediate code:

| | |
|-----------|------------------------------|
| <u>B1</u> | 1) $i = 1$ |
| <u>B2</u> | 2) $j = 1$ |
| <u>B3</u> | 3) $t1 = 10 * i$ |
| | 4) $t2 = t1 + j$ |
| | 5) $t3 = 8 * t2$ |
| | 6) $t4 = t3 - 88$ |
| | 7) $a[t4] = 0.0$ |
| | 8) $j = j + 1$ |
| | 9) if $j \leq 10$ goto (3) |
| <u>B4</u> | 10) $i = i + 1$ |
| | 11) if $i \leq 10$ goto (2) |
| <u>B5</u> | 12) $i = 1$ |
| <u>B6</u> | 13) $t5 = i - 1$ |
| | 14) $t6 = 88 * t5$ |
| | 15) $a[t6] = 1.0$ |
| | 16) $i = i + 1$ |
| | 17) if $i \leq 10$ goto (13) |

Conclusion : Basic block 1 → statement 1
 Basic block 2 → statement 2
 Basic block 3 → statement 3 through 9
 Basic block 4 → statement 10 & 11
 Basic block 5 → statement 12
 Basic block 6 → statement 13 through 17

Flow Graph :

