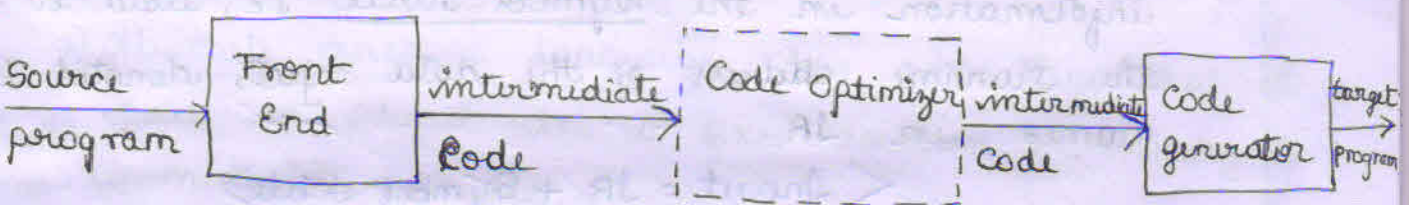
CONTENTS :

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 a appropriate target machine code 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 represen of the source program produced by the front end along w information in the symbol table i.e., used to determi 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 tripl

(b) Virtual machine representat^n : byte codes & stack mach codes

(c) Linear representat^n such as postfix notat^n

(d) Graphical representat^n such as syntax trees & 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 instruct^n.

(ii) Syntatic & semantic errors have been already detected

2) The Target Program :

* The Output of code generator is target program.
* The instⁿ set architecture of the target machine has a significant impact on the design of code generator.
* Most common architecture are:

(a) CISC : It has few registers, has maximum of 2 operands & variety of addressing mode, variable length instructions & instructⁿ with side effects.

(b) RISC : It has many registers, has maximum of 3 operands with simple addressing modes, & relatively simple instructⁿ 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 generatⁿ 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 optimizat^n.

> 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^n select^n.

> Uniformity & completeness of the instruct^n set are imp factors.

> If we do not care about the efficiency of the target program, instruct^n select^n is straightforward.

> For eg:

```
x = y + z  ⇒  LD   R0, y
              ADD  R0, R0, z
              ST   x, R0
```

∴ produces redundant LD & store

eg2:

```
a = b + c  ⇒  LD   R0, b
d = a + b     ADD  R0, R0, c
              ST   a, R0      ──→ REDUNDANT
              LD   R0, a
              ADD  R0, R0, c
              ST   d, R0
```

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

> For eg:

```
a = a + 1  ⇒  LD  R0, a
              ADD R0, R0, #1
              ST  a, R0
```

replaced by → INC a

4) Register Allocation:

* Instruct^n 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 multiplicat^n & division.
* Multiplicat^n inst^n is of the form

  M x, y

  where x → Multiplicand, is the odd register of an even/odd register pair.
  y → Multiplier, is ~~the~~ a single register
  ⇒ Product → occupies the entire even/odd register pair.
* Division inst^n is of the form

  D x, y

  where x → dividend, occupies even register
  y → divisor, occupies odd/even register
  ⇒ quotient → stored in ~~even~~ odd register
  remainder → stored in even register

Eg: two & 3 address code sequences

t = a + b	t = a + b
t = t * c	t = t + c
t = t / d	t = t / d

Optimal machine-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, 32
ST R1, t	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 instruct^{n} are independent their evaluation order can be changed.
* Some computat^{n} 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

a+b-(c+d)*e ⇒ t1 = a+b, t2 = c+d, t3 = e*t2, t4 = t1-t3

Reorder ↓

t2 = c+d
t3 = e*t2
t1 = a+b
t4 = t1-t3

⇒

```
MOV R0, a
ADD R0, b
MOV t1, R0
MOV R1, c
ADD R1, d
MOV R0, e
MUL R0, R1
MOV t1, 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
```

THE

8.2 THE TARGET LANGUAGE:

For designing a good code generator, we need to have familiarity with target machine & its instruct^n 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 instruct^n format & instruct^n 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 addrss instruct^n can have 2 operands or 1 operands 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 addrss

(iii) It has n no of registers denoted by $R_0, R_1, R_2, \ldots, R_{n-1}$

* Various types of instruct^n that are used by target m/c

(i) Load Instruct^n
(ii) Store Instruct^n
(iii) Computational Instruct^n
(iv) Unconditional Instruct^n
(v) Conditional Instruct^n

(i) Load Instruct^n: Used to copy the data into distinat^n operand which must be a register.

SYNTAX: LD dst, addr

where addr operand → register or memory locat

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

SYNTAX: ST dst, sr

where dst → destination of it is a memo location

sr → register.

Computational operation.

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

SYNTAX: OP dst, SRC1, SRC2.

where 1st operand, dst → destination

2nd & 3rd operand → Operands where R values fetched for operatⁿ to be p

Eg1: ADD R0, R1, R2 // R0 = R1 + R2

Eg2: SUB R0, R0, R1 // R0 = R0 − R1

Eg3: MUL R2, R0, R1 // R2 = R0 * R1

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

SYNTAX: BR label

where BR → BRanch instructⁿ

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

SYNTAX: Bcond sr, label

where B stands from Branch,

Cond can be LT (less than), GT (Greater than), LTE (less than or equal), GTE (Greater than or equal)

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

Eg1: BZ R0, T1 // Branch to T1, if R0 contains -ve value

Eg2: BLTZ R1, TR // Branch to TR, if R1 contains either 0 or -ve value

* Different addressing modes supported by generalized target machine:

1) Direct addressing mode
2) Indexed —11— _____
3) Integer Indexed —11— _____
4) Indirect —11— _____
5) Immediate —11— _____

(i) Direct A/M:

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

Eg: ~~LDP~~ LD R1, x // Load value stored in memory locat^n x into R1

(ii) Indexed A/M: The data can be accessed from a memory locat^n 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 R1, a(R2) // Access the data stored in R1 = Contents (a + contents (R2))

(iii) Indexed A/M where memory locat^n is integer

It is same as previous one except that a memory locat^n is identified as integer.

Eg: LD R1, 100(R2) // R1 = contents (100 + contents (R2))

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

```
LD   R1, *(R2)        // R2 contains memory locat the data stored in that memory locatn 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 ← 100
```

EXERCISE :

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

```
LD   R1, y            // R1 = y
LD   R2, z            // R2 = z
ADD  R1, R1, R2       // R1 = R1 + R2
ST   x, R1            // x = R1
```

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

```
LD   R1, p            // R1 ⇐ p
LD   R2, 0(R1)        // R2 = contents (0 + contn
ST   x, R2            // x = R2
```

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

```
LD   R1, p        // R1 = p
LD   R2, y        // R2 = y
ST   0(R1), R2    // contents(0 + contents(R1)) = R2.
```

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   R2, a[R1]    // R2 = contents(a + contents(R1))
ST   b, R2        // b = R2
```

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

```
LD   R1, j        // R1 = j
LD   R2, C.       // R2 = c
MUL  R1, R1, 8    // R1 = R1 * c
ST   a[R1], R2    // contents(a + contents(R1)) = R2
```

6. Generate m/c code for 3 address statement

if x < y goto L

```
LD   R1, x        // R1 = x
LD   R2, y        // R2 = y
SUB  R1, R1, R2.  // R1 = R1 - R2
BLTZ 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 location or constant have additional cost of 1.

* For example:

a) LD R0, R1 ⇒ cost = 1

b) LD R0, M ⇒ cost = 2.

c) LD R1, *100(R2) ⇒ cost = 3

* Cost of Addressing mode:

	Mode	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 instruct^{n} sequence

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

Cost = 1 + cost(AM)

Cost = 1 + 1 = 2

Cost = 1 + 1 = 2

Cost = 1 + 0 = 1

Cost = 1 + 1 = 2

Total Cost = 7

2.

```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

	Cost = Cost (AM) + 1.
LD R0, i	Cost = 1 + 1 = 2
MUL R0, R0, 8	Cost = 1 + 1 = 2
LD R1, a(R0)	Cost = 1 + 1 = 2.
ST b, R1	Cost = 1 + 1 = 2
	Total cost = 8

3.

```
LD  R0, C
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

	Cost = cost(A.M) + 1
LD R0, C	1 + 1 = 2
LD R1, i	1 + 1 = 2
MUL R1, R1, 8	1 + 1 = 2
ST a(R1), R0	1 + 1 = 2
	Total cost = 8

4. LD R0, P
LD R1, 0(R0)
ST x, R1

	Cost = Cost(A.M) + 1
LD R0, P	1 + 1 = 2
LD R1, 0(R0)	1 + 1 = 2. // ∵ of constar
ST x, R1	1 + 1 = 2
Total cost = 6	

5. LD R0, P
LD R1, x
ST 0(R0), R1

	Cost = Cost(A.M) + 1
LD R0, P	1 + 1 = 2
LD R1, x	1 + 1 = 2
ST 0(R1), R1	1 + 1 = 2
Total cost = 6	

6. LD R0, x
LD R1, y
SUB R0, R0, R1
BLTZ *R3, R0

	Cost = 1 + cost(A.M)
LD R0, x	1 + 1 = 2
LD R1, y	1 + 1 = 2
SUB R0, R0, R1	1 + 0 = 1
BLTZ *R3, R0	1 + 1 = 2 <∵ indirect A
Total cost = 7	

8.3 Address In The Taryet Code

* We show how names in the IR can be converted into addrus in the target code by looking at code generat^{n} for simple procedure calls & returns using static & stack allocation.

* Each executing programs runs in its own logical addrus space that was partitioned into 4 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 execut^{n}.

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

* 2 standard storage allocat^{n} strategies namely:
  (a) Static allocation
  (b) Stack allocation.

* In Static allocation, the position of an activat^{n} record in memory is fixed at compile time.

* In Stack allocat^{n}, a new activat^{n} record is pushed onto stack for each execut^{n} of the procedure. The record is popped when the activat^{n} ends.

(a) STATIC ALLOCATION

* The 3 address code for funct^n call consists following statements
  1. call
  2. return
  3. halt
  4. action

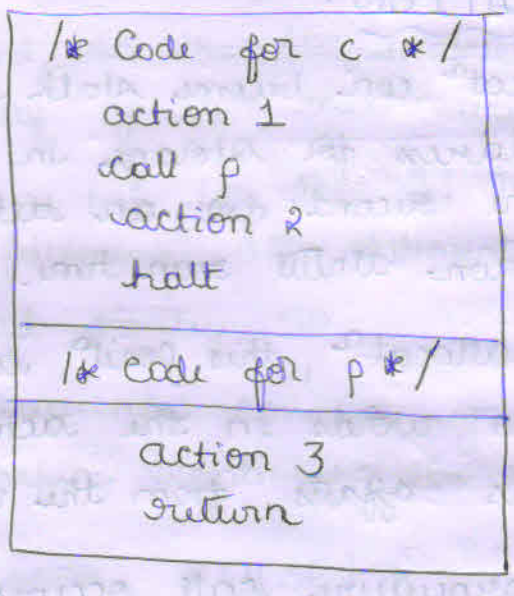
* Call statement used for funct^n call, it has to mail the control to the funct^n along with save the status of current funct^n.
* Return statement to give control back to called funct^n.
* Action defines other operat^n or instruct^n for assign or flow control statements.
* Halt indicates the complet^n of operat^n of calle funct^n.
* Let us consider code needed to implement the simplest static allocat^n case:

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

```
ST   callee.staticArea, #here + 20
BR   callee.codeArea
```

→ ST instruct^n saves return address
→ BR transfers control to the target code for called procedure.
→ callee.staticArea, callee.codeArea are con referring to the address of activation reco & first instruct^n for the called procedure.
→ #here +20 in the ST instruct^n is return addr
→ Cost of 2 instruct^n is 5.

3 address code.



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

```
100 : ACTION 1                    /* Code for c */
120 : ST 364, #140                //Code for action 1
                                  // Store return addrss
                                  140 in locatn 364
132 : BR 200                      // call p
140 : ACTION2
160 : HALT                        //return to O.S
      .......
200 : ACTION3
220 : BR *364                     // return to address
                                  saved in 364
      .......
300 :                             // 300-363 hold activatn record
304 :                             for c
                                  // return address
                                  // local data for c
      .......
364 :                             //364-451 holds activatn
                                  // record for p
368 :                             // return address
                                  // local data for p
```

(b) STACK ALLOCATION:

* Static allocat^n can become stack allocat^n by using relative address for storage in activat^n records. The posit^n of the record for an activat^n of a procedure is not known until run time.

* In stack allocat^n, this posit^n is usually stored in a register, so words in the activat^n records can be accessed as offsets from the value in this regist

* When a procedure call occurs, the calling procedure increments SP & transfers control to the ca procedure.

* After control returns to the caller, it decrements SP, thereby deallocating the activat^n records of th called procedure.

* The code for the 1st procedure initializes the stac by setting SP to the start of the stack area in the memory.

```
LD   SP, #Stack start        // Initialize the stac
Code for the first procedure
HALT                         // terminate execut$^n$
```

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

```
ADD  SP, SP, #caller. record Size   // increment
                                    stack poi
ST   0(SP), #here +16               // save retu
                                    address.
BR   Callee. code Area              // jump to th
                                    Callee
```

* The attribute caller.recordsize represents size of an activatn record, so the ADD instructn leaves SP pointing to the begining of the next activatn record.

* The source #here +16 in the ST instructn is the address of the instn following BR statement, it is saved in the address pointed to by SP.

* The return sequence consists of 2 parts. The called procedure transfer control to the return address using

BR ~~GOTO~~ *0(SP) // return to caller

→ The reason for using *0(SP) in the BR instructn is that we need 2 levels of indirectn : 0(SP) is the address of the first word in the activatn record & *0(SP) is the return addr saved there.

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

SUB SP, SP, #caller.recordSize
// decrement 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 activatn of q can be alive at the same time

```
// code for m
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 #
```

```
100: LD   SP, #600      // code for sm
                        // Initialize the stack.
108: ACTION 1           // code for action 1
128: ADD  SP, SP, #msize // call sequence begins
136: ST   0(SP), #152   // push return address
144: BR   300           // call q
152: SUB  SP, SP, #msize // restore SP
160: ACTION 2
180: HALT
     . . . . . .
200: ACTION 3           // code for p
220: BR *0(SP)          // return
     . . . . . .
300: ACTION             // code for q
                        // contains a conditional
                        to 456
320: ADD  SP, SP, #qsize
328: ST   0(SP), #344   // push return address
```

```
336: BR    200          // call p
344: SUB   SP, SP, #q size
352: ACTION 5
372: ADD   SP, SP, #q size
380: ST    0(SP), #396  // push return addrss
388: BR    300          // call q
396: SUB   SP, SP, #q size
404: ACTION 6
424: ADD   SP, SP, #q size
432: ST    0(SP), #440  // push return addr
440: BR    300          // call q
448: SUB   SP, SP, #q size
456: BR    *0(SP)       // return
600:                    // stack starts here
```

RUN TIME ADDRESS FOR NAMES

* Assumptn: A name in a 3-addrss 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:
  - → x = 0
  - → Suppose the symbol table entry for x contains a relative address 12
  - → x is in statically allocated area begining at address static
  - → the actual runtime address of x is static + 12
  - → The actual assignment : Static [12] = 0
  - → For a static area starting at address

100: LD 112, #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 ~~flow~~ follow other blocks, representing jumps in the instru
* Useful for discussing code generation

BASIC BLOCKS

* Definition:

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

- → Flow of control enters at the begining of ba 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:

Algorithm: Partitioning 3 address instructns into basic blocks.

Input: A sequence of 3-address instructions.

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

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

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

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

Eg: 8.7

```
1) i = 1
2) j = 1
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 <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
```

Solution

leader
leader
leader.

leader

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 leader

14) t6 = 88 * t5 leader

15) a[t6] = 1.0

16) i = i + 1

17) if i <= 10 goto (13)

Conclusion:

* Instruct^n 1, 2, 3, 10, 12 & 13 are leaders.
* Since basic block of each leader contains all instruct^n 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 use 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 comp 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 & next use information

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

Input : A basic block B of three-address statement. 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 & next use information of x, y & z.

Method : Start at the last statement in B & scan backwards to the begining 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 & 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 & z to i.

NOTE:
If the 3-adder statement i is of the form
"x = +y" or "x = y"
the steps is same as above ignoring z. This algo stores the informatn whether the code is live or dead & whether the variable has next use info or not in the symbol table.

FLOW GRAPHS

*Definatn: A flow graph is a graphical dipiction of sequence of instructn with control flow edges, which show how the control flow from 1 basicblock to other basic block.

* → A node represents basic block
An edge represents control flow from 1 Basic block to

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

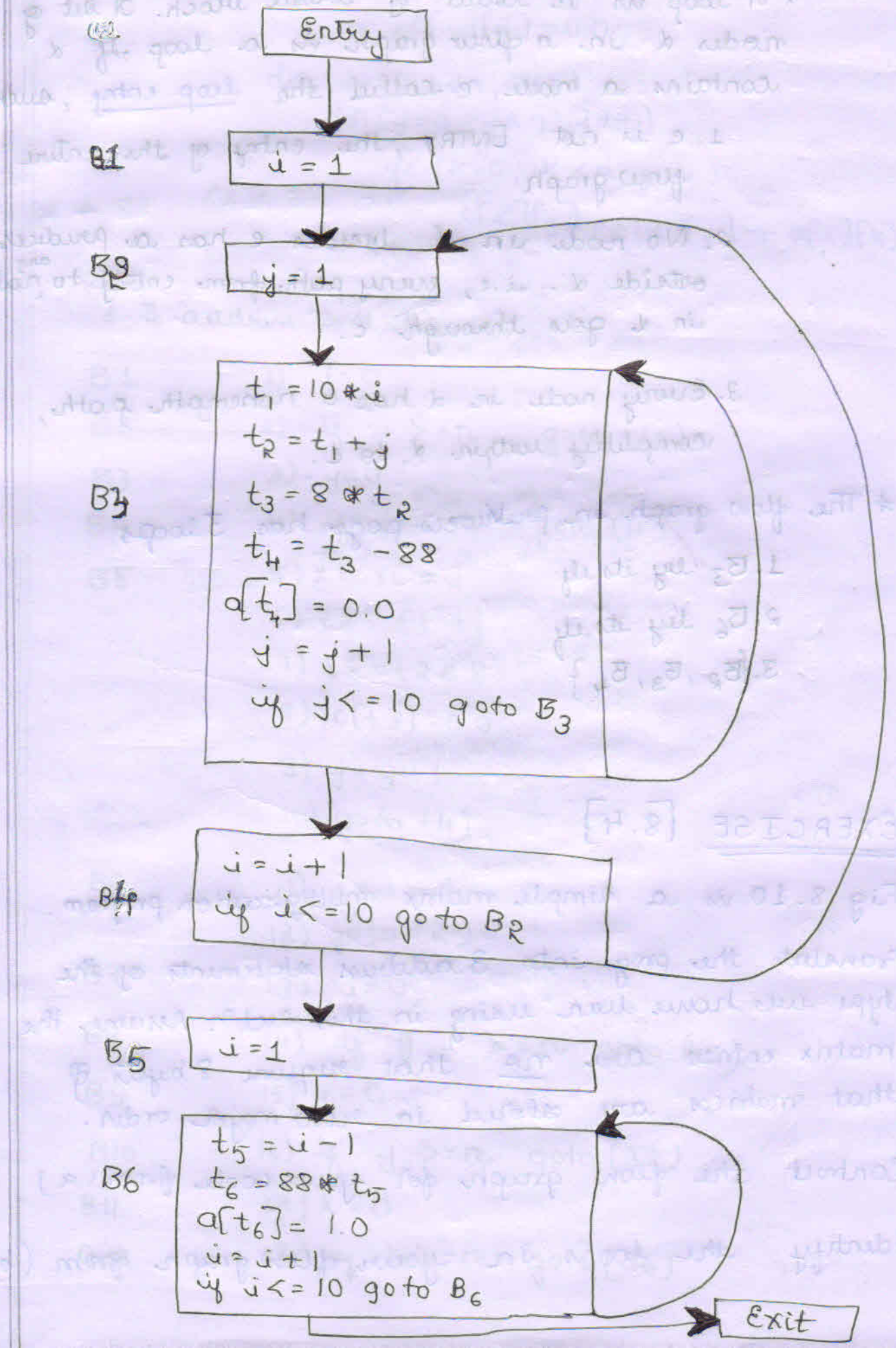
* Write flow graph for below source code

```
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 ~~for~~ refer to eq: 8.7



LOOPS

* A loop is a collectn of basic block. A set of nodes α in a flow graph is a loop, if α contains a node e called the loop entry, such

1. e is not ENTRY, the entry of the entire flow graph
2. No node in α besides e has a predecessor outside α. i.e., every path from entry to any node in α goes through e.
3. Every node in α has a nonempth path, completly within α, 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 sectn. Assume the matrix entries are no that require 8 bytes of that matrices 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 ~~inst~~ 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

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) t4 = n * i

20) t5 = t4 + j

21) t6 = t5 * 8

22) t7 = c[t6]

23) t8 = n * i

24) t9 = ~~(t9 * 8 t8 * k)~~ t8 + k

25) t10 = ~~t9~~ t9 * 8

26) t11 = a[t10]

27) t12 = n * k

28) t13 = t12 + j

29) t14 = t13 * 8

30) t15 = b[t14]

31) t16 = t11 * t15

32) t17 = t7 + t16

33) c[t16] = t17

34) k = k + 1

35) goto (18)

B14

36) j = j + 1

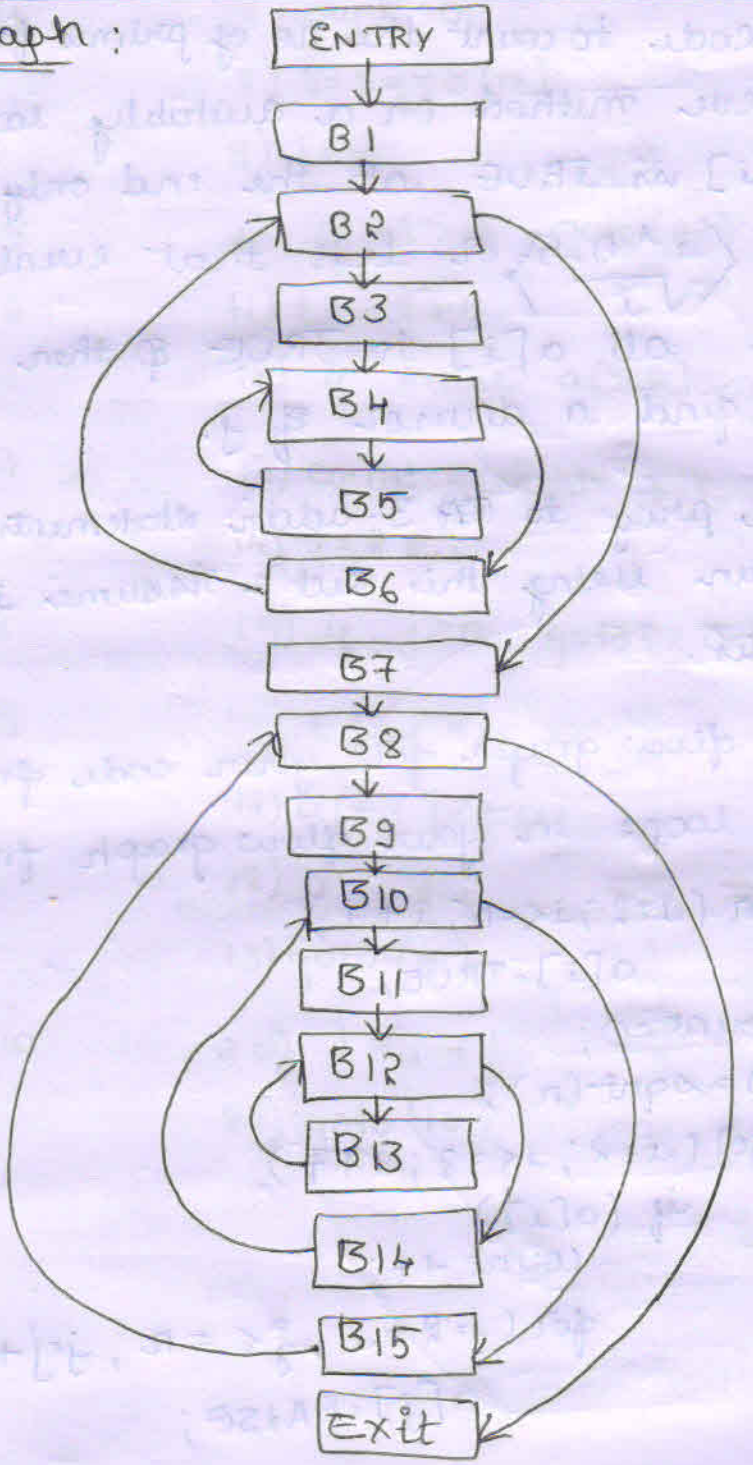
37) goto (16)

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, ~~B8~~, B14}
5. {B12, B13}

(2) Fig 8.11 is code to count the no of primes from 2 to n using the sieve method on a suitably large array. That is, a[i] is TRUE at the end only if there is no prime $\leq \langle i \wedge 0.5 \rangle$ or less that 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 sect^n. 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;
s = sqrt(n);
for (i=2; i<=s; i++)
{   if (a[i])
    {   count++;
        for (j=2*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] = TRUE
5) i = i + 1
6) goto (2)

B4 7) count = 0

8) S = Sqrt(n)

9) i = 2

B5 10) if i > s goto (22)

B6 11) t2 = i * 4

12) if False a[t2] goto (20)

B7 13) count = count + 1

14) j = 2 * i

B8 15) if j > n goto (20)

B9 16) t3 = j * 4

17) a[t3] = FALSE

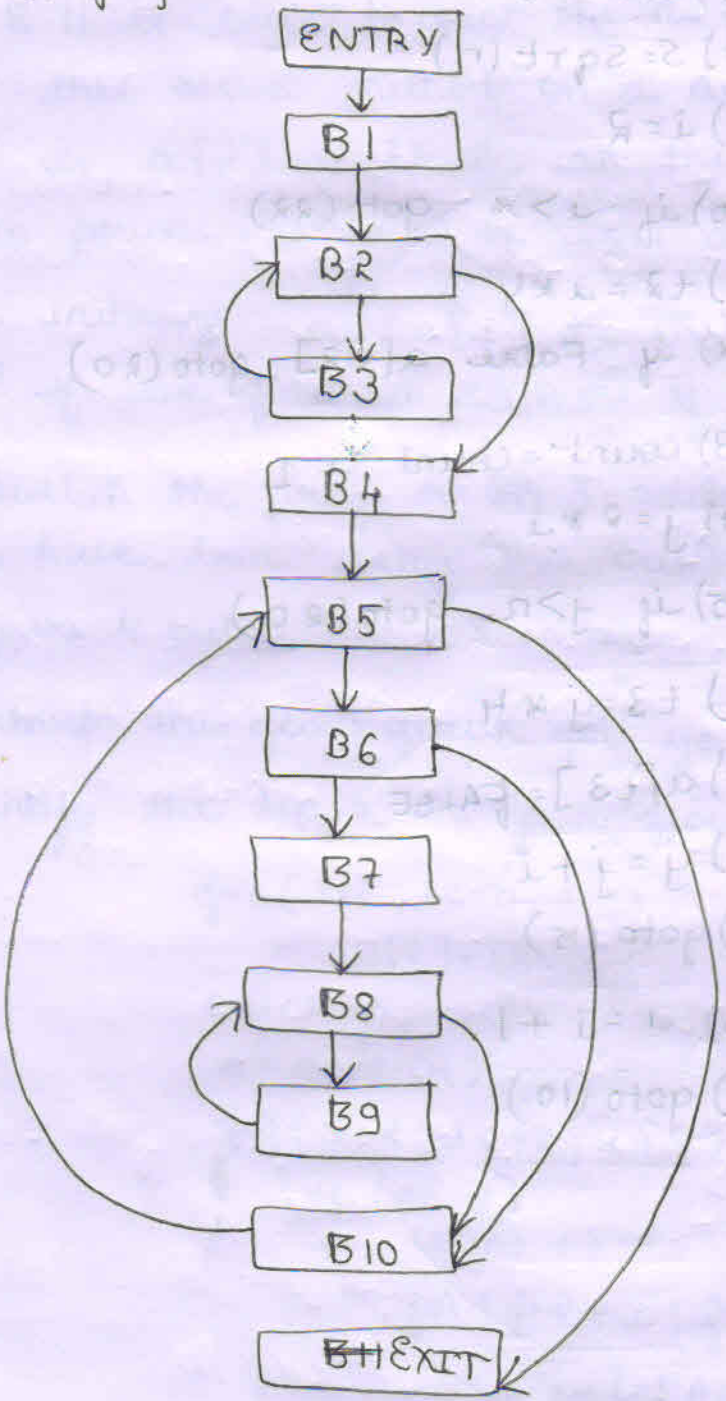
18) j = j + i

19) goto (15)

B10 20) 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 optimisat^n → More efficient without changing its output & no side affects

* Optimizat^n

  - local optimizat^n [within basic blocks] ↓ <simple> ✓
  - Global optimizat^n [entire program] ↓ <complex>

* We study local optimizat^n.

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 collect^n 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 a 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 & definat^n, prior to s, of the operands used by s.

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

Ø also attached to N is the list of variables for which it is the last definat^n 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 representat^n of a basic block results in several code improving transformat^n such as:

1. Eliminates local common subexpressions.
2. Eliminates dead code
3. Reordering of statements that donot depend on one another.
4. Apply algebraic laws for reordering operands of 3-addr instruct^n.

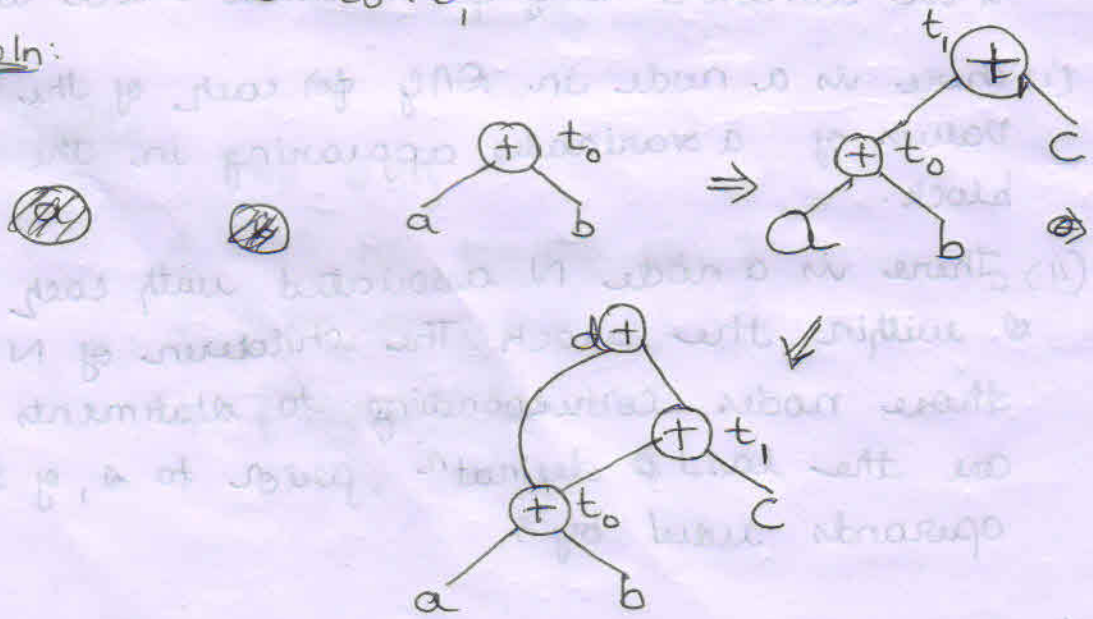
Eg: Give DAG representat^n of following basic block.

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$

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.

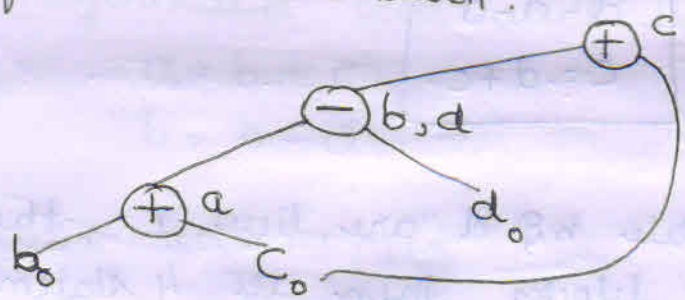
* FOR eg; consider the following statements in a basic block:

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.

* DAG for the basic block:

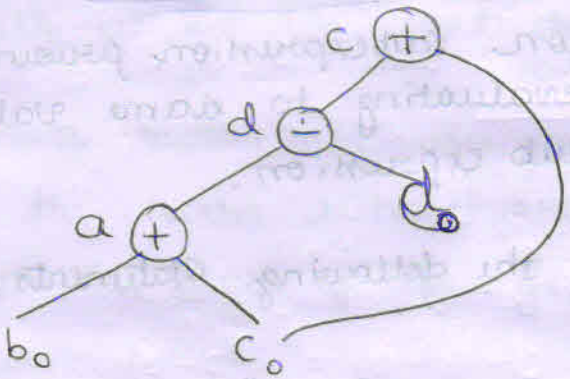


* Expression "a-d" present in statement ② & ④ are common sub expres ∴ both places will be evaluated to same time.
* Expn "b + c" used in statement ① is different from statement ③ & hence not common sub expression.
* Observe the following points:

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

→ Once b deleted, d can be used to recieve 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

```
a = b + c
d = a - d
c = d + c
```

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

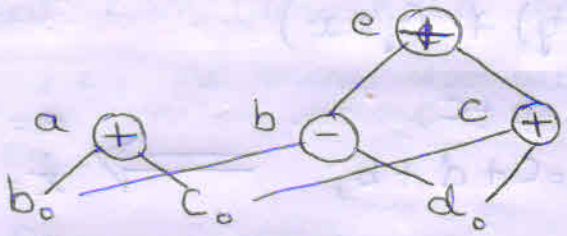
c) DEAD CODE ELIMINATION:

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

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

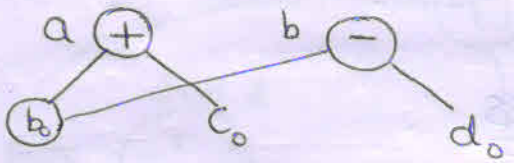
```
a = b + c
b = b - d
c = c + d
e = b + c
```

* DAG:



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

Resulting DAG



→ we can remove node @ & 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

* Strength reduction

* Constant folding

	Expensive		Cheaper
(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)	$x / 2$	=	$x * 0.5$
(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.

Eg: $(x * y) + (y * x)$

Eg:
```
a = b + c;
e = c + d + b;
```
⟹
```
a = b + c
t = c + d
e = t + b
```
⇒
```
// t is not
outside
a = b
e = c
```

* Some algebraic laws are not obvious

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 access in DAG is as follows:

1. An assignment from an array

Eg: $x = a[i]$

is represented by creating a node with "=[]" & 2 children representing initial of array, a_0 & index i. Variable x becomes label of this node.

2. An assignment to an array

Eg: $a[j] = y$

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

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

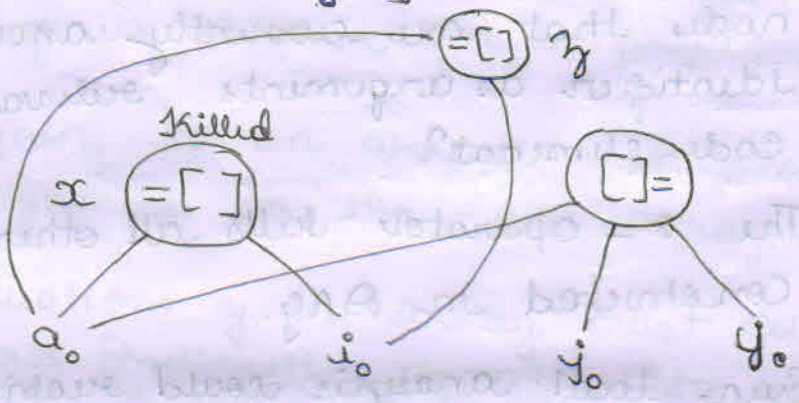
* Eg: Draw DAG for the block

$x = a[i]$

$a[j] = y$

$z = a[i]$

Soln:

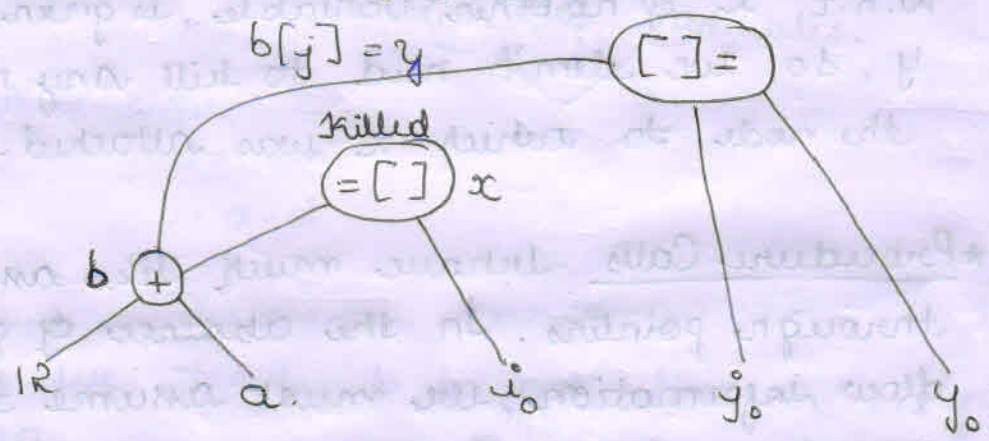


* Eg2: Draw DAG for the block

$b = 12 + a$

$x = b[i]$

$b[j] = y$



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

f) POINTER ASSIGNMENTS & PROCEDURE CALLS

* When we assign indirectly through a pointer say,

```
x = *p
*q = y
```

* We donot 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 elimenat?
* The *= operator kills all other nodes ~~as~~ for constructed in DAG.
* Even local analysis could restrict the ~~poo~~ of a pointer. For eg:

```
p = &x
*p = y
```

w.k.t x, & 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 wh it has access.
* Thus if procedure P is in the scope of variable a call to P both uses the node with attach variable x & kills that node.

9) Reassembling Basic Blocks from DAG's

After converting a basic block into a DAG & applying optimizat^n 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: ~~DA~~ 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 ori

∴ (i) less expensive since substration is replaced by assignment.

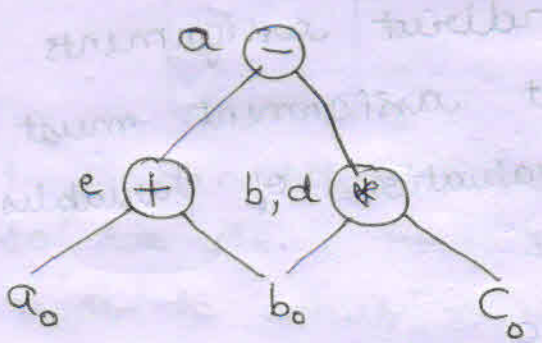
(ii) By global analysis, we can eliminate the use of computation of b outside the block by replac it by uses of d.

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

a) Only a is live on exit from the block

b) a, b & c are live on exit from the block

a) Only a is live

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

b) a, b & 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. Donot forget to include the comparision $i \leq 10$

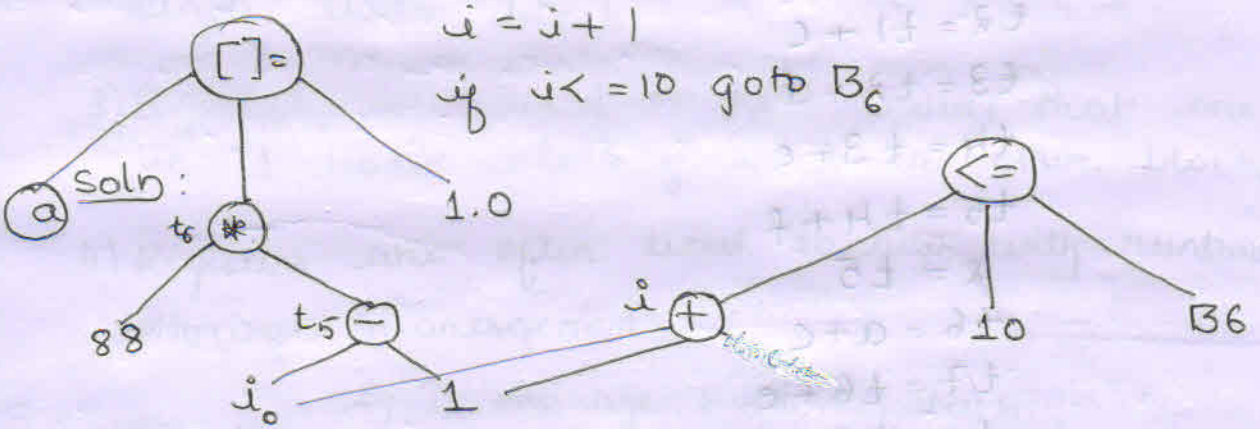
B6 : $t_5 = i - 1$

$t_6 = 88 * t5$

$a[t6] = 1.0$

$i = i + 1$

if $i <= 10$ goto B_6



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

$t_1 = 10 * i$

$t_2 = t_1 + j$

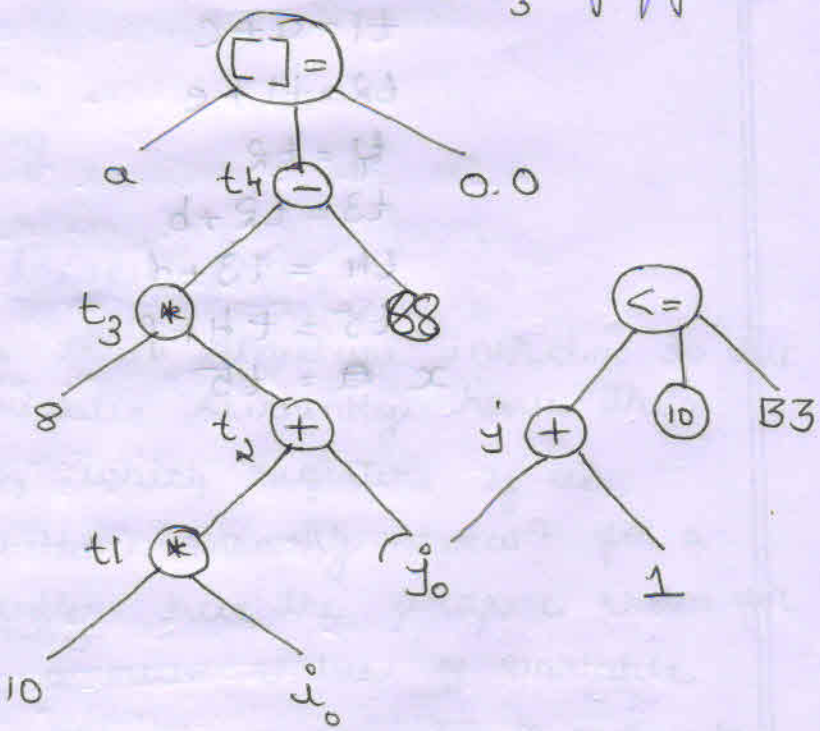
$t_3 = 8 * t_2$

$t_4 = t_3 - 88$

$a[t_4] = 0.0$

$j = j + 1$

if $j <= 10$ goto B_3



b) a, b & 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. Donot forget to include the comparision $i \leq 10$

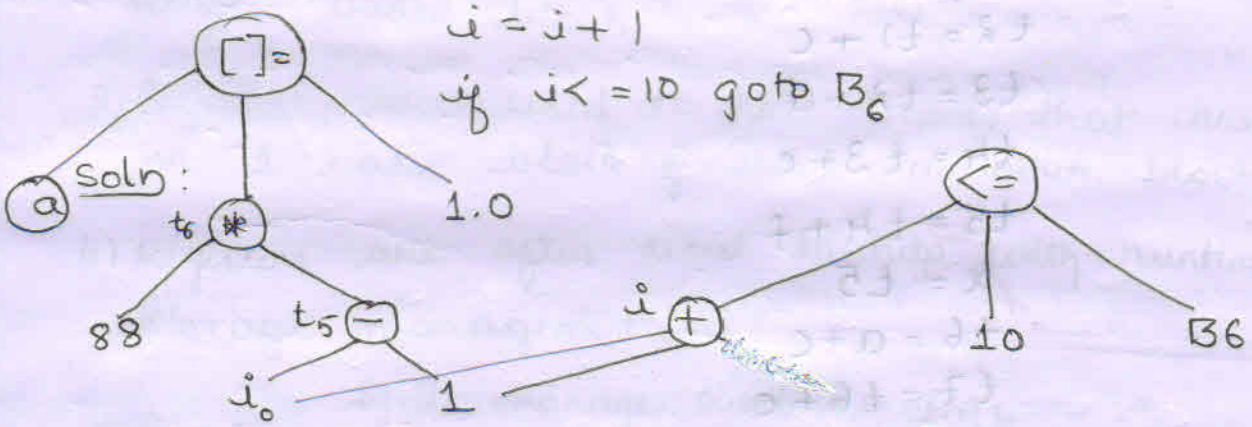
B6 : $t_5 = i - 1$

$t_6 = 88 * t5$

$a[t6] = 1.0$

$i = i + 1$

if $i <= 10$ goto B_6



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

$t_1 = 10 * i$

$t_2 = t_1 + j$

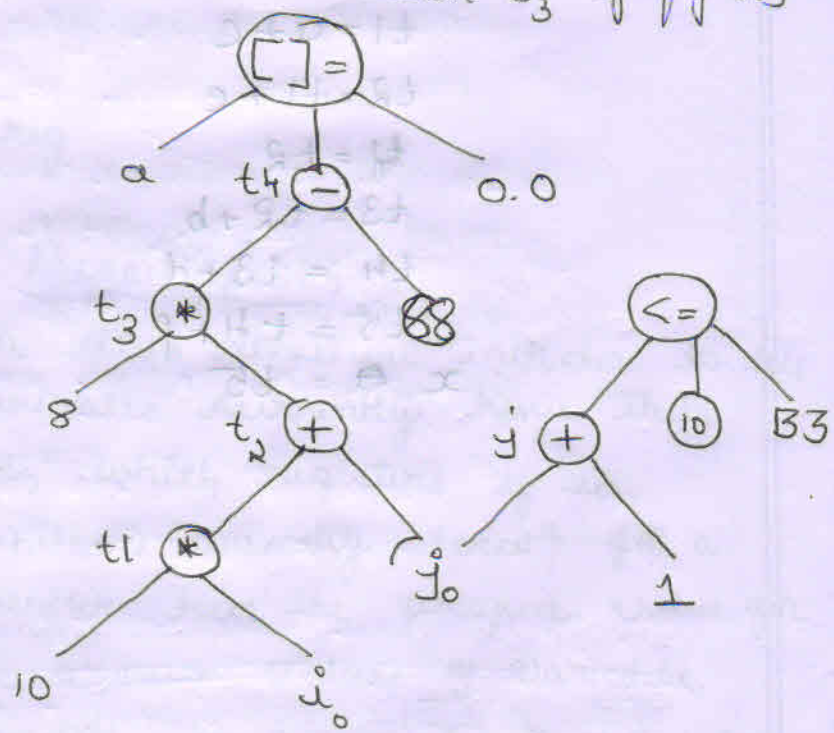
$t_3 = 8 * t_2$

$t_4 = t_3 - 88$

$a[t_4] = 0.0$

$j = j + 1$

if $j <= 10$ goto B_3



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

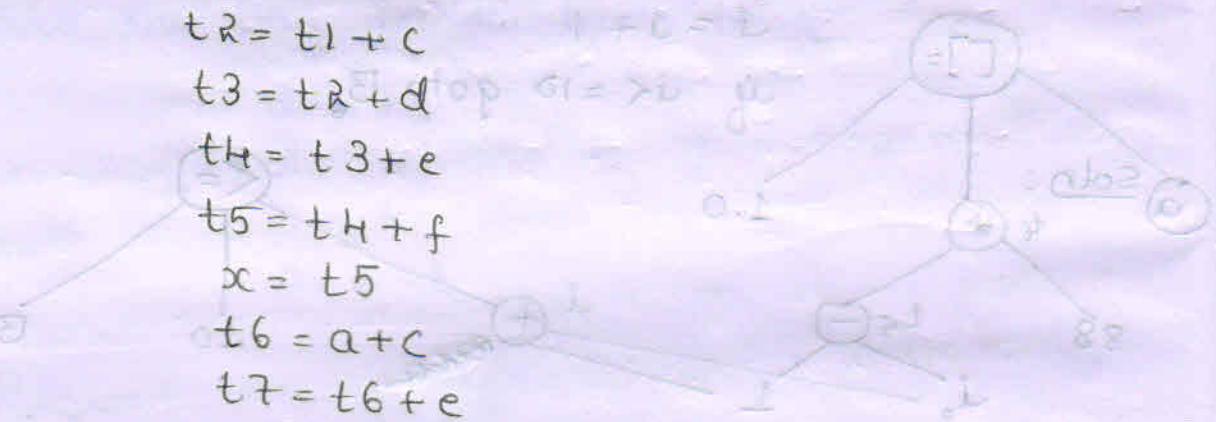
```
x = a+b+c+d+e+f;
y = a+c+e;
```

a) Give the 3 address statements for this block

b) Use the associative & commutative laws to modify the block to use the fewest possible no of instructions, assuming both x & y are live on exit from the block.

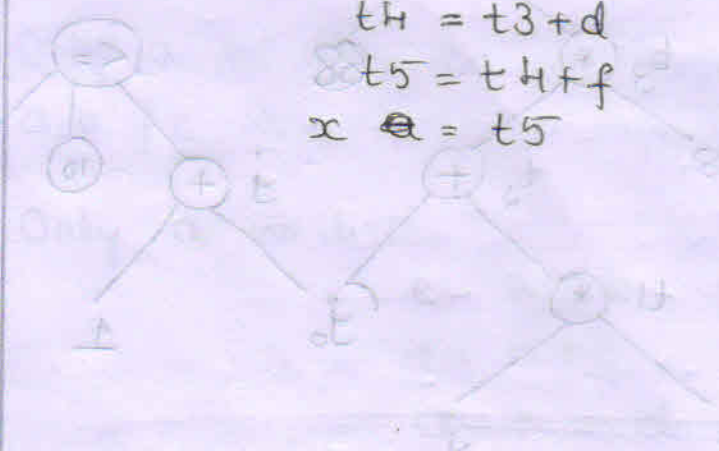
a) 3 address instⁿ

```
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
y = t2
t3 = t2+b
t4 = t3+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 operat^n must be in registers inorder 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, maintainance 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 inorder to tell
- what program variables currently have their values in a register, which registers if so.
- we need to know whether memory locat^n 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 nam whose current value present in registers. Initial all register descriptors are empty. As the code generat^n progresses, each registers will hold the value of zro or more names.

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

Code Generat^n 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 informat^n such as variab that are live on exit from the block.

* In a 3 address instruct^n, eg: $x = y + z$, we ~~treat~~ treat + as a generic (~~sugar~~) operator & ADD as the equivalent m/c instruct^n. A posible improv to the algorithm is to generate code for both $x = y + z$ & $x = z + y$ whenever + is commutat operator & pick the better code sequence.

* Machine instruct^n for Operations

For a 3-address inst^n such as $x = y + z$ do the follo

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

Call them R_x, R_y & R_z

2. If y is not in R_y, then issue an instruct^n LD R_y, y' where y' is one of the memory locat^n for y.

3. Similarly if z is not in R_z, issue an instruct^n LD R_z, z' where z' is a location for z.

4. Issue the instruct^n ADD R_x, R_y, R_z.

* Machine instruct^n 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 instruct^n 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 temporay 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 doesnot say that its value is located in the ~~(to)~~ memory locat^n say x, generate the instruct^n ST x, R where R → register
x → value exists @ the end of the block.

* Managing Register & Address descriptors

As the code-generat^n algorithm issues load, store & other m/c instruct^n, it needs to update the register & address descriptors.

* The rules are as follows:

1> For instruct^n LD R, x

a) Change register descriptor for R so it holds only x.

b) Change the addr descriptor for x by adding register R as an additional locat^n.

2> For instruct^n ST x, R

change addr descriptor for x to include its own memory locat^n.

3> For operation such as ADD R_x, R_y, R_z implies $x = y + z$

a) Change register descriptor for R_x so that it holds only x.

b) change the address descriptor of x so that its only locat^n is R_x.

c) Remove R_x 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 R_y,

a) Add x to the register descriptor for R_y.

b) change the address descriptor for x so that its only locat^n is R_y.

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

t = a - b

u = a - c

v = t + u

a = d

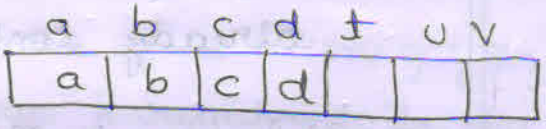
d = v + u

Soln:

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

Assume registers are enough, reuse the registers whenever possible.

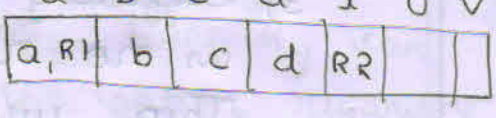
t = a - b



⇒ LD R1, a;

LD R2, b;

SUB R2, R1, R2



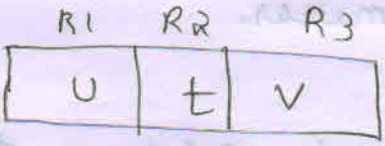
u = a - c

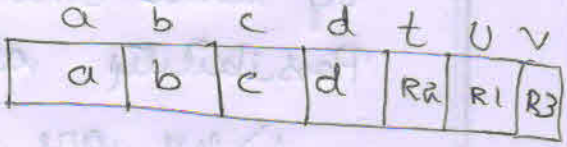
⇒ LD R3, c

SUB R1, R1, R3.

v = t + u

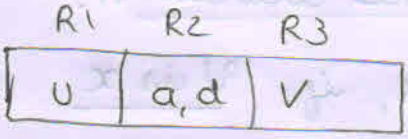
⇒ ADD R3, R2, R1

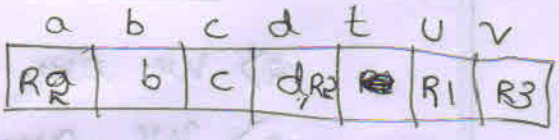




a = d

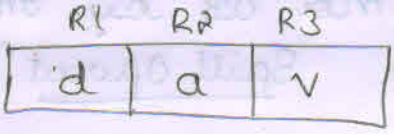
⇒ LD R2, d

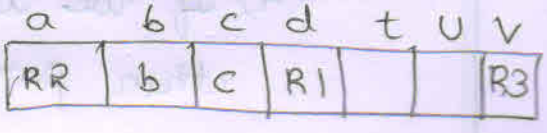




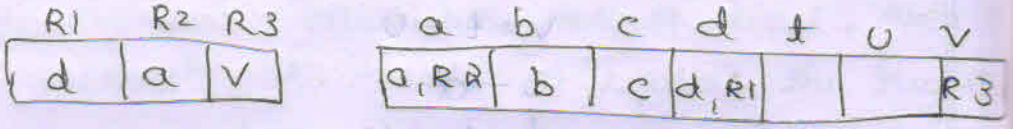
d = v + u

⇒ ADD R1, R3, R1





exit
ST a, R2
ST d, R1



Design of the Function getReg

Consider picking R_y 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 R_y.

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

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 instruct^{n} to place a copy of V in its own memory locat^{n}

ST V, R

NOTE: For select^{n} of the register R_x. The issues & ~~operat^{n}~~ 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 & z, since our m/c instruct^{n} allows 2 registers to be the same in one instruct^{n}.

2. If y is not used after instruct^{n} I, & 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

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

1> 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 & stack machine codes

(c) Linear representation such as postfix notation

(d) Graphical representation such as syntax trees & 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) Syntatic & semantic errors have been already detected.

2) The target program:

* The o/p of code generator is target program.
* The instruct^{n} set architecture of the target machine has a significant impact on the design of code generator.
* Most common architecture are:
  (a) CISC : It has few registers, has maximum of 2 operands of variety of addressing mode, variable length of instruction with side effects.
  (b) RISC : It has many registers, has maximum of 3 operands of simple addressing modes, relatively simple instruct^{n} 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 locat^{n} in memory & immediately executed.

b) Relocatable machine language program allows subprograms to be compiled seperately.

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 quality 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^n.

> Uniformity & completeness of the instruction set are imp factors.

> If we donot care about the efficiency of the target program, instruction select^n is straight forward.

> For eg:

```
x = y + z   ⇒  LD   R0, y
               ADD  R0, R0, z
               ST   x, R0
```

∴ produces redundant LD & ST

> Eg 2:

```
a = b + c   ⇒  LD   R0, b
d = a + b      ADD  R0, R0, C
               ST   a, R0      ┐
               LD   R0, a      ┘ → REDUNDANT!
               ADD  R0, R0, C
               ST   d, R0
```

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

> For eg:

```
a = a + 1   ⇒  LD  R0, a
               ADD R0, R0, 1
               ST  a, R0
```

can be replaced by → INC a

4) Register Allocation:

* Instruct^n 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 multiplicat^n & division.
* Multiplicat^n instruct^n is of the form

  M 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 instruct^n is of the form

  D 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 = a + b
t = t * c	t = t + c
t = t / d	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, 32
ST R1, t	D R0, d
	ST R1, t

5) Evaluational Order :

* The order in which computat^n are performed can effect the efficiency of the target code.
* When instruction are independent their evaluat^n order can be changed.
* Some computat^n 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)

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

$\Rightarrow$

$t1 = a+b$
$t2 = c+d$
$t3 = e*t2$
$t4 = t1-t3$

$\Rightarrow$

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

REORDER

$t2 = c+d$
$t3 = e*t2$
$t1 = a+b$
$t4 = t1-t3$

$\Rightarrow$

```
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 sequence of consecutive 3 address code [TAC] such that:

→ Flow of control enters at the begining 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

B2 : 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 <= 10 goto B_3

B4 : 10) i = i + 1
11) if i <= 10 goto B_2

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 <= 10 goto B_6

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 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 a variables appearing in the basic block.

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

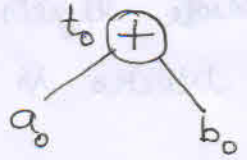
(iii) Node N is labelled by the operator applied at ⊗ & also attached to N is the list of variables for which it is last definatⁿ within the block.

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

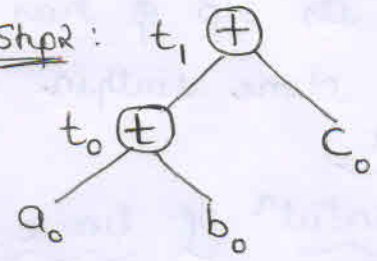
* DAG representatⁿ of basic block results in several code improving transformatⁿ such as:
  1) Eliminate local common subexpressions
  2) Eliminate dead code
  3) Reordering of statements that donot depend on one another
  4) Apply algebraic laws for reordering operands of 3-addr instructⁿ.

★ Eg: DAG representation for basic block : $t_0 = a + b$
$t_1 = t_0 + c$
$d = t_0 + t_1$

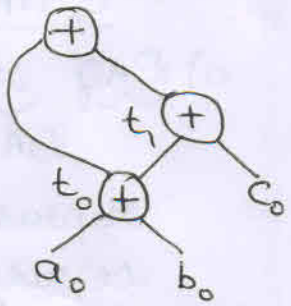
Step1:



Step2 :



Step3 : (Final)



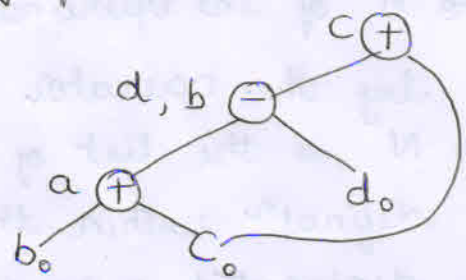
b) Finding local Common Subexpressions :

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

★ For eg: Conside following statements in a basic block

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

DAG for above basic block :

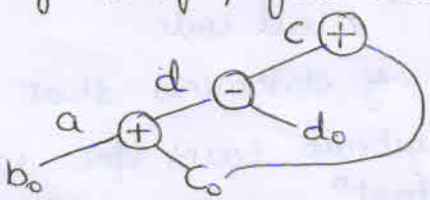


★ If b is not live on exit from the block, then we donot need to compute that variable

∴ b can be deleted

★ Once b deleted, d can be used to recieve 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:

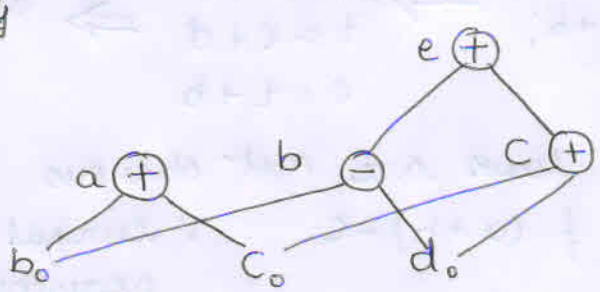
$$a = b + c$$
$$d = a - d$$
$$c = d + c$$

c) Dead code elimination

* We delete from a DAG any node (#) with no ancestors that has no live variables attached. Repeated applicat^n of this transformat^n will remove all nodes from the DAG that corresponds to dead code.
* For eg: Consider following 3 address statements in basic block

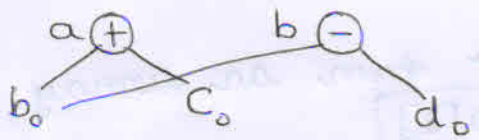
$$a = b + c$$
$$b = b - d$$
$$c = c + d$$
$$e = b + c$$

* 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.

	Expensive	Cheaper
* Algebraic properties (i)	$X+0=0+X$	$= X$
(ii)	$X*1=1*X$	$= X$
(iii)	$X-0$	$= X$
(iv)	$X/1$	$= X$
* Strength reduction (v)	X^2	$= X*X$
(vi)	$2*X$	$= X+X$
(v)	$X/2$	$= X*0.5$
* Constant folding (vi)	$2*3.14$	$= 6.28$

* Commutativity & Associativity
  - → DAG construct can help us here
  - → Apart from chcking **left op right**, we could also chck **right op left** for commutativity.

Eg: $(x*y)+(y*x)$

Eg2:
```
a = b + c;
e = c + d + b;
```
⟹
```
a = b + c
t = c + d
e = t + b
```
⟹ "If t not reused outside the block.
```
a = b + c
e = a + d
```

* Some algebraic laws are not obvious
  Eg: $a+(b-c) \neq (a+b)-c$ ∵ careful about the parenthesis.

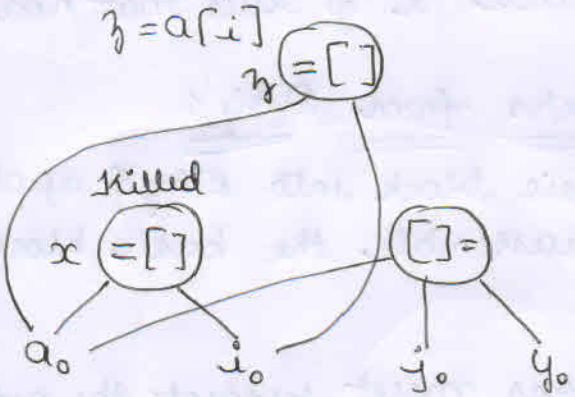
c) Array Refrences

* Array refrences cannot be treated like casual variables.
* The proper way to represent array access in DAG is as follows.
  1. An assignment from an array
     Eg: `x = a[i]`
     is represented by creating a node with operator "=[]" & 2 children representing initial value of array, a_0 & index i. Variable x becomes label of this node.
  2. An assignment to an array
     Eg: `a[j] = y`

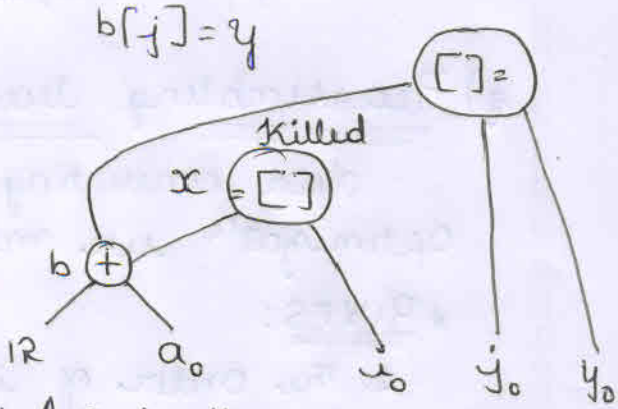
is represented by a new node with operator "[ ]=" & 3 children representing a_0, j & y. There is no variable labeling this node. Also creat^n of this node kills all currently constructed node whose value depends on a_0.

*Eg: DAG for the block

(i) x = a[i]
a[j] = y
z = a[i]



(ii) b = 12 + a
x = b[i]
b[j] = y



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

f) Pointer assignments & procedure calls.

* When we assign indirectly through a pointer. say,

```
x = *p
*q = y
```

* We donot 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:

```
P = &x
*p = y
```

W.k.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 & kills that node.

g) Reassembling basic blocks from DAG's

After converting a basic block into DAG & applying Optimizatn 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

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

* If both b & d are live on exit & 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
```

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

---

(3) For the following program 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:

B1	1) i = 1
B2	2) j = 1
B3	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 <= 10 goto (3)
B4	10) i = i + 1
	11) if i <= 10 goto (2)
B5	12) i = 1
B6	13) t5 = i - 1
	14) t6 = 88 * t5
	15) a[t6] = 1.0
	16) i = i + 1
	17) if i <= 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 :