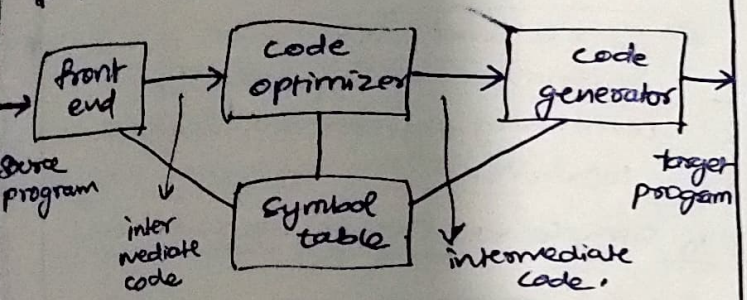
CD5

Code Generation: Issues in the design of a code Generator, The target language, Addresses in the target code, Basic Blocks and Flow graphs, optimization of Basic Blocks, peephole optimization, Register Allocation and Assignment.

Machine-Independent optimization: The principal sources of optimizations, Introduction to Data-Flow Analysis.

Code Generation:

* final phase.



Properties derived:

* correctness
* high quality
* Efficient use of resource of target machine.
* Quick code generation.

→ Output of code generation phase is machine code / object code.

Forms of object code:

* Absolute code.
* Relocatable machine code.
* Assembler code.

* It allows sub programs to be compiled seperately.
* Allows linking and loading of already compiled subroutines

* code that Containg refe-rences to actual addresses within programs address space.
* It can be placed directly in the memory and execution starts immediately.

Issues in the Design of a Code generator:

① **Correct code.**
- ↳ correctness.
- ↳ High-quality.
- ↳ Efficient use of resources of target code.
- ↳ Quick code generation.

② **Input to the code generator**
- ↳ IR + Symbol table.
- ↳ IR can be
  - ⇒ Linear repr (postfix)
  - ⇒ 3-Address repr (Quadruples)
  - ⇒ Virtual machine repr (stack machine code.)
  - ⇒ Graphical repr (syntax trees and dags).
- ↳ error-free.

③ **Target Programs (output)**
- ↳ Forms of object code.
- ↳ Usage of architectures like RISC, CISC & Stack based etc...

④ **Memory Management**
- ↳ Names —mapped to→ address of data objects in run time memory.
- ↳ makes use of symbol table. 3-Address statement refers to symbol table.
- ↳ Labels in 3-address have to be converted to address of instructions.

⑤ **Code generator main tasks:**

① **Instruction selection:**
- ↳ must map IR program into a code sequence.

* It makes code generation process somewhat easier.
* Slower because assembling, linking and loading is required.

- ↳ the complexity of performing this mapping is determined by factors.
  - i ⇒ level of the ir.
  - ii ⇒ nature of the instruction-set architecture.

(iii) → the desired quality of the generated code.

↳ speed of instruction and machine idioms are two important factors in selection of instruction.

↳ quality of generated code is determined by its speed and size.

(iv) **Register allocation and assignment:**

↳ **problem:** deciding what values to hold in what registers.

↳ **Registers:** fastest computational unit but do not have enough of them to hold all values.

↳ **Two subproblems are:**

Register Allocation: selecting set of variables that will reside in registers at each point in the program.

Register assignment: selecting specific register that a variable resides in.

(v) **Instruction ordering:**

* → Evaluation order is an important factor.

↳ requires a small number of registers to hold intermediate results

The Target Language:

* Familiarity with the target machine and its instruction set
* The target computer is a byte addressable machine with 4 bytes to a word.
* It has n general-purpose registers, R0, R1, ----, Rn-1.
* It has two-address instructions of the form:

(op, source, destination)

op-code ↓ data fields

* It has the following op-codes:

MOV (move source to destination)
ADD (add source to destination)
SUB (subtract source from destination)

* Source and destination of an instruction are specified by combining registers and memory locations.

A simple Target Machine Model

* **Load operations:** LD r, x
  LD r1, r2
* **Store operations:** ST x, r
* **Computation operations:**

OP dst, src1, src2

Ex: SUB r1, r2, r3 <=> r1 = r2 - r3

* **unconditional jumps:** BR L
* **conditional jumps:**

Bcond r, L

Ex: BLTZ r, L.

EXPT. No. _____ Date _____

Addresses in the code:

⇒ Target machine has a variety of addressing modes.

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c+contents (R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents (c+contents (R))	1
literal	#c	c	1

three address statement $x = y - z$ can be implemented by the machine instructions:

```
t1 = z        LD R1, y
t2 = y        LD R2, y
t3 = t1 - t2  SUB R1, R1, R2
x = t3        ST x, R1
```

(i) $b = a[i]$

```
LD R1, i
MUL R1, R1, 8
LD R2, a(R1)
ST b, R2
```

(ii) $a[j] = c$

```
LD R1, c
LD R2, j
MUL R2, R2, 8
ST a(R2), R1
```

(iv) $x = *p$

```
LD R1, p
LD R2, 0(R1)
ST x, R2
```

(v) $*p = y$

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

BASIC BLOCKS

* Basic Block is a sequence of consecutive statements in which flow of control enters at the beginning of the block and leaves at the end without halt or possibility of branching.

Ex:
(1) T1 = b + c
(2) T2 = T1 + d
(3) a = T2

Basic Block ✓

Characteristics:

* They do not contain any jump statements.
* There is no possibility of branching or getting halt in the middle.
* All the statements execute in the same order they appear.
* They do not lose the flow control of the program.

So, intermediate code is portioned into basic blocks. Each basic blocks become the nodes of a flow graph.

Partitioning Algorithm:

Step 1: First determine the leader statements.

Rules for finding leaders

* The first 3-address instruction in the Intermediate code is a leader.
* Any instruction that immediately follows a conditional or unconditional jump (goto) is a leader.
* Any instruction that is the target of a conditional or unconditional jump (goto) is a leader.

Step 2: Basic Block is formed starting at the leader and ends just before the next leader statement.

* Intermediate code to set a 10* 10 matrix to an identity matrix.

Ex :-

1. a = 10
2. b = 20
3. if a<b goto L1
4. c = a+b
5. goto L2
6. L1 : d = a-b
7. e = d*2
8. L2 : f = c+e.

As per Algorithm :

Leaders :-

Line 1 : Leader (first statement).

Line 3 :- Conditional branch

↳ Line 4 : c = a+b.

Line 5 : Unconditional jump.

↳ The target of jump is Line 8 (becomes leader).

Label targets : L1, L2

↓ target of line 3 ↓ target of line 5.

Leaders :- (Lines) 1, 4, 6, 8.

Blocks :-

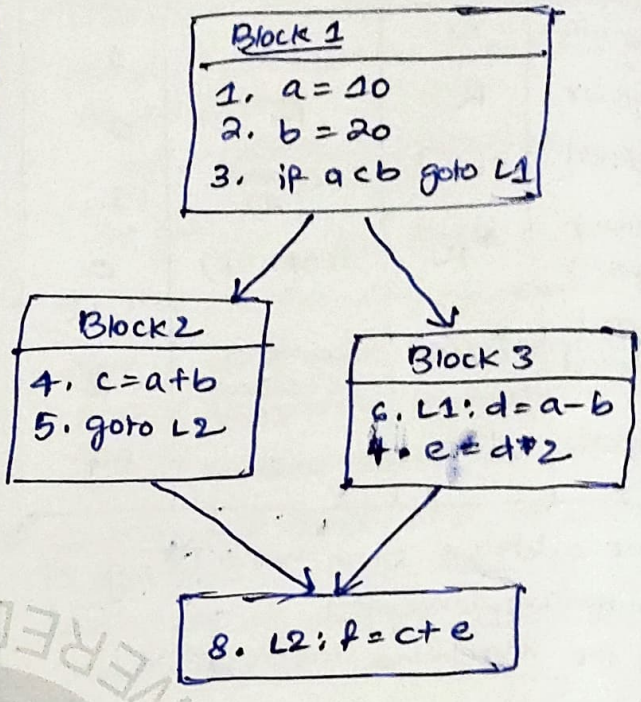
Block 1 :- Line (1-3)

Block 2 :- Lines (4-5)

Block 3 :- Lines (6-7)

Block 4 :- Line 8.

Control flow Diagram :



Transformations on Basic Blocks :

* A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
* Two important classes of transformation are :
  - ⇒ Structure-preserving transformations.
  - ⇒ Algebraic transformations.

1. Structure preserving transformation

(a) Common subexpression elimination

a := b+c	→	a := b+c
b := a-d		b := a-d
c := b+c		c := b+c
d := a-d		d := b

KESHAV MEMORIAL ENGINEERING COLLEGE

EXPT. No. _____ Date _____

(b) Dead code elimination

⇒ Suppose x is dead, that is, never subsequently used, at the point where the statement

x := y + z

appears in a basic block.

⇒ Then this statement may be safley removed without changing the value of the basic block.

(c) Renaming temporary variables:

⇒ t := b + c (t is temporary)

↓

u := b + c (u is new temporary)

⇒ If all uses of 't' can be changed to 'u' without changing the value of the basic block, such a block is called a normal-form block.

(d) Interchange of statements:

t1 := b + c
t2 := x + y

can be interchanged.

2) Algebraic transformations:

* can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Ex: x = x + 0
y = y * 1
x := y ** 2 ⇒ x := y * y.

Loops in Flow Graphs:

* A loop is a collection of nodes in a flow graph such that
1. All nodes in the collection are strongly connected.
2. The collection of nodes has a unique entry.

* A loop that contains no other loop is called an inner loop.

Nodes → computations
edges → flow of control.

Dominators: In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d.

denoted by :- d dom n

* Every initial node dominates all the remaining nodes in the flow graph.
* the entry of a loop dominates all nodes in the loop.
* every node dominates itself.

Ex:
1. i=0
2. L1: if i < n goto L2
3. goto L3
4. L2: print(i)
5. i = i+1
6. goto L1
7. L3: exit.

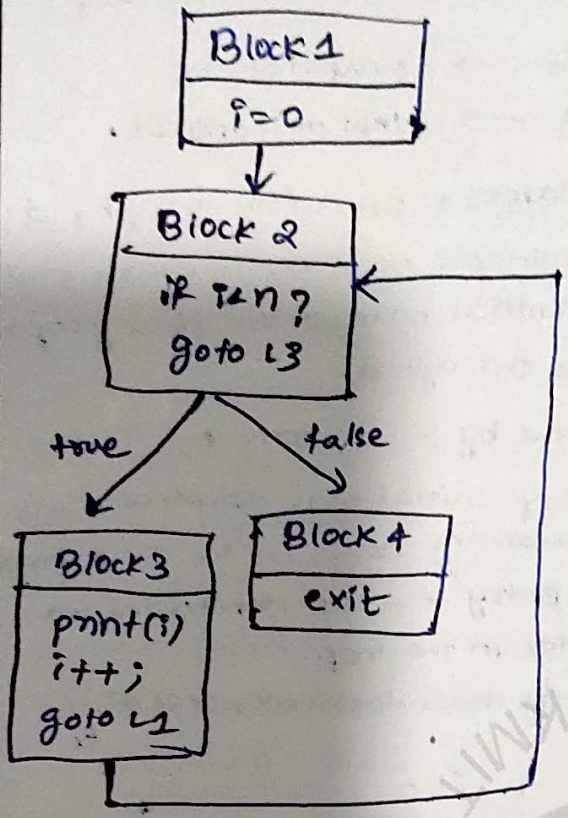
⇒ Identify Blocks:

B1: Line 1. (initialization)
B2: Lines (2-3) (loop condition)
B3: Lines (4-6) (loop)
B4: Line 7 (exit).

⇒ Control flow Graph (PTO...)

⇒ Dominator Analysis

B1: {A} (entry block)
B2: {A,B}
B3: {A,B,C} (every path through A,B)
B4: {A,B,D}



* one way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails.

* If a → b,
  b → head
  a → tail
  this type of edges are called as back edges.

* Ex:-

  7 → 4 4 DOM 7
  10 → 7 7 DOM 10
  4 → 3

* Given backedge n → d,
  natural loop :-
  → d plus the set of nodes that can reach n without going through d.
  → d → header of the loop.

Reducible flow graph:

* special flow graphs, for which several code optimization transformations are especially easy to perform,
  ↳ loops are unambiguously defined
  ↳ dominators can be easily calculated.
  ↳ data flow analysis problems can be solved efficiently.

* Exclusive use of structured flow-of-control statements
  if-then-else
  while-do
  continue
  break
  } graphs are always reducible

* There are no jumps into the middle of loops from outside. (entry is through its header),

Def:- A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and backward edges with the

EXPT. No.
Date

following properties:

* the forward edges from an acyclic graph in which every node can be reached from initial node of G.
* The back edges consist only of edges where heads dominate their tails.

Optimization of Basic Blocks.

Two types of

1. Structure-preserving Transformations.
2. Algebraic Transformations.

1. Structure preserving Transformations:

⇒ common sub-expression elimination:

* common sub-expression need not to be computed over and over again.
* Instead they can be computed once and kept in store where it's referenced when encountered again:

Repeated (Transformations).

Peephole optimization:

two phases:

Global optimization: Transformations are applied to a large program segments that include functions, procedures and loops.

⇒ Machine independent optimization is called as global optimization.

Local optimization: Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

⇒ peephole optimization is the best example for local optimization.

Peephole optimization: Optimizing the target code of a block.

* A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole).

Characteristics of peephole optimization:

1 ✓ Redundant-instructions elimination.
2 ✓ Flow-of-control optimizations.
3 ✓ Algebraic simplifications.
4 ✓ Use of machine idioms.
5 ✓ Unreachable code.

① ⇒ (1) MOV R0, a
(2) MOV a, R0
↓
can be removed because 'a' value is already stored in R0.

⑤ ⇒ removal of unreachable code
> Unlabeled instruction immediately following an unconditional jump may be removed.

Example:-

```
#define debug 0
IF(debug){
    print debugging information
}
```

⇓ Translated.

```
IF debug = 1 goto L2
    goto L2.
L1: print debugging information
L2: - - - - - - - - - - (a)
```

⇓ Optimized

If debug ≠ 1 goto L2
print debugging information
L2: ---------- (b)

↓ Statement of (b) evaluates to constant true it can be replaced by

If debug ≠ 0 goto L2
print debugging information
L2: ---------- (c)

(2) Flows The unnecessary jumps can be eliminated in either the intermediate code (or) the target code.

(3) $x = x * 0$
(or)
$x = x * 1$.

(5) Reduction in Strength:

* replaces expensive operations by equivalent cheaper ones on the target machine.
* $(x * x)$ is a variably cheaper than call to an exponentiation value.

(4) use of machine idioms
* efficient auto increment and auto decrement mode.
* improves the quality of code.

$i = i+1 \Rightarrow i++$
$i = i-1 \Rightarrow i--$

Register Allocation and Assignment

* Four strategies for deciding what values in a program should reside in a register and which register each value should reside.

1. Global Register Allocation:

⇒ Simple code generation algorithm does local (block based) register allocation.
⇒ This resulted that all live variables be stored at the end of the block.
⇒ To save some of these stores and their corresponding loads, we might rearrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally).
* Frequently used variables are stored in fixed registers.
* So, assign some fixed registers to hold most active variables in each inner loop.

2. Usage counts:

* we can save cost by keeping a variable x in register for the duration of loop L.
⇒ Sum over all blocks (B) in a loop (L).
⇒ for each use of x before any definition in the block we add one unit of saving.

EXPT. No. _____
Date _____

Usage cost:

Σ blocks B in L

use(x, B) + 2 * live(x, B)

use(x, B): number of time x is used in B prior to any definition of x in B.

live(x, B): 1 if x is live on exit of B otherwise 0.

3) Register Allocation for outer loops:

If outerloop L1
innerloop L2

a) The names allocated registers in L2 need not be allocated registers in L1 - L2.

⇒ if we choose to allocate x a register in L2 but not L1, we must load x on entrace to L2 and store x on exit from L2.

4) Register allocation by Graph colouring:

⇒ If all the registers are occupied/in use, if we want a register to store a new variable, one of the register content is to be stored into memory location (spilling).

⇒ Register allocation by Graph coloring is a simple and systematic technique for register allocation and management.

1st pass: Target machine instructions are selected with an assumptions that there are ∞ number of symbolic registers are available.

* Names used in 3-Address code become names of symbolic registers.
* three addresses instructions become machine language instructions.
* once the instruction is selected, 2nd pass assigns physical registers to symbolic registers.

2nd pass: For each procedure, a register-inference graph is constructed, in which nodes are symbolic registers, and an edge connect two nodes if one is live at the point where the second is defined.

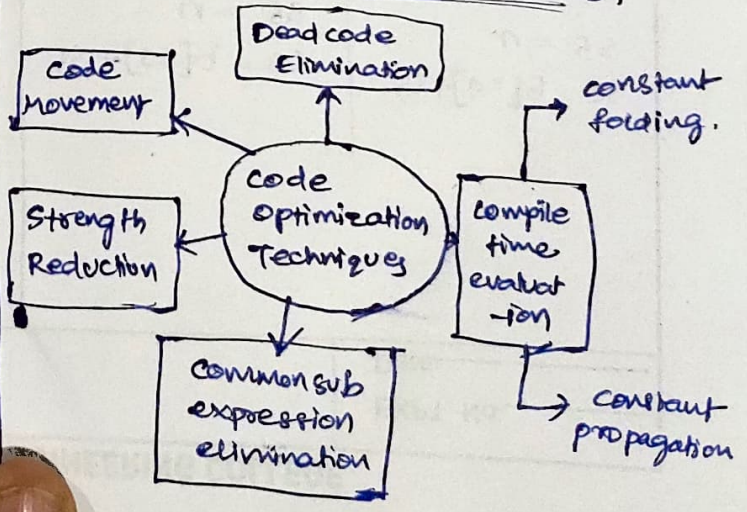
Machine Independent optimizations:

* to enhance the performance of the code.
* Improve code by making it consume fewer resources (CPU, memory, registers).
* process involves -
  ⇒ Eliminating the unwanted code lines.
  ⇒ Rearranging the statements of the code.

Advantages:

* optimized code has fast execution speed.
* " " utilizes the memory efficiently.
* optimized code gives better performance.

Principle Sources of optimizations:



1. Compile time Evaluation :

A) Constant Folding :

circumference of circle = (22/7) * Diameter.

→ evaluated at compile time.
→ replaced with its result 3.14.
→ this saves the time at runtime.

B) Constant propagation :

pi = 3.14, radius = 10

Area = pi * radius * radius.

→ substitutes values at compile time
→ then evaluates 3.14 * 10 * 10.
→ replaced with result 314.
→ This saves the time at runtime.

2) common subexpression elimination;

Before	After (opt)
S1 = 4 * i	S1 = 4 * i
S2 = a[S1]	S2 = a[S1]
S3 = 4 * j	S3 = 4 * j
S4 = 4 * i	S5 = n
S5 = n	S6 = b[S1] + S5
S6 = b[S4] + S5	

3) code Movement :

Before	After (opt)
for (int j=0; j<n; j++) { x = y + z; a[j] = 6 * j; }	x = y + z; for (int j=0; j<n; j++) { a[j] = 6 * j; }

4) Dead code Elimination :

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

5) Strength Reduction ;

Before	After
B = A * 2	B = A + A

* Additional optimization in Global ;

⇒ Forward copy propagation.
⇒ code motion.
⇒ Loop strength reduction.
⇒ Induction variable elimination.

EXPT. No.
Date

Translation of expression into stream of atoms:

⇒ a+b-c

(ADD, a, b, T1)

(SUB, T1, c, T2)

⇒ LOD ⇒ Load the first operand into a register.

STO ⇒ store the result back to memory

⇒ atoms:

```
LOD R1, a
ADD R1, b
STO R1, T1
LOD R1, T1
SUB R1, c
STO R1, T2
```

Introduction to Data flow Analysis

* refers to a body of techniques that derive information about the flow of data along program execution paths.
* Analyses that determines the information regarding definition and use of data in program.

Basic Terminology:

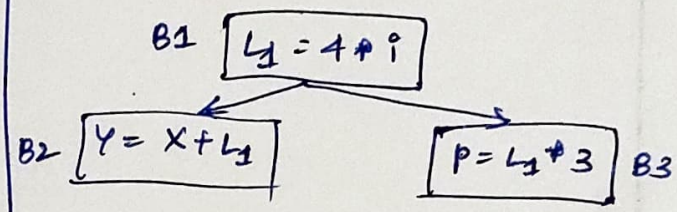
⇒ Definition Point: Place/line in which a variable is defined.

⇒ Reference Point: Place/line in which a variable is referred.

⇒ Evaluation Point: Place/line in which operations (arithmetic) is performed.

Dataflow properties:

Available expression: An expression 'a+b' is said to be available at a program point 'x', if none of its operands gets modified before their use.



Here 4*i is availble in B2 and B3.

Reaching Definition: A definition D is reaching to a point x if D is not killed or redefined before that point.