Code Generator:-

Issues in the design of code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Choice of evaluation order.

Input to code generator:-

* It consists of the intermediate representation of source program, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

Target Program:- / object program

* The output of the code generator is the target program.

Object code Forms:-

1. Absolute Machine Language. (executable code)
2. Relocatable Machine Language. (object files for linker)
3. Assembly Language. (facilitates debugging)

* Absolute machine language can be placed in a fixed memory location and can be immediately executed.

- Relocatable machine language program allows subprograms to be compiled separately. Relocatable object modules can be linked together and loaded for execution by a linking loader.
- Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro facilities of assembler in generating code.

if memory → register → load

register → memory → store

Memory Management :-

- Mapping the names in the source program to addresses of data objects is done by the front end and the code generator.
- The type in a declaration determines the width i.e., the amount of storage, needed for the declared name.

Instruction Selection :-

- Selecting best instructions will improve the efficiency of the program.

The three address code statements

$P := Q + R$ can be converted as

```
MOV , Q, R0
ADD   R, R0
MOV   R0, P.
```

Register Allocation:-

- A key problem in code generation is deciding what values to hold in what registers.
- Use of registers make the computation faster in comparision to that of memory, so efficient utilization of registers is important.
- The usage of registers is subdivided into two sub problems:

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

Register assignment: select a specific register that a variable reside in.

Choice of evaluation order:-

- The order in which the computations are performed will effect the efficiency of the target code.
- Some computational orders, some will require only fewer registers to hold the intermediate results.

Target Machine:

- The Target machine has the following characteristics.

1. It is byte addressable.
2. It has 4 bytes per word.
3. It has n general purpose registers, $R_0, R_1, \ldots R_n$.
4. It has two address instruction of the form

$$\underset{\text{opcode}}{\text{op}} \quad \underbrace{\text{source, destination}}_{\text{data fields}}$$

Eg.	MOV	(move S to D)
	ADD	(Add S to D)
	SUB	(subtract S foomD)

+ The addressing modes and their associated cost are as shown below.

Mode	Form	Address	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 indicates constant to.	Source to be a costant	1

Instruction cost :-

- Instruction cost is nothing but 1 plus cost associated with source and destination addressing modes.

$$I.C = 1 + S.C + D.C$$

Eg: $a = b + c$
$\quad\quad 2 + 3$

1. MOV b, R_0 2 — $1 + 1 + 0 = 2$
ADD $\overset{3}{c}$, $R_0 \overset{(2)}{5}$ — $1 + 1 + 0 = 2$
MOV R_0, a — $1 + 0 + 1 = 2$

$T.C = 6$

2. Mov b, a — $1 + 1 + 1 = 3$
Add c, a — $1 + 1 + 1 = 3$

$T.C = 6$

3. Mov $*R_1$, $*R_0$ — $1 + 0 + 0 = 1$
ADD $*R_2$, $*R_0$ — $1 + 0 + 0 = 1$

$T.C = 2$

Machine Dependent Optimization / Peep Hole Optimization:

- Peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window".

- Peephole: a short sequence of target instructions that may be replaced by a shorter/faster sequence.

- Common techniques applied in peephole optimization are:
  - Redundant instruction elimination.
  - Eliminating Unreachable code
  - Flow of control optimizations
  - Algebraic simplification
  - Strength reduction
  - Use of machine idioms.

Redundant Instruction Elimination:-

* If the target code contains

```
Mov R, a
Mov a, R
```

We can delete the second instruction because the first instruction ensures that the value of a is already in the register R. But if the second instruction has a label we cannot delete /remove it.

Eliminating Unreachable Code:-

- Unreachable code is a part of the program code that is never accessed because of programming constructs.

```
void add_ten(int x)
{
    return x+10;
    printf ("%d", x); //unreachable code
}
```

Flow of Control Optimizations:-

- If we have jumps to jumps, then these unnecessary jumps can be removed.

Eg: ①

```
goto L1
 --
L1: goto L2
   ⇓
goto L2
 ..-
L1: goto L2
```

②

```
goto L1                      goto L2
 - - —                        - —
 - - —                        - —
L1: goto L2        =>        L1: goto L2
 - - -                        - - -
 - —                          - - -
L2: - —                      L2: - —
 - -
 - -
```

Algebraic Simplification:-

- Eliminate the instructions like the following

$x = x + 0$

$x = x * 1$

- The quality of the intermediate code can be improved by taking the advantage of algebraic identities.

- Some of the algebraic ~~ed~~tentities are.

Name	Example.
Additive identity	x+0 = x
Multiplicative identity	x*1 = x
Multiply with 0	x*0 = 0

Strength Reduction:-

Expensive operation	Less expensive operation
y = x*2	Y = x+x
y = x**2	y = x*x
y = x*32	y = x<<5
y = x/8	y = x>>3

Use of machine Idioms:-

- The target instructions have equivalent machine ~~instruction~~ operations to perform some operations.
- We can replace the target instructions by equivalent machine instructions in order to improve the efficiency.

Eg: a = a+1

```
MOV a, R
ADD #1, R   →  INC a
MOV R, a
```

Code Generation Algorithm:-

. Code generation algorithm takes a sequence of three address instructions as input and generates the target code.

. An esiential part of this algorithm is a function getReg(l) which selects registers for each memory location associated with the three address instruction. l,

. This algorithm uses descriptors to keep track of registers addresses for variables.

. The register descriptor is used to keep track of registers where variables are stored.

. The address descriptor keeps track of the locations where the current value of the variable can be found.

Function getreg():-

The function getreg() when called upon to return a location where the computation specified by the three address statement $x = y \text{ op } z$ should be performed, returns a location L as follows,

1) First it searches for a register already containing the name y, if ~~sea~~ such register exists and if y is not live and has no further use after execution of $x := y \text{ op } z$, then return the register of y for L.

2) Otherwise getreg() searches for an empty register and if an empty register is available, then it returns in for L.

3) If no empty register exists and if x has further use in the block or ~~op~~ then getreg() finds a suitable occupied register. The register is empty by storing its value in the proper memory location, 'M'.

1. Generate code for the following expression.

$x = (a-b)+(a-c)+(a-c)$

Three address code can be written as

$t_1 := a-b$

$t_2 := a-c$

$t_3 := t_1+t_2$

$t_4 := t_3+t_2$

$x := t_4$

Using the code generation algorithm the target code can be generated as:

Statement	Generated Code	Register Descriptor	Address Descriptor
$t_1 = a-b$	MOV a, R_0 SUB b, R_0	All registers are empty R_0 contains a R_0 contains t_1	t_1 in R_0
$t_2 := a-c$	MOV a, R_1 SUB c, R_1	R_1 contains a R_1 contains t_2	t_1 in R_0 t_2 in R_1
$t_3 := t_1+t_2$	ADD R_1, R_0	R_0 contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
$t_4 = t_3+t_2$	ADD R_1, R_0	R_0 contains t_4	t_4 in R_0
$x = t_4$	MOV Ra, x	R_0 contains x	x in R_0 and memory

2. $x = a+b*c$

Three address code can be written as

$t_1 := b*c$

$t_2 := a+t_1$

$x = t_2$

R_0, R_1 are registers

~~MOV b, R_0~~
~~MUL c, R_0~~
~~ADD a, R_0~~
~~MOV R_0, x~~

MOV b, R_0
MUL c, R_0
MOV a, R_1
ADD R_1, R_0
MOV R_0, x

3. $x = (a+b) - ((c+d) - e)$

Three address code

$t_1 := a + b$
$t_2 := c + d$
$t_3 := t_2 - e$
$t_4 := t_1 - t_3$
$x := t_4$

R_0, R_1 are registers

```
MOV   a, R0
ADD   b, R0    , R0 -> t1
MOV   c, R1
ADD   d, R1 ,  R1 -> t2
SUB   e, R1 ,  R1 -> t3
SUB   R1, R0,  R0 -> t4
MOV   R0, x
```

Evaluation order / Instruction Scheduling.

4. Generate the code for the following expression.

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

Three address code

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

(or)

$t_1 := c + d$
$t_2 := e - t_1$
$t_3 := a + b$
$t_4 := t_3 - t_2$

R_0, R_1 are registers

```
(1+1+0) MOV, a, R0
(1+1+0) ADD  b, R0        R0 -> t1
(1+1+0) MOV  c, R1
(1+1+0) ADD  d, R1        R1 -> t2
        ~~SUB e, R1       R1 -> t3~~
        ~~SUB R0, R1      R1 -> t4~~
(1+0+1) MOV  R0, t1 ,     R0 -> t1
(1+1+0). MOV e, R0 ,      R0 -> e
(1+0+0) SUB  R1, R0 ,     R0 -> t3
(1+1+0) MOV  t1 , R1 ,    R1 -> t1
(1+0+0) SUB  R0, R1 ,     R1 -> t4
(1+0+1) MOV  R1 , t4
```

T.C = 18

Now, if we change the ordering sequence of the above three address code

$t_2 := c + d$

$t_3 := e - t_2$

$t_1 := a + b$

$t_4 := t_1 - t_3$

```
MOV    c, Ro                       1+1+0 = 2
ADD    d, Ro  ⟹ Ro ⟶ t2            1+1+0 = 2
MOV    e, R1                       1+1+0 = 2
SUB    Ro, R1       R1 ⟶ t3        1+0+0 = 1
MOV    a, Ro                       1+1+0 = 2
ADD    b, Ro        Ro ⟶ t1        1+1+0 = 2
SUB    R1, Ro       Ro ⟶ t4        1+0+0 = 1
MOV    Ro, t4                      1+0+1 = 2
                                   _________
                                   T.C = 14
```

Generate code for indexing and pointer operations.

		i in Register Ri	i in Memory Mi
Indexing	a = b[i]	MOV b(Ri), R	MOV Mi, R MOV b(R), R
	b[i] = a	MOV a, b(Ri)	MOV Mi, R MOV a, b(R)
Pointer	a = *p	MOV *Ri, a	MOV Mi, R MOV *R, R
	*p = a	MOV a, *Ri	MOV Mi, R MOV a, *R

* Assume three registers are available, Generate the code for the following expressions.

$x = a[i] + 1$ $\quad\quad$ $a[i] = a[i] + b[j]$

Assume i, j are in memory locations.
let R_0, R_1, R_2 be the registers.

$x = a[i] + 1$

Three address code :-

	MOV M, R_0	
$t_1 = a[i]$	MOV a(R_0), R_0	$R_0 \to t_1$
$t_2 = t_1 + 1$	ADD #1, R_0	$R_0 \to t_2$
$x = t_2$	MOV R_0, x	

$a[i] = a[i] + b[j]$

Three address code :-

$t_1 = a[i]$ →	{ MOV M, R_0 MOV a(R_0), R_1	$R_1 \to t_1$
$t_2 = b[j]$ →	{ MOV M_2, R_2 MOV b(R_2), R_2	$R_2 \to t_2$
$t_3 = t_1 + t_2$ →	ADD R_1, R_2	$R_2 \to t_3$
~~a~~a[i] = t_3 →	MOV R_2, a[R_0)	

Interprocedural Optimization:-

* It is a kind of code optimization in which collection of optimization techniques are used to improve the performance of the program that contains many of the frequently used function of blocks,

Eg: Global common sub expression.