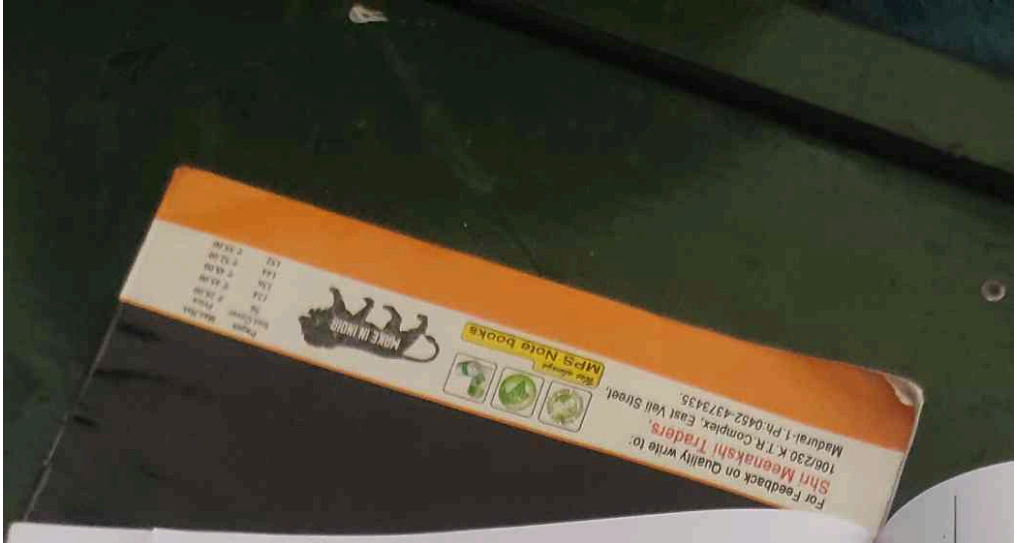

ISSUES IN THE DESIGN OF A CODE GENERATOR:

1. Input to Code Generator
2. Target Program
3. Memory Management
4. Instruction Selection
5. Register Allocation
6. Evaluation order.

Input to Code Generator:

* The input to the code generator consists of the intermediate representation of the source program produced by the front end together with information in the symbol table that is used to determine the run time addresses of data objects denoted by the names in the intermediate representation.
* There are several forms of intermediate representation
  - → linear representations such as postfix notation
  - → three address statements such as quadruples, triples and indirect triples.
  - → Virtual machine representation such as stack machine code.
  - → graphical representation such as syntax trees and dags

Target Program:

* The output of the code generator is the target program
* The target program may take on a variety of forms
    1) absolute machine language
    2) relocatable machine language
    3) assembly language.
* Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed.
* Producing a relocatable machine language program as output allows subprograms to be compiled separately.
* A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
* Producing an assembly language program as output makes the process of code generation easier.

Memory Management:

* Mapping names in the source program to address of data objects in run time memory is done by the front end of a compiler and the code generators.
* A name in a three address statement refers to a symbol-table entry for the name.
* The type in a declaration determines the width i.e) the amount of storage needed for the declared name.
* From the symbol table information, a relative address can be determined for the name in a data area of the procedure.
* Static allocation and stack allocation are used to convert the intermediate representation into addresses in the target code.

Instruction Selection:

* The factors of instruction selection are,
  a) uniformity
  B) completeness
  c) machine idioms
  d) instruction speeds.

* If we do not care about the efficiency of the target program instruction selection is straightforward.
* For each type of three address statement, we can design a code skeleton for generating the target code.
* Every three address statement of the form $x = y + z$ can be translated into the code sequence

```
MOV  y  R0
ADD  z  R0
MOV  R0 x
```

* Unfortunately this kind of statement-by-statement code generation often produces poor code.
* For eg) the sequence of statement

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

would be translated into

```
MOV  b  R0
ADD  c  R0
MOV  R0 a
MOV  a  R0
ADD  e  R0
MOV  R0 d
```

* Here the fourth statement is redundant
* The quality of the generated code is determined by its size and speed.
* If the target machine has an "increment" instruction (INC) then the three address statements a = a+1 may be implemented by the single instruction

```
INC a
```

rather than by a sequence of instructions

```
MOV a R0
ADD #1 R0
MOV R0 a
```

Register Allocation:

* Instructions involving register operands are usually shorter and faster than those involving operands in memory.
* Therefore efficient utilization of register is important in generating good code.
* The use of register is divided into two sub programs.
  1) During register allocation
     ↳ set of variables that will reside in register are selected.

2) During register assignment
   ↳ The specific register that a variable will reside in is picked.

* Certain machine require register pairs for some operands and results.
* For eg) Integer multiplication and division involve register pairs
* The multiplication instruction of the form.

  MUL x, y

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

  y → the multiplier is a single register

  → The product occupies the entire even/odd register pair.

* The division instruction of the form

  DIV x y

  → 64 bit divident occupies an odd/even register pair, whose even register is x. y is a divisor.

  → After division even register holds the remainder and odd register holds the quotient.

Choice of Evaluation order:

* The order in which computations are performed can affect the efficiency of the target program
* Picking a best order is NP-complete problem.
* We shall avoid the problem by generating the code for the three address statements in the order in which they have been produced by the intermediate code generator.

TARGET MACHINE:

* The target computer for producing the target code is a byte addressable machine with four bytes to a word and n-general purpose registers $R_0, R_1, \ldots R_{N-1}$. It has two address instructions of the form

op source destination

in which

op → is an opcode

source and destination → are data fields

* It has the following opcodes,

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

* The source and destination of an instruction are specified by combining registers and memory locations with address modes.
* The address modes together with their assembly language forms and associated costs are as follows:

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
Indexed	C(R)	C+content(R)	1
Indirect Register	*R	contents(R)	0
indirect indexed	*C(R)	contents(C+content(R))	1
literal	#C	constant C	1

Example:

MOV R0, M

→ Stores the contents of register R0 into memory location M.

MOV #(R0) M

→ stores the value #+contents(R0) into memory location M

Instruction Cost:

* The cost of an instruction to be one plus the cost associated with the source and destination address modes. (indicated as added cost in the table for address modes above).

* Example:

* The statement a = b+c can be implemented by many different instruction sequences.

1. MOV b, R0
   ADD c, R0
   MOV R0, a
   cost = 6

2. MOV b, a
   ADD c, a
   cost = 6

Assuming R0, R1, and R2 contain the addresses of a, b, and c respectively

3. MOV *R1 *R0
   ADD *R2 *R0
   cost = 2

Assuming R_1 and R_2 contain the values of b and c respectively, and that the value of b is not needed after the assignment

4) ADD R2 R1
   MOV R1 a