*) Issues in Design of a code generator :

In code generator phase, various issues can arises:

1) Input to the code generator
2) Target program
3) Memory management
4) Instruction selection
5) Register allocation
6) Evaluation order

1) Input to the code generator :

- The input to the code generator contains the intermediate representation of source program and the information of the symbol table.
- Intermediate representation has several choices :
  - a) postfix notation.
  - b) syntax tree
  - c) Three address code
- we assume front end produces low-level intermediate representation. i.e., values of names in it can directly manipulated by machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2) Target program :

The target program is the output of the code generator. The output can be :

a) Assembly language :

It allows subprogram to be separately compiled.

b) Relocatable machine lang :

It makes the process of code generation easier.

c) Absolute machine lang :

It can be placed in a fixed location in memory and can be executed immediately.

3) Memory management :

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in source program to address of data is co-operating done by front end and code generator.
- Local variables are stack allocation in activation record while global variables are in static area.

4) Instruction selection :

- Nature of instruction set to the target machine should be complete and uniform.
- when you consider efficiency of target machine then the instruction speed and machine idioms are imp factors.
- The quality of generated code can be determined by its speed and size.

Eg: The 3 address code is :

a := b + c

d := a + e

Assembly code is :

```
MOV  b, R0     R0 -> b
ADD  c, R0     R0 -> c + R0
MOV  R0, a     a -> R0
MOV  a, R0     R0 -> a
ADD  e, R0     R0 -> e + R0
MOV  R0, d     d -> R0
```

5) Register allocation :

The Registers can be accessed faster than memory.

The following sub-problems arise when we use registers:

Register allocation :

In Register allocation, we select the set of variables that will reside in registers.

Register assignment :

In Register assignment, we pick the register that contains variables.

Eg: $t = a+b$

$t = t * c$

$t = t / d$

```
MOV   a, R0
ADD   b, R0
MUL   c, R0
DIV   d, R0
MOV   R0, t
```

6) Evaluation order :

The code generator determines the order in which instructions are executed. The target code efficiency is influenced by order of computations.

*) Object code forms :

Assume that target compute uses following instructions

1) Load operations :

Load memory word to register. i.e., LD R_1 , X (Load the value in location x to register.

2) Store operation :

store register to memory i.e., ST X, R_1 (store the value in Register R_1 into location x).

3) Computational operations :

It is in form of OP dst , src1, src2
where op is operator like ADD, SUB, ---
dst, src1, src2, are locations or registers

Eg: ADD R_1 , R_2, R_3 $(R_1 = R_2 + R_3)$
INC X

4) unconditional Jump :

It means without checking any conditon, the control will be transferred to corresponding location.

It is in form BR L.

5) conditional Jump :

It means checking the condition, the control will be transferred to corresponding location.

If the condition is true then it will goes to the corres-ponding location otherwise next statement will be executed.

BLTZ r, L

where, B → Branch

LT → less than

Z → zero

r → register.

L → location

Assume that our target machine has variety of addressing modes.

Addressing mode	Form	Address
Absolute	M	M
Register	R	R
Indexed	c(R)	c + content(R)
Indirect - register	*R	content (R)
Indirect - indexed	* c (R)	content(c+ content(R))
literal (or)	#	N/A.

*) code generation algorithm :

The algorithm follows a queue of three address statements. The address statements are of form x : y op z simulates specific tasks.

The task description is given with the help of following algorithm :

1). The initial step is to invoke a function getreg( ) to capture the location L where the optimization of y op z is stored.

2) Manage the descriptive address of b to determine b'. If the value of b is present in memory and registers both , then the value of b' will be considered. If there is the absence of a value of b in L , then there is a need to generate the instruction Mov b' , to copy a value of b in L.

3) A new instruction op c' is to be generated. updation of address description of a takes place to depict that a is stored in L. If a is already Present in L , then update the descripter and remove a from all other descriptors.

4) After reaching the stage where b or c have no further uses, alter the register description. Now the register will not store the values of b and c.

Eg: $w := (x-y) + (x-z) + (x-z)$

The three address codes :

$a := x-y$

$b = x-z$

$c = a+b$

$w = b+c$

code sequence for above problem is

statement	code generation	Register descriptor	Address descriptor
$a=x-y$	MOV x, R_0 SUB y, R_0	R_0 contains a	a in R_0
$b=x-z$	MOV x, R_1 SUB z, R_1	R_1 contains b R_0 contains a	a in R_0 b in R_1
$c=a+b$	ADD R_1, R_0	R_0 contains c R_1 contains b	c in R_0 b in R_1
$w=b+c$	ADD R_1, R_0 MOV R_0, w	R_0 contains w	w in R_0 w in R_0 & memory.

*) Register allocation and assignment :

[Register allocation is only within a basic block. It follows top-down approach.

Local register allocation :)

Instructions with register operands are faster than memory operands.

various strategies for register allocation and assignment

1) assign specific value in target program to certain register.

*) base addresses.

*) arthimatic computations

*) top of the stack.

1) Global Register allocation :

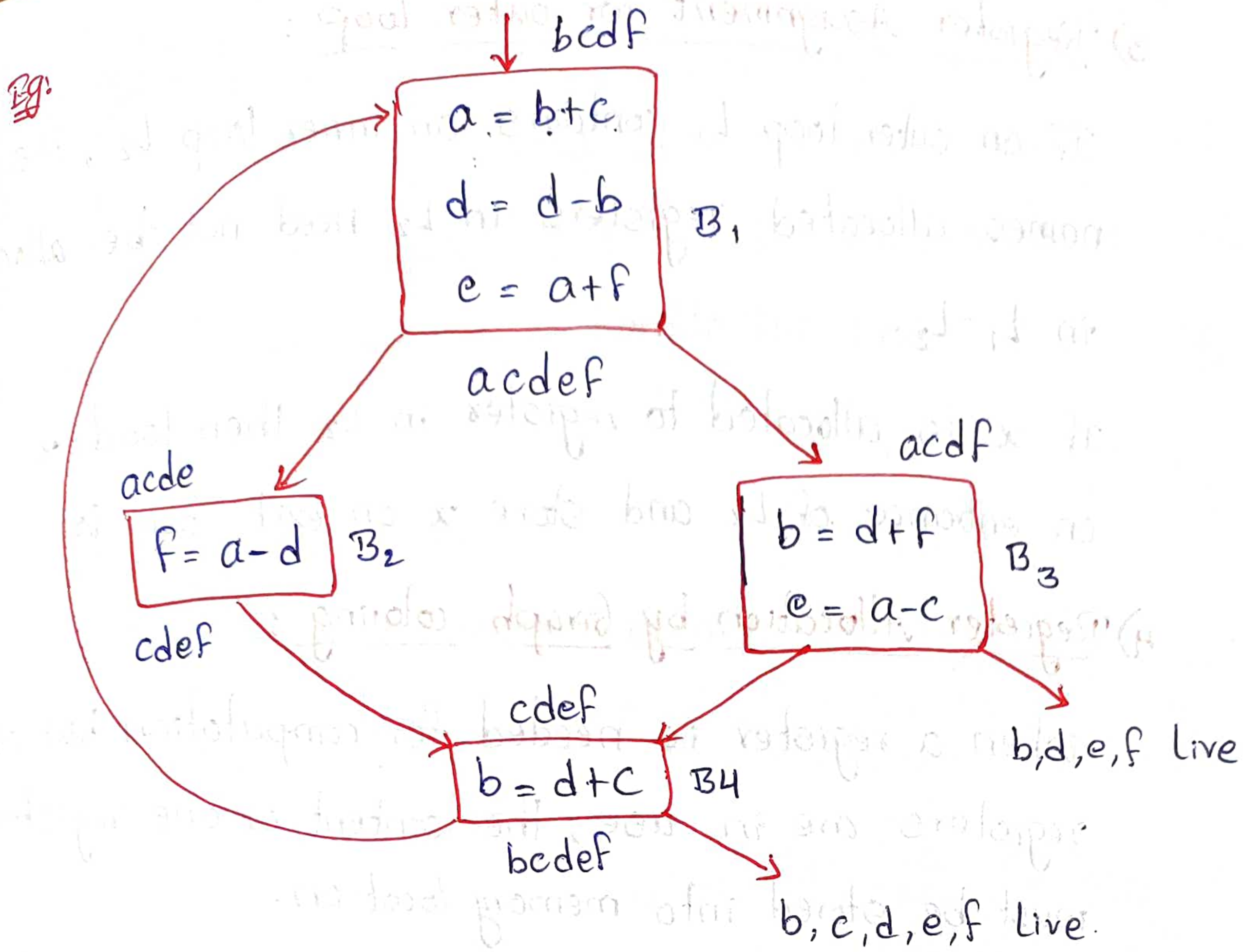
- keep frequently used value in a fixed register.
- Assign some fixed number of registers to hold most active values in each inner loop.

2) Usage count :

- Count a savings of one of each use of x in loop L
- If x is allocated a register, then count a savings of two for each block in L.

$$\Sigma \ use(x, B) + 2 * live(x, B)$$

Eg:



use $(a,B_1) + 2 * \text{live}(a,B_1) = 0 + (2*1) = 2$

" $(a,B_2) + 2 * \text{live}(a,B_2) = 1 + (2*0) = 1$

" $(a,B_3) + 2 *$ " $(a,B_3) = 1 + (2*0) = 1$

" $(a,B_4) + 2 *$ " $(a,B_4) = 0 + (2*0) = 0$

$\overline{\underline{4}}$

Similarly

$a = 4$

$b = 6$

$c = 3$

$d = 6$

$e = 4$

$f = 4$

R_0	R_1	R_2
b	d	a/e/f

4

U-5

3) Register Assignment for outer loop :

If an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocat in $L_1 - L_2$.

If x is allocated to register in L_2 then load x on entrance of L_2 and store x on exit from L_2.

4) Register Allocation by Graph coloring :

when a register is needed for computation but all registers are in use, the content of one register must be stored into memory location.

Two passes are used :

1) Target machine instructons are selected
2) a register - interference graph is constructed.