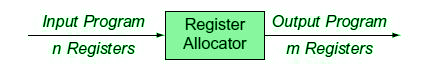
Code Generator

A code generator is expected to have an understanding of the target machine’s runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

* **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
* **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
* **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine’s instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
* **Register allocation** : A program has a number of values to be maintained during the execution. The target machine’s architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
* **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

# **Register Allocations in Code Generation**

[Registers](https://www.geeksforgeeks.org/registers-8085-microprocessor/) are the fastest locations in the memory hierarchy. But unfortunately, this resource is limited. It comes under the most constrained resources of the target processor. Register allocation is an NP-complete problem. However, this problem can be reduced to graph colouring to achieve allocation and assignment. Therefore a good register allocator computes an effective approximate solution to a hard problem.



**Figure –** Input-Output

The register allocator determines which values will reside in the register and which register will hold each of those values. It takes as its input a program with an arbitrary number of registers and produces a program with finite register set that can fit into the target machine.

**Allocation vs Assignment:**

**Allocation –**  
Maps an unlimited namespace onto that register set of the target machine.

* **Reg. to Reg. Model:** Maps virtual registers to physical registers but spills excess amount to memory.
* **Mem. to Mem. Model:** Maps some subset of the memory location to a set of names that models physical register set.

Allocation ensures that code will fit the target machine’s reg. set at each instruction.  
**Assignment –**  
Maps an allocated name set to physical register set of the target machine.

* Assumes allocation has been done so that code will fit into the set of physical registers.
* No more than **‘k’** values are designated into the registers, where ‘k’ is the no. of physical registers.

**General register allocation is a NP complete problem:**

* Solved in polynomial time, when (no. of required registers) <= (no. of available physical registers).
* An assignment can be produced in linear time using Interval-Graph Coloring.

# **Design Issues**

In the code generation 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 the source program and the information of the symbol table. The source program is produced by the front end.
* Intermediate representation has the 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 the 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 language:** It makes the process of code generation easier.

c) **Absolute machine language:** 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 the source program to address of data is co-operating done by the front end and code generator.
* Local variables are stack allocation in the activation record while global variables are in static area.

## 4. Instruction selection:

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

### **Example:**

The Three address code is:

1. a:= b + c
2. d:= a + e

Inefficient assembly code is:

1. MOV b, R0              R0→b
2. ADD c, R0   R0      c + R0
3. MOV R0, a               a   →   R0
4. MOV a, R0      R0→  a
5. ADD e, R0               R0  →       e + R0
6. MOV R0, d               d    →  R0

## 5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

**Register allocation:** In register allocation, we select the set of variables that will reside in register.

**Register assignment:** In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

### **For example:**

Consider the following division instruction of the form:

1. D x, y

Where,

**x** is the dividend even register in even/odd register pair

**y** is the divisor

**Even register** is used to hold the reminder.

**Old register** is used to hold the quotient.

## 6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

# **Data flow analysis**

* To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
* Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
* For this kind of optimization user defined chaining is one particular problem.
* Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

1. x = a + b;
2. x = 6 \* 3
3. In this code, the first assignment of x is useless. The value computer for x is never used in the program.
4. At compile time the expression 6\*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs more global information. For example, consider the following code:

1. a = 1;
2. b = 2;
3. c = 3;
4. if (....) x = a + 5;
5. else x = b + 4;
6. c = x + 1;

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7. But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

* Which variables are guaranteed to have constant values
* Which variables will be used before being redefined

Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

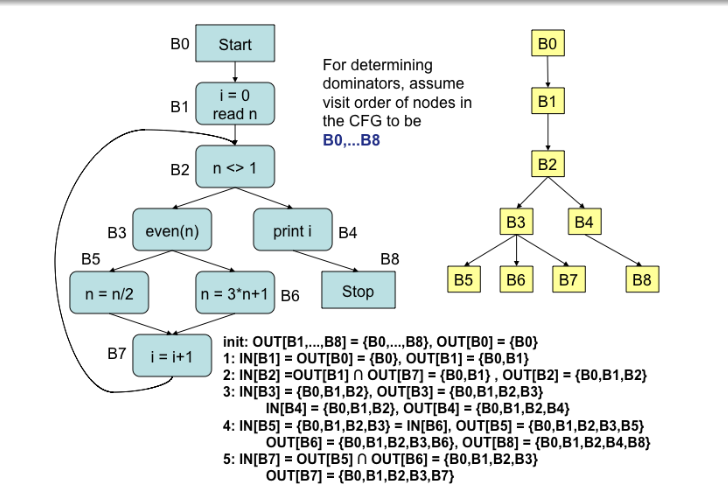
The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

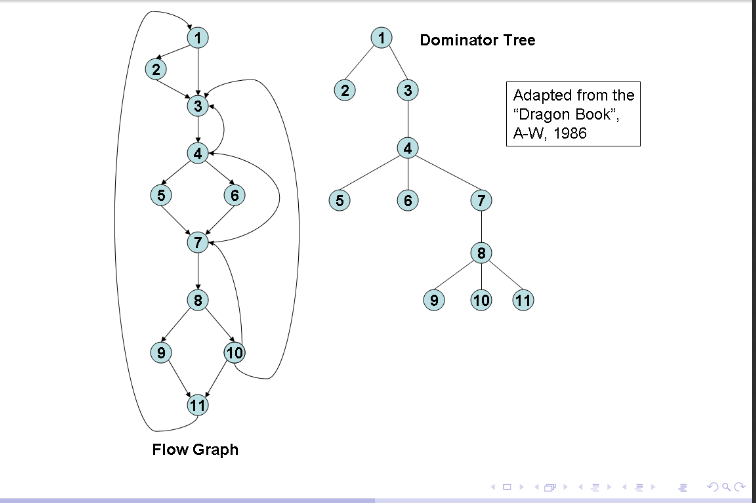
**CONTROL FLOW ANALAYSIS**

Control-flow analysis (CFA) helps us to understand thestructure of control-flow graphs (CFG)To determine the loop structure of CFGsFormulation of conditions for code motion use dominatorinformation, which is obtained by CFAConstruction of the static single assignment form (SSA)requires dominance frontier information from CFAIt is possible to use interval structure obtained from CFA tocarry out data-flow analysisFinding Control dependence, which is needed inparallelization, requires CFA

We say that a nodedin a flow graphdominatesnoden,writtend dom n, if every path from the initial node of theflow graph tongoes throughdInitial node is the root, and each node dominates only itsdescendents in the dominator tree (including itself)The nodex strictly dominates y, ifxdominatesyandx6=yxis theimmediate dominatorofy(denotedidom(y)), ifxis the closest strict dominator ofyAdominator treeshows all the immediate dominatorrelationshipsPrinciple of the dominator algorithmIfp1,p2,...,pk, are all the predecessors ofn, andd6=n,thend dom n, iff d dom pi for each i

Example:





**Simple code generator:**

A code generator generates target code for a sequence of three- address statements and

effectively uses registers to store operands of the statements.

• For example: consider the three-address statement a := b+c It can have the following sequence of codes:

ADD Rj, Ri Cost = 1

(or)

ADD c, Ri Cost = 2

(or)

MOV c, Rj Cost = 3

ADD Rj, Ri

**Register and Address Descriptors:**

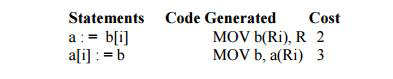
• A register descriptor is used to keep track of what is currently in each registers. The register

descriptors show that initially all the registers are empty.

• An address descriptor stores the location where the current value of the name can be found at run time.

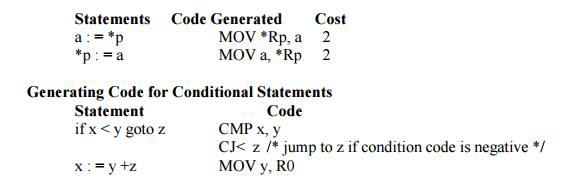
**Generating Code for Indexed Assignments**

The table shows the code sequences generated for the indexed assignmen a:= b[ i ] and a[ i ]:= b



**Generating Code for Pointer Assignments**

The table shows the code sequences generated for the pointer assignments a : = \*p and \*p : = a



if x < 0 goto z ADD z, R0

MOV R0,x

CJ< z

**GENERATING CODE FROM DAGs**

The advantage of generating code for a basic block from its dag representation is that from a dag we can easily see how to rearrange the order of the final computation sequence than we can start from a linear sequence of three-address statements or quadruples.

**Rearranging the order**

The order in which computations are done can affect the cost of resulting object code. For example, consider the following basic block:

t1 : = a + b

t2 : = c + d

t3 : = e - t2

t4 : = t1 - t3

**Generated code sequence for basic block:**

MOV a , R0

ADD b , R0

MOV c , R1

ADD d , R1

MOV R0 , t1

MOV e , R0

SUB R1 , R0

MOV t1 , R1

SUB R0 , R1

MOV R1 , t4

**Rearranged basic block:**

Now t1 occurs immediately before t4.

t2 : = c + d

t3 : = e - t2

t1 : = a + b

t4 : = t1 - t3

**Revised code sequence:**

MOV c , R0

ADD d , R0

MOV a , R0

SUB R0 , R1

MOV a , R0

ADD b , R0

SUB R1 , R0

MOV R0 , t4

In this order, two instructions **MOV R0 , t1 and MOV t1 , R1** have been saved.

**Example: Consider the DAG shown below**

Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3). Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6). Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that. The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t8 : = d + e

t6 : = a + b

t5 : = t6 - c

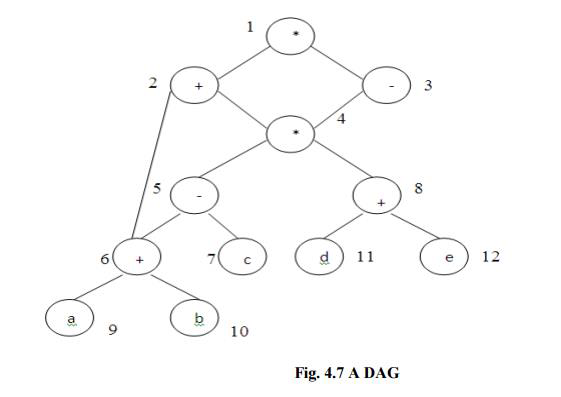
t4 : = t5 \* t8

t3 : = t4 - e

t2 : = t6 + t4

t1 : = t2 \* t3

This will yield an optimal code for the DAG on machine whatever be the number of registers.



**Fig. 4.7 A DAG**