Module-3

Code Generation: Factors involved in code generation:

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

The following issue arises during the code generation phase:

1. Input to code generator –
   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.
2. Target program –
   The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
   - Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
   - Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
   - Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.
3. Memory Management –
   Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. Instruction selection –
   Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.
   For example, the respective three-address statements would be translated into the latter code sequence as shown below:

   ```
   P:=Q+R
   S:=P+T


   MOV Q, R0
   ADD R, R0
   MOV R0, P
   MOV P, R0
   ADD T, R0
   MOV R0, S
   ```

   Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.
5. Register allocation issues –
   Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:
   1. During Register allocation – we select only those set of variables that will reside in the registers at each point in the program.
   2. During a subsequent Register assignment phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

```
M a, b
```

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

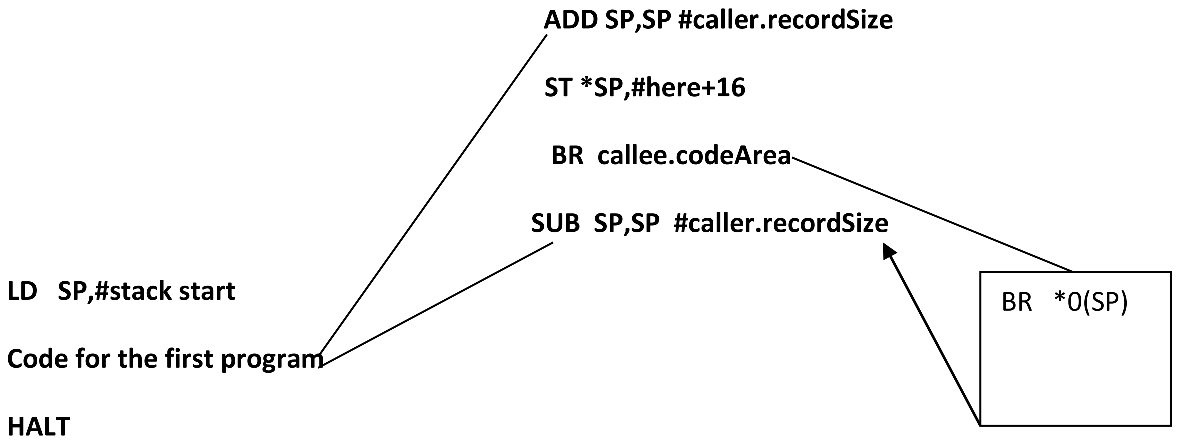
6. Evaluation order –
   The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.
7. Approaches to code generation issues: Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

Simple code generation using stack allocation in compiler design:

In stack allocation a new activation record is pushed onto the stack for each execution of the procedure. The record is popped when the activation ends. Stack allocation• Static allocation can become stack allocation by using relative addresses for storage in activation records.

- The position of the activation record is not know until runtime
- Must use relative address to access elements of the activation record
- We need a register to keep track of the top of the stack



```
ADD SP,SP #caller.recordSize
ST *SP,#here+16
BR  callee.codeArea
SUB  SP,SP  #caller.recordSize

LD   SP,#stack start
Code for the first program
HALT

BR   *0(SP)
```

Assumptions:

- First word in each activation is the return address
- start address of p, q, and m: 100, 200, and 300

• stack starts at 600

```
action1        //code for m
call  q
action2
halt

action3        //code for p
return

action4      //code for q
call p
action5
call  q
action5
call  q
return
```

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

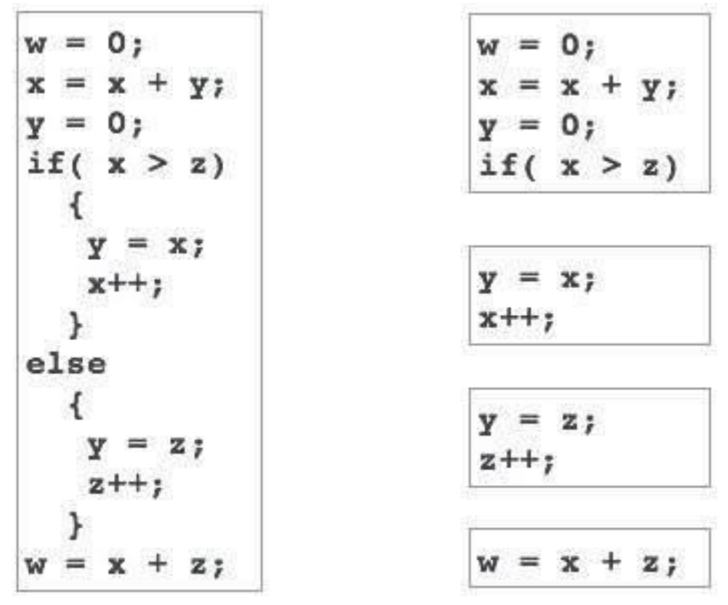
- **Search header statements of all the basic blocks from where a basic block starts:**

- ○ **First statement of a program.**
  - ○ **Statements that are target of any branch (conditional/unconditional).**
  - ○ **Statements that follow any branch statement.**

- **Header statements and the statements following them form a basic block.**

- **A basic block does not include any header statement of any other basic block.**

Basic blocks are important concepts from both code generation and optimization point of view.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

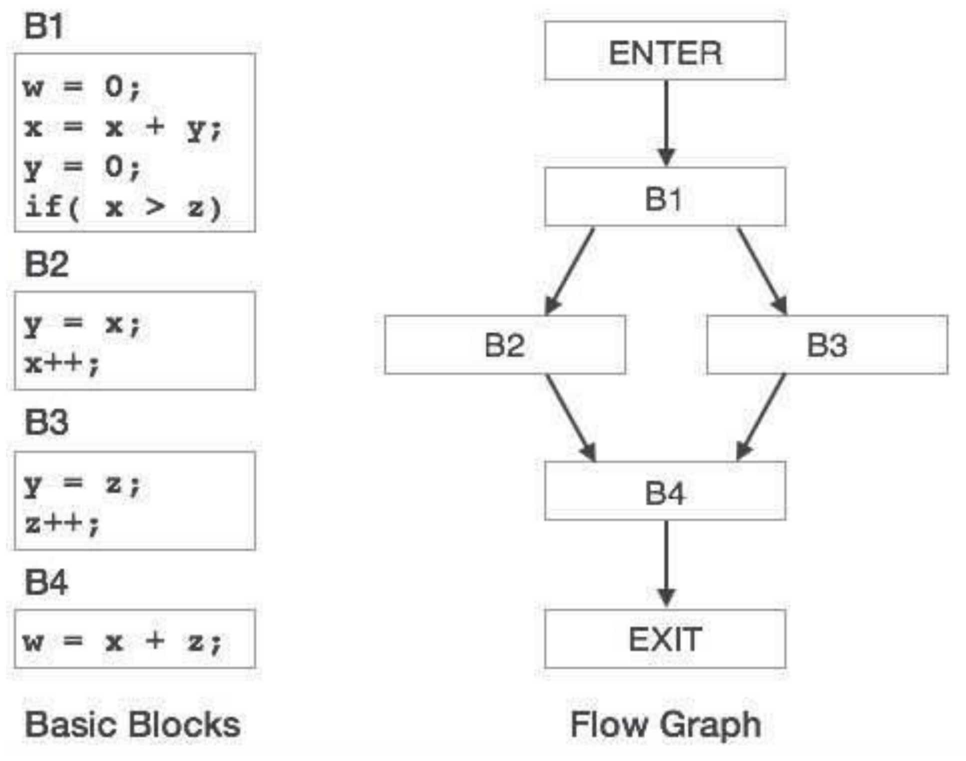
```
w = x + z;
```

Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.


B1
```
w = 0;
x = x + y;
y = 0;
if( x > z)
```
B2
```
y = x;
x++;
```
B3
```
y = z;
z++;
```
B4
```
w = x + z;
```

Basic Blocks

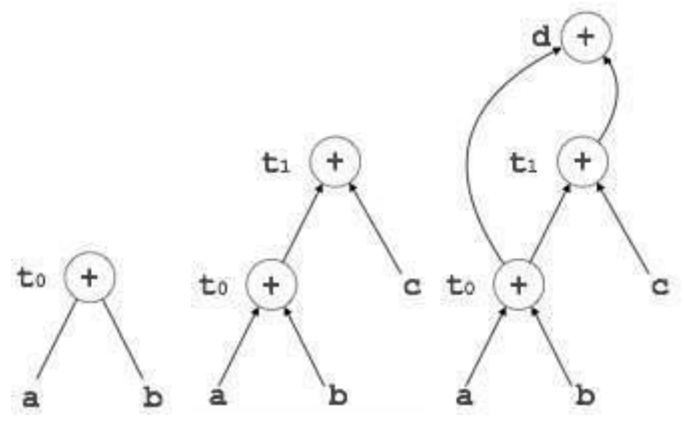
Flow Graph

Directed Acyclic Graph:(DAG)

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- **Leaf nodes represent identifiers, names or constants.**
- **Interior nodes represent operators.**
- **Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.**
- **Example:**

- $t_0 = a + b$
- $t_1 = t_0 + c$
- $d = t_0 + t_1$



Code Optimization:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

n optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- **The output code must not, in any way, change the meaning of the program.**
- **Optimization should increase the speed of the program and if possible, the program should demand less number of resources.**
- **Optimization should itself be fast and should not delay the overall compiling process.**

Efforts for an optimized code can be made at various levels of compiling the process.

- **At the beginning, users can change/rearrange the code or use better algorithms to write the code.**
- **After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.**
- **While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.**

Optimization can be categorized broadly into two types : machine independent and machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
   item = 10;
   value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
   value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
   {
   int y, z;
   y = 10;
   z = x + y;
   return z;
   }
```

```
int add_ten(int x)
   {
   int y;
   y = 10;
   y = x + y;
   return y;
   }
```

```
int add_ten(int x)
   {
   int y = 10;
   return x + y;
   }
```

```
int add_ten(int x)
   {
   return x + 10;
   }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- **MOV x, R0**
- **MOV R0, R1**

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code:

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
   return x + 10;
   printf("value of x is %d", x);
}
```

In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

Flow of control optimization:

here are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :   INC R1
```

Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.**

- **Induction analysis : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.**

- **Strength reduction : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.**

Dead-code Elimination

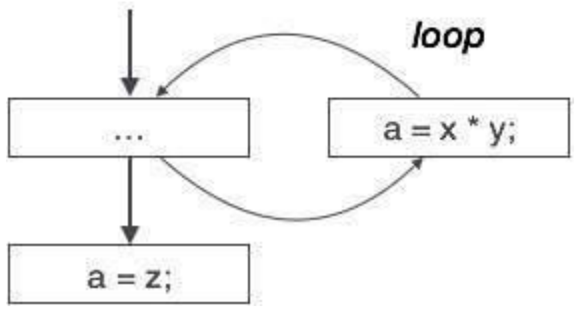
Dead code is one or more than one code statements, which are:

- **Either never executed or unreachable,**
- **Or if executed, their output is never used.**

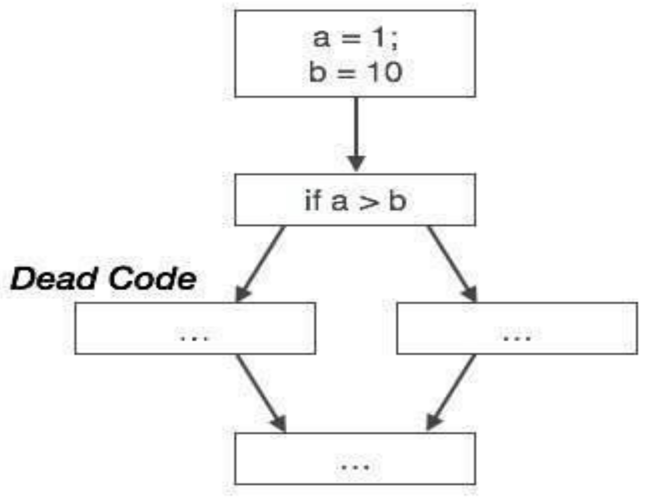
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



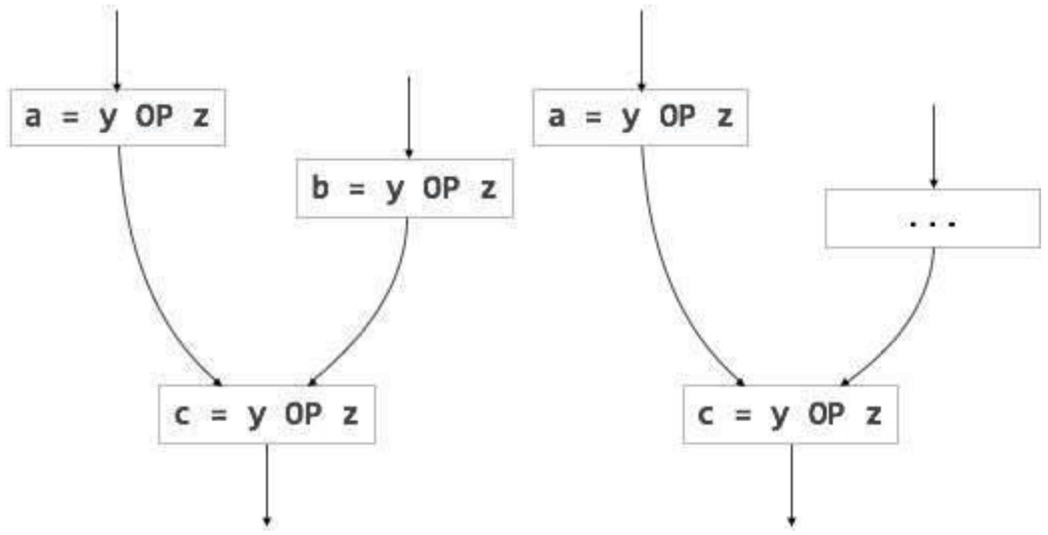
The above control flow graph depicts a chunk of program where variable 'a' is used to assign the output of expression 'x * y'. Let us assume that the value assigned to 'a' is never used inside the loop.Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.



Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
   a = y OP z;
}
else
{
   ...
}
c = y OP z;
```

We assume that the values of operands (y and z) are not changed from assignment of variable a to variable c. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
   ...
   tmp = y OP z;
   a = tmp;
   ...
}
else
{
   ...
   tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

Common subexpression elimination

common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value

In the following code:

```
a = b * c + g;
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

if the cost of storing and retrieving `tmp` is less than the cost of calculating `b * c` an extra time.

The possibility to perform CSE is based on available expression analysis (a data flow analysis). An expression `b*c` is available at a point *p* in a program if:

- **every path from the initial node to p evaluates `b*c` before reaching *p*,**
- **and there are no assignments to `b` or `c` after the evaluation but before *p*.**

The cost/benefit analysis performed by an optimizer will calculate whether the cost of the store to `tmp` is less than the cost of the multiplication; in practice other factors such as which values are held in which registers are also significant.

Compiler writers distinguish two kinds of CSE:

- **local common subexpression elimination works within a single basic block**
- **global common subexpression elimination works on an entire procedure,**