MODULE - V

CODE OPTIMIZATION AND GENERATION

SYLLABUS:

Code Optimization - Principal sources of optimization, Machine dependent and machine independent optimizations, Local and global optimizations. Code generation - Issues in the design of a code generator, Target Language, A simple code generator.

INTRODUCTION

- The code generated by the compiler can be made faster or take less space or both. For that, some transformations can be applied on the code, and is called optimization or optimization transformations.
- Compilers that can perform optimizations are called optimizing compilers.
- Code optimization is an optional phase and it must not change the meaning of the program.
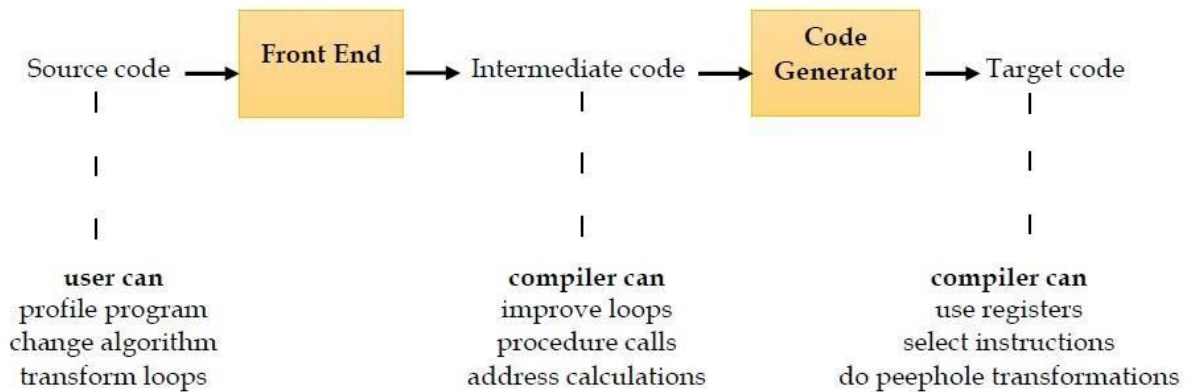
Need For Optimization Phase in Compiler

- Code produced by a compiler may not be perfect in terms of execution speed and memory space.
- Optimization by hand takes much more effort and time.
- Machine level details like instructions and addresses are not known to the programmer.
- Advanced architecture features like instruction pipeline requires optimized code.
- Structure reusability and maintainability of the code are improved.

Criteria for Code Optimization

- It should preserve the meaning of the program i.e., it should not change the output or produce error for a given input.
- Eventually it should improve the efficiency of the program by a reusable amount. Sometimes it may increase the size of the code or may slow the program slightly but it should improve the efficiency.
- It must be worth with the effort, i.e., the effort put on optimization must be worthy when compared with the improvement.

Optimization can be applied in 3 places.

- Source code
- Intermediate code
- Target code

Source code → **Front End** → Intermediate code → **Code Generator** → Target code

user can
profile program
change algorithm
transform loops

compiler can
improve loops
procedure calls
address calculations

compiler can
use registers
select instructions
do peephole transformations

Types of Optimization:

There are two types of optimization:

1. Machine dependent optimization – run only in particular machine.
2. Machine independent optimization – used for any machine.

Optimization can be done in 2 phases:

1. Local optimization:

Transformations are applied over a small segment of the program called basic block, in which statements are executed in sequential order.

2. Global optimization

Transformations are applied over a large segment of the program like loop, procedures, functions etc. Local optimization must be done before applying global optimization.

Basic Block

Basic block is a sequence of consecutive 3 address statements which may be entered only at the beginning and when entered statements are executed in sequence without halting or branching.

To identify basic block, we have to find **leader** statements. Rules for leader statements are

Algorithm 8.5 : Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

Flow Graphs

+ It is the pictorial representation of control flow analysis in a program. It shows the relationship among basic blocks.

+ Nodes are basic blocks and edges are control flow. It is a directed graph $G = (N, E\ n_0)$.

Where,   N – set of basic blocks

E – set of control flows

n_0 - starting node

+ If there is a directed edge from B1 to B2, the control transfers from the last statement of B1 to the first statement of B2. B1 is called predecessor of B2 and B2 is successor of B1.

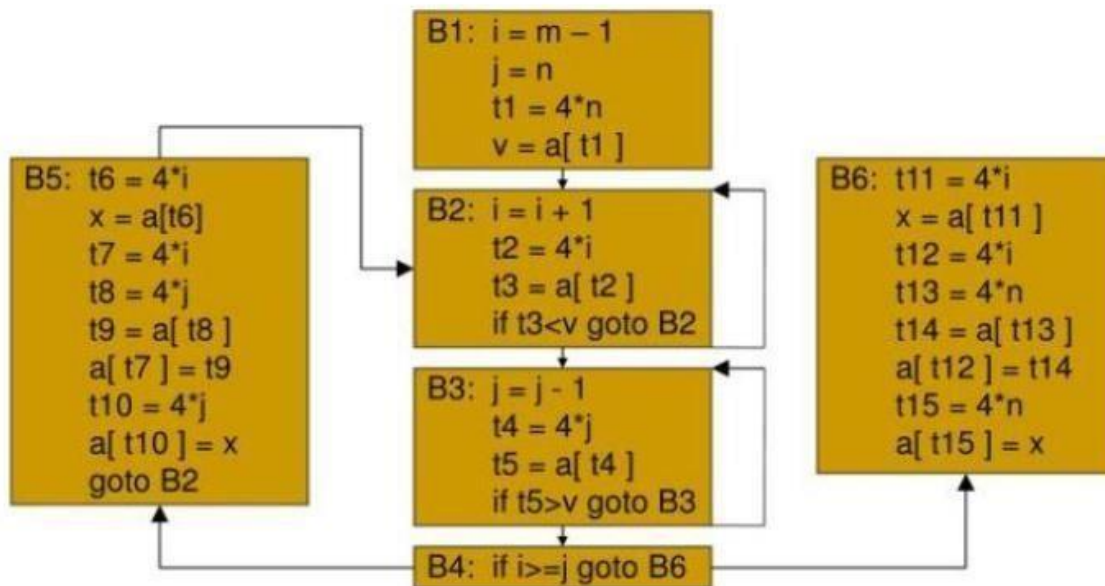**Consider the code for quick sort**

```
void quicksort(m, n)
int m, n;
{
   int I, j;
   if (n <= m ) return;
   /* fragment begins here */
   i = m-1; j = n; v = a[n];
   while(1)   {
      do i = i+1; while( a[i] < v );
         do j = j-1; while( a[j] > v );
         if( i >= j ) break;
         x = a[i]; a[i] = a[j];    a[j] = x;
      }
   x = a[i]; a[i] = a[n]; a[n]= x;
   /* fragment ends here */
   quicksort(m, j); quicksort(i+1, n);
}
```

Three address code for quick sort fragment

1	$i = m - 1$		16	$t_7 = 4 * I$
2	$j = n$		17	$t_8 = 4 * j$
3	$t_1 = 4 * n$		18	$t_9 = a[t_8]$
4	$v = a[t_1]$		19	$a[t_7] = t_9$
5	$i = i + 1$		20	$t_{10} = 4 * j$
6	$t_2 = 4 * i$		21	$a[t_{10}] = x$
7	$t_3 = a[t_2]$		22	goto (5)
8	if $t_3 < v$ goto (5)		23	$t_{11} = 4 * I$
9	$j = j - 1$		24	$x = a[t_{11}]$
10	$t_4 = 4 * j$		25	$t_{12} = 4 * i$
11	$t_5 = a[t_4]$		26	$t_{13} = 4 * n$
12	if $t_5 > v$ goto (9)		27	$t_{14} = a[t_{13}]$
13	if $i >= j$ goto (23)		28	$a[t_{12}] = t_{14}$
14	$t_6 = 4 * i$		29	$t_{15} = 4 * n$
15	$x = a[t_6]$		30	$a[t_{15}] = x$

Flow graph for quick sort

```
                          ┌─────────────────────┐
                          │ B1:  i = m − 1      │
                          │      j = n          │
                          │      t1 = 4*n       │
                          │      v = a[ t1 ]    │
                          └─────────────────────┘
                                    │
┌──────────────────┐      ┌─────────────────────┐      ┌──────────────────────┐
│ B5: t6 = 4*i     │      │ B2:  i = i + 1      │      │ B6: t11 = 4*i        │
│     x = a[t6]    │      │      t2 = 4*i       │      │     x = a[ t11 ]     │
│     t7 = 4*i     │      │      t3 = a[ t2 ]   │      │     t12 = 4*i        │
│     t8 = 4*j     │      │      if t3<v goto B2│      │     t13 = 4*n        │
│     t9 = a[ t8 ] │      └─────────────────────┘      │     t14 = a[ t13 ]   │
│     a[ t7 ] = t9 │      ┌─────────────────────┐      │     a[ t12 ] = t14   │
│     t10 = 4*j    │      │ B3:  j = j - 1      │      │     t15 = 4*n        │
│     a[ t10 ] = x │      │      t4 = 4*j       │      │     a[ t15 ] = x     │
│     goto B2      │      │      t5 = a[ t4 ]   │      └──────────────────────┘
└──────────────────┘      │      if t5>v goto B3│
                          └─────────────────────┘
                          ┌─────────────────────┐
                          │ B4:  if i>=j goto B6│
                          └─────────────────────┘
```

Principle Sources of Optimization

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

1. **Function preserving transformations**
   a. Common subexpression elimination
   b. Copy propagation
   c. Dead code elimination
   d. Constant folding

2. **Loop optimization**
   a. Code motion
   b. Induction variable elimination
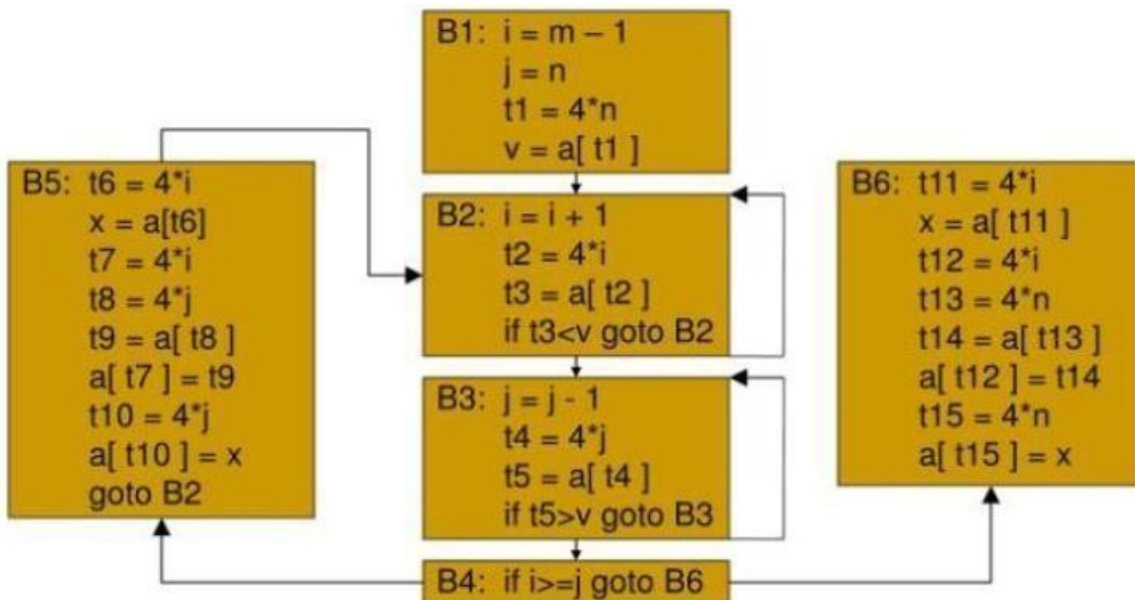   c. Reduction in strength

Function-Preserving Transformations

+ There are a number of ways in which a compiler can improve a program without changing the function it computes.

+ Some function preserving transformations examples are given below

Common Sub Expression Elimination (CSE)

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.
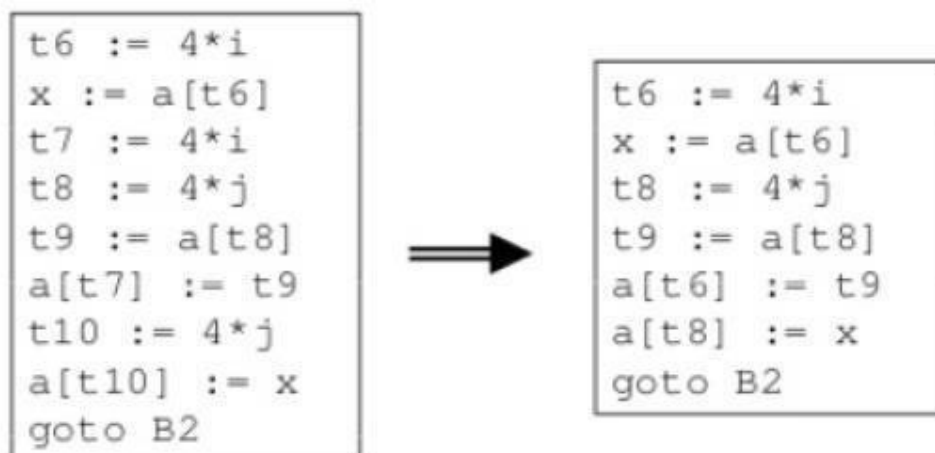
We can avoid re-computing the expression if we can use the previously computed value.

- Consider the flow graph of quick sort fragment

B1: i = m − 1
j = n
t1 = 4*n
v = a[t1]

B5: t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2

B2: i = i + 1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2

B3: j = j - 1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3

B4: if i>=j goto B6

B6: t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x

EXAMPLE 1

B₅

```
t6  :=  4*i
x  :=  a[t6]
t7  :=  4*i
t8  :=  4*j
t9  :=  a[t8]
a[t7]  :=  t9
t10  :=  4*j
a[t10]  :=  x
goto B2
```

⟹

```
t6  :=  4*i
x  :=  a[t6]
t8  :=  4*j
t9  :=  a[t8]
a[t6]  :=  t9
a[t8]  :=  x
goto B2
```

+ Assignments of the form **f : = g** called copy statements, or copies for short. The idea behind the **copy-propagation** transformation is to use **g** for **f**, whenever possible after the copy statement f: = g.

+ Copy propagation means use of one variable instead of another. Copy statements introduced during common subexpression elimination.

EXAMPLE 1

The assignment **x : = t_3** in block **B$_5$** of Flow graph is a copy.

This change may not appear to be an improvement, but it gives us the opportunity to eliminate the assignment to x.

ADVANTAGE

One advantage of copy propagation is that it often turns the copy statement into dead code.

Dead-Code Elimination

+ A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.

+ A related idea is dead or useless code, statements that compute values that never get used.

+ While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

EXAMPLE 1

Consider B5 of flow graph.

B5: x = t3
a[t2] = t5
a[t4] = t3
goto B2

Copy propagation followed by dead-code elimination removes the assignment to x and transforms into:

a[t2] = t5
a[t4] = t3
goto B2

EXAMPLE 2

```
i=0;
if(i==1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Note that in this example, the "if" statement is a dead code, whereas, "a=b+5" is an unreachable code.

Constant Folding

  - If all operands are constants in an expression, then it can be evaluated at compile time itself. The result of the operation can replace the original evaluation in the program.

  - This will improve the run time performance and reducing code size by avoiding evaluation at compile- time.

EXAMPLE

a=3.14157/2 can be replaced by a=1.570 thereby eliminating a division operation.

Loop Optimization

  - The running time of a program may be improved if the number of instruction in an inner loop is decreased, even if we increase the amount of code outside the loop.

Loop optimization techniques are:

- Code Motion (or Code movement)
- Induction Variables
- Strength Reduction
- Loop jamming (merging/combining)
- Loop unrolling

Code Motion

+ An important modification that decreases the amount of code in a loop is code motion.

+ Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times

+ Ex: Loop optimization – loop invariant code motion

+ A fragment of code that resides in the loop and computes the same value of each iteration is called loop invariant code.

EXAMPLE 1

```
for i = 1 to 100 begin

{

z := 1;

x := 25 * a ;

y := x + z ;

end;

}
```

→

```
x := 25 * a ;

for i = 1 to 100 begin

{

z := 1;

y := x + z ;

end;

}
```

Here x := 25 * a ; is a loop variant. Hence in the optimised program it is computed only once before entering the for loop. y := x + z ; is not loop invariant. Hence it cannot be subjected to frequency reduction.

EXAMPLE 2

Evaluation of limit-2 is a loop-invariant computation in the following while-statement:
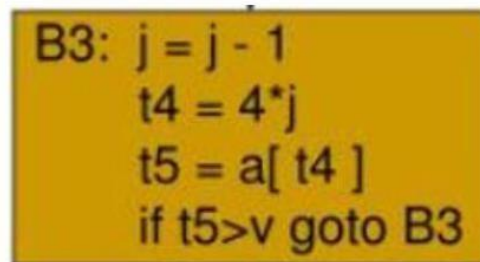
> while (i <= limit - 2) /* statement does not change limit*/

Code motion will result in the equivalent of

> t= limit - 2;
> while (i<=t) /* statement does not change limit or t */

Induction Variables

- Loops are usually processed inside out. For example consider the loop around B3.

- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

Reduction In Strength

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

EXAMPLE

As the relationship t4:=4*j surely holds after such an assignment to t4 in Figure. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold.

We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time.

Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Figure
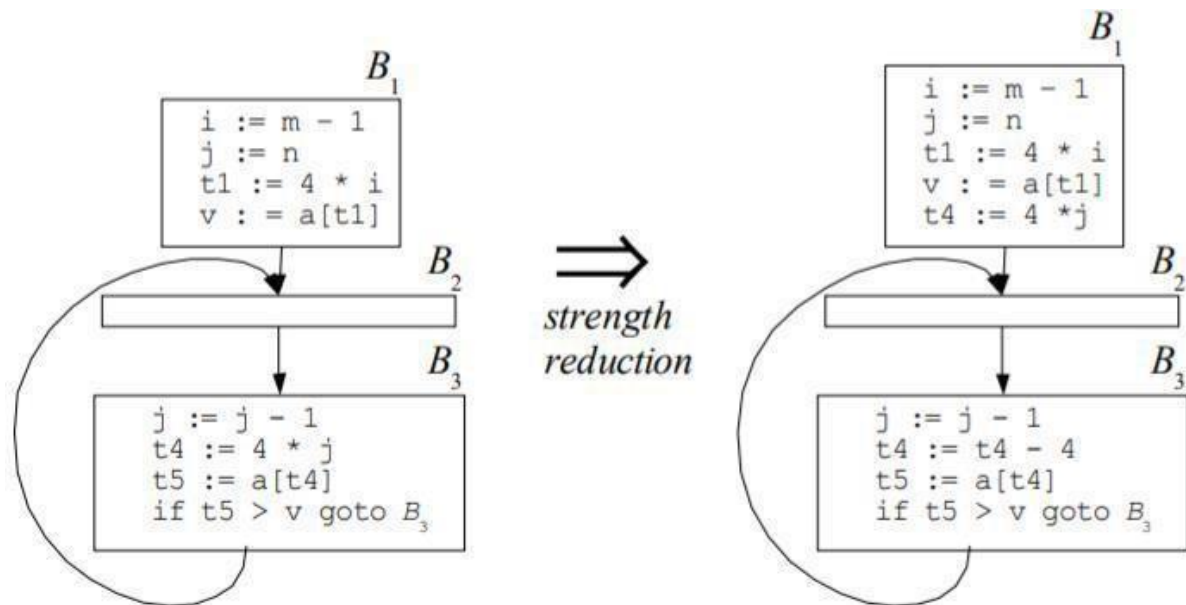
▼ The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Loop Jamming:

If the operations performed can be done in a single loop then, merge or combine the loops.

```
// Program with multiple loops
int main()
{
    for (i = 0; i < n; i++)
        A[i] = i + 1;
    for (j = 0; j < n; j++)
        B[j] = j - 1;

}

return 0;

}
```

The above program code can be converted to,

```
// Program with one loop when multiple loops are merged
int main()
{
    for (i = 0; i < n; i++) {
        A[i] = i + 1;
        B[i] = i - 1;
    }
    return 0;
}
```

Loop Unrolling:

If there exists, simple code which can reduce the number of times the loop executes then, the loop can be replaced with these codes.

```
// Program with loops
int main()
{
    for (i = 0; i < 3; i++)
        cout << "Cd";
    return 0;
}
```

This can be converted to,

```
// Program with simple code without loops
int main()
{
    cout << "Cd";
    cout << "Cd";
    cout << "Cd";
    return 0;
}
```

Optimization of Basic Blocks

There are two types of basic block optimizations:
1. Structure preserving transformations
2. Algebraic transformations

(Note: In **structure preserving transformations**, write about:

- Dead code elimination
- Copy propagation (variable propagation and constant propagation)
- Common sub expression elimination
- Strength reduction
- Constant folding
- Interchange of two independent statements)

Copy Propagation:

It is of two types, Variable Propagation, and Constant Propagation.

Variable Propagation:

$$x = y \qquad \Rightarrow z = y + 2 \ (Optimized\ code)$$

$$z = x + 2$$

Constant Propagation:

$$x = 3 \qquad \Rightarrow z = 3 + a \ (Optimized\ code)$$

$$z = x + a$$

Strength Reduction:

Replace expensive statement/ instruction with cheaper ones.

```
x = 2 * y (costly)   ⇒   x = y + y (cheaper)
x = 2 * y (costly)   ⇒ x = y << 1 (cheaper)
```

Constant Folding:

Solve the numeric expressions and use the result in its place, so that the compiler need not calculate the result all the time.

Solve the constant terms which are continuous so that compiler does not need to solve this expression.

Example:

```
x = 2 * 3 + y    ⇒ x = 6 + y   (Optimized code)
```

Algebraic Transformations:

+ It represents another important class of optimizations on basic blocks.

$$x + 0 = 0 + x = x$$
$$x - 0 = x$$
$$x * 1 = 1 * x = x$$
$$x / 1 = x$$

+ Another class of algebraic optimization includes reduction in strength.

$$x ** 2 = x * x$$
$$2 * x = x + x$$
$$x / 2 = x * 0.5$$

+ associative laws may also be applied to expose common subexpression.

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

+ With the intermediate code might be

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

+ If t is not needed outside this block, the sequence can be

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

Local and Global Optimization: (If asked about the different techniques used - Write about the basic block optimization techniques in local optimization and the loop optimization techniques in the global optimization)

(Many of the structure preserving transformations can be done with the help of Directed Acyclic Graphs - DAG)

Directed Acyclic Graph

- In compiler design, a DAG is an abstract syntax tree with a unique node for each value. DAG is an useful data structure for implementing transformation on basic block. DAG is constructed from three address code.

- Common subexpression can be detected by noticing, as a new node m is about to added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$
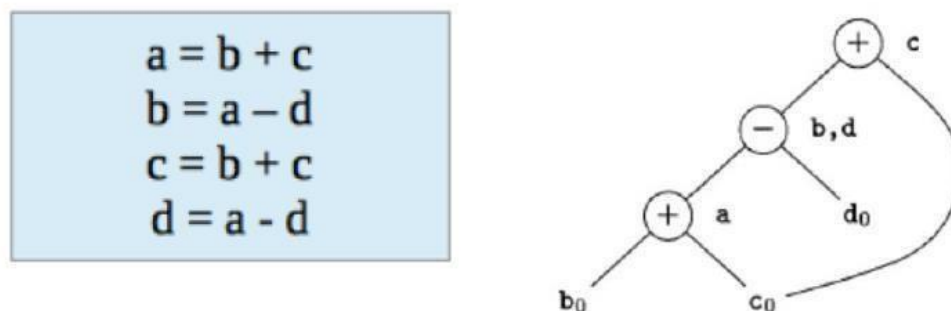
- When we construct the node for the third statement **c = b + c**, we know that the use of **b** in **b + c** refers to the node labeled -, because that is the most recent definition of **b**.

Application of DAG

➢ Determine the common subexpression.

➢ Determine which names are used in the block and compute outside the block.

➢ Determine which statement of the block could have their computed value outside the block.

➢ Simplify the list of quadruples by eliminating common subexpression and not performing the assignment of the form x = y and unless it is a must.

Rules For The Construction Of A DAG

1. In a DAG Leaf node represents identifiers, names, constants. Interior node represents operators.

2. While constructing DAG, there is a check made to find if there is an existing node with same children. A new node is created only when such a node does not exist. This action allows us to detect common subexpression and eliminate the same.

3. Assignment of the form x = y must not be performed until unless it is a must.

EXAMPLE 1
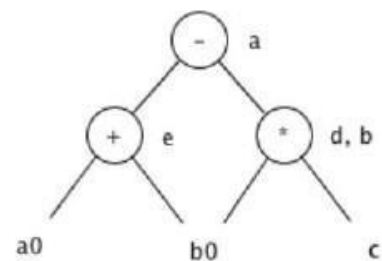
$a = b + c$
$b = b - d$
$c = c + d$
$e = b + c$
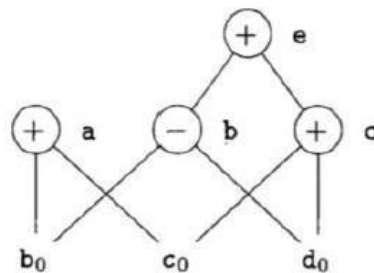
The two occurrences of the sub-expressions **b + c** computes the same value.

Value computed by **a** and **e** are the same.

EXAMPLE 2

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

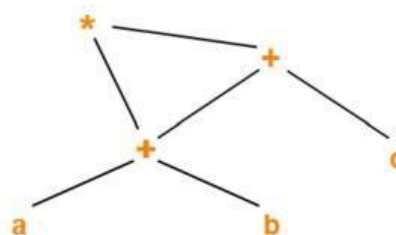## EXAMPLE 3

$(a + b) * (a + b + c)$
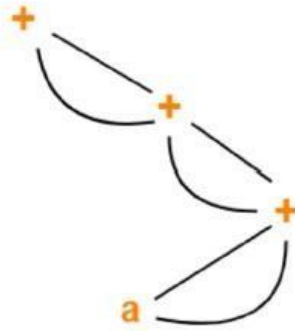
Three address code will be

$t1 = a + b$
$t2 = t1 + c$
$t3 = t1 * t2$

EXAMPLE 4

$$(((a+a)+(a+a))+((a+a)+(a+a)))$$

Machine Independent Optimization:

(All the above discussed optimization techniques are machine independent. i.e., we did the optimization on three address code statements.)

Machine Dependent Optimization:

(When asked for 5 or 10 marks write about Peephole optimization)

Machine dependent optimization is done on the target code. This is an optional step. The most common machine dependent optimization technique is the Peephole optimization technique.

Peephole Optimization:

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replace these instructions replacing by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program.

The peephole optimization can be applied to the target code using the following characteristic.

1. Redundant Load/Store
2. Remove Unreachable Code
3. Flow of Control Optimization
4. Algebraic Simplifications
5. Use of Machine Idioms

Redundant Load/Store

The redundant load and store instructions can be eliminated. For example,

```
MOV   R0, x

MOV   x, R0
```

We can eliminate the second instruction since x is in already R0.

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.

For example,

```
void fun()
{
        int a=10, b=20, c;
        c = a * 10;
        return  c;
        b = b * 15;
        return b;
}
```
Here when the statement „return c" is encountered, its flow will be transferred to the caller of this function. Hence the two statements below it will never get executed. Thus we can consider those two statements at unreachable and rewrite the code as,

```
void fun()
{
        int a=10, b=20, c;
        c = a * 10;
        return c;
}
```

Now, we can see that in the above optimized code, „b=20" has become a dead code since it is no longer used inside this function. Hence we can eliminate it.

Flow of Control Optimization:

There 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
```

Algebraic Simplifications

➕ It represents another important class of optimizations on basic blocks.

$$x + 0 = 0 + x = x$$
$$x - 0 = x$$
$$x * 1 = 1 * x = x$$
$$x / 1 = x$$

➕ Another class of algebraic optimization includes reduction in strength.

$$x ** 2 = x * x$$
$$2 * x = x + x$$
$$x / 2 = x * 0.5$$

Use of Machine Idioms

Some target instructions have equivalent machine instructions to perform some type of operations. Hence we can replace these target instructions with the corresponding machine instructions to improve the efficiency.

For example, some machines have auto-increment and auto-decrement addressing modes. These modes can be used in the code for a statement like i = i+1.

CODE GENERATION

+ The final phase in our compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.
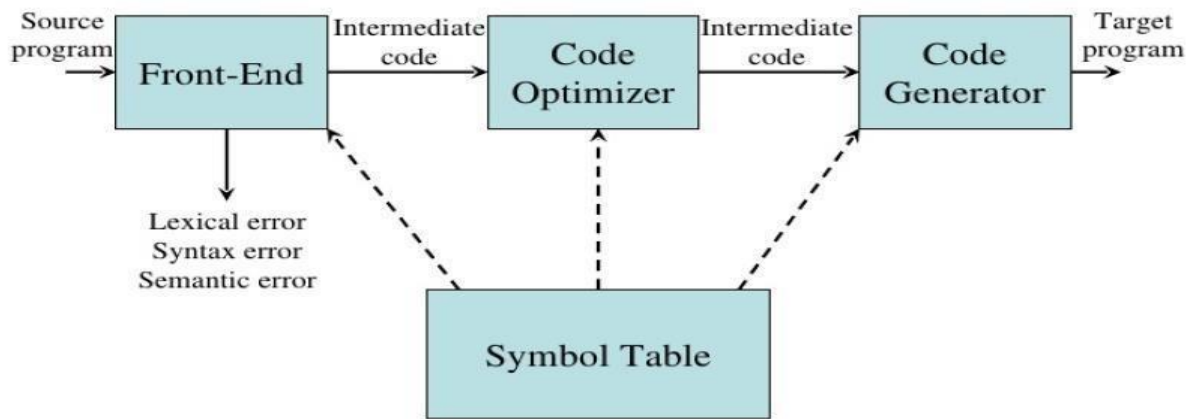
Issues in the Design of a Code Generator:

+ The following issues arise during the code generation phase:

    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.
- There are several choices for the intermediate language including postfix notation, three address representation such as quadruple, virtual machine

representations such as stack machine code, and graphical representations such as syntax trees and DAGS.

- We assume that prior to code generation the front end scanned, parsed and translated the source program into a reasonably detailed intermediate representation, along with type checking.
- The code generation phase can therefore proceed on the assumption that its input is free of errors.

Target Programs

- The output of the code generator is the target program. This output may take on a variety of forms- absolute machine language, relocatable machine language or assembly language.
- Producing an absolute machine language as output has the advantage that it can be placed in a fixed location in memory and intermediate 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 somewhat easier.
- The instruction set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code. The most common target machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer) and stack based.
- The RISC machine has many registers, three-address instructions, simple addressing modes and a relatively simple instruction set architecture. In contrast, a CISC machine has few registers, two-address instructions, a variety of addressing modes, several register classes, variable length instructions.
- In stack based machine, operations are done by pushing operands onto the stack and then performing the operations on the operands at the top of the stack. To achieve high performance, the top of the stack is typically kept in registers.

Memory Management

Mapping of variable names to address is done co-operatively by the front end and code generator. Name and width are obtained from symbol table. Width is the amount of storage needed for that variable. Each three-address code is translated to addresses and instructions during code generation. A relative addressing is done for each instruction.

Instruction Selection

The code generator must map the IR (Intermediate Representation) into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as,

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code

If the IR is **high level**, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation often produces poor code that needs further optimization. If IR reflects some of the **low level** details of the underlying machine, then the code generator can use this information to generate more efficient code sequence.

The **nature** of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. **Uniformity** and **completeness** of the instruction set are important factors.

Instruction speed and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each common three-address statement, a general code can be designed.

Eg: x = y + z

     MOV y, R0

     ADD z, R0

     MOV R0, x

But if we are looking for the efficiency of the program, we need to look for several other factors too. For example, consider the below statements.

a = b + c

d = a + e

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0--------------------can be avoided.

ADD e, R0

MOV R0, d

The **quality** of the generated code is usually determined by its speed and size. For example, if the target machine has an increment instruction INC, then the three-address statement a = a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, adds one to the register, and then store the result back into a.

That is, instead of writing
MOV a, R0

ADD #1, R0

MOV R0, a
We can simply write, INC a.

Register allocation:

+ A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually not have enough of then to hold all values.

+ The use of registers is often subdivided into two sub problems:

> Register allocation, during which we select the set of variables that will reside in registers at each point in the program.

> Register assignment, during which we pick the specific register that a variable will reside in.

Choice of Evaluation Order:

The order of evaluation can affect the efficiency of target code. Some order requires fewer registers and instructions than others.

Picking the best order is an NP-complete problem. This can be solved up to an extent by code optimization in which the order of instruction may change.

Target Machine

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

- Our target computer is a byte-addressable machine with four bytes to a word and n general purpose registers, R0, R1, R2…. Rn-1. It has two address instructions of the form

op source, destination

in which op is an op-code and source and destination are data fields. It has the following op-codes

- ➤ 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+ contents(R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c+ contents (R))	1

MOV R0, M – stores the contents of register R0 into memory location M.

MOV 4(R0), M – stores the value contents(4 + contents(R0))

MOV *4(R0), M – stores the value contents(contents(4 + contents(R0)))

MODE	FORM	ADDRESS	ADDED COST
Immediate	#C	C	1

MOV #1, R0 – load constant 1 into register R0.

Instruction Cost

- Cost of an instruction is one plus the costs associated with the source and destination address modes, indicated by add cost in the above table.

- This cost corresponds to the length of the instruction. Address modes involving registers have cost zero, while those with a memory location or literal in them have cost one, because such operands have to be stored with the instruction.

- We should clearly minimize the length of instructions. Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.

Here are some examples

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

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

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

4. ADD   R2, R1
   MOV   R1, a   **cost = 3**

Simple Code Generator

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

The code generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. **A Register Descriptor** keeps track of what is currently in each register. It is consulted whenever a new register is needed.

2. **An Address Descriptor** keeps track of the location where the current value of the name can be found at run time. The location might be a register, a stack location or a memory address. This information can be stores in the symbol table and is used to determine the accessing method for a name.

Code Generation Algorithm

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form x := y op z perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation y op z should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.

4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions. That is, it takes a

register which is already occupied, move its contents into some memory using the instruction MOV R, M. and then uses the register R.

Generating Code For Assignment Statements

The assignment d := (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

$$t := a - b$$
$$u := a - c$$
$$v := t + u$$
$$d := v + u \quad \text{with d live at the end.}$$

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t := a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u := a - c	MOV a, R1 SUB c, R1	R_0 contains t R1 contains u	t in R_0 u in R1
v := t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d := v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for other type of statements

Statements	Code Generated	Cost
a := b[i]	MOV b(R_i), R	2
a[i] := b	MOV b, a(R_i)	3

Statements	Code Generated	Cost
a := *p	MOV *R_p, a	2
*p := a	MOV a, *R_p	2

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x := y +z
 if x < 0 goto z
	MOV y, R_0
	 ADD z, R_0
	 MOV R_0,x
	 CJ< z