CD : COMPILER DESIGN

Code Optimization and Generation

Marwadi University
Marwadi Chandarana Group
NAAC A+

Department of CE

Unit no : 5
Code Optimization and Generation
(01CE0714)

Prof.Shilpa Singhal

Outline :

Optimization Techniques

Peephole Optimization

Global Data Flow Analysis

Issues in design of Code generators

Basic Block and Flow Graphs

Register Allocation and Assignments

DAG Representation of Basic Block

Marwadi University
Marwadi Chandarana Group
NAAC A+

Department of CE

Unit no : 5
Code Optimization and Generation
(01CE0714)

Prof.Shilpa Singhal

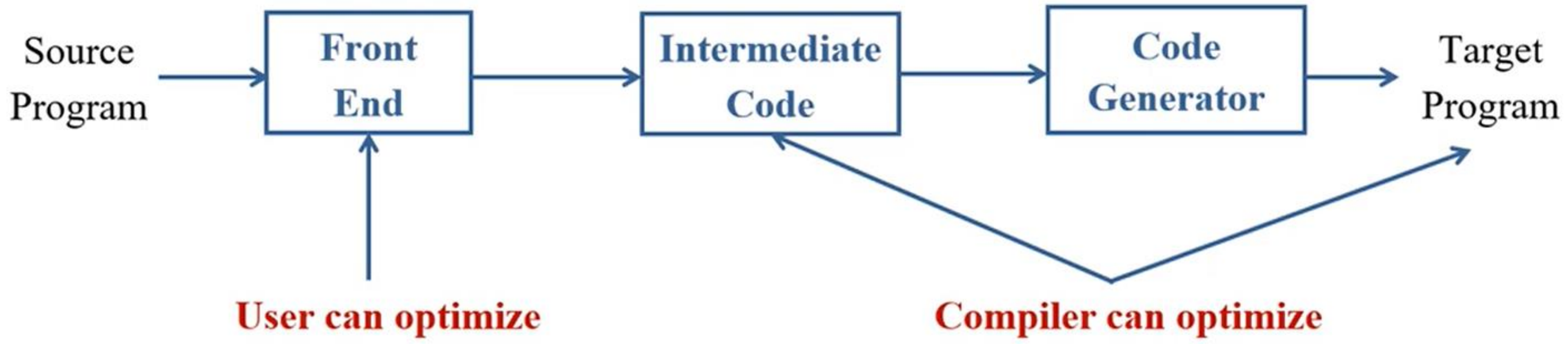
What is Optimization

- Optimization is the process of transforming a piece of code to make more efficient (in terms of time or space) without changing its output or side-effects.
- User can able to see that code runs faster and/or consumes less memory.

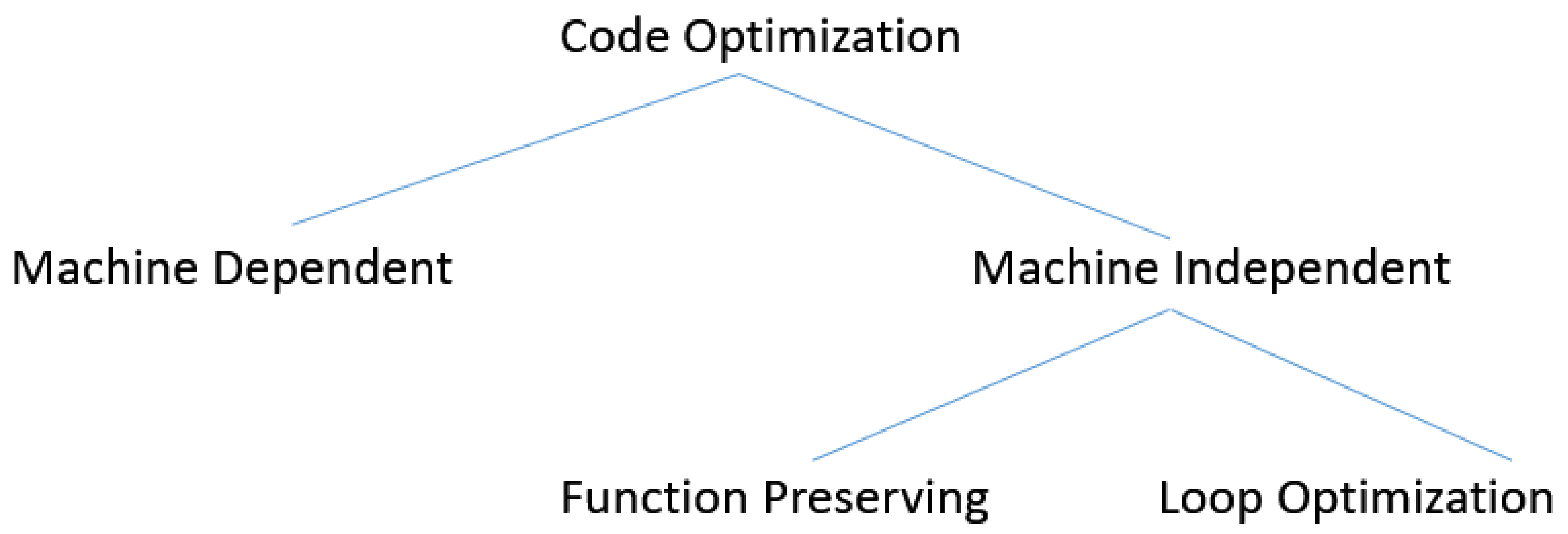
Advantages

- Optimized code has faster execution speed
- Optimized code utilizes the memory efficiently
- Optimized code gives better performance

What is Optimization



Optimization Technique



Optimization Technique

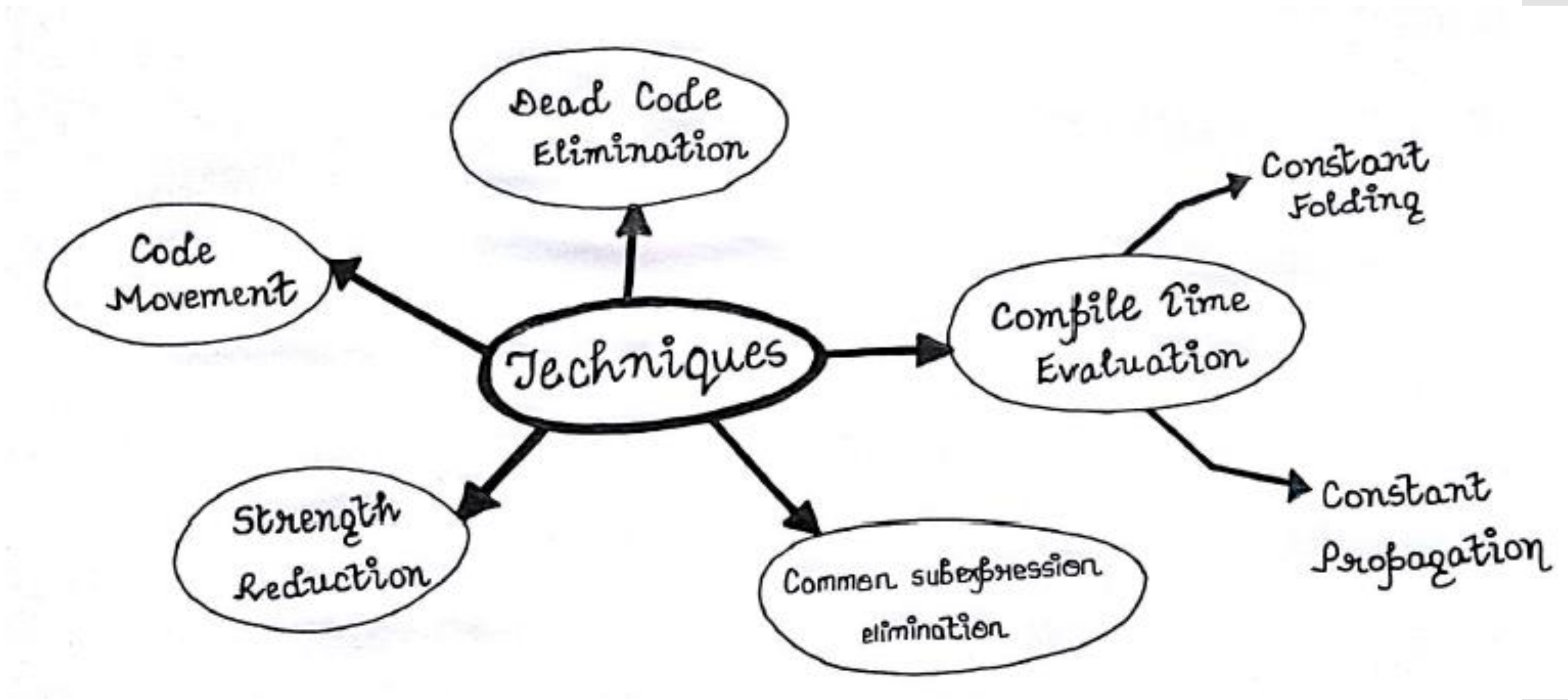
Function Preserving

- Common Sub Expression Elimination
- Constant Folding
- Copy Propagation
- Dead Code Elimination

Loop Optimization

- Code Motion
- Strength Reduction

Optimization Technique



1.Compile Time Evaluation

- It means Shifting of computation from run time to compile time.
- There are two methods

1. **Folding** : in this technique, the computation of constant is done at compile time instead of run time. Example : length = (22/7) * d
2. **Constant Propagation** : In this technique, the value of variable is replaced and computation of expression is done at compilation time.

   Example : pi=3.14, r=2

Area = pi*r*r → 3.14*2*2 (computation is done during compilation)

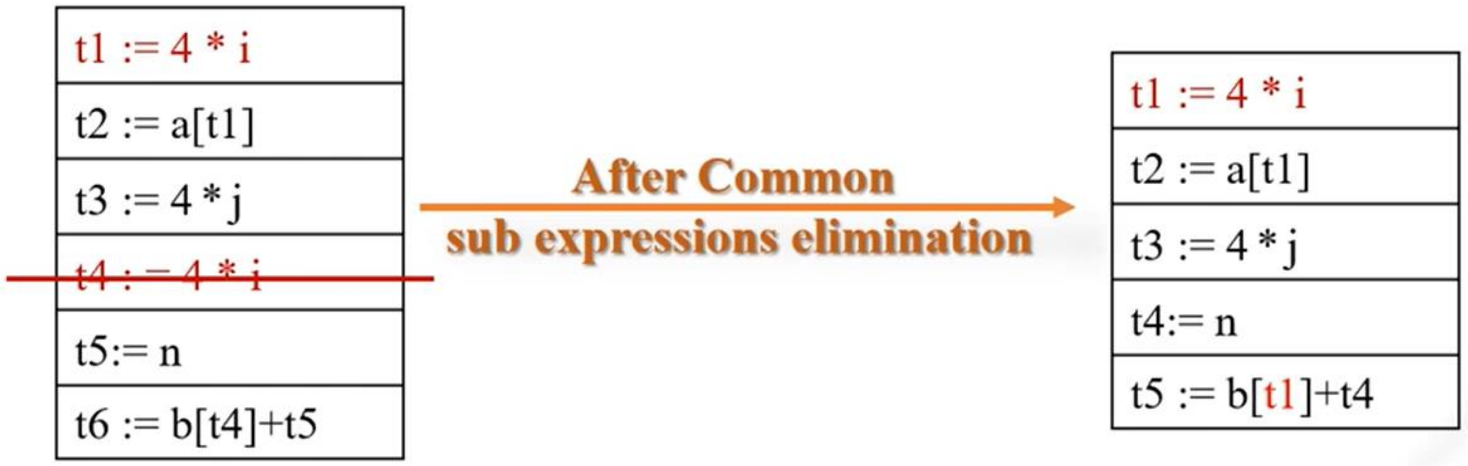
2.Common Sub Expression Elimination

- The common subexpression is an expression appearing repeatedly in the program which is computed previously.
- If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.

```
t1 = 4*i                      t1 = 4*i
t2  = a[t1]                   t2 = a[t1]
t3 = 4*j          ──────>     t3 = 4*j
t4 = 4*i
t5= b[t4] + 1                 t4 = b[t1] + 1
```

2.Common Sub Expression Elimination

- Example:



t1 := 4 * i
t2 := a[t1]
t3 := 4 * j
~~t4 : = 4 * i~~
t5:= n
t6 := b[t4]+t5

After Common sub expressions elimination

t1 := 4 * i
t2 := a[t1]
t3 := 4 * j
t4:= n
t5 := b[t1]+t4

3.Copy propagation

It means used of one variable instead of another.

Example

A = PI

Area = **A** * r * r

$\Downarrow$

Area = **PI** *r *r

Here variable A is eliminated.

4.Code Motion

- If variables used in a computation within a loop are not altered within the loop, the calculation can be performed outside of the loop and the results used within the loop.

```
While (i <= limit-2)
{
        --------
        --------
}
```

$\longrightarrow$

```
n = limit-2
While (i <= n)
{
        --------
        --------
}
```

4.Code Motion

- Example:

```
for (i=1; i <= 100 ; i++)
        {
                z = i ;
                x = 25 * a ;
                y = x + z ;
        }
```

```
x = 25 * a ;
for (i=1; i <= 100 ; i++)
        {
                z = i ;
                y = x + z ;
        }
```

✓ Here x = 25 * a; is loop invariant. Hence in the optimized program it is computed only o
before entering the for loop. y = x + z; is not loop invariant. Hence it cannot be subjected
frequency reduction

5.Reduction in Strength

- Strength of certain operators are higher than others.
- Replaces less efficient instructions with more efficient ones. For example, in array subscripting, an add instruction replaces a multiply instruction.
- Example: Replacement of multiplication by Addition.

```
for(i=1;i<=50;i++)
{
        count = i*7;
}
```

```
temp=7
for(i=1;i<=50;i++)
{
        count= temp;
        temp = temp + 7;
}
```

6. Dead Code Elimination

- The variable is said to be Dead at a point in a program, if the value contained into it is never been used.
- Eliminated code that can not be reached or where the results are not subsequently used.
- Example :

```
int i=0,x;
If (i==1)
{
        x=x+i;
}
```

Here ‘if’ statement is dead code because the condition never gets satisfied.

6. Dead Code Elimination

Example :

```
main()
        {
        …………
        a=5;
        if(a==5)
                {
                        c++;
                        printf("%d",c);
                }
        else
                {
                        k++;
                        printf("this is dead code");
                }
        }
```

Peephole Optimization

- It 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 instruction (called peephole or window) and replacing these instructions by shorter or faster sequence whenever possible.
- Peephole is a small, moving window on the target program.

Goals :

- Improve Performance
- Reduce code size

Types of Peephole Optimization

1.Redundant Instruction Elimination

2.Flow of Control Optimization

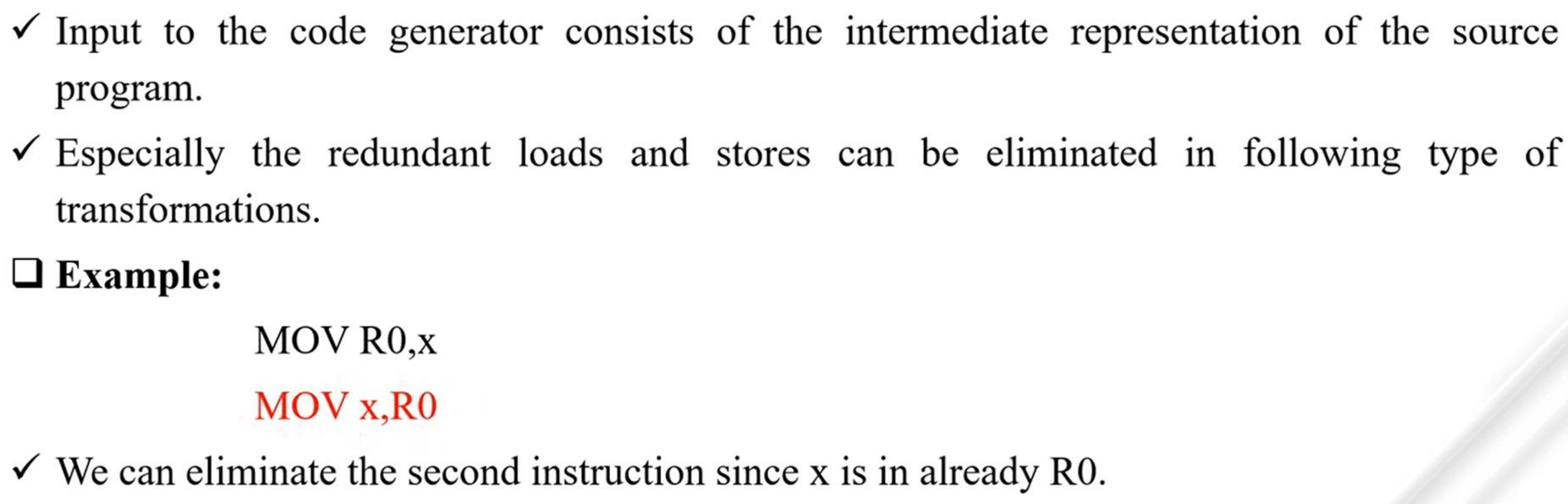
3.Unreachable Code/Dead Code

4. Algebraic Specification

5. Reduction in Strength

6.Machine Idioms

1.Redundant Instruction Elimination



✓ Input to the code generator consists of the intermediate representation of the source program.

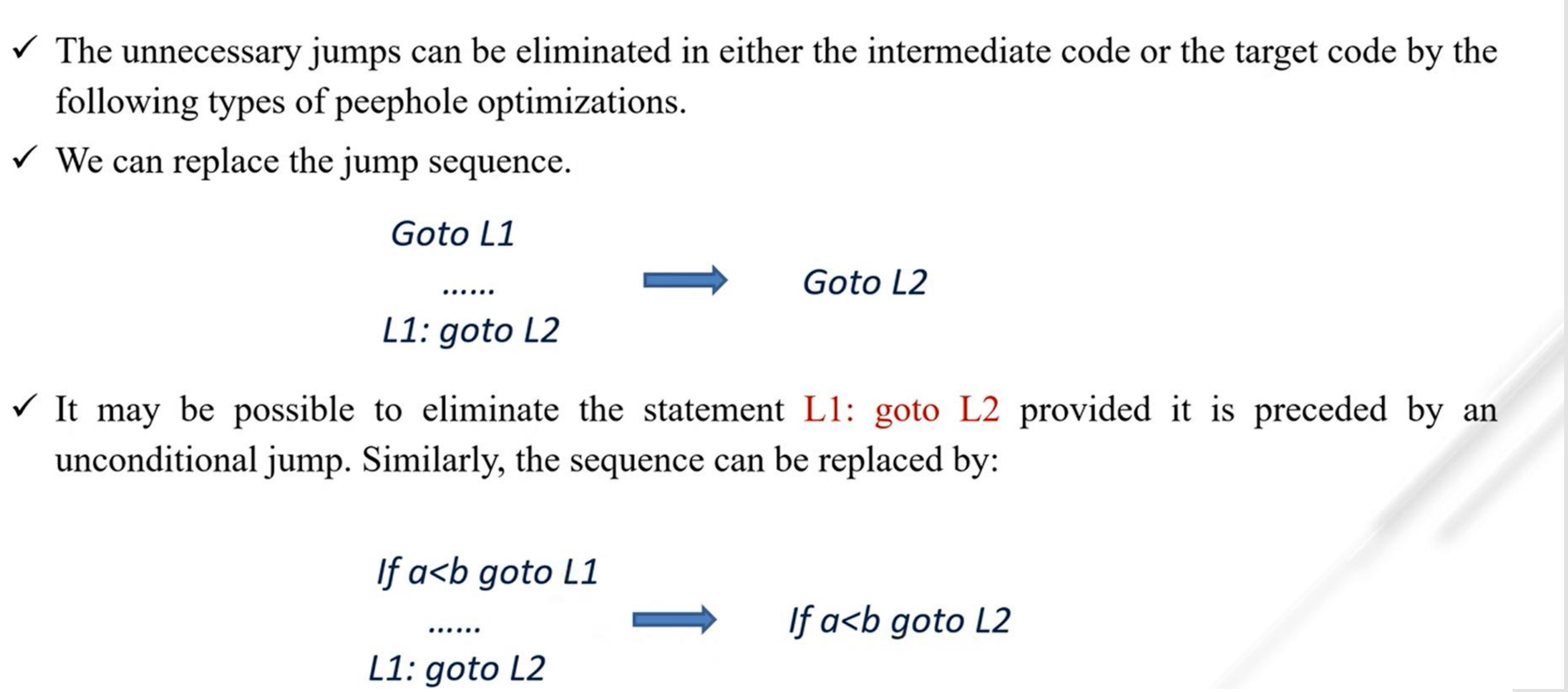
✓ Especially the redundant loads and stores can be eliminated in following type of transformations.

❑ **Example:**

```
MOV R0,x
MOV x,R0
```

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

2.Flow of Control Optimization



- ✓ The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.
- ✓ We can replace the jump sequence.

```
Goto L1
......          ⟹     Goto L2
L1: goto L2
```

- ✓ It may be possible to eliminate the statement L1: goto L2 provided it is preceded by an unconditional jump. Similarly, the sequence can be replaced by:

```
If a<b goto L1
......          ⟹     If a<b goto L2
L1: goto L2
```

3.Unreachable Code

```
int dead(void)
{
   int a=1;
   int b=5;
   int c=a+b;
   return c;
// c will be returned
// The remaining part of code is dead code, never reachable
   int k=1;
   k=k*2;
   k=k+b;
   return k;
// This dead code can be removed for optimization
}
```

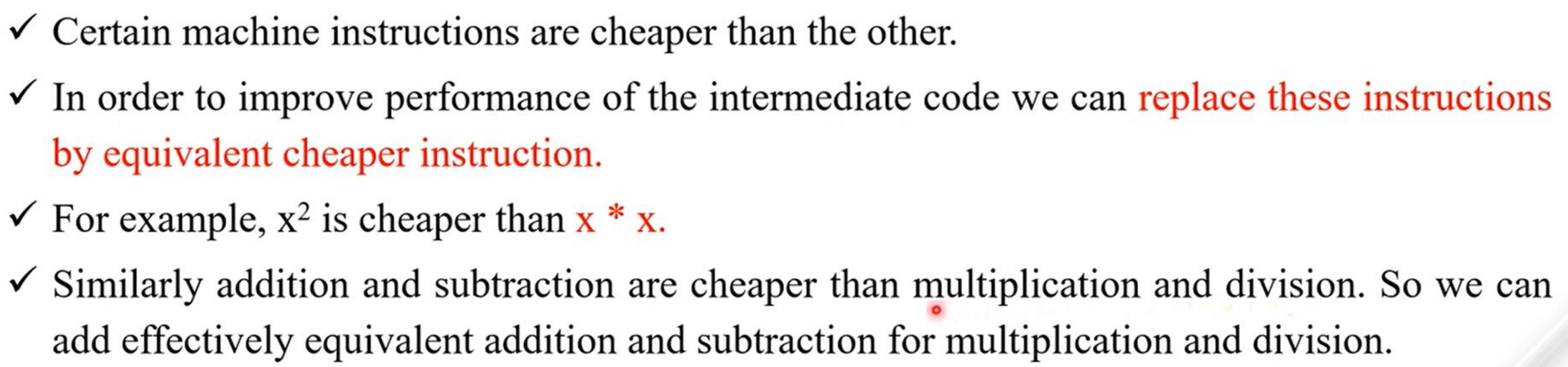
4. Algebraic Specification

- Peephole optimization is an effective technique for algebraic specification.
- The statements such as

$$x = x + 0 \quad \text{or} \quad x = x * 1$$

can be eliminated by peephole optimization.

5. Reduction in Strength



- ✓ Certain machine instructions are cheaper than the other.
- ✓ In order to improve performance of the intermediate code we can replace these instructions by equivalent cheaper instruction.
- ✓ For example, x^2 is cheaper than $x * x$.
- ✓ Similarly addition and subtraction are cheaper than multiplication and division. So we can add effectively equivalent addition and subtraction for multiplication and division.

6.Machine Idioms

- The target instructions have equivalent machine instructions for performing some operations.
- Hence we can replace these target instructions by equivalent machine instruction in order to improve the efficiency.
- Example : Some machines have auto-increment and auto-decrement addressing modes.
- These modes can be used in code for the statements : i=i+1 or i=i-1.

Global Data Flow Analysis

- In order to do code optimization and a good job of code generation , a compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- Data flow equations are the equations representing the expressions that are appearing in the flow graph.
- Data flow information can be collected by setting up and solving systems of equations that relate information at various point in a program.

Data Flow Analysis

- It is analysis of flow of data in control flow graph i.e the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done. In general it’s the process in which values are computed using data flow analysis. The data flow property represents information which can be used for optimization.

Global Data Flow Analysis

- A typical equation has the form

Out[S] = gen[S] U (in[S] – kill[S])

And can be read as

“the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement”.

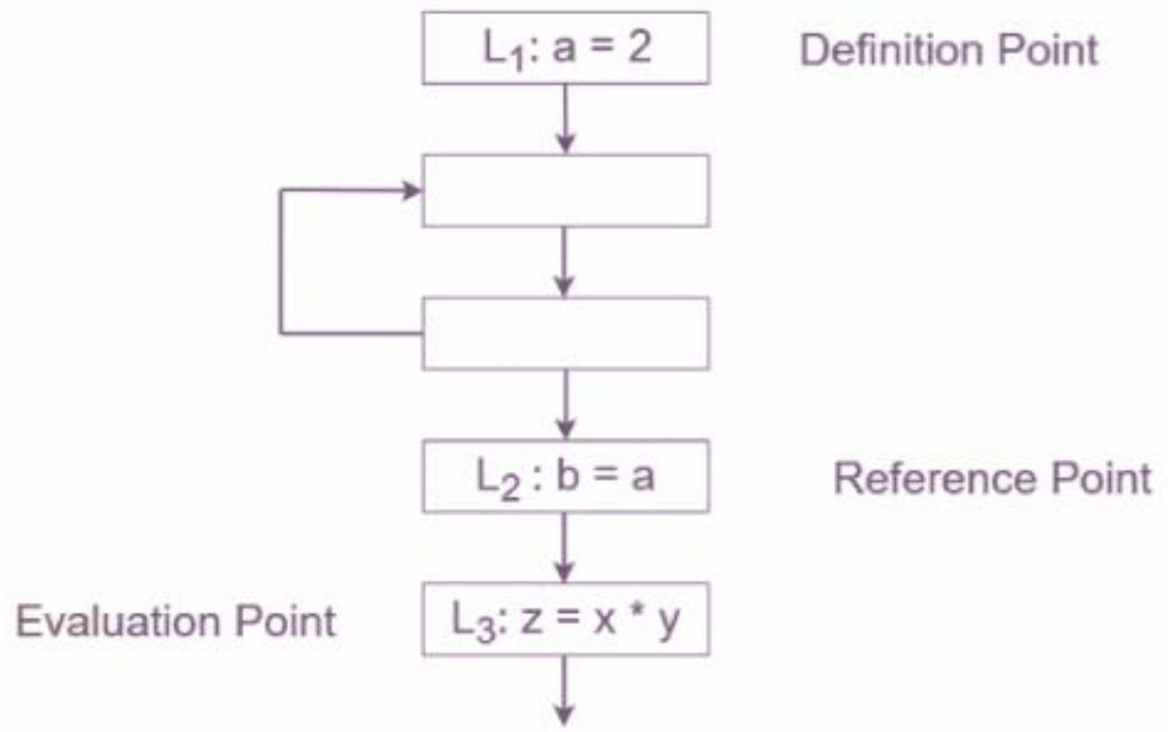
•**Out[S]**: This represents the set of information (e.g., variables or data properties) that "flows out" or is valid at the end of the statement S.
•**gen[S]**: This represents the set of information generated by S itself. For example, if S defines a new variable or computes a new value, this is part of gen[S].
•**in[S]**: This represents the set of information available or "flows in" at the beginning of the statement S, which could have been defined by previous statements.
•**kill[S]**: This represents the set of information invalidated or "killed" by S, such as reassigning a variable that was previously defined.

Global Data Flow Analysis

The details of how data-flow analysis are set up and solved depend on three factors.

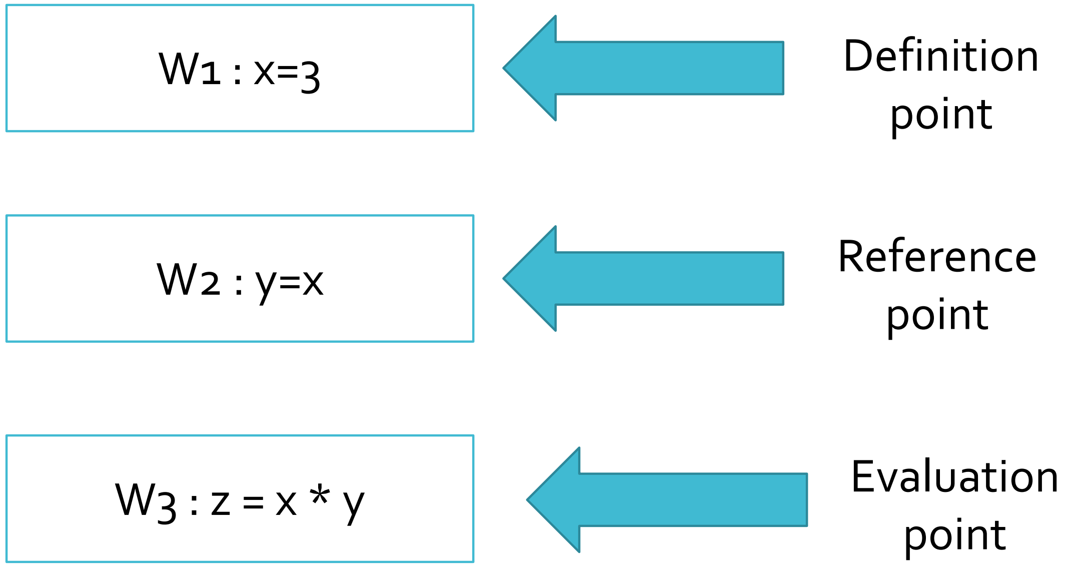
1. The notion of generating and killing depend on the desired information. i.e. for some problem, instead of proceeding along with the flow of control and defining out[S] in terms of in[S], we need to proceed backwards and defines in[S] in terms of out[S].
2. Since data flows along control paths, data-flow analysis is affected by the control constructs in a program.
3. There are subtleties that go along with such statements as procedure calls, assignment through pointer and even assignments to array variables.

Data Flow Properties



Data Flow Properties

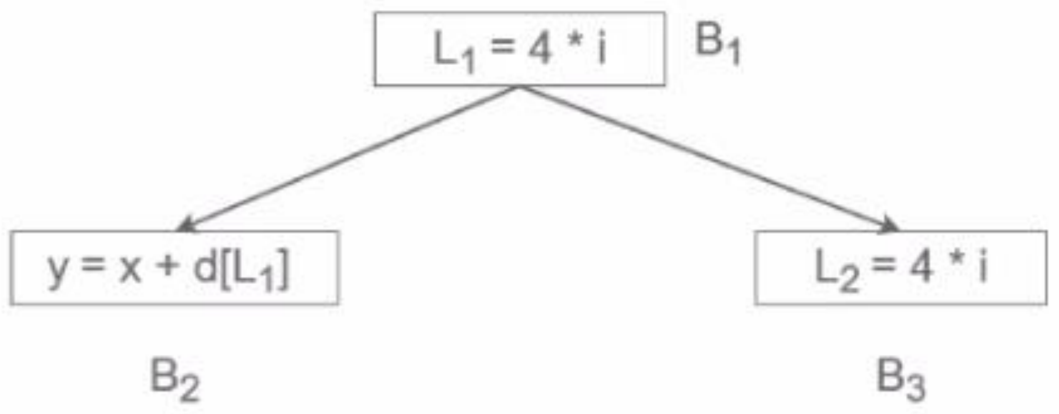
- A program point containing the definition is called **Definition Point.**
- A program point at which a reference to a data item is made is called **reference point.**
- A program point at which some evaluating expression is given is called **evaluation point.**



Available Expression

1. The expression x+y is said to be available at its evaluation point
2. Neither of two operands get modified before their use.

Example

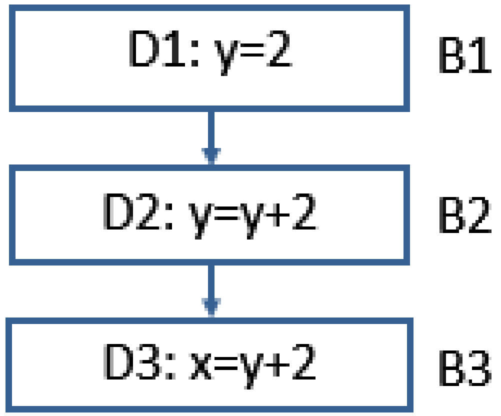


Expression 4 * i is available for block B_2 , B_3

Reaching Definition

Reaching Definition:

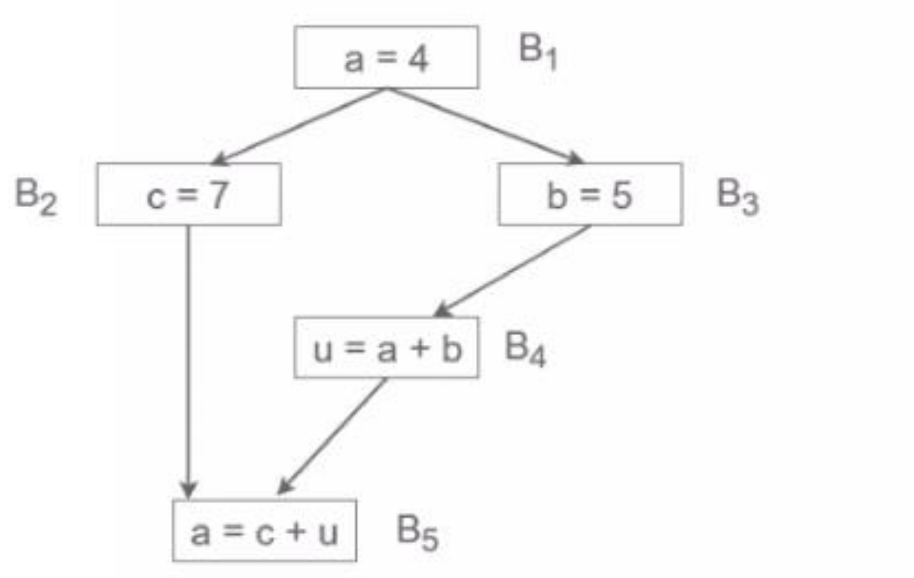
A definition D reaches a point x if there exists a path from D to x in which D is not killed or redefined.



D1 is reaching definition for block B2,but the definition D1 isn’t reaching for block B3,because it is killed by definition D2 in block B2.

Live Variable

- A live variable x is live at point p, if there is path from p to the exit, along which the value of x is used before its redefined.
- Otherwise the variable is said to be dead at the point.



a is live at block B_1 , B_3 , B_4 but killed at B_5

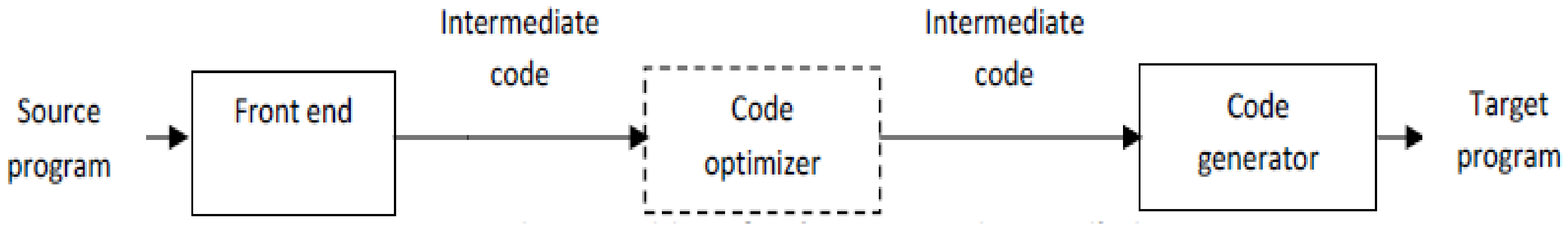
Busy Expression

Busy Expression:

- An expression e is said to be busy expression along some path pi…pj if an evaluation of e exists along some path pi…pj and no definition of any operand exist before its evaluation along path.

Role of Code Generator

- The final phase of compilation process is code generation.
- It takes an intermediate representation of the source program as input and produces an equivalent target program as output.
- Main task of Code Generator:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering



Code Generator

- **Instruction selection:**
  - Choose appropriate target-machine instructions to implement the IR statements.
- **Register allocation and assignment:**
  - Decide what values to keep in which registers.
- **Instruction ordering:**
  - Decide in what order to schedule the execution of instructions.

Code Generator

Target code should have following property:

- Correctness
- High quality
- Efficient use of resources of target code
- Quick code generation

Issues in Code Generation

Issues in Code Generation

Issues in design of code generation are:

1. **Input to the code generator**
2. **Target program**
3. **Memory management**
4. **Instruction selection**
5. **Register allocation**
6. **Choice of evaluation order**
7. **Approaches to code generation**

1.Input to the code generator

- Input to the code generator consists of the intermediate representation of the source program.
- There are several types for the intermediate language,
  - Quadruples
  - syntax trees Or DAGs.
  - postfix notation
- The detection of semantic error should be done before submitting the input to the code generator.
- The code generation phase requires complete error free intermediate code as an input.

2.Target Program

- The output of the code generator is the target program. The output may take on a variety of forms:

1. **Absolute machine language (executable code)**

   Language has the advantage that it can be placed in a fixed location in memory and immediately executed. (executable)

2. **Relocatable machine language (Object files or linker)**

   Producing a relocatable machine language program as output is that the subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. (object model / '.o' file)

3. **Assembly language**

   Producing an assembly language program as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

3.Memory Management

- This refers to use of malloc/free, new/delete, and related issues.
- Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.
- We assume that a name in a three-address statement refers to a symbol table entry for the name.
- From the symbol table information, a relative address can be determined for the name in a data area.

4.Instruction Selection

- If we do not care about the efficiency of the target program, instruction selection is straightforward. It requires special handling. For example, the sequence of statements
- 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, so we can eliminate that statement.

5.Register Allocation

- Instructions containing register operands are usually shorter and faster than that of using in memory or involving operands in memory.
- The use of registers is often subdivided into two sub problems:

1. **Register allocation:** we select the set of variables that will reside in registers at a point in the program.
2. **Register assignment:** select a specific register that a variable reside in.

Complications imposed by the hardware architecture:

- Finding an optimal assignment of registers to variables is difficult, even with single register value.
- Mathematically the problem is NP-complete.

Register Allocation (Example)

- T = A + B
- T = T * C
- T = T / D

- Load R1, A
- Add R1, B
- Mul R1, C
- Division R1, D
- Store T,R1

6.Choice of evaluation

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold **i**ntermediate results than others.
- Picking a best order is another difficult, NP-complete problem.
- When instructions are independent, their evaluation order can be changed.

Choice of evaluation (Example)

a + b – (c + d) * e

```
T1 = a + b
T2 = c + d
T3 = e * t2
T4 = T1 – T3
```

```
MOV a, R0
ADD b, R0
MOV R0, T1
MOV c, R1
ADD d, R1
MOV e, R0
MUL R1, R0
MOV T1, R1
SUB R0, R1
MOV R1, T4
```

reorder

```
T2 = c + d
T3 = e * T2
T1 = a + b
T4 = T1 – T3
```

```
MOV c, R0
ADD d, R0
MOV e, R1
MUL R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, T4
```

7.Approaches to code generator

- The most important criterion for a code generator is that it produces correct code.
- Correctness takes on special significance because of the number of special cases that code generator might face.
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

Basic Blocks

- A basic block is sequence of 3-address statements where control enters at the beginning and leaves only at the end without any jumps or halts.
- In order to find the basic blocks, we need to find the leaders in the program. Then a basic block will start from one leader to the next leader but not including next leader.

Identifying leaders in basic block:

1. 1st statement is leader.
2. Statement that is target of conditional or unconditional statement is a leader.
   - if (i<10) goto 100 (Line number 100 is a leader)
   - goto 200 (Line number 200 is a leader)
3. Statement that follows immediately a conditional or unconditional statement is a leader.

Identifying leaders in Basic Block

- Program in HLL:

```
fact(x)
{
        int f = 1;
        for(i = 2; i <= x; i++)
                f = f * i;
        return f;
}
```

Output of ICG
(3-address code)

```
1) f = 1;
2) i = 2;
3) if (i > x) go to 9
4) t1 = f * i;
5) f = t1;
6) t2 = i + 1;
7) i = t2;
8) goto(3);
9) goto calling program
```

Partition into Basic Blocks

Output of ICG
(3-address code)

```
Leader ──> 1) f = 1;                    Block 1
           2) i = 2;

Leader ──> 3) if (i > x) go to 9        Block 2

Leader ──> 4) t1 = f * i;
           5) f = t1;
           6) t2 = i + 1;               Block 3
           7) i = t2;
           8) goto(3);

Leader ──> 9) goto calling              Block 4
           program
```

Flow Graphs

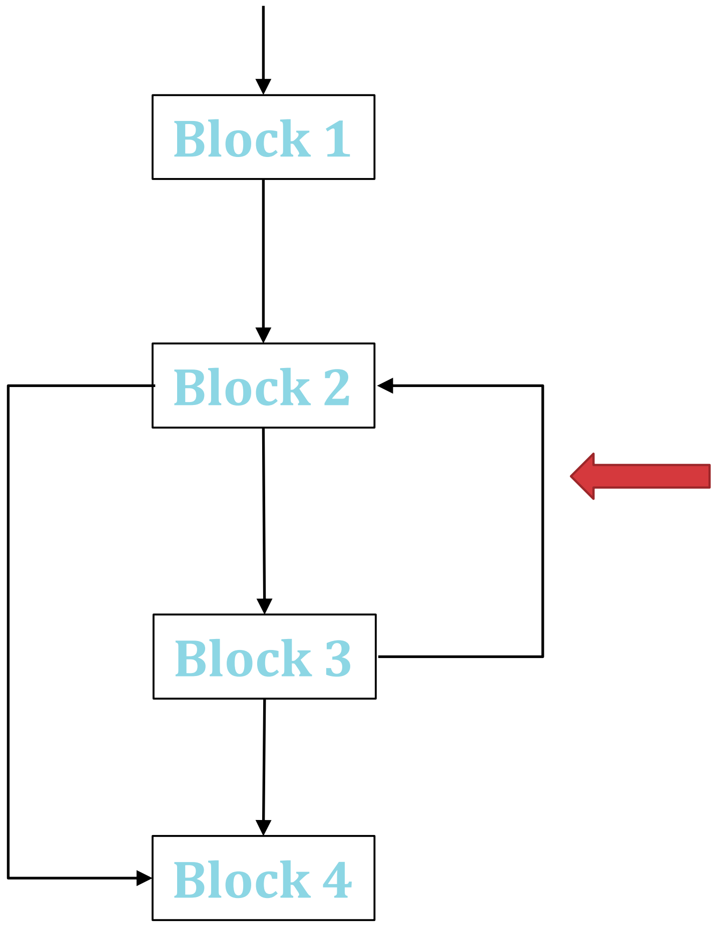
- A flow graph is a graphical representation of a sequence of instructions with control flow edges.
- A flow graph can be defined at the intermediate code level or target code level.
- A control flow graph (CFG) is a directed graph with basic blocks Bi as vertices and with edges Bi -> Bj if Bj can be executed immediately after Bi

Successor and Predecessor Blocks:

- Suppose the CFG has an edge B1 -> B2
- Basic block B1 is a predecessor of B2
- Basic block B2 is a successor of B1

Flow Graphs

Flow Graph



Cycle is detected so code optimization can be done.

Register Allocation and Assignment

Register Allocation and Assignment

- Efficient utilization of registers is important in generating good code.
- There are four strategies for deciding what values in a program should reside in a registers and which register each value should reside.
- Strategies are:
  - Global Register Allocation
  - Usage Count
  - Register Assignment for Outer Loop
  - Register Allocation for Graph Coloring

Global Register Allocation

Strategies adopted while doing the global register allocation are:

- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
- Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
- The registers are not already allocated may be used to hold values local to one block.
- In certain languages like C or Bliss programmer can do the register allocation by using register declaration to keep certain values in register for the duration of the procedure.
- Example:

```
{
    register int x;
}
```

Usage Count

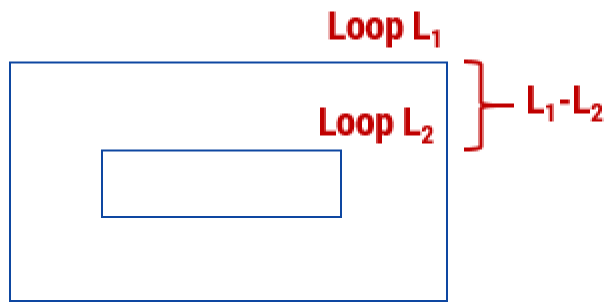
- The usage count is the count for the use of some variable x in some register used in any basic block.
- The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation.
- The approximate formula for usage count for the Loop L in some basic block B can be given as,

$$\Sigma_{\text{block B in L}} (\text{use}(x,B) + 2 * \text{live}(x,B))$$

- Where use(x,B) is number of times x used in block B prior to any definition of x live(x,B) =1 if x is live on exit from B; otherwise live(x)=0.

Register Assignment for Outer Loop

- Consider that there are two loops L1 is outer loop and L2 is an inner loop, and allocation of variable a is to be done to some register. The approximate scenario is as given below,



Following criteria should be adopted for register assignment for outer loop,

- If a is allocated in loop L2 then it should not be allocated in L1 - L2.
- If a is allocated in L1 and it is not allocated in L2 then store a on entrance to L2 and load a while leaving L2.
- If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exit from L2.

Register Allocation for Graph Coloring

- The graph coloring works in two passes. The working is as given below,
- In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
- In the second pass the register inference graph is prepared.
- In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.
- Then a graph coloring technique is applied for this register inference graph using k- color.
- The k-colors can be assumed to be number of assignable registers.
- In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node (actually a variable) is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.

DAG Representation of Basic Block

Directed Acyclic Graph. (DAG)

- DAG represents **Directed Acyclic Graph.**
- Syntax tree and DAG both are the graphical representations but Syntax tree does not find the **common sub expressions** but DAG does.
- Another use of DAG is application of optimization technique of basic block.
- To apply the optimization technique on Basic block ,DAG is constructed from three address code

Algorithm for Constructing DAG

- We assume the three address statement could of following types,

  Case (i) x:=y op z

  Case (ii)x:=op y

  Case (iii) x:=y
- With the help of following steps the DAG can be constructed.
- **Step 1:** If y is undefined then create node(y). Similarly if z is undefined create a node(z)
- **Step 2:**
  - Case(i) create a node(op) whose left child is node(y) and node(z) will be the right child. Also check for any common sub expressions.
  - Case(ii) determine whether is a node labeled op, such node will have a child node(y).
  - Case(iii) node n will be node(y).
- **Step 3:** Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

DAG Representation of Basic Block

Example 1:

```
a=b+c
b=a-d
c=b+c
d=a-d
```

Example 2:

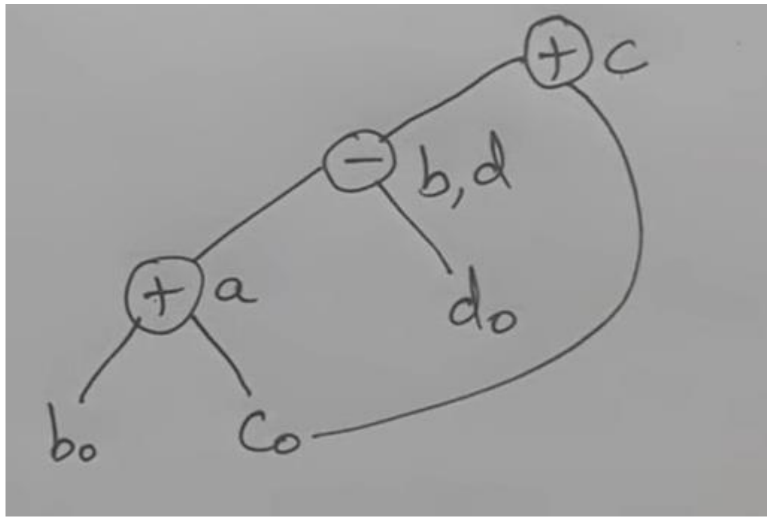
```
a=b+c
b=b-d
c=c+d
e=b+c
```

Example 3:

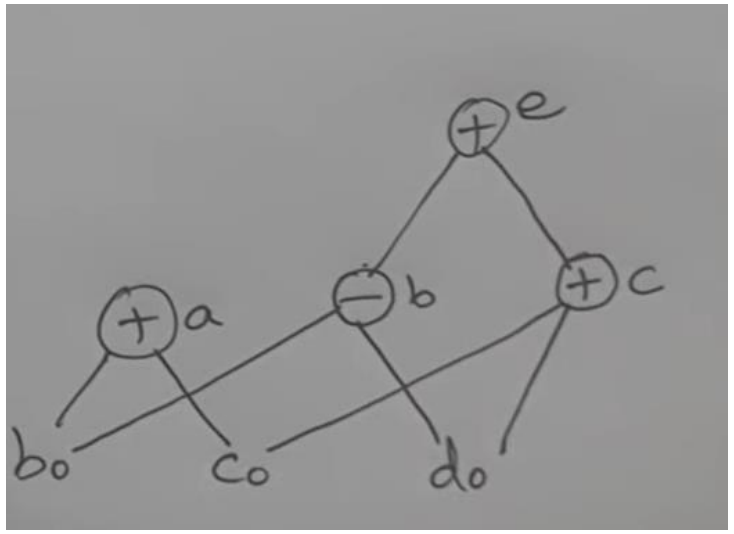
a=b*-c+b*-c (Expression)

DAG Representation of Basic Block

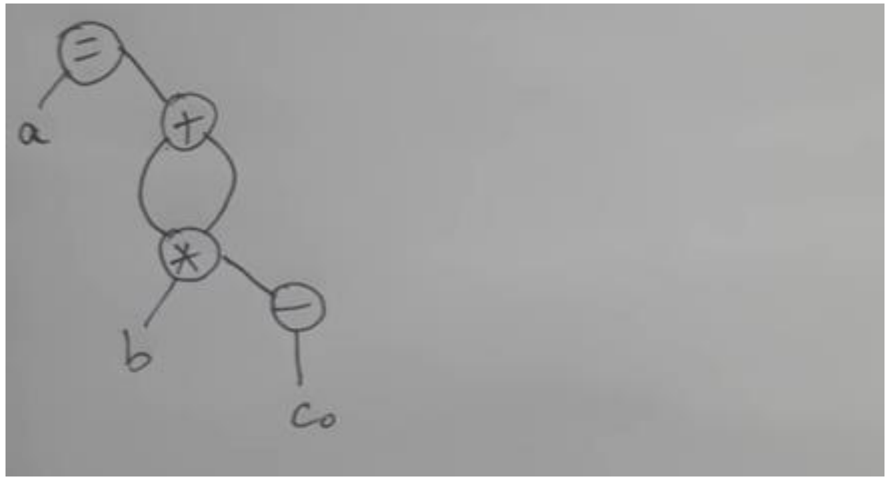
Example 1:



Example 2:

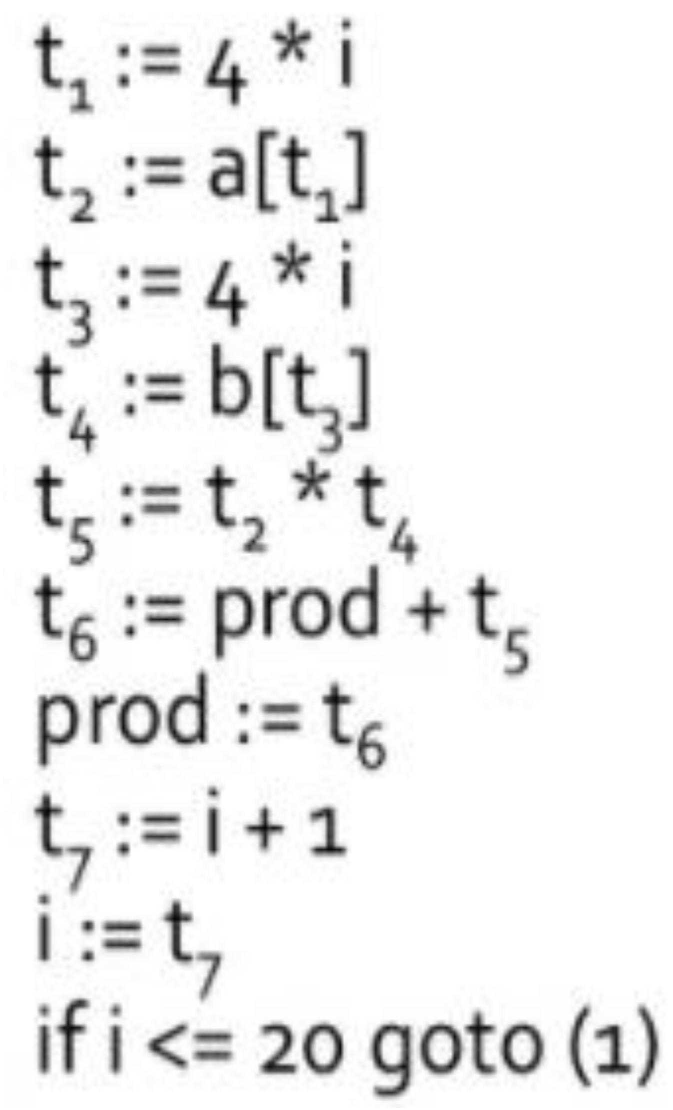


Example 3:



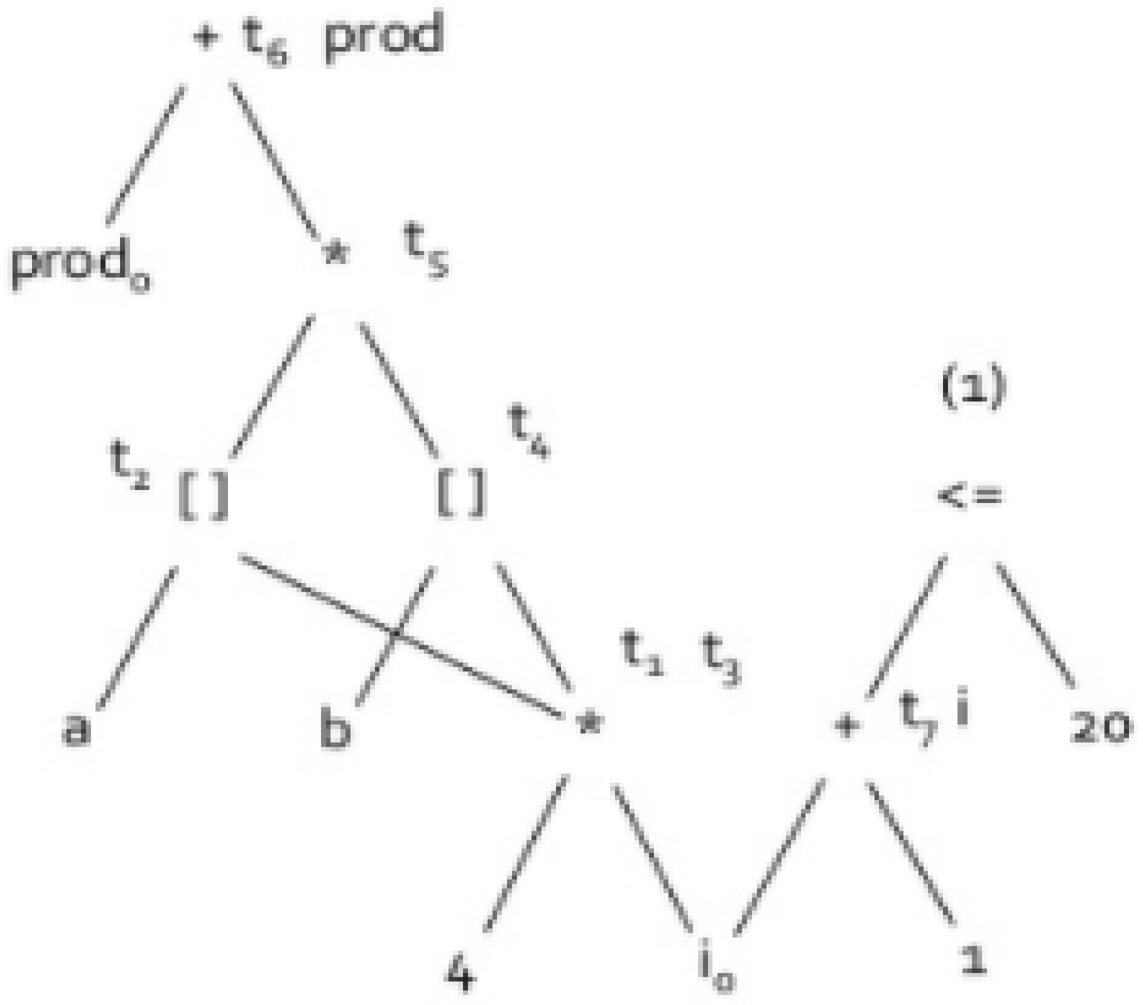
DAG Representation of Basic Block

Example 4:



```
t1 := 4 * i
t2 := a[t1]
t3 := 4 * i
t4 := b[t3]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i <= 20 goto (1)
```

DAG Representation of Basic Block



Applications of DAG

The DAGs are used in:

- Determining the common sub-expressions.
- Determining which names are used inside the block and computed outside the block.
- Determining which statements of the block could have their computed value outside the block.
- Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form x:=y unless and until it is a must.

Questions

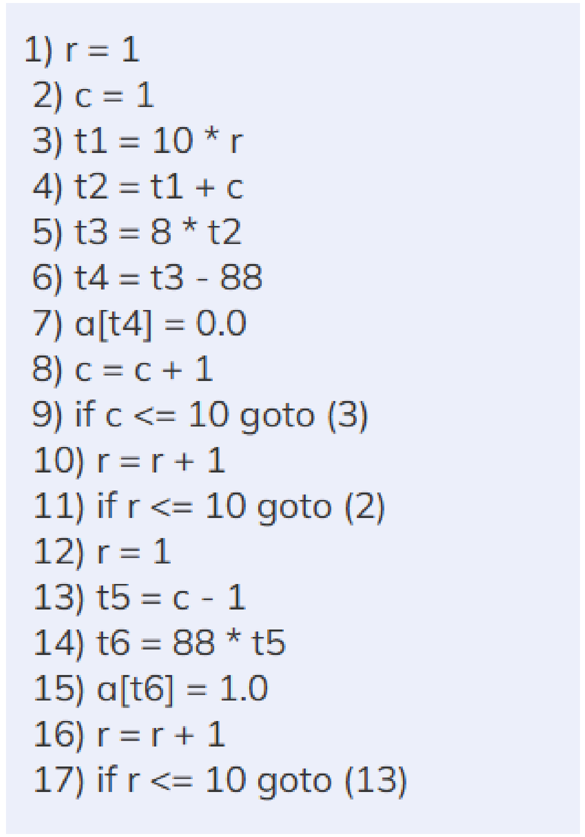
1.Explain the various types of Optimization Techniques.

2.What is Peephole Optimization. Explain the types of Peephole Optimization techniques.

3.What are different strategies of Register Allocation and Assignment.

4. Explain the issues in Code Generation.

5.Find the basic blocks in the following 3-address code.



```
1) r = 1
 2) c = 1
 3) t1 = 10 * r
 4) t2 = t1 + c
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) c = c + 1
 9) if c <= 10 goto (3)
 10) r = r + 1
 11) if r <= 10 goto (2)
 12) r = 1
 13) t5 = c - 1
 14) t6 = 88 * t5
 15) a[t6] = 1.0
 16) r = r + 1
 17) if r <= 10 goto (13)
```

Thanks

Marwadi University
Marwadi Chandarana Group
NAAC A+

Prof. Shilpa Singhal