COMPILERS: SYNTHESIS PHASE

Module - 6

St. Francis Institute of Technology, Ms. Ankita Karia

TOPICS TO COVER

Intermediate Code Generation:

1. Types of Intermediate codes: Syntax tree, Postfix notation,
2. Three address codes: Triples and Quadruples, indirect triple.

Code Optimization:

1. Need and sources of optimization
2. Code optimization techniques: Machine Dependent and Machine Independent.

Code Generation: Issues in the design of code generator, code generation algorithm. Basic block and flow graph.

INTERMEDIATE CODE GENERATION

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

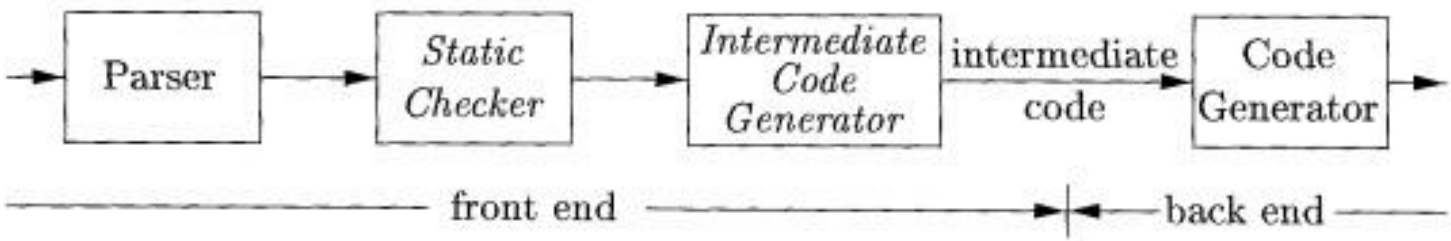
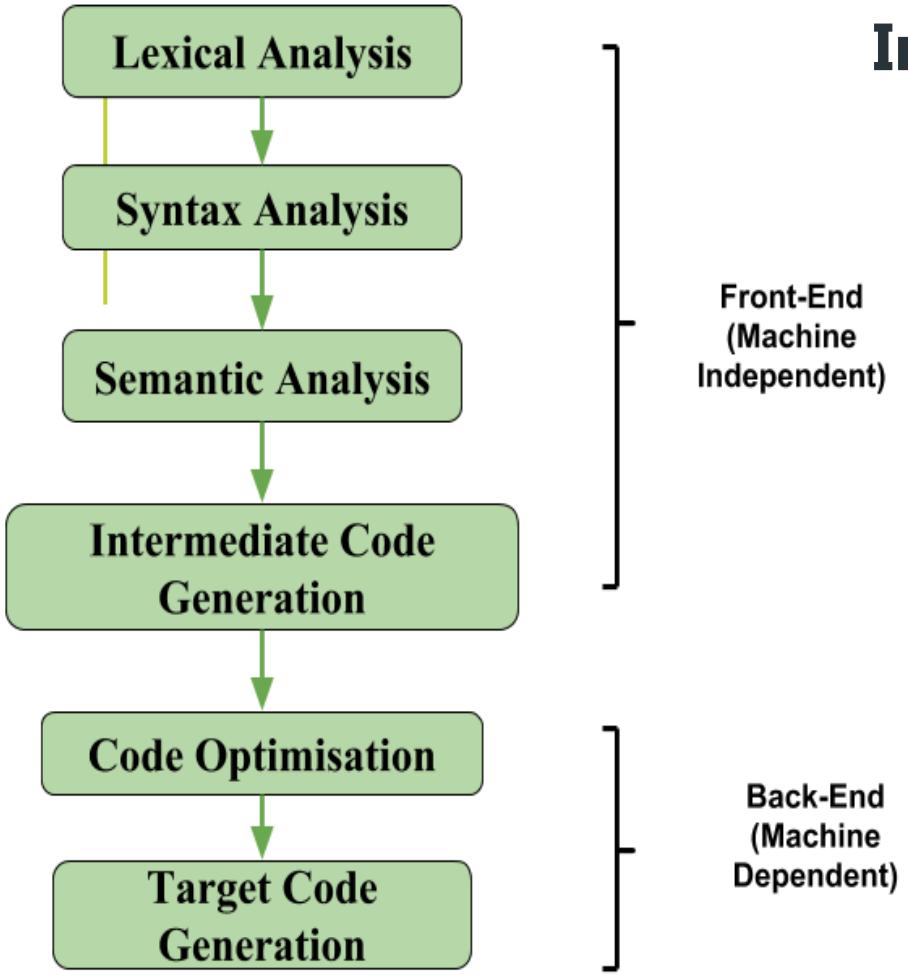


Figure 6.1: Logical structure of a compiler front end

Assume that the front end a compiler is organized as shown above;

Where Parsing, Static checking, and Intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing.

Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing. **For example,** static checking assures that a break-statement in C is enclosed within a while-, for-, or switch-statement; an error is reported if such an enclosing statement does not exist.



Benefits of using Machine-Independent Intermediate code are:

1. Because of the machine-independent intermediate code, portability will be enhanced.
2. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
3. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.

NEED FOR INTERMEDIATE CODE

1.Suppose we have x no of the source language and y no of the target language:

1. **Without ICG** – we have to change each source language into target language directly, So, for each source-target pair we will need a compiler .Hence we need (x*y) compilers, which can be a very large number and which is literally impossible.

2. **With ICG –** we will need only x number of compiler to convert each source language into intermediate code. We will also need y compiler to convert the intermediate code into y target languages. so, we will need only (x+y) no of the compiler with ICG which is way lesser than x*y no of the compiler.

2.Re-targeting is facilitated : a compiler for a different machine can be created by attaching a back-end(which generate target code) for the new machine to an existing front-end(which generate intermediate code).

NEED FOR INTERMEDIATE CODE

3. **Machine independent:** A Machine independent code-optimizer can be applied to the intermediate code. So this can be run on any machine.

4. **Simplicity:** Intermediate code is simple enough to be easily converted to any target code. So ICG reduces the overhead of target code generation.

5. **Complexity :** Intermediate code is Complex enough to represent all complex structure of high-level languages.

6. **Modification:** We can easily modify our code to get better performance and throughput by applying optimization technique to the Intermediate code.

INTERMEDIATE REPRESENTATION

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as

1. **High-Level IR:** High-level intermediate codes closely resemble the source language, making them simple to build from source code. This also makes it easier to update the code to enhance execution. **E.g. Syntax Tree**
2. **Low-Level IR** – A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.

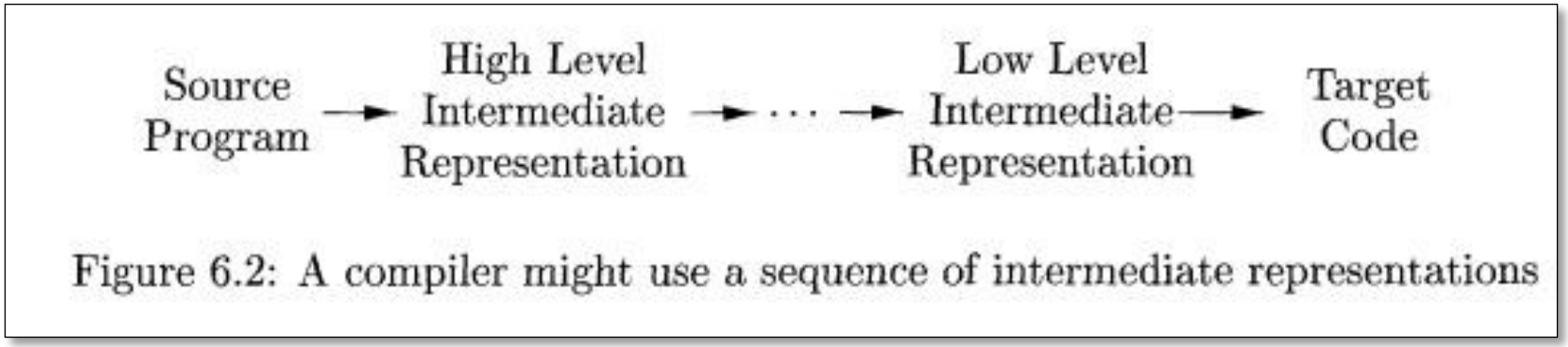


Figure 6.2: A compiler might use a sequence of intermediate representations

The following are commonly used intermediate code representations:

POSTFIX NOTATION, SYNTAX TREES, THREE ADDRESS CODE

POSTFIX NOTATION

- Compilers find it difficult to distinguish between the operators and parentheses. So, for this, we have postfix notation in compiler design.
- Postfix notation is also known as **Reverse Polish Notation**.
- In this notation, the operators are written after the operands, not like the infix in which the operator is in-between the operands.
- For example, the infix notation (5+6)*7 will be written as 56+7* in postfix notation.
- The postfix representation of the expression

(a – b) * (c + d) + (a – b)

ab – cd + *ab -+

ADVANTAGES OF POSTFIX NOTATION

1. **No need for parentheses:** In polish notation, there is no need for parentheses while writing the arithmetic expressions as the operators come before the operands.

2. **Efficient Evaluation:** The evaluation of an expression is easier in polish notation because in polish notation stack can be used for evaluation.

3. **Easy parsing:** In polish notation, the parsing can easily be done as compared to the infix notation.

4. **Less scanning:** The compiler needs fewer scans as the parentheses are not used in the polish notations, and the compiler does not need to scan the operators and operands differently.

Why is stack used for RPN?

The use of a stack is advantageous for Reverse polish notation evaluation because it simplifies the process of keeping track of operands and operators. The stack provides a convenient way to store and retrieve the operands, and the last two operands can be easily retrieved when an operator is encountered.

THREE ADDRESS CODE

Three-address code is a linearized representation of a Syntax Tree or a DAG in which explicit names correspond to the interior nodes of the graph

1. Addresses and Instructions
2. Quadruples
3. Triples
4. Indirect Triples
5. Static Single-Assignment Form

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
Thus a source-language expression like **x+y*z** might be translated into the sequence of three-address instructions

$$t1 = y * z$$
$$t2 = x + t1$$

where t1 and t2 are compiler-generated temporary names. This solving of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

t2 = a * t1

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

t2 = a * t1

t3 = a + t2

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

EXAMPLE OF 3AC & ITS CORRESPONDING DAG

a + a * (b – c) + (b – c) * d

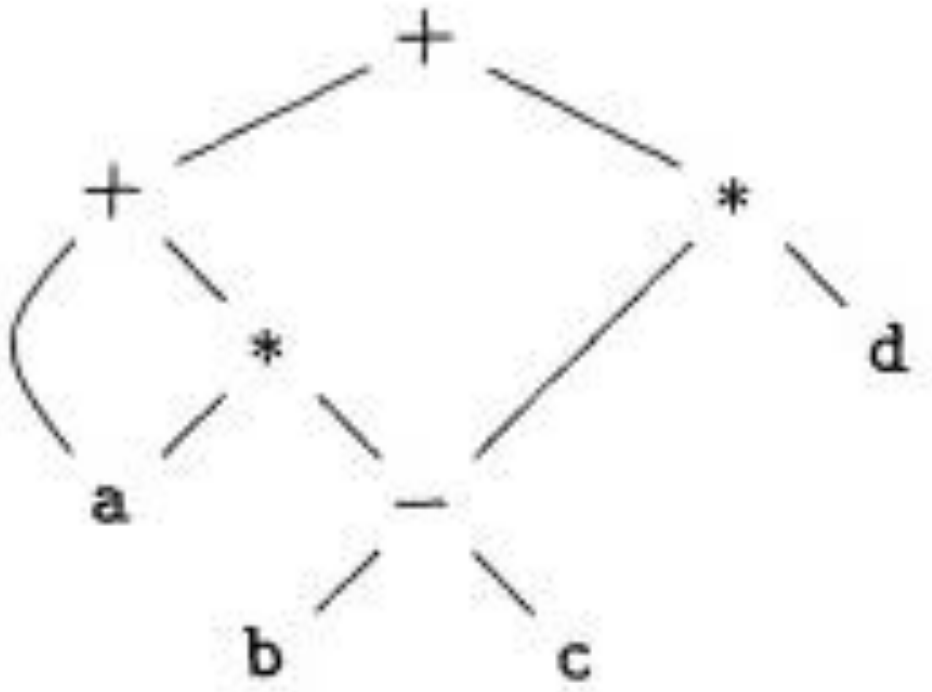
t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

t5 = t3 + t4



(a) DAG

QUADRUPLES

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure.

In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called “QUADRUPLES," “TRIPLES," and “INDIRECT TRIPLES.“

A quadruple (or just "quad!') has four fields, which we call op, arg1: arg2, and result.

For instance, the three-address instruction x = y + z is represented by

placing + in op,
y in arg1,
z in arg2, and
x in result. The following are some exceptions to this rule:

EXCEPTIONS TO THE RULES OF QUADRUPLES

1. Instructions with unary operators like x = minus y or x = y do not use arg2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.

2. Operators like param use neither arg2 nor result.

3. Conditional and unconditional jumps put the target label in result.

Three-address code for the assignment a = b * - c + b * - c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

TRIPLES

- A ***TRIPLE*** has only three fields, which we call ***op, arg_x,* and *arg_2.***
- The ***result*** field in QUADRUPLES is used primarily for temporary names.
- Using triples, we refer to the result of an operation ***x*** *op* *y* by its position, rather than by an explicit temporary name.
- Thus, instead of the temporary t1 a triple representation would refer to position (0).
- Parenthesized numbers represent pointers into the triple structure itself.

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

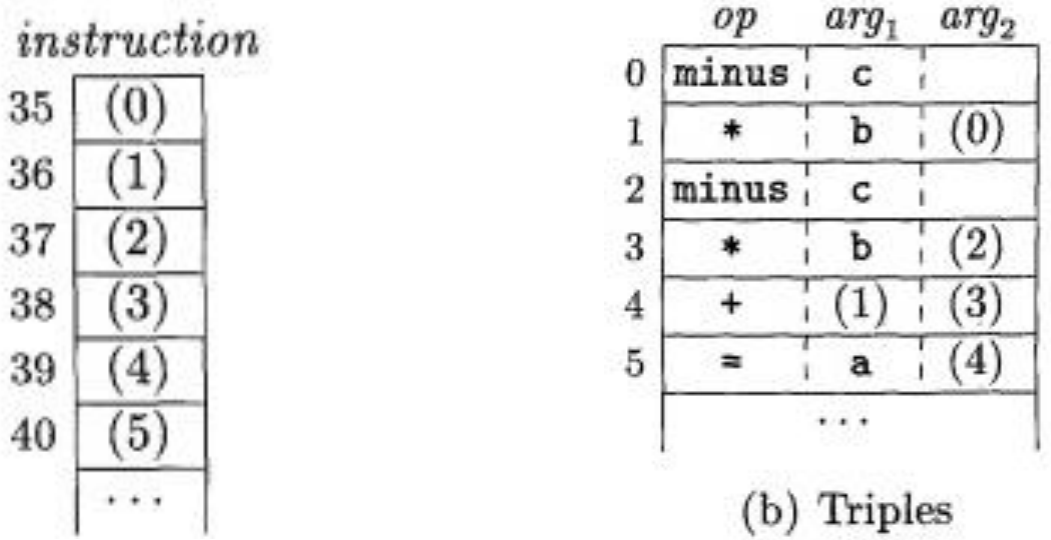
(b) Triples

INDIRECT TRIPLES

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

For example, let us use an array *instruction* to list pointers to triples in the desired order.

Then, the triples might be represented as follows;



	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

STATIC SINGLE-ASSIGNMENT FORM (SSA)

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
- All assignments in SSA are to variables with distinct names; hence the term *static single-assignment.*

SSA is a property of an intermediate representation, which requires that each variable is assigned exactly once , and every variable is defined before it is used.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( f l a g )
        x = -1;
 e l s e
        x =  1;
y = x * a;
```

Has two control-flow paths in which the variable x gets defined.

STATIC SINGLE-ASSIGNMENT FORM

- $\emptyset - function$ function to combine the two definitions of x:

```
if ( f l a g )
        x1 = -1;
 e l s e
        x2 = 1;
x3 = Ø(x1, x2);
```

- Here, ϕ(x1,x2) has the value x1 if the control flow passes through the true part of the conditional
- And the value x2 if the control flow passes through the false part.
- That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

SSA uses a notational convention called the $\emptyset - function$

DIRECTED ACYCLIC GRAPH

Nodes in a **syntax tree** represent constructs in the source program; the children of a node represent the meaningful components of a construct.

A **Directed Acyclic Graph (hereafter called a *DAG)*** for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression.

DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Directed Acyclic Graph (DAG) is a special kind of **Abstract Syntax Tree**.

Each node of it contains a unique value.

It does not contain any cycles in it, hence called **Acyclic**.

A DAG is constructed for optimizing the basic block.

A DAG is usually constructed using **Three Address Code**.

Transformations such as dead code elimination and common sub expression elimination are then applied.

Applications- **DAGs are used for the following purposes-**

1. To determine the expressions which have been computed more than once (called common sub-expressions).
2. To determine the names whose computation has been done outside the block but used inside the block.
3. To determine the statements of the block whose computed value can be made available outside the block.
4. To simplify the list of **Quadruples** by not executing the assignment instructions x:=y unless they are necessary and eliminating the common sub-expressions.

CODE OPTIMIZATION

- Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called **"code improvement" or "code optimization.“**

- Compilers that apply code-improving transformations are called optimizing compilers. Optimizations are classified into two categories. They are

  a) **Machine Independent Optimizations: Ar**e program transformations that improve the target code without taking into consideration any properties of the target machine.

  b) **Machine Dependent Optimizations:** Are based on register allocation and utilization of special machine-instruction sequences.

CRITERIA FOR CODE OPTIMIZATION

- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program.

- A transformation must, on the average, speedup programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

Flow analysis is a fundamental prerequisite for many important types of code improvement. Generally control flow analysis precedes data flow analysis.

1. **Control flow analysis (CFA) represents** flow of control usually in form of graphs, CFA constructs such as control flow graph, Call graph.

2. **Data flow analysis (DFA)** is the process of asserting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

WHY TO OPTMIZE CODE

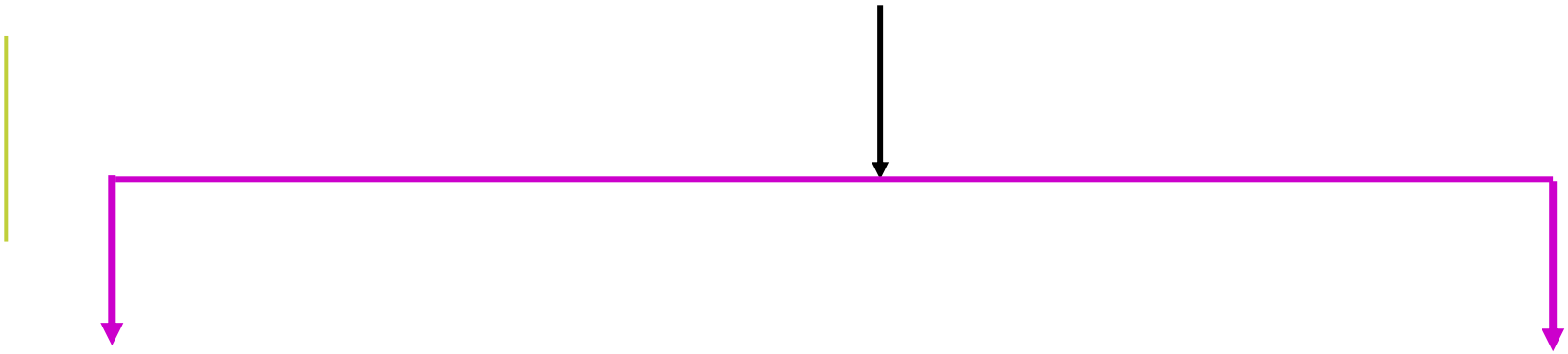
- Optimizing code in compiler design is important because it directly affects the performance of the compiled code. A well-optimized code runs faster and consumes fewer resources, leading to improved overall system performance and reduced energy consumption.

- Additionally, optimization can reduce the size of the generated code, which is important for embedded systems with limited memory.

- The optimization process can also help identify and eliminate bottlenecks in the code, leading to more efficient algorithms and improved software design.

- Overall, optimization is a critical step in the compiler design process that can greatly improve the end-user experience.

TYPES OF CODE OPTIMIZATION



Machine Independent

1. Constant Folding & Propagation
2. Common Subexpression Elimination
3. Copy Propagation
4. Dead Code Elimination
5. Algebraic Simplification & Strength Reduction
6. Loop Optimization
   - Code Movement or Frequency Reduction
   - Loop Jamming
   - Loop Unrolling

Machine Dependent

1. Peephole Optimization

CONSTANT FOLDING & PROPAGATION

> Constant Folding & Constant Propagation is clubbed in one technique known as **COMPILE TIME EVALUATION**

Constant Folding: This is a powerful optimization technique that **involves evaluating constant expressions at compile time and replacing them with their computed results.** By performing computations during compilation rather than runtime, constant folding eliminates the need for repetitive calculations, resulting in faster and more efficient code execution. The technique focuses on simplifying and optimizing code by replacing the original expressions with their resolved values.

For e.g. C = 22 / 7 * r * r **INSTEAD WRITE** C = 3.14*r*r

By eliminating the need for runtime computations, constant folding improves code efficiency and enhances the overall performance of the program.

CONSTANT FOLDING - EXAMPLE

Suppose we have been given a code snippet?

```
1. int main(){
2. int a=1*4-9;
3. int b=a-1;
4. cout<<b;
5. }
```

In which lines of code would the compiler use the constant folding to reduce the overall execution time.

OUTPUT AFTER CONSTANT FOLDING IS:

```
int main(){
    int a = -5
    int b = a - 1;
    cout<<b;
    }
```

CONSTANT FOLDING & PROPAGATION

> Constant Folding & Constant Propagation is clubbed in one technique known as **COMPILE TIME EVALUATION**

Constant Propagation: is a local optimization technique that substitutes the values of variables and expressions whose values are known beforehand In other words, Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

```
int a = 4;
int b = (a+1)*2;
```

INSTEAD WRITE

```
int b = (4+1)*2;
```

CONSTANT FOLDING

```
int a =4;
int b = 10;
```

EXAMPLE

OPTIMIZE THE FOLLOWING CODE

```
int func(){
    int a = 30;
    int b = 9 - a / 5;
    int c;
    c = b * 4;
    if (c > 10) {
        c = c - 10;
    }
    return c * (60 / a);
}
```

After applying COMPILE TIME EVALUATION

```
int funct(){
    int a = 30;
    int b = 3;
    int c;
    c = 12;
    if (12 > 10) {
        c = c - 10;
    }
    return c * 2;
}
```

Common Subexpression Elimination (CSE)

- ✓ An occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.

- ✓ We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.

- ✓ Elimination of such an expression is known as Common sub-expression elimination. The advantage of this elimination method is to make the computation faster and better by avoiding the re-computation of the expression. In addition, it utilizes memory efficiently.

- ✓ THUS **CSE** is about eliminating redundant expressions.

EXAMPLE:

```
a = b * c + g
d = b * c  * h
```

POST CSE

```
temp = b * c
a = temp + g
d = temp * h
```

TYPES OF CSE

- The two types of elimination methods in common sub-expression elimination are:

1. **Local Common Sub-expression elimination–** It is used within a single basic block. Where a basic block is a simple code sequence that has no branches.
2. **Global Common Sub-expression elimination–** It is used for an entire procedure of common sub-expression elimination.

```
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
```

POST CSE

The assignments to t7 and t10 compute the common subexpressions 4 * i and 4 * j, respectively. These steps have been eliminated which uses t6 instead of t7 and t8 instead of t10 .

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

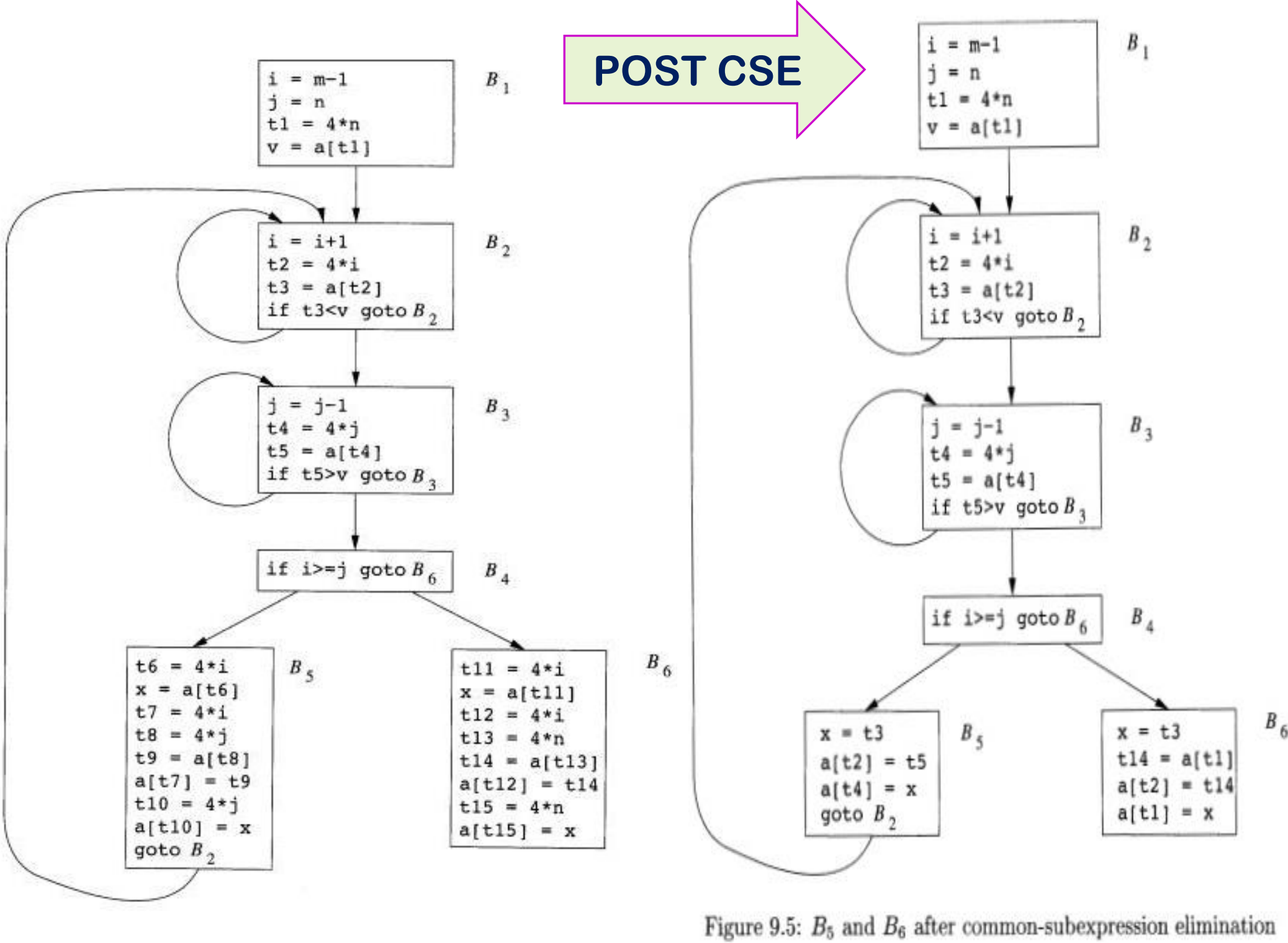


Figure 9.5: B_5 and B_6 after common-subexpression elimination

Copy Propagation

- ✓ Copy Propagation is an optimization technique used in compiler design.
- ✓ The assignments of the form u = v are called as ***Copy Statements or Copies***
- ✓ It replaces the irrelevant variable assignments with the previously calculated values that are not changed in the future. this approach improves memory utilization, and efficiency of the code and therefore requires less time for overall computation

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

POST COPY PROPAGATION

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

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

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:

```
if(1)
  printf("SPCC lec");
else
    printf("Free Lec");
```

```
#include <stdio.h>
int main() {
    int x = 5;
    int y = 10;
    int z;
    z = x + y;
   printf("The value of x is: %d\n", x);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    int x = 5;
    printf("The value of x is: %d\n", x);
    return 0;
}
```

EXAMPLE

```
int funct(){
    int a = 30;
    int b = 3;
    int c;
    c = 12;
    if (c > 10) {
        c = c - 10;
    }
    return c * 2;
}
```

After applying Dead Code Elimination

```
int funct()
{
        return 4;
}
```

Algebraic Simplification & Strength Reduction

- ✓ Algebraic identities represent another class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

  x + 0 = 0 + x = x

  x * 1 = 1 * x = x

  b || TRUE = TRUE

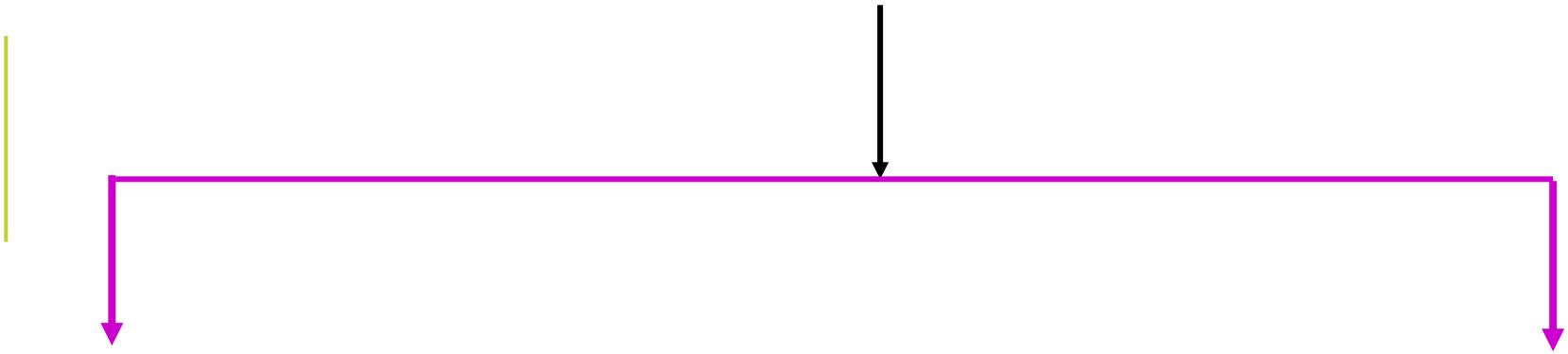
  b || FALSE = b

to eliminate computations from a basic block.

- ✓ **Strength Reduction** – replacing expensive machine instructions by cheaper ones

EXPENSIVE OPERATION	CHEAPER OPERATION
x^2	$x * x$
$2*x$	$x + x$
$x / 2$	$x * 0.5$

TYPES OF CODE OPTIMIZATION



Machine Independent

1. Constant Folding & Propagation
2. Common Subexpression Elimination
3. Copy Propagation
4. Dead Code Elimination
5. Algebraic Simplification & Strength Reduction
6. Loop Optimization
   - Code Movement or Frequency Reduction
   - Loop Jamming / Loop Fusion
   - Loop Unrolling

Machine Dependent

1. Peephole Optimization

LOOP OPTIMIZATION

- Loop optimization is a code optimization technique that focuses on improving the performance of loops in computer programs. It aims to reduce the number of operations or iterations required to complete a loop.

- Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops.

- It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

Loop Optimization is a machine independent optimization.

Whereas

Peephole optimization is a machine dependent optimization technique.

Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

LOOP OPTIMIZATION

Code Movement / Code Motion

Also known as Frequency Reduction or Loop Invariant

In frequency reduction, the amount of code in the loop is decreased.

A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

Only those statements or expressions that do not affect a program's semantics and give the same result even after not shifting the statements can be placed just before the loop.

Before optimization:

```
while(i<100)
{
 a = Sin(x)/Cos(x) + i;
 i++;
}
```

After optimization:

```
t = Sin(x)/Cos(x);
while(i<100)
{
 a = t + i;
 i++;
}
```

Code Movement / Code Motion

EXAMPLE 2

Before optimization:

```
a = 100
while(a > 0)
{
    x = y + z
    if(a % x = = 0)
        printf("%d", a);
}
```

After optimization:

```
a = 100
x = y + z
while(a > 0)
{
    if(a % x = = 0)
        printf("%d", a);
    a - -;
}
```

LOOP OPTIMIZATION

Loop Jamming / Loop Fusion

This technique combines two or more loops which have the same index variable and number of iterations.

This technique reduces the time taken in compiling all the loops.

Before optimization:

```
#include <stdio.h>
 void main() {
    int a[10], b[10], i;
    for(i = 0; i < 10; i++)
        a[i] = 1;
    for(i = 0; i < 10; i++)
        b[i] = 2;
}
```

After optimization:

```
#include <stdio.h>
 void main() {
    int a[10], b[10], i;
    for(i = 0; i < 10; i++) {
        a[i] = 1;
        b[i] = 2;
    }
}
```

LOOP OPTIMIZATION

Loop Unrolling

- ✓ Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.
- ✓ In this, either the loop is removed or iterations are reduced. It expands the content of a control loop, and in some cases eliminates the need of the control loop entirely.
- ✓ The goal of this technique is to minimize the execution time of a coded algorithm. Unrolling loops helps to minimize the number instructions because the number of end of loops check is reduced, and it helps the compiler generate assembly that uses parallel instructions.
- ✓ Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

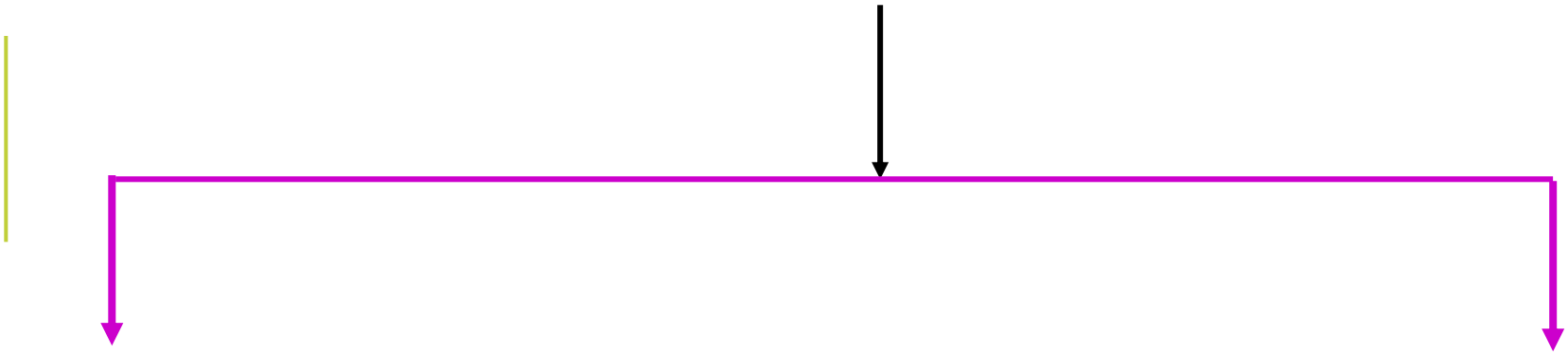
Before optimization:

```
for(i=0;i<10;i++)
        printf("Hi");
```

After optimization:

```
for(i=0;i<10;i=i+2) {
        printf("Hi");
        printf("Hi");  }
```

TYPES OF CODE OPTIMIZATION



Machine Independent

1. Constant Folding & Propagation
2. Common Subexpression Elimination
3. Copy Propagation
4. Dead Code Elimination
5. Algebraic Simplification & Strength Reduction
6. Loop Optimization
   - Code Movement or Frequency Reduction
   - Loop Jamming / Loop Fusion
   - Loop Unrolling

Machine Dependent

1. Peephole Optimization

PEEPHOLE OPTIMIZATION

- Code optimization that is applied to a small section of the code is known as peephole optimization in compiler design.

- It is called local optimization because it works by evaluating a small section of the generated code, generally a few instructions, and optimizing them based on some predefined rules.

- The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.

- It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

OBJECTIVES OF PEEPHOLE OPTIMIZATION

The following are the objectives of peephole optimization in compiler design:

- **Increasing code speed:** Peephole optimization in compiler design seeks to improve the execution speed of generated code by removing redundant instructions or unnecessary instructions.
- **Reduced code size:** Peephole optimization in compiler design seeks to reduce generated code size by replacing the long sequence of instructions with shorter ones.
- **Getting rid of dead code:** Peephole optimization in compiler design seeks to get rid of dead code, such as unreachable code, redundant assignments, or constant expressions that have no effect on the output of the program.
- **Simplifying code:** Peephole optimization in compiler design also seeks to make generated code more understandable and manageable by removing unnecessary complexities.

WORKING OF PEEPHOLE OPTIMIZATION

The working of Peephole optimization in compiler design can be summarized in the following steps:

Step 1 – Identify the peephole: In the first step, the compiler finds the small sections of the generated code that needs optimization.

Step 2 – Apply the optimization rule: After identification, in the second step, the compiler applies a predefined set of optimization rules to the instructions in the peephole.

Step 3 – Evaluate the result: After applying optimization rules, the compiler evaluates the optimized code to check whether the changes make the code better than the original in terms of speed, size, or memory usage.

Step 4 – Repeat: The process is repeated by finding new peepholes and applying the optimization rules until no more opportunities to optimize exists.

PROGRAM TRANSFORMATIONS USED IN PEEPHOLE OPTIMIZATION

1. **Eliminating Redundant Loads and Stores:** Redundant load and store elimination is also one of the peephole optimization techniques that seeks to reduce redundant memory accesses in a program. This optimization works by finding code that performs the same memory access many times and removes the redundant accesses.

If we see the instruction sequence

```
LD a, R0
ST R0,  a
```

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register R0.

PROGRAM TRANSFORMATIONS USED IN PEEPHOLE OPTIMIZATION

2. **Eliminating Unreachable Code:** Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1. In the intermediate representation, this code may look like

```
if debug == 1 goto LI
goto L2

L I : print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what is the value of debug, the code sequence above can be replaced by

```
if debug != 1 goto L2
print debugging information
L2:
```

PROGRAM TRANSFORMATIONS USED IN PEEPHOLE OPTIMIZATION

```
        if debug != 1 goto L2
        print debugging information
L2:
```

If **`debug` is set to 0** at the beginning of the program, constant propagation would transform this sequence into

```
        if 0 != 1 goto L2
        print debugging information
L2:
```

Now the argument of the first statement always evaluate to ***true,*** so the statement can be replaced by **`goto L2.`** Then all statements that print debugging information are unreachable and can be eliminated one at a time.

PROGRAM TRANSFORMATIONS USED IN PEEPHOLE OPTIMIZATION

3. **Flow-of-Control Optimizations:** Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

For example; The sequence

```
if a < b goto LI
 LI: goto L2
```

Can be replaced by the sequence

```
if a < b goto L2
 LI: goto L2
```

PROGRAM TRANSFORMATIONS USED IN PEEPHOLE OPTIMIZATION

4. **Algebraic Simplification and Reduction in Strength:** Algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

$$x = x + 0$$

or

$$x = x * 1$$

in the peephole.

Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine.

Code Generation:

Issues in the design of code generator

Code generation algorithm.

Basic block and flow graph.

CODE GENERATION

- The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program

- The **code generation techniques** presented below **can be used whether or not an optimizing phase occurs before code generation**

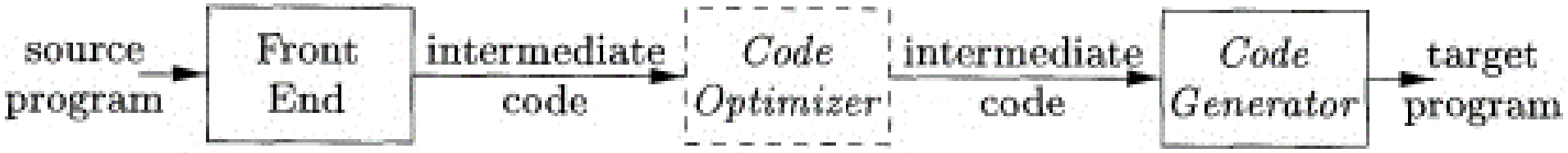


Figure 8.1: Position of code generator

The requirements imposed on a code generator are severe. The target program **must preserve the semantic meaning of the source program and be of high quality**; that is, it **must make effective use of the available resources** of the target machine. Moreover, the code generator itself must run efficiently.

CODE GENERATION

- A code generator has three primary tasks:
  - Instruction selection,
  - Register allocation and assignment, and
  - Instruction ordering.

a) **Instruction selection** involves choosing appropriate target-machine instructions to implement the IR statements.

b) **Register allocation and assignment** involves deciding what values to keep in which registers.

c) **Instruction ordering** involves deciding in what order to schedule the execution of instructions.

ISSUES IN THE DESIGN OF A CODE GENERATOR

1) Input to code generator

- The input to the code generator consists of the intermediate representation of the source program produced by front end , together 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, DAGs, etc. The code generation phase just proceeds on an assumption that the input is free from all syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

ISSUES IN THE DESIGN OF A CODE GENERATOR

2) The Target Program

- The target program is the output of the code generator. The output may be **absolute machine language, relocatable machine language, or assembly language.**
- **Absolute machine language as output** has the advantages that it can be placed in a fixed memory location and can be immediately executed. For example, WATFIV is a compiler that produces the absolute machine code as output.
- **Relocatable machine language as an output** allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by a 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 the macro-facilities of assemblers in generating code. And we need an additional assembly step after code generation.

ISSUES IN THE DESIGN OF A CODE GENERATOR

3) 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 play a major role when efficiency is considered.

But if we do not care about the efficiency of the target program then instruction selection is straightforward.

For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form **x = y + z**, can be translated into the code sequence

```
LD   R0, y        // R0 = y          (load y into register R0)
ADD  R0, R0, z    // R0 = R0 + z     (add z to R0)
ST   x, R0        // x = R0          (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

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

would be translated into

```
LD  R0, b        // R0 = b
ADD R0, R0, c    // R0 = R0 + c
ST  a, R0        // a = R0
LD  R0, a        // R0 = a
ADD R0, R0, e    // R0 = R0 + e
ST  d, R0        // d = R0
```

Here, **the fourth statement is redundant** since it loads a value that has just been stored, and so is the third if **a** is not subsequently used.

- The quality of the generated code is usually determined by its speed and size.
- On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations.
- A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

3) Instruction Selection (Contd..)

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 stores the result back into a:

```
LD   R0, a        // R0 = a
ADD  R0, R0, #1   // R0 = R0 + 1
ST   a, R0        // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine-code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

ISSUES IN THE DESIGN OF A CODE GENERATOR

4) 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 is subdivided into two subproblems:
    1. During **Register allocation –** we select only those sets 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.

CODE GENERATION ALGORITHM

1. **Code generator** is used to produce the target code for three-address statements.
2. It uses registers to store the operands of the three address statement.
3. It sets up register and address descriptors, then generates machine instructions that give you CPU-level control over your program.

Register and Address Descriptors:

- ✓ **A register descriptor** contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.

- ✓ **An address descriptor** is used to store the location where current value of the name can be found at run time.

CODE GENERATION ALGORITHM

The algorithm takes as **input a sequence of three-address statements** constituting a basic block. For each three-address statement of the form **x : = y op z**, perform the following actions:

Code Generation Algorithm:

- ✓ Invoke a function ***getreg()*** to determine the location L where the result of the computation y op z should be stored.

- ✓ Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

- ✓ Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors

- ✓ If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z

CODE GENERATION - EXAMPLE

Example:- d : = (a-b) + (a-c) + (a-c)

Three Address Code→

t = a - b
u = a - c
v = t + u
d = v + u

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

BASIC BLOCKS & FLOW GRAPH

- **Basic Block** is a straight line code sequence that has no branches in and out branches except to the entry and at the end respectively.

- Basic Block is a set of statements that always executes one after other, in a sequence.

Properties of Basic Blocks

- The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

- Control will leave the block without halting or branching, except possibly at the last instruction in the block.

HOW TO CREATE BASIC BLOCKS

- *Our first job is to partition a sequence of three-address instructions into basic blocks.*
- *We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction.*
- *In the absence of jumps and labels, control proceeds sequentially from one instruction to the next.*

ALGORITHM : 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.

ALGORITHM : PARTITIONING THREE-ADDRESS INSTRUCTIONS INTO BASIC BLOCKS. (CONTD....)

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.

EXAMPLE

Rule 3 says **each instruction following a jump is a leader**

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

Leader, By Rule 1

Leader, By Rule 2

Leader, By Rule 2

Leader, By Rule 3

Leader, By Rule 3

Leader, By Rule 2

Rule 2 says Target of Jumps are LEADERS

To find the other leaders, we first need to find the jumps.

EXAMPLE

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

B1

```
1)   i = 1
```

B2

```
2)   j = 1
```

B3

```
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
```

B4

```
10)  i = i + 1
11)  if i <= 10 goto (2)
```

B5

```
12)  i = 1
```

B6

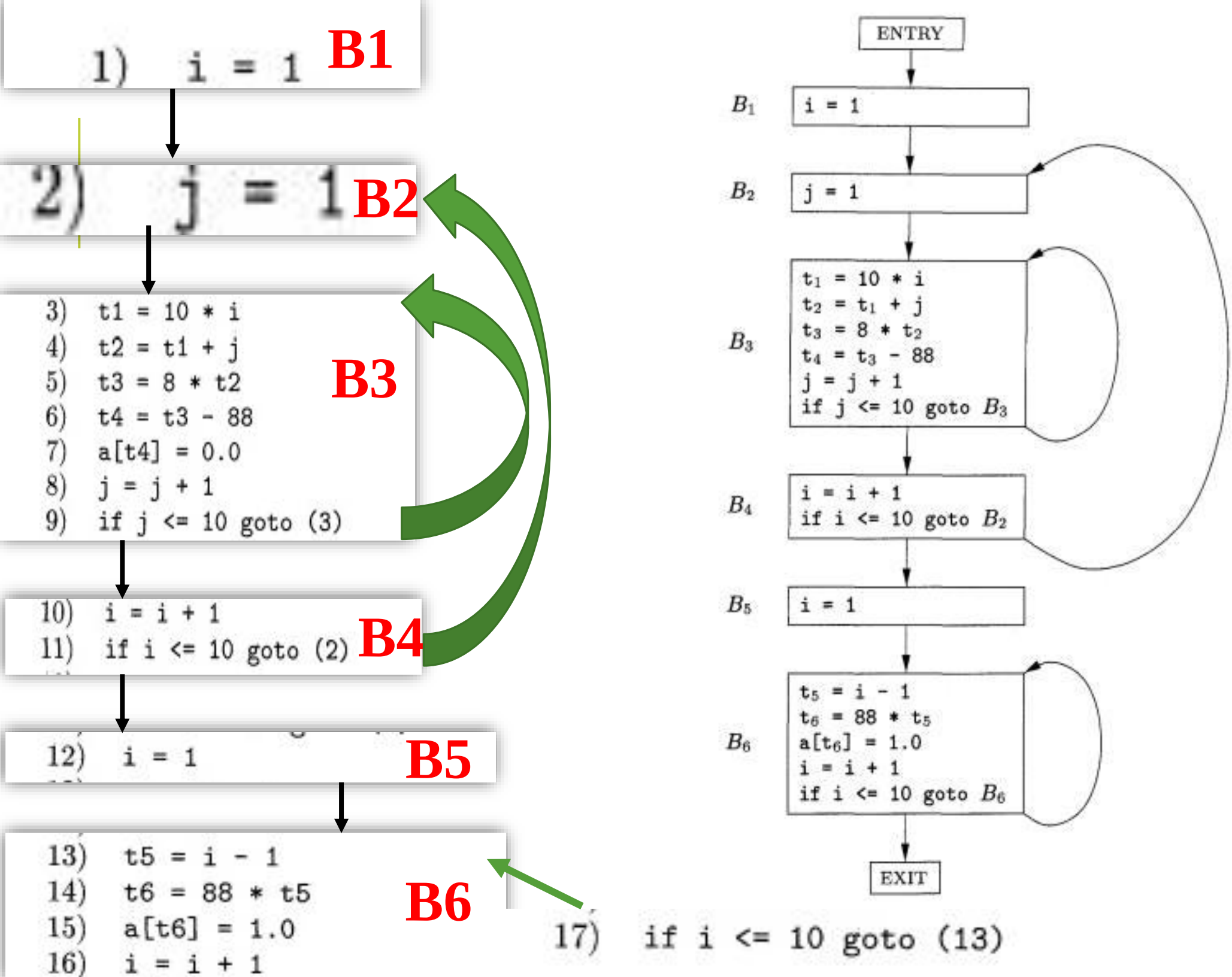
```
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Flow Graph

A directed graph in which the flow of control information is added to the basic blocks.

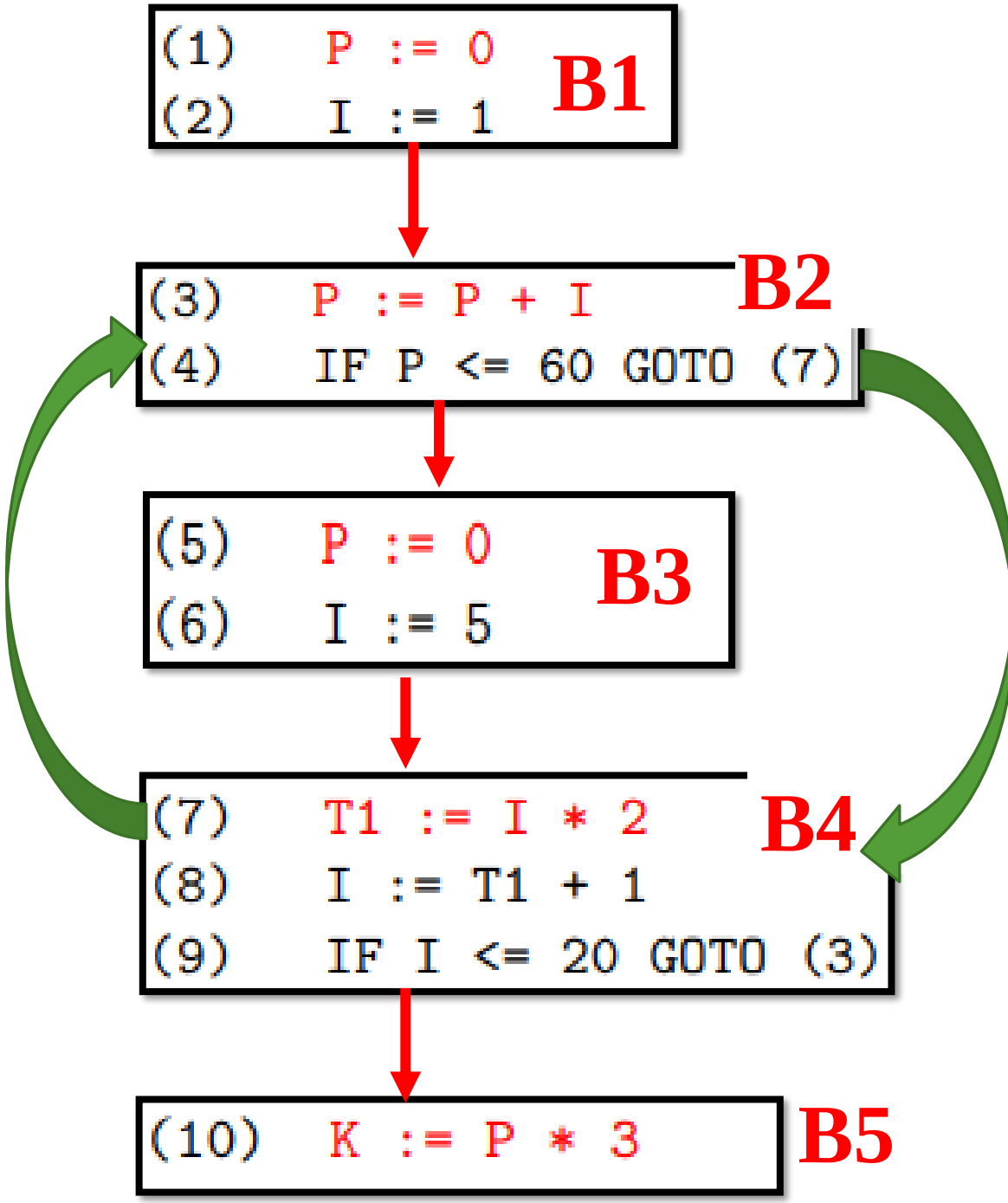
Rules:

1. Basic Blocks are the nodes of a flow graph
2. The block which has the initial statement of the code is the initial block.
3. An edge is drawn from B1 to B2, if B2 immediately follows B1


1) i = 1
B1
2) j = 1
B2
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
B3
10) i = i + 1
11) if i <= 10 goto (2)
B4
12) i = 1
B5
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
B6
ENTRY
B1
i = 1
B2
j = 1
B3
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
j = j + 1
if j <= 10 goto B3
B4
i = i + 1
if i <= 10 goto B2
B5
i = 1
B6
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
EXIT

EXAMPLE 2

```
(1)    P := 0
(2)    I := 1
(3)    P := P + I
(4)    IF P <= 60 GOTO (7)
(5)    P := 0
(6)    I := 5
(7)    T1 := I * 2
(8)    I := T1 + 1
(9)    IF I <= 20 GOTO (3)
(10)   K := P * 3
```



EXAMPLE 3

(1) PROD = 0

(2) I = 1

(3) T2 = addr(A) – 4

(4) T4 = addr(B) – 4

(5) T1 = 4 x I

(6) T3 = T2[T1]

(7) T5 = T4[T1]

(8) T6 = T3 x T5

(9) PROD = PROD + T6

(10) I = I + 1

(11) IF I <=20 GOTO (5)

EXAMPLE 4

```
(1)  i := 1
(2)  IF i>n GOTO (14)
(3)  IF i>=n GOTO (6)
(4)  x := A[i]
(5)  GOTO (11)
(6)  IF x<=4 GOTO (11)
(7)  T1 := x*2
(8)  T2 := A[i]
(9)  x := T1+T2
(10)  GOTO (6)
(11)  x := x+5
(12)  i := i+1
(13)  GOTO (2)
```

INDUCTION VARIABLE ELIMINATION

- Another important optimization is to find induction variables in loops and optimize their computation.

- A variable *x* is said to be an **"induction variable"** if there is a positive or negative constant c such that each time *x* is assigned, its value increases by c.

TYPES OF THREE ADDRESS STATEMENTS

PENDING

UNIVERSITY QUESTIONS

- ✓ Explain the different code optimization techniques in compiler design.
- ✓ Explain different types of Intermediate code representations.
- ✓ What is the need of Intermediate code generation? Explain any two Intermediate Code Generation forms with example
- ✓ Explain Machine Independent Code Optimization techniques
- ✓ Write Short Note on
  - ✓ Code generation issues
  - ✓ Peephole Optimization
- ✓ Explain the concept of basic blocks and flow graph with example
- ✓ What is Code optimization? Explain with example the following code optimization techniques
    - ✓ Common Subexpression Elimination
    - ✓ Code motion
    - ✓ Dead Code Elimination
    - ✓ Constant Propagation

UNIVERSITY QUESTIONS

(a) Apply dead code elimination techniques for following code

```
int count;
void foo( )
{
    int i;
    i=1;
    count=1;
    count=2;
    return
    count=3;
}
```

Generate three address code for following code

```
while(a<b) do
    if(c<d) then
            x=y+2
    else
            x=y-2
```

UNIVERSITY QUESTIONS

Construct Three address code for the following program

```
For(i=0;i<10;i++)
{
If (i<5)
a=b+c*3;
else
x=y+z;
}
```

B.	What are the different ways of representing Intermediate code? Explain with suitable example.	**10**
A.	Explain different issues in code generation phase of compiler.	**10**
B.	Construct DAG for the following expression **x= m+p/q – t+ p/q *y**	**10**

UNIVERSITY QUESTIONS

What is three-address code? Generate three-address code for – **5**

```
while (a<b) do
    if(c<d) then
        x:=y+z
    else
        x:=y-z
```