



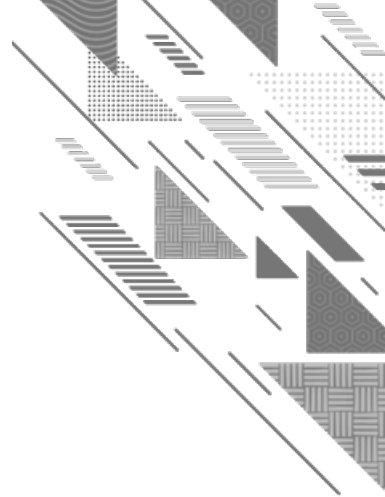

## Unit – 7

# Code Generation & Optimization





# Topics to be covered

- Issues in the design of a code generator
  - The Target machine
  - Basic block and flow-graph
  - Transformation on basic block
  - A simple code generator
  - Code optimization
- 
- 



# Issues in the design of a code generator

# Issues in design of Code Generator

## ► Issues in Code Generation are:

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

# Input to code generator

- ▶ Input to the code generator consists of the intermediate representation of the source program.
- ▶ Types of intermediate language are:
  1. Postfix notation
  2. Three address code
  3. Syntax trees or DAGs
- ▶ 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.

# Target program

► The output may be in form of:

1. **Absolute machine language:** Absolute machine language program can be placed in a memory location and immediately execute.
2. **Relocatable machine language:** The subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution.
3. **Assembly language:** Producing an assembly language program as output makes the process of code generation easier, then assembler is require to convert code in binary form.

# Memory management

- ▶ **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.

# Instruction selection

- ▶ 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

MOV b, R0

ADD c, R0

ADD e, R0

MOV R0, d

- ▶ So, we can eliminate redundant statements.



# Register allocation

- ▶ The use of registers is often subdivided into two sub problems:
- ▶ During **register allocation**, we select the **set of variables** that will reside in registers at a point in the program.
- ▶ During a subsequent **register assignment** phase, we pick the **specific register** that a variable will reside in.
- ▶ Finding an optimal assignment of registers to variables is difficult, even with single register value.
- ▶ Mathematically the problem is **NP-complete**.

# 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 intermediate results than others.
- ▶ Picking a best order is another difficult, **NP-complete problem**.

# Approaches to code generation

- ▶ The most important criterion for a code generator is that it produces correct code.
- ▶ The design of code generator should be in such a way so it can be implemented, tested, and maintained easily.



# The Target Machine

# Target machine

- ▶ We will assume our target computer models a three-address machine with:
  1. load and store operations
  2. computation operations
  3. jump operations
  4. conditional jumps
- ▶ The underlying computer is a byte-addressable machine with  $n$  general-purpose registers,  $R_0, R_1, \dots, R_n$

# Addressing Modes

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	$k(R)$	$k + \text{contents}(R)$	1
Indirect register	$*R$	$\text{contents}(R)$	0
Indirect indexed	$*k(R)$	$\text{contents}(k + \text{contents}(R))$	1

# Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

► Calculate cost for following:

MOV B,R0 ADD C,R0 MOV R0,A	

# Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

► Calculate cost for following:

MOV *R1 ,*R0 MOV *R1 ,*R0	





# Basic blocks and flow graph



# Basic Blocks

- ▶ A basic block is a **sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end** without halt or possibility of branching except at the end.
- ▶ The following sequence of three-address statements forms a basic block:

$t1 := a * a$

$t2 := a * b$

$t3 := 2 * t2$

$t4 := t1 + t3$

$t5 := b * b$

$t6 := t4 + t5$

# Algorithm: Partition into basic blocks

**Input:** A sequence of three-address statements.

**Output:** A list of basic blocks with each three-address statement in exactly one block.

**Method:**

1. We first determine the set of **leaders**, for that we use the following rules:
  - I. The first statement is a leader.
  - II. Any statement that is the target of a conditional or unconditional goto is a leader.
  - III. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

# Example: Partition into basic blocks

begin

prod := 0;

i := 1;

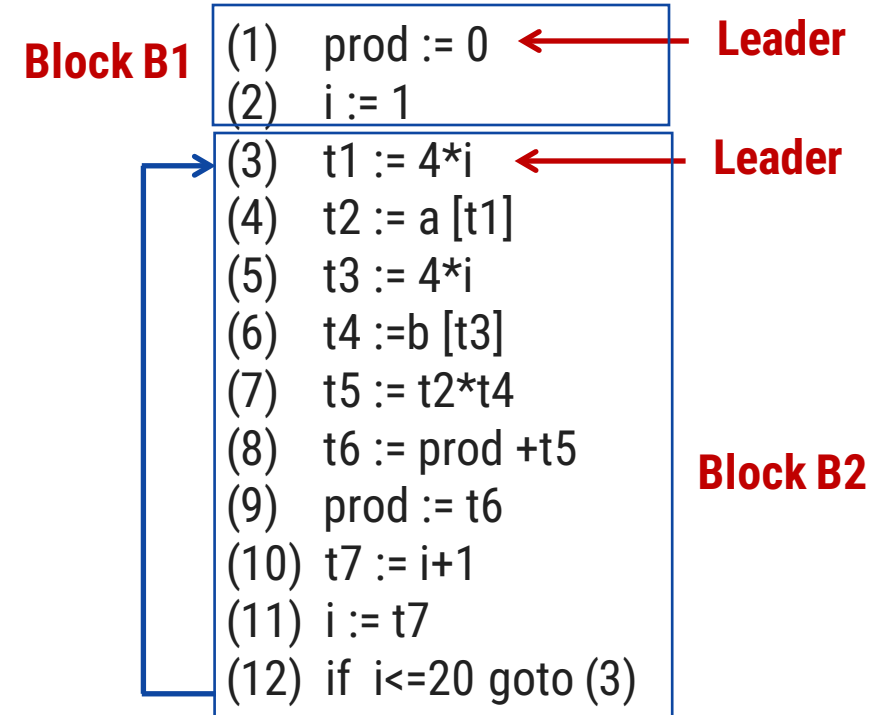
do

prod := prod + a[t1] \* b[t2];

i := i+1;

while i <= 20

end



**Three Address Code**



# Transformation on Basic Blocks

Optimization of Basic block



# Transformation on Basic Blocks

- ▶ A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- ▶ Many of these **transformations** are useful for **improving the quality of the code**.
- ▶ Types of transformations are:
  1. Structure preserving transformation
  2. Algebraic transformation

# Structure Preserving Transformations

- ▶ Structure-preserving transformations on basic blocks are:
  1. Common sub-expression elimination
  2. Dead-code elimination
  3. Renaming of temporary variables
  4. Interchange of two independent adjacent statements

# Common sub-expression elimination

- ▶ Consider the basic block,

a:= b+c

b:= a-d

c:= b+c

d:= a-d

- ▶ The second and fourth statements compute the same expression, hence this basic block may be transformed into the equivalent block:

a:= b+c

b:= a-d

c:= b+c

d:= b



# Dead-code elimination

- ▶ Suppose  $x$  is dead, that is, never subsequently used, at the point where the statement  $x := y + z$  appears in a basic block.
- ▶ Above statement may be safely removed without changing the value of the basic block.

# Renaming of temporary variables

- ▶ Suppose we have a statement  
 $t := b + c$ , where  $t$  is a temporary variable.
- ▶ If we change this statement to  
 $u := b + c$ , where  $u$  is a new temporary variable,
- ▶ Change all uses of this instance of  $t$  to  $u$ , then the value of the basic block is not changed.
- ▶ In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary.
- ▶ We call such a basic block a *normal-form* block.

# Interchange of two independent adjacent statements

- ▶ Suppose we have a block with the two adjacent statements,

$t1 := b + c$

$t2 := x + y$

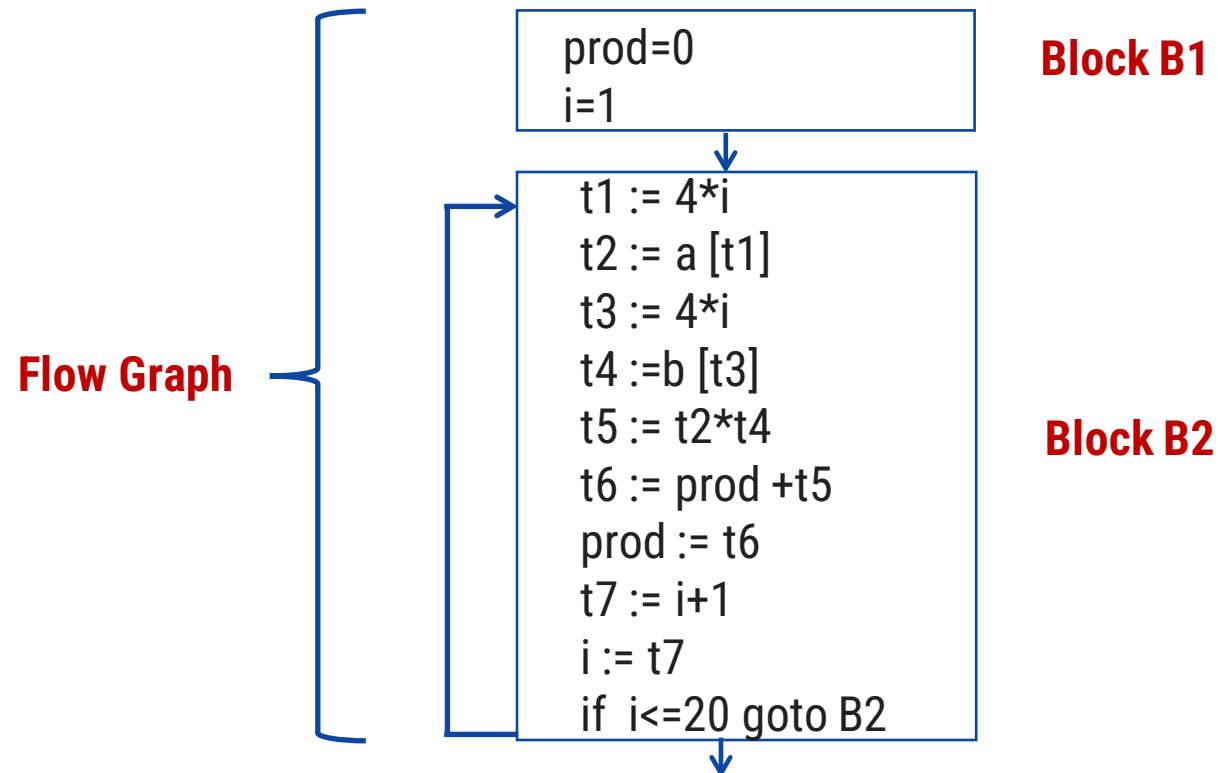
- ▶ Then we can interchange the two statements without affecting the value of the block if and only if neither  $x$  nor  $y$  is  $t1$  and neither  $b$  nor  $c$  is  $t2$ .
- ▶ A normal-form basic block permits all statement interchanges that are possible.

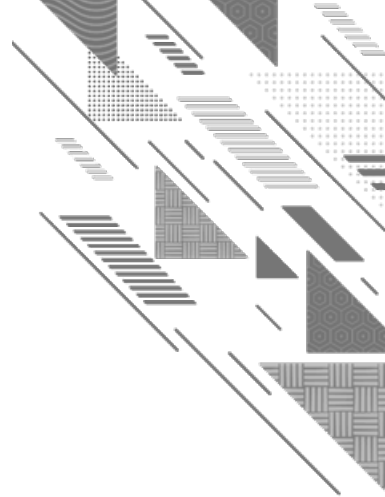
# Algebraic Transformation

- ▶ Countless algebraic transformation can be used to change the set of expressions computed by the basic block into an algebraically equivalent set.
- ▶ The useful ones are those that **simplify expressions or replace expensive operations by cheaper one**.
- ▶ Example:  **$x=x+0$  or  $x=x*1$**  can be eliminated.

# Flow Graph

- ▶ We can add flow-of-control information to the set of basic blocks making up a program by constructing a direct graph called a **flow graph**.
- ▶ Nodes in the flow graph represent computations, and the edges represent the flow of control.
- ▶ Example of flow graph for following three address code:





# A simple code generator

# A simple code generator

- ▶ The code generation strategy generates target code for a sequence of three address statement.
- ▶ It uses function **getReg()** to assign register to variable.
- ▶ The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
- ▶ **Address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- ▶ **Register descriptor** is used to keep track of what is currently in each register. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.

# A Code Generation Algorithm

- ▶ The algorithm takes a sequence of three-address statements as input. For each three address statement of the form  $x := y \text{ op } z$  perform the various actions. Assume  $L$  is the location where the output of operation  $y \text{ op } z$  is stored.
- 1. Invoke a function `getReg()` to find out the location  $L$  where the result of computation  $y \text{ op } z$  should be stored.
- 2. Determine the present location of 'y' by consulting address description for y if y is not present in location  $L$  then generate the instruction **MOV y' , L** to place a copy of y in L
- 3. Present location of z is determined using step 2 and the instruction is generated as **OP z' , L**
- 4. If  $L$  is a register then update it's descriptor that it contains value of x. update the address descriptor of x to indicate that it is in L.
- 5. 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  $x := y \text{ op } z$  those register will no longer contain y or z.



# Generating a code for assignment statement

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

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

$t := a - b$

$u := a - c$

$v := t + u$

$d := v + u$



# Optimization

Machine independent optimization



# Code Optimization

- ▶ Code Optimization is a program transformation technique which, tries to **improve the code** by **eliminating unnecessary code lines** and arranging the statements in such a sequence that speed up the execution without wasting the resources.

## Advantages

---

1. Faster execution
2. Better performance
3. Improves the efficiency

# Code Optimization techniques (Machine independent techniques)

## Techniques

---

1. Compile time evaluation
2. Common sub expressions elimination
3. Code Movement or Code Motion
4. Reduction in Strength
5. Dead code elimination

# Compile time evaluation

- ▶ Compile time evaluation means shifting of computations from run time to compile time.
- ▶ There are two methods used to obtain the compile time evaluation.

## Folding

- ▶ In the folding technique the computation of constant is done at compile time instead of run time.

Example :  $\text{length} = (22/7) * d$

- ▶ Here folding is implied by performing the computation of  $22/7$  at compile time.

## Constant propagation

- ▶ In this technique the value of variable is replaced and computation of an expression is done at compilation time.

Example :  $\text{pi} = 3.14; r = 5;$

$\text{Area} = \text{pi} * r * r;$

- ▶ Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of  $3.14 * 5 * 5$  is done during compilation.

# Common sub expressions elimination

- ▶ The common sub expression 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.
- ▶ Example:

t1 := 4 * i
t2 := a + 2
t3 := 4 * j
<del>t4 := - 4 * i</del>
t5 := n
t6 := b[t1] + t5

Before Optimization

# Code Movement or Code Motion

- ▶ Optimization can be obtained by **moving some amount of code outside the loop** and placing it just before entering in the loop.
- ▶ It won't have any difference if it executes inside or outside the loop.
- ▶ This method is also called **loop invariant computation**.
- ▶ **Example:**

```
While(i<=max-1)
{
    sum=sum+a[i];
}
```

**Before Optimization**



```
N=max-1;
While(i<=N)
{
    sum=sum+a[i];
}
```

**After Optimization**

# Reduction in Strength

- ▶ priority of certain operators is higher than others.
- ▶ For instance strength of  $*$  is higher than  $+$ .
- ▶ In this technique the higher strength operators can be replaced by lower strength operators.
- ▶ Example:

$$A = A * 2$$

**Before Optimization**

$$A = A + A$$

**After Optimization**

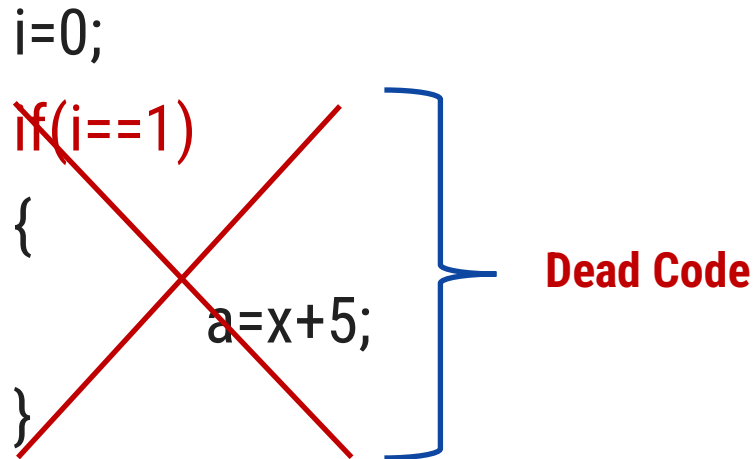


# Dead code elimination

- ▶ The variable is said to be **dead at a point in a program** if the value contained into it is never **been used**.
- ▶ The code containing such a variable supposed to be a dead code.
- ▶ **Example:**

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

**Dead Code**



- ▶ If statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.



# Optimization

Machine dependent optimization



# Machine dependent optimization

- ▶ Machine dependent optimization may vary from machine to machine.
- ▶ Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture.
- ▶ Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

## Techniques

1. Register allocation

Number of register may vary from machine to machine.

Used register may be of 32-bit register or 64 bit register.

2. Use of addressing modes

Addressing mode also vary from machine to machine.

3. Peephole optimization

# 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 replacing these instructions by shorter or faster sequence whenever possible.
- ▶ Peephole is a **small, moving window** on the target program.

# Redundant Loads & Stores

- ▶ 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.

# 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*

.....

*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*

.....

*L1: goto L2*



# Algebraic simplification

- ▶ Peephole optimization is an effective technique for algebraic simplification.
- ▶ The statements such as  $x = x + 0$  or  $x := x * 1$  can be eliminated by peephole optimization.

# 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.



# Machine idioms

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

# References

## Books:

### **1. Compilers Principles, Techniques and Tools, PEARSON Education (Second Edition)**

Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

### **2. Compiler Design, PEARSON (for Gujarat Technological University)**

Authors: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman



**Thank You**

