COMPILER DESIGN

Chapter 8

Code Generation

OUTLINE

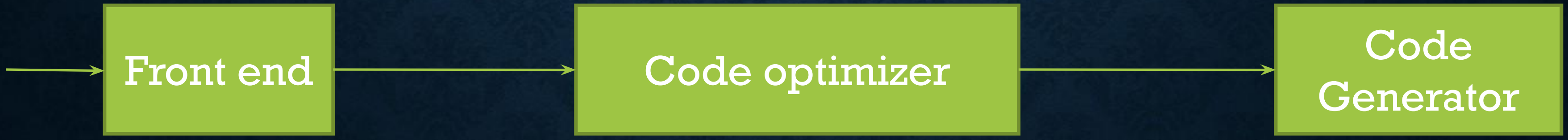
- Code Generation Issues
- Target language Issues
- Addresses in Target Code
- Basic Blocks and Flow Graphs
- Optimizations of Basic Blocks
- A Simple Code Generator
- Peephole optimization

INTRODUCTION

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- The requirements imposed on a code generator are severe
    - The target program must preserve the semantic meaning of the source program
    - Target program must be of high quality; that is, it must make effective use of the available resources of the target machine.

INTRODUCTION

- Compilers that need to produce efficient target programs, include an optimization phase prior to code generation.
- The optimizer maps the IR into IR from which more efficient code can be generated.
- In general, the code optimization and code-generation phases of a compiler, often referred to as the *back* end, may make multiple passes over the IR before generating the target program.



INTRODUCTION

- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering
- Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements.
- Register allocation and assignment involves deciding what values to keep in which registers.
- Instruction ordering involves deciding in what order to schedule the execution of instructions.

ISSUES IN THE DESIGN OF CODE GENERATOR

- The most important criterion for a code generator is that it produce correct code.
- Correctness takes on special significance because of the number of special cases that a 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.

Issues in the Design of Code Generator

Input to the code generator

- The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

Issues in the Design of Code Generator

- **Input to the code generator**

- The many choices for the IR include three-address representations such as
  quadruples,
  triples,
  indirect triples;

virtual machine representations such as
  bytecodes and
  stack-machine code;

linear representations such as
  postfix notation;

and graphical representations such as
  syntax trees and
  DAG's.

Issues in the Design of Code Generator

<u>The target program</u>

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.

The most common target-machine architectures are
- RISC (reduced instruction set computer),
- CISC (complex instruction set computer), and
- stack based.

Issues in the Design of Code Generator

The target program

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.

In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers.

Issues in the Design of Code Generator

The target program

- Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.

- Producing a relocatable machine-language program (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Issues in the Design of Code Generator

The target program

- Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module.

- If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

Issues in the Design of Code Generator

The target program

- 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.
- The price paid is the assembly step after code generation.

Issues in the Design of Code Generator

Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code.

Issues in the Design of Code Generator

Instruction Selection

The level of the IR

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

Issues in the Design of Code Generator

Instruction Selection

The nature of the instruction-set architecture

- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors.

- If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

- On some machines, for example, floating-point operations are done using separate registers.

Issues in the Design of Code Generator

<u>Instruction Selection</u>

The nature of the instruction-set architecture

- Instruction speeds and machine idioms are other important factors.
- If we do not care about the efficiency of the target program, instruction selection is straightforward.
- For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.

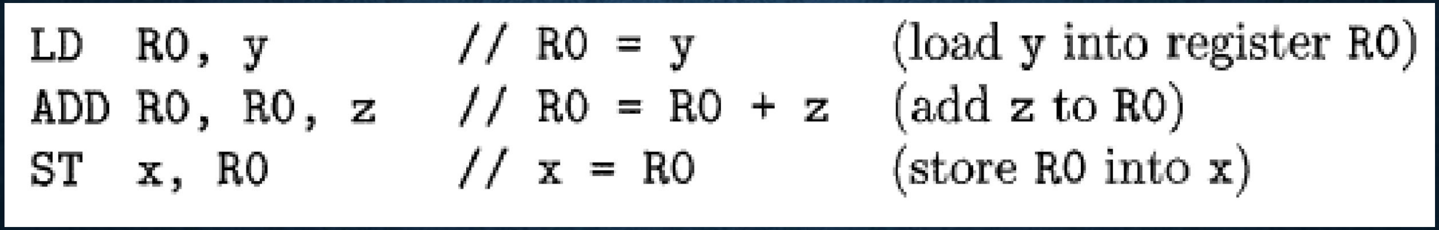
Issues in the Design of Code Generator

Instruction Selection

The nature of the instruction-set architecture

- For example, every three-address statement of the form

  x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence



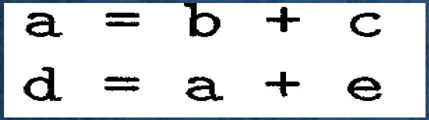
```
LD   R0, y        // R0 = y         (load y into register R0)
ADD  R0, R0, z    // R0 = R0 + z    (add z to R0)
ST   x, R0        // x = R0         (store R0 into x)
```

Issues in the Design of Code Generator

Instruction Selection

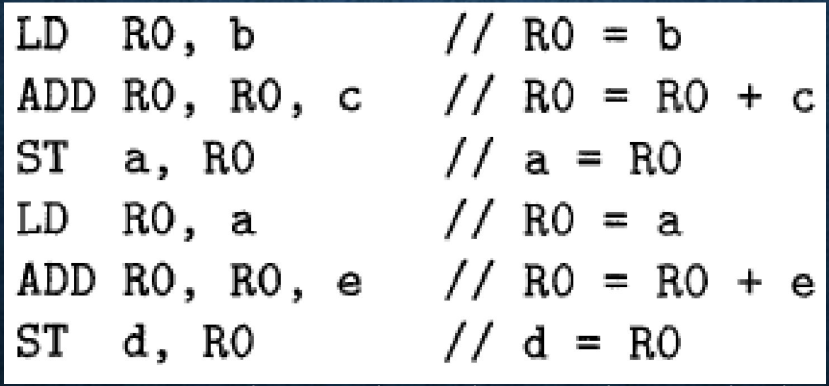
The nature of the instruction-set architecture

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.

Issues in the Design of Code Generator

Instruction Selection

The desired quality of the generated code.

- 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.

Issues in the Design of Code Generator

Instruction Selection

The desired quality of the generated code.

- 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 Code Generator

Register Allocation

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
- Values not held in registers need to reside in memory.
- Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

Issues in the Design of Code Generator

Register Allocation

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

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

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

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

4. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Issues in the Design of Code Generator

Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem.
- Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

THE TARGET LANGUAGE

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

A SIMPLE TARGET MACHINE MODEL

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- The underlying computer is a byte-addressable machine with n general-purpose registers, RO, R1, . . . , Rn - 1.
- A full-fledged assembly language would have scores of instructions.
- To avoid hiding the concepts in a myriad of details, we shall use a very limited set of instructions and assume that all operands are integers.
- Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction.

A SIMPLE TARGET MACHINE MODEL

We assume the following kinds of instructions are available:

- **Load operations:** The instruction **LD dst, addr** loads the value in location **addr** into location **dst**. This instruction denotes the assignment **dst = addr**. The most common form of this instruction is **LD r, x** which loads the value in location **x** into register **r**. An instruction of the form **LD rl, r2** is a register-to-register copy in which the contents of register **r2** are copied into register **rl**.
- **Store operations**: The instruction **ST x, r** stores the value in register **r** into the location **X**. This instruction denotes the assignment **x = r.**

A SIMPLE TARGET MACHINE MODEL

- **Computation operations** of the form ***OP*** **dst, srcl, src2**, where ***OP*** is a operator like ADD or SUB, and **dst, srcl,** and **src2** are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by ***OP*** to the values in locations **srcl** and **src2**, and place the result of this operation in location **dst**.

- For example, **SUB rl ,r2,r3** computes **rl = r2 – r3** Any value formerly stored in **rl** is lost, but if **rl** is **r2** or **r3**, the old value is read first. Unary operators that take only one operand do not have a **src2**.

A SIMPLE TARGET MACHINE MODEL

- **Unconditional jumps:** The instruction **BR** L causes control to branch to the machine instruction with label L. (BR stands for branch.)
- **Conditional jumps** of the form **Bcond r,** ***L,*** where **r** is a register, L is a label, and **cond** stands for any of the common tests on values in the register **r.**
- For example, **BLTZ r,** L causes a jump to label L if the value in register **r** is less than zero, and allows control to pass to the next machine instruction if not.

A SIMPLE TARGET MACHINE MODEL

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a variable name x referring to the memory location that is reserved for x (that is, the L-value of x).
- A location can also be an indexed address of the form **a(r)**, where **a** is a variable and **r** is a register. The memory location denoted by **a(r)** is computed by taking the L-value of **a** and adding to it the value in register **r.**
- For example, the instruction **LD R1, a(R2**) has the effect of setting **Rl = contents (a + contents (R2))**where **contents(x)** denotes the contents of the register or memory location represented by **x**. This addressing mode is useful for accessing arrays, where **a** is the base address of the array (that is, the address of the first element), and *r* holds the number of bytes past that address we wish to go to reach one of the elements of array **a**.

A SIMPLE TARGET MACHINE MODEL

- A memory location can be an integer indexed by a register. For example, **LD R1, 100(R2)** has the effect of setting **R1 = contents(100 + contents(R2))** that is, of loading into **R1** the value in the memory location obtained by adding **100** to the contents of register **R2**. This feature is useful for following pointers.
- We also allow two **indirect addressing modes**: ***r** means the memory location found in the location represented by the contents of register **r** and ***100(r)** means the memory location found in the location obtained by adding **100** to the contents of **r**.
- For example, **LD R1, * 100 (R2)** has the effect of setting **R1=contents(contents(l00 + contents(R2))),** that is, of loading into **R1** the value in the memory location stored in the memory location obtained by adding **100** to the contents of register **R2**.

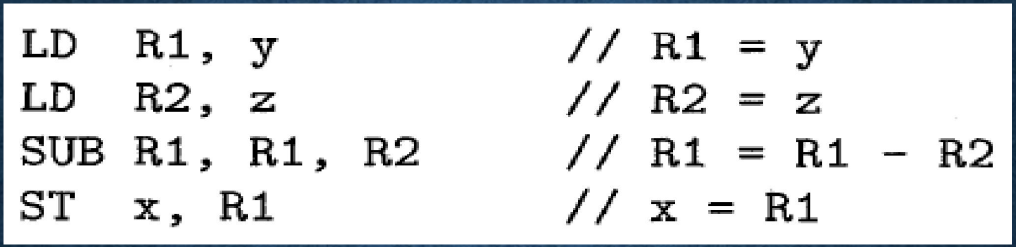
A SIMPLE TARGET MACHINE MODEL

Finally, we allow an **immediate constant addressing mode**. The constant is prefixed by **#**. The instruction **LD R1, #100** loads the integer **100** into register **R1**, and **ADD R1, R1, #I00** adds the integer **100** into register **R1**.

Comments at the end of instructions are preceded by //.

A SIMPLE TARGET MACHINE MODEL

The three-address statement x = y - *z* can be implemented by the machine instructions:



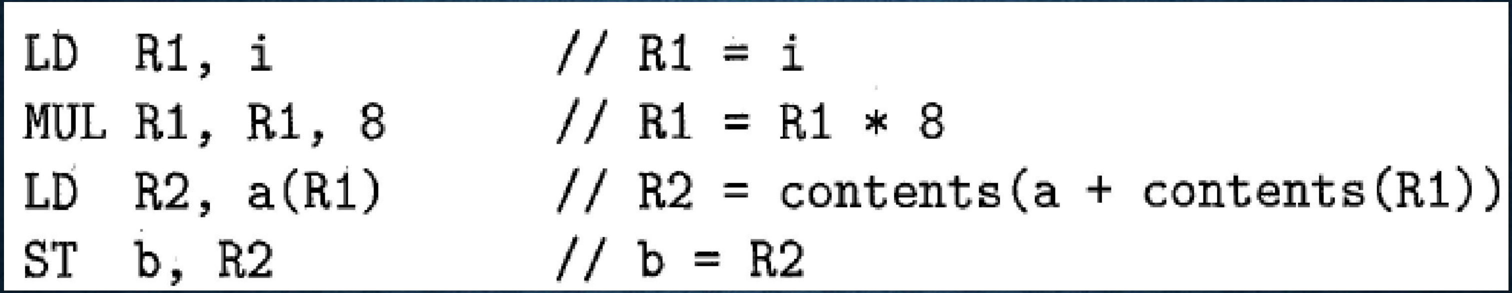
```
LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST  x, R1           // x = R1
```

- We can do better, perhaps. One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible.

- For example, y and/or z may have been computed in a register, and if so we can avoid the **LD** step(s). Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed

A SIMPLE TARGET MACHINE MODEL

Suppose **a** is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of **a** are indexed starting at 0. We may execute the three-address instruction **b** = **a** **[i]** by the machine instructions:



```
LD  R1, i           // R1 = i
MUL R1, R1, 8       // R1 = R1 * 8
LD  R2, a(R1)       // R2 = contents(a + contents(R1))
ST  b, R2           // b = R2
```

That is, the second step computes 8i, and the third step places in register **R2** the value in the ith element of **a** - the one found in the location that is 8i bytes past the base address of the array a.

A SIMPLE TARGET MACHINE MODEL

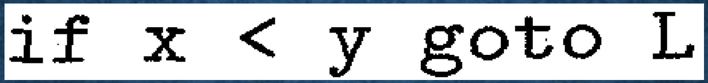
Similarly, the assignment into the array **a** represented by three-address instruction

a[j] = c is implemented by:

```
LD   R1, c              // R1 = c
LD   R2, j              // R2 = j
MUL  R2, R2, 8          // R2 = R2 * 8
ST   a(R2), R1          // contents(a +  contents(R2)) = R1
```

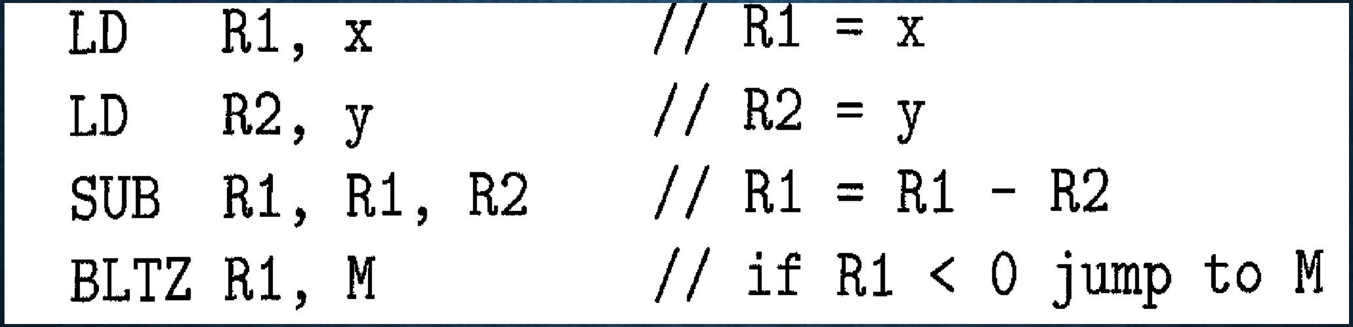
A SIMPLE TARGET MACHINE MODEL

consider a conditional-jump three-address instruction like



```
if x < y goto L
```

The machine-code equivalent would be something like:



```
LD   R1, x          // R1 = x
LD   R2, y          // R2 = y
SUB  R1, R1, R2     // R1 = R1 - R2
BLTZ R1, M          // if R1 < 0 jump to M
```

- Here, **M** is the label that represents the first machine instruction generated from the three-address instruction that has label L.
- As for any three-address instruction, we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored.

PROGRAM AND INSTRUCTION COSTS

- We often associate a cost with compiling and running a program.
- Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.
- Determining the actual cost of compiling and running a program is a complex problem.
- Finding an optimal target program for a given source program is an undecidable problem in general, and many of the sub problems involved are NP-hard.
- In code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

PROGRAM AND INSTRUCTION COSTS

- we shall assume each target-language instruction has an associated cost.
- For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.
- This cost corresponds to the length in words of the instruction.
- Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.

PROGRAM AND INSTRUCTION COSTS

Some examples:

- The instruction LD RO, R1 copies the contents of register R1 into register RO. This instruction has a cost of one because no additional memory words are required.

- The instruction LD RO, M loads the contents of memory location M into register RO. The cost is two since the address of memory location **M** is in the word following the instruction.

- The instruction LD R1, *100(R2) loads into register R1 the value given by contents(contents(l00 + contents(R2))). The cost is three because the constant 100 is stored in the word following the instruction.

BASIC BLOCKS AND FLOW GRAPHS

- graph representation of intermediate code that is helpful for discussing code generation.

- Code generation benefits from flow graph
  - We can do a better job of register allocation if we know how values are defined and used.
  - We can do a better job of instruction selection by looking at sequences of three-address statements.

BASIC BLOCKS AND FLOW GRAPHS

- The representation is constructed as follows:

1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the properties that

   (a) 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.

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

2. The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

BASIC BLOCKS AND FLOW GRAPHS

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.
- This idea is formalized in the following algorithm.

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

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.

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.

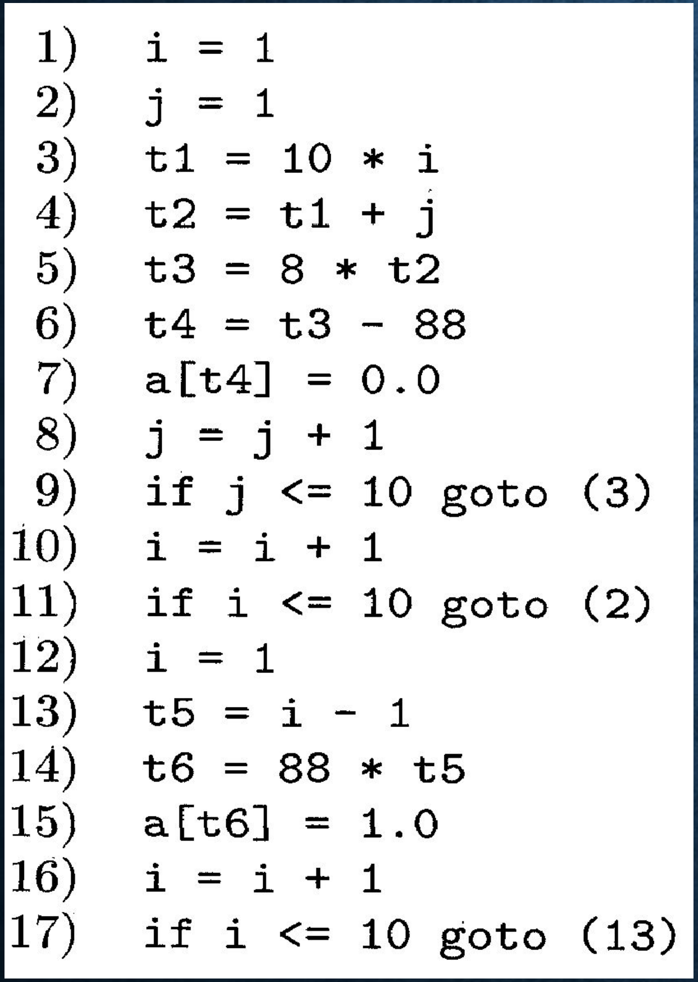
BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- 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.

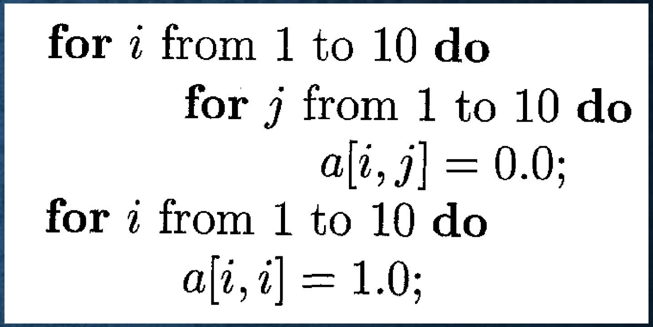
BASIC BLOCKS AND FLOW GRAPHS

<u>**Example**</u>



```
 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)
```

Figure 1: Intermediate code to set a 10 X 10 matrix to an identity matrix

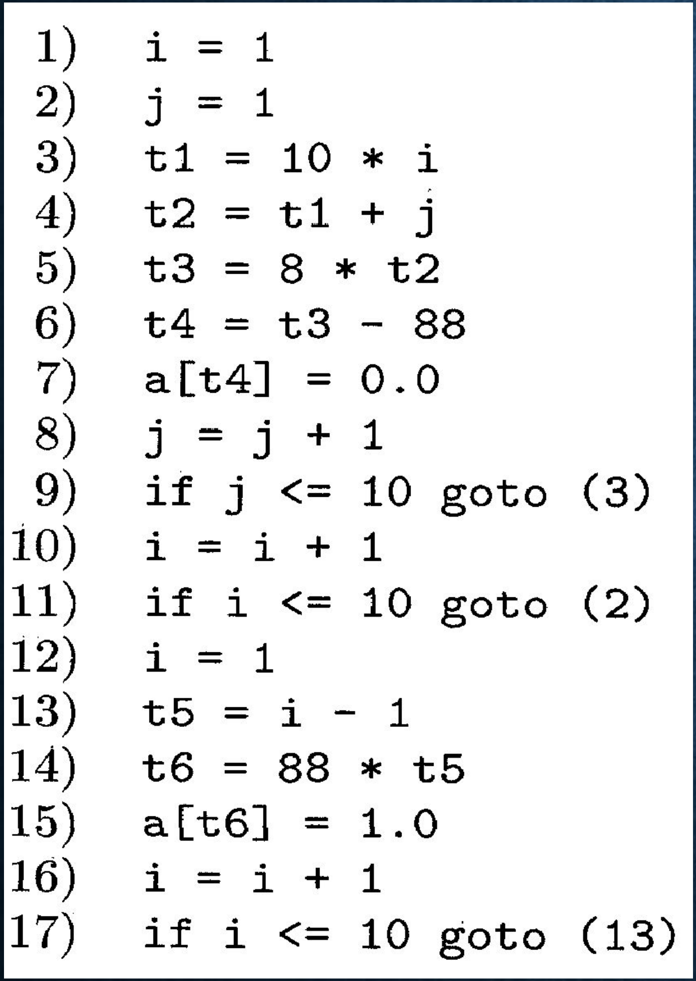


```
for $i$ from 1 to 10 do
    for $j$ from 1 to 10 do
        $a[i, j] = 0.0;$
for $i$ from 1 to 10 do
    $a[i, i] = 1.0;$
```

Figure 2: Source code for Fig. 1

BASIC BLOCKS AND FLOW GRAPHS

Example

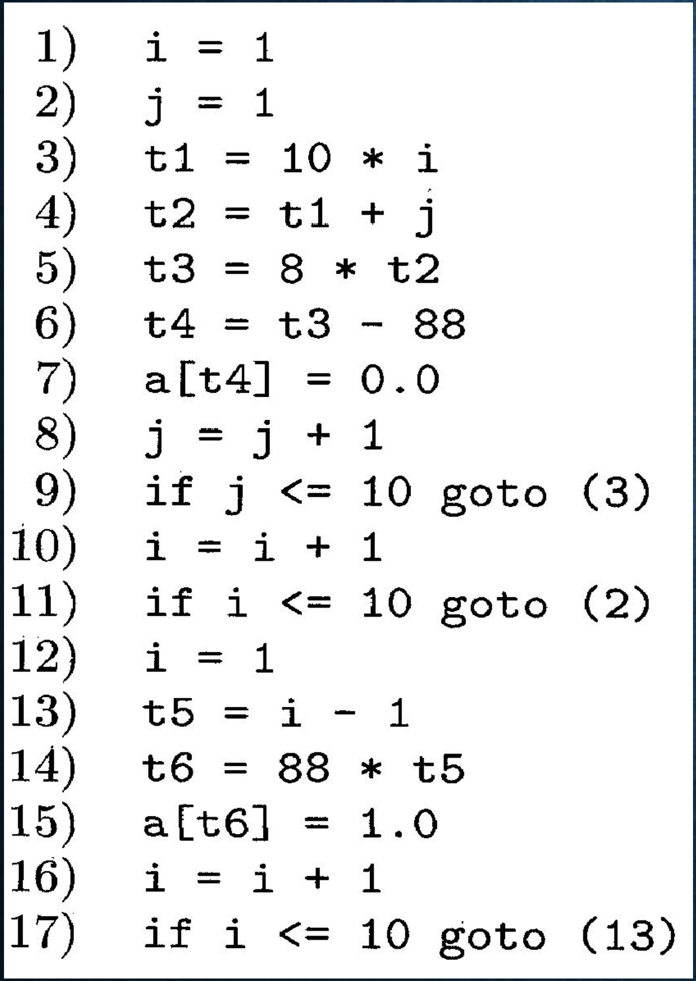


```
 1)   i = 1
 2)   j = 1
 3)   t1 = 10 * i
 4)   t2 = t1 + j
 5)   t3 = 8 * t2
 6)   t4 = t3 - 88
 7)   a[t4] = 0.0
 8)   j = j + 1
 9)   if j <= 10 goto (3)
10)   i = i + 1
11)   if i <= 10 goto (2)
12)   i = 1
13)   t5 = i - 1
14)   t6 = 88 * t5
15)   a[t6] = 1.0
16)   i = i + 1
17)   if i <= 10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

- First, instruction 1 is a leader by rule (1) of Algorithm . To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, **2,** and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

BASIC BLOCKS AND FLOW GRAPHS

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)
```

- We conclude that the leaders are instructions 1, **2,** 3, 10, 12, and 13.
- The basic block of each leader contains all the instructions from itself until just before the next leader.
- Thus, the basic block of 1 is just 1, for leader **2** the block is just **2.**
- Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17.

BASIC BLOCKS AND FLOW GRAPHS

Next-Use Information

- Knowing when the value of a variable will be used next is essential for generating good code.
- If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

- The *use* of a name in a three-address statement is defined as follows.
  - Suppose three-address statement i assigns a value to *x*.
  - If statement *j* has *x* as an operand, and control can flow from statement i to *j* along a path that has no intervening assignments to *x,* then we say **statement *j* *uses* the value of *x* computed at statement *i.***
- We further say that ***x* is *live* at statement *i.***

BASIC BLOCKS AND FLOW GRAPHS

Next-Use Information

- We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

- Our algorithm to determine liveness and next-use information makes a backward pass over each basic block. We store the information in the symbol table.

BASIC BLOCKS AND FLOW GRAPHS

Next-Use Information

Algorithm: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block *B* of three-address statements. We assume that the symbol table initially shows all nontemporary variables in *B* as being live on exit.

OUTPUT: At each statement *i:* x = y + *z* in *B,* we attach to i the liveness and next-use information of x, y, and *z.*

METHOD: We start at the last statement in *B* and scan backwards to the beginning of *B.* At each statement *i: x* = y + *z* in *B,* we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and *z* to "live" and the next uses of y and *z* to **i.**

BASIC BLOCKS AND FLOW GRAPHS

Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The nodes of the flow graph are the basic blocks.
- There is an edge from block B to block *C* if and only if it is possible for the first instruction in block *C* to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:
  - There is a conditional or unconditional jump from the end of B to the beginning of C.
  - C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.
- We say that B is a predecessor of C, and *C* is a successor of B.

BASIC BLOCKS AND FLOW GRAPHS

Flow Graphs

- Often we add two nodes, called the entry and exit, that do not correspond to executable intermediate instructions.
- There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code.
- There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.
- If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

BASIC BLOCKS AND FLOW GRAPHS

Example

- The set of basic blocks constructed in previous Example yields the flow graph of Fig. 3.
- The entry points to basic block B1, since B1 contains the first instruction of the program.
- The only successor of B1 is B2, because B1 does not end in an unconditional jump, and the leader of B2 immediately follows the end of B1.

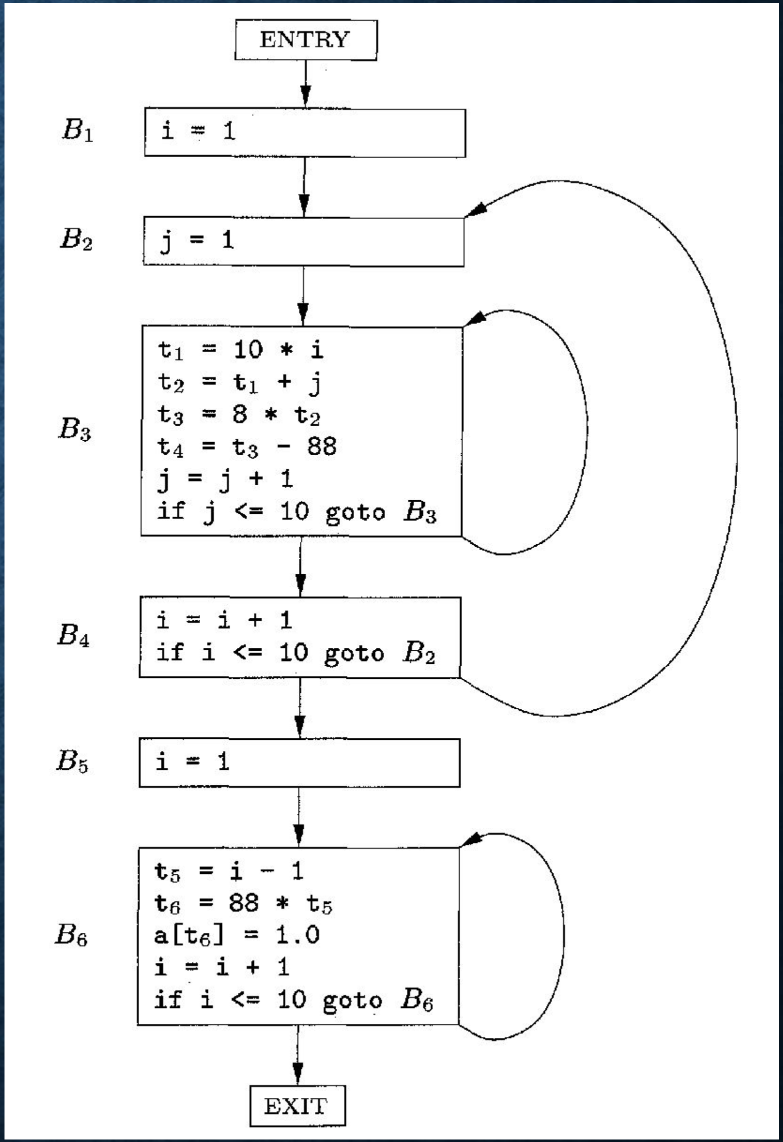


Figure 3: Flow graph from Fig. 2

BASIC BLOCKS AND FLOW GRAPHS

Example

- Block B3 has two successors. One is itself, because the leader of B3, instruction **3,** is the target of the conditional jump at the end of B3, instruction 9. The other successor is B4, because control can fall through the conditional jump at the end of B3 and next enter the leader of B4.
- Only B6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B6.

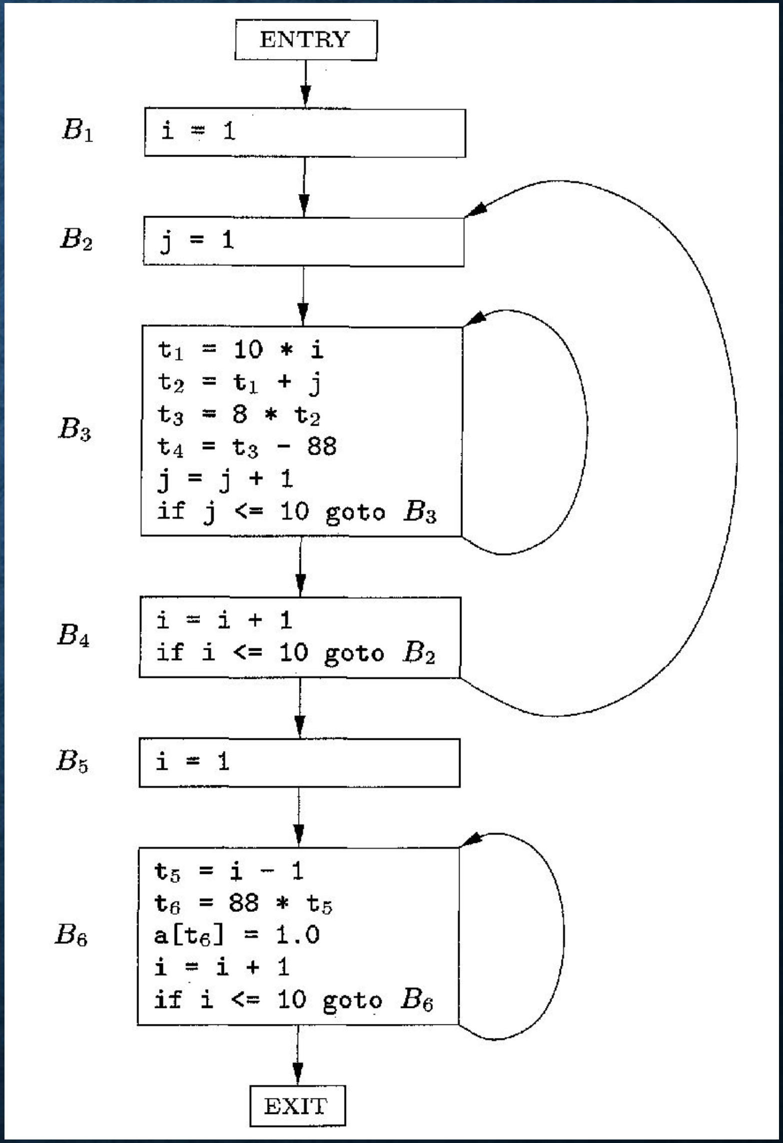


Figure 3: Flow graph from Fig. 2

OPTIMIZATION OF BASIC BLOCKS

We can often obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself.

It is a complex subject, with many different techniques to consider.

1. The DAG Representation of Basic Blocks
2. Finding Local Common Subexpressions
3. Dead Code Elimination
4. The Use of Algebraic Identities

OPTIMIZATION OF BASIC BLOCKS

1. <u>The DAG Representation of Basic Blocks</u>

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph). We construct a DAG for a basic block as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis.

OPTIMIZATION OF BASIC BLOCKS

1. **The DAG Representation of Basic Blocks**

The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.

a) We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.

b) We can eliminate *dead code,* that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.

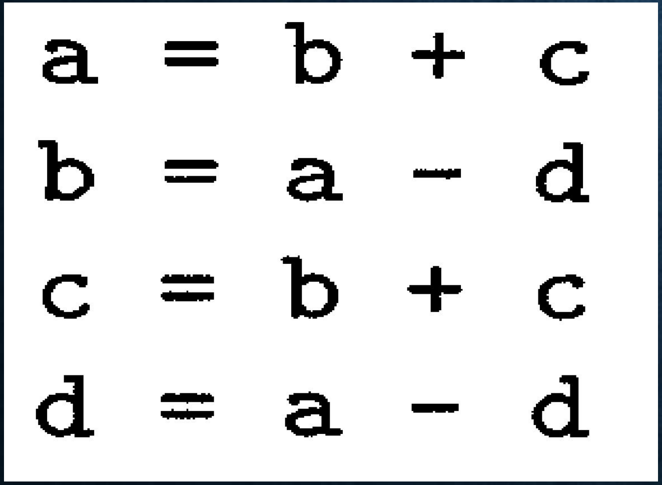
d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes t hereby simplify t he computation.

OPTIMIZATION OF BASIC BLOCKS

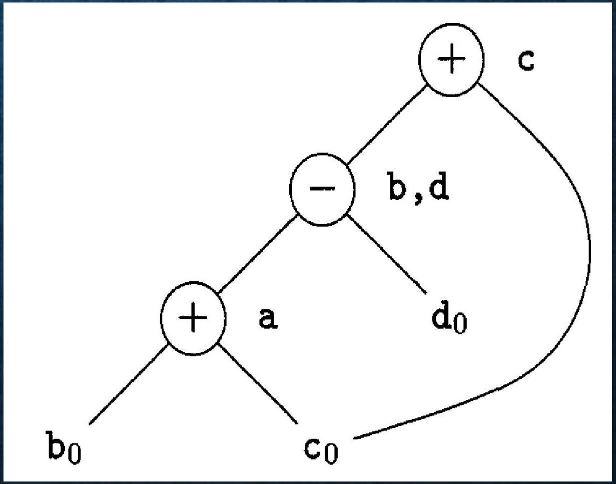
2. Finding Local Common Subexpressions

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

Example : A DAG for the block



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

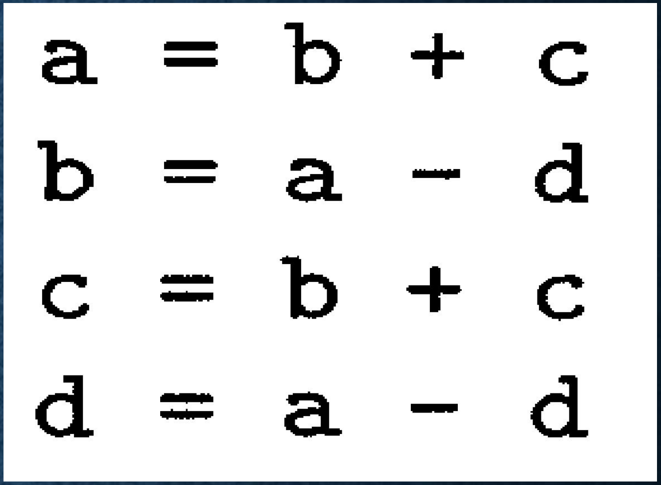


OPTIMIZATION OF BASIC BLOCKS

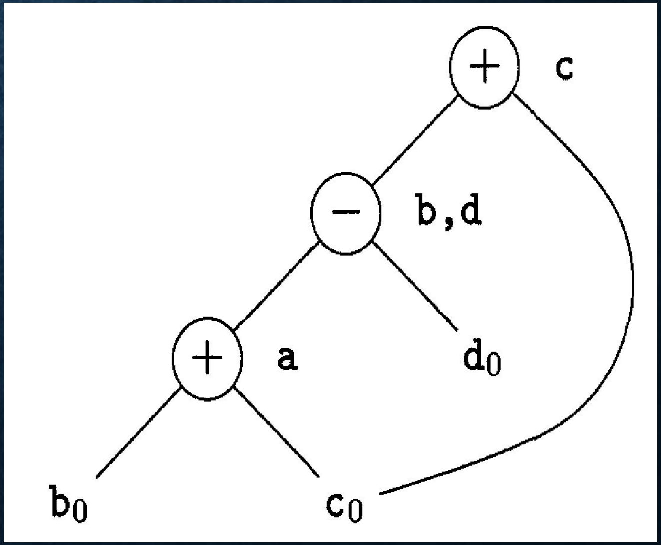
2. Finding Local Common Subexpressions

When we construct the node for the third statement c = b + c, we know that the use of b in b + c refers to the node labeled -, because that is the most recent definition of b. Thus, we do not confuse the values computed at statements one and three.

However, the node corresponding to the fourth statement d = a - d has the operator - and the nodes with attached variables a and d0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add d to the list of definitions for the node labeled -.



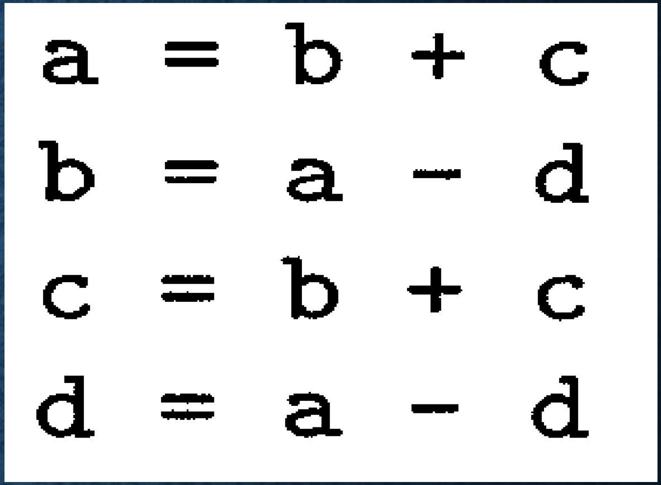
```
a = b + c
b = a - d
c = b + c
d = a - d
```



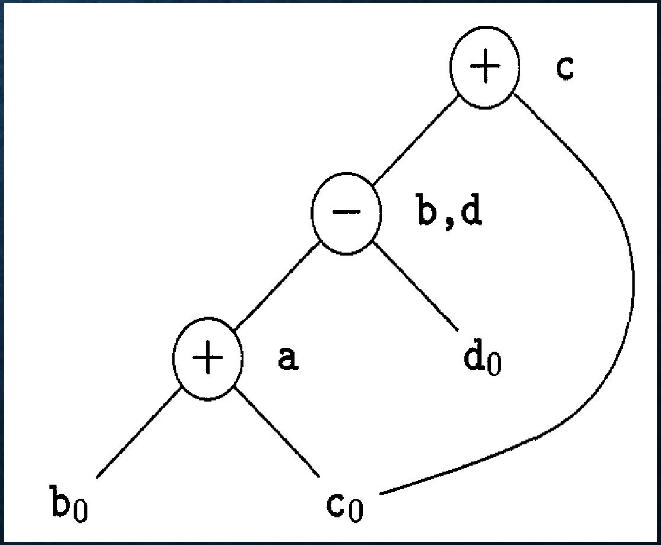
OPTIMIZATION OF BASIC BLOCKS

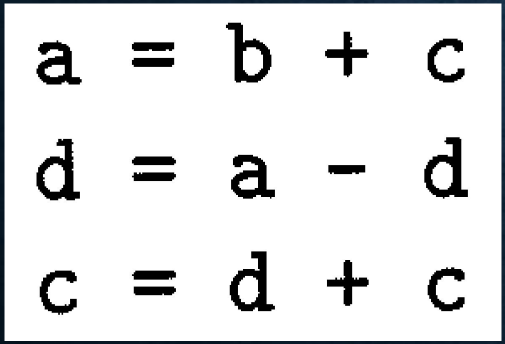
2. Finding Local Common Subexpressions

It might appear that, since there are only three nonleaf nodes in the DAG , the basic block in Example can be replaced by a block with only three statements. In fact, if b is not live on exit from the block, then we do not need to compute that variable, and can use d to receive the value represented by the node labeled -. The block then becomes



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





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

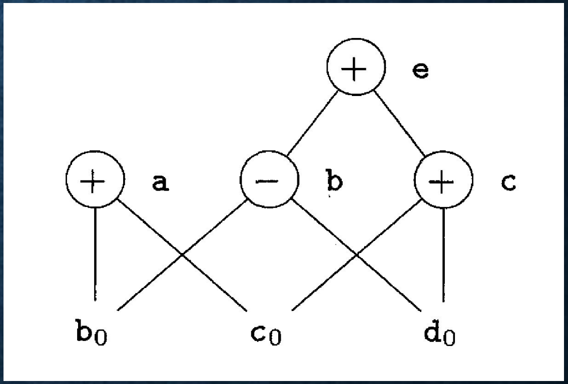
OPTIMIZATION OF BASIC BLOCKS

2. Finding Local Common Subexpressions

Example

When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence

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



is the same, namely b0 + c0. That is, even though b and c both change between the first and last statements, their sum remains the same, because b + c = *(b - d) + (c + d).*

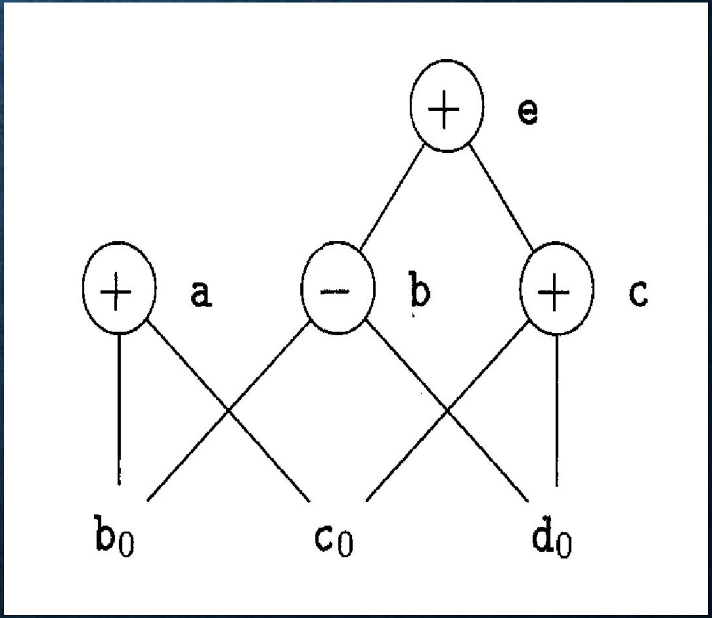
OPTIMIZATION OF BASIC BLOCKS

3. Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Example

If, in Fig., a and b are live but c and e are not, we can immediately remove the root labeled ***e*.** Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached.



OPTIMIZATION OF BASIC BLOCKS

4. The Use of Algebraic Identities

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

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

to eliminate computations from a basic block.

Another class of algebraic optimizations includes local reduction in strength, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	$=$	$x \times x$
$2 \times x$	$=$	$x + x$
$x/2$	$=$	$x \times 0.5$

OPTIMIZATION OF BASIC BLOCKS

4. The Use of Algebraic Identities

A third class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their value. Thus the expression *2* * 3.14 would be replaced by 6.28. Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity. For example, suppose the language reference manual specifies that * is commutative; that is, $x * y = y * x$. Before we create a new node labeled * with left child *M* and right child N, we always check whether such a node already exists. However, because * is commutative, we should then check for a node having operator *, left child N, and right child *M*.

A SIMPLE CODE GENERATOR

- Now, we shall consider an algorithm that generates code for a single basic block. It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.
- One of the primary issues during code generation is deciding how to use registers to best advantage.

A SIMPLE CODE GENERATOR

There are four principal uses of registers:

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries - places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.
- Registers are used to hold (global) values that are computed in one basic block and used in other blocks, for example, a loop index that is incremented going around the loop and is used several times within the loop.
- Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

These are competing needs, since the number of registers available is limited.

A SIMPLE CODE GENERATOR

- The algorithm in this section assumes that some set of registers is available to hold the values that are used within the block. Typically, this set of registers does not include all the registers of the machine, since some registers are reserved for global variables and managing the stack.
- We assume that the basic block has already been transformed into a preferred sequence of three-address instructions, by transformations such as combining common subexpressions.
- We further assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register.
- The machine instructions are of the form
    - LD reg, mem
    - ST mem, reg
    - OP reg, reg, reg

A SIMPLE CODE GENERATOR

1. Register and Address Descriptors

- Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers.
- After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store.
- In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so.
- We also need to know whether the memory location for a given variable currently has the proper value for that variable, since a new value for the variable may have been computed in a register and not yet stored.

A SIMPLE CODE GENERATOR

1. Register and Address Descriptors

The desired data structure has the following descriptors:

1. For each available register, a register descriptor keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.
2. For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

- An essential part of the algorithm is a function getReg(I), which selects registers for each memory location associated with the three-address instruction I.
- Function getReg has access to the register and address descriptors for all the variables of the basic block, and may also have access to certain useful data-flow information such as the variables that are live on exit from the block.
- While we do not know the total number of registers available for local data belonging to a basic block, we assume that there are enough registers so that, after freeing all available registers by storing their values in memory, there are enough registers to accomplish any three-address operation.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

- In a three-address instruction such as $x = y + z$, we shall treat + as a generic operator and ADD as the equivalent machine instruction.
- We do not, therefore, take advantage of commutativity of +. Thus, when we implement the operation, the value of y must be in the second register mentioned in the ADD instruction, never the third.
- A possible improvement to the algorithm is to generate code for both $x = y + z$ and $x = z + y$ whenever + is a commutative operator, and pick the better code sequence.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

Machine Instructions for Operations

For a three-address instruction such as x = y + z, do the following:

1. Use getReg(x = y + z) to select registers for x, y, and z. Call these R_X, R_Y, and R_Z.

2. If y is not in R_Y, (according to the register descriptor for R_Y), then issue an instruction ***LD*** $\boldsymbol{R_Y}$***, y'***, where y' is one of the memory locations for y (according to the address descriptor for y).

3. Similarly, if z is not in R_Z, issue and instruction ***LD*** $\boldsymbol{R_Z}$***, z',*** where z' is a location for *z*.

4. Issue the instruction ***ADD*** $\boldsymbol{R_X, R_Y, R_Z}$.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

Machine Instructions for Copy Statements

- There is an important special case: a three-address copy statement of the form x = y.
- We assume that getReg will always choose the same register for both x and y.
- If y is not already in that register R,, then generate the machine instruction LD R,, y.
- If y was already in R,, we do nothing.
- It is only necessary that we adjust the register description for R, so that it includes x as one of the values found there.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

Ending the Basic Block

- As we have described the algorithm, variables used by the block may wind up with their only location being a register.
- If the variable is a temporary used only within the block, that is fine; when the block ends, we can forget about the value of the temporary and assume its register is empty.
- However, if the variable is live on exit from the block, or if we don't know which variables are live on exit, then we need to assume that the value of the variable is needed later.
- In that case, for each variable x whose location descriptor does not say that its value is located in the memory location for x, we must generate the instruction ST x, R, where R is a register in which x's value exists at the end of the block.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

Managing Register and Address Descriptors

As the code-generation algorithm issues load, store, and other machine instructions, it needs to update the register and address descriptors.
The rules are as follows:

1. For the instruction LD R, x

    (a) Change the register descriptor for register R so it holds only x.

    (b) Change the address descriptor for x by adding register R as an additional location.

2. For the instruction ST x, R, change the address descriptor for x to include its own memory location.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

Managing Register and Address Descriptors

3. For an operation such as ADD R_X, $R_{Y,}$ R_Z implementing a three-address instruction

$x = y + x$

(a) Change the register descriptor for Rx so that it holds only *x*.

(b) Change the address descriptor for x so that its only location is fix.

(c) Remove Rx from the address descriptor of any variable other than x.

4. When we process a copy statement x = y, after generating the load for y into register R_Y, if needed, and after managing descriptors as for all load statements (per rule 1):

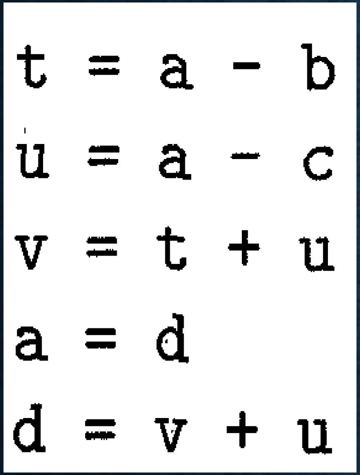
(a) Add x to the register descriptor for R_Y,.

(b) Change the address descriptor for x so that its only location is R_Y .

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

Example : Let us translate the basic block consisting of the three-address statements



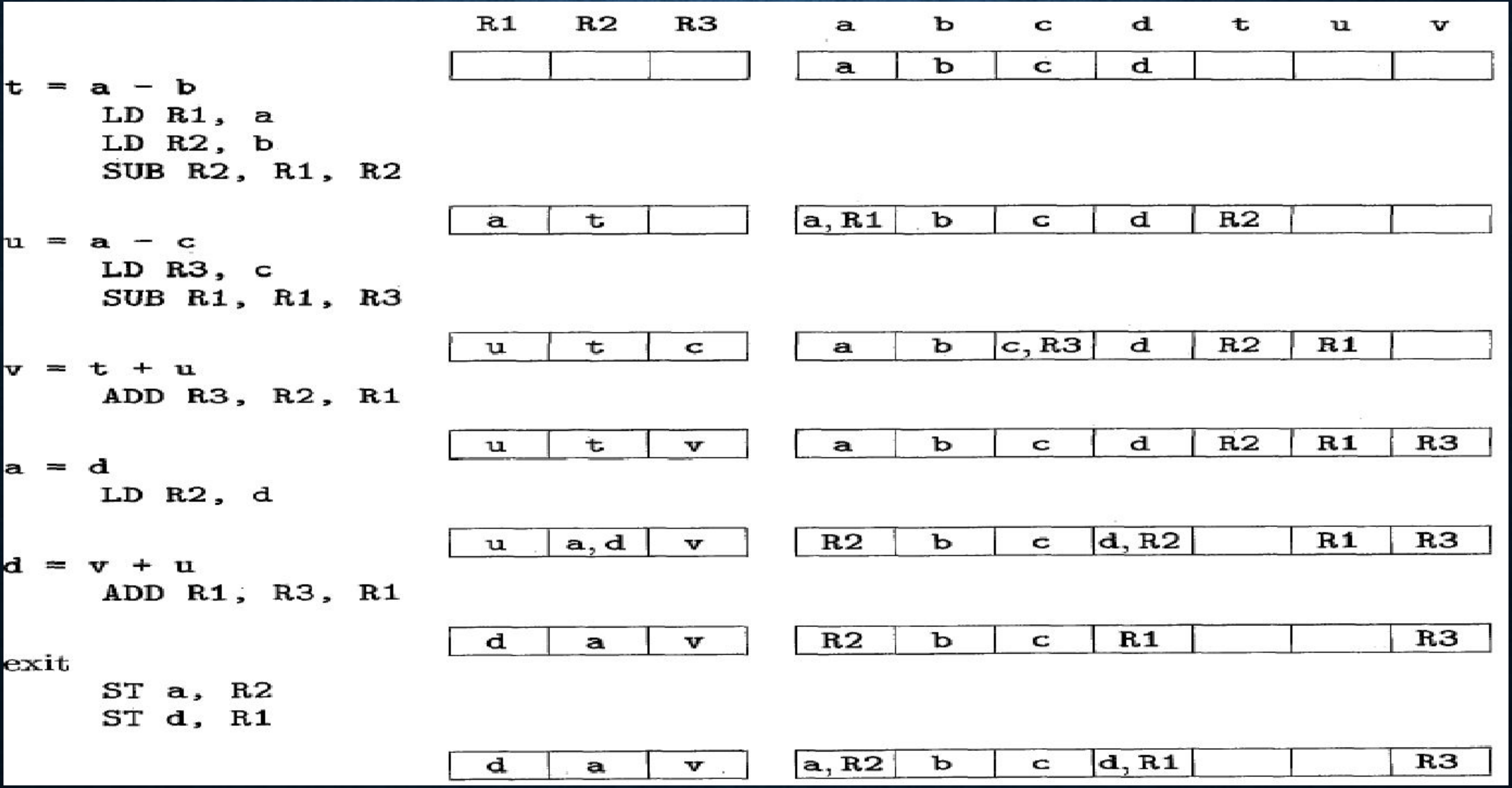
```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

- Here we assume that t, u, and v are temporaries, local to the block, while a, **b,** c, and d are variables that are live on exit from the block.
- Since we have not yet discussed how the function *getReg* might work, we shall simply assume that there are as many registers as we need, but that when a register's value is no longer needed (for example, it holds only a temporary, all of whose uses have been passed), then we reuse its register.

A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

A summary of all the machine-code instructions generated is in Fig. The figure also shows the register and address descriptors before and after the translation of each three-address instruction.

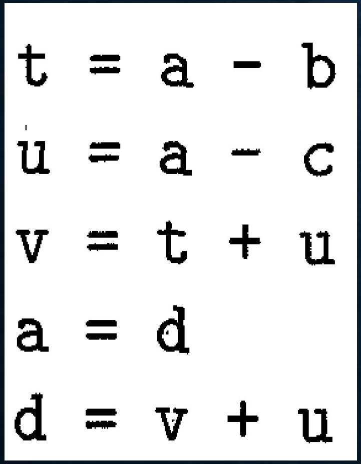


	R1	R2	R3	a	b	c	d	t	u	v
				a	b	c	d			
t = a - b LD R1, a LD R2, b SUB R2, R1, R2										
	a	t		a, R1	b	c	d	R2		
u = a - c LD R3, c SUB R1, R1, R3										
	u	t	c	a	b	c, R3	d	R2	R1	
v = t + u ADD R3, R2, R1										
	u	t	v	a	b	c	d	R2	R1	R3
a = d LD R2, d										
	u	a, d	v	R2	b	c	d, R2		R1	R3
d = v + u ADD R1, R3, R1										
	d	a	v	R2	b	c	R1			R3
exit ST a, R2 ST d, R1										
	d	a	v	a, R2	b	c	d, R1			R3

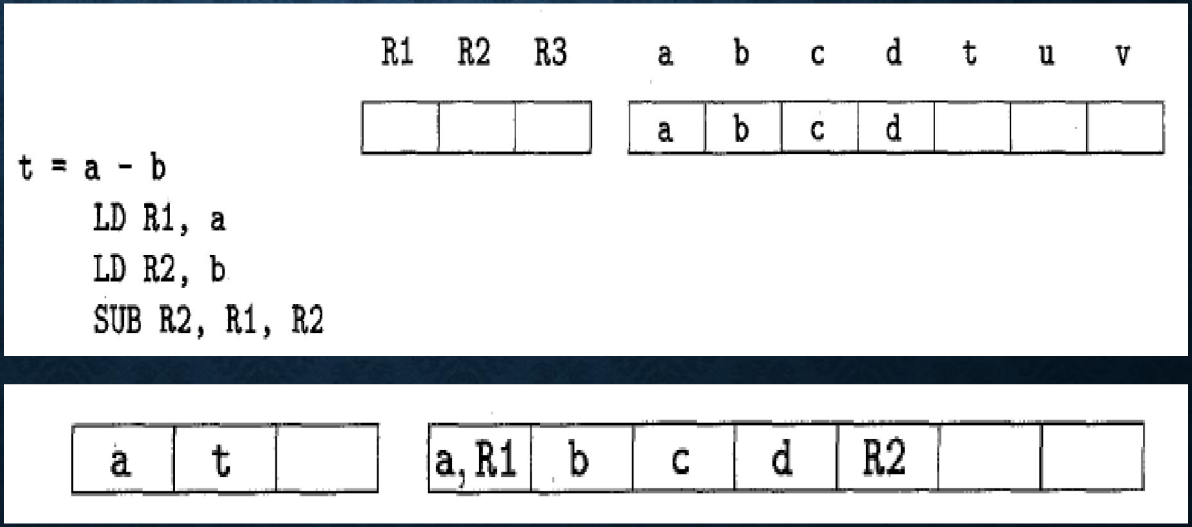
A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

- For the first three-address instruction, t = a - b we need to issue three instructions, since nothing is in a register initially. Thus, we see a and b loaded into registers R1 and R2, and the value t produced in register R2. Notice that we can use R2 for t because the value b previously in R2 is not needed within the block. Since b is presumably live on exit from the block, had it not been in its own memory location (as indicated by its address descriptor), we would have had to store R2 into b first. The decision to do so, had we needed R2, would be taken by getReg.



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

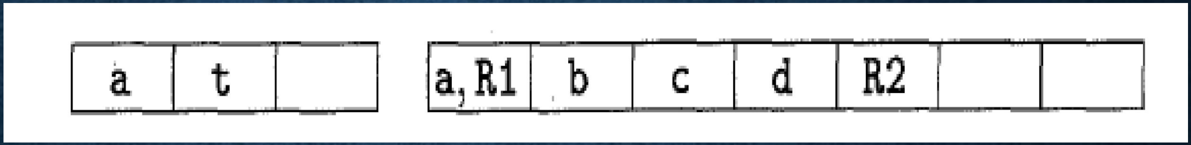


```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
```

A SIMPLE CODE GENERATOR

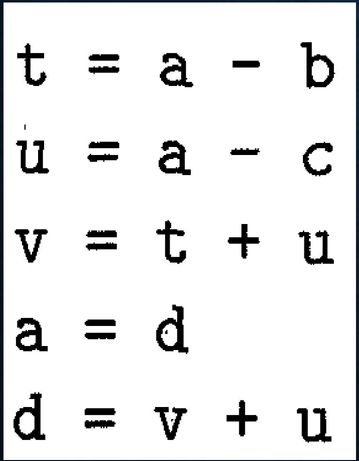
2. The Code-Generation Algorithm

- The second instruction, u = a - c, does not require a load of a, since it is already in register R1. Further, we can reuse R1 for the result, u, since the value of a, previously in that register, is no longer needed within the block, and its value is in its own memory location if a is needed outside the block. Note that we change the address descriptor for a to indicate that it is no longer in R1, but is in the memory location called a.

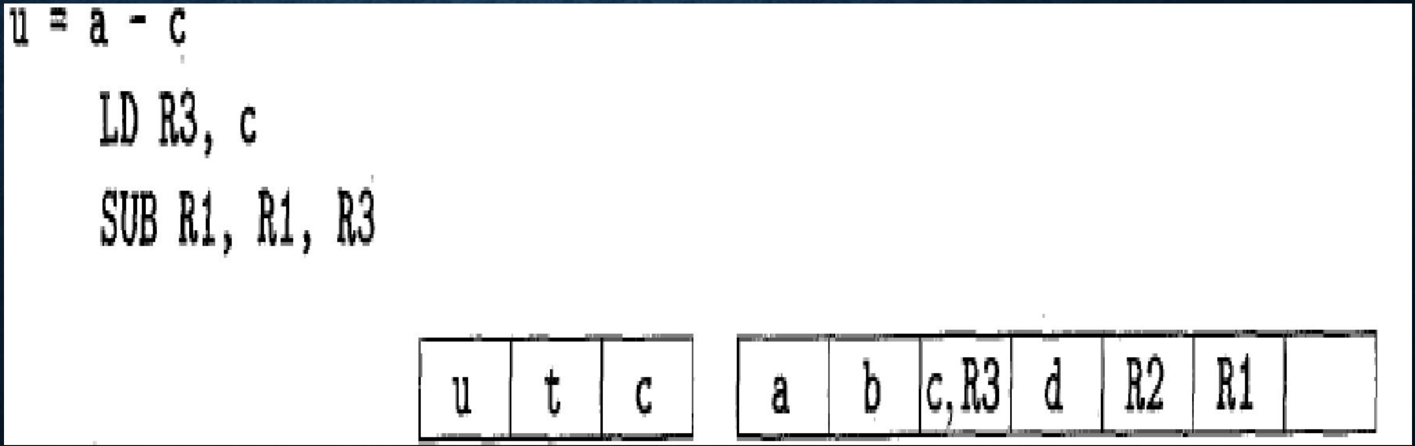


a	t	

a, R1	b	c	d	R2		



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



```
u = a - c
    LD R3, c
    SUB R1, R1, R3
```

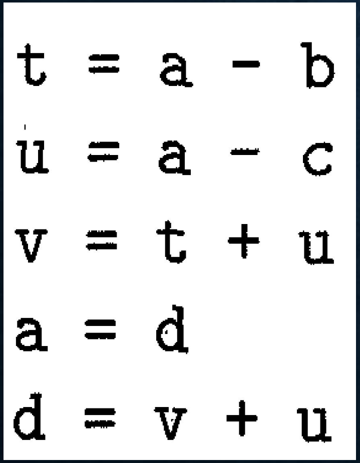
u	t	c

a	b	c, R3	d	R2	R1	

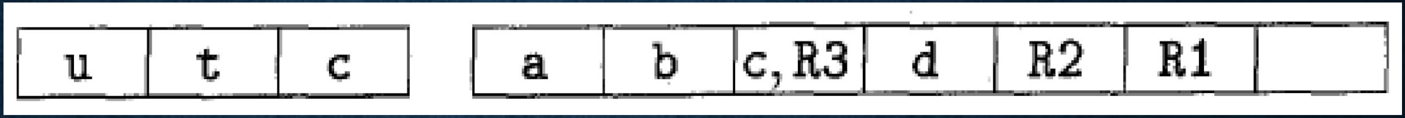
A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

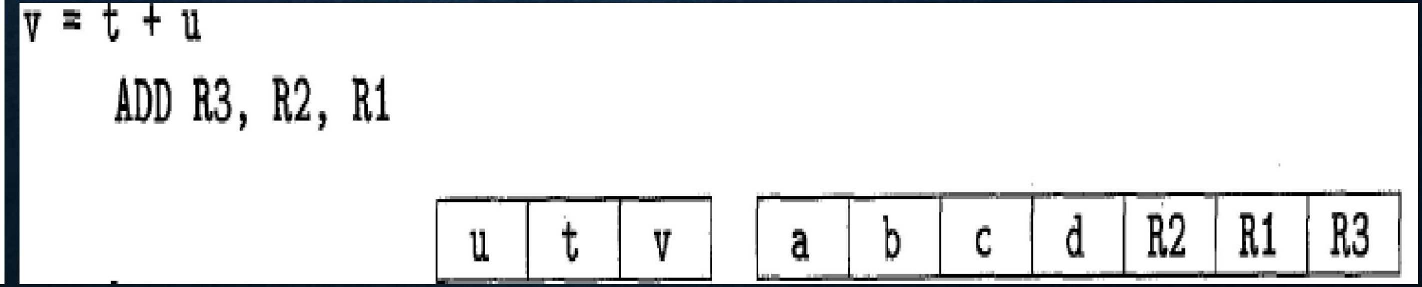
- The third instruction, v = t + u, requires only the addition. Further, we can use R3 for the result, v, since the value of c in that register is no longer needed within the block, and c has its value in its own memory location.



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



u	t	c	a	b	c, R3	d	R2	R1	



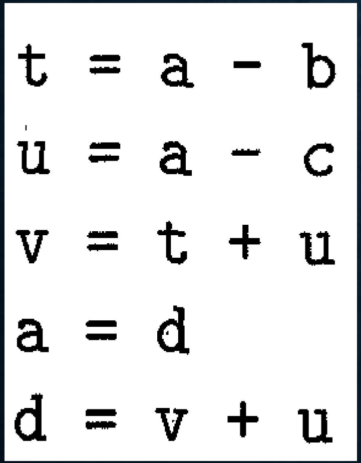
```
v = t + u
    ADD R3, R2, R1
```

u	t	v	a	b	c	d	R2	R1	R3

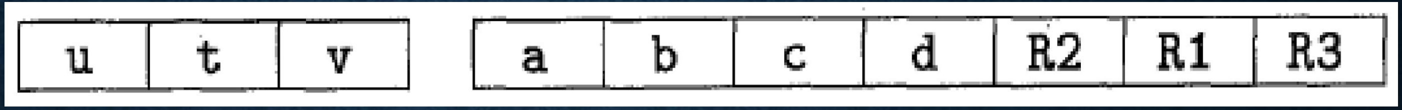
A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

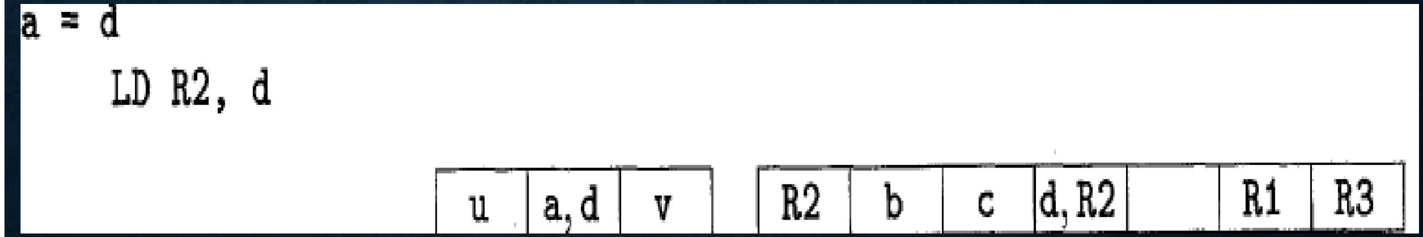
- The copy instruction, a = d, requires a load of d, since it is not in memory.
- We show register R2's descriptor holding both a and d.
- The addition of a to the register descriptor is the result of our processing the copy statement, and is not the result of any machine instruction.



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



u	t	v	a	b	c	d	R2	R1	R3



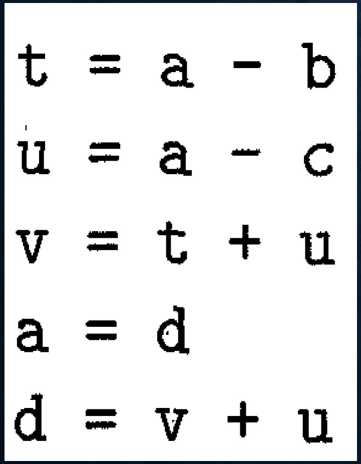
```
a = d
    LD R2, d
```

u	a, d	v	R2	b	c	d, R2		R1	R3

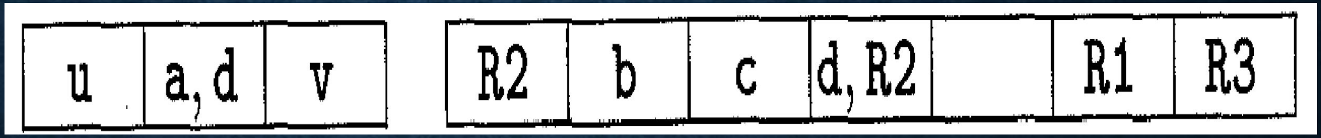
A SIMPLE CODE GENERATOR

2. The Code-Generation Algorithm

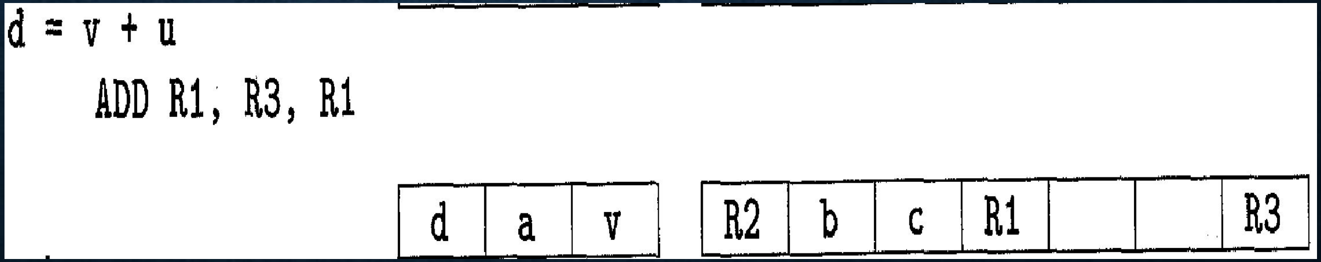
- The fifth instruction, d = v + u, uses two values that are in registers.
- Since u is a temporary whose value is no longer needed, we have chosen to reuse its register R1 for the new value of d.
- Notice that d is now in only R1, and is not in its own memory location.
- The same holds for a, which is in R2 and not in the memory location called a.
- As a result, we need the machine code for the basic block that stores the live-on-exit variables a and d into their memory locations. We show these as the last two instructions.



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



u	a, d	v	R2	b	c	d, R2		R1	R3

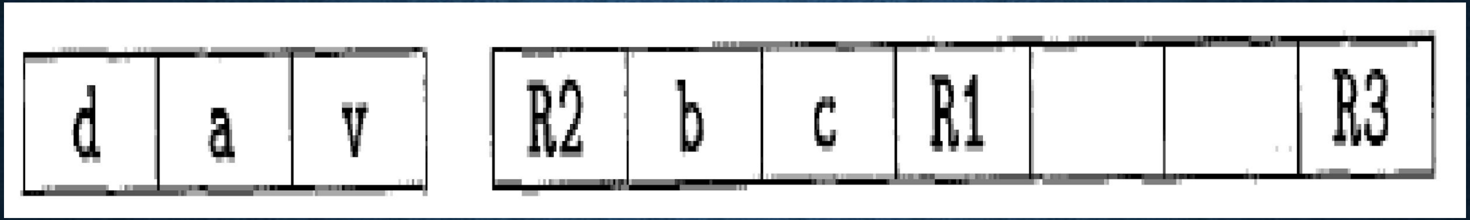


```
d = v + u
    ADD R1, R3, R1
```

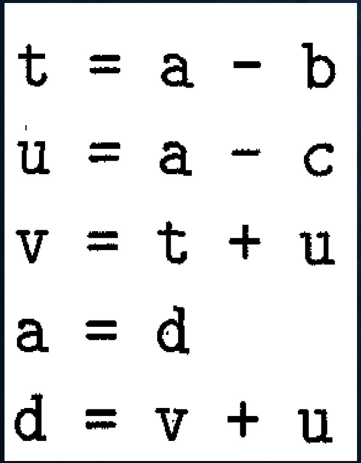
d	a	v	R2	b	c	R1			R3

A SIMPLE CODE GENERATOR

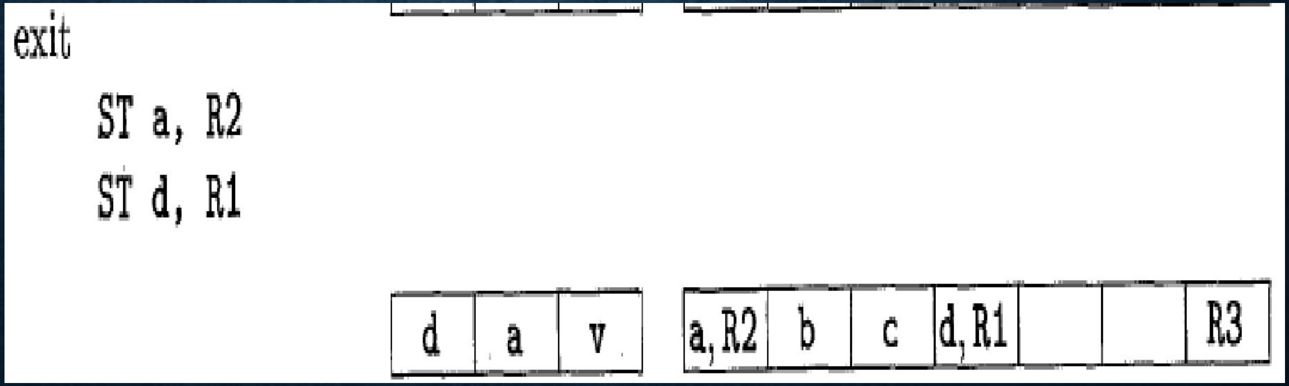
2. The Code-Generation Algorithm



d	a	v	R2	b	c	R1			R3



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



```
exit
    ST a, R2
    ST d, R1
```

d	a	v	a, R2	b	c	d, R1			R3

A SIMPLE CODE GENERATOR

3. Design of the Function *getReg*

- Lastly, let us consider how to implement getReg(I), for a three-address instruction I.

- We begin our examination with the case of an operation step, for which we again use x = y + *z* as the generic example.

- First, we must pick a register for y and a register for *z.* The issues are the same, so we shall concentrate on picking register ***Ry*** for y.

A SIMPLE CODE GENERATOR

3. Design of the Function *getReg*

Rules for picking register Ry for y

- If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.
- If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.
- The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

A SIMPLE CODE GENERATOR

<u>3. Design of the Function *getReg*</u>

<u>Possibilities for value of R</u>

- Let R be a candidate register, and suppose v is one of the variables that the register descriptor for *R* says is in R.
- We need to make sure that v's value either is not really needed, or that there is somewhere else we can go to get the value of R.
- The possibilities are:
  - If the address descriptor for v says that v is somewhere besides R, then we are OK.
  - If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.

A SIMPLE CODE GENERATOR

3. Design of the Function *getReg*

Possibilities for value of R

- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

A SIMPLE CODE GENERATOR

3. Design of the Function *getReg*

Selection of the register Rx

Now, consider the selection of the register *Rx*. The issues and options are almost as for y, so we shall only mention the differences.

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for *Rx*. This statement holds even if x is one of y and z, since our machine instructions allows two registers to be the same in one instruction.
2. If y is not used after instruction I and *R,* holds only *y* after being loaded, if necessary, then *R,* can also be used as *Rx*. A similar option holds regarding *x* and *R,*.

PEEPHOLE OPTIMIZATION

- While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program.
- A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

PEEPHOLE OPTIMIZATION

- The peephole is a small, sliding window on a program.
- The code in the peephole need not be contiguous, although some implementations do require this.
- It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- In general, repeated passes over the target code are necessary to get the maximum benefit.

PEEPHOLE OPTIMIZATION

we shall give the following examples of program transformations that are characteristic of peephole optimizations:

1. Redundant-instruction elimination
2. Flow-of-control optimizations
3. Algebraic simplifications
4. Use of machine idioms

PEEPHOLE OPTIMIZATION

1. Eliminating Redundant Loads and Stores

If we see the instruction sequence

LD a, RO

ST RO, 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 **RO.**
- Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction.
- Put another way, the two instructions have to be in the same basic block for this transformation to be safe.
- Redundant loads and stores of this nature would not be generated by the simple code generation algorithm of the previous section.

PEEPHOLE OPTIMIZATION

2. Eliminating Unreachable Code

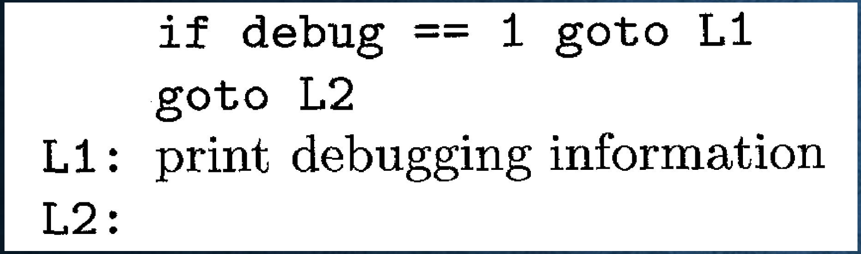
- Another opportunity for peephole optimization is the removal of unreachable instructions.
- An unlabeled instruction immediately following an unconditional jump may be removed.
- This operation can be repeated to eliminate a sequence of instructions.
- For example, for debugging purposes, a large program 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 L1
      goto L2
L1:   print debugging information
L2:
```

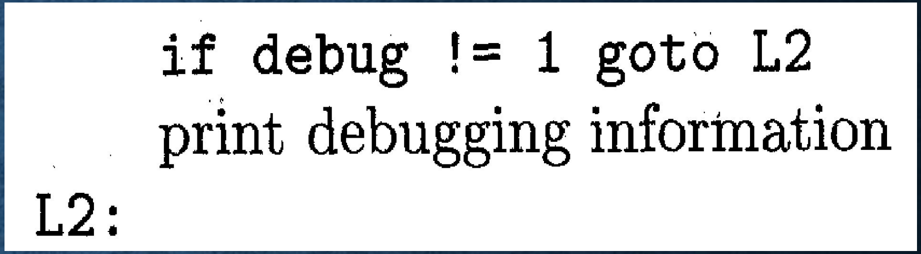
PEEPHOLE OPTIMIZATION

2. Eliminating Unreachable Code

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



```
     if debug == 1 goto L1
     goto L2
L1:  print debugging information
L2:
```

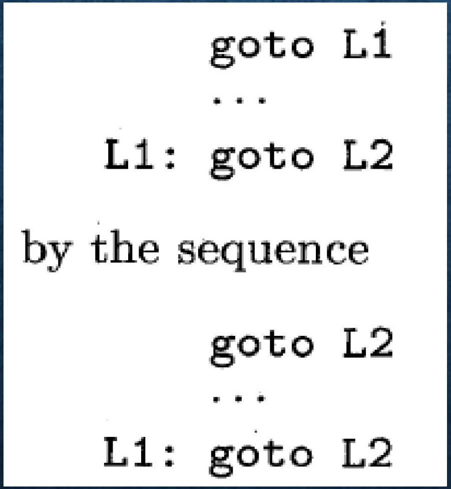


```
     if debug != 1 goto L2
     print debugging information
L2:
```

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.
- We can replace the sequence



```
        goto L1
        ...
    L1: goto L2
```

by the sequence

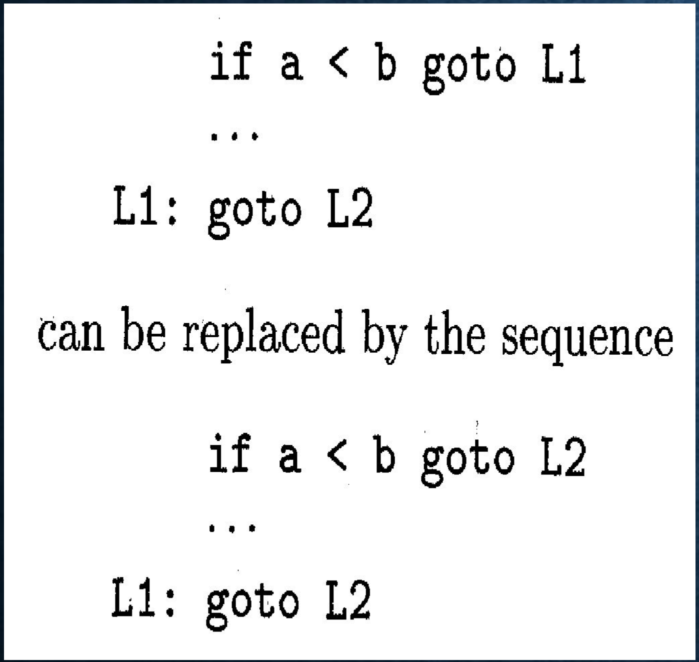
```
        goto L2
        ...
    L1: goto L2
```

- If there are now no jumps to **L1,** then it may be possible to eliminate the statement **L1:** goto **L2** provided it is preceded by an unconditional jump.

PEEPHOLE OPTIMIZATION

3. Flow-of-Control Optimizations

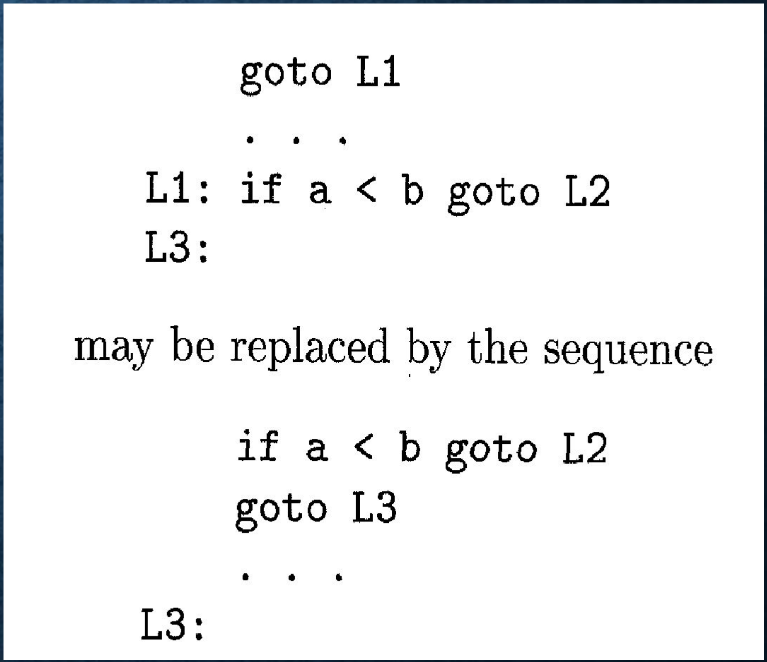
Similarly, the sequence



```
        if a < b goto L1
        ...
    L1: goto L2
```

can be replaced by the sequence

```
        if a < b goto L2
        ...
    L1: goto L2
```



```
        goto L1
        . . .
    L1: if a < b goto L2
    L3:
```

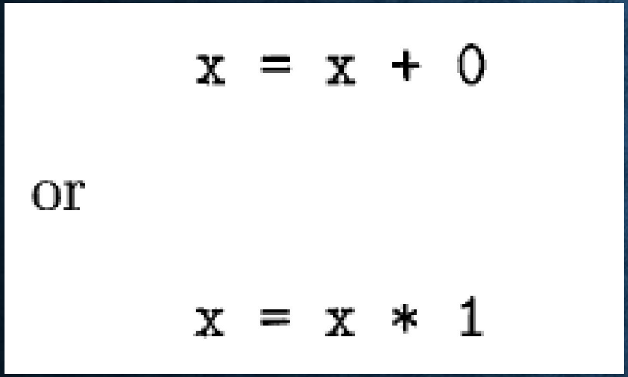
may be replaced by the sequence

```
        if a < b goto L2
        goto L3
        . . .
    L3:
```

PEEPHOLE OPTIMIZATION

4. Algebraic Simplification and Reduction in Strength

- These 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.

PEEPHOLE OPTIMIZATION

5. Use of Machine Idioms

- The target machine may have hardware'instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.
- These modes can also be used in code for statements like `x=x+l`.

BOOTSTRAPPING

How to write compiler for a language in the same language (first time!)?

- In computer science, **bootstrapping** is the technique for producing a self-compiling compiler - that is, compiler (or assembler) written in the source programming language that it intends to compile.

BOOTSTRAPPING

- Compilers are of two kinds: native and cross.
- Native compilers are written in the same language as the target language.
- e.g. SMM is a compiler for the language S that is in a language that runs on machine M and generates output code that runs on machine M.
- Cross compilers are written in different language as the target language.
- e.g. SNM is a compiler for the language S that is in a language that runs on machine N and generates output code that runs on machine M.

CROSS COMPILER

- A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.

CROSS COMPILER

Use

The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in several situations:

- ***Embedded computers*** where a device has extremely limited resources. For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food.

CROSS COMPILER

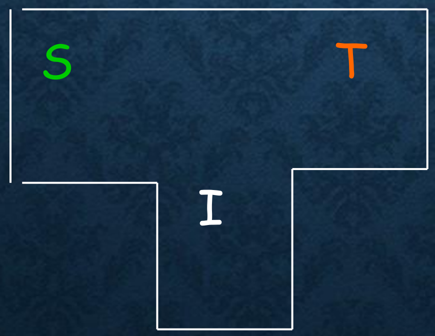
Use

- ***Compiling for multiple machines***.

For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.

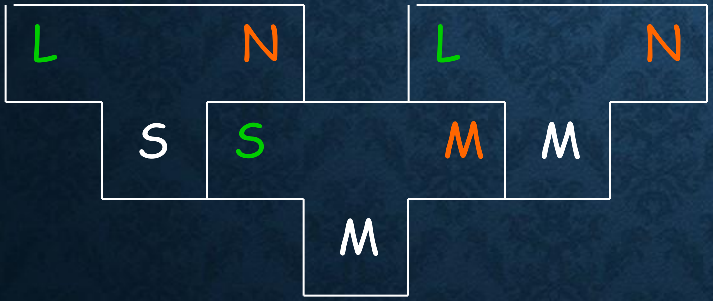
BOOTSTRAPPING

- A compiler can be characterized by three languages: the source language (S), the target language (T), and the implementation language (I)
- The three language S, I, and T can be quite different. Such a compiler is called cross-compiler
- This is represented by a T-diagram as:



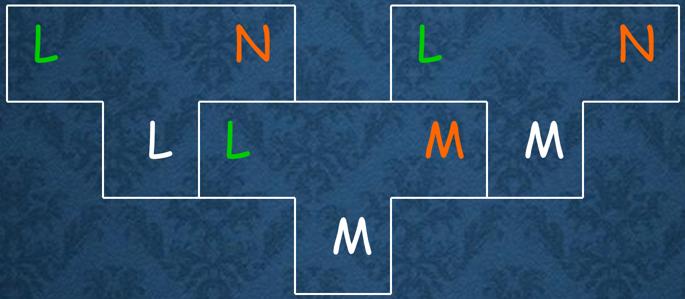
BOOTSTRAPPING

- Write a cross compiler for a language L in implementation language S to generate code for machine N
- Existing compiler for S runs on a different machine M and generates code for M
- When Compiler L_SN is run through S_MM we get compiler L_MN

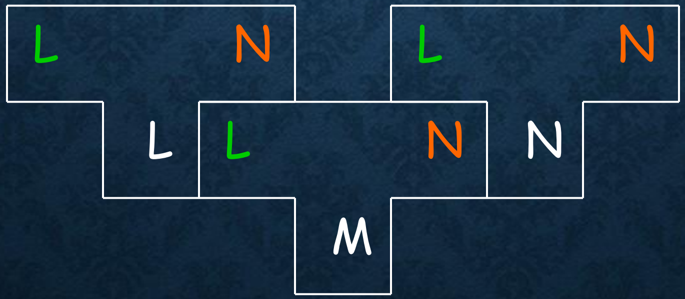


BOOTSTRAPPING A COMPILER

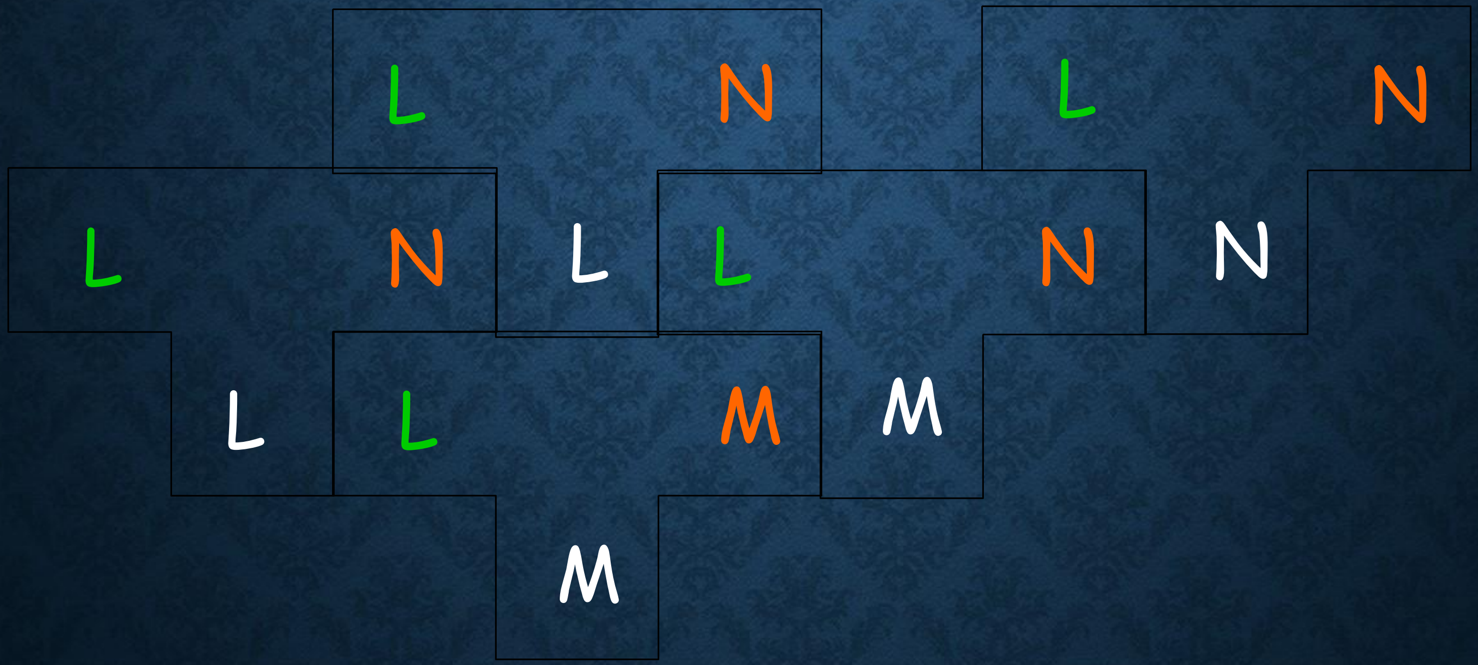
- Suppose L_LN is to be developed on a machine M where L_MM is available



- Compile L_LN second time using the generated compiler



BOOTSTRAPPING A COMPILER: THE COMPLETE PICTURE


L
N
L
N
L
N
L
L
N
N
L
L
M
M
M