Unit-6
Code Generation

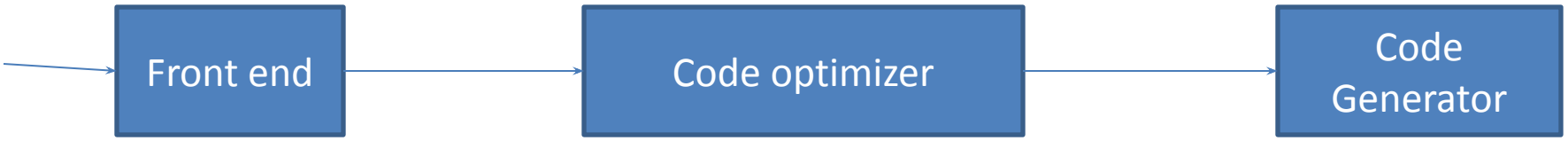
Dr.C.Kavitha
AP(SG)
Dept. of CSE
PSG College of Technology

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
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering



Issues in the Design of Code Generator

- The most important criterion is that it produces correct code

1. Input to the code generator

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :

  a. Linear representation such as postfix notation

  b. Three address representation such as quadruples

  c. Virtual machine representation such as stack machine code

  d. Graphical representations such as syntax trees and DAGs.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. The target program

  - Common target architectures are: RISC, CISC and Stack based machines
  - Assume a very simple RISC-like computer with addition of some CISC-like addressing modes are used
- The code generator has to be aware of the nature of the target language for which the code is to be transformed.
- The target machine can have either CISC or RISC processor architecture.
- The target code may be absolute code, re-locatable machine code or assembly language code.
- Absolute code can be executed immediately as the addresses are fixed.
- But in case of re-locatable it requires linker and loader to place the code in appropriate location and map (link) the required library functions.
- If it generates assembly level code then assemblers are needed to convert it into machine level code before execution.
- Re-locatable code provides great deal of flexibilities as the functions can be compiled separately before generation of object code.

3. Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine.

Complexity of mapping is determined by

- The level of the IR
- The nature of the instruction-set architecture-The instructions of target machine should be complete and uniform.
- The desired quality of the generated code-The quality of the generated code is determined by its speed and size.

```
x=y+z

LD    R0, y
ADD       R0, R0, z
ST    x, R0
```

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

LD    R0, b
ADD       R0, R0, c
ST    a, R0
LD    R0, a
ADD R0, R0, e
ST    d, R0
```

4. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- **Two sub problems**
  - **Register allocation**: selecting the set of variables that will reside in registers at each point in the program (Code generator decides what values to keep in the registers.)
  - **Resister assignment**: selecting specific register that a variable reside in (decides the registers to be used to keep these values)
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division
- Certain machine requires even-odd register pairs for some operands and results.

  For example , consider the division instruction of the form :D x, y

  where, x - dividend even register in even/odd register pair y-divisor
- even register holds the remainder
- odd register holds the quotient

```
t=a+b
t=t*c
T=t/d

L     R1, a
A     R1, b
M     R0, c
D     R0, d
ST    R1, t
```

5. Evaluation of order

- The code generator decides the order in which the instruction will be executed.
- It creates schedules for instructions to execute them.
- 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.

Target language

A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

Addressing Modes

- variable name: x
- indexed address: a(r) like LD R1, a(R2) means loads R1 the value given by contents(a+contents(R2))
- integer indexed by a register : like LD R1, 100(R2)
- Indirect addressing mode: LD R1, *100(R2)

Contents(contents(100+contents(R2)))

- immediate constant addressing mode: like LD R1, #100

b = **a [i]**

LD R1, i //**R1** = **i**

MUL R1, R1, 8 //**R1** = **Rl** * **8**

LD R2, a(R1) //**R2**=**contents(a+contents(R1))**

ST b, R2 //**b** = **R2**

a[j] = **c**

```
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
```

conditional-jump three-address instruction

If x<y goto L

```
LD R1, x          // R1 = x
LD R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M        // i f R1 < 0 jump t o M
```

costs associated with the addressing modes

- LD R0, R1 cost = 1
- LD R0, M cost = 2
- LD R1, *100(R2) cost = 3

Basic blocks and flow graphs

- Partition the intermediate code into 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.
- The basic blocks become the nodes of a flow graph

Rules for finding leaders

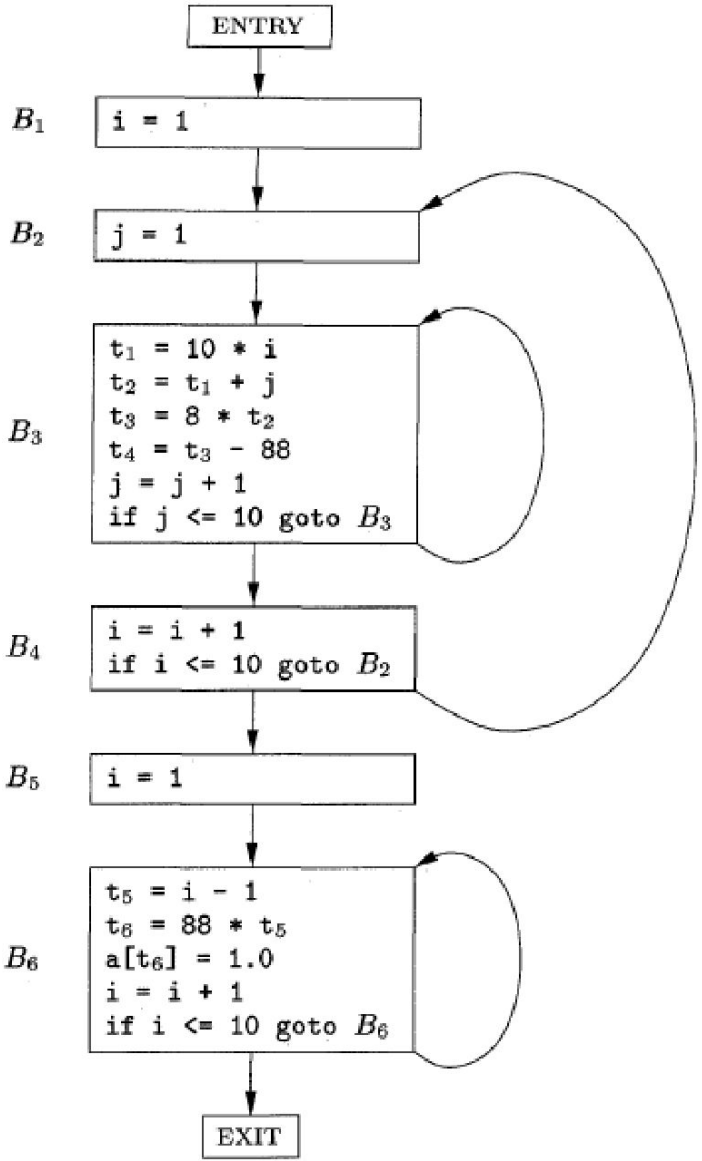
- The first three-address instruction in the intermediate code is a leader.
- Any instruction that is the target of a conditional or unconditional jump is a leader.
- Any instruction that immediately follows a conditional or unconditional jump is a leader.

Intermediate code to set a 10*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;
```

```
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)
```

Flow graph based on Basic Blocks



Liveness and 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.
- Algorithm:
  - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  - In the symbol table, set x to "not live" and "no next use.“
  - In the symbol table, set y and *z* to "live" and the next uses of y and *z* to i**.**

DAG representation of basic blocks

- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- 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.
- 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.
- Certain nodes are designated *output nodes.* These are the nodes whose variables are *live on exit* from the block.

Code improving transformations

- We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.
- We can eliminate *dead code,* that is, instructions that compute a value that is never used.
- 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.
- We can apply algebraic laws to reorder operands of three-address instructions, and sometimes t hereby simplify t he computation.

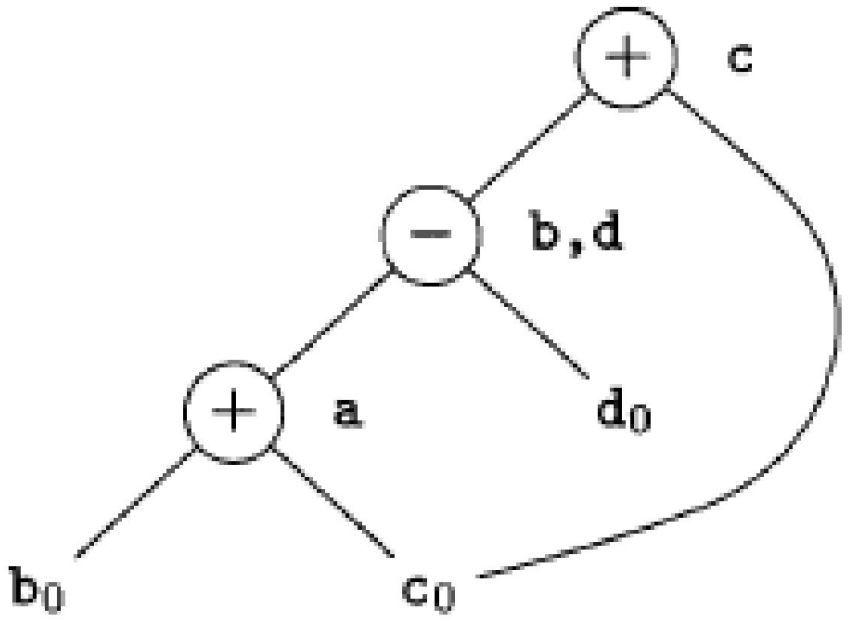
Optimization of Basic Blocks:

- Common sub-expression elimination
- Dead code elimination
- Use of Algebraic Identities

DAG for basic block

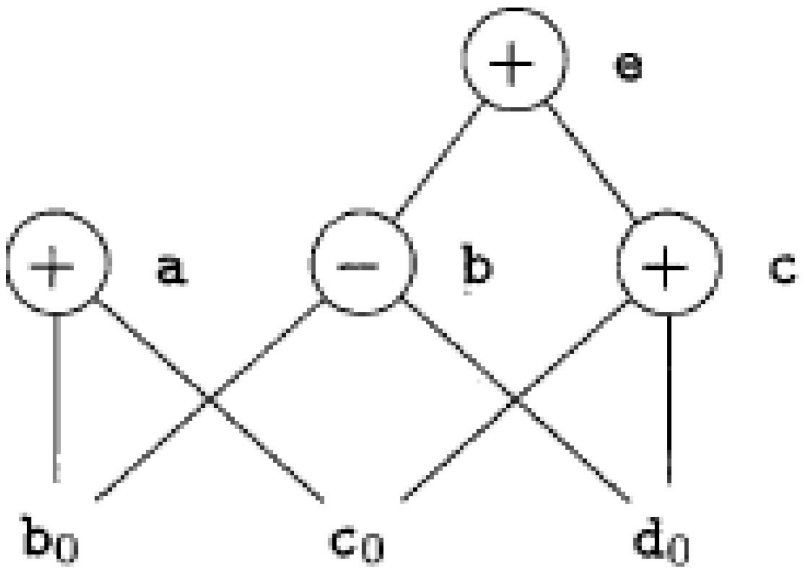
```
a = b + c
b = a - d
c = b + c
d = a - d
```

a : = b + c
b : = a - d
c : = b + c
d : = b



DAG for basic block

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

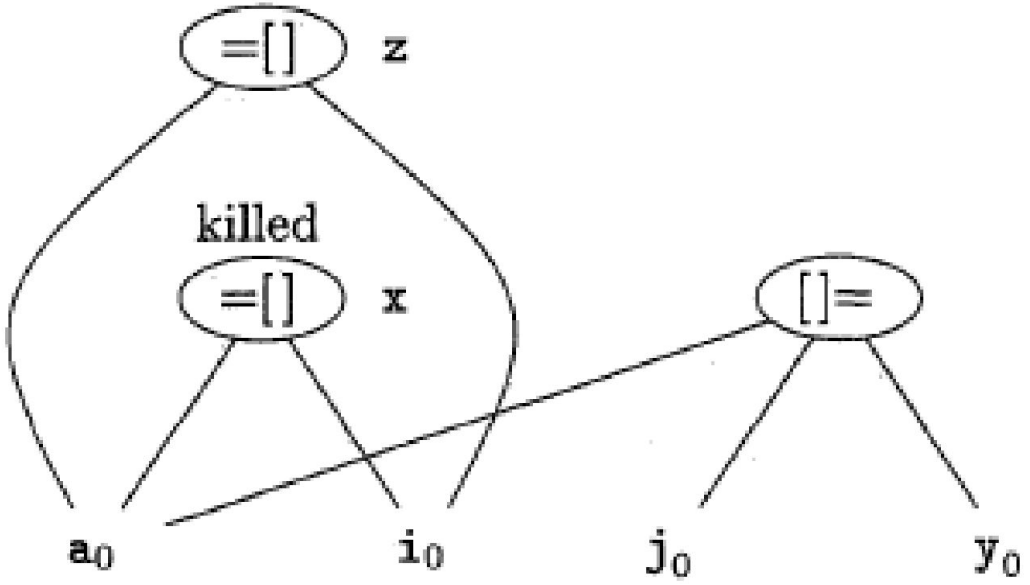


array accesses in a DAG

- An assignment from an array, like x = a [i], is represented by creating a node with operator =[] and two children representing the initial value of the array, a0 in this case, and the index i. Variable x becomes a label of this new node.
- An assignment to an array, like a [j] = y, is represented by a new node with operator []= and three children representing a0, j and y. There is no variable labeling this node. What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on a0**. A** node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

DAG for a sequence of array assignments

```
x = a[i]
a[j] = y
z = a[i]
```



Dead-code elimination

- It is possible that a program contains a large amount of dead code.
- This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
- Suppose the statement x:= y + z appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

Use of Algebraic Identities

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x *1 can be eliminated from a basic block without changing the set of expression.
- Sometimes the unexpected common sub expression is generated by the relational operators like <=, >=, <, >, +, = etc.
- Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments

  a :=b+c

  e :=c+d+b

  the following intermediate code may be generated:

  a :=b+c

  t :=c+d

  e :=t+b

Rules for reconstructing the basic block from a DAG

- The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.
- Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.
- Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.
- Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.
- Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

A simple code Generator

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 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.

Descriptors for data structure

- 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.
- 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 code-generation algorithm:

Machine Instructions for Operations

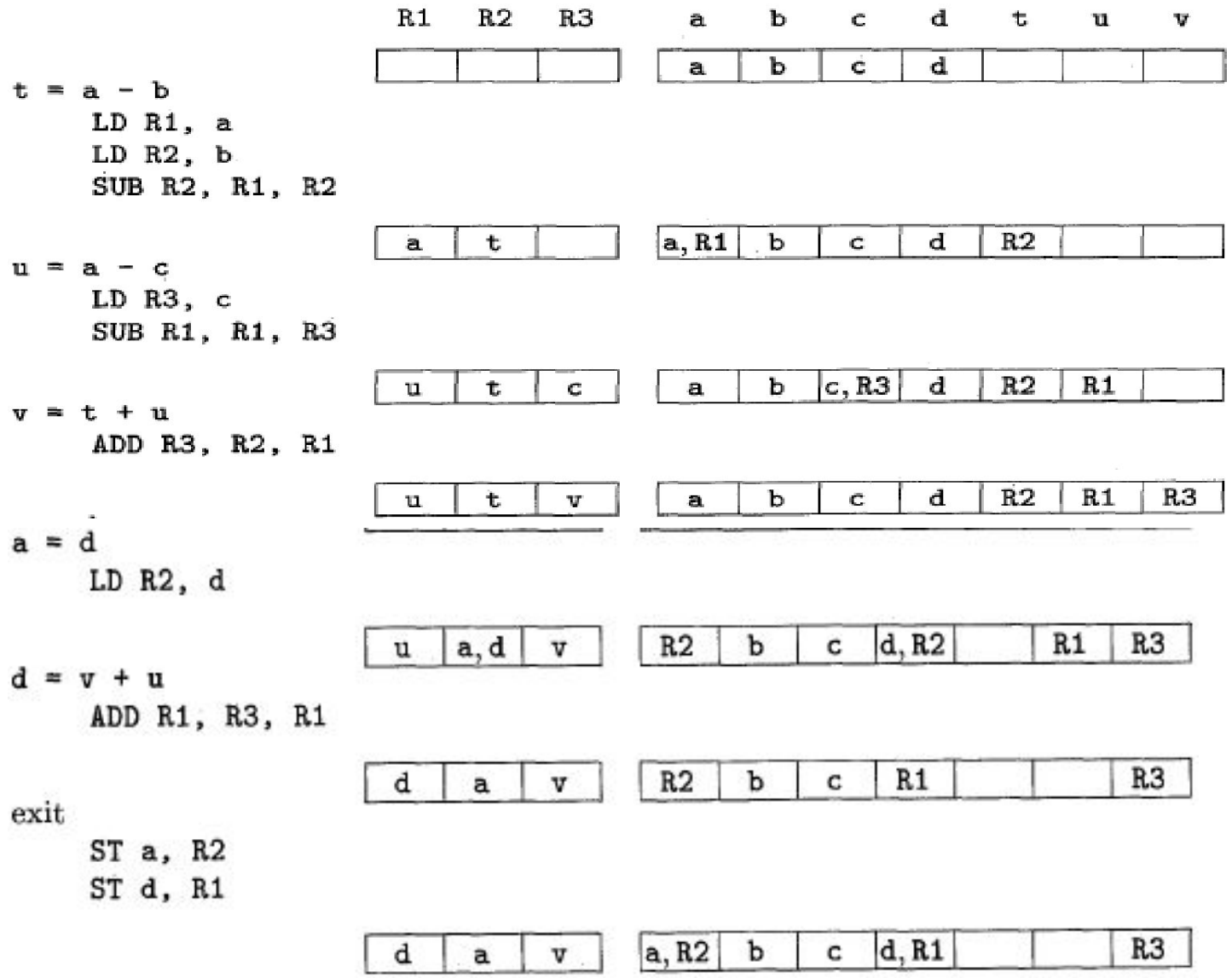
- Use getReg(x = y + z) to select registers for x, y, and z. Call these R_x, R_y and R_z.
- If y is not in R_y (according to the register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
- Similarly, if z is not in R_z, issue and instruction LD R_z, z', where z' is a location for *x* .
- **Issue the instruction ADD R_x , R_y, R_z.**

  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

Rules for updating the register and address descriptors

1. For the instruction LD R, x
   1. Change the register descriptor for register R so it holds only x.
   2. 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.
3. For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction x = y + *x*
   1. Change the register descriptor for R_x so that it holds only *x.*
   2. Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
   3. Remove R_x 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 I):
   1. Add x to the register descriptor for R_y.
   2. Change the address descriptor for x so that its only location is R_y .

Instructions generated and the changes in the register and address descriptors



Design of Function getReg

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

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, Invoke a function getreg to select registers for x, y, and z. Call these R_x, R_y and R_z.

To select registers perform the following actions: (R_y for y)

1. If y is currently in a register, pick a register already containing y as Ry. Do not issue a machine instruction to load this register, as none is needed.
2. If y is not in a register, but there is a register that is currently empty, pick one such register as Ry.
3. Difficult case occur when y is not in 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.

Design of Function getReg (Cont..)

x=y+z

Suppose v is one of the variables that the register descriptor for R says is in R. Make sure that v's value either is not really needed or there is somewhere else we can get the value of v.

Possibilities are

1. If the address descriptor for v says that v is somewhere besides R, then we are OK.
2. If v is x, the variable 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.
3. 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.
4. If we are not OK by one of the first three 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.

Peephole Optimization

Peephole Optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

- The objective of peephole optimization is:

- To improve performance
- To reduce memory footprint
- To reduce code size

Peephole optimization Techniques

1. Eliminating Redundant Loads and Stores
2. Eliminating Unreachable code
3. Flow-of-control optimizations
4. Algebraic simplifications
5. Use of machine idioms

1. Eliminating Redundant Loads and Stores

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed.

- LD a, R0
  ST R0, a

We can delete the store instruction because the first instruction ensures that the value of ‘a’ has already been loaded in to register R0.

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

Eliminating jumps over jumps

- if debug == 1 goto **L1**
  goto L2
  L I : print debugging information
  L2:
- if debug !=1 goto **L2**
  print debugging information
  L2:

3. Flow-of-control optimizations

goto L1
...
Ll: goto L2

Can be replaced by:

goto L2
...
Ll: goto L2

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

Can be replaced by:

if a<b goto L2
...
Ll: goto L2

4. Algebraic simplifications

Eliminate instructions like the following:

- x=x+0
- x=x*1
- Reduction-in-strength transformations can be applied in the peep-hole to replace expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For ex-ample, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be approximated as multiplication by a constant, which may be cheaper.

5**. Use of machine idioms**

The target machine may have hardware instructions to implement certain spe-cific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some ma-chines 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 + 1.