

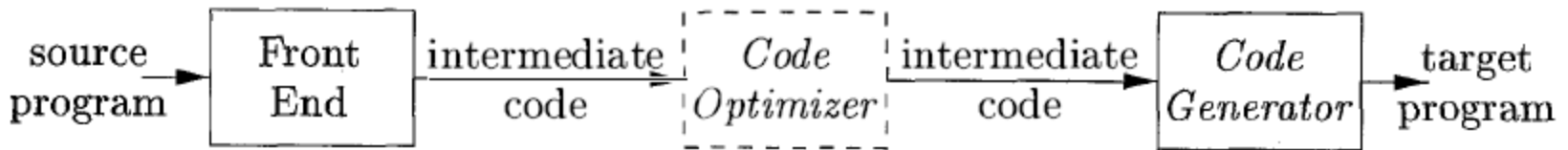
Principles of Compiler Design

Chapter 8

Code Generation and Optimization

Code Generation

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



- A **code generator** has **three primary tasks**: **instruction selection**, **register allocation and assignment**, and **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 is that it produces correct code.
- Input to the **code generator**
 - IR + Symbol table
 - We assume **front end** produces **low-level IR**, i.e. values of names in it can be directly manipulated by the *machine instructions*.
 - **Syntactic** and **semantic errors** have been already detected.
- **The target program [Output]**
 - Common target architectures are:
 - RISC(Reduced Instruction Set Computer),
 - CISC(complex instruction set computer) and
 - Stack based machines

Contd...

- 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**.
- **Stack-based machines** almost **disappeared** because it was **felt** that the **stack organization** was too limiting and required too many swap and **copy** operations.
- In this chapter we use a very simple **RISC-like** computer with addition of some **CISC-like addressing modes**.

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.
- 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)
```

Contd...

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

$$\begin{aligned}a &= b + c \\ d &= a + e\end{aligned}$$

would be translated into

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

- Here, the fourth statement is **redundant** since it loads a value that has just been *stored*, and so is the third if *a* is *not* subsequently used.
- The quality of the generated code is usually determined by its **speed** and **size**.

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.
- 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.
- Finding an optimal assignment of registers to variables is difficult, even with single-register machines.

Contd...

- Consider the two three-address code sequences in which the only difference in (a) and (b) is the operator in the second statement.

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

(a)

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

(b)

- The shortest assembly-code sequences for (a) and (b) are given as

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

(a)

```
L   R0, a
A   R0, b
A   R0, c
SRDA R0, 32
D   R0, d
ST  R1, t
```

(b)

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.
- We assume the following kinds of instructions are available:
 - **Load operations:**
 - LD dst, addr loads the value in location addr into location dst.
 - LD r, x which loads the value in location x into register r.
 - LD r1 , r2 is a register-to-register copy in which the contents of register r2 are copied into register r1 .
 - **Store operations:** S1 x, r stores the value in register r into the location x.
 - **Computation operations:** OP dst, src1 , src2 , where OP is a operator like ADD or SUB, and dst, src1 , and src2 are locations, not necessarily distinct.
 - **Unconditional jumps:** BR L causes control to branch to the machine instruction with label L. (BR stands for branch.)
 - **Conditional jumps:** 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.

Contd...



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 .
- 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 **R1 = 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 .

Contd...

- A memory location can be an **integer indexed by a register**.
 - For example, `LD R1, 100 (R2)` has the effect of setting $R1 = \text{contents}(100 + \text{contents}(R2))$, that is, of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2.
- 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 $R1 = \text{contents}(\text{contents}(100 + \text{contents}(R2)))$, 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.
- 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, # 100` adds the integer 100 into register R1.

Contd...

- **Example1:** We may execute the three-address instruction $\mathbf{b = a[i]}$ by the machine instructions(a is an array whose elements are 8-byte values) :

```
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
```

- **Example2:** $\mathbf{a[j] = c}$ 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
```

Contd...

- **Example3:** To implement a simple pointer indirection, such as the three-address statement $x = *p$, we can use machine instructions like:

```
LD  R1, p           // R1 = p
LD  R2, 0(R1)        // R2 = contents(0 + contents(R1))
ST  x, R2            // x = R2
```

- **Example4:** 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
```

Basic Blocks and Flow Graphs

- Partition the intermediate code into **basic blocks**, with the properties that
 - 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**, whose edges indicate which blocks can follow which other blocks.

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.

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.

Contd...

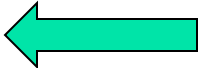
- **METHOD:** First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:
 1. The **first three-address** instruction in the intermediate code is a leader.
 2. Any instruction that is the target of a conditional or unconditional jump is a leader.
 3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Then, for each leader, its basic block consists of **itself** and **all** instructions up to but not including the *next leader or the end* of the intermediate program.

Contd..

Example: Intermediate code to set a 10 x 10 matrix to an identity matrix

```
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)
```

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;



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.

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 .

Algorithm: Determine 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 .

Contd...

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 .

Here we have used $+$ as a symbol representing any operator. If the three-address statement i is of the form $x = + Y$ or $x = y$, the steps are the same as above, ignoring z . Note that the order of steps (2) and (3) may not be interchanged because x may be y or z .

Flow Graphs

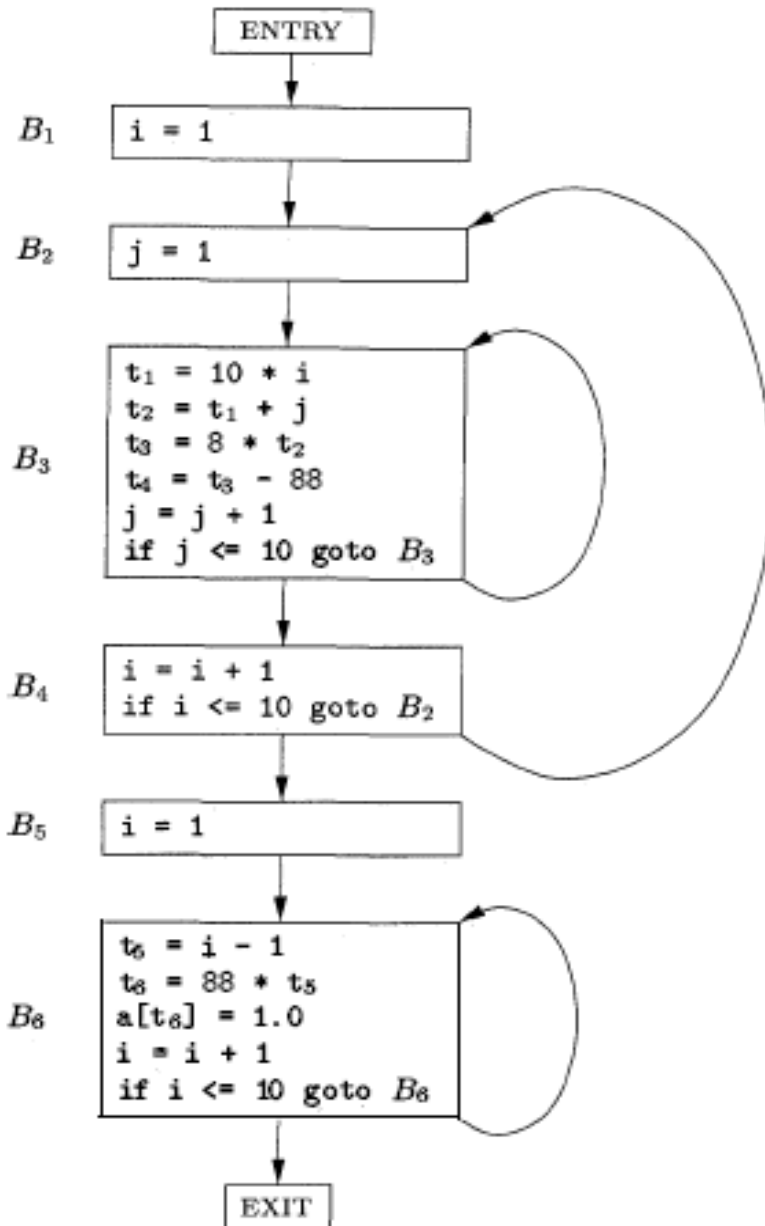


- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately 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 .

Contd...

- 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.

Contd...



Example: The set of basic blocks constructed in Example of slide10 yields the below flow graph. 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 .

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 .

DAG Representation of Flow Graphs

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.

Contd...

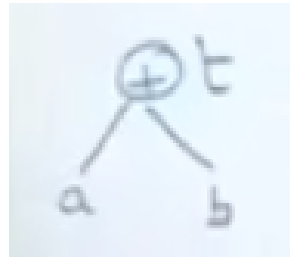


The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.

- a) We can **eliminate local common** sub expressions, 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 thereby simplify the computation.

Summery of DAG

- DAG Representation of a basic block
 - In a DAG **internal** Node represents operators
 - Leaf Node represents **identifiers** , **constants**
 - Internal Node also represents **result** of expressions
 - Example $t = a + b$



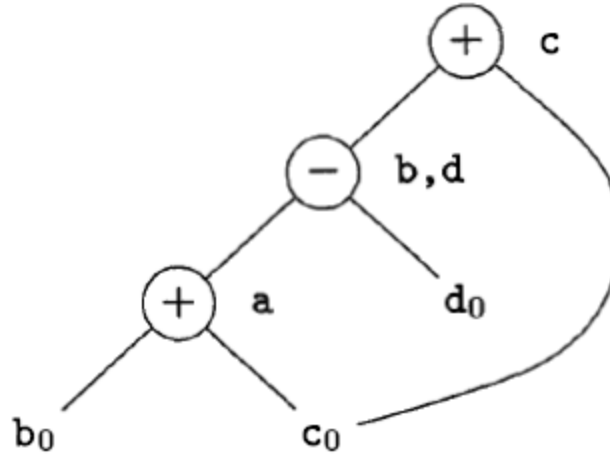
Example2. $a*(b-c)+(b-c)*d$



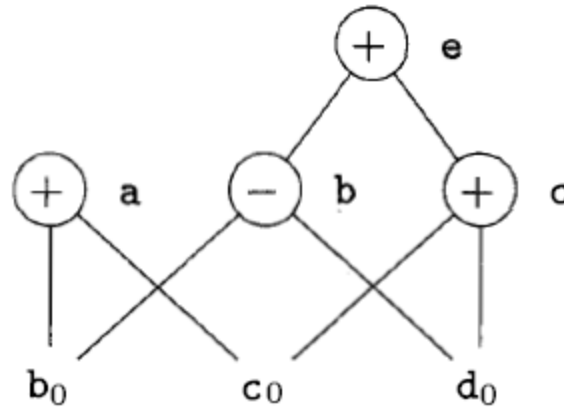
Contd...

Example: A DAG for the block

$a = b + c$
 $b = a - d$
 $c = b + c$
 $d = a - d$



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



A Simple Code Generator

- We 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. 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 sub expression 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.

Contd...



Register and Address Descriptors

- 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.

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.

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.

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 , then issue an instruction *LD* R_y , y' , where y' is one of the memory locations.
3. Similarly, if z is not in R_z , issue an instruction *LD* R_z , z' , where z' is a location for z .
4. Issue the instruction *ADD* R_x , R_y , R_z .

Contd...

■ 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
 - Change the register descriptor for register R so it holds only x.
 - 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 Rx , Ry , Rz implementing a three-address instruction $x = y + z$
 - (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 Rx .
 - (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 Ry , if needed, and after managing descriptors as for all load statements (per rule 1) :
 - (a) Add x to the register descriptor for Ry .
 - (b) Change the address descriptor for x so that its only location is Ry .

Example

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

Contd...

Design of the Function *getReg(I)*

Consider instruction $x = y + z$. First, we must pick a register for y and a register for z . The rules are as follows:

1. 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.
2. If y is not in a register, but there is a register that is currently empty, pick one such register as R_y .
3. 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.

Contd...

Possibilities for the values of R

- (a) If the address descriptor for v says that v is somewhere besides R , then we are OK.
- (b) 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.
- (c) 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.
- (d) If we are not OK by one of the first two cases, then we need to generate the store instruction $\text{ST } v, R$ to place a copy of v in its own memory location. This operation is called a spill.

Contd...

Selection of the register R_x

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x ; . 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 , in the sense described for variable v in item (3c) , and R_y holds only y after being loaded, if necessary, then R_y can also be used as R_x ; . A similar option holds regarding z and R_z .

End of Ch8...

Thank You
?