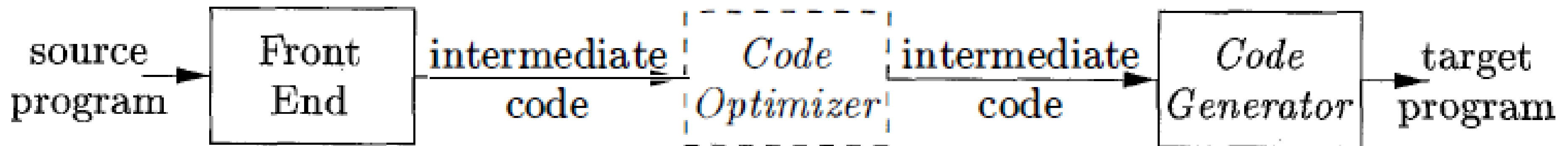
Code Generation

Md. Khorshed Alam

CSE, NDUB

Introduction

- Takes as input intermediate representation (IR) produced by the font end of the compiler, along with relevant symbol table information, & produces as output a semantically equivalent target program



Primary tasks

- Instruction selection
- Register allocation & assignment
- Instruction ordering
- Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements.
- Register allocation & 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 a Code Generator

1. 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 .
- 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; & graphical representations such as syntax trees and DAG's.

2. The Target Program

- 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**.

- ❑ 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.

Different Types of 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. 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 & 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.
- Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions & use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

3. 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**
- 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.

Affecting Factors

- 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*
- Instruction speeds & 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.

- For example, every three-address statement of the form **x = y + z**, where **x, y, & 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)
```

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
```

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

a = b + c

d = a + e

Would be translated into

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

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

- The quality of the generated code is usually determined by its speed & 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.

❖For example, if the target machine has an "increment" instruction (INC) , then the three-address statement **a = a + 1** may be implemented more efficiently by the single instruction **INC a**, rather than by a more obvious sequence that loads **a** into a register, adds one to the register, and then stores the result back into **a**:

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

4. 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 & faster than those involving operands in memory, *so efficient utilization of registers is particularly important*.

The use of registers is often subdivided into two subproblems:

- 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.

- **NP-complete problem!!**

5. Evaluation Order

- 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.
- However, picking a best order in the general case is a difficult NP-complete problem.

A Simple Target Machine Model

- Our target computer models a three-address machine with load & store operations, computation operations, jump operations, & conditional jumps.
- The underlying computer is a byte-addressable machine with n general-purpose registers, R_0, R_1, . . . , R_{n-1} .
- A full-fledged assembly language would have scores of instructions.
- Most instructions consists of an operator, followed by a target, followed by a list of source operands.
- A label may precede an instruction.

Basic Instructions

- *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` r_1, r_2 is a *register-to-register copy* in which the contents of register r_2 are copied into register r_1.
- *Store* operations: The instruction `ST` x, r stores the value in register r into the location x. This instruction denotes the assignment $x = r$.

- *Computation* operations of the form $OP\ dst, src_1, src_2$, where OP is a operator like ADD or SUB, and dst, src_1, and src_2 are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by OP to the values in locations src_1 and src_2, and place the result of this operation in location dst. For example, SUB r_1, r_2, r_3 computes $r_1 = r_2 - r_3$. Any value formerly stored in r_1 is lost, but if r_1 is r_2 or r_3, the old value is read first. Unary operators that take only one operand do not have a src_2.

- *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 B*cond* 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.

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 $\text{R1} = \mathit{contents}(\texttt{a} + \mathit{contents}(\text{R2}))$, where $\mathit{contents}(x)$ 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 memory location can be an integer indexed by a register. For example, `LD R1, 100(R2)` has the effect of setting R1 = $contents(100 + contents(\texttt{R2}))$, 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, as we shall see in the example below.

- 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 = $contents(contents(100 + contents(\texttt{R2})))$, 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.

Example 8.2: 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
```

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`.

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
```

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
```

The assignment through a pointer `*p = y` is similarly implemented in machine code by:

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

Finally, 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 & Instructions Costs

Cost = 1 + costs associated with addressing modes of the operands

- This cost corresponds to the length in words of the instruction.
- Addressing modes involving:
  - *registers have zero additional cost*,
  - *a memory location or constant have an additional cost of one*
- **because such operands have to be stored in the words following the instruction**

Examples

- The instruction `LD R0, R1` copies the contents of register `R1` into register `R0`. This instruction has a cost of one because no additional memory words are required.

- The instruction `LD R0, M` loads the contents of memory location `M` into register `R0`. 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 $contents(contents(100 + contents(\texttt{R2})))$. The cost is three because the constant 100 is stored in the word following the instruction.

More Examples

Instructions	Cost/instruction	Cost

MOV b, R_0
ADD c, R_0
MOV R_0, a
	1+1 =2
	1 + 1 =2
	1 + 1 =2
		Total=06

MOV b, a ADD c, a	1 + 2 =3 1 + 2 = 3	Total = 06
MOV *R_1, *R_0 ADD *R_2, *R_0	1 + 0 = 1 1 + 0 = 1	Total = 02
ADD R_2, R_1 MOV R_1, a	1 + 0 = 1 1 + 1 = 2	Total = 03

Basic Blocks and Flow Graphs

- **Construction Rules:**

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

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

Algorithm 8.5: 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.

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. □

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**
  $a[i, i] = 1.0;$

```
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)
```

Example: Consider the fragment of source code.
It computes the dot product of two vectors **a** & **b** of length **20**.

```
begin
        prod := 0;
        i := 1;
        do begin
                prod := prod + a [i] * b [i];
                i := i +1
        end
        while i <= 20
end
```

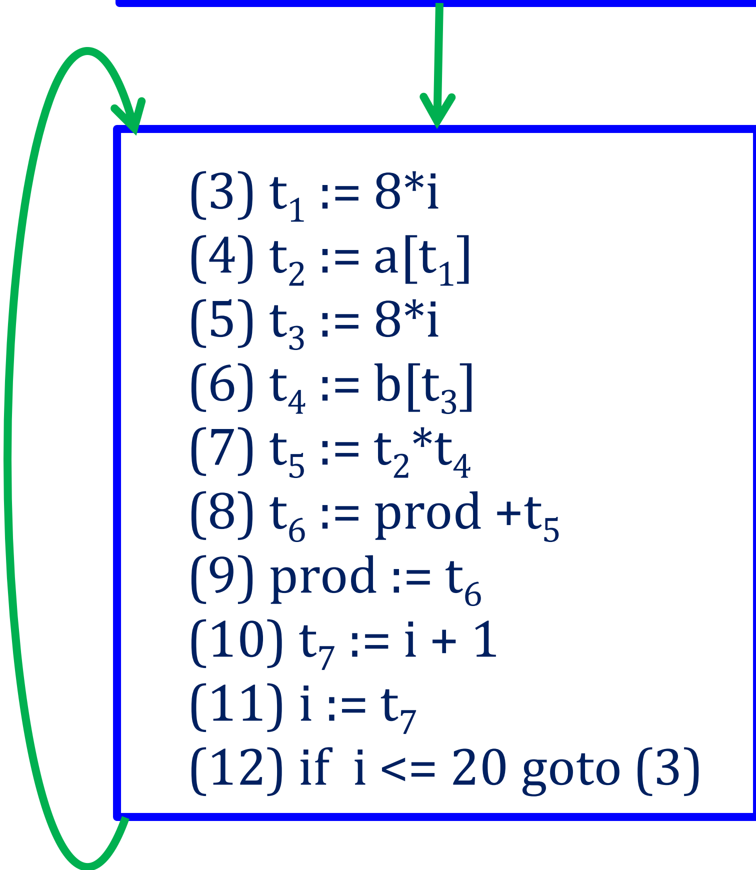
B_1

(1) prod : = 0
(2) i := 1

B_2

(3) t_1 := 8*i
(4) t_2 := a[t_1]
(5) t_3 := 8*i
(6) t_4 := b[t_3]
(7) t_5 := t_2*t_4
(8) t_6 := prod +t_5
(9) prod := t_6
(10) t_7 := i + 1
(11) i := t_7
(12) if i <= 20 goto (3)

Flow Graphs



- **First, instruction 1 is a leader by rule (1)**
- To find the other leaders, we first need to find the jumps.

➢ **03 jumps:** all conditional, at instructions **9, 11, 17**

- By rule (2) , **the targets of these jumps are leaders**; instructions **3, 2, 13**
- Then, by rule (3) , **each instruction following a jump is a leader**; those are instructions **10, 12**
- Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

- Leaders: instructions **1, 2, 3, 10, 12, 13**.
- *The basic block of each leader contains all the instructions from itself until just before the next leader.*

- ❑ basic block of **1 is just 1**
- ❑ Leader 2: block is **just 2**
- ❑ Leader 3: instructions **3 through 9**
- ❑ Leader 10: blocks **10 & 11**
- ❑ Leader 12: block is just **12**
- ❑ Leader 13: blocks **13 through 17**

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 03-address instructions, & **B does not end in an unconditional jump**

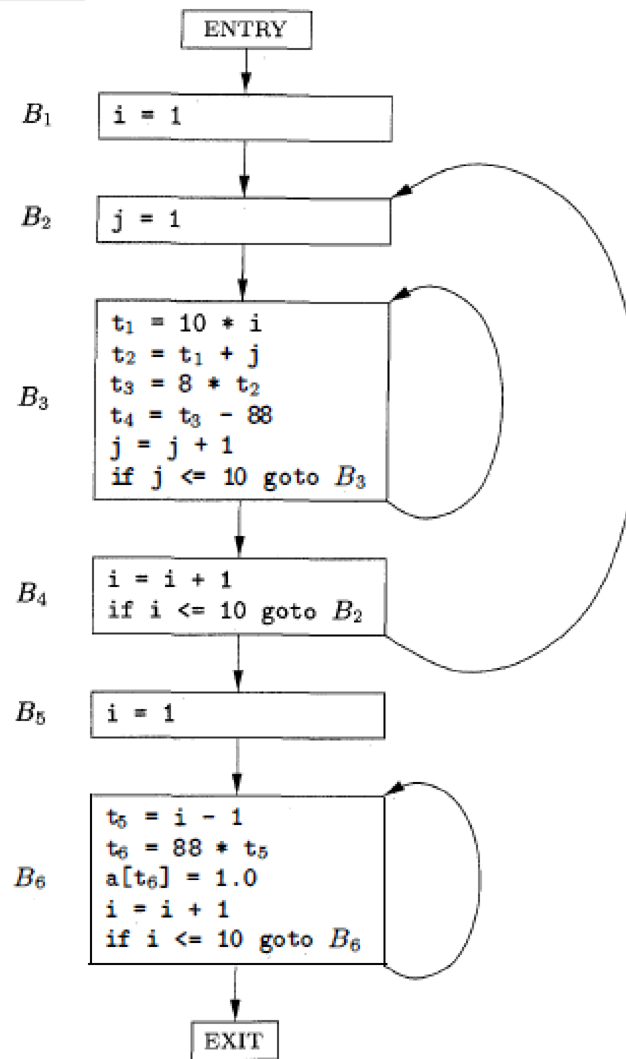
- **We say that B is a predecessor of C , & C is a successor of B** .

- Often we add two nodes, called the **entry** & **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.

- **Example**: The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9.

- **The entry points to basic block B_1, since B_1 contains the first instruction of the program**

- **The only successor of B_1 is B_2 , because B_1 does not end in an unconditional jump, & the leader of B_2 immediately follows the end of B_1 .**




```
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)
```

Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4.

Only B_6 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 B_6. □

Code Generator

- **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.

Assumptions

Machine instructions:

- ❑*LD reg, mem*
- ❑ *ST mem, reg*
- ❑*OP reg, reg, reg*

Descriptors

- **Code generation algo uses descriptors to keep track of register contents & addresses for names:**

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 Code-Generation Algorithm

The code-generation 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` we perform the following actions:

1. Invoke a function *getreg* to determine the location `L` where the result of the computation `y` *op* `z` should be stored. `L` will usually be a register, but it could also be a memory location. We shall describe *getreg* shortly.

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

3. Generate the instruction `OP z',L` where `z'` is a current location of `z`. Again, prefer a register to a memory location if `z` is in both. Update the address descriptor of `x` to indicate that `x` is in location `L`. If `L` is a register, update its descriptor to indicate that it contains the value of `x`, and remove `x` from all other register descriptors.

4. If the current values of `y` and/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 no longer will contain `y` and/or `z`, respectively.

If the current three-address statement has a unary operator, the steps are analogous to those above, and we omit the details. An important special case is a three-address statement x := y. If y is in a register, simply change the register and address descriptors to record that the value of x is now found only in the register holding the value of y. If y has no next use and is not

live on exit from the block, the register no longer holds the value of y.

If y is only in memory, we could in principle record that the value of x is in the location of y, but this option would complicate our algorithm, since we could not then change the value of y without preserving the value of x. Thus, if y is in memory we use *getreg* to find a register in which to load y and make that register the location of x.

Once we have processed all three-address statements in the basic block, we store, by MOV instructions, those names that are live on exit and not in their memory locations. To do this we use the register descriptor to determine what names are left in registers, the address descriptor to determine that the same name is not already in its memory location, and the live variable information to determine whether the name is to be stored. If no live-variable information has been computed by data-flow analysis among blocks, we must assume all user-defined names are live at the end of the block.

The Function *getreg*

The function *getreg* returns the location L to hold the value of x for the assignment x := y *op* z. A great deal of effort can be expended in implementing this function to produce a perspicacious choice for L. In this section, we discuss a simple, easy-to-implement scheme based on the next-use information collected in the last section.

1. If the name y is in a register that holds the value of no other names (recall that copy instructions such as x := y could cause a register to hold the value of two or more variables simultaneously), and y is not live and has no next use after execution of x := y *op* z, then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block, or *op* is an operator, such as indexing, that requires a register, find an occupied register R. Store the value of R into a memory location (by MOV R, M) if it is not already in the proper memory location M, update the address descriptor for M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.

4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

to hold the value of ... might be

Example 9.5. The assignment `d := (a-b) + (a-c) + (a-c)` might be translated into the following three-address code sequence

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

with `d` live at the end. The code-generation algorithm given above would produce the code sequence shown in Fig. 9.10 for this three-address statement sequence. Shown alongside are the values of the register and address descriptors as code generation progresses. Not shown in the address descriptor is the fact that `a`, `b`, and `c` are always in memory. We also assume that `t`, `u` and `v`, being temporaries, are not in memory unless we explicitly store their values with a `MOV` instruction.

STATEMENTS	CODE GENERATED	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
		registers empty	
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

- The 1st call of *getreg* returns **R0** as the location in which to compute **t**.
- Since **a** is not in **R0**, we generate instructions **MOV a, R0** & **SUB b, R0**.
- Update the register descriptor to indicate that **R0** contains **t**.
- Proceeds in this way until the last three-address statement **d:=v + u** has been proceed.
- **R1** becomes empty because **u** has no next use.
- Generate **MOV R0, d** to store the **live variable d** at the end of the block
- **COST= 2 + 2 + 2 + 2 + 1 + 1 + 2 =12**
- Can be reduced to **11** (**How??**)