南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Chapter 6: Code Generation

Yepang Liu

liuyp1@sustech.edu.cn

# Outline

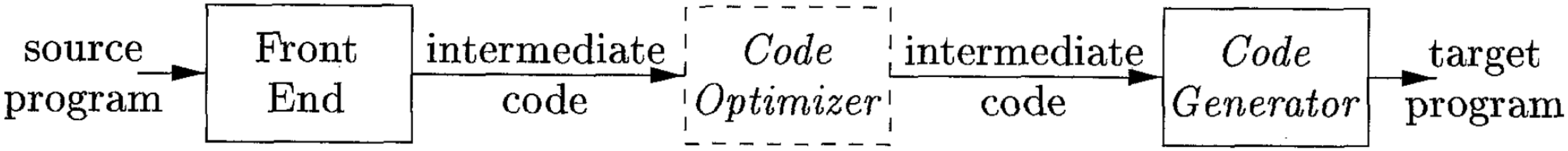
- Design Concerns
- The Target Language
- Addresses in the Target Code
  - Handling procedure calls and returns
  - Handling names in intermediate code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# Code Generator

- Input: IR + symbol table; Output: target program
- There is often an optimization phase before code generation
- Three primary tasks of a code generator:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

**Allocation:** What values should reside in registers?

**Assignment:** Which register should be used?



# Design Issues

- Design goals:
  - Correctness (the most important)
  - Ease of implementation, testing, and maintenance
- Many choices for the input IR:
  - Three-address representations: quadruples, triples, indirect triples
  - VM representations: bytecodes and stack-machine code
  - Graphical representations: syntax trees and DAG's
- Many possible target programs:
  - RISC (reduced instruction set computer), CISC (complex instruction…)
  - Absolute machine-language programs; relocatable machine-language programs (object modules, addresses are relative, need to be linked); assembly-language programs

# Outline

- Design Issues
- The Target Language
- Addresses in the Target Code
  - Handling procedure calls and returns
  - Handling names in intermediate code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# A Simple Target Machine Model

- A three-address machine with load and store, computation, jump, and conditional jump operations

| Type | Form | Effect |
|---|---|---|
| Load | LD $dst, addr$ | load the value in location $addr$ into location $dst$, where $dst$ is often a register |
| Store | ST $x, r$ | store the value in register $r$ into the location $x$ |
| Computation | $OP\ dst, src_1, src_2$ | apply the operation $OP$ to the values in locations $src_1$ and $src_2$, and place the result in location $dst$ |
| Unconditional jumps | BR $L$ | jump to the machine instruction with label $L$ |
| Conditional jumps | B$cond\ r, L$ | jump to label $L$ if the value in register $r$ pass the test B$cond$, e.g., less than zero |

# Addressing Modes (寻址模式)

**In instructions, a location can be:**

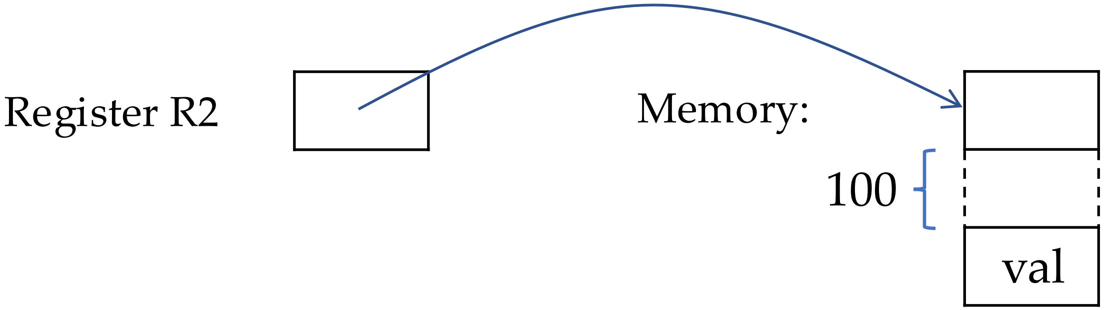
- **Variable name $x$:** the memory location reserved for $x$ ($x$'s $l$-value)
- $\boldsymbol{a(r)}$: $a$ is a variable and $r$ is a register; the memory location is computed by taking the $l$-value of $a$ and adding to it the value in register $r$ (this is very useful for accessing arrays)

# Addressing Modes (寻址模式)

**In instructions, a location can be:**

Indirect addressing mode

- **constant($r$):** a memory location can be an integer indexed by a register
  - LD R1, 100(R2) has the effect: R1 $=$ $contents(100 + contents(\text{R2}))$

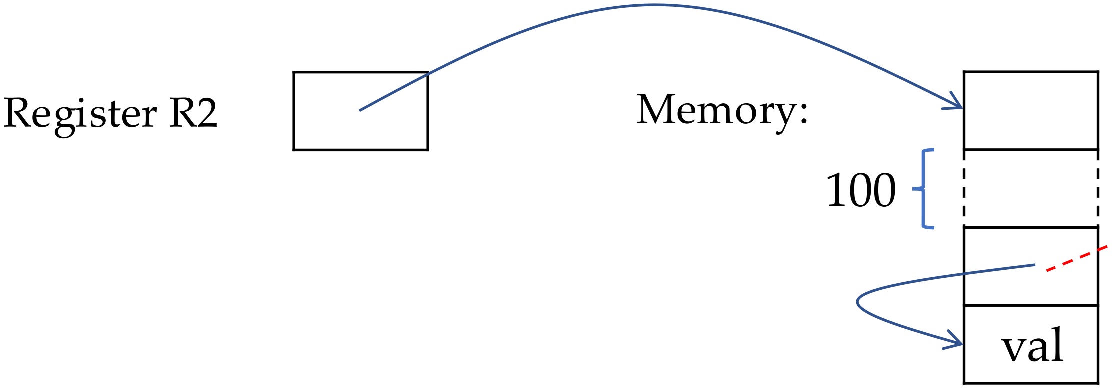


# Addressing Modes (寻址模式)

**In instructions, a location can be:**

Indirect addressing mode

- **constant($r$):** a memory location can be an integer indexed by a register
  - LD R1, 100(R2) has the effect: R1 $= contents(100 + contents(\text{R2}))$
- $*$ **constant($r$):** the memory location found in the location obtained by adding the constant to the contents of $r$ (two indirect addressing modes)
  - LD R1, $*$ 100(R2) has the effect: R1 $= contents(contents(100 + contents(\text{R2})))$



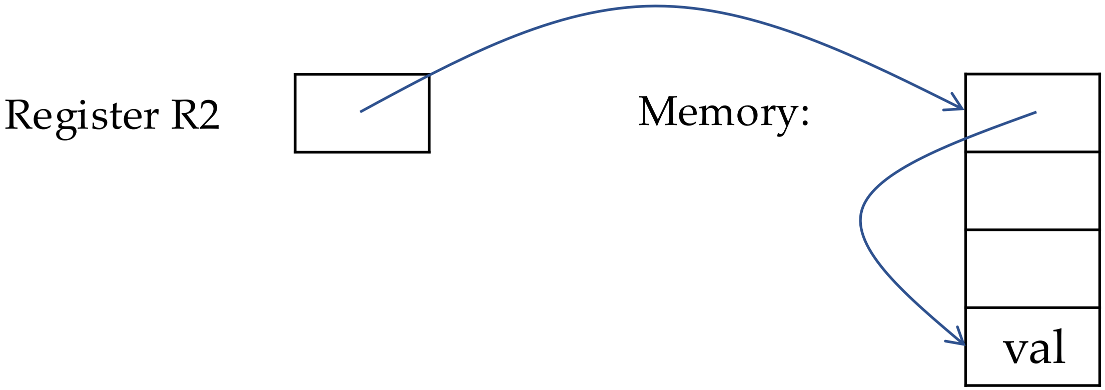
In the 1st case, the target value is stored here.

In the 2nd case, the memory address of the target value is stored here.

# Addressing Modes (寻址模式)

**In instructions, a location can be:**

- $* \boldsymbol{r}$**:** the memory location found in the location represented by the contents of register $r$ (two indirect addressing modes)
  - LD R1, $*$ R2 has the effect: R1 $= contents(contents(contents(R2)))$



# Addressing Modes (寻址模式)

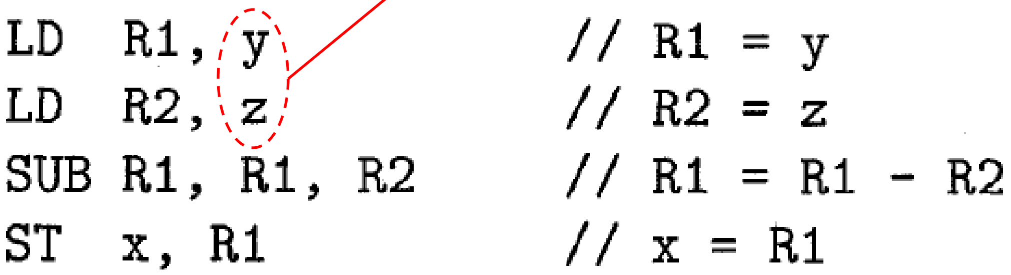
**In instructions, a location can be:**

- **#constant:** immediate constant addressing mode
  - LD R1, #100 loads the integer 100 into register R1

# Examples (1)

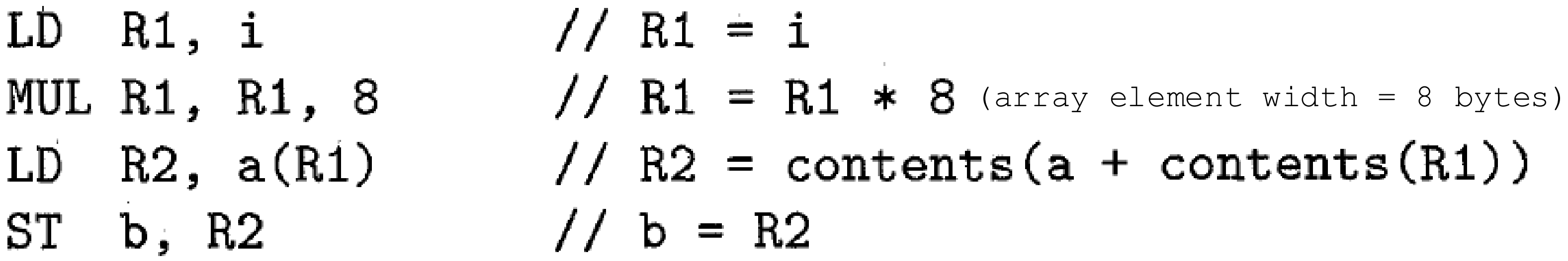
- $x = y - z$

Will be further replaced with real addresses



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

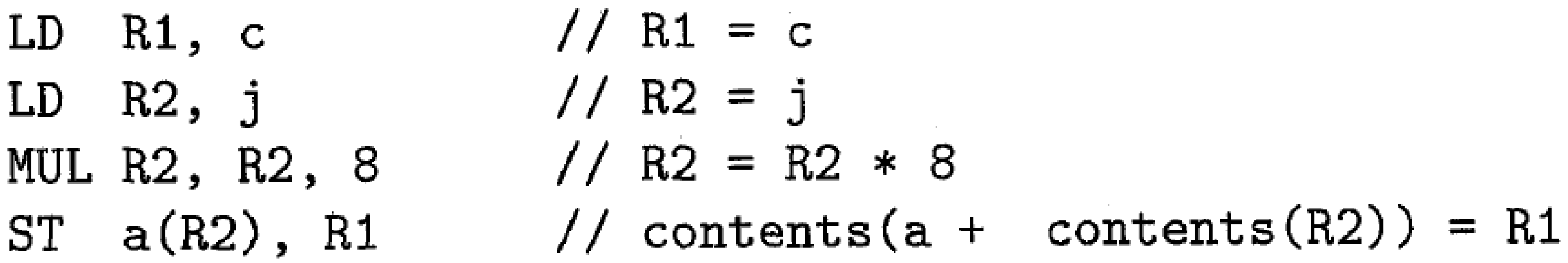
- $b = a[i]$



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

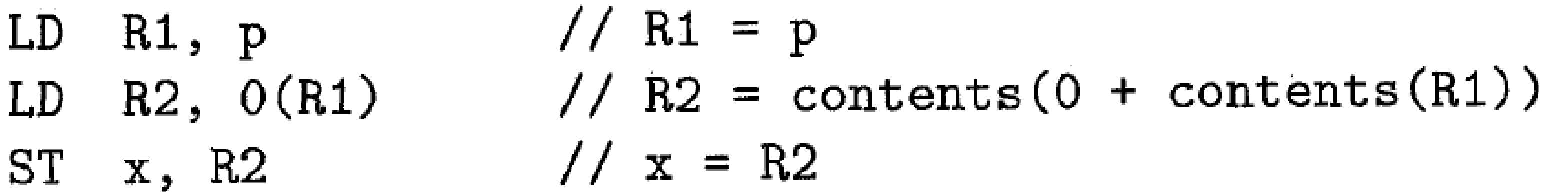
# Examples (2)

- $a[j] = c$



```
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
```

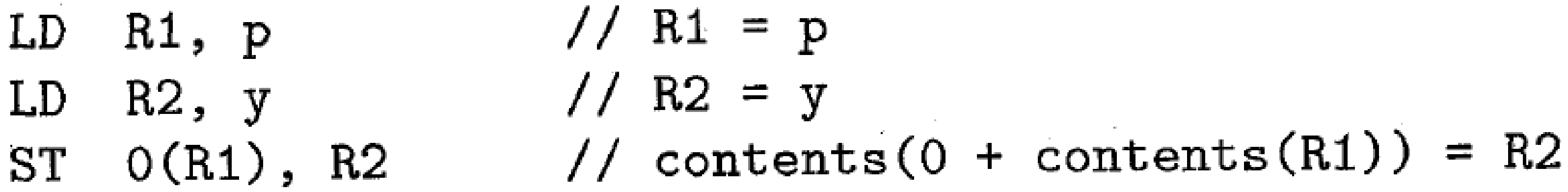
- $x =* p$



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

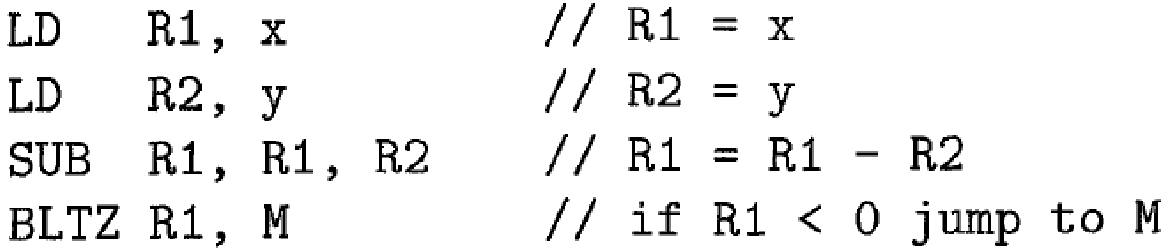
# Examples (3)

- $*p = y$



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

- 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           // if R1 < 0 jump to M
```

`M` is a label that represents the first machine instruction generated from the three-address instruction that has label L

# Outline

- Design Issues
- The Target Language
- Addresses in the Target Code
  - Handling procedure calls and returns
  - Handling names in intermediate code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# Addresses in the Target Code

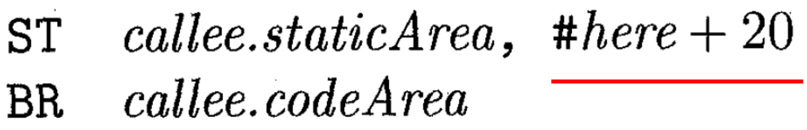
- How to generate code for procedure calls and returns?
  - Static allocation (静态分配)
  - Stack allocation (栈式分配)

- How to replace names in IR by code to access storage locations?

# Static Allocation (静态分配)

- The size and layout of activation records are determined by the code generator via the information in the symbol table
  - *staticArea* gives the address of the beginning of an activation record

- Target program code for the three-address code: call *callee*



```
ST    callee.staticArea,  #here + 20
BR    callee.codeArea
```

Store the return address (the address of the instruction after BR) at the beginning of the callee's activation record in the *Stack* area of the run-time memory[1, 2]

*codeArea* gives the address of the first instruction of the *callee* in the *Code* area of the run-time memory

1. In the example, the return address is stored at the beginning of the callee's activation record, which is different from what we discussed in the last chapter, but this is fine since the return address is a fixed length item. The order among actual parameters, return value, control link, saved machine status does not matter.
2. Why 20? 3 constants + 2 instructions = 5 words

# Static Allocation (静态分配)

- Code for the *return* statement in a *callee*

$$\mathtt{BR} \quad *\mathit{callee.staticArea}$$

Transfer control to the address saved at the beginning of the *callee*'s activation record

**Note:** Why*? Because $\mathit{callee.staticArea}$ is the address of the beginning of the activation record; the return address is stored there.

# Example

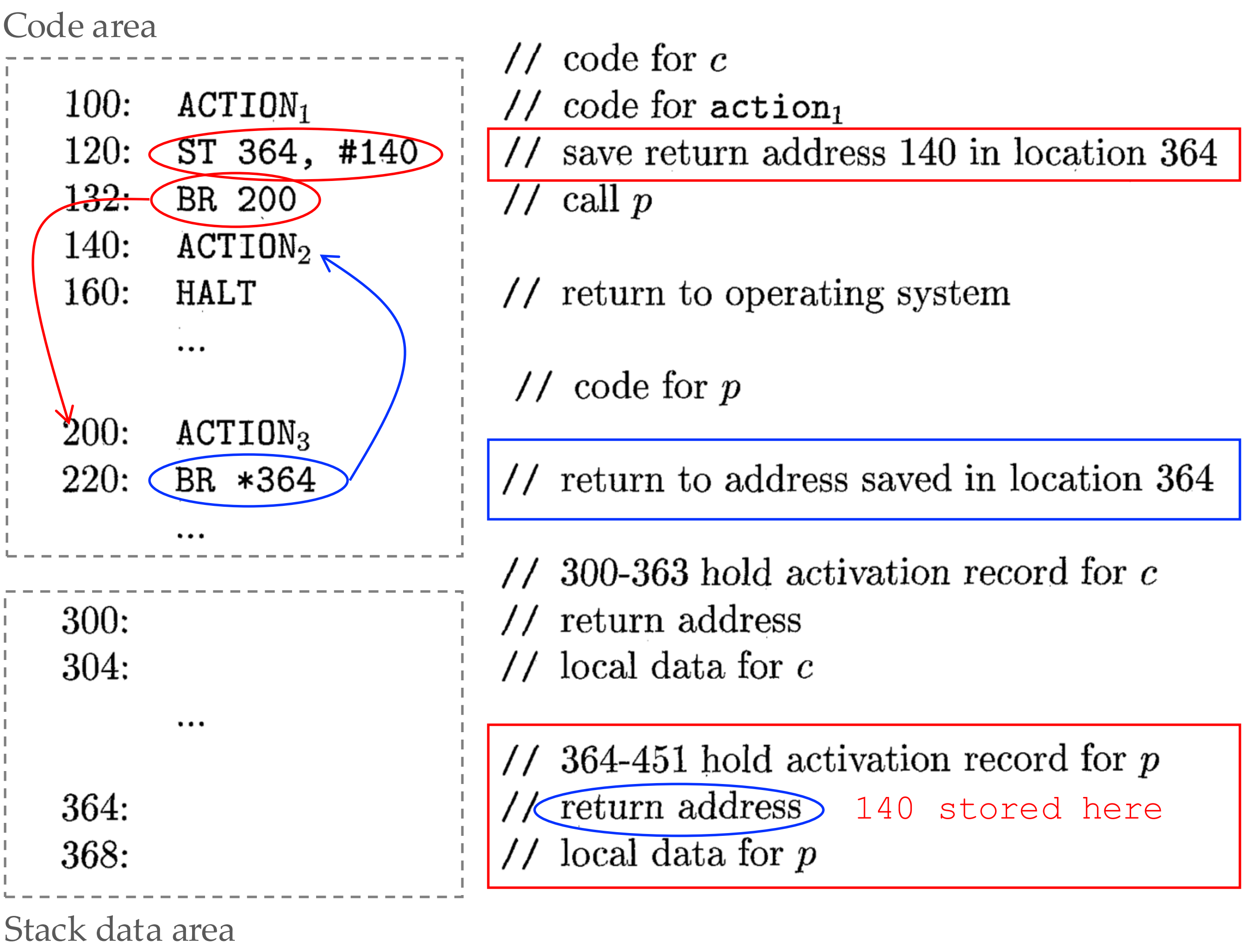
**Note:** Here the return address is stored at the beginning of the callee's activation record, which is different from the last chapter. This is fine since the order among actual parameters, returned values, and saved machine status does not matter.

Three-address code for c:

```
action1
call p
action2
halt
```

Three-address code for p:

```
action3
return
```



Code area

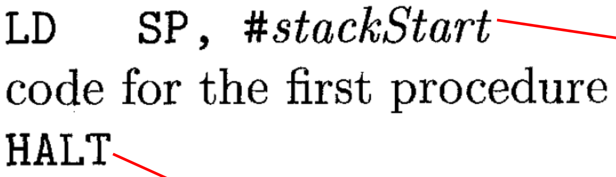
```
                    // code for c
100: ACTION1        // code for action1
120: ST 364, #140   // save return address 140 in location 364
132: BR 200         // call p
140: ACTION2
160: HALT           // return to operating system
     ...
                    // code for p
200: ACTION3
220: BR *364        // return to address saved in location 364
     ...
```

```
                    // 300-363 hold activation record for c
300:                // return address
304:                // local data for c
     ...
                    // 364-451 hold activation record for p
364:                // return address   140 stored here
368:                // local data for p
```

Stack data area

# Stack Allocation (栈式分配)

- Static allocation uses absolute addresses
  - It only works in the simplest case, not suitable for real cases
- Static allocation can become stack allocation by using relative addresses for storage in activation records
  - Maintain in a register SP a pointer to the beginning of the activation record on top of the stack
- The code for the first procedure (`main`)

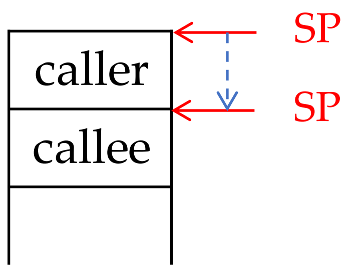


```
LD    SP, #stackStart
code for the first procedure
HALT
```

**Initialization:** setting SP to the start of the *stack* area in run-time memory

Terminate execution

# Stack Allocation (栈式分配)



- A procedure calling sequence

Additional work comparing to static allocation

Each takes 4 bytes

```
ADD   SP, SP, #caller.recordSize      // increment stack pointer
ST    *SP, #here + 16                 // save return address *
BR    callee.codeArea                 // jump to the callee
```

- The return sequence

```
BR   *0(SP)                           // return to caller        (done in callee)
SUB  SP, SP, #caller.recordSize       // decrement stack pointer (done in caller)
```

Additional work comparing to static allocation

* Return address is at the beginning of the activation record

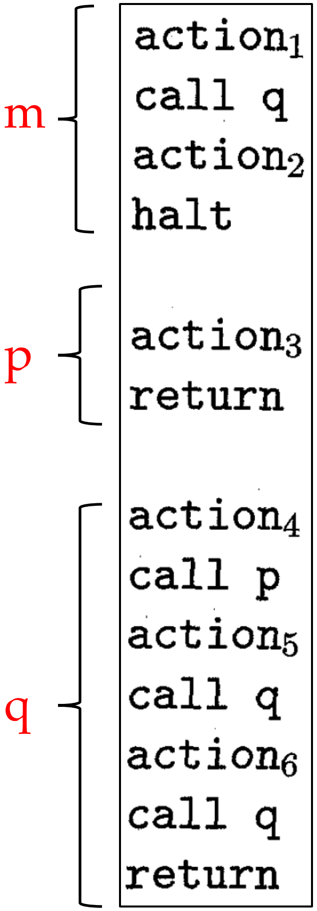
# Example

Calling sequence

m -> q

Return sequence

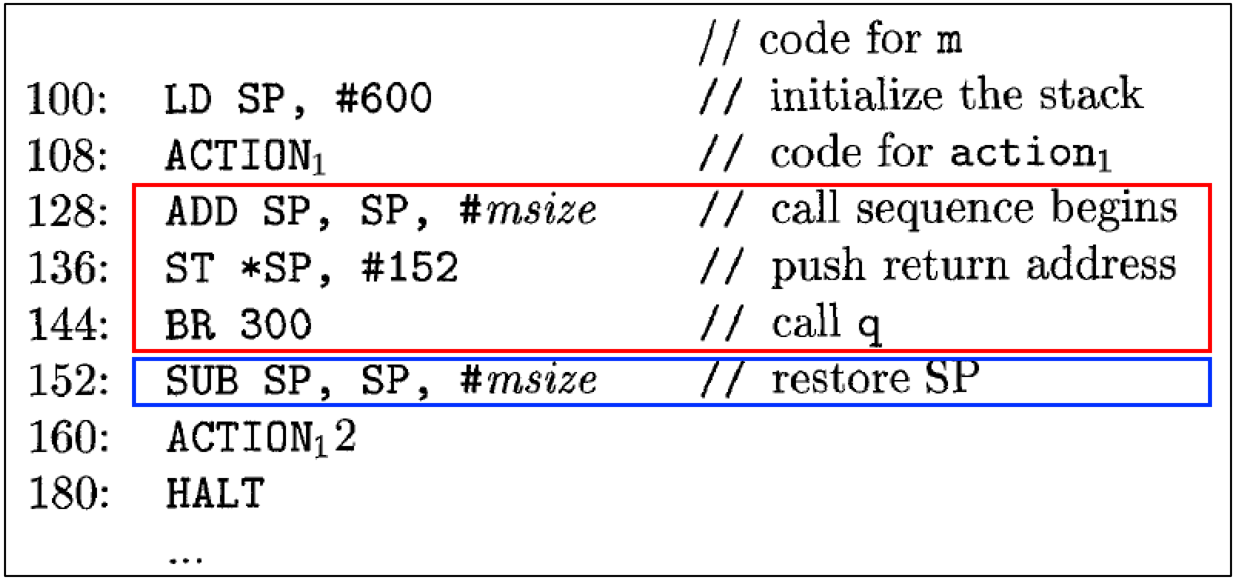
q -> m



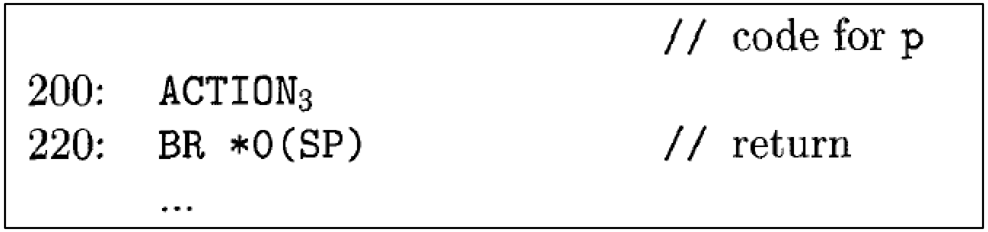
```
action1
call q
action2
halt

action3
return

action4
call p
action5
call q
action6
call q
return
```



```
                                // code for m
100:  LD SP, #600               // initialize the stack
108:  ACTION1                   // code for action1
128:  ADD SP, SP, #msize        // call sequence begins
136:  ST *SP, #152              // push return address
144:  BR 300                    // call q
152:  SUB SP, SP, #msize        // restore SP
160:  ACTION12
180:  HALT
      ...
```



```
                                // code for p
200:  ACTION3
220:  BR *0(SP)                 // return
      ...
```

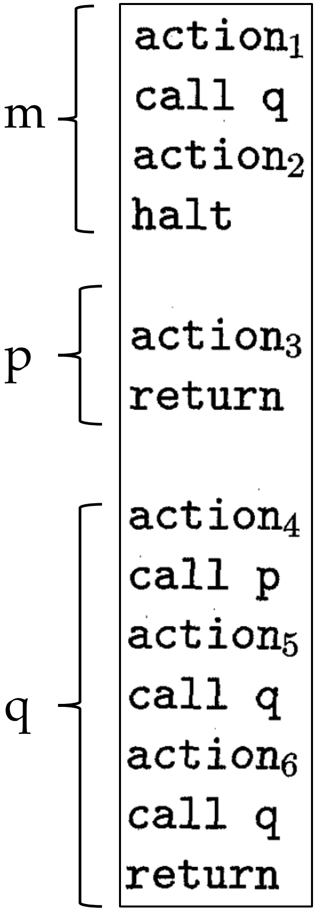
# Example

Calling sequence

m -> q

Return sequence

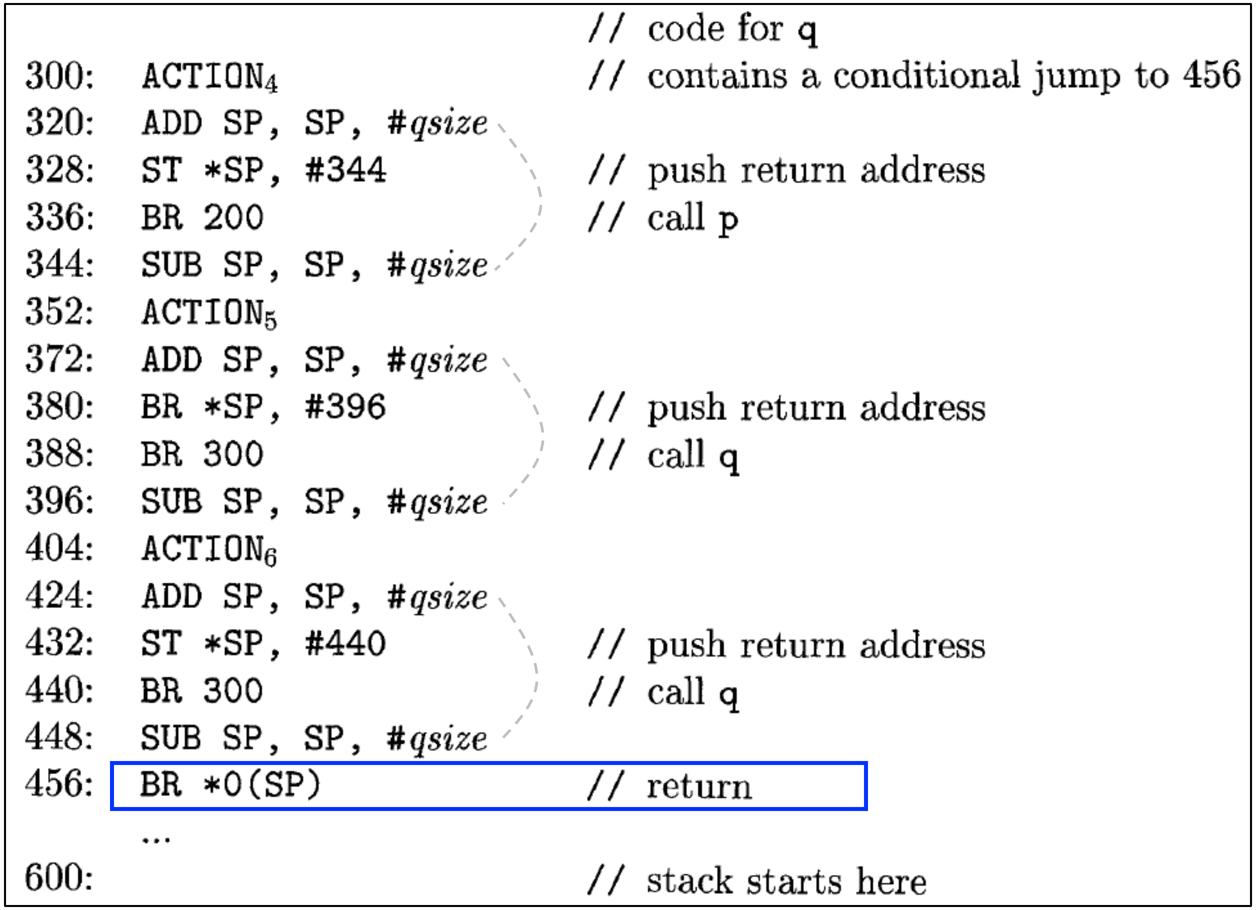
q -> m



m
```
action1
call q
action2
halt
```

p
```
action3
return
```

q
```
action4
call p
action5
call q
action6
call q
return
```



```
                                    // code for q
300:  ACTION4                       // contains a conditional jump to 456
320:  ADD SP, SP, #qsize
328:  ST *SP, #344                  // push return address
336:  BR 200                        // call p
344:  SUB SP, SP, #qsize
352:  ACTION5
372:  ADD SP, SP, #qsize
380:  BR *SP, #396                  // push return address
388:  BR 300                        // call q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST *SP, #440                  // push return address
440:  BR 300                        // call q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)                     // return
      ...
600:                                // stack starts here
```

# Outline

- Design Issues
- The Target Language
- Addresses in the Target Code
  - Handling procedure calls and returns
  - Handling names in intermediate code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# Addresses in the Target Code

- How to generate code for procedure calls and returns?
  - Static allocation (静态分配)
  - Stack allocation (栈式分配)

- How to replace names in IR by code to access storage locations?

# Run-Time Addresses for Names

- A name in a three-address statement corresponds to a symbol-table entry

- Statement `x = 0`
  - Suppose the symbol-table entry for `x` contains a relative address 12
  - If `x` is in a statically allocated area (i.e., `static`):
    - The effect of `x = 0`: `static[12] = 0`
    - Target code: `LD 112, #0` (suppose static area starts at address 100)
  - If `x` is in stack (in an activation record):
    - `LD 12(SP), #0`

# Outline

- Design Issues
- The Target Language
- Addresses in the Target Code
- Basic Blocks and Flow Graph (基本块和流图)
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# Basic Block and Flow Graph

- Graph representation of intermediate code
  - Partition the intermediate code into ***basic blocks***
    - The flow of control can only enter the basic block through its first instruction
    - Control will not leave the block, except possibly at the last instruction in the block (no halting/branching in the middle)
  - ***Flow graphs***: basic blocks are nodes and the edges indicate which block can follow which other blocks

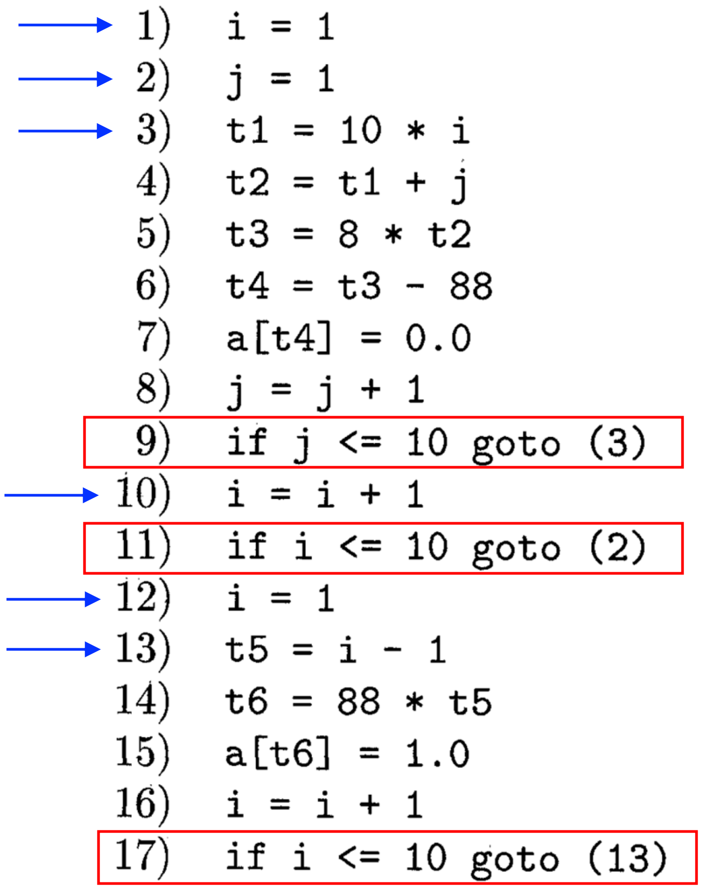
- Flow graphs form the basis of code optimization
  - They describe how control flows among basic blocks
  - We can know how values are defined and used

# Partitioning Three-Address Instructions into Basic Blocks

- **Input:** A sequence of three-address instructions
- **Output:** A list of basic blocks (each inst. is assigned to one block)
- **Method:**
  - Rules for finding *leader instructions* (首指令, the 1st instruction of a basic block)
    1. The first instruction in the entire intermediate code is a leader
    2. Any instruction that is the target of a (un)conditional jump is a leader
    3. Any instruction that immediately follows a (un)conditional 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

# Basic Block Example

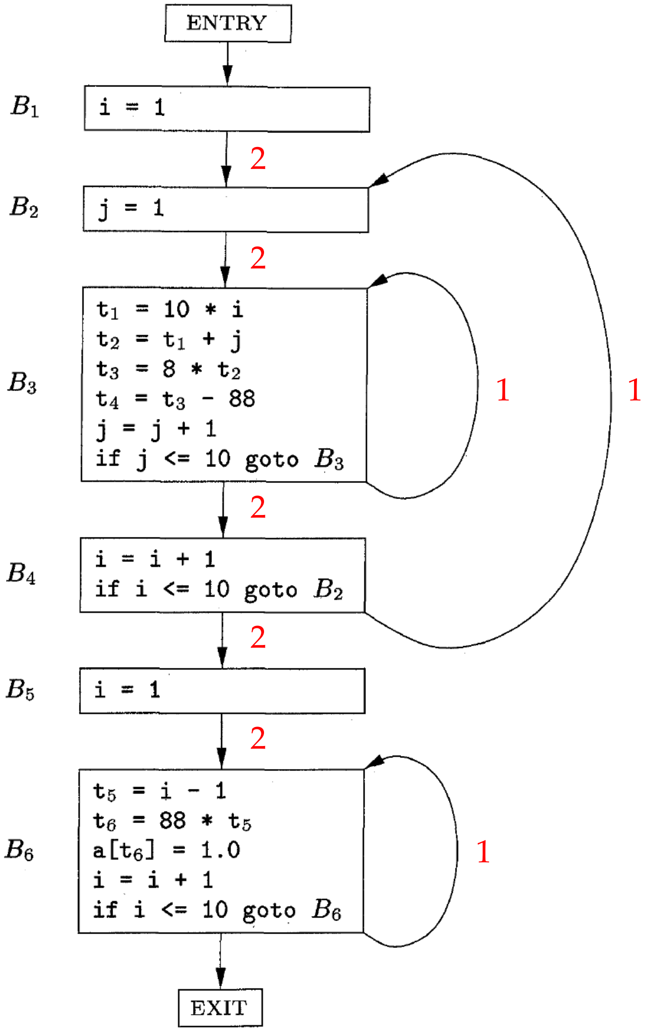
- Leader instructions:
  - The first instruction (rule #1): 1
  - Targets of jumps (rule #2): 3, 2, 13
  - Instructions immediately following jumps (rule #3): 10, 12
- Basic blocks
  - 1-1 2-2
  - 3-9 10-11
  - 12-12 13-17



```
 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 Graphs

- The nodes of the flow graphs are the basic blocks
  - There is an edge from block $B$ to block $C$ **iff** it is possible for the first instruction in $C$ to immediately follow the last instruction in $B$
- Possible reasons for the introduction of edges ($B \rightarrow C$)
  - **Case 1:** There is a (un)conditional jump from the end of $B$ to the beginning of $C$
  - **Case 2:** $C$ immediately follows $B$ in the original three-address code, and $B$ does not end in an unconditional jump
  - $B$ is the *predecessor* (前驱) of $C$; $C$ is the *successor* (后继) of $B$
- Two special nodes: *entry* (入口结点) and *exit* (出口结点)
  - They do not correspond to executable instructions
  - There is an edge from the entry to the first executable node of the flow graph
  - There is an edge to the exit from any basic block that could be executed last



# Loops (循环)

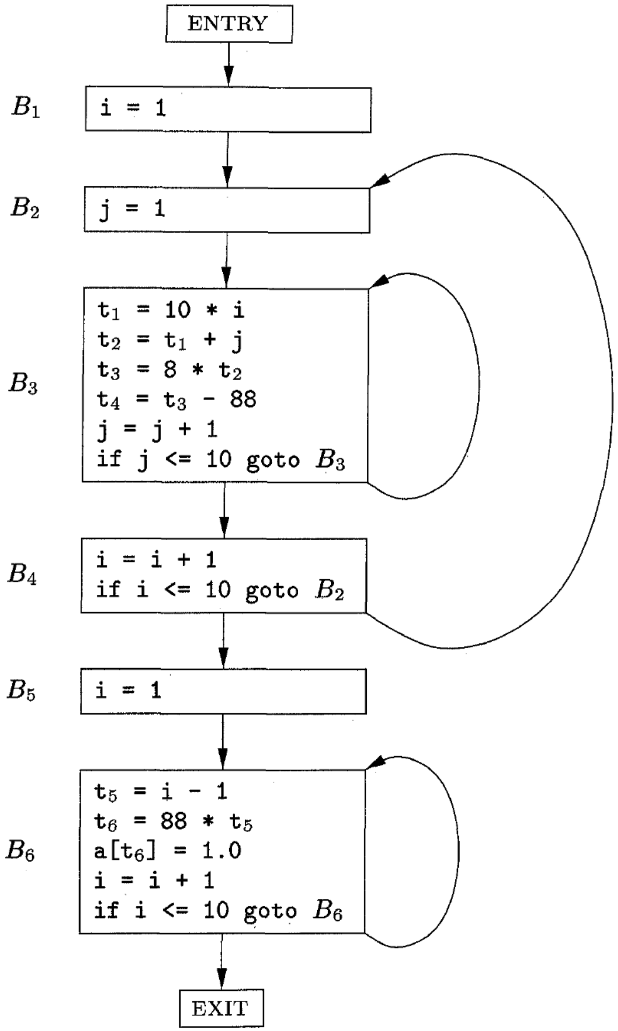
- Programs often spend most of the time in executing loops. It is important for compilers to generate efficient code for loops.

- **Definition of loops:**
  - A loop $L$ is a set of nodes in the flow graph
  - $L$ contains a node $e$ called the *loop entry* (循环入口)
  - No node in $L$ except $e$ has a predecessor outside $L$. That is, every path from the entry of the entire flow graph to any node in $L$ goes through $e$.*
  - Every node in $L$ has a nonempty path, completely within $L$, to $e$

* We say $e$ dominates the other nodes in L

# Loop Examples

- $\{B_3\}$
- $\{B_2, B_3, B_4\}$
  - $B_2$ is the loop entry
  - $B_1$, $B_5$, $B_6$ are not in the loop
    - There is a path from ENTRY to $B_1$ that does not go through $B_2$
    - $B_5$ and $B_6$ have no nonempty paths to $B_2$
- $\{B_6\}$



# Outline

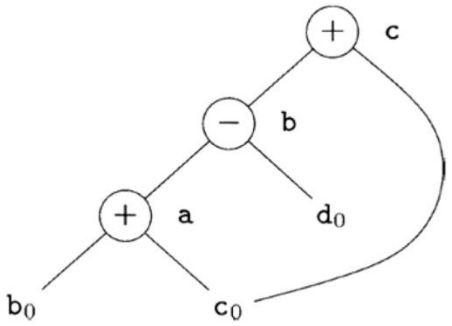
- Design Issues
- The Target Language
- Addresses in the Target Code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

We can often obtain a substantial improvement in the running time by performing *local optimizations* within each basic block

# The DAG Representation of Basic Blocks

- The DAG (directed acyclic graph) of a basic block depicts the relationships among the values of all variables in a basic block when it executes (data dependence)
- DAG enables several code-improving transformations:
  - Eliminate local common subexpressions (局部公共子表达式)
  - Eliminate dead code (死代码)
  - Apply algebraic laws (代数恒等式) to reorder operands of instructions

```
a = b + c
b = a – d
c = b + c
```



# Constructing DAG’s

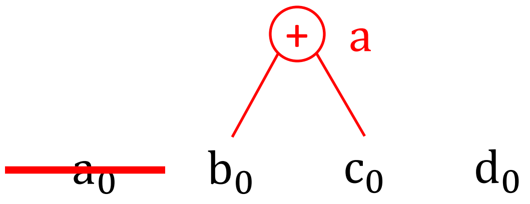
```
a = b + c
b = a – d
c = b + c
```

$a_0$ $b_0$ $c_0$ $d_0$

**Step 1:** Create a node for each of the initial values of the variables in the basic block

# Constructing DAG's

a = b + c
b = a – d
c = b + c



The variable a get a new value.
Old value $a_0$ is never used (killed)

**Step 2:** Process $a = b + c$

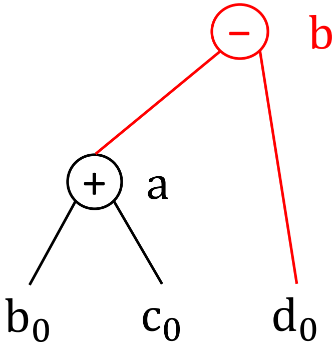
- Create a node $N$ for the statement
- Label it with +
- Attach $a$ to $N$
- The children of $N$ are those nodes corresponding to the last definitions of $b$ and $c$

# Constructing DAG's

a = b + c

b = a − d

c = b + c

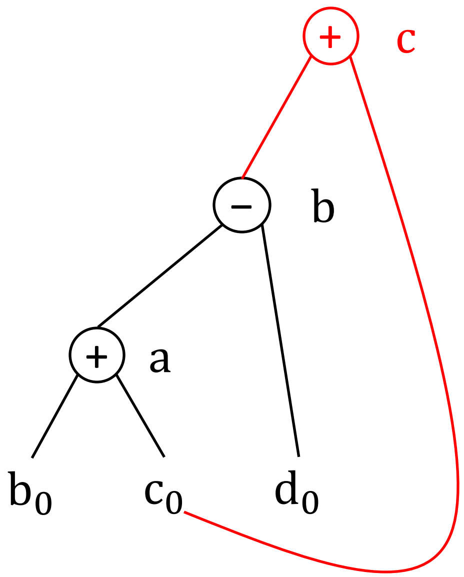


**Step 3:** Process $b = a - d$

- Create a node $N$ for the statement
- Label it with −
- Attach $b$ to $N$
- The children of $N$ are those nodes corresponding to the last definitions of $a$ and $d$

# Constructing DAG's

a = b + c
b = a – d
c = b + c

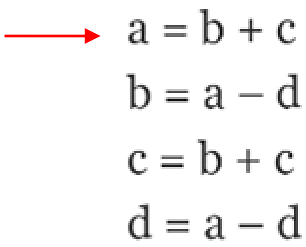


**Step 4:** Process $c = b + c$

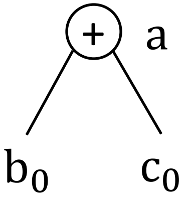
- Create a node $N$ for the statement
- Label it with +
- Attach $c$ to $N$
- The children of $N$ are those nodes corresponding to the last definitions of $b$ and $c$

# Finding Local Common Subexpression

- When creating a node $M$, check if there exists a node $N$, which has the same operator and children nodes (order matters) with $M$
- If such a node $N$ exists, we do not create the node $M$ but simply use $N$ to represent $M$

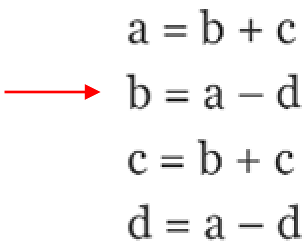


```
a = b + c
b = a − d
c = b + c
d = a − d
```

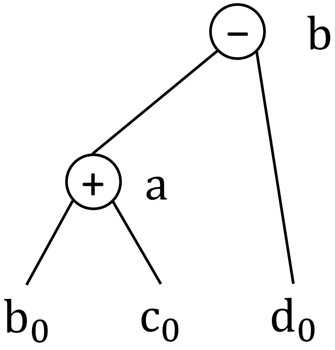


# Finding Local Common Subexpression

- When creating a node $M$, check if there exists a node $N$, which has the same operator and children nodes (order matters) with $M$
- If such a node $N$ exists, we do not create the node $M$ but simply use $N$ to represent $M$



```
a = b + c
b = a – d
c = b + c
d = a – d
```



# Finding Local Common Subexpression

- When creating a node $M$, check if there exists a node $N$, which has the same operator and children nodes (order matters) with $M$
- If such a node $N$ exists, we do not create the node $M$ but simply use $N$ to represent $M$

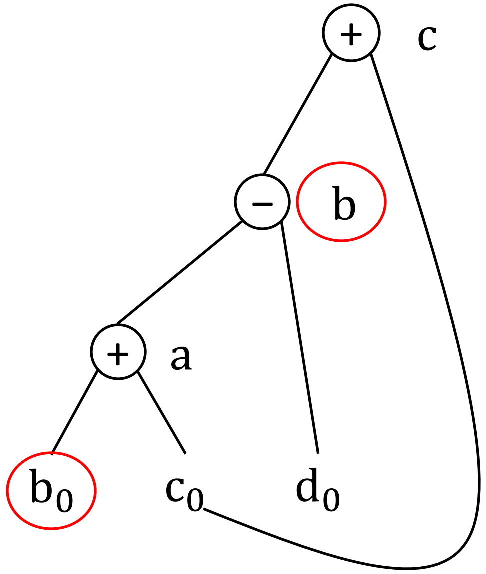
$a = \boxed{b + c}$

$b = a - d$

→ $c = \boxed{b + c}$

$d = a - d$

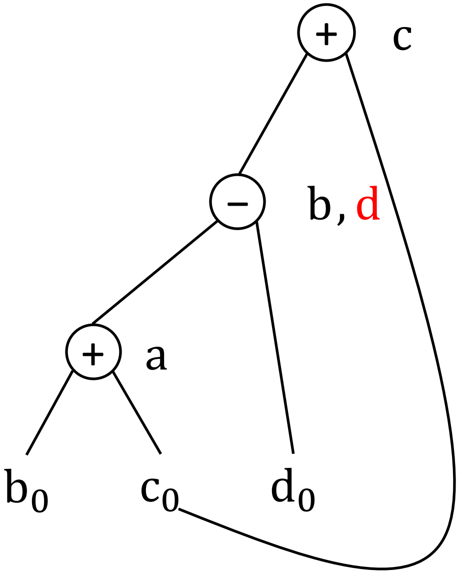
Are the two $b + c$ common subexpressions?



# Finding Local Common Subexpression

- When creating a node $M$, check if there exists a node $N$, which has the same operator and children nodes (order matters) with $M$
- If such a node $N$ exists, we do not create the node $M$ but simply use $N$ to represent $M$

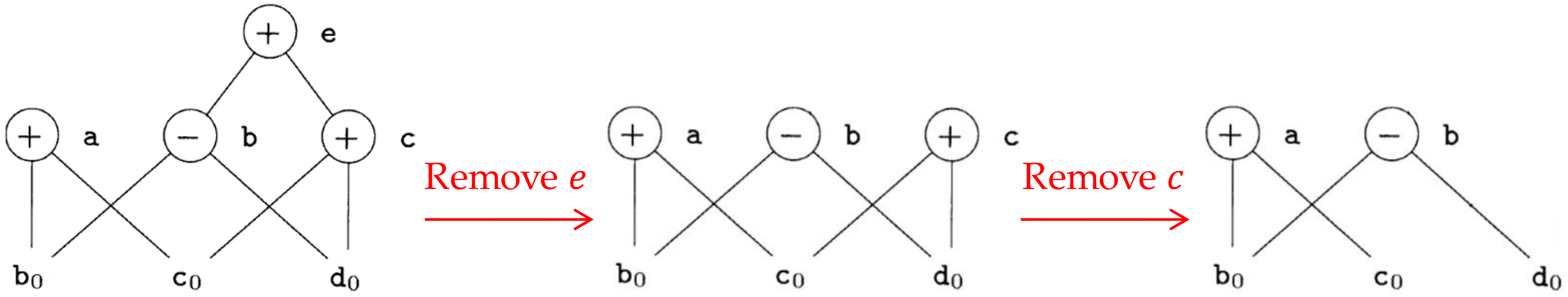
```
  a = b + c
  b = a − d
  c = b + c
→ d = a − d
```



Are the two a - d common subexpressions?

# Dead Code Elimination

- We can delete from a DAG any root (node without ancestors) that has no live variables attached
- Repeatedly applying the above transformation will remove all nodes corresponding to dead code
- Suppose in the example below, *c* and *e* are not live (they have no next uses), but *a* and *b* are live



# The Use of Algebraic Laws

- Eliminate computations (消除计算步骤) from a basic block
  - $x + 0 = 0 + x = x \quad x - 0 = x$
  - $x \times 1 = 1 \times x = x \quad x/1 = x$
- Reduction in strength (降低计算强度, replacing a more expensive operator by a cheaper one)
  - $2 \times x = x + x \qquad x/2 = x \times 0.5$
- Constant folding (常量合并)
  - $2 \times 3.14 = 6.28$

We can implement such optimizations by looking for patterns in a DAG

# Outline

- Design Issues
- The Target Language
- Addresses in the Target Code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# Code Generator

- Generate machine instructions from three-address code
  - Assume there is exactly one machine instruction for each operator
  - Assume some registers are available to hold the values used in a basic block

- Primary goal: avoid generating unnecessary loads and stores, i.e., making the best use of registers

- Four principal uses of registers:
  1. Hold the operands to perform operations
  2. Hold temporaries
  3. Hold global values
  4. Help with run-time storage management (e.g., holding stack pointers)

# Code Generation Algorithm Overview

- The basic process:

1. **Load:** Considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers

2. **Computation:** After generating the loads, it generates the operation

3. **Store:** Then, if there is a need to store the result into a memory location, it also generates the store instruction

Only generates loads when necessary

Try not to overwrite the register whose value is still of use

# Code Generation Algorithm Overview

- Two important data structures

1. Register descriptor (寄存器描述符): For each available register, keeping track of the variable names whose current value is in that register

2. Address descriptor (地址描述符): For each program variable, keeping track of the locations where the current value of that variable can be found
   - A location may be a register, a memory address, a stack location

# Algorithm Details (1)

- An essential part of the algorithm: $getReg(I)$
    - Select registers for each memory location associated with the three-address instruction $I$, according to the register/address descriptors, and data-flow info (e.g., live variables on block exit)

Obviously, how $getReg()$ selects registers affects the quality of the generated machine code

# Algorithm Details (2)

**Note:** Here, we assume $getReg()$ is given. It will be explained later.

- For a three-address instruction $x = y + z$ (here, + is a generic operator), do the following:

1. Use $getReg()$ to select registers $R_x$, $R_y$, and $R_z$ for $x$, $y$, and $z$

2. If $y$ is not in $R_y$, according to the register descriptor for $R_y$, then generate an instruction
   - LD $R_y$, $y$’ // $y$’ is a mem loc for $y$ according to $y$’s address descriptor

3. Similar to step 2, generate LD $R_z$, z’ if necessary

4. Generate instruction ADD $R_x$, $R_y$, $R_z$

# Algorithm Details (3)

- For a copy instruction $x = y$, do the following:
  - We assume $getReg()$ will always select the same registers for $x$ and $y$
  - If $y$ is not in that register $R_y$, then generate an instruction LD $R_y, y$
- Ending a basic block
  - For temporary variables used within the block, when the block ends, we can forget about their value and assume their register is empty
  - If a variable is live on block exit (or if we don't know the liveness), generate ST $x, R_x$ for each $x$ whose address descriptor does not say that its current value is in the memory location for $x$ (the value in register is newer than that in memory)

# Algorithm Details (4)

Update register and address descriptors with four rules:

1. For the instruction LD $R$, $x$:
   a) Change the register descriptor for $R$ so it holds only $x$
   b) Change the address descriptor for $x$ by adding $R$ as an additional location
   c) Remove $R$ from the address descriptor of any variable other than $x$

2. For the instruction ST $x$, $R$:
   a) Change the address descriptor for $x$ to include its own memory location*

* Values in registers cannot be older than those in memory.

Register descriptor: For each available register, keep track of the variable names whose current value is in that register

Address descriptor: For each program variable, keep track of the locations where the current value of that variable can be found.

# Algorithm Details (5)

Update register and address descriptors with four rules:

3. For an operation such as ADD $R_x, R_y, R_z$ for implementing a three-address instruction $x = y + z$:

   a) Change the register descriptor for $R_x$ so it holds only $x$

   b) Change the address descriptor for $x$ so that its only location is $R_x$

   c) Remove $R_x$ from the address descriptor of any variable other than $x$

Register descriptor: For each available register, keep track of the variable names whose current value is in that register

Address descriptor: For each program variable, keep track of the locations where the current value of that variable can be found.

# Algorithm Details (6)

Update register and address descriptors with four rules:

4. When processing a copy statement $x = y$:

   a) If LD $R_y, y$ is generated, manage descriptors using rule 1

   b) Add $x$ to the register descriptor for $R_y$

   c) Change the address descriptor for $x$ so that its only location is $R_y$

Register descriptor: For each available register, keep track of the variable names whose current value is in that register

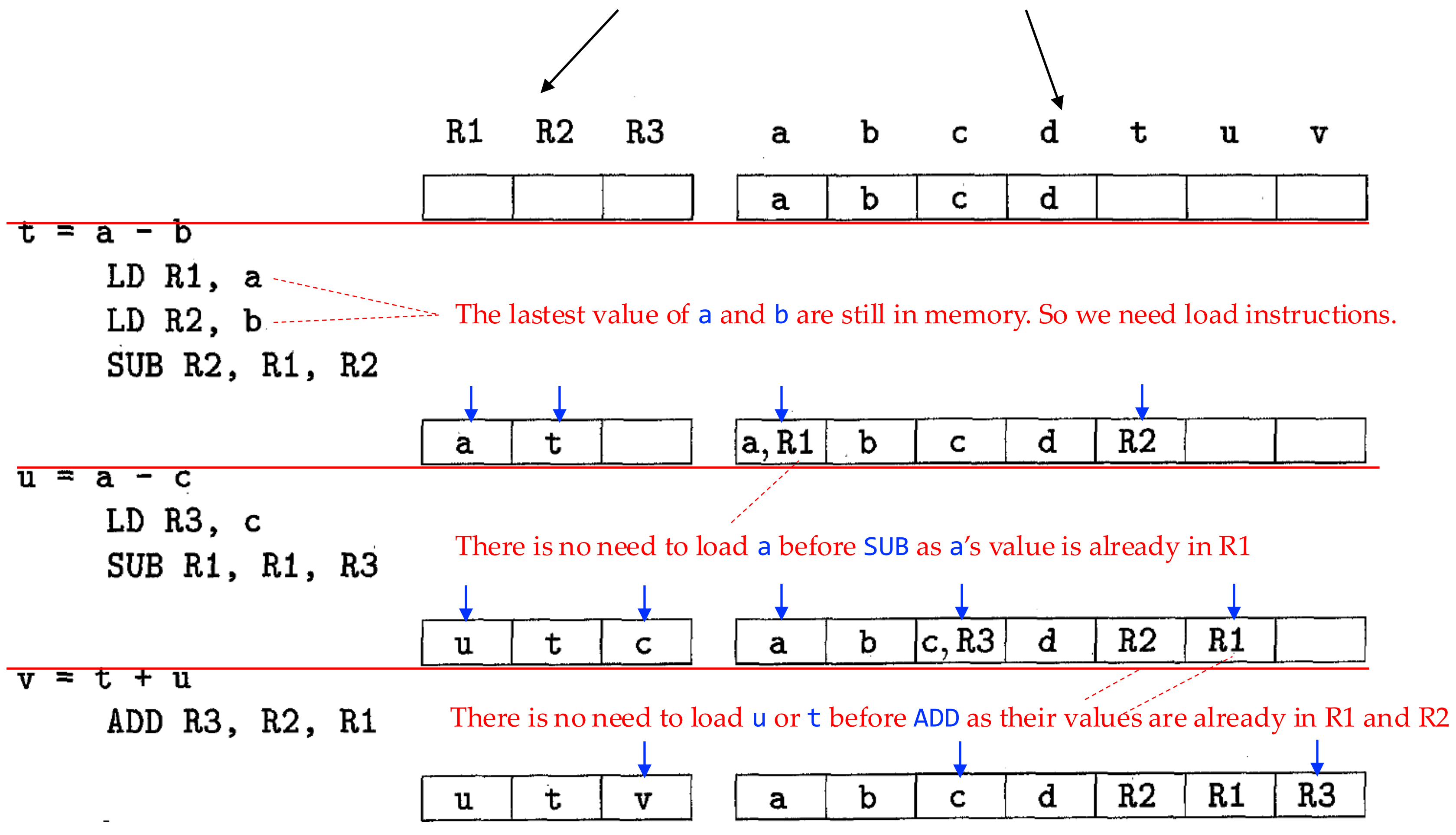
Address descriptor: For each program variable, keep track of the locations where the current value of that variable can be found.

# Example

- a, b, c, d are live at the block exit (have next uses)
- t, u, v are temporaries, local to the block

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

# Example



Register descriptor

Address descriptor

| 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
```

The lastest value of a and b are still in memory. So we need load instructions.

| R1 | R2 | R3 |
|---|---|---|
| a | t | |

| a | b | c | d | t | u | v |
|---|---|---|---|---|---|---|
| a, R1 | b | c | d | R2 | | |

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

There is no need to load a before SUB as a's value is already in R1

| R1 | R2 | R3 |
|---|---|---|
| u | t | c |

| a | b | c | d | t | u | v |
|---|---|---|---|---|---|---|
| a | b | c, R3 | d | R2 | R1 | |

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

There is no need to load u or t before ADD as their values are already in R1 and R2

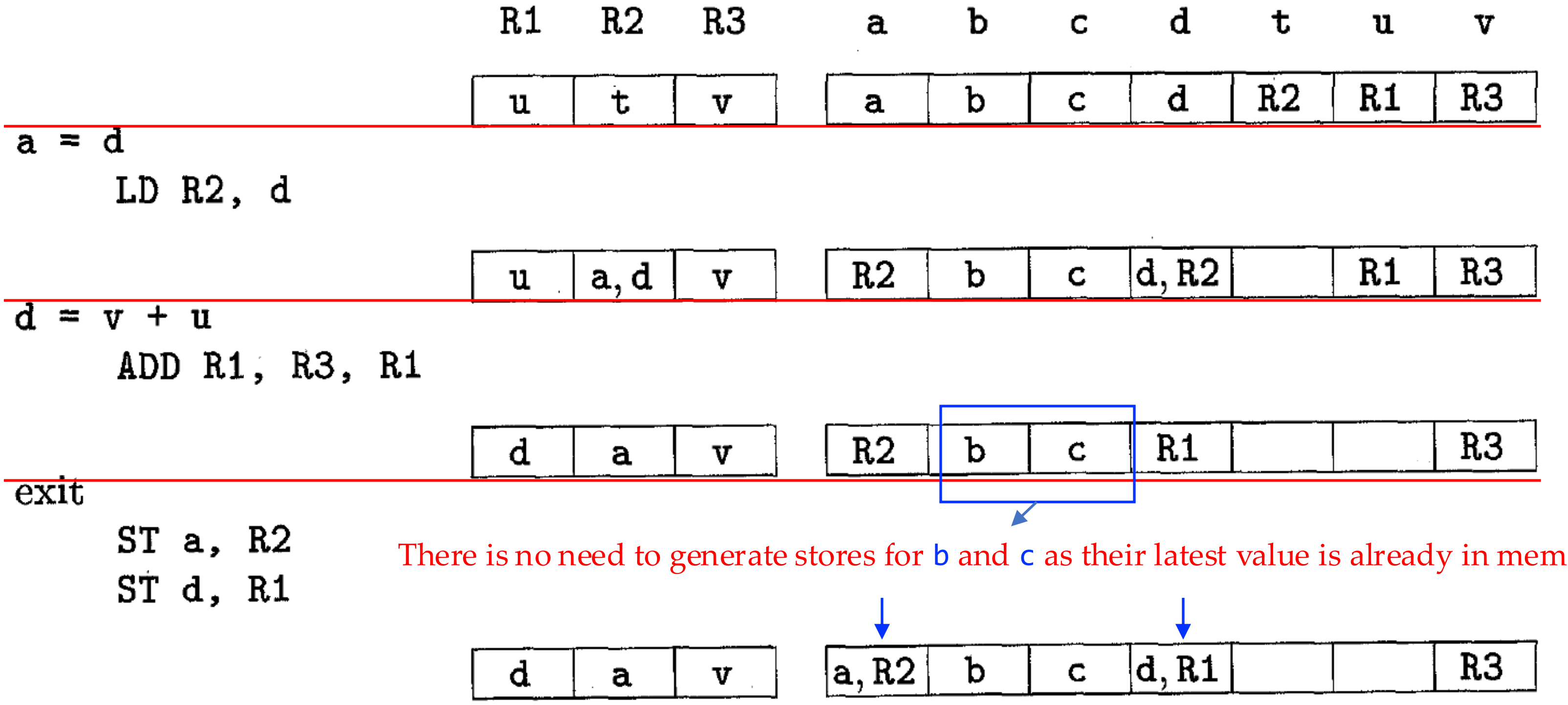
| R1 | R2 | R3 |
|---|---|---|
| u | t | v |

| a | b | c | d | t | u | v |
|---|---|---|---|---|---|---|
| a | b | c | d | R2 | R1 | R3 |

↓ annotates the changes in register/address descriptor

# Example

Why do we generate store instruction only at block exit? Why only for a and d?



| | R1 | R2 | R3 | a | b | c | d | t | u | v |
|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

There is no need to generate stores for b and c as their latest value is already in memory

# Design of the Function $getReg$

- **Goal:** Avoid too much data exchange with memory (by LD and ST)
- **Task:** Pick registers for operands and result of each three-addr inst
- For the generic example $x = y + z$, pick register $R_y$ for operand y:
  - **Case 1:** If y is currently in a register, pick a register already containing y as $R_y$. Nothing else needs to be done.
  - **Case 2:** If y is not in a register, but there is a register that is currently empty, pick one such register as $R_y$
  - **Case 3:** What if y is not in any register and there is no register that is currently empty?

# Design of the Function $getReg$

y is not in register and there is no register that is currently empty:

- Basic idea: pick one allowable register and make it “safe” to reuse
- Let $R$ be a candidate, and suppose $v$ is a variable in $R$’s register descriptor
  - It is safe to reuse $R$ if:
    - $v$’s address descriptor says we can find $v$ somewhere besides $R$
    - $v$ is $x$ and $x$ is not the other operand $z$
    - $v$ is not used later (not live after the instruction)
  - Otherwise, generate ST $v$, $R$ to place a copy of $v$ in its own memory location (*spill*, 溢出)
- If $R$ holds multiple variables, repeat the process for each such variable $v$
  - At the end, $R$’s score is the number of generated store instructions. We can pick the candidate with the lowest score

* If $v$ is $x$, since $x$ is going to be redefined, it is ok to discard its old value. Why requiring $x$ is not $z$? Because as $R$ is chosen to be reused, $y$’s value will be loaded into $R$. If $x$ is $z$, that means the current value of $z$ is in $R$, then it needs to be stored to avoid data loss.

# Design of the Function $getReg$

- Picking register $R_x$ for result $x$ in $x = y + z$:
  - Since a new value of $x$ is being computed, a register that holds only $x$ is always an acceptable choice for $x$
  - If $y$ is not used after the instruction, and $R_y$ holds only $y$, then $R_y$ can also be used as $R_x$ (same for $R_z$)
  - If $y$ is useful after the instruction and there is no register that is currently empty, then spill is needed.

- Pick registers $R_x$ and $R_y$ for $x = y$:
  - Pick $R_y$ first (as above)
  - Then choose $R_x = R_y$

# Peephole Optimization (窥孔优化)

- A simple but effective technique for locally improving target code by examining a sliding window of target instructions (the *peephole*, 窥孔) and replacing the instruction sequences within the peephole with *shorter* or *faster* sequences*
  - Redundant-instruction elimination (冗余指令消除)
  - Flow-of-control optimizations (控制流优化)
  - Algebraic simplifications (代数简化)
  - Use of machine idioms (机器特有指令的使用)

* Can also be applied directly after intermediate code generation to improve the IR

# Redundant Loads and Stores

- A naïve algorithm (not the one we introduced) may generate the following code:
  - LD $R_0$, a
  - ST a, $R_0$
- We can delete the store instruction if there is no jump to it (that is, the two instructions are in the same basic block)
  - If there is a jump to the store instruction, we could not be sure that the LD instruction is always executed before the store instruction (there might be other instructions putting a new value of a into $R_0$)

# Unreachable Code

- Example 1: An unlabeled instruction immediately following an unconditional jump can be removed
- Example 2: Jumps over jumps (级联跳转)

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

→

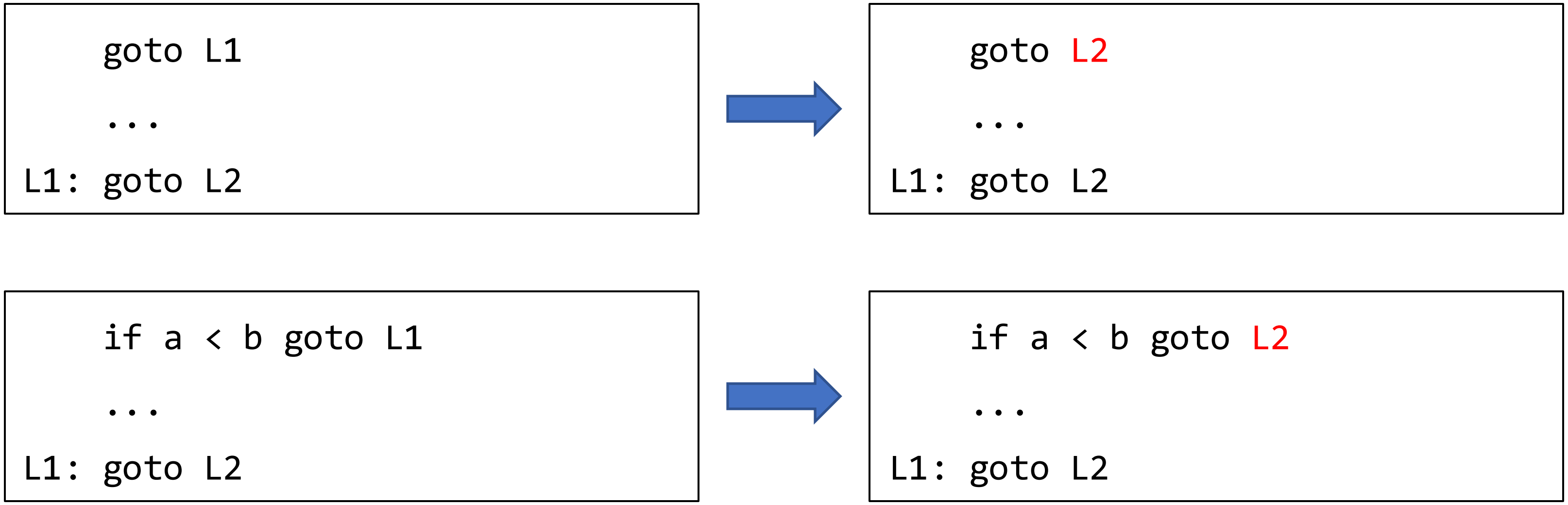
```
    if debug != 1 goto L2
    print debugging information
L2:
```

Assuming there is no other jump to L1

How to optimize the code if we know **debug** is a constant 0?

# Flow-of-Control Optimizations

- Unnecessary jumps



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

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

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

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

When can we further eliminate `L1: goto L2`?

# Outline

- Design Issues
- The Target Language
- Addresses in the Target Code
- Basic Blocks and Flow Graph
- Optimization of Basic Blocks
- A Simple Code Generator
- Register Allocation and Assignment

# Register Allocation and Assignment

- Instructions involving only register operands are faster than those involving memory operands
  - Efficient utilization of registers is vitally important in generating good code
- Register allocation (寄存器分配)
  - Decide at each point of a program what values should reside in registers
- Register assignment (寄存器指派)
  - Decide at each point of a program in which register each value should reside
- There exist many strategies for register allocation and assignment

# The Simple Strategy

- Assign specific values to certain registers
  - Assign array base addresses to a group of registers
  - Assign arithmetic computational values to another group
  - Assign the top of the stack to a fixed register …
- Advantage: Simplifies the design and implementation of a compiler
- Disadvantage: Inefficient uses of registers
  - Certain registers may go unused, while many loads and stores are generated for the other registers
- It is reasonable to reserve a few registers for base addresses, stack pointers, and the like

# Global Register Allocation

- The earlier algorithm uses registers to hold values for the duration of a single basic block
  - All live variables were stored (if necessary) at the end of each block
- To save stores and loads, we can assign registers to frequently used variables and keep these registers consistent across block boundaries (globally)
  - E.g., assign certain registers to hold the most active values in a loop
- Another strategy is to estimate the benefits of putting a variable in register (via static analysis or profiling) and allocate registers according to the estimations

# Reading Tasks

- Chapter 8 of the dragon book
  - 8.1 Issues in the Design of a Code Generator
  - 8.2 The Target Language
  - 8.3 Addresses in the Target Code
  - 8.5 Optimizations of Basic Blocks (8.5.1 – 8.5.4)
  - 8.6 A Simple Code Generator
  - 8.7 Peephole Optimization (8.7.1 – 8.7.4)
  - 8.8 Register Allocation and Assignment (8.8.1)