南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Chapter 7: Code Generation

Yepang Liu

liuyp1@sustech.edu.cn

# Outline

- Design Concerns
- 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

- 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

# 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); assembly-language programs

# 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

# 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 $\boldsymbol{x}$:** the memory location reserved for $x$ ($x$'s $l$-value)

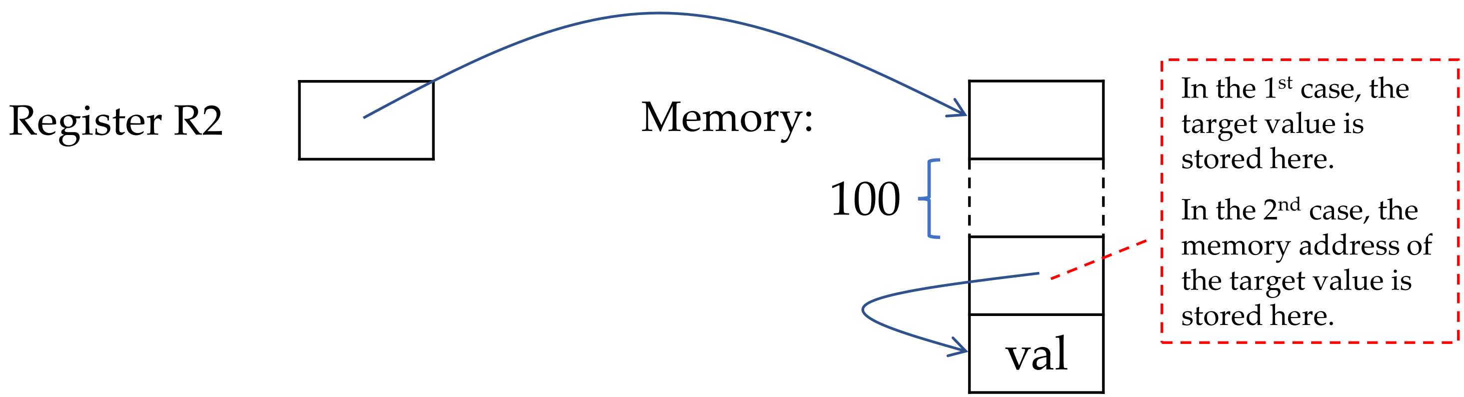
- $\boldsymbol{a(r)}$: $a$ is a variable and $r$ is a register; the memory location is computed by taking the $l$-value of $a$ and adding to it the value in register $r$ (this is very useful for accessing arrays)

# Addressing Modes (寻址模式)

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

Indirect addressing mode

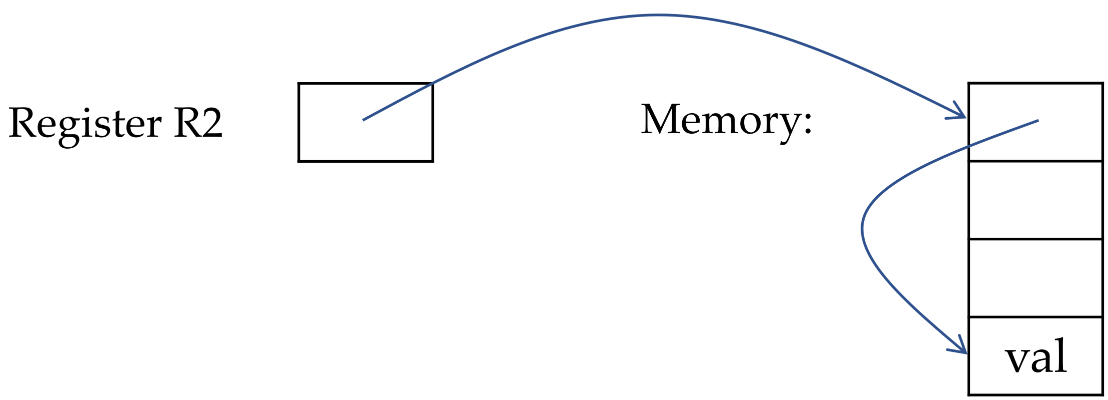
- **constant($r$):** a memory location can be an integer indexed by a register
  - LD R1, 100(R2) has the effect: R1 $= contents(100 + contents(\text{R2}))$
- **$*$ constant($r$):** the memory location found in the location obtained by adding the constant to the contents of $r$ (two indirect addressing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



# Addressing Modes (寻址模式)

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

- $*$ ***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
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
- 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
  - Constant *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

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

* Why 20? 3 constants + 2 instructions = 5 words

# Static Allocation (静态分配)

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

$$\text{BR} \quad *callee.staticArea$$

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

# Example

**Note:** Here the return address is stored at the beginning of the callee's activation record, which is different from Chapter 6. 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: | $\texttt{ACTION}_1$ | // code for $\texttt{action}_1$ |
| 120: | ST 364, #140 | // save return address 140 in location 364 |
| 132: | BR 200 | // call $p$ |
| 140: | $\texttt{ACTION}_2$ | |
| 160: | HALT | // return to operating system |
| | ... | |
| | | // code for $p$ |
| 200: | $\texttt{ACTION}_3$ | |
| 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 |
| 368: | | // local data for $p$ |

Data area

# Stack Allocation (栈式分配)

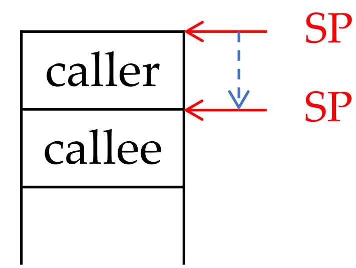
- Static allocation uses absolute addresses
- Static allocation can become stack allocation by using relative addresses for storage in activation records
  - Maintain in a register `SP` a pointer to the beginning of the activation record on top of the stack
- The code for the first procedure (`main`)

```
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`

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

p:
```
action3
return
```

q:
```
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
```

# 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`