Compilation Principle
编 译 原 理

第20讲：目标代码生成(2)

张献伟

xianweiz.github.io

DCS290, 06/03/2021

Review Questions

- What is runtime environment?

  The environment where the target program will be executed.

- For the static memory region, what are placed there?

  Code, global and static variables. Composing an executable image

- What is activation record?

  Each execution of a procedure is called activation, and AR is to manage the info needed by the execution.

- What are registers $SP and $FP used for?

  $SP points to the top of stack; $FP points to the base of current frame

- What are the schemes reference counting and tracing?

  Garbage collection to reclaim unused heap space.

Translating IR to Machine Code

- Machine code generation is machine ISA dependent*
  - Complex instruction set computer (CISC): x86
  - Reduced instruction set computer (RISC): ARM, MIPS, RISC-V

- Three primary tasks
  - Instruction selection[指令选取]
    - Choose appropriate target-machine instructions to implement the IR statements
  - Register allocation and assignment[寄存器分配]
    - Decide what values to keep in which registers
  - Instruction ordering[指令排序]
    - Decide in what order to schedule the execution of instructions

* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行？)

Instruction Selection[指令选取]

- Code generation is to map the IR program into a code sequence that can be executed by the target machine [选择适当的目标机器指令来实现IR]
  - ISA of the target machine
    - If there is ‘INC’, then for a = a + 1, ‘INC a’ is better than ‘LD a, ADD a, 1’
  - Desired quality of the generated code
    - Many different generations, naïve translation is usually correct but very inefficient

TAC code:

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

Target code:

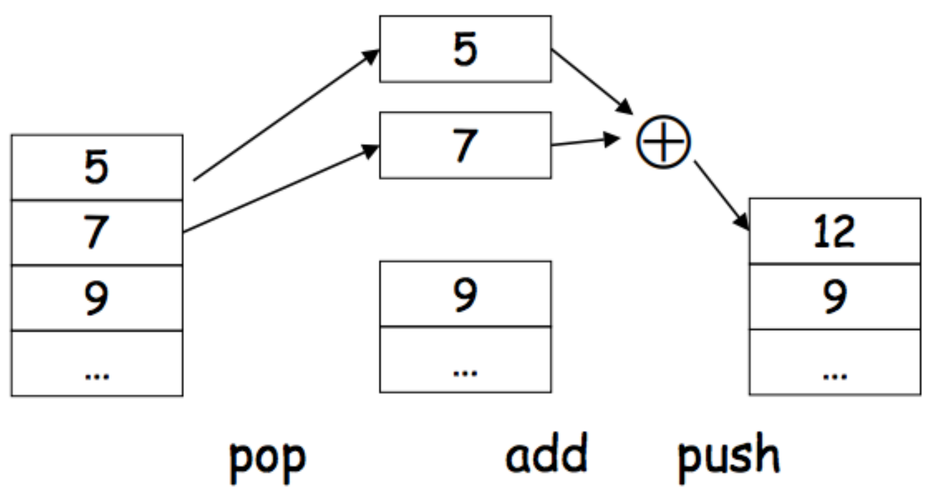
```
LD R0, b          // R0 = b
ADD R0, R0, c     // R0 = R0 + c
ST a, R0          // a = R0
LD R0, a          // R0 = a
ADD R0, R0, e     // R0 = R0 + e
ST d, R0          // d = R0
```

Register Allocation & Evaluation Order

- **Register allocation**: a key problem in code generation is deciding what values to hold in what registers[寄存器分配]
  - Registers are the fastest storage unit but are of limited numbers
    - Values not held in registers need to reside in memory
    - Insts involving register operands are much shorter and faster
  - Finding an optimal assignment of registers to variables is NP-hard
- **Evaluation order**: the order in which computations are performed can affect the efficiency of the target code [执行顺序]
  - Some computation orders require fewer registers to hold intermediate results than others
  - However, picking a best order in the general case is NP-hard

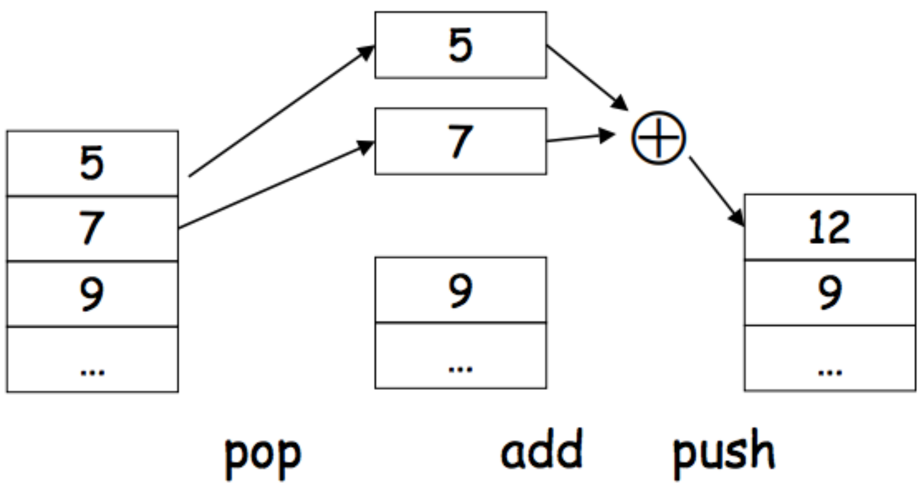
Stack Machine [栈式计算机]

- A simple evaluation model[一个简单模型]
  - No variables or registers
  - A stack of values for intermediate results
- Each instruction[指令任务]
  - Takes its operands from the top of the stack [栈顶取操作数]
  - Removes those operands from the stack [从栈中移除操作数]
  - Computes the required operation on them [计算]
  - Pushes the result on the stack [将计算结果入栈]



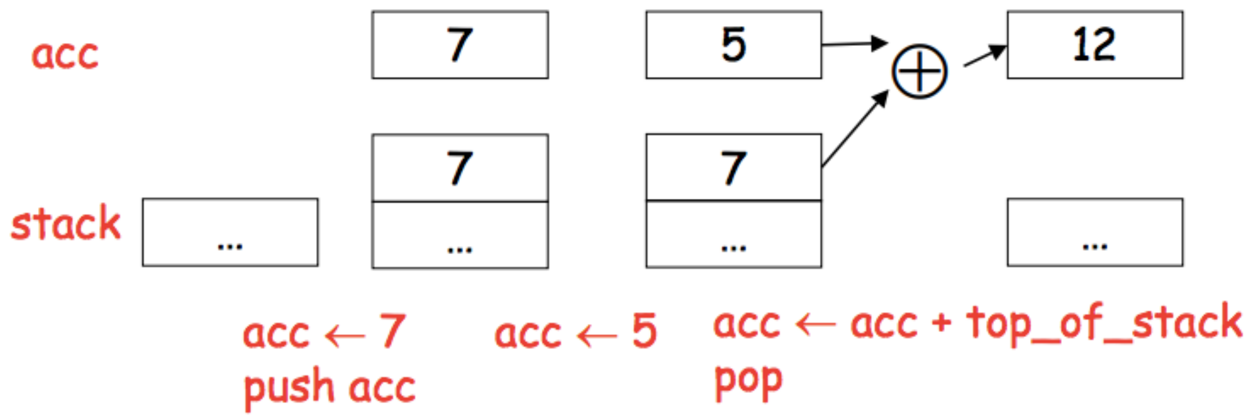
Example

- Consider two instructions
  - *push i* - place the integer *i* on top of the stack
  - *add* - pop two elements, add them and put the result back on the stack
- A program to compute *7 + 5*
  - *push 7*
  - *push 5*
  - *add*



Optimize the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- **Idea**: keep the top of the stack in a register (called *accumulator*) [使用寄存器]
  - Register accesses are much faster
- The “add” instruction is now
  - *acc ← acc + top_of_stack*
  - Only one memory operation



From Stack Machine to MIPS

- The compiler generates code for a stack machine with accumulator
  - The accumulator is kept in MIPS register *$t0*
  - Stack machine instructions are implemented using MIPS instructions and registers
  - We want to run the resulting code on the MIPS processor (or simulator)
- The stack is kept in memory
  - The stack grows towards lower addresses (standard convention)
  - The address of next stack location is kept in MIPS register *$sp*
    - The top of the stack is at address *$sp + 4*
  - A block of stack space, called **stack frame**, is allocated for each function call
    - A stack frame consists of the memory between $fp which points to the base of the current stack frame, and the $sp
    - Before func returns, it must pop its stack frame, and restore the stack

MIPS Architecture

- Load/store architecture
  - Only load and store instructions can access memory
  - All other instructions access only registers
    - E.g., all arithmetic and logical operations involve only registers (or constants that are stored as part of the instructions)
- Word size is 32 bits, all instructions are encoded in a single 32-bit word format
  - Arithmetic
    - e.g., add des, src1, src2 // des = src1 + src2
  - Comparison
    - e.g., sge des, src1, src2 // des $\leftarrow$ 1 if src1 $\geq$ src2, 0 ow
  - Branch/jump
    - e.g., bge src1, src2, lab // branch to lab if src1 $\geq$ src2
  - Load, store, and data movement
    - E.g., lw des, addr // load the word at addr into des
    - E.g., move des, src1 // copy the contents of src1 to des

MIPS Architecture (cont.)

- 32 registers
  - 31 of these are general-purpose that can be used in any of the instructions
  - The last one (*zero*), is to contain the number zero at all times
- While general-purpose, there are guidelines specifying how each of the registers should be used
  - $0 is always zero, $a0,...,$a4 are for arguments
  - $sp saves stack pointer, $fp saves frame pointer

Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 ⋯ 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 ⋯ 9.
s0 - s7	16 - 23	Saved Registers 0 ⋯ 7.
k0 - k1	26 - 27	Kernel Registers 0 ⋯ 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

Example MIPS Instructions

- *la reg1 addr*
  - Load address into reg1
- *li reg imm*
  - reg ← imm
- *lw reg1 offset(reg2)*
  - Load 32-bit word from address reg2 + offset into reg1
- *sw reg1 offset(reg2)*
  - Store 32-bit word in reg1 at address reg2 + offset
- *add reg1 reg2 reg3*
  - reg1 ← reg2 + reg3
- *move reg1 reg2*
  - reg1 <- reg2
- *sge reg1 reg2 reg3*
  - reg1 ← (reg2 >= reg3)

Example MIPS Assembly

- The stack-machine code for *7 + 5* in MIPS:

Stack-machine	**MIPS**	**Comment**
acc <- 7	li $t0 7	Load constant 7 into $t0
push acc	addi $sp $sp -4 sw $t0 0($sp)	Decrement sp to make space Copy the value to stack
acc <- 5	li $t0 5	Load constant 5 into $t0
acc <- acc + top_of_stack	lw $t1 4($sp) add $t0 $t0 $t1	Load value from $sp+4 into $t1 Add $t0+$t1 = 5 + 7
pop	add $sp $sp 4	Pop constant 7 off stack

中山大學 SUN YAT-SEN UNIVERSITY

A Small Language

- A language with integers and integer operations

```
P → D; P | D
D → def id(ARGS) = E;
ARGS → id, ARGS | id
E → int | id | if E1 = E2 then E3 else E4
      | E1 + E2 | E1 – E2 | id(E1,…,En)
```

- Example: program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                 if x = 2 then 1 else
                   fib(x - 1) + fib(x – 2)
```

Code Generation Considerations

- We used to store values in unlimited temporary variables, but registers are limited --> must reuse registers[重复使用寄存器]
- Must save/restore registers when reusing them [保存-恢复]
  - E.g. suppose you store results of expressions in $t0
  - When generating *E -> E₁ + E₂*,
    - E_1 will first store result into $t0
    - E_2 will next store result into $t0, overwriting E_1's result
    - Must save $t0 somewhere before generating E_2
- Registers are saved on and restored from the stack

  Note: $sp - stack pointer register, pointing to the top of stack
  - Saving a register $t0 on the stack:

    ```
    addiu $sp, $sp, -4     # Allocate (push) a word on the stack
    sw $t0, 0($sp)         # Store $t0 on the top of the stack
    ```
  - Restoring a value from stack to register $t0:

    ```
    lw $t0, 0($sp)         # Load word from top of stack to $t0
    addiu $sp, $sp, 4      # Free (pop) word from stack
    ```

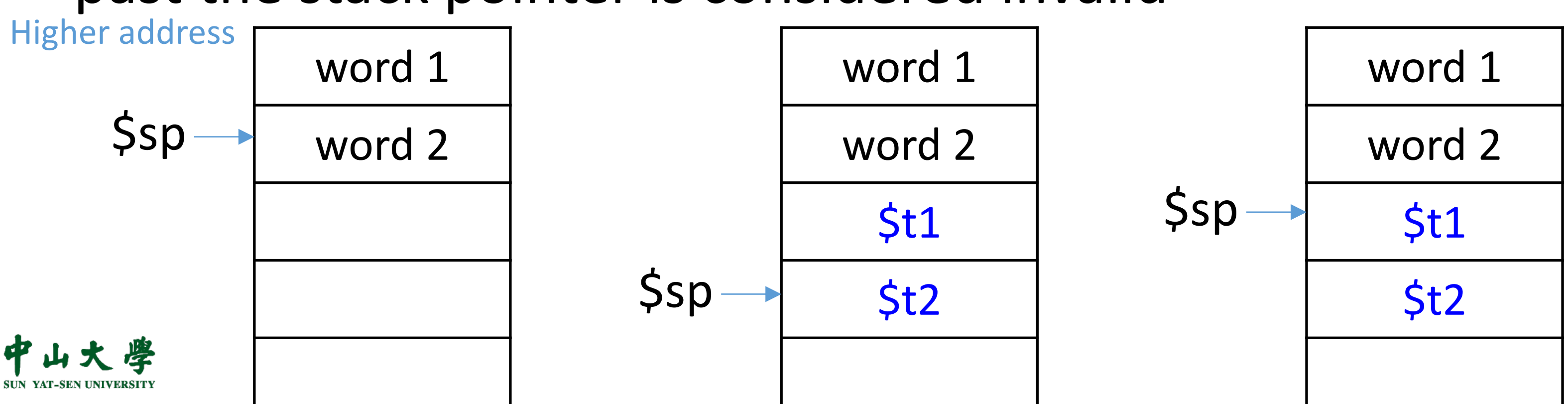
Stack Operations [栈操作]

- To **push** elements onto the stack
  - To move stack pointer $sp down to make room for the new data
  - Store the elements into the stack
- For example, to push registers $t1 and $t2 onto stack

```
sub $sp, $sp, 8
sw $t1, 4($sp)
sw $t2, 0($sp)
```

```
sw $t1, -4($sp)
sw $t2, -8($sp)
sub $sp, $sp, 8
```

- **Pop** elements simply by adjusting the $sp upwards
  - Note that the popped data is still present in memory, but data past the stack pointer is considered invalid



SUN YAT-SEN UNIVERSITY 中山大學

Code Generation Strategy

- For each expression *e* we generate MIPS code that:
  - Computes the value of *e* into *$t0*
  - Preserves *$sp* and the contents of the stack
- We define a code generation function *cgen(e)*
  - Its result is the code generated for *e*

- Code generation for constants
  - The code to evaluate a constant simply copies it into the register: *cgen(i) = li $t0 i*
    - Note that this also preserves the stack, as required

Code Generation for ALU

- Default

```
cgen(e1 + e2):
        # stores result in $t0
        cgen(e1)
        # pushes $t0 on stack
        addiu $sp $sp -4
        sw $t0 0($sp)
        # overwrites result in $t0
        cgen(e2)
        # pops value of e1 to $t1
        lw $t1 4($sp)
        addiu $sp $sp 4
        # performs addition
        add $t0 $t1 $t0
```

```
cgen(e1 + e2):
        # stores result in $t0
        cgen(e1)
        # copy result of $t0 to $t1
        move $t1 $t0
        # stores result in $t0
        cgen(e2)
        # performs addition
        add $t0 $t1 $t0
```

- Possible optimization: put the result of *e1* directly in register *$t1*?

Code Generation for Conditional

- We need flow control instructions
- New instruction: *beq reg1 reg2 label*
  - Branch to label if *reg1 == reg2*
- New instruction: *b label*
  - Unconditional jump to *label*

```
cgen(if e1 == e2 then e3 else e4):
        cgen(e1)
        # pushes $t0 on stack
        addiu $sp $sp -4
        sw $t0 0($sp)
        # overwrites $t0
        cgen(e2)
        # pops value of e1 to $t1
        lw $t1 4($sp)
        addiu $sp $sp 4
        # performs comparison
        beq $t0 $t1 true_branch
    false_branch:
        cgen(e4)
        b end_if
    true_branch:
        cgen(e3)
    end_if:
```

Caller/Callee Conventions

- Important registers should be saved across function calls
  - Otherwise, values might be overwritten
- But, who should take the responsibility?
  - The <u>caller</u> knows which registers are important to it and should be saved
  - The <u>callee</u> knows exactly which registers it will use and potentially overwrite
  - However, in the typical "block box" programming, caller and callee don't know anything about each other's implementation
- Potential solutions
  - **Sol1**: <u>caller</u> to save any important registers that it needs before calling a func, and to restore them after (but not all will be overwritten)
  - **Sol2**: <u>callee</u> saves and restores any registers it might overwrite (but not all are important to caller)

Caller/Callee Conventions (cont.)

- Caller and callee should cooperate

- Caller: save and restore any of the following caller-saved registers that it cares about

  \$t0-\$t9 \$a0-\$a3 \$v0-\$v1

  - The callee may freely modify these registers, under the assumption that the caller already saved them

- Callee: save and restore any of the following callee-saved registers that it uses

  \$s0-\$s7 \$ra

  - The caller may assume these registers are not changed by the callee

Detailed Calling Steps

- The **caller** sets up for the call via these steps[调用者]
  - 1) Make space on stack for and save any caller-saved registers
  - 2) Pass arguments by pushing them on the stack, one by one, right to left
  - 3) Execute a jump to the function (saves the next inst in $ra)

- The **callee** takes over and does the following[被调用者]
  - 4) Make space on stack for and save values of $fp and $ra
  - 5) Configure frame pointer by setting $fp to base of frame
  - 6) Allocate space for stack frame (total space required for all local and temporary variables)
  - 7) Execute function body, code can access params at positive offset from $fp, locals/temps at negative offsets from $fp

Detailed Calling Steps (cont.)

- When ready to exit, the **callee** does following[调用退出]
  - 8) Assign the return value (if any) to $v0
  - 9) Pop stack frame off the stack (locals/temps/saved regs)
  - 10) Restore the value of $fp and $ra
  - 11) Jump to the address saved in $ra

- When control returns to the **caller**, it cleans up from the call with the steps[调用返回]
  - 12) Pop the parameters from the stack
  - 13) Restore value of any caller-saved registers, pops spill space from stack

Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New instruction: *jal label*
  - Jump to label, after saving address of next instruction in $ra

```
cgen(f(e1, …, en)):
        # pushes arguments (reverse order)
        cgen(en)
        addiu $sp $sp -4
        sw $a0 0($sp)
        …
        cgen(e1)
        addiu $sp $sp -4
        sw $a0 0($sp)
        # saves FP
        addiu $sp $sp -4
        sw $fp 0($sp)
        # pushes return address
        addiu $sp, $sp, -4
        sw $ra, 0($sp)
        # begins new AR in stack
        move $fp, $sp
        # jumps to func entry (update $ra)
        jal f_entry
```

Code Generation for Function Definition

- New instruction: *jr reg*
  - Jump to address in register reg

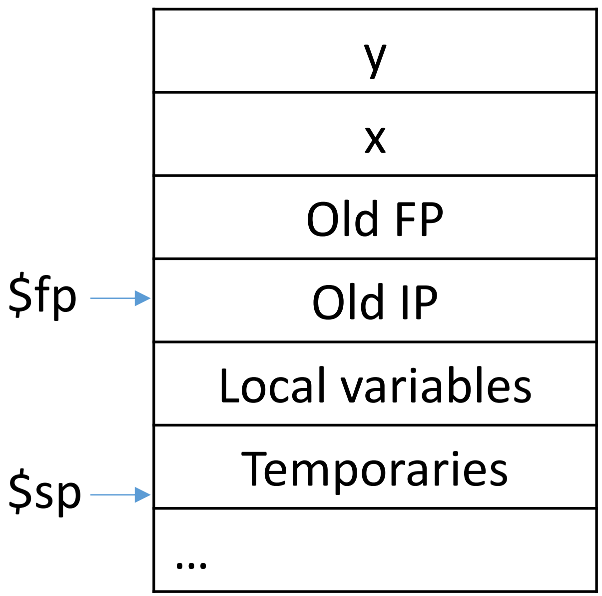
```
cgen(def f(x1,…,xn) = e):
f_entry: cgen(e)
          # removes AR from stack
          move $sp $fp
          # pops return address
          sw $ra 0($sp)
          addiu $sp $sp 4
          # pops old FP
          lw $fp 0($sp)
          addiu $sp $sp 4
          # jumps to return address
          jr $ra
```

Code Generation for Variables

- The "variables" of a function are just its 'parameters'
  - They are all in the AR
  - Pushed by the caller
- **Problem**: because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp
  - Thus, access to locations in the stack frame cannot use $sp-relative addressing
- **Solution**: use the frame pointer $fp instead
  - Always points to the return address on the stack
  - Since it does not move, it can be used to find the variables

Example

- Local variables are referenced from an offset from $fp
  - $fp is pointing to old $ip (return address)
- For a function *def f(x,y) = e* the activation and frame pointer are set up as follows:



x: +8($fp)
y: +12($fp)
First local variable: -4($fp)

The parameters are pushed right to left by the caller
The locals are pushed left to right by the callee

Example

```
double fun1(int p1, double p2, int p3) {
    int i, j;
    res = fun2(p1*p2, j);
    return res;
}
```

```
double fun2(double ar, int ib) {
    int i, r1;
    double res;
    …
    return res;
}
```