计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第24讲：目标代码生成(2)

张献伟

xianweiz.github.io

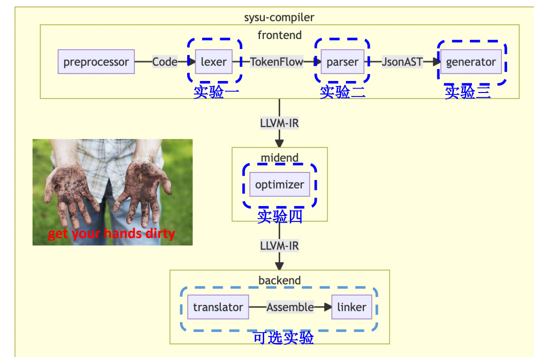
DCS290, 6/20/2024

理论 · Final Exam

- 考试时间地点
  - 7.4/周四，14:30 – 16:30
  - 东B101
- 关于试卷
  - 中文（专业术语标注英文）
  - A、B卷，学院指定
- 成绩计算
  - 期末：60%
  - 平时：40%
    - 课堂：15%
    - 作业：25%
- 题型及分值
  - 一、判断题（10分）
    - 10小题，每小题1分
  - 二、填空题（10分）
    - 几个小题，10个空，每空1分
  - 三、简答题（15 - 20分）
    - 3-4小题，每小题5分
  - 四、应用题（60 - 65分）
    - 3小题，每小题10 - 25分
- 主要内容
  - 词法分析
  - 语法分析
  - 语义分析
  - 代码生成及优化

实验 · Projects

- 编译器构造
  - 课堂参与（10%）- 签到、练习等
  - Project 1（20%）- Lexical Analysis
  - Project 2（20%）- Syntax/Semantic Analysis
  - Project 3（20%）- IR Generation
  - Project 4（30%）- Code Optimization

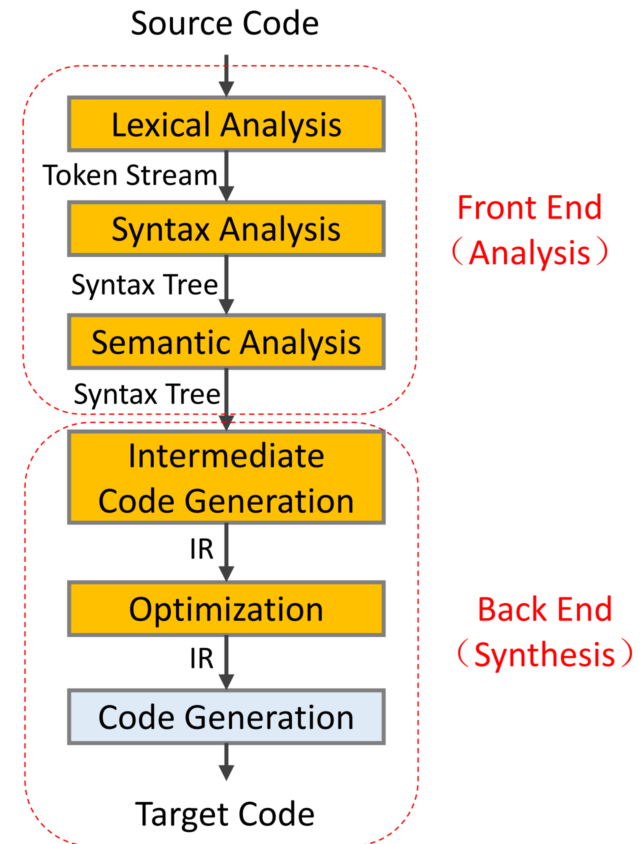


101 计划

- 奖项奖励
  - 代码贡献：为实验仓库提交文档或代码，作出重要贡献，且拉取请求被成功合并
  - 性能实现：给实验四成绩最好的几个人
  - 建言献策：实验报告反馈意见中肯、提出了有价值的issue
  - 乐于助人：经常帮助其他同学
  - 。。。。。。

Target Code Generation[目标代码生成]

- What we have now
  - Optimized IR of the source program
    - And, symbol table
- Target code
  - Binary (machine) code
  - Assembly code
- Goals of target code generation
  - Correctness: the target program must preserve the semantic meaning of the source program
  - High-quality: the target program must make effective use of the available resources of the target machine
  - Fast: the code generator itself must runs efficiently



Translating IR to Machine Code[翻译]

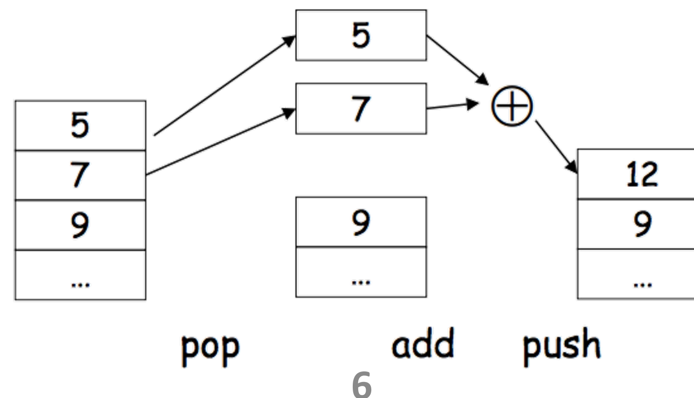
- Machine code generation is machine ISA dependent*
  - Complex instruction set computer (CISC): x86
  - Reduced instruction set computer (RISC): ARM, MIPS, RISC-V

- 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？)

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]



中山大学 SUN YAT-SEN UNIVERSITY

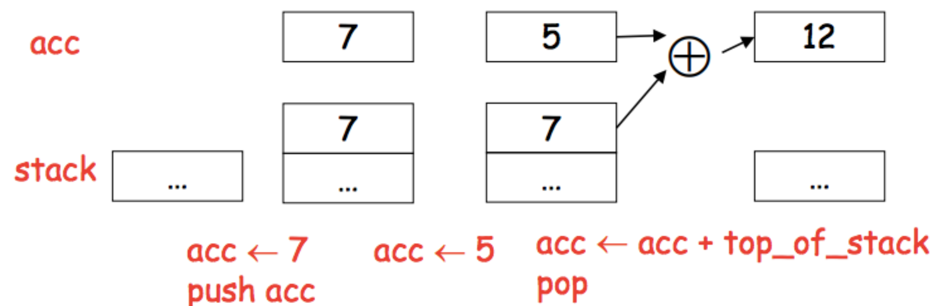
NSCC GZ

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

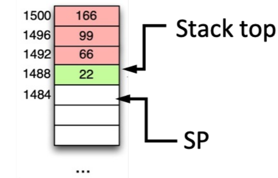
push 7
push 5
add

- **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 RISC-V

- The compiler generates code for a stack machine with accumulator
  - The accumulator is kept in RISC-V register *a0*
  - Stack machine insts are implemented using RISC-V insts and registers
  - We want to run the resulting code on the RISC-V 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 a register *sp*
    - The top of the stack is now 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

The RISC-V Architecture[架构]

- 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)
- Each instruction is 32 bits long in memory
- Byte addressable memories with 64-bit addresses
- Only immediate and displacement addressing modes (12-bit field)
  - Absolute: via the *lui* instruction (i.e., x0-offset)
  - PC-relative: via *auipc*, *jal* and *br** instructions
  - Register offset: via *jalr*, *addi* and all memory instructions

The RISC-V Registers[架构]

- 32, 64-bit general purpose registers (GPRs) + PC
  - called x0, … , x31 (*x0* is hardwired to the value 0)
    - *x0* can be used as target reg for any inst whose result is to be discarded
- 32, 64-bit floating point registers - FPRs (each can hold a 32-bit single precision or a 64-bit double precision value)
  - called f0, f1, … , f31
- A few special purpose registers (example: floating point status)

Numbers hardware understands

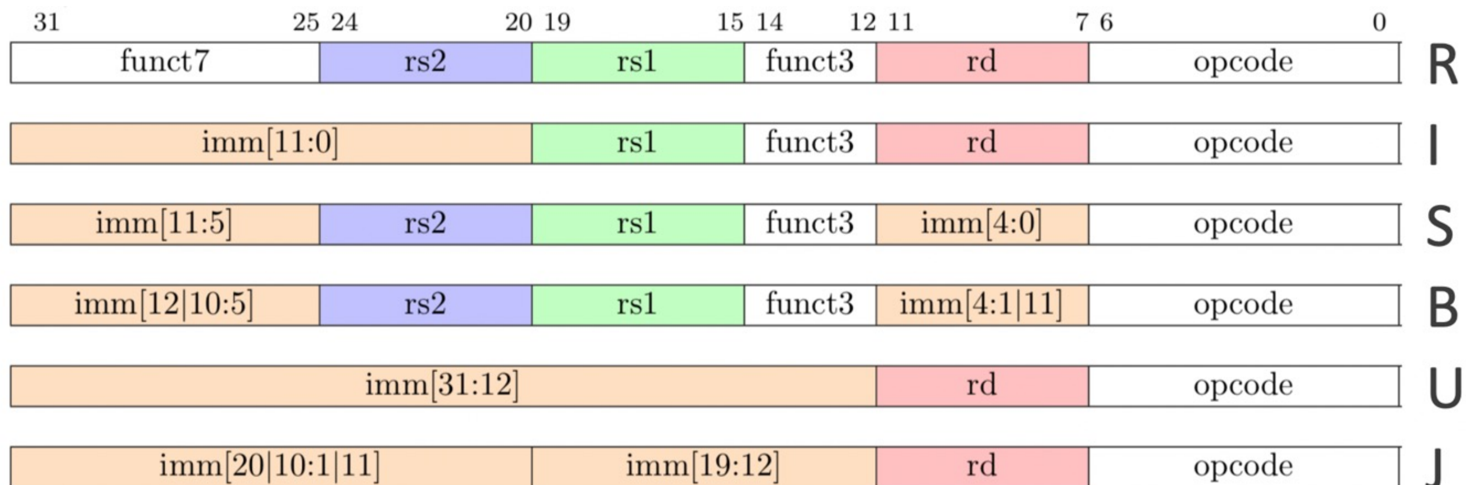
Human-friendly symbolic names in assembly code

Register name	Symbolic name	Description
32 integer registers		
x0	Zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporary
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function argument / return value
x12–17	a2–7	Function argument
x18–27	s2–11	Saved register
x28–31	t3–6	Temporary

Name	ABI Mnemonic	Meaning
f0 -f7	ft0 – ft7	Temporary Registers
f8 – f9	fs0 – fs1	Saved Registers
f10 – f11	fa0 – fa1	Argument and Return Registers
f12 – f17	fa2 – fa7	Argument Registers
f18 – f27	fs2 – fs11	Saved Registers
f28 – f31	ft8 – ft11	Temporary Registers

RISC-V Instructions[指令]

- All RISC-V instructions are 32 bits long, have 6 formats
  - R-type: instructions using **r**egister-register
  - I-type: instructions with **i**mmediates, loads
  - S-type: **s**tore instructions
  - B-type: **b**ranch instructions (beq, bge)
  - U-type: instructions with **u**pper immediates
  - J-type: **j**ump instructions (jal)



https://riscv.org/wp-content/uploads/2018/05/13.15-13-50-Talk-riscv-base-isa-20180507.pdf

Example RISC-V Instructions

- *la reg1 addr* **Pseudo**
  - Load address into reg1
- *li reg imm* **Pseudo**
  - reg ← imm
- *lw reg1 offset(reg2)* **Pseudo**
  - Load 32-bit word from address reg2 + offset into reg1
- *sw reg1 offset(reg2)* **Pseudo**
  - Store 32-bit word in reg1 at address reg2 + offset
- *add reg1 reg2 reg3*
  - reg1 ← reg2 + reg3
- *mv reg1 reg2* **Pseudo**
  - reg1 <- reg2
- *slt rd rs1 rs2*
  - rd ← (rs1 < rs2) ? 1 : 0

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
li rd, imm	addi rd, x0, imm #reg=imm+0	
mv rd, rs	addi rd, rs, 0	

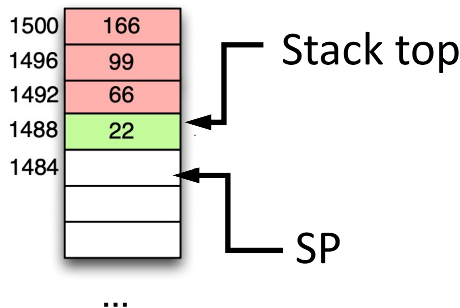
Pseudo-instructions: shorthand syntax for common assembly idioms

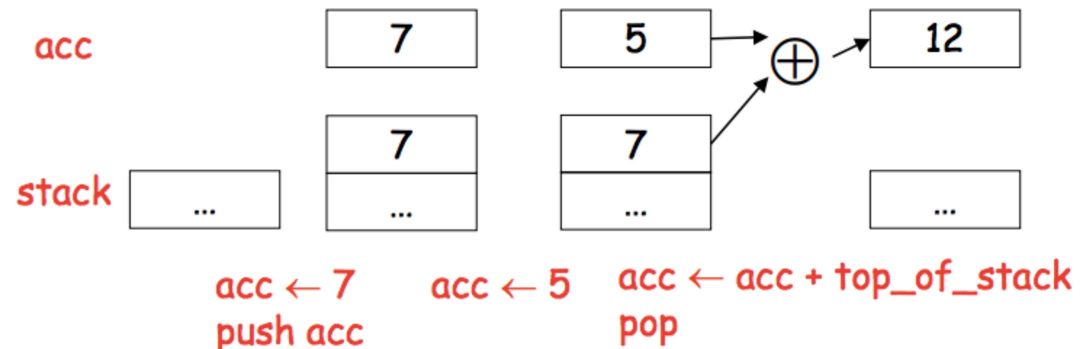
https://cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf

Example RISC-V Assembly

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

Stack-machine	RISC-V	Comment
acc <- 7	li a0 7	Load constant 7 into *a0*
push acc	sw a0 0(sp) addi sp sp -4	Copy the value to stack Decrement *sp* to make space
acc <- 5	li a0 5	Load constant 5 into *a0*
acc <- acc + top_of_stack	lw t1 4(sp) add a0 a0 t1	Load value from *sp+4* into *t1* Add *a0*+*t1* = 5 + 7
pop	add sp sp 4	Pop constant 7 off stack





A Small Language

- A language with integers and integer operations

$$P \rightarrow D; P \mid D$$
$$D \rightarrow \text{def id}(ARGS) = E;$$
$$ARGS \rightarrow \text{id}, ARGS \mid \text{id}$$
$$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$
$$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

- Example: program for computing the Fibonacci numbers:

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

A Small Language (cont.)

```c
1  #include<stdio.h>
2
3  typedef long long LL;
4  LL n, i;
5
6  LL fibo(LL n) {
7    if (n == 0)
8      return 0;
9    else if (n == 1)
10     return 1;
11   else
12     return fibo(n - 1) + fibo(n - 2);
13 }
14
15 int main() {
16   n = 5;
17
18   printf("The fibonacci series is :\n");
19   for (i = 1; i <= n; i++) {
20     printf("%lld ", fibo(i));
21   }
22 }
```

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

```asm
fibo:
        # Argument n is in a0
        beqz a0, is_zero        # n = 0?
        addi t0, a0, -1         # Hack: If a0 == 1 then t0 == 0
        beqz t0, is_one         # n = 1?

        # n > 1, do this the hard way

        addi sp, sp, -16        # Make room for two 64-Bit words on stack
        sd a0, 0(sp)            # Save original n
        sd ra, 8(sp)            # Save return address

        addi a0, a0, -1         # Now n-1 in a0
        jal fibo                # Calculate fibo(n-1)

        ld t0, 0(sp)            # Get original n from stack
        sd a0, 0(sp)            # Save fibo(n-1) to stack in same place
        addi a0, t0, -2         # Now n-2 in a0
        jal fibo                # Calculate fibo(n-2)

        ld t0, 0(sp)            # Get result of fibo(n-1) from stack
        add a0, a0, t0          # add fibo(n-1) and fibo(n-2)

        ld ra, 8(sp)            # Get return address from stack
        addi sp, sp, 16         # clean up stack

        # Fall through

is_zero:
is_one:
        ret
```

https://github.com/scotws/RISC-V-Fibonacci/blob/master/fibonacci-naive.s

Code Generation Considerations[考虑]

- 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[寄存器保存-恢复]
  - E.g. suppose you store results of expressions in *a0*
  - When generating $E \text{ -> } E_1 + E_2$,
    - E_1 will first store result into *a0*
    - E_2 will next store result into *a0*, overwriting E_1's result
    - Must save *a0* somewhere before generating E_2

Code Gen Considerations (cont.)

- Registers are saved on and restored from the stack

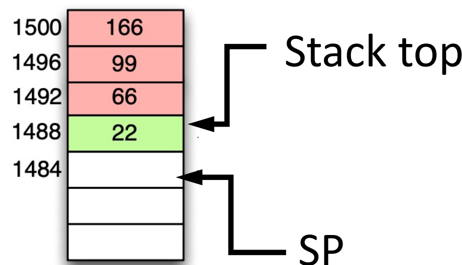
Note: *sp* - stack pointer register, pointing to the top of stack

- Saving a register *a0* on the stack:

```
addiu sp, sp, -4      # allocate (push) a word on the stack
sw a0, 4(sp)          # store a0 on the top of the stack
```

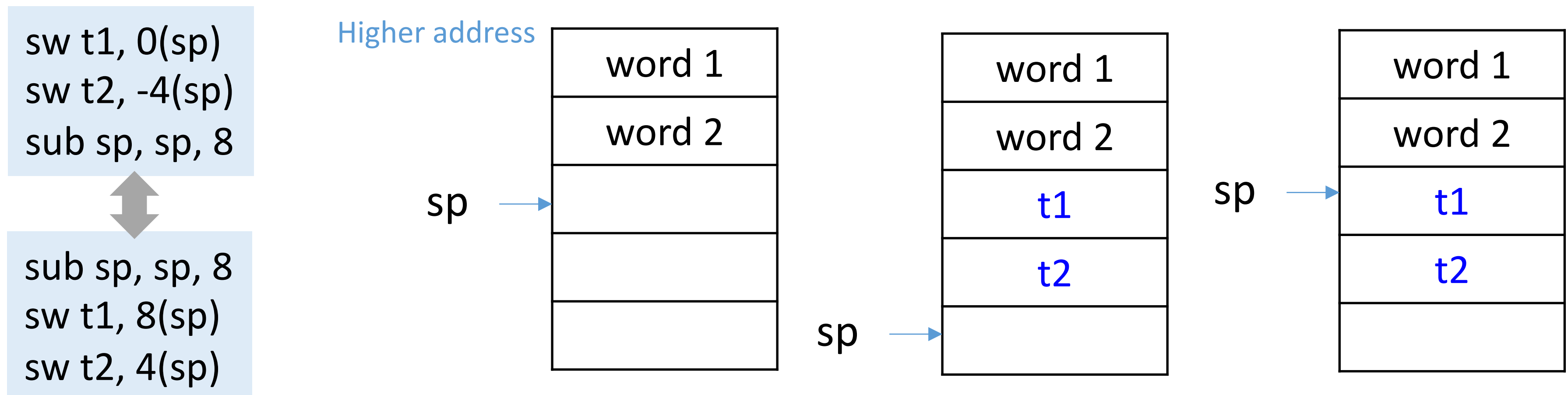
- Restoring a value from stack to register *a0*:

```
lw a0, 4(sp)          # load word from top of stack to a0
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
- **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 / undefined



Code Generation Strategy

- For each expression *e* we generate RISC-V code that:
  - Computes the value of *e* into *a0* (i.e., the accumulator)
  - 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 a0 i*
    - Note that this also preserves the stack, as required