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

Compilation Principle
编 译 原 理

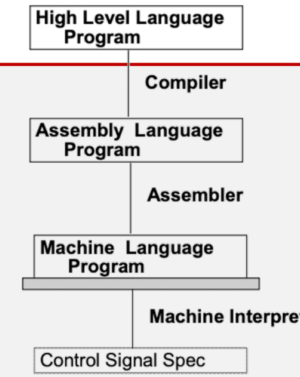
第24讲：目标代码生成(2)

张献伟

xianweiz.github.io

DCS290, 6/6/2023

Review Questions



- Q1: what is global constant propagation?

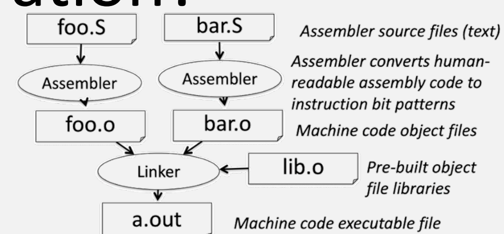
  Substituting values of known constants at compile time, across basic blocks.

- Q2: usage of data flow analysis?

  To determine the property at a given point through value calculation.

- Q3: input and output of target code generation?

  Input: optimized IR; output: machine code



- Q4: assembler vs. linker?

  Assembler: assembly --> machine code, .o file, have address holes

  Linker: machine code --> machine code, executable file, fill in holes

- Q5: primary tasks of target code generation?

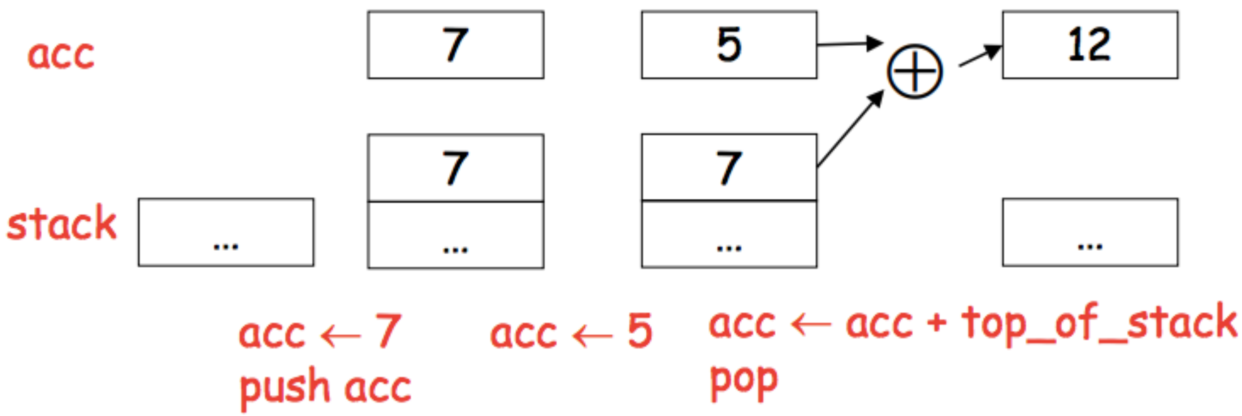
  Instruction selection, register allocation and assignment, instruction ordering

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

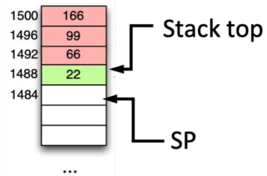
push 7
push 5
add

- **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 instructions are implemented using RISC-V instructions 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

https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

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

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

Human-friendly symbolic names in assembly code

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)

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R
imm[11:0]		rs1	funct3	rd	opcode	I
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B
imm[31:12]				rd	opcode	U
imm[20|10:1|11]		imm[19:12]		rd	opcode	J

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

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

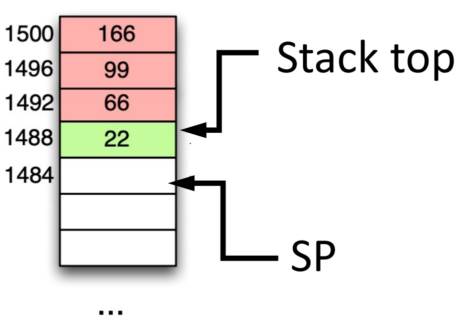
Pseudo-instructions: shorthand syntax for common assembly idioms

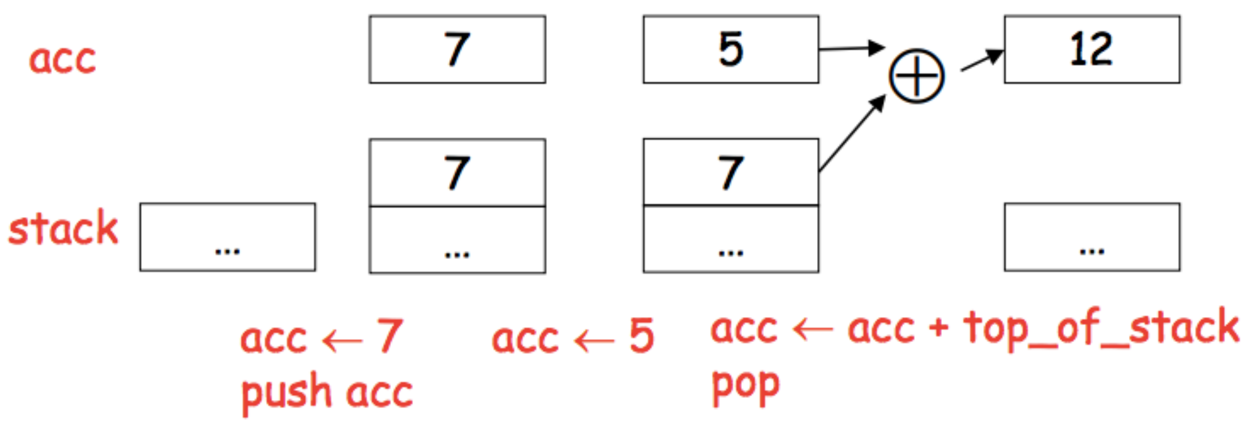
https://cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf

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

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

- 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)
```


https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

A Small Language (cont.)

```c
#include<stdio.h>

typedef long long LL;
LL n, i;

LL fibo(LL n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibo(n - 1) + fibo(n - 2);
}

int main() {
  n = 5;

  printf("The fibonacci series is :\n");
  for (i = 1; i <= n; i++) {
    printf("%lld ", fibo(i));
  }
}
```

```
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]

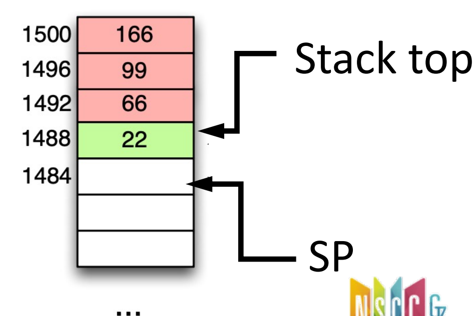
- We used to store values in unlimited temporary variables, but registers are limited --> must reuse registers[重复使用寄存器]
- Must save/restore registers when reusing them[保存-恢复]
  - E.g. suppose you store results of expressions in *a0*
  - When generating $E \text{ -> } E_1 + E_2$,
    - E_1 will first 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
- 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
    ```



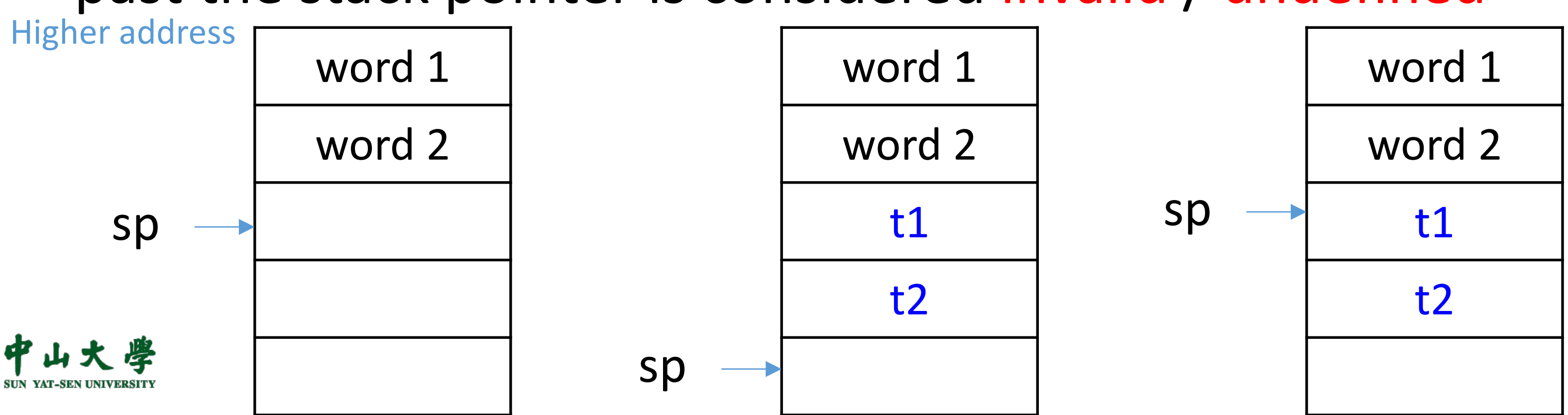
Stack Operations[栈操作]

- 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

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

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

- **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

https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

Code Generation for ALU

- Default

```
cgen(e1 + e2):
        # stores result in a0
        cgen(e1)
        # pushes a0 on stack
        addiu sp sp -4
        sw a0 4(sp)
        # overwrites result in a0
        cgen(e2)
        # pops value of e1 to t1
        lw t1 4(sp)
        addiu sp sp 4
        # performs addition
        add a0 t1 a0
```

→

```
cgen(e1 + e2):
        # stores result in a0
        cgen(e1)
        # copy result of a0 to t1
        mv t1 a0
        # stores result in a0
        cgen(e2)
        # performs addition
        add a0 t1 a0
```

- Possible optimization: put the result of *e1* directly in register *t1*? What if 3 + (7 + 5)?

SUN YAT-SEN UNIVERSITY 中山大學

https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

NSCC GZ

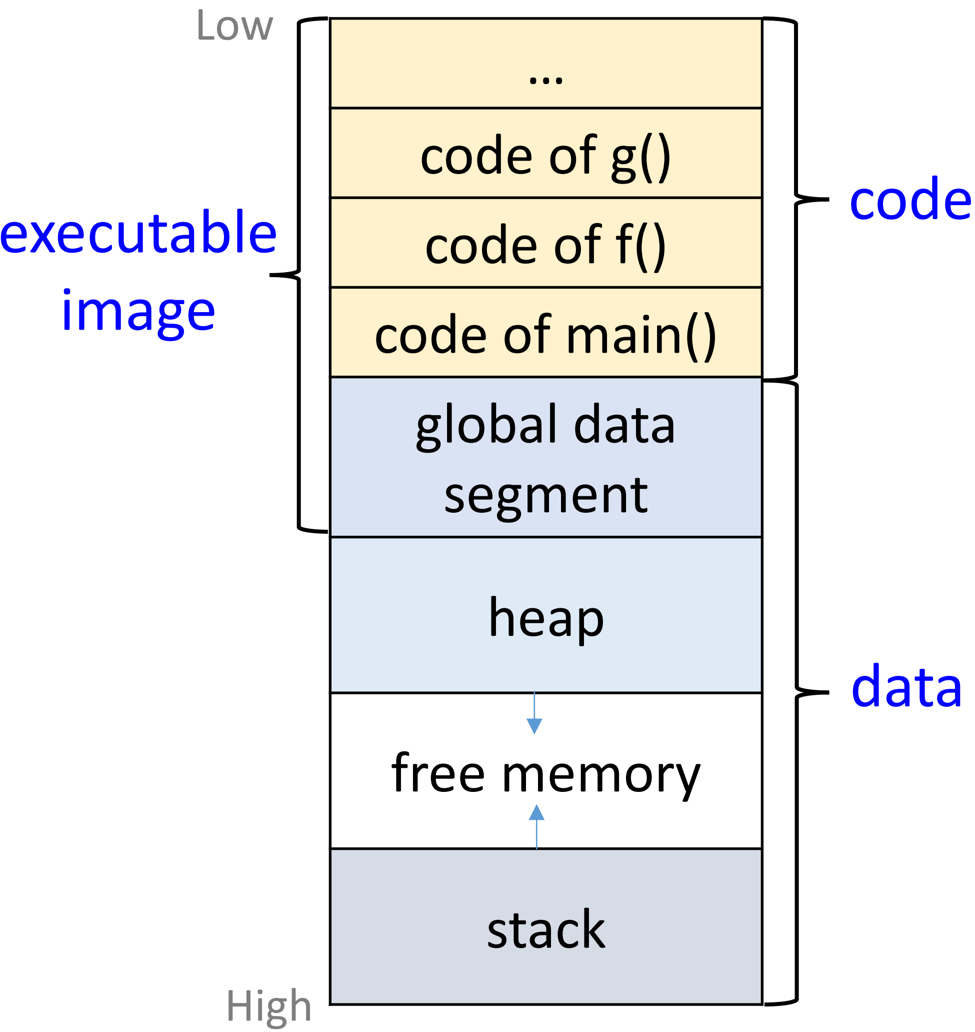
Code Generation for Conditional

- We need flow control instructions
- New instruction: *beq reg1 reg2 label*
  - Branch to label if *reg1 == reg2*
    - Ow, does nothing and move on to the next command
- New instruction: *b label*
  - Unconditional jump to *label*

```
cgen(if e1 == e2 then e3 else e4):
        cgen(e1)
        # pushes a0 on stack
        addiu sp sp -4
        sw a0 4(sp)
        # overwrites a0
        cgen(e2)
        # pops value of e1 to t1
        lw t1 4(sp)
        addiu sp sp 4
         # performs comparison
        beq a0 t1 true_branch
    false_branch:
        cgen(e4)
        b end_if
    true_branch:
        cgen(e3)
    end_if:
```


https://www.d.umn.edu/~rmaclin/cs5641/Notes/L19_CodeGenerationI.pdf

Example Memory Layout



- Code
  - the size of the generated target code is fixed at compile time
- Global/**static**
  - the size of some program data objects, e.g., global constants, are known at compile time
- **Stack**
  - store dynamic data structures
- **Heap**
  - manage long-lived data

Activation[活动]

- Compiler typically allocates memory in the unit of procedure[以过程调用为单位]
- Each execution of a procedure is called as its **activation**[活动]
  - An execution of a procedure starts at the beginning of the procedure body
  - When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called
- **Activation record** (AR)[活动记录] is used to manage the information needed by a single execution of a procedure
- **Stack** is to hold activation records that get generated during procedure calls

ARs in Stack Memory[在栈中管理]

- Manage ARs like a stack in memory[AR栈管理]
  - On function entry: AR instance allocated at top of stack
  - On function return: AR instance removed from top of stack
- Hardware support[硬件支持]
  - Stack pointer (SP) register[栈指针]
    - *SP* stores address of top of the stack
    - Allocation/de-allocation can be done by moving *SP*
  - Frame pointer (FP) register[帧指针]
    - *FP* stores base address of current frame
    - **Frame**: another word for activation record (AR)
    - Variable addresses translated to an offset from *FP*
      - Always points to the top of current AR as long as invocation is active
  - *FP* and *SP* together delineate the bounds of current AR

Contents of ARs

- Example layout of a function AR

Temporaries	临时变量
Local variables	局部变量
Saved Caller/Callee Register Values	保存的寄存器值
Saved Caller's Return Address (ra)	保存的调用者返回地址
Saved Caller's AR Frame Pointer (FP)	保存的调用者帧指针
Parameters	参数
Return Value	返回值

- Registers such as *FP* and *ra* overwritten by callee → Must be saved to/restored from AR on call/return
  - Caller's *ra*: where to execute next on function return (a.k.a. instruction pointer: instruction following function call)
  - Caller's *FP*: where *FP* should point to on function return
  - Saved Caller/Callee Registers: other registers (will discuss)

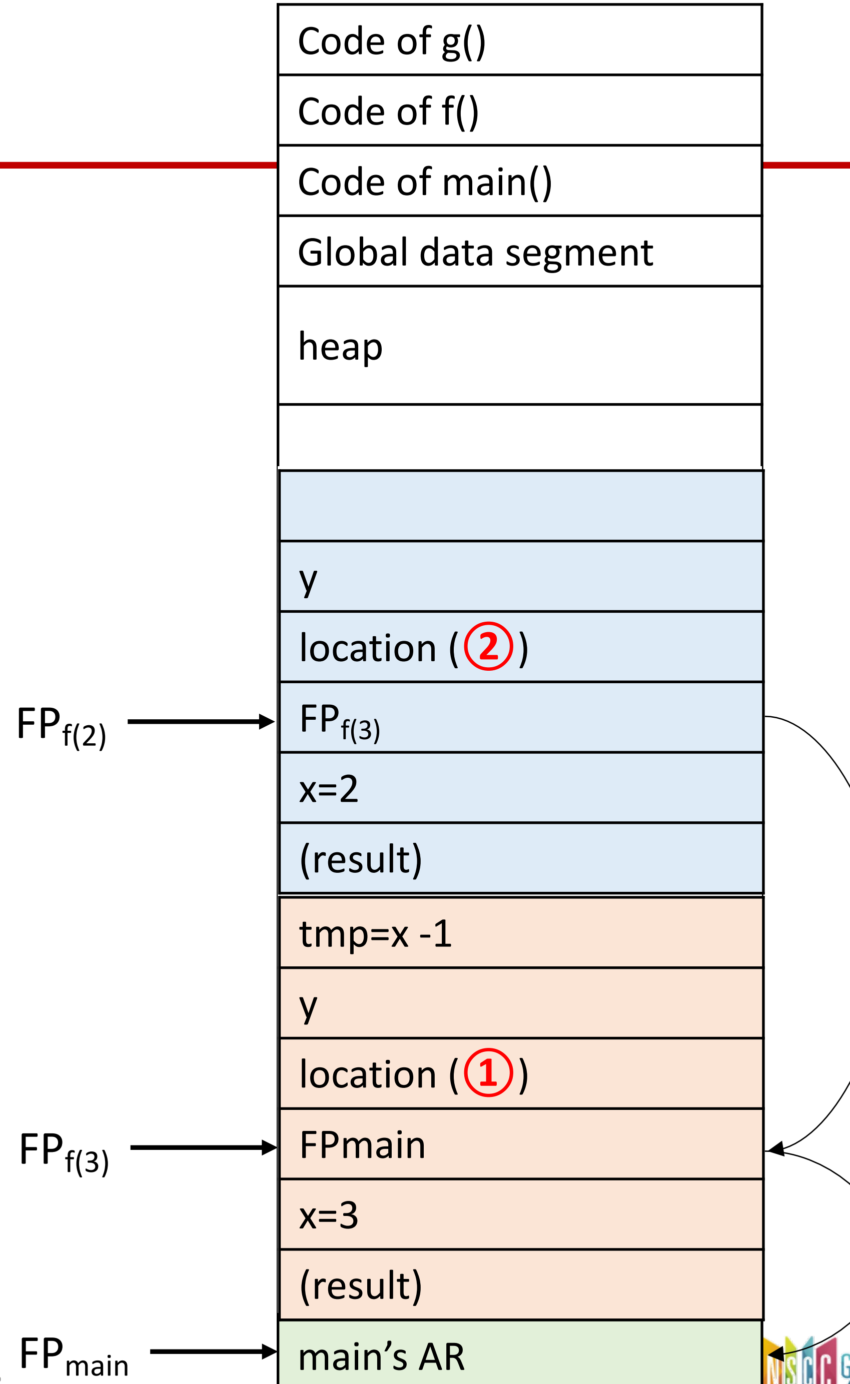
Example

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Return Address (ra)
Saved Caller's AR Frame Pointer (FP)
Parameters
Return Value

```
int g() {
  return 1;
}

int f(int x) {
  int y;
  if (x==2)
    y = 1;
  else
    y = x + f(x-1);
   ② ...
   return y;
}

int main() {
  f(3);
  ① ...
}
```



https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Caller/Callee Conventions[规范]

- Important registers should be saved across function calls
  - Otherwise, values might be overwritten

- But, who should take the responsibility?
  - The caller knows which registers are important to it and should be saved[调用者知道哪些重要]
  - The callee knows exactly which registers it will use and potentially overwrite[被调用者知道哪些会被覆写]
  - However, in the typical “block box” programming, caller and callee don’t know anything about each other’s implementation
- Potential solutions
  - **Sol1**: caller to save any important registers that it needs before calling a func, and to restore them after (but not all will be overwritten)[调用者保存任何重要寄存器，但并非所有都会覆写]
  - **Sol2**: callee saves and restores any registers it might overwrite (but not all are important to caller)[被调用者保存并恢复任意可能覆写，但并非所有都重要]

Caller/Callee Conventions (cont.)

- Caller and callee should cooperate
  - Caller: the function making the call
  - Callee: the function that is being called

- Callee-saved registers (**preserved registers**): the registers that a function promises to leave unchanged[预留寄存器]
  - The caller may assume these registers are not changed by the callee
- Caller-saved registers (**non-preserved registers**): the registers that a function does not promise to leave unchanged[非预留寄存器]
  - The callee may freely modify these registers, under the assumption that the caller already saved them

https://cs61c.org/sp23/projects/proj2/calling-convention/

RISC-V Calling Conventions

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

  t0-t6 a0-a7 ra

  a0-a7 for function arguments, a0-a1 for return values

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

  s0-s11 sp

  s0 is fp

a0 - a7 (x10 - x17): eight argument registers to pass parameters and two return values (a0-a1)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf#:~:text=In%20the%20standard%20RISC-V%20calling%20convention%2C%20the%20stack,must%20besaved%20by%20the%20caller%20if%20later%20used.

The Caller Perspective

Caller-saved: t0-t6 a0-a7 ra
Callee-saved: s0-s11 sp

- We we call a function, that function promises to not modify any of the preserved registers[调用者：这些预留寄存器不会被改动]
  - I.e., when the function returns, we can be sure that the preserved registers have not changed
    - The called function may modify across the calling, but finally restores
- However, that function is allowed to freely modify any of the non-preserved registers[调用者：这些非预留寄存器会被随意改动]
  - I.e., after calling a function and the function returns, every non-preserved register now contains garbage
    - Garbage refers to unknown values, even if the values in non-preserved remain unchanged across the function call (just assume changed)

```
addi s0, x0, 5    # s0 contains 5
jal ra, func.     # call a function
addi s0, s0, 0    # s0 still contains 5 here!
```

```
addi t0, x0, 5    # t0 contains 5
jal ra, func      # call a function
addi t0, t0, 0    # t0 contains garbage!
```

The Callee Perspective

Caller-saved: t0-t6 a0-a7 ra
Callee-saved: s0-s11 sp

- We we write a function, we are allowed to freely change any of the non-preserved registers
  - I.e., those non-preserved ones are supposed to be saved by the caller
- However, we must promise to not change any of the preserved ones
  - I.e., if to use the preserved registers during the function, we must save the values on the stack at the function start, then restore at the function end

```
# Prologue
addi sp, sp, -12 # decrement stack
sw ra, 4(sp) # store ra value on the stack
sw s0, 8(sp) # store s0 value on the stack
sw s1, 12(sp) # store s1 value on the stack

# do stuff in the function

# Epilogue
lw ra, 4(sp) # restore ra value from the stack
lw s0, 8(sp) # restore s0 value from the stack
lw s1, 12(sp) # restore s1 value from the stack
addi sp, sp, 12 # increment stack

# finish up any loose ends

ret # return from function
```

Detailed Calling Steps

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Return Address ($ra)
Saved Caller's AR Frame Pointer ($FP)
Parameters
Return Value

- 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[传参数]
  - 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.)

Temporaries
Local variables
Saved Caller/Callee Register Values
Saved Caller's Return Address ($ra)
Saved Caller's AR Frame Pointer ($FP)
Parameters
Return Value

- When ready to exit, the **callee** does following[调用退出]
  - 8) Assign the return value (if any) to a0[返回值]
  - 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