编译原理
6. Activation Record

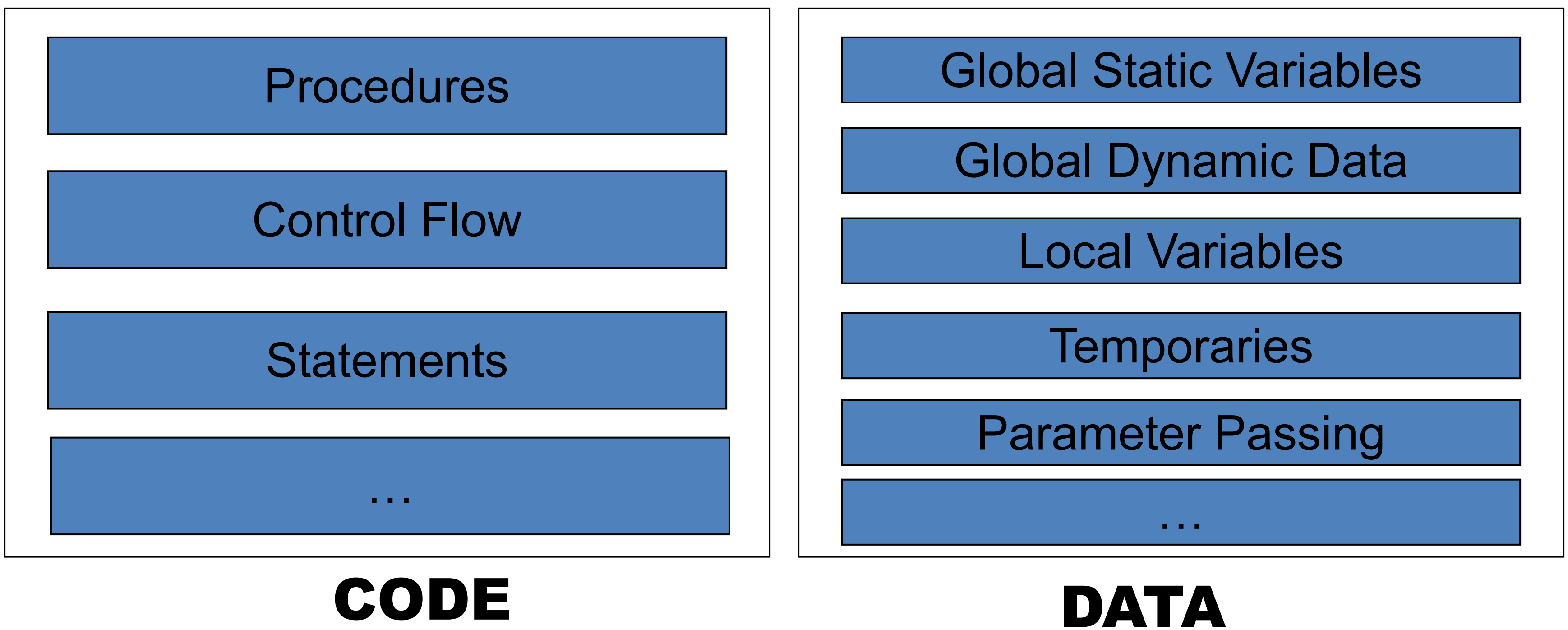
rainoftime.github.io
浙江大学
计算机科学与技术学院

课程内容

1. Introduction
2. Lexical Analysis
3. Parsing
4. Abstract Syntax
5. Semantic Analysis
6. **Activation Record**
7. Translating into Intermediate Code
8. Basic Blocks and Traces
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

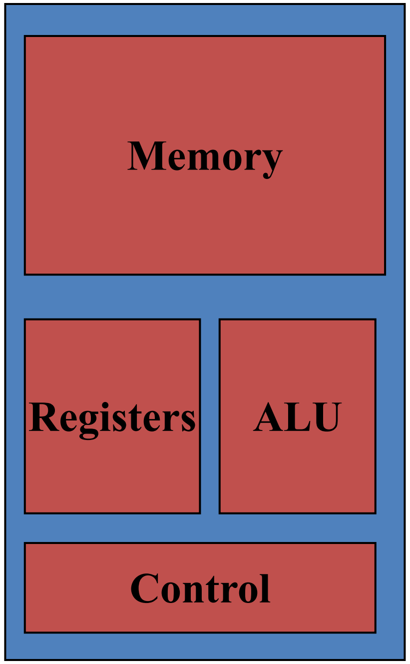
Run-time Environments

- A compiler should translate all "CODE" to assembly instructions and allocate space for DATA"
- To do all these, must know details of modern processors!
  - and the impact on code generation



Overview of a Modern Processor

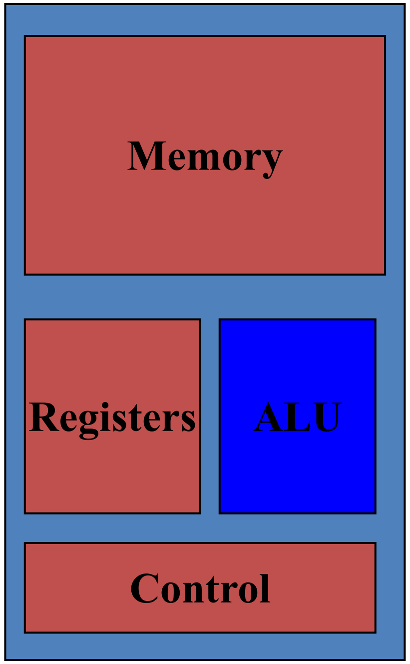
- ALU 算术逻辑单元
- Control
- Memory
- Registers



Modern Processor: Arithmetic and Logic Unit

- **Most arithmetic and logic operation**
  - add rax, rbx ; rax = rax + rbx
  - mov rax, rbx ; rax = rbx

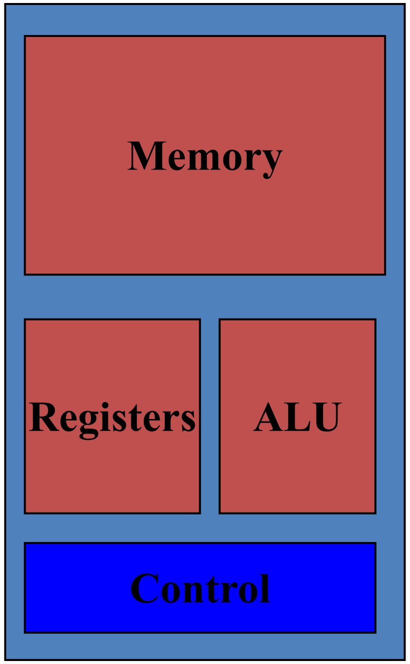
- **Operands:**
  - immediate 立即数
  - register
  - memory



Modern Processor: Control

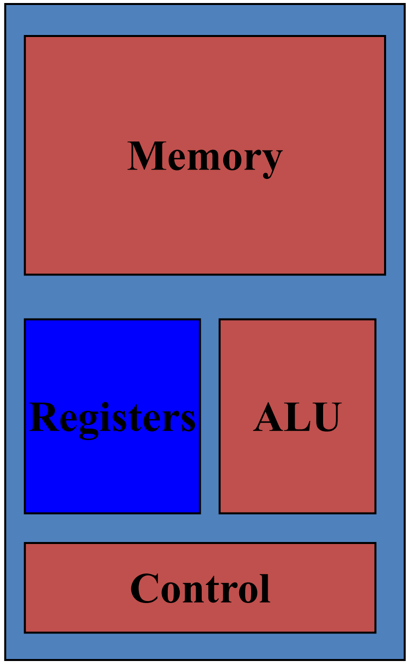
- **Executing instructions**
  - Instructions are in memory (pointed by PC)

```
for (;;)
  instruction = *PC;
  PC++;
  execute (instruction);
```



Modern Processor: Registers

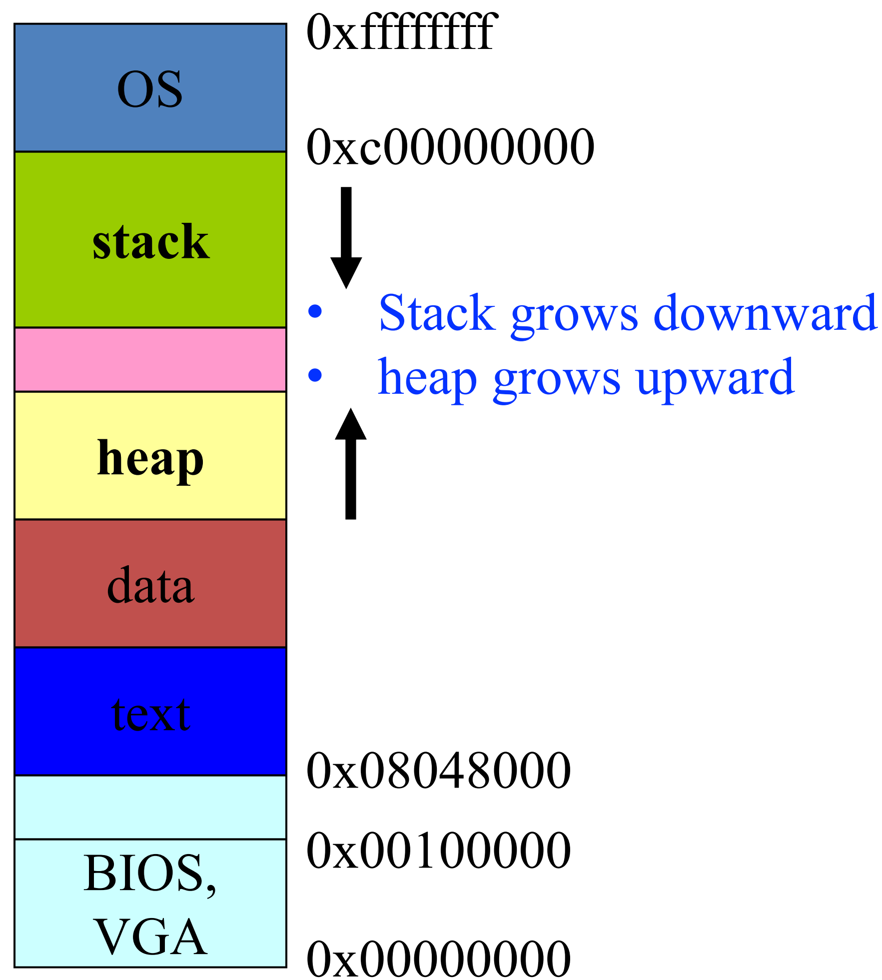
- **Limited but high-speed**
  - More on RISC than x86
- **Most are general-purpose**
  - But some are of special use
  - E.g., rsp, rbp in x86-64



Modern Processor: Memory

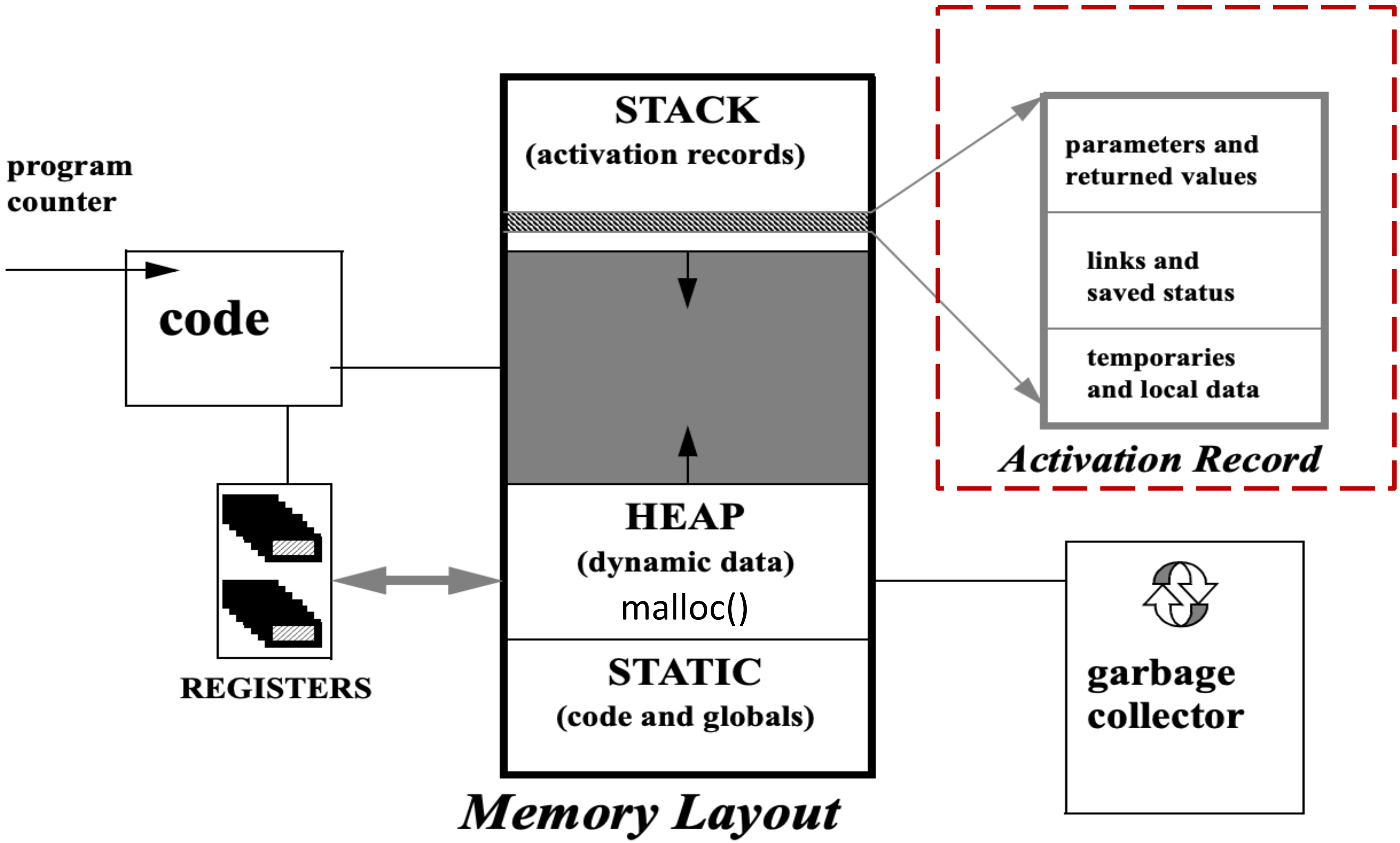
- **Address space is the way how programs use memory**

Dest, Src	C Analog
`mov rax, 0x4`	`temp = 0x4;`
`mov [rax], -147`	`*p = -147;`
`mov rax, [rdx]`	`temp = *p;`



A typical layout of 32-bit x86/Linux

Runtime Memory Layout of Programs



Focus of this Lecture: Activation Record

Activation Record: for Function/Procedure/Method/..

When talking about functions, we may think of:

- **<u>A</u>pplication <u>P</u>rogramming <u>I</u>nterface**
  - Interfaces between source programs
- **<u>A</u>pplication <u>B</u>inary <u>I</u>nterface (e.g., x86 ABI)**
  - Contracts between binary programs
    - Even compiled from other languages by other compilers
  - Conventions on low-level details
    - How to pass arguments?
    - how to return values?
    - how to make use of registers?
    - …

Outline

1. Stack Frame
2. Use of Registers
3. Frame-Resident Variables
4. Block Structure
6. Stack Frame in Tiger

1. Stack Frame

Activation Record/Stack Frame

```
function f(x:int): int =
  let var y :== x+x
   in if y<10
     then f(y)
     else return y-1
  end
```

- There are recursive calls, many of these x's exist simultaneously.

- **An invocation of function P is an <u>activation</u> of P**
- **How to hold local variables?**
  - Each invocation has its own instantiation of local variables
  - Function calls behave in last-in-first-out (LIFO) fashion
  - Use a LIFO data structure – a **stack**

Activation Record/Stack Frame

- Activation record or stack frame(栈帧): a piece of memory on the stack for a function
- The stack frame connects the caller to the callee, e.g.,
  - Relevant machine state (saved registers, return address)
  - Space for return value
  - Space for local data
  - Pointer to activation for accessing **non-local** data

Main problem: how to **layout** the activation record so that the caller and callee can **communicate** properly ?

Activation Record/Stack Frame的设计

- **活动记录的具体组织和实现不唯一**
  - 即使是同一语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异

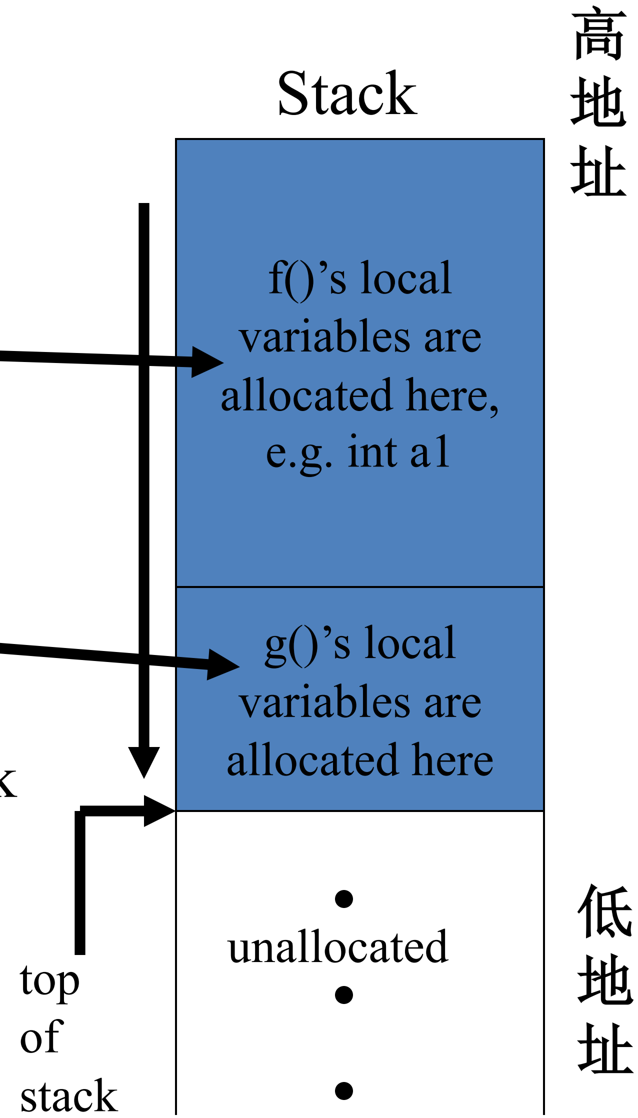
- **本节剩下的内容**
  1. 《深入理解计算机系统》中的Stack Frame例子
  2. 讨论相关优化: 寄存器的使用
  3. 讨论frame-resident variables
  4. Tiger语言嵌套函数的实现
  5. Tiger编译器的典型Stack Frame

Relating the Code to the Stack

```
f() {
   int a1;
   ...
   g();
   ...
}
```

space for f()'s local variables gets allocated on the stack

space for g()'s local variables is allocated below f() on the stack

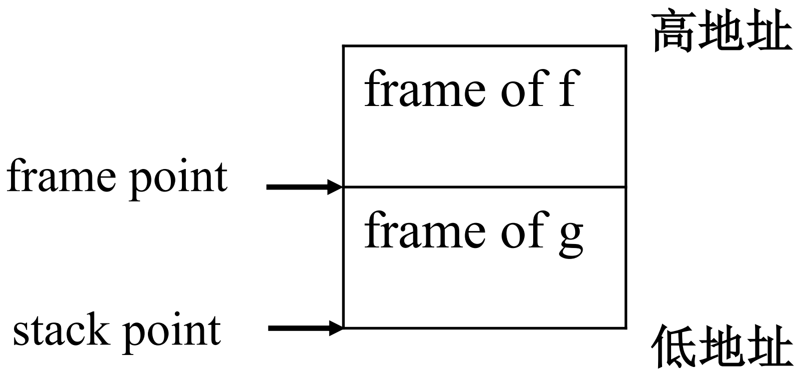


Stack frame从高地址往低地址增长

Frame and Stack Pointers

✓ Stack pointer: **栈顶寄存器**
- x86: esp 、rsp
- ARM: SP

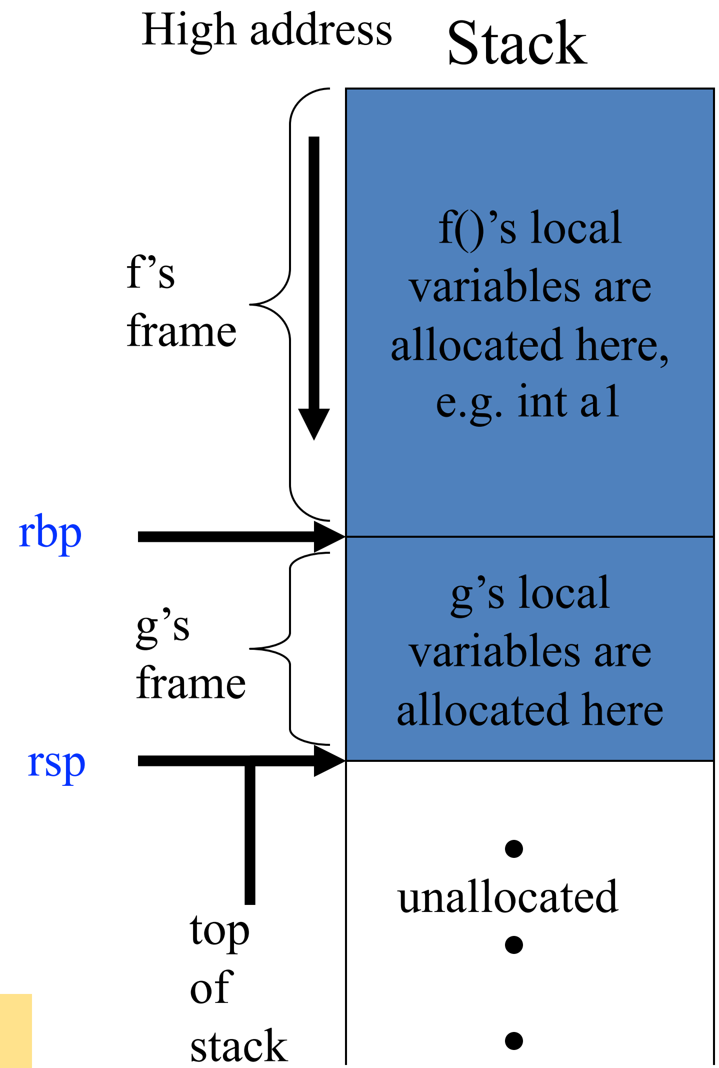
✓ Frame pointer: **基址寄存器**
- x86: ebp、 rbp
- ARM: FP



- **Frame Pointer** (base pointer, **基址寄存器)**
  - Points to the **start of the current frame**
  - The compiled code references local variables and arguments by using offsets to the frame pointer
- **Stack Pointer (栈顶寄存器)**
  - Points to the **end of the current frame**
  - Referring to the top of the stack

Example: Frame and Stack Pointers

```
f() {
   int a1;
   ...
   g();  // callee
   ...
}
```



While executing the callee g(),

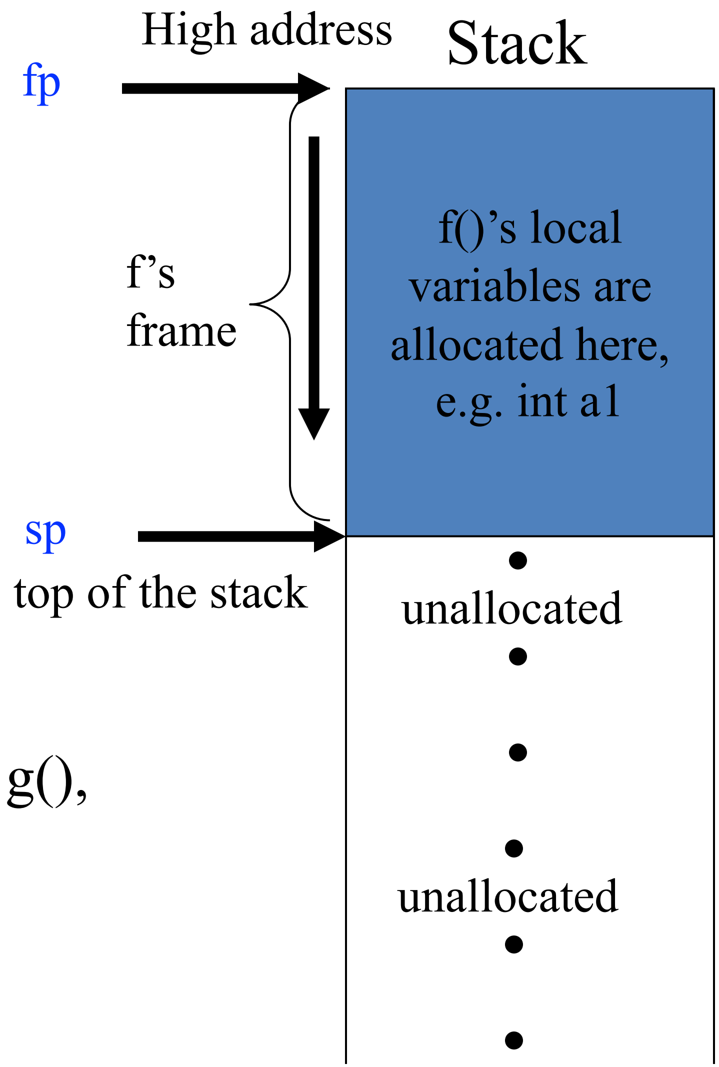
- Frame pointer rbp points to the beginning of g's frame
- Stack pointer rsp points to the top of the stack

问题: f调用g的完整过程，Stack frame以及frame pointer, stack point会发生什么变化?

Step I: Calling a Function

- **rsp**: Stack pointer, 栈顶寄存器
- **rbp**: Frame pointer, 基址寄存器

```
f() {
   int a1, a2;
   ...                  PC
   g(a1, a2);
   ...
}
```

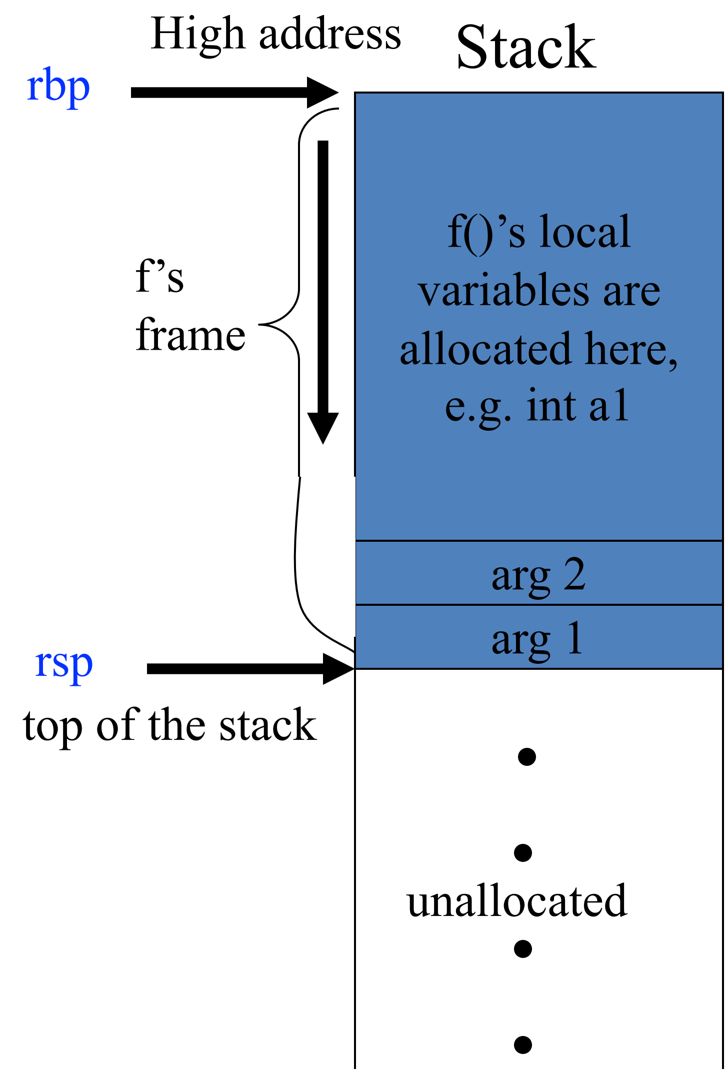


- When the PC is here, just before calling g(), the current stack frame look like this:

Step I: Calling a Function

- rsp: Stack pointer, 栈顶寄存器
- rbp: Frame pointer, 基址寄存器

```
f() {
   int a1, a2;
   ...
   g(a1, a2);     <——— PC
   ...
}
```

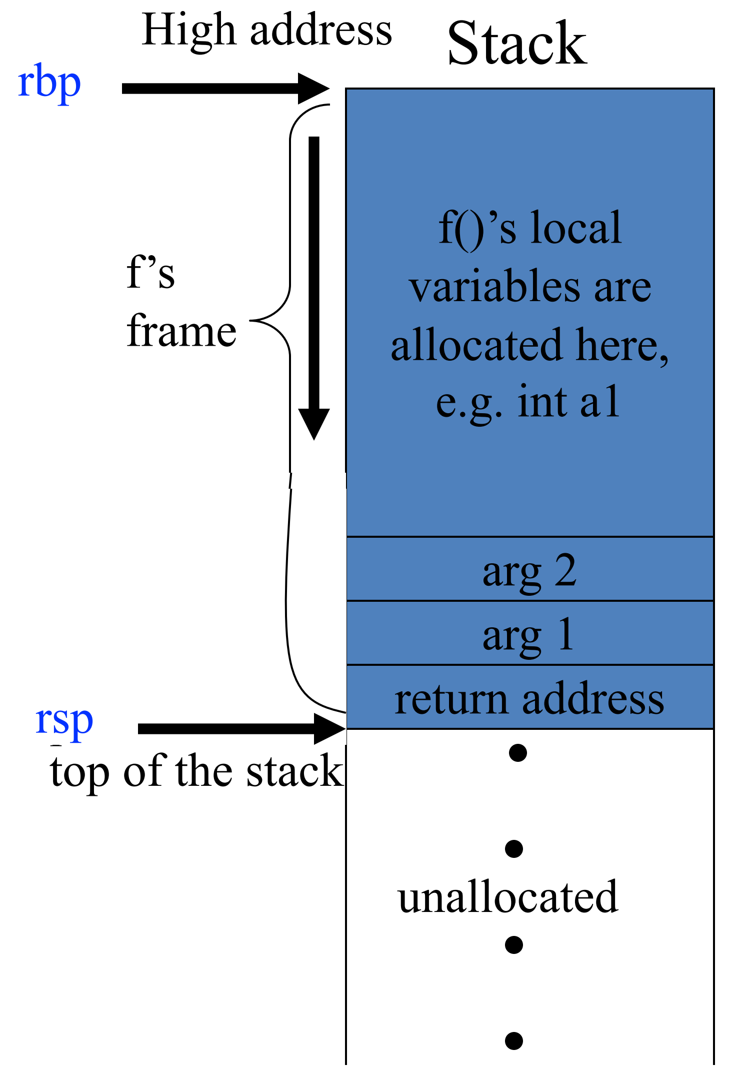


- Push arguments onto the stack

Step I: Calling a Function

- **rsp: Stack pointer, 栈顶寄存器**
- **rbp: Frame pointer, 基址寄存器**

```
f() {
  int a1, a2;
  ...
  g(a1, a2);        <-- PC
  ...
}
```

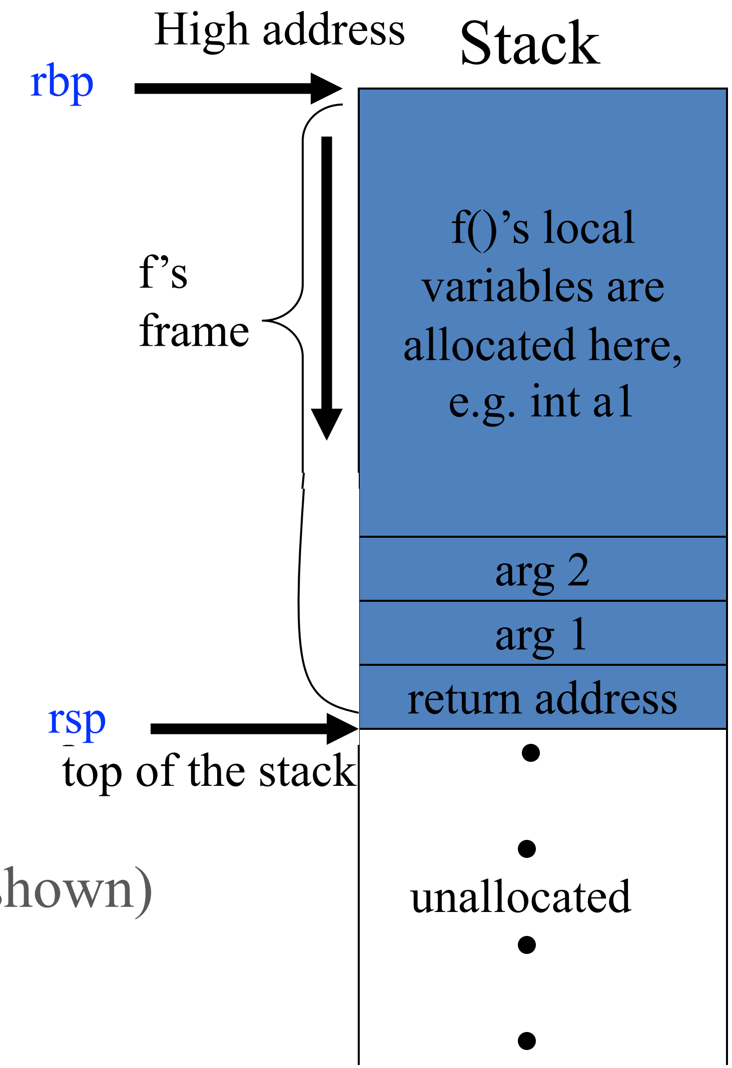


- Push arguments onto the stack
- Push the return address onto the stack

Step I: Calling a Function

- **rsp: Stack pointer, 栈顶寄存器**
- **rbp: Frame pointer, 基址寄存器**

```
f() {
   int a1, a2;
   ...
   g(a1, a2);      <- PC
   ...
}
```



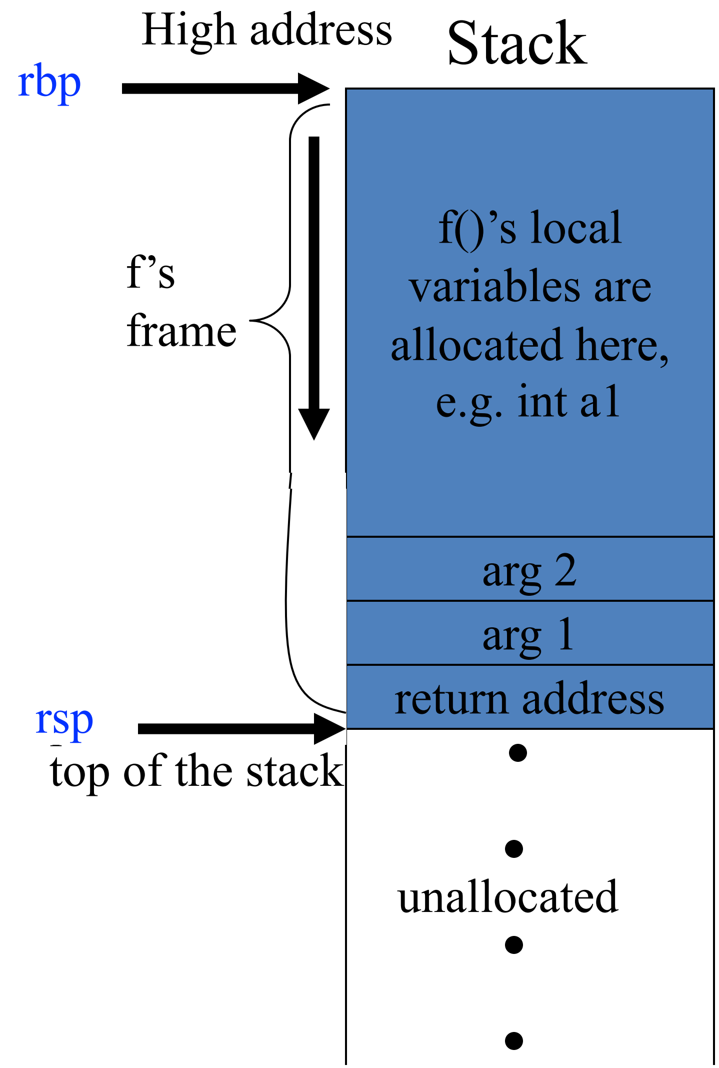
- Push arguments onto the stack
- Push the return address onto the stack
- Save caller-save registers on stack (not shown)
- Jump to called function g (changes PC)

Step II: Entering the Callee

- rsp: Stack pointer, 栈顶寄存器
- rbp: Frame pointer, 基址寄存器

```
g(int v1, v2) {
   local var's
   ...
}
```

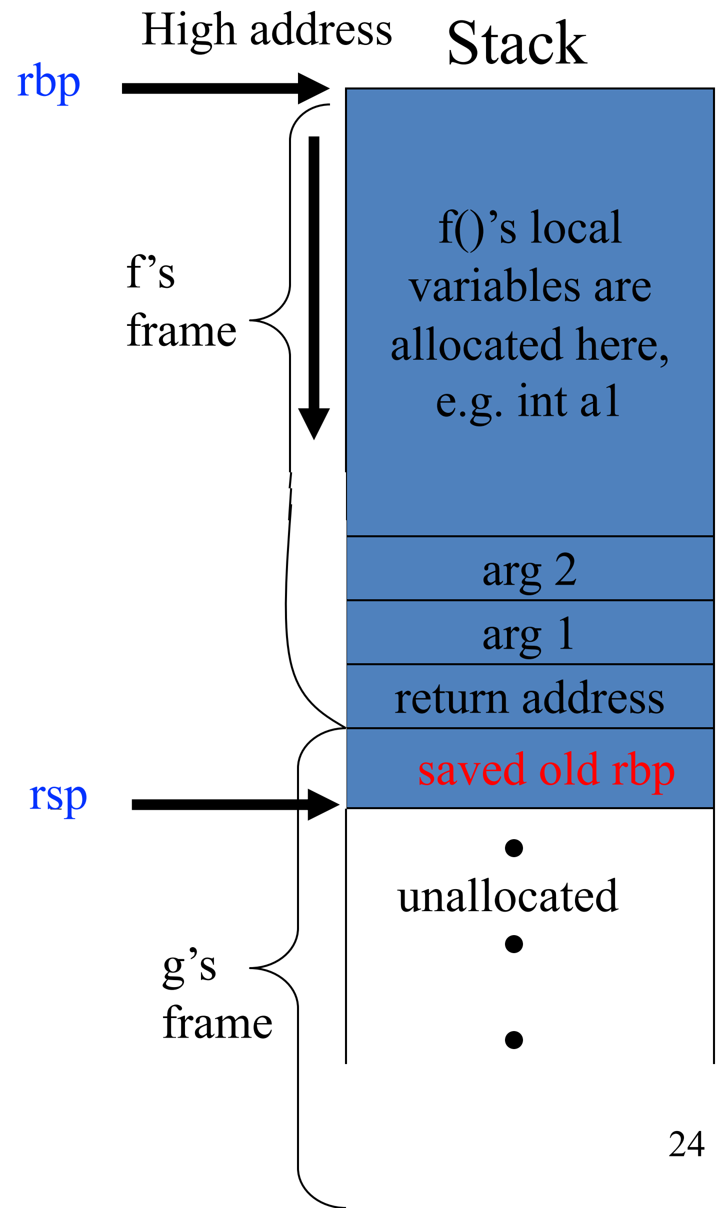
- push rbp : save the old frame pointer by pushing it onto the stack



Step II: Entering the Callee

- **rsp**: Stack pointer, 栈顶寄存器
- **rbp**: Frame pointer, 基址寄存器

```
g(int v1, v2) {
   local var's
   …
}
```



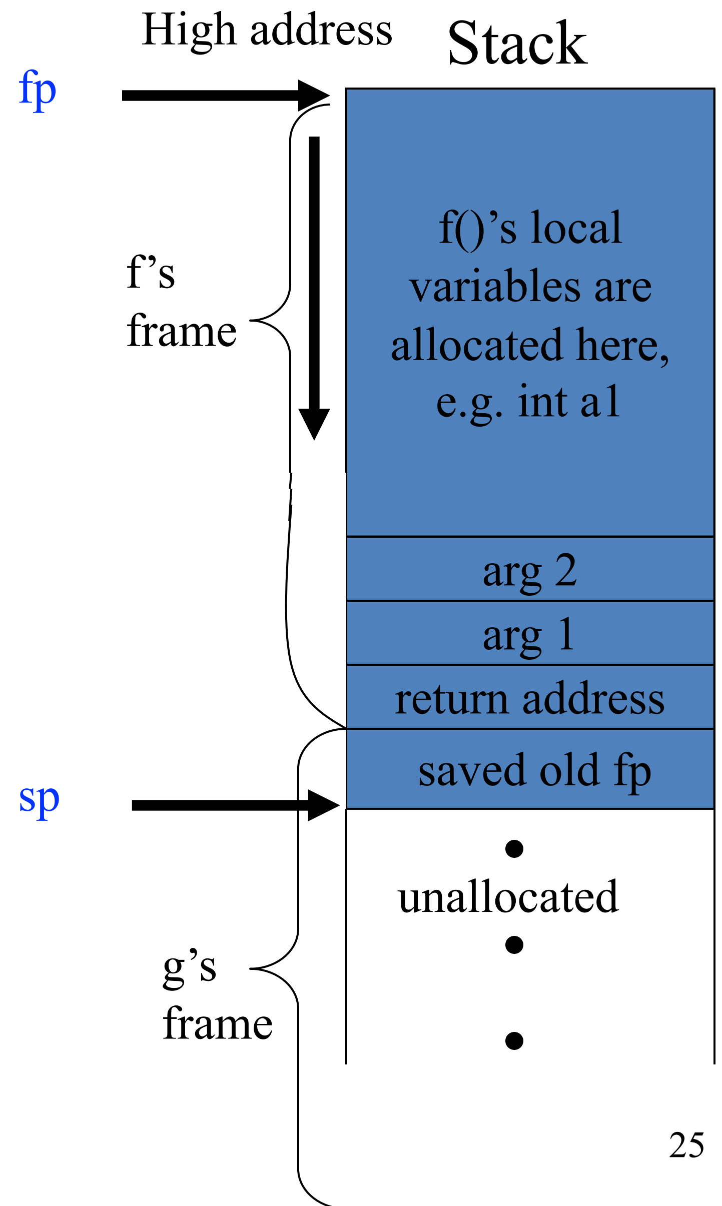
- push rbp : save the old frame pointer by pushing it onto the stack
  - 方便返回caller f时恢复f的栈帧
  - 注意: stack point rsp往下移动了

Step II: Entering the Callee

- **rsp**: Stack pointer, 栈顶寄存器
- **rbp**: Frame pointer, 基址寄存器

```
g(int v1, v2) {
   local var’s
   …
}
```

- push rbp : save the old frame pointer by pushing it onto the stack
- mov rbp, rsp : reset frame pointer (rbp) to the current stack pointer (rsp)!
  (让rbp指向新的stack frame起点)

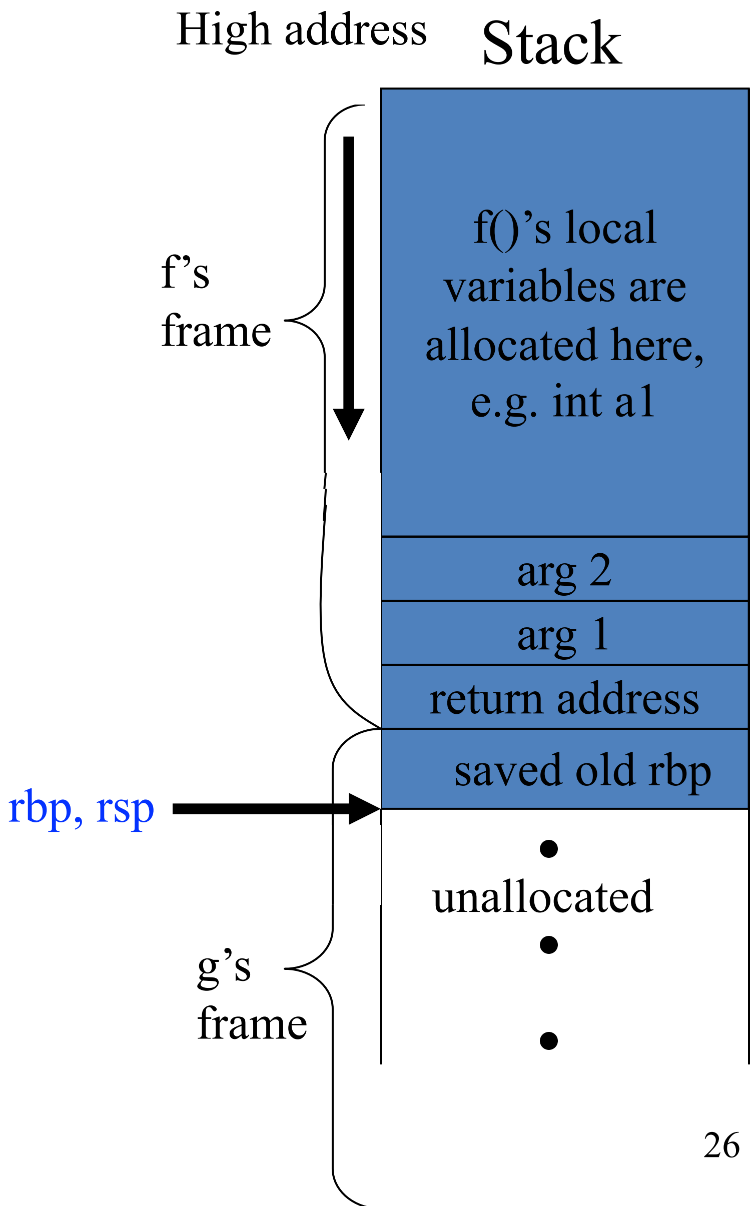


Step II: Entering the Callee

- rsp: Stack pointer, 栈顶寄存器
- rbp: Frame pointer, 基址寄存器

```
g(int v1, v2) {
  local var's
  ...
}
```

- push rbp : save the old frame pointer by pushing it onto the stack
- mov rbp, rsp : reset frame pointer (rbp) to the current stack pointer (rsp)!
  - rbp指向了新frame的起始
  - rsp, rbp目前指向同一位置

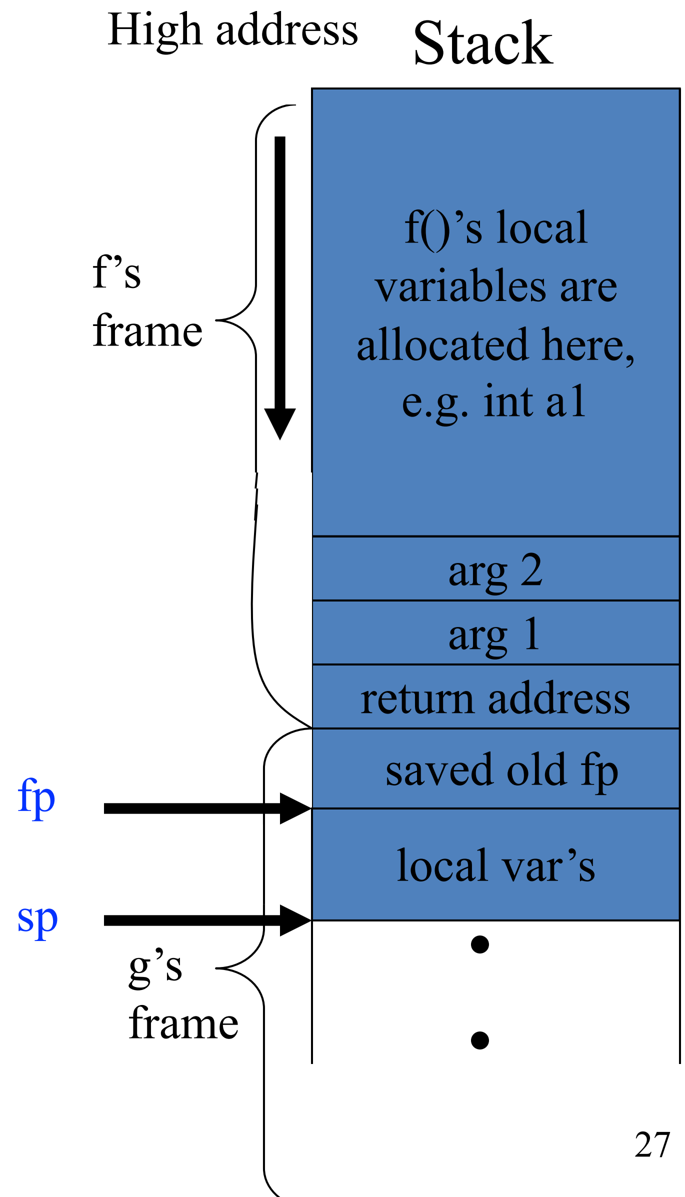


Step II: Entering the Callee

- rsp: Stack pointer, 栈顶寄存器
- rbp: Frame pointer, 基址寄存器

```
g(int v1, v2) {
   local var's
   ...
}
```

- push rbp : save the old frame pointer by pushing it onto the stack
- mov rbp, rsp : reset frame pointer (rbp) to the current stack pointer (rsp)!
- save any callee-saver registers on stack (not shown)
- allocate local variables by decrementing the stack ptr

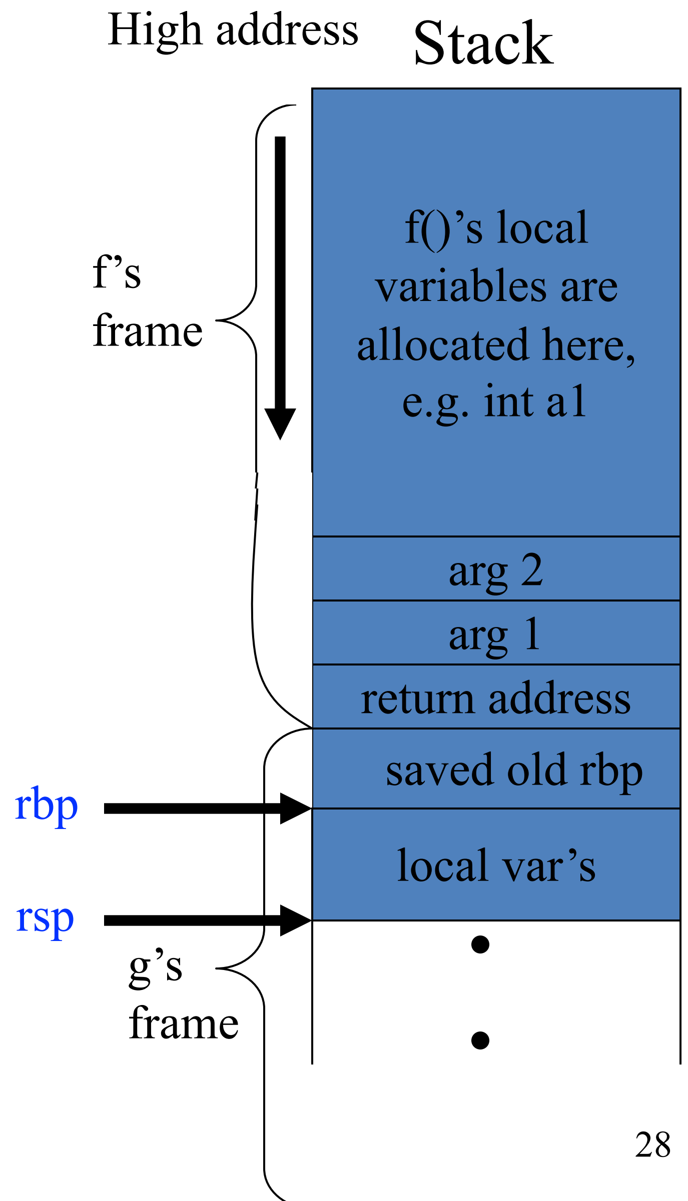


Step III: Exiting the Callee

- rsp: Stack pointer, 栈顶寄存器
- rbp: Frame pointer, 基址寄存器

```
g(int v1, v2) {
   local var's
   ...
}
```

- g restores callee-save registers (not shown)
- mov rsp rbp: deallocate locals by reseting stack ptr to current frame ptr

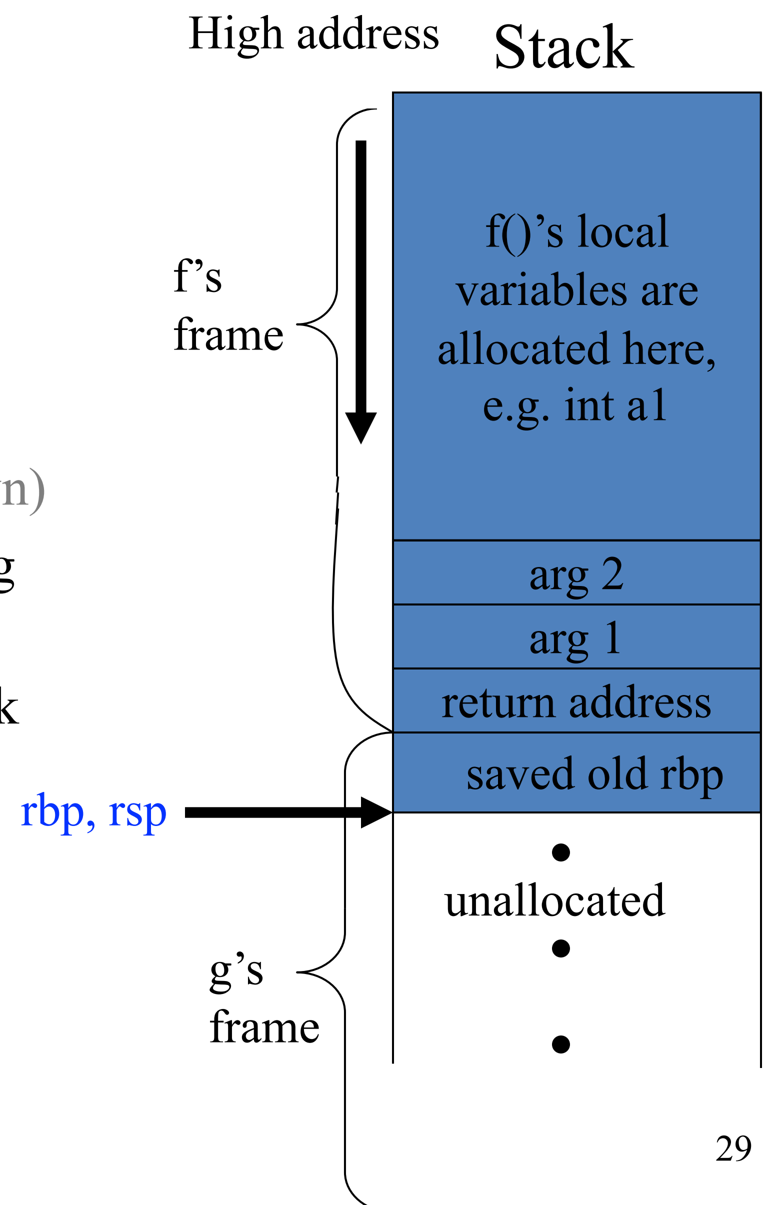


Step III: Exiting the Callee

- rsp: Stack pointer, 栈顶寄存器
- rbp: Frame pointer, 基址寄存器

```
g(int v1, v2) {
  local var's
  ...
}
```

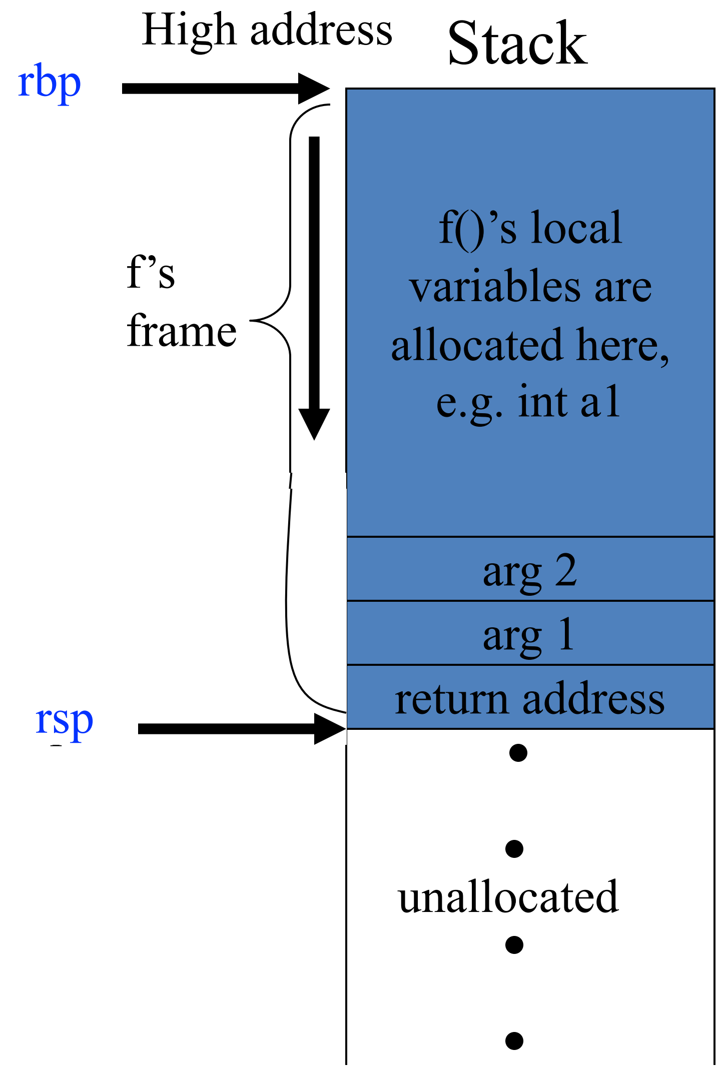
- g restores callee-save registers (not shown)
- mov rsp rbp: deallocate locals by reseting stack ptr to current frame ptr
- pop rbp: saved frame pointer off the stack and into the frame ptr
  - 把当前栈顶元素(saved old rbp)赋给rbp



Step III: Exiting the Callee

- **rsp: Stack pointer, 栈顶寄存器**
- **rbp: Frame pointer, 基址寄存器**

```
g(int v1, v2) {
  local var's
  ...
}
```



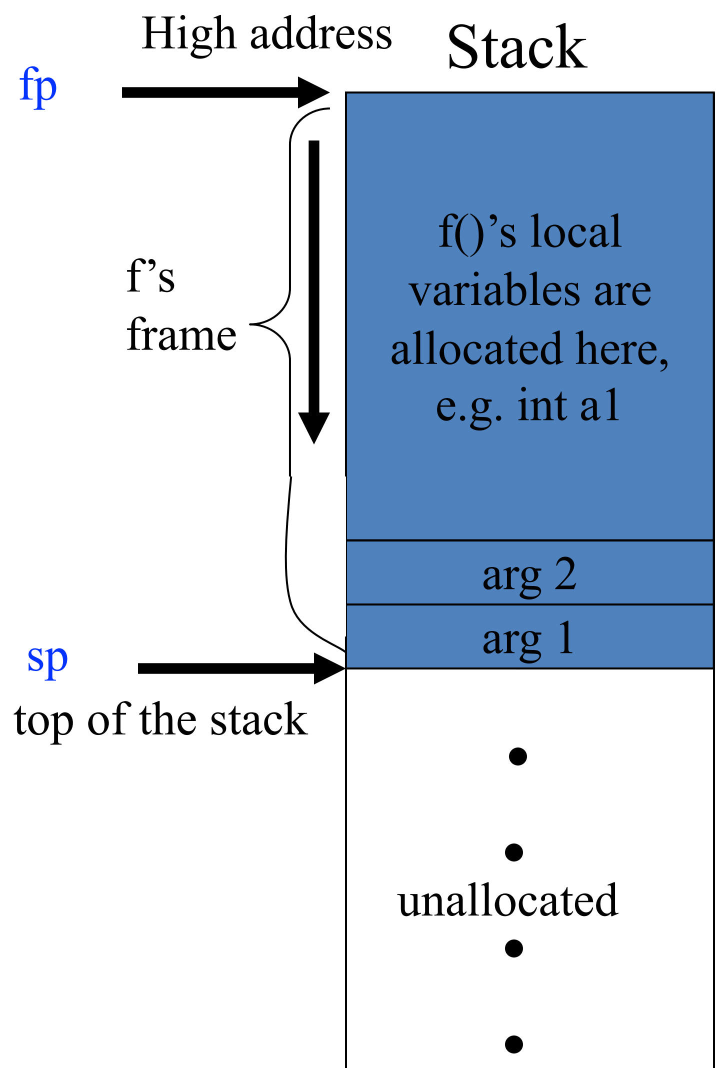
- g restores callee-save registers (not shown)
- mov rsp rbp: deallocate locals by reseting stack ptr to current frame ptr
- pop rbp: saved frame pointer off the stack and into the frame ptr
  - rbp指向的caller(函数)的stack frame起点
- ret: pop the saved return address off the stack and jump to this location

Step III: Exiting the Callee

- **rsp: Stack pointer, 栈顶寄存器**
- **rbp: Frame pointer, 基址寄存器**

```
g(int v1, v2) {
  local var's
  ...
}
```

- g restores callee-save registers (not shown`
- mov rsp rbp: deallocate locals by reseting stack ptr to current frame ptr
- pop rbp: saved frame pointer off the stack and into the frame ptr
- ret: pop the saved return address off the stack and jump to this location



Summary: Stack Frame

Suppose a function f(…) calls thefunction g(a1,…, an)

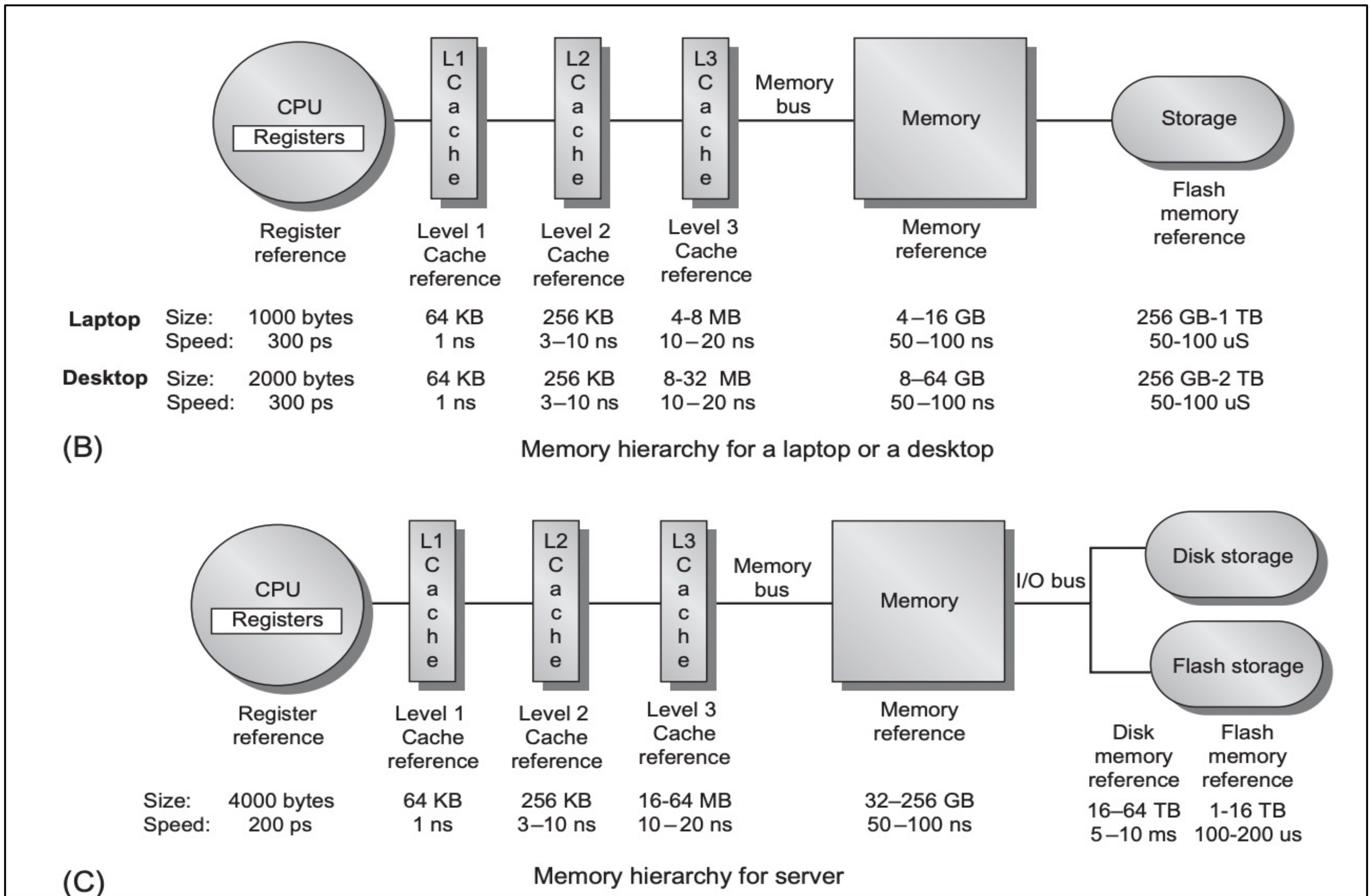
- When f calls g:
  - The stack pointer points to the first argument that f passes to g
  - g allocates a frame by simply subtracting the frame size from the stack pointer (SP)
- When entering g:
  - save the old frame pointer FP in memory in the frame
  - FP = SP
- When g exists:
  - SP = FP
  - fetch back the saved old frame point(FP)

3. Use of Registers

How to reduce memory traffic?

Recap: Memory Hierarchy

- Accessing registers is MUCH faster than accessing memory!



Computer Architecture Quantitative Approach, Sixth Edition

Reducing Memory Allocation in Stack Frame

- **Problem**: Putting everything in the stack frame can cause the memory traffic
- **Solution**: Hold as much of the frame as possible in registers
  - (Some) function parameters
  - Function return address
  - Function return value
  - (Some) Local variables
  - (Some) Intermediate results of expressions (temporaries)

Using Registers: Parameter Passing

- **Tiger的参数方式: Call-by-value**
  - Values of the actual arguments are passed and established as values of formal parameters.
  - Modification to formals have no effect on actuals

```
function swap(x : int, y : int) =
  let var t : int := x in x := y; y := t end
```

- Parameters are passed on the stack for most machines designed in 1970s
  - **Problem**: caused the memory traffic

Using Registers: Parameter Passing

- **Problem**: passing parameter stack causes memory traffic
- **Solution**: parameter-passing convention on modern machines
  - The first k arguments (k = 4 or 6) are passed in registers
    - **X86-64:** rdi, rsi, rcx, rdx; **ARM：** r0~r3
  - The rest are passed on the stack

```
int g(long long x) {
  return x + 1;
}

void f() {
  g(10086);
}
```

```
g(long long): # @g(long long)
        push rbp
        mov rbp, rsp
        mov qword ptr [rbp - 8], rdi
        mov rax, qword ptr [rbp - 8]
        add rax, 1
        pop rbp
        ret
f(): # @f()
        push rbp
        mov rbp, rsp
        mov rdi, 10086
        call g(long long)
        pop rbp
        ret
```

rdi传第一个参数

Using Registers: Parameter Passing

- **New problem:** extra memory traffic caused by passing arguments in registers!
- The need for "saving" the status of registers
  - Suppose ***f*** uses register **r** to hold a local variable and calls ***g***, and ***g*** also uses **r** for its own calculations
  - **r** must be saved (stored into a stack frame) before ***g*** uses it, and restored (fetched back from the frame) after ***g*** finishes using it.

- **r** is a **caller-save register** if the caller must save and restore it
  - E.g., rdi, rsi, rcx, rdx
- **r** is a **callee-save register** if it is the responsibility of the callee
  - E.g., rbx, rbp (frame pointer)

如果 *f* 在用edi传参前，要先把edi当前的值存入stack frame(调用完再恢复)，那最终并没有避免stack frame的访存操作!

Example: Call-Save Registers

- rbx and rbp are callee-save registers

```c
void
swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;

    *xp = t1;
    *yp = t0;
}
```

```
swap:
   push   rbp  ;保持rbp的值
   mov    rbp, rsp
   push   rbx  ;保存rbx的值

     ...; 假设这里修改了rbx
     ...

   pop     rbx  ;栈顶元素赋给rbx
   pop     rbp  ;栈顶元素赋给rbp
   ret
```

Set Up

Body

Finish

More about Register Saving Conventions

- Should value reside in caller-save or callee-save registers?
  - Not so easy to determine, and no general rules
  - We'd also come back to this issue later in register allocation!

%rax	Return value – Caller saved	%r8	Argument #5 – Caller saved
%rbx	**Callee** saved	%r9	Argument #6 – Caller saved
%rcx	Argument #4 – Caller saved	%r10	Caller saved
%rdx	Argument #3 – Caller saved	%r11	Caller Saved
%rsi	Argument #2 – Caller saved	%r12	**Callee** saved
%rdi	Argument #1 – Caller saved	%r13	**Callee** saved
%rsp	Stack pointer	%r14	**Callee** saved
%rbp	**Callee** saved	%r15	**Callee** saved

x86-64 64-bit Register Conventions

除了"用寄存器传参"这一特殊情况，其他情况也可能涉及"如何使用callee-saved/caller-saved寄存器"的权衡(本节先不考虑)

Using Registers: Parameter Passing (cont'd)

```
1:  f(int a) {
2:      int z = ...
3:      h(z);
4:      ...
5:      int t = a + 2;
6:      …
}
```

- Suppose ***f*** received its parameter in register $\mathbf{r_1}$
- Suppose ***f*** passes ***z*** to the callee ***h*** via register $\mathbf{r_1}$
- ***f*** should save the old contents of $\mathbf{r_1}$ in stack frame before calling ***h*** !

Such memory traffic was supposedly avoided by passing arguments in registers!

Using Registers: Parameter Passing (cont'd)

```
1:  f(int a) {
2:      int z = ...
3:      h(z);
4:      ...
5:      int t = a + 2;
6:      …
}
```

- Suppose *f* received its parameter in register $\mathbf{r_1}$
- Suppose *f* passes *z* to the callee *h* via register $\mathbf{r_1}$
- *f* should save the old contents of $\mathbf{r_1}$ in stack frame before calling *h* **!**

How to avoid the extra memory traffic?

1. **If the parameter *a* is a dead variable at the point where *h* is called, then *f* can overwrite $\mathbf{r_1}$ without saving it.**

例如: 在调用h后a不再使用(不存在第5行)

Using Registers: Parameter Passing (cont'd)

```
1:  f(int a) {
2:     int z = ...
3:     h(z);
4:     ...
5:     int t = a + 2;
6:     …
}
```

- Suppose *f* received its parameter in register $\mathbf{r_1}$
- Suppose *f* passes *z* to the callee *h* via register $\mathbf{r_1}$
- *f* should save the old contents of $\mathbf{r_1}$ in stack frame before calling ***h*** **!**

How to avoid the extra memory traffic?

2. Use global register allocation: different functions use different set of registers to pass arguments

例如: f可用寄存器$\mathbf{r_1}$接收参数，但通过寄存器$\mathbf{r_2}$给f传参

Using Registers: Parameter Passing (cont'd)

```
1:  f(int a) {
2:      int z = ...
3:      h(z);
4:      ...
5:      int t = a + 2;
6:      …
}
```

- Suppose ***f*** received its parameter in register $\mathbf{r_1}$
- Suppose ***f*** passes ***z*** to the callee ***h*** via register $\mathbf{r_1}$
- ***f*** should save the old contents of $\mathbf{r_1}$ in stack frame before calling ***h*** **!**

How to avoid the extra memory traffic?

3. Leaf procedures 不调用其他过程的为叶子过程(Leaf procedure)。叶子过程不必将传入的参数保存到存储器中

Using Registers: Parameter Passing (cont'd)

How to avoid the extra memory traffic?

1. Parameter x is a dead variable at the point where $h(z)$ is called. Then $f(x)$ can overwrite $\mathbf{r_1}$ without saving it.

2. Use **global register allocation:** Different functions use different set of registers to pass arguments

3. Leaf procedures: parameters of leaf procedures can be allocated in registers without causing any extra memory traffic

4. Use **register windows** (as on SPARC): Each function invocation can allocate a fresh set of registers

Using Registers: Return Address

- If the *call* instruction within *g* is at address *a*, then (usually)
  - The right place to return to is $a + 1$, the next instruction in *g*.
  - This is called the *return address*.

- On modern machines, the *call* instruction merely puts the return address in a designated register.

- A nonleaf procedure will have to write it to the stack (unless interprocedural register allocation is used)，a leaf procedure will not.

Using Registers: Return Value

- **Return value**: placed in designated register by callee function.
  - X86-64系统整型返回值： rax

Using Registers: Locals and Temporaries

- **(Some) Local variables**
- **(Some) Intermediate results of expressions (temporaries)**

To be discussed in register allocation section

4. Frame-Resident Variables

既然很多地方都可以用寄存器，那还需要在stack frame中分配内存空间吗?

Frame-Resident Variables

- **A variable will be allocated in stack frames because**
  - It is *passed by reference*, so it must have a memory address
  - Its *address is taken*, e.g., &a in the C language;
  - It is accessed by a procedure nested inside the current one;
  - The value is too big to fit into a single register;
  - The variable is an array, for which address arithmetic is necessary to extract components;
  - The register holding the variable is needed for a specific purpose, such as parameter passing (as described above);
  - The are too many locals and temporaries – "spill" [To be discussed in Register Allocation]

Frame-Resident Variables

- **The variable escapes for any of the reasons**
  - It is *passed by reference*, so it must have a memory address
  - Its *address is taken*, e.g., &a in the C language;
  - It is accessed by a procedure nested inside the current one

- E.g., **pass-by-reference** (supported in Pascal, but **not in Tiger**)
  - Locations of the actuals are passed;
  - References to the formals include implicit indirection to access values of the actuals.
  - Modifications to formals do change actuals!

Summary

- **Registers hold**
  - Some parameters
  - Return address
  - Return value
  - Some local variables and temporaries
- **Stack frame holds**
  - Variables passed by reference or have their address taken (&)
  - Variables that are accessed by procedures nested within current one.
  - Variables that are too large to fit into register file.
  - Array variables (address arithmetic needed to access array elements).
  - Spilled registers (Too many local variables to fit into register file, so some must be stored in stack frame)

2. Block Structure

- Static Link
- Display
- Lambda Lifting

Motivation: Implementing Block Structure

- **Block Structure**: In languages allowing nested function declarations (such as Tiger), the inner functions may use variables declared in outer functions.

- We can access local variables through the **Frame Pointer** (notice, the actual value of FP is unknown until runtime, but the each local-variable's offset to FP is known at compile time!)

- **Problem**: How can h access the "non-local" variables m?

```
int f (int x, int y)
{
  int m;
  int g (int z)
  {
    int h ()
    {
      return m+z;
    }
    return 1;
  }
  return 0;
}
```

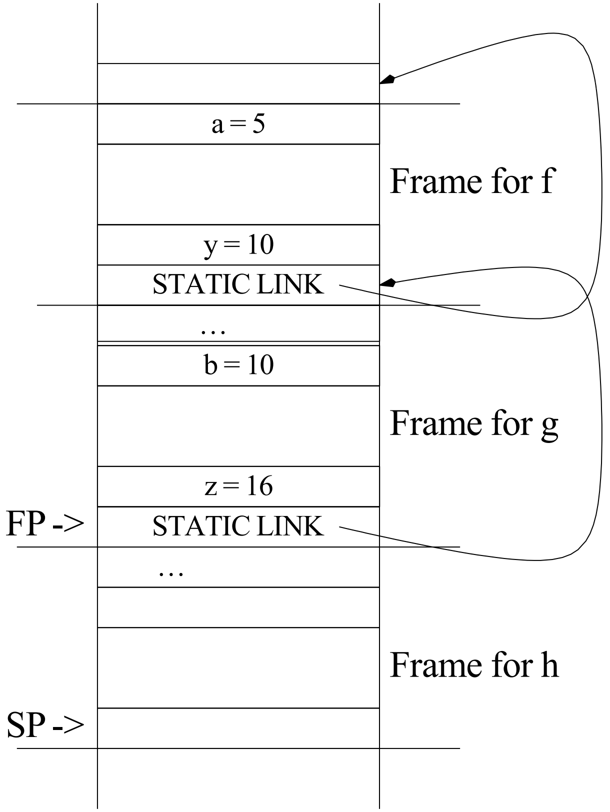
Strategies for Implementing Block Structure

- **Static link (重点)**
  - Whenever a function *g* is called, it can be passed a pointer to the frame of the function statically enclosing *g*; this pointer is the ***static link***.
- **Lambda Lifting**
  - When *f* calls *g*, each variable of *f* that is actually accessed by *g* (or by any function nested inside *g*) is passed to *g* as an extra argument. This is called ***lambda lifting***.
- **Display**
  - A global array can be maintained, containing – in position i – a pointer to the frame of the most recently entered procedure whose static nesting depth is i. This array is called a ***display***

I: Static Link

- **The static link**: Whenever *g* is called, it is passed pointer to most recent activation record of *f* that **immediately encloses *g* in program text**

```
let
   function f(): int =
     let
       var a := 5
       function g(y: int): int =
         let
          var b := 10
          function h(z: int): int =
             if z > 10 then h(z / 2)
             else z + b * a
         in
          y + a + h(16)
        end
     in
       g(10)
     end
in f() end
```



I: Static Link

- The static link: Whenever *g* is called, it is passed pointer to most recent activation record of *f* that immediately encloses *g* in program text

> The static link is a pointer to the activation record of the enclosing procedure

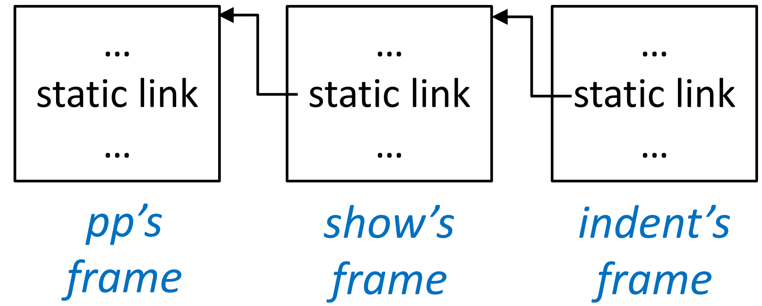
- **Using static links to access non-local data**
  - Each function is annotated with its enclosing depth
  - When a function at depth *n* accesses a variable at depth *m*
    - Emit code to climb up *n-m* links to visit the appropriate activation record

Example: Static Links

```
type tree = {key: string, left: tree, right:
tree}

function prettyprint(tree: tree) : string =
  let
    var output := “”
    function write(s: string) =
      output := concat(output,s)

    function show(n:int, t: tree) =
      let function indent(s: string) =
            (for i := 1 to n
             do write(“ ”));
             output := concat(output, s);
             write("\n"))
      in if t=nil
         then indent(".")
         else (indent(t.key));
               show(n+1, t.left);
               show(n+1, t.right))
      end
    in show(0, tree); output
  end
```



How can indent use output from *prettyprint*’s frame?
It starts with its own static link, then fetch *show*’s, then fetches *output*.

Example: Static Links

```
int f (int x, int y)
{
  int m;
  int g (int z)
  {
    int h ()
    {
      return m+z;
    }
    return 1;
  }
  return 0;
}
```

```
int f (link,int x, int y) {
  int m;
  int g (link, int z){
    int h (link){
      return link->
                 prev->m+
             link->z;
    }
    return 1;
  }
  return 0;
}
```

Pros and Cons of Static Links

- **Pros**
  - Little extra overhead on parameter passing
- **Cons**

  The overhead to climb up a static link chain to access non-locals
  - Need a chain of indirect memory references for each variable access
  - Number of indirect references = difference in nesting depth between variable declaration function and use function
  - Functions may be deeply nested!

II: Lambda Lifting

- When g calls f, each variable of g that is actually accessed by f (or by any function *nested inside* f) is passed to f as an extra argument.
  - Rewriting the program by treating non-local variables as formal parameter

- The translation/transormation process starts with the inner-most procedures and works its way outwards

Example: Lambda Lifting

```
int f (int x, int y)
{
  int m;
  int g (int z)
  {
    int h ()
    {
      return m+z;
    }
    return 1;
  }
  return 0;
}
```

```
int f (int x, int y)
{
  int m;
  int g (int z)
  {
    int h (int &m, &z)
    {
      return m+z;
    }
    return 1;
  }
  return 0;
}
```

Example: Lambda Lifting

```
int f (int x, int y)
{
  int m;
  int g (int z)
  {
    int h ()
    {
      return m+z;
    }
    return 1;
  }
  return 0;
}
```

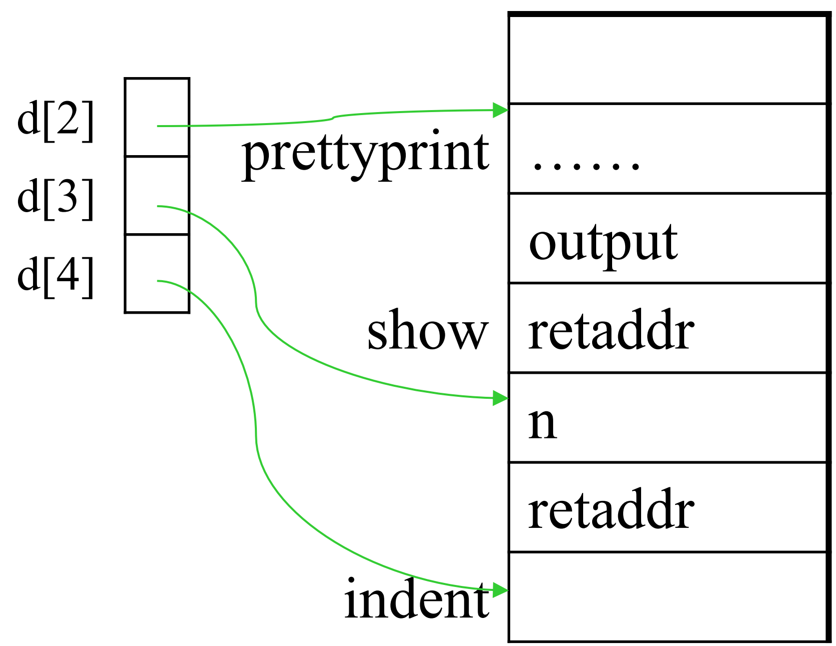
```
int f (int x, int y)
{
  int m;
  int g (int &m, int z)
  {
    int h (int &m, &z)
    {
      return m+z;
    }
    return 1;
  }
  return 0;
}
```

III: Display

- **Display: a global array of pointers to frames**
  - It keeps track of the lexical nesting structure of the program
  - In position i – a pointer to the frame of the most recently entered procedure whose static nesting depth is i.
  - Essentially, it points to the currently set of activation records that contain accessible variables

main	1
prettyprint	2
write	3
show	3
indent	4

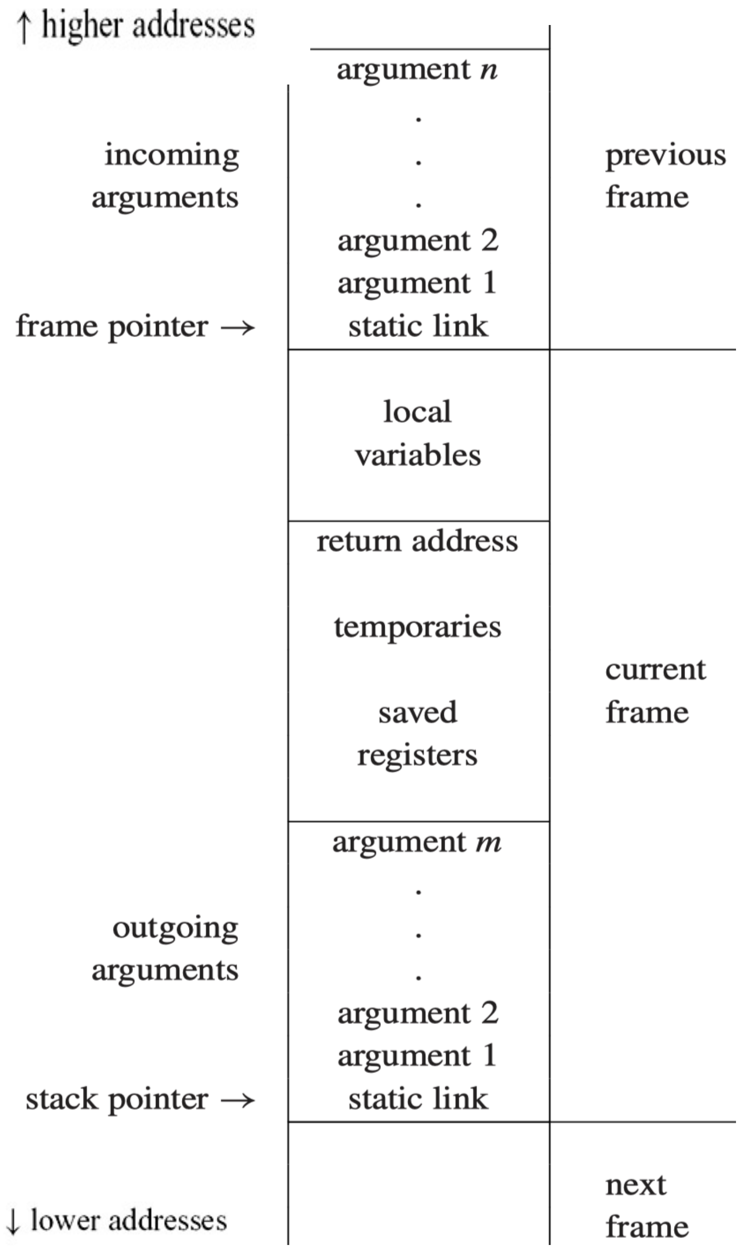
Static nesting depth



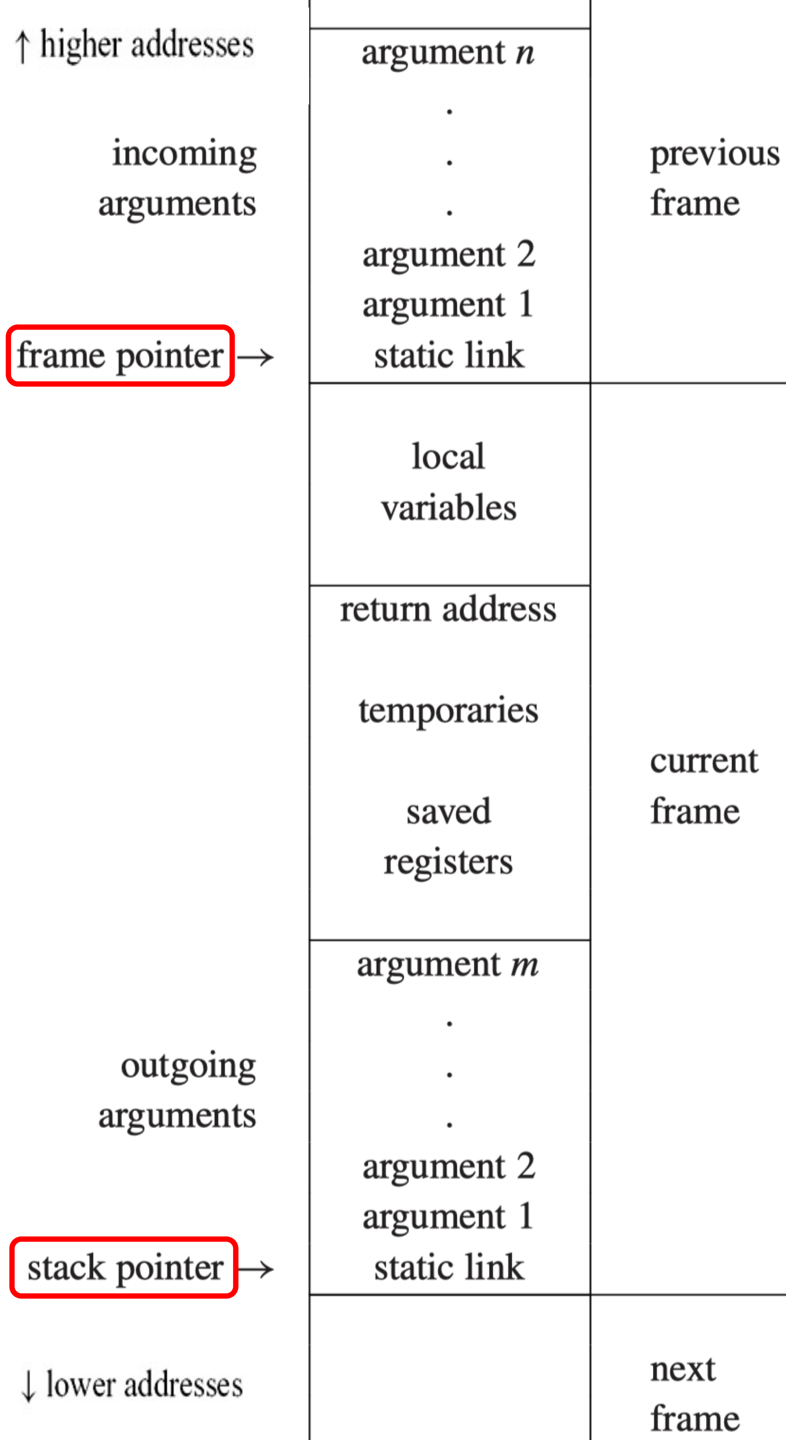
4. A Typical Stack Frame Layout for Tiger

注: 和1. Stack Frame中提到的Stack frame可能有所不同，考试以这一部分为主

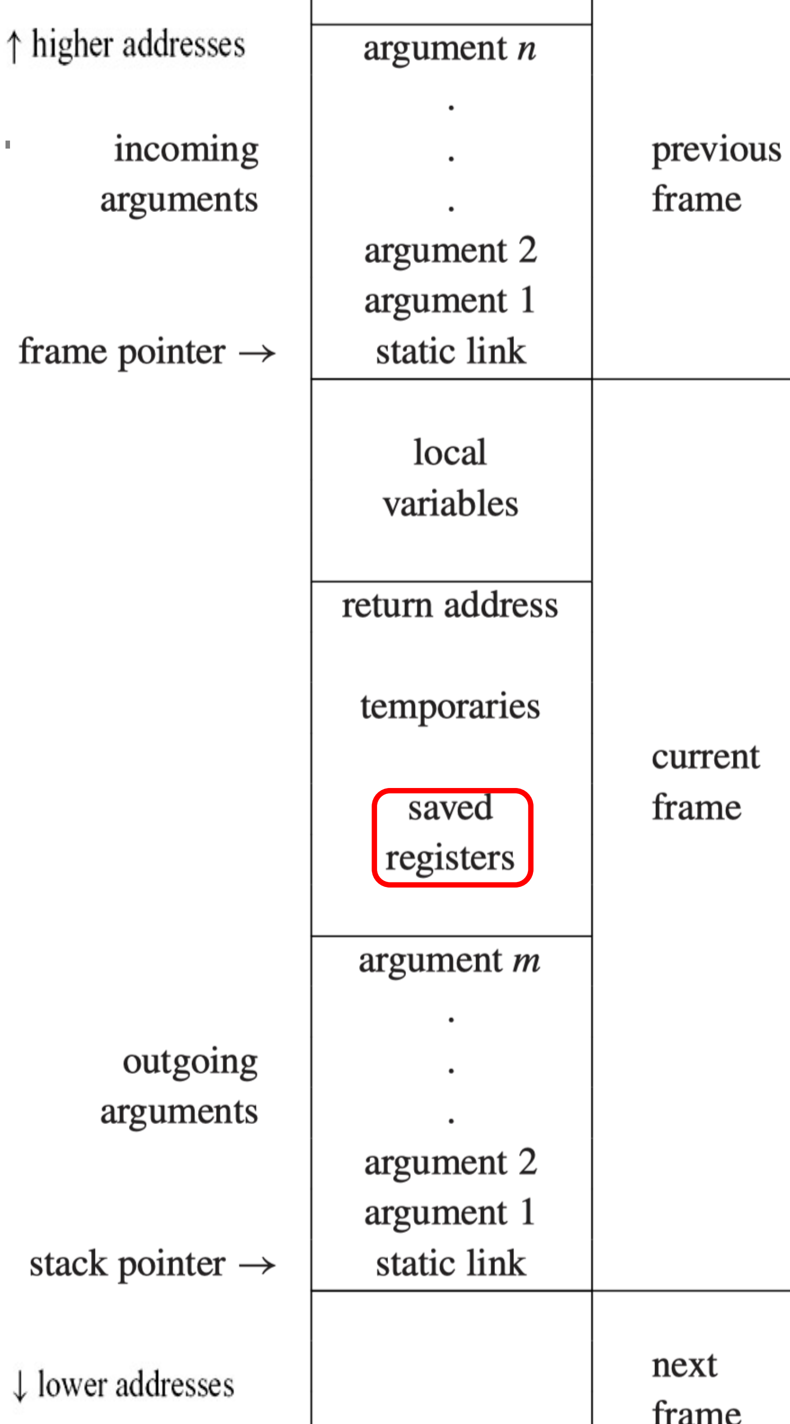
A Typical Stack Frame Layout for Tiger



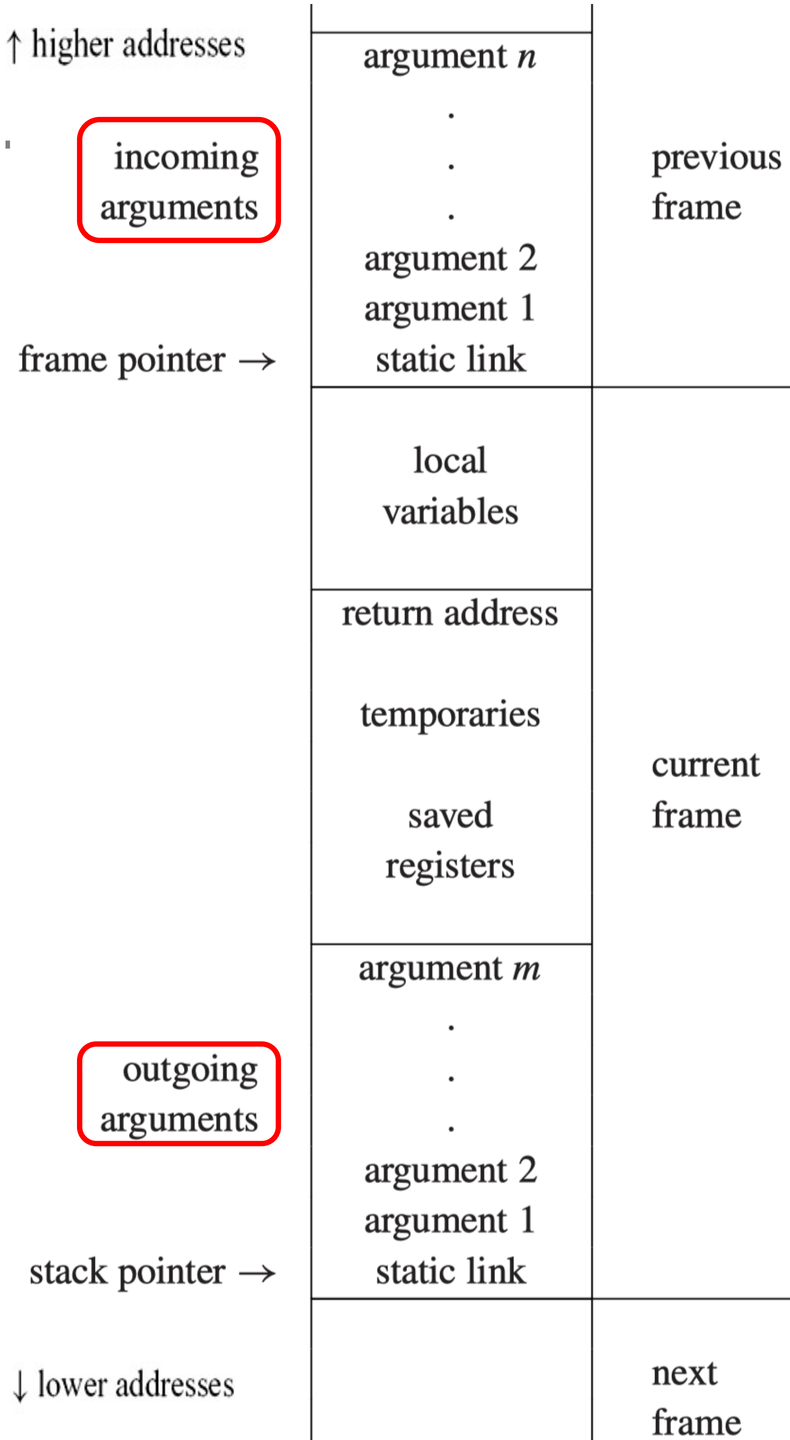
- incoming arguments: passed by the caller
- return address: where (within the calling function) control should return:
  - created by the CALL instruction
- some local variables are in this frame, other local variables are kept in registers
- saved registers: make room for other uses of the registers
- outgoing argument: pass parameters to other functions
- static link



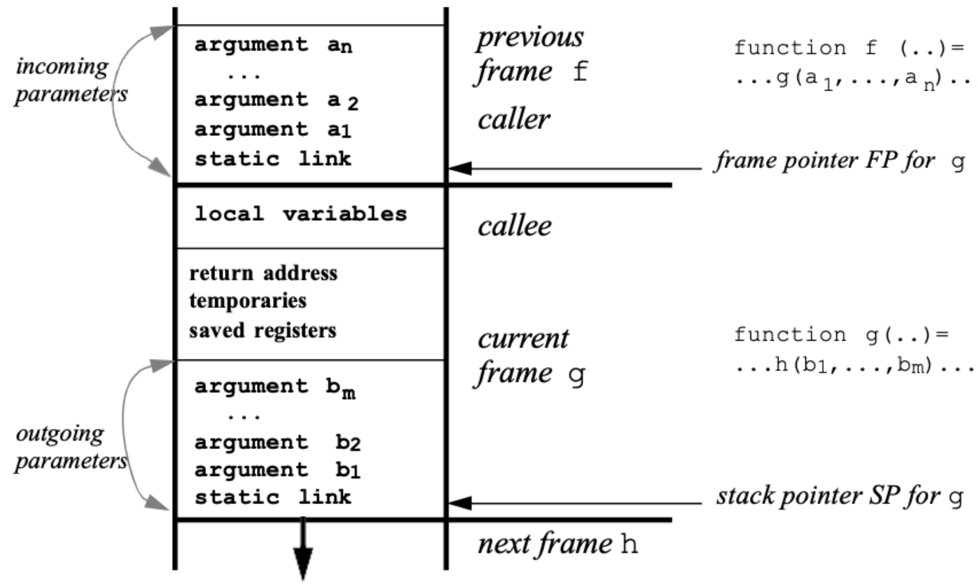
- Frame point为特定寄存器(如rbp, SP)，其值为栈上的内存地址
- 该地址的内存中所保存值是stack link (某个函数的frame point)

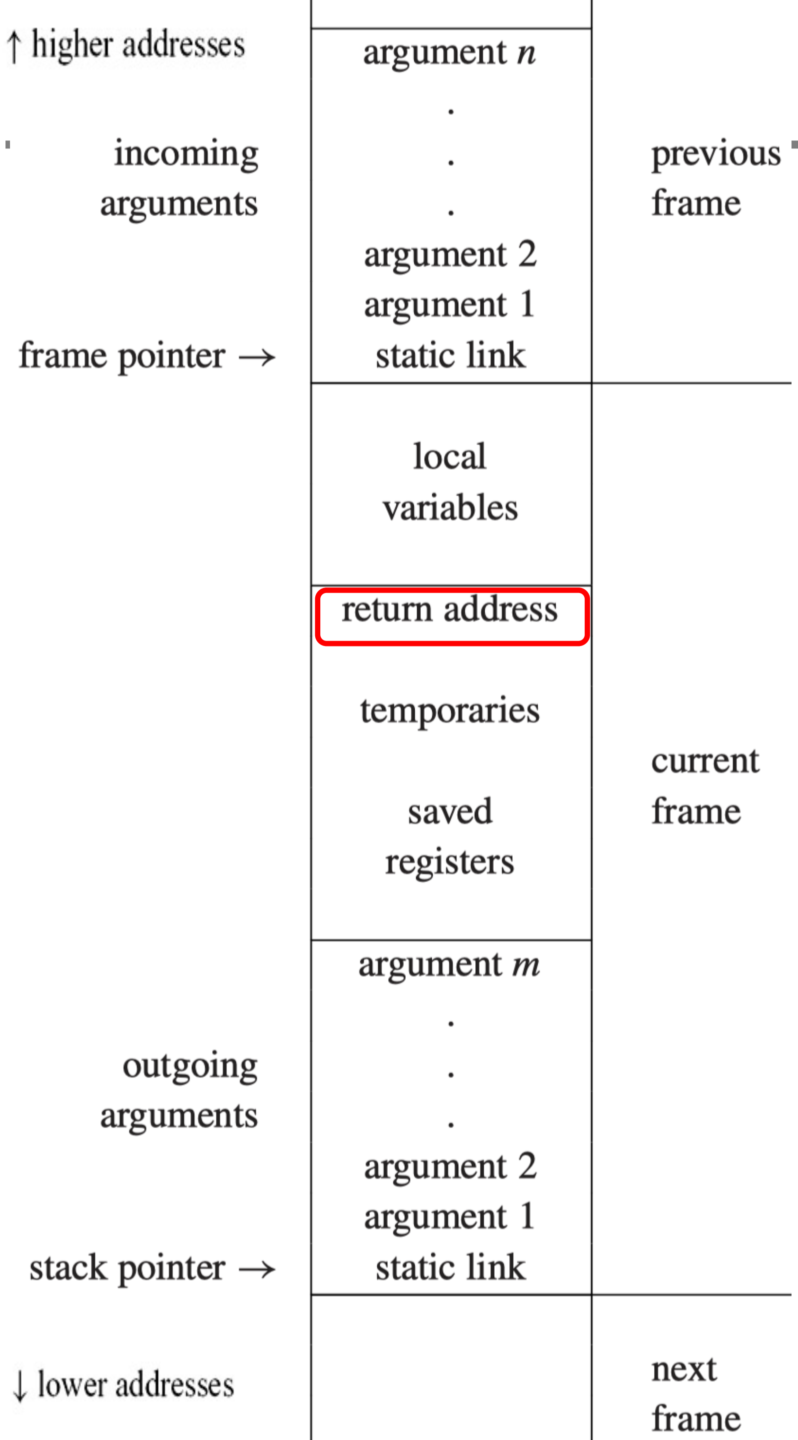


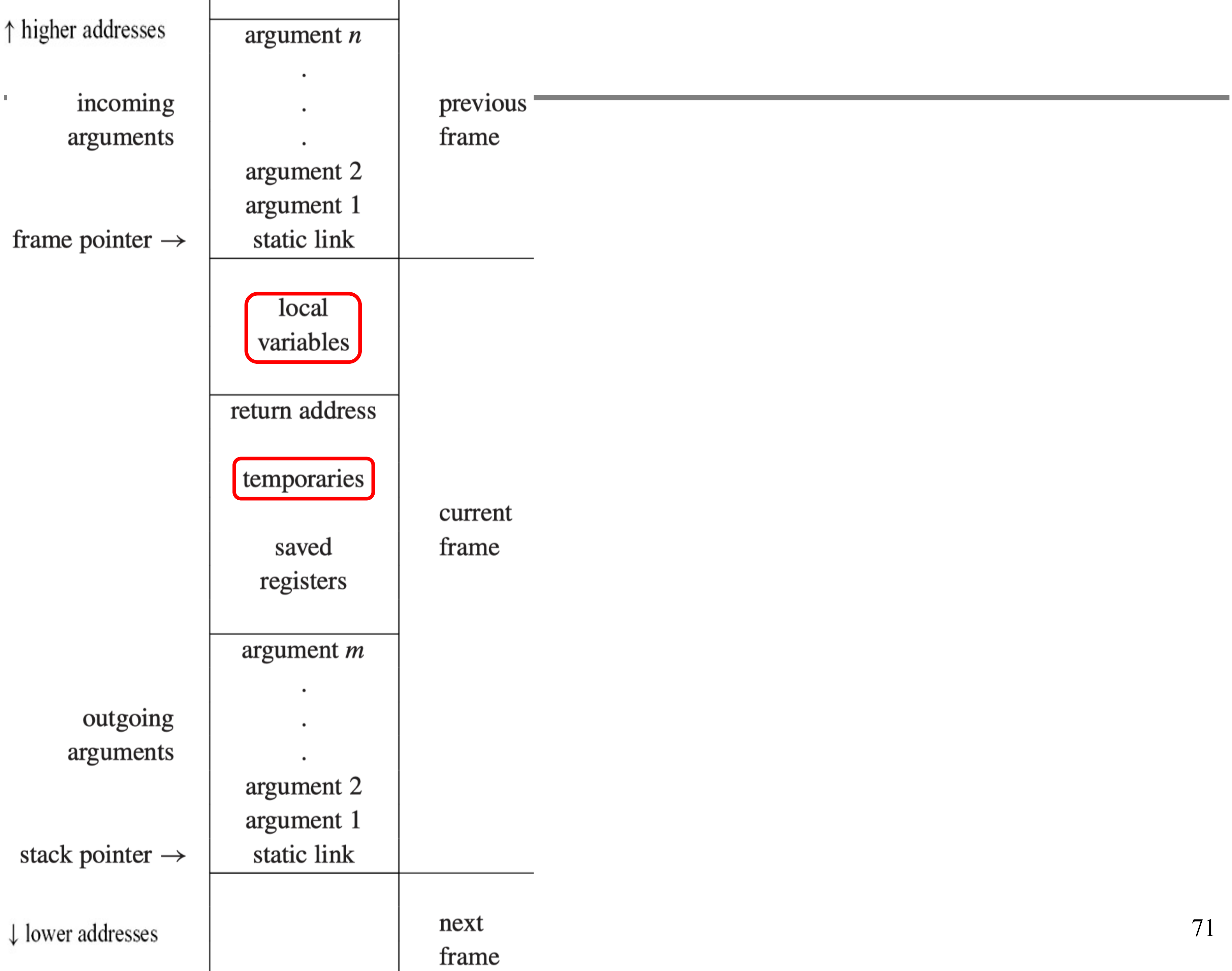
↑ higher addresses
incoming arguments
frame pointer →
stack pointer →
↓ lower addresses
argument n
.
.
.
argument 2
argument 1
static link
local variables
return address
temporaries
saved registers
argument m
.
.
.
argument 2
argument 1
static link
previous frame
current frame
next frame
outgoing arguments

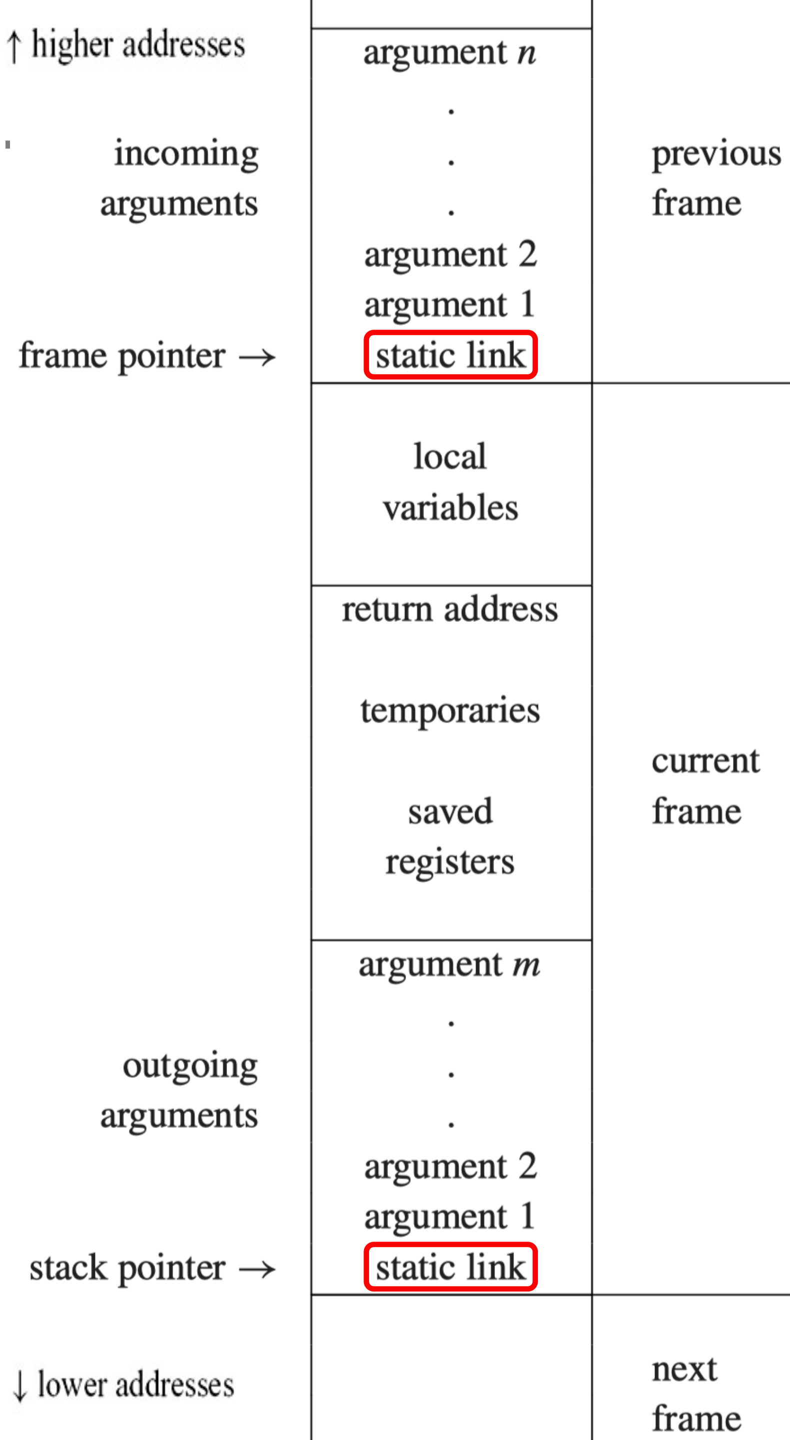

↑ higher addresses
incoming arguments
frame pointer →
outgoing arguments
stack pointer →
↓ lower addresses
argument n
argument 2
argument 1
static link
local variables
return address
temporaries
saved registers
argument m
argument 2
argument 1
static link
previous frame
current frame
next frame

- 一个具体的例子


incoming parameters
outgoing parameters
argument an
...
argument a2
argument a1
static link
local variables
return address
temporaries
saved registers
argument bm
...
argument b2
argument b1
static link
previous frame f
caller
callee
current frame g
next frame h
function f (..)=
...g(a1,...,an)...
frame pointer FP for g
function g(..)=
...h(b1,...,bm)...
stack pointer SP for g


↑ higher addresses
argument n
.
.
.
incoming arguments
argument 2
argument 1
frame pointer →
static link
previous frame
local variables
return address
temporaries
saved registers
current frame
argument m
.
.
.
outgoing arguments
argument 2
argument 1
stack pointer →
static link
↓ lower addresses
next frame


↑ higher addresses
incoming arguments
argument n
argument 2
argument 1
static link
previous frame
frame pointer →
local variables
return address
temporaries
saved registers
current frame
outgoing arguments
argument m
argument 2
argument 1
static link
stack pointer →
next frame
↓ lower addresses



On each procedure call or variable access, a chain of zero or more fetches is required; the length of the chain is just the difference in static nesting depth between the two functions involved.

NOTE: Static links may skip dynamic frames between *f* and *g*. They always point to the most recent frame of the routine that **statically encloses** the current routine.

Limitation of Stack Frame

- **Hard to support higher-order function:** The combination of *nested functions* and *functions as arguments & returns*.

	Pascal , Tiger	C	ML, LISP, Haskell
Nested functions	√	×	√
Procedure passed as arguments and results	×	√	√

- In languages supporting high-order functions, it may be necessary to keep local variables after a function has returned
- But until know, we assume local variables will not be used after *f* returns (so we use the stack)!

Thank you all for your attention

Frames in The Tiger Compiler

6.2 Frames in The Tiger Compiler

- The frame interface will look something like this:

```
/* frame.h */
typedef struct F_frame_ *F_frame;
typedef struct F_access_ *F_access;
typedef struct F_accessList_ *F_accessList;
struct F_accessList_ {F_access head; F_accessList tail;};
F_frame F_newFrame(Temp_label name, U_boolList formals);
Temp_label F_name(F_frame f);
F_accessList F_formals(F_frame f);
F_access F_allocLocal(F_frame f, bool escape);
```

- The abstract interface *frame.h* is implemented by a module specific to the target machine. E.g., *mipsframe.c*

```
/* mipsframe.c */
#include "frame.h"
...
```

Frames in The Tiger Compiler

```
/* frame.h */
typedef struct F_frame_ *F_frame;
F_frame F_newFrame(Temp_label name, U_boolList formals);
Temp_label F_name(F_frame f);
```

- The type *F_frame* holds information about formal parameters and local variables allocated in this frame.
  - U_boolList formals: which parameters escape

```
F_newFrame(g, U_BoolList(TRUE,
                             U_BoolList(FALSE,
                U_BoolList(FALSE, NULL))))
```

Frames in The Tiger Compiler

```
/* frame.h */
typedef struct F_access_ *F_access;
```

- The *F_access* type describes formals and locals that may be in the frame or in registers.
  - An abstract data type. The contents of *struct F_access_* are visible only inside the *Frame* module:
- e.g., *InFrame(X)*, *InReg(t_{84})*

```
/* mipsframe.c */
#include "frame.h"
struct F_access_ {
  enum {inFrame, inReg} kind;
  union {
    int offset; /* InFrame */
    Temp_temp reg; /* InReg */
  } u;
};
static F_access InFrame(int offset);
static F_access InReg(Temp_temp reg);
```

Frames in The Tiger Compiler

```
/* frame.h */
F_accessList F_formals(F_frame f);
```

- The *F_formals* interface function extracts a list of k “accesses” denoting the locations where the formal parameters will be kept at run time, as seen from inside the callee.
- Parameters may be seen differently by the caller and the callee.
  - parameters are passed on the stack
    - caller: offset from the stack pointer
    - callee: offset from the frame pointer
  - parameters are passes through registers, e.g.,
    - caller: register 6
    - callee: register 13
- “Shift of View”

Frames in The Tiger Compiler

- This "shift of view" depends on the calling conventions of the target machine.
- it must be handled by the *Frame* module, starting with *newFrame*.
- For each formal parameter, *newFrame* must calculate two things:
  - How the parameter will be seen from inside the function (in a register, or in a frame location);
  - What instructions must be produced to implement the "view shift."

Representation of Frame Descriptions

- The implementation module *Frame* is supposed to keep the representation of the *F_frame* type secret from any clients of the *Frame* module.
- *F_frame* is a data structure holding:
  - The locations of all the formals,
  - Instructions required to implement the "view shift,"
  - The number of locals allocated so far,
  - The label at which the function's machine code is to begin

Representation of Frame Descriptions

- Suppose function g has three parameters with the first parameter escapes

		Pentium	MIPS	Sparc
Formals	1	`InFrame`(8)	`InFrame`(0)	`InFrame`(68)
	2	`InFrame`(12)	`InReg`(t_{157})	`InReg`(t_{157})
	3	`InFrame`(16)	`InReg`(t_{158})	`InReg`(t_{158})
View Shift		$M[\mathrm{sp}+0] \leftarrow fp$	$\mathrm{sp} \leftarrow \mathrm{sp} - K$	`save %sp,-K,%sp`
		$\mathrm{fp} \leftarrow \mathrm{sp}$	$M[\mathrm{sp}+K+0] \leftarrow$ `r2`	$M[\mathrm{fp}+68] \leftarrow$ `i0`
		$\mathrm{sp} \leftarrow \mathrm{sp} - K$	$t_{157} \leftarrow$ `r4`	$t_{157} \leftarrow$ `i1`
			$t_{158} \leftarrow$ `r5`	$t_{158} \leftarrow$ `i2`

- Why move $r4$ and $r5$ to t_{157} and t_{158}?

```
function m(x:int, y:int) = (h(y,y); h(x,x))
```

Local Variables

- Some local variables are kept in the frame; others are kept in registers.
- To allocate a new local variable, the semantic analysis phase calls

```
F_access F_allocLocal(F_frame f, bool escape);
```

- If *escape = True*, *F_allocLocal* returns an *InFrame* access
- If *escape = False*, *F_allocLocal* can return an *InReg* access
- When to call *F_allocLocal*?

Local Variables

```
function f() =
  let var v := 6
  in (print(v);
      let var v := 7
      in print(v)
      end;
      print(v);
      let var v := 8
      in print(v)
      end;
      print(v))
  end
```

```
void f() {
  int v = 6;
  print(v);
  {int v = 7;
   print(v);}
   print(v);
  {int v = 8;
   print(v);}
   print(v);
}
```

- Three different variables
- Variable-declaration blocks nested inside the body of a function.
- **What is the result?**

- As each variable declaration is encountered during processing, *allocLocal* will be called to allocate a temporary or new space in the frame, associated with the name *v*.
- As each end (or closing brace) is encountered, the association with *v* will be forgotten–but the space is still reserved in the frame.
- A distinct temporary or frame slot for every variable declared within the entire function

Local Variables

```
function f() =
  let var v := 6
  in (print(v);
      let var v := 7
      in print(v)
      end;
      print(v);
      let var v := 8
      in print(v)
      end;
      print(v))
  end
```

```
void f() {
  int v = 6;
  print(v);
  {int v = 7;
   print(v);}
   print(v);
  {int v = 8;
   print(v);}
   print(v);
}
```

- Three different variables
- Variable-declaration blocks nested inside the body of a function.
- print 6 7 6 8 6

- The register allocator will use as few registers as possible to represent the temporaries.
  - The second and third v variables could be held in the same temporary
- A clever compiler might also notice two frame-resident variables could be allocated to the same slot.

Calculating Escapes

- When calling *allocLocal*, it is important to know whether the variable escapes or not.
- A *findEscape* function can look for escaping variables and record this information in the escape fields of the abstract syntax.
- How to implement *findEscape*?
  - Traverse the entire abstract syntax tree, looking for escaping uses of every variable.
  - Use environments to record whether the particular variable escapes.

```
/* escape.h */
void Esc_findEscape(A_exp exp);
/* escape.c */
static void traverseExp(S_table env, int depth, A_exp e);
static void traverseDec(S_table env, int depth, A_dec d);
static void traverseVar(S_table env, int depth, A_var v);
```

Calculating Escapes

For example

- Whenever a variable or formal-parameter declaration *x* is found at static function-nesting depth *d*, e.g.,:

```
x = A_VarDec(pos, symbol("a"), typ, init, escape)
```

- enter *<a, EscapeEntry(d, &(x->u.var.escape))>* into the environment
- whenever *a* is used at depth > *d*, set x->u.var.escape = True

- Other situations (variable addresses are taken explicitly or there are call-by-reference parameters) are similar.

Temporaries and Labels

- The compiler's semantic analysis phase will want to choose registers for parameters and local variables, and choose machine-code addresses for procedure bodies.
- But it is too early to determine them exactly.
- **Temporary**: a value that is temporarily held in a register
- **Label**: some machine-language location whose exact address is yet to be determined

Temporaries and Labels

- *Temps*: abstract names for local variables
- *Labels*: abstract names for static memory addresses

```
/* temp.h */
typedef struct Temp_temp_ *Temp_temp;
Temp_temp Temp_newtemp(void);

typedef S_symbol Temp_label;
Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

typedef struct Temp_tempList_ *Temp_tempList;
struct Temp_tempList_ {Temp_temp head; Temp_tempList tail;}
Temp_tempList Temp_TempList(Temp_temp head, Temp_tempList
tail);
typedef struct Temp_labelList_ *Temp_labelList;
struct Temp_labelList_{Temp_label head; Temp_labelList tail;}
Temp_labelList Temp_LabelList(Temp_label head, Temp_labelList
tail);
```

Temporaries and Labels

- *Temps*: abstract names for local variables
- *Labels*: abstract names for static memory addresses

```
/* temp.h */
typedef struct Temp_temp_ *Temp_temp;
Temp_temp Temp_newtemp(void);

typedef S_symbol Temp_label;
Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

typedef struct Temp_tempList_ *Temp_tempList;
struct Temp_tempList_ {Temp_temp head; Temp_tempList tail;}
Temp_tempList Temp_TempList(Temp_temp head, Temp_tempList
tail);
typedef struct Temp_labelList_ *Temp_labelList;
struct Temp_labelList_{Temp_label head; Temp_labelList tail;}
Temp_labelList Temp_LabelList(Temp_label head, Temp_labelList
tail);
```

Temporaries and Labels

- *Temps*: abstract names for local variables
- *Labels*: abstract names for static memory addresses

```
/* temp.h */
typedef struct Temp_temp_ *Temp_temp;
Temp_temp Temp_newtemp(void);

typedef S_symbol Temp_label;
Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

typedef struct Temp_tempList_ *Temp_tempList;
struct Temp_tempList_ {Temp_temp head; Temp_tempList tail;}
Temp_tempList Temp_TempList(Temp_temp head, Temp_tempList
tail);
typedef struct Temp_labelList_ *Temp_labelList;
struct Temp_labelList_{Temp_label head; Temp_labelList tail;}
Temp_labelList Temp_LabelList(Temp_label head, Temp_labelList
tail);
```

There could be different functions with the same name in different scopes

Temporaries and Labels

- *Temps*: abstract names for local variables
- *Labels*: abstract names for static memory addresses

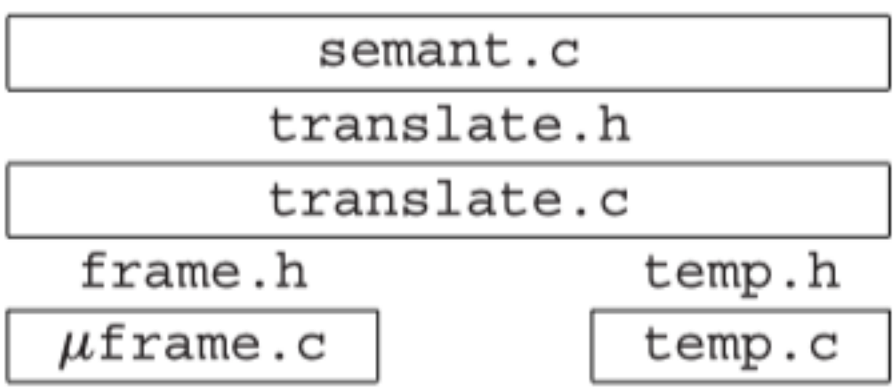
```
/* temp.h */
typedef struct Temp_temp_ *Temp_temp;
Temp_temp Temp_newtemp(void);

typedef S_symbol Temp_label;
Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

typedef struct Temp_tempList_ *Temp_tempList;
struct Temp_tempList_ {Temp_temp head; Temp_tempList tail;}
Temp_tempList Temp_TempList(Temp_temp head, Temp_tempList
tail);
typedef struct Temp_labelList_ *Temp_labelList;
struct Temp_labelList_{Temp_label head; Temp_labelList tail;}
Temp_labelList Temp_LabelList(Temp_label head, Temp_labelList
tail);
```

Two Layers of Abstraction

- The *frame.h* and *temp.h* interfaces provide machine-independent views of memory-resident and register-resident variables.
  - We need not care where variables are exactly stored.
- The Translate module augments this by handling the notion of nested scopes (via static links), providing the interface translate.h to the Semant module.
- Why named Translate?



Two Layers of Abstraction

```
/* translate.h */
typedef struct Tr_access_ *Tr_access;
typedef ... Tr_accessList ...
Tr_accessList Tr_AccessList(Tr_access head, Tr_accessList tail);
Tr_level Tr_outermost(void);
Tr_level Tr_newLevel(Tr_level parent, Temp_label name,
U_boolList formals);
Tr_accessList Tr_formals(Tr_level level);
Tr_access Tr_allocLocal(Tr_level level, bool escape);
```

- The goal: handling nesting scopes
- Why do we need to handle nesting scopes?

Two Layers of Abstraction

```
type tree = {key: string, left: tree, right: tree}
function prettyprint(tree: tree) : string =
  let
    var output := “”
    function write(s: string) =
      output := concat(output,s)

    function show(n:int, t: tree) =
      let function indent(s: string) =
            (for i := 1 to n
             do write(“ ”));
             output := concat(output, s);
             write("\n"))
      in if t=nil
         then indent(".")
         else (indent(t.key));
               show(n+1, t.left);
               show(n+1, t.right))
      end
    in show(0, tree); output
  end
```

frame of *f1*
frame of *f2*
frame of *f3*
frame of *indent*

Two Layers of Abstraction

```c
/* translate.h */
typedef struct Tr_access_ *Tr_access;
typedef ... Tr_accessList ...
Tr_accessList Tr_AccessList(Tr_access head, Tr_accessList tail);
Tr_level Tr_outermost(void);
Tr_level Tr_newLevel(Tr_level parent, Temp_label name,
U_boolList formals);
Tr_accessList Tr_formals(Tr_level level);
Tr_access Tr_allocLocal(Tr_level level, bool escape);
```

- The goal: handling nesting scopes
- Why do we need to handle nesting scopes?
  - for implementing block structure
  - for calculating escaping variables
- How to achieve this goal?
  - create nesting levels
  - associate a nesting level to each function and each variable

Two Layers of Abstraction

```
/* translate.h */
...
Tr_level Tr_newLevel(Tr_level parent, Temp_label name,
U_boolList formals);
...
Tr_access Tr_allocLocal(Tr_level level, bool escape);
```

- When and how to create nesting levels?
  - *transDec* creates a new "nesting level" for each function by calling *Tr_newLevel*
- How to associate a nesting level to each function?
  - keep the nesting level of each function in its FunEntry (stored in the environment)
- How to associate a nesting level to each variable?
  - When Semant processes a local variable declaration at level *lev*, it calls *Tr_allocLocal(lev,esc)* to create the variable in this level
  - Semant records *Tr_access* in each *VarEntry* in the value env

Two Layers of Abstraction

```
/* new versions of VarEntry and FunEntry */
struct E_enventry_ {
  enum {E_varEntry, E_funEntry} kind;
  union {
    struct {Tr_access access; Ty_ty ty;} var;
    struct {Tr_level level; Temp_label label;
            Ty_tyList formals; Ty_ty result;} fun;
  } u;
};

E_enventry E_VarEntry(Tr_access access, Ty_ty ty);
E_enventry E_FunEntry(Tr_level level, Temp_label label,
Ty_tyList formals, Ty_ty result);

/* inside translate.c */
struct Tr_access_ {Tr_level level; F_access access;};
```

Managing Static Links

- We use the Translate module to manage static links.
- Why not use the Frame module to manage them?
  - Frame should be independent of the specific source language being compiled
  - Many source languages do not have nested function declarations.
- Translate knows that each frame contains a static link. The static link is passed to a function in a register and stored into the frame.
  - just like a parameter
- We will treat the static link as a parameter (as much as possible)

Managing Static Links

```
/* translate.h */
typedef struct Tr_access_ *Tr_access;
typedef ... Tr_accessList ...
Tr_accessList Tr_AccessList(Tr_access head, Tr_accessList tail);
...
Tr_accessList Tr_formals(Tr_level level);
...
```

- When Semant calls *Tr_formals(level)*, it will get the *access* values of the original parameters.

Keeping Track of Levels

```
/* translate.h */
typedef struct Tr_access_ *Tr_access;
typedef ... Tr_accessList ...
Tr_accessList Tr_AccessList(Tr_access head, Tr_accessList tail);
Tr_level Tr_outermost(void);
Tr_level Tr_newLevel(Tr_level parent, Temp_label name,
U_boolList formals);
Tr_accessList Tr_formals(Tr_level level);
Tr_access Tr_allocLocal(Tr_level level, bool escape);
```

- *Tr_outermost*: returns the outermost level
- The level within which the Tiger main program is nested
- All "library" functions are declared at this outermost level, which does not contain a frame or formal parameter list.