The HW/SW Interface

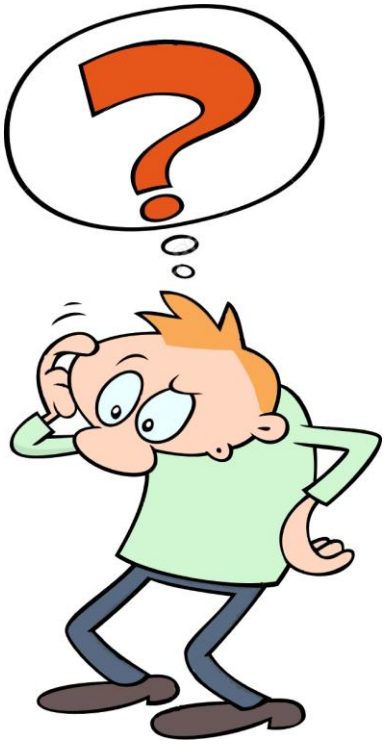
RISC-V Procedures

x0	hard-wired zero
ra (x1)	Caller
sp (x2)	Callee
gp (x3)	-
tp (x4)	-

t0-2 (x5-7)	Caller
s0,1 (x8,9)	Callee
a0-7 (x10-17)	Caller
s2-s11 (x18-27)	Callee
t3-6 (x28-31)	Caller

Module Outline

- Problem Definition
- The Runtime Stack
- Solving Control Transfer
- Solving Parameter Passing
- Solving Local Storage Allocation
- The Calling Convention
- The Runtime Stack and Stack-Based Language
- Module Summary



Problem Definition

Calling Procedures/Functions/Methods

```
…
s = sumto(DATA, a1, a2);
…
```

```
long sumto(long *a, int from, int to)
{
  long sum = 0;
  int i;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```

Calling Procedures/Functions/Methods

- **Problems to solve**

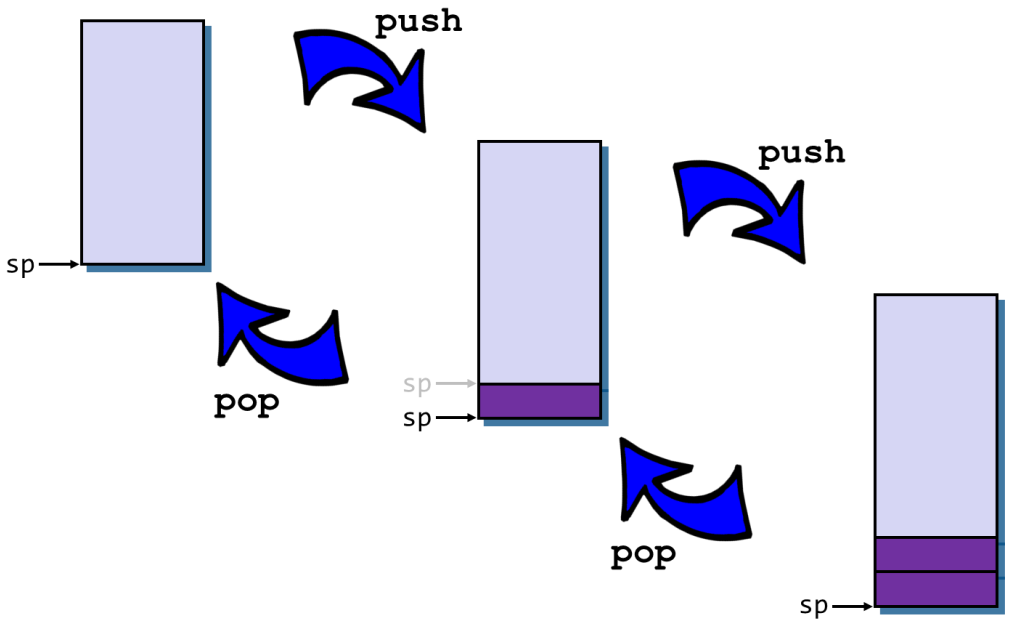
1. **Control transfer**
   pass control to `sumto` when the function is invoked, return to the calling code when `sumto` ends

2. **Parameter passing**
   pass arguments in caller to `sumto` such that `sumto` can access them. `sumto` needs to pass a return value back to the caller

3. **Storage for local variables**
   allow `sumto` to store and access local variables for the duration of its execution

```
…
s = sumto(DATA, a1, a2);
…
```

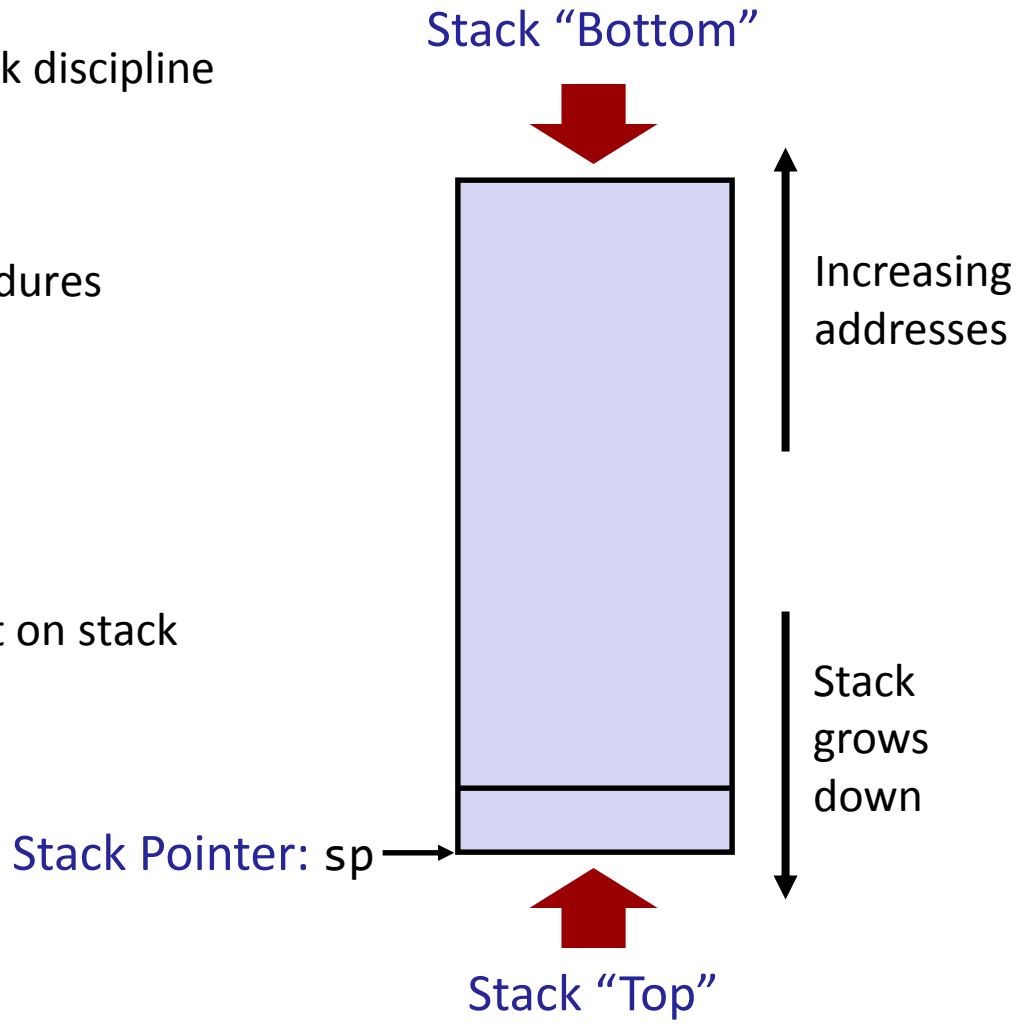
```
long sumto(long *a, int from, int to)
{
  long sum = 0;
  int i;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```



The Runtime Stack

The Runtime Stack

- Region of memory managed with stack discipline (last in, first out)
- Provides temporary storage for procedures
- Grows toward lower addresses (for historical reasons)
- Register sp (x2) points to top element on stack



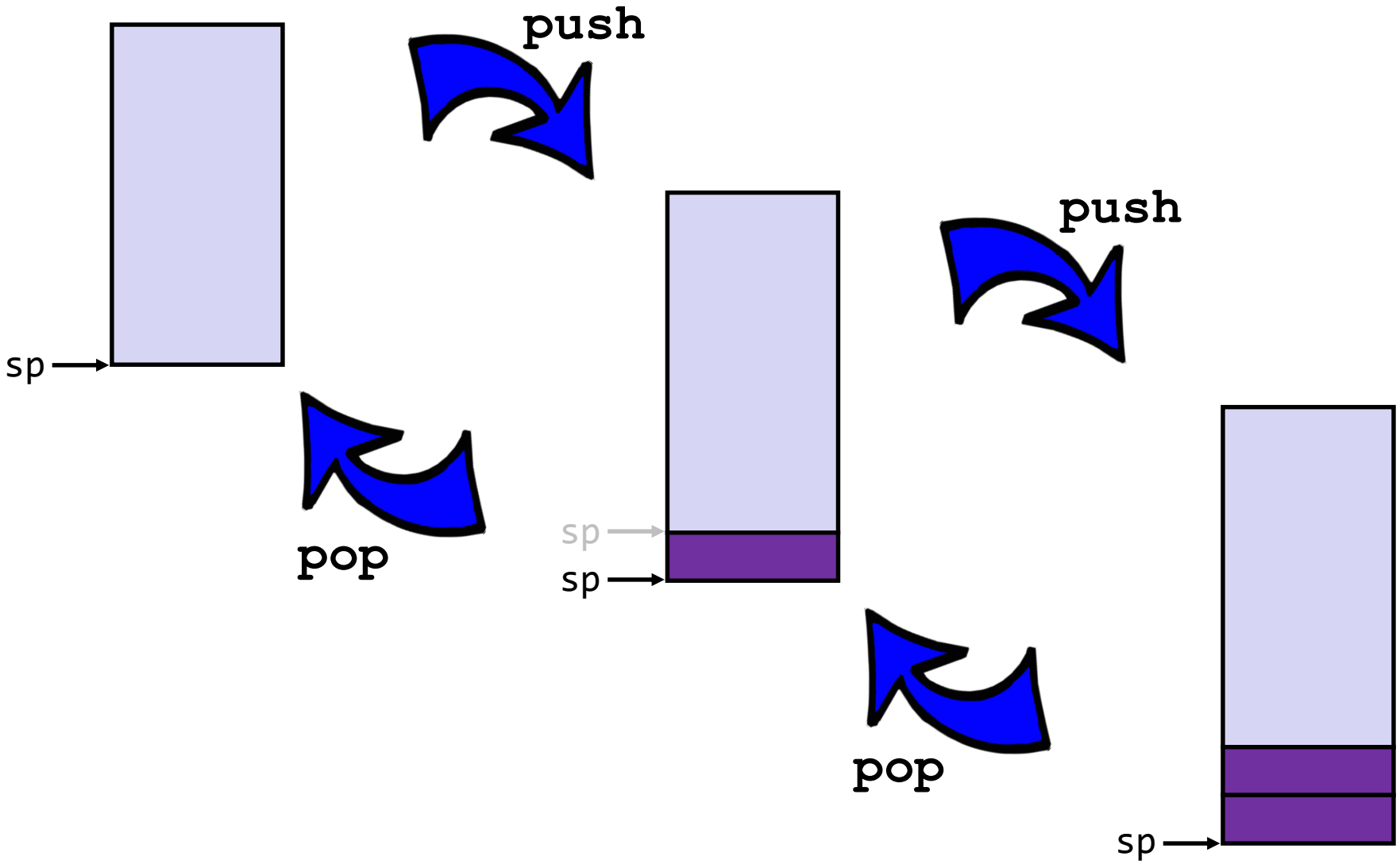
Pushing and Popping Data

- **Push operation**
  - push a register on top of the stack
  - two part operation
    1. decrease stack pointer
    2. store element at `sp`

- **Pop operation**
  - pop the topmost element on the stack into a register
  - inverse of push
    1. load element at `sp`
    2. increment stack pointer

- **Many architectures have explicit push/pop instructions**
  - RISC-V doesn't

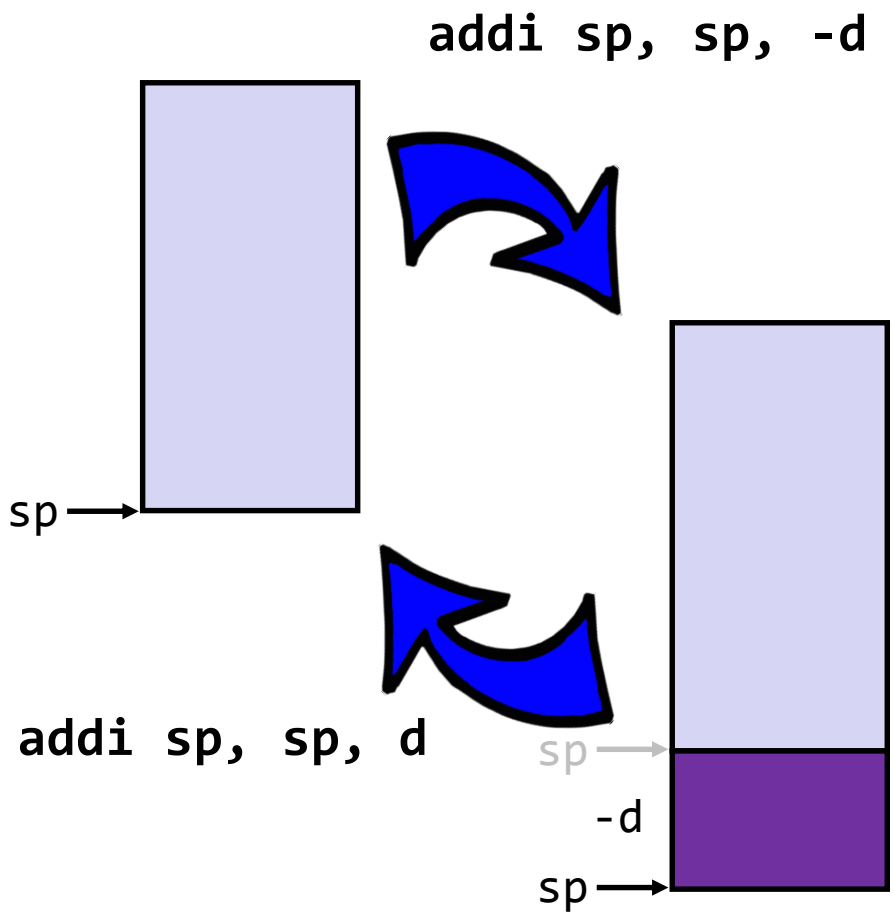
Pushing and Popping Data



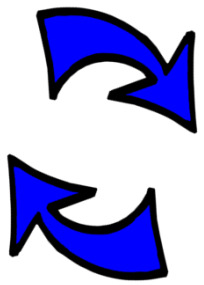
Allocating / Deallocating Memory on the Stack

- `addi sp, sp, -<amount>`
  - Decrement sp by amount
- `addi sp, sp, <amount>`
  - Increment sp by amount
- Coalesce multiple stack operations

```
addi  sp, sp, -24
sd    ra, 0(sp)
sd    fp, 8(sp)
sd    x9, 16(sp)
…
ld    x9, 16(sp)
ld    fp, 8(sp)
ld    ra, 0(sp)
addi  sp, sp, 24
```



```
jal ra, <label>
```



```
jalr x0, 0(ra)
```

Solving Control Transfer

Control Transfer: Naïve Approach

```
void foo(…)
{
  …
  s = sumto(DATA, a1, a2);
  …
}
```

```
00010188 <foo>:
 10188: addi  sp,sp,-32
 …
 1019c: ld    a2,0(sp)
 101a0: ld    a1,8(sp)
 101a4: addi  a0,gp,-104 # 11c68 <DATA>
 101a8: beq   x0, x0, <sumto>          # goto sumto
 101ac: ld    ra,24(sp)
 …
```

```
long sumto(long *a,
       int from, int to)
{
  long i, sum = 0;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```

```
00010160 <sumto>:
 10160: mv    a4,a0
 …
 10178: add   a0,a0,a5
 1017c: addi  a1,a1,1
 10180: j     10168 <sumto+0x8>
 10184: beq   x0, x0, 0x101ac      # go back to foo
```

Control Transfer: Why it doesn't work

```
void foo(…)
{
  …
  s = sumto(DATA, a1, a2);
  …
}
```

```
00010188 <foo>:
 …
 101a8: beq    x0, x0, <sumto>
 101ac: ld     ra,24(sp)
 …
```

```
void bar(…)
{
  …
  res = sumto(arr, 0, 5);
  …
}
```

```
00012248 <bar>:
 …
 12268: beq    x0, x0, <sumto>
 1226c: ld     ra,24(sp)
 …
```

```
long sumto(long *a,
       int from, int to)
{
  long i, sum = 0;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```

```
0000000000000044 <sumto>:
  44:       addi    sp,sp,-64
  ...
  a8:       mv      a0,a5
  ac:       ld      s0,56(sp)
  b0:       addi    sp,sp,64
  b4:       beq     x0, x0, 101ac or 1226c ??
```

Solving Procedure Control Flow

- **Store the return address whenever a procedure is called**
  - caller stores return address at known location
  - callee sets PC to return address
  - works nicely also for nested procedure calls

- **Architectural support**
  - **Invoking a procedure: `call <label>`**
    - store address of next instruction into known location
    - continue program at <procedure>
      `PC = <label>`
  - **Returning from a procedure: `ret`**
    - load return address from known location into PC
      `PC = <return address>`

RISC-V Procedure Call Instructions

- **Procedure call: jump and link**
  - Address of following instruction stored in register `ra (return address, x1)`
  - PC = <label>

```
jal  ra, <label>
```

- **Procedure return: jump and link register**
  - Like `jal`, but jumps to 0 + address in `ra`
  - Use `x0` as `rd` (i.e., does not link)

```
jalr x0, 0(ra)
```

- **Special uses**
  - jal x0, <label> used for unconditional branches
  - jalr used for computed jumps in switch statements

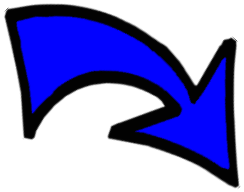
Solving Procedure Control Flow

```
void foo(…)
{
  …
  s = sumto(DATA, a1, a2);
  …
}
```

```
00010188 <foo>:
 …
 101a8: jal   ra,10160 <sumto>
 101ac: ld    ra,24(sp)
 …
```

zero (x0)	
ra (x1)	101ac
x2	
x3	
x4	
…	
pc	10160

jal ra, <label>



jalr x0, 0(ra)

zero (x0)	
ra (x1)	101ac
x2	
…	
pc	101ac

```
long sumto(long *a,…)
{
  long sum = 0;
  …
  return sum;
}
```

```
00010160 <sumto>:
 …
 10180: j     10168
 10184: jalr  x0, 0(ra)
```

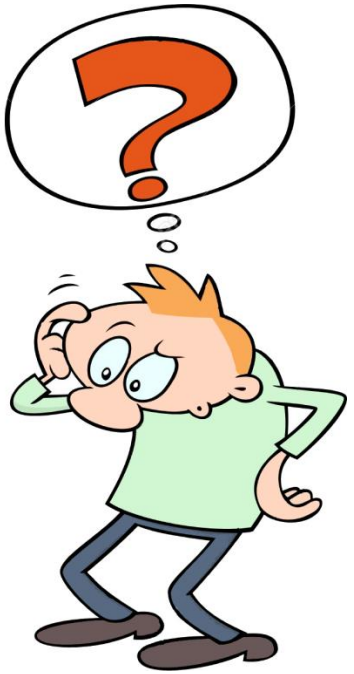
RISC-V Jump and Link Instructions

- **Reality check**

```
$ riscv64-unknown-elf-gcc -march=rv64g -mabi=lp64d -Og -S sumto.c
$ riscv64-unknown-elf-gcc -march=rv64g -mabi=lp64d -Og -o sumto sumto.c
$ riscv64-unknown-elf-objdump -d sumto.o > sumto.dis
```

```
foo:
  addi   sp,sp,-32
  sd     ra,24(sp)
  addi   a1,sp,8
  mv     a0,sp
  call   getparm
  ld     a2,0(sp)
  ld     a1,8(sp)
  lui    a0,%hi(DATA)
  addi   a0,a0,%lo(DATA)
  call   sumto
  ld     ra,24(sp)
  addi   sp,sp,32
  jr     ra
```
sumto.s

```
00010184 <foo>:
 10184: addi   sp,sp,-32
 10188: sd     ra,24(sp)
 1018c: addi   a1,sp,8
 10190: mv     a0,sp
 10194  auipc  ra,0x0
 10198: jalr   ra # 10198 <foo+0x14>
 1019c: ld     a2,0(sp)
 101a0: ld     a1,8(sp)
 101a4: addi   a0,gp,-104 # 11c68 <DATA>
 101a8: jal    ra,10160 <sumto>
 101ac: ld     ra,24(sp)
 101b0: addi   sp,sp,32
 101b4: ret
```
sumto.dis

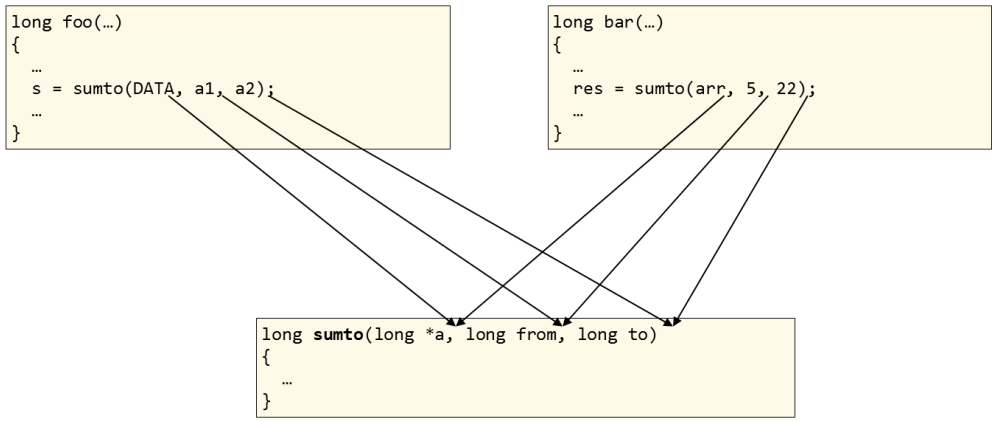


RISC-V Jump and Link Instructions

- **Pseudoinstruction `call <label>` to implement calls**
  - translated by assembler/linker into actual instruction sequence
  - target address encoded as an **offset relative to program counter**
  - offset resolved when assembling/linking the executable
    - short call: target offset +- 1MiB (20 bit signed * 2)
      ```
      jal   ra, <offset>
      ```
    - far call: far or unknown targets
      ```
      auipc ra, <bits 32:12 of offset>
      jalr  ra, <bits 11:0 of offset>
      ```
    - show relocations with
      ```
      $ objdump –dr
      ```

- **Pseudoinstructions `jr <reg>/ret` to return from calls**
  - pseudoinstruction for
    ```
    jalr x0, 0(ra)
    ```

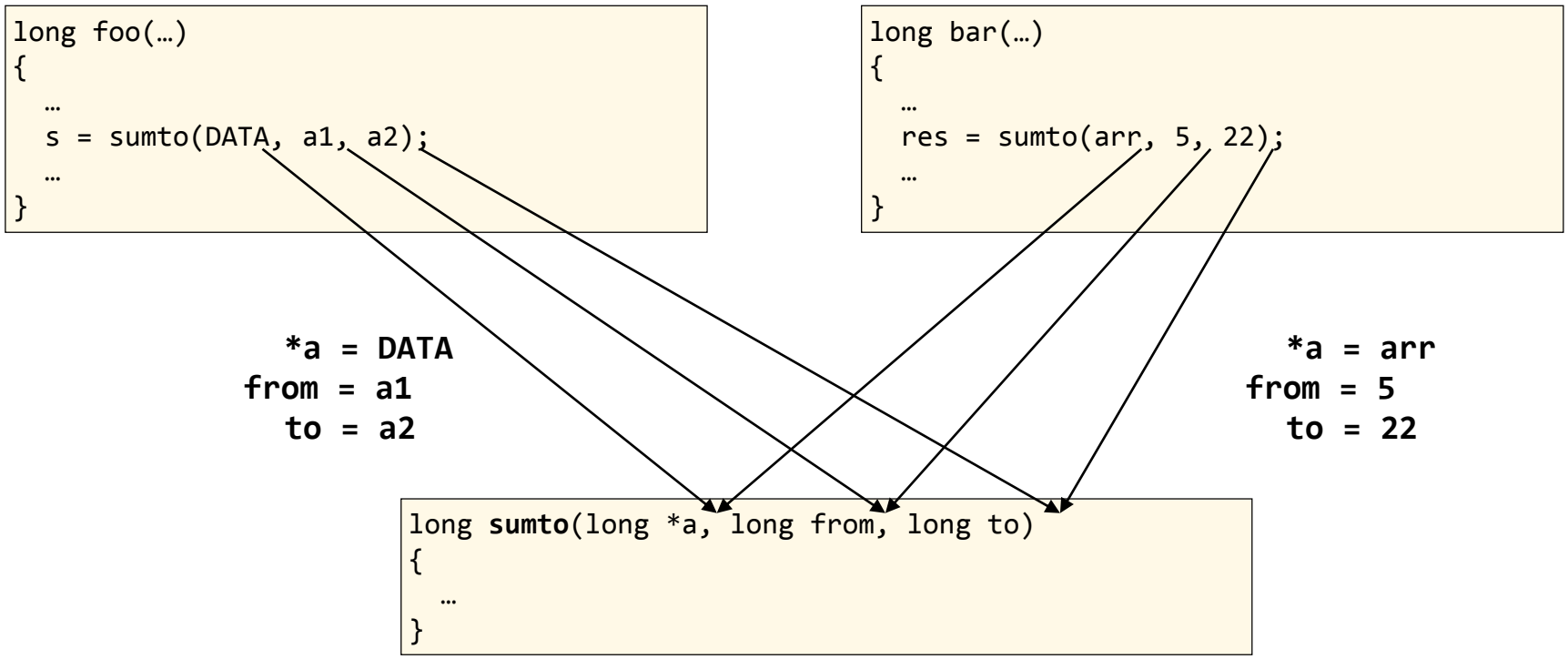
```
00010184 <foo>:
 10184: addi  sp,sp,-32
 10188: sd    ra,24(sp)
 1018c: addi  a1,sp,8
 10190: mv    a0,sp
 10194  auipc ra,0x0
 10198: jalr  ra # 10198 <foo+0x14>
 1019c: ld    a2,0(sp)
 101a0: ld    a1,8(sp)
 101a4: addi  a0,gp,-104 # 11c68 <DATA>
 101a8: jal   ra,10160 <sumto>
 101ac: ld    ra,24(sp)
 101b0: addi  sp,sp,32
 101b4: ret                    sumto.dis
```



Solving Parameter Passing

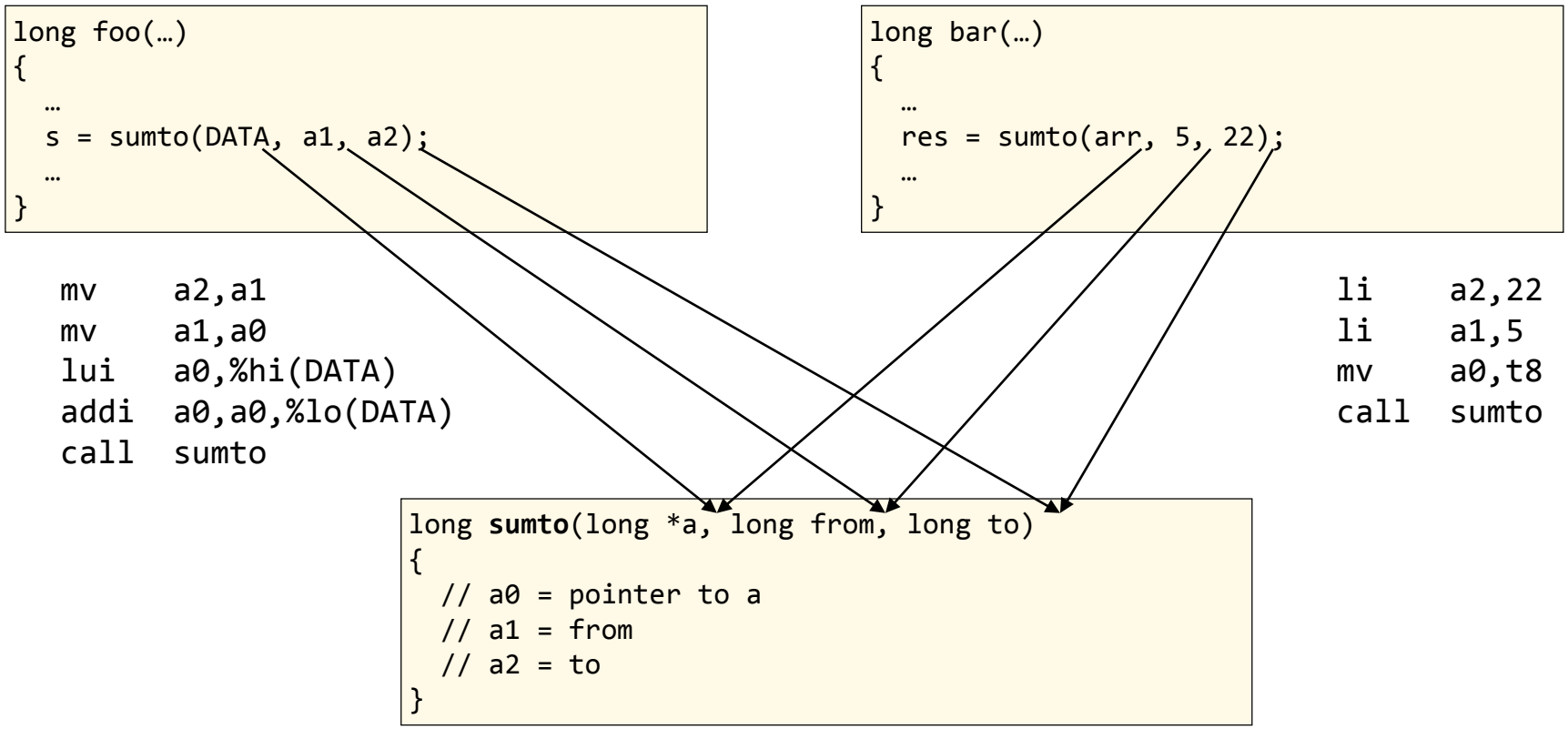
Parameter Passing

- Need a mapping between arguments and parameters



Solving Parameter Passing

- **Pass parameters in registers and on the runtime stack**
  - need a convention that defines which parameter maps to which register
  - RISC-V: pass first 8 parameters in registers a0-a7 (x10-x17), parameters >8 on stack



```
locals:
      addi  sp,sp,-208    # make room on stack
      mv    a0,sp         # a0 = sp
      addi  a1,sp,8       # a1 = sp+8
      sd    s0,192(sp)    # save s0
      sd    s1,184(sp)    # save s1
      sd    ra,200(sp)    # save ra
      …
      ld    ra,200(sp)    # restore ra
      ld    s0,192(sp)    # restore s0
      ld    s1,184(sp)    # restore s1
      addi  sp,sp,208     # restore sp
      jr    ra            # return
```

Solving Local Storage Allocation

Where do Local Variables Go?

```
long locals(void)
{
  long a, b;
  long from, to, sum=0;
  long array[20];

  init_ab(&a, &b);

  from = a+b;
  to   = 3*a + 2*b;

  init_array(array);

  for (long i=from; i<to; i++) {
    sum += array[i];
  }

  return sum;
}
```

Where do Local Variables Go?

- Could try to allocate local variables to a (fixed) memory address

```
long locals(void)
{
  long a, b;
  long from, to, sum=0;
  long array[20];

  init_ab(&a, &b);

  from = a+b;
  to   = 3*a + 2*b;

  init_array(array);

  for (long i=from; i<to; i++) {
    sum += array[i];
  }

  return sum;
}
```

```
0x00010788: a
0x00010790: b
0x00010798: from
0x000107a0: to
…
```

```
        …
        ld      a4,%lo(from.1508)(s0)
.L2:
        lui     a5,%hi(to.1509)
        ld      a5,%lo(to.1509)(a5)
        ble     a5,a4,.L5
        …
```

Local Variable Mapping

- **Fails for recursive procedures**

```
int foo(int n)
{
  int a, b = 1;

  for (a=0; a<n; a++) {
    b = b + foo(n-1);
  }

  return b;
}
```

```
0x00010788: a
0x00010790: b
```

```
foo(2):
  b=1;
  a=0;
  a<n? yes: b=b + foo(1):
                    b=1;
                    a=0;
                    a<n? yes: b=b + foo(0)
                                        b=1;
                                        a=0;
                                        a<n? no
                                        return b (=1)
                                  b=1 + 1 = 2
                    a++ (=1)
                    a<n? no
                    return b (=2)
              =1 + 2 = 3
  a++ (=2)
  a<n? NO!
```

Solving Local Variable Mapping

- **Allocate on runtime stack**

```
long locals(void)
{
  long a, b;
  long from, to, sum=0;
  long array[20];

  init_ab(&a, &b);

  from = a+b;
  to   = 3*a + 2*b;

  init_array(array);

  for (long i=from; i<to; i++) {
    sum += array[i];
  }

  return sum;
}
```

- **Observations about locals**
  - some on stack (a,b, array)
  - some in registers (from, to, sum)
  - some eliminated (i)

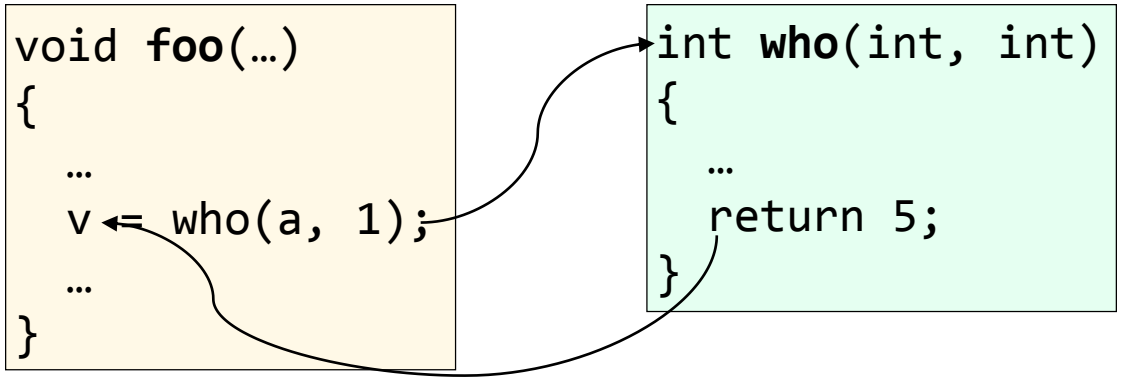
```
locals:
      addi  sp,sp,-208    # make room on stack
      mv    a0,sp         # a0 = sp               (= &a, a at mem[sp+0])
      addi  a1,sp,8       # a1 = sp+8             (= &b, b at mem[sp+8])
      sd    s0,192(sp)    # save s0
      sd    s1,184(sp)    # save s1
      sd    ra,200(sp)    # save ra
      call  init_ab       # init_ab(&a, &b)
      ld    a5,0(sp)      # a5 = a
      ld    s0,8(sp)      # s0 = b
      addi  a0,sp,16      # a0 = sp+16            (=&array)
      slli  s1,a5,1       # s1 = a<<1 = 2*a
      slli  a4,s0,1       # a4 = b<<1 = 2*b
      add   s1,s1,a5      # s1 = 2*a + a = 3*a
      add   s0,a5,s0      # s0 = a + b            (=from)
      add   s1,s1,a4      # s1 = 3*a + 2*b        (=to)
      call  init_array    # init_array(&array)
      bge   s0,s1,.L4     # to >= from ? goto .L4 (skip if nothing to do)
      addi  a3,sp,16      # a3 = sp+16            (=&array)
      slli  a5,s0,3       # a5 = from<<3 = from*8 (=offset into array[from])
      slli  a4,s1,3       # a4 = to<<3 = to*8     (=offset into array[to])
      add   a5,a3,a5      # a5 = &array[from]
      add   a4,a4,a3      # a4 = &array[to]
      li    a0,0          # a0 = 0                (=sum)
.L3:  ld    a3,0(a5)      # a3 = array[from]
      addi  a5,a5,8       # a5 = from+8           (=offset of next element)
      add   a0,a0,a3      # a0 = a0 + array[from]
      bne   a5,a4,.L3     # from != to? goto .L3
      ld    ra,200(sp)    # restore ra
      ld    s0,192(sp)    # restore s0
      ld    s1,184(sp)    # restore s1
      addi  sp,sp,208     # restore sp
      jr    ra            # return
.L4:  ld    ra,200(sp)    # restore ra
      ld    s0,192(sp)    # restore s0
      ld    s1,184(sp)    # restore s1
      li    a0,0          # return value = 0
      addi  sp,sp,208     # restore sp
      jr    ra            # return
```

x0	hard-wired zero
ra (x1)	Caller
sp (x2)	Callee
gp (x3)	-
tp (x4)	-

t0-2 (x5-7)	Caller
s0,1 (x8,9)	Callee
a0-7 (x10-17)	Arguments, return value
s2-s11 (x18-27)	Callee
t3-6 (x28-31)	Caller

The Calling Convention

The Calling Convention



- The calling procedure is the **caller**, the called function is the **callee**

- **The Calling convention**: specification that defines
  - how parameters are passed
    - registers, stack
  - how return values are passed
    - register(s), stack
  - how registers are handled

Calling Convention on RISC-V

- **Arguments passed to functions via registers a0 – a7**
  - If more than 8 integral parameters, then pass rest on stack
  - Values are returned in a0 (or a0/a1 if the returned value is larger than one register)

- **All references to stack frame via stack pointer sp**

x0 hard-wired zero	t0-2 (x5-7)
ra (x1)	s0,1 (x8,9)
sp (x2)	a0-7 (x10-17)
gp (x3)	s2-s11 (x18-27)
tp (x4)	t3-6 (x28-31)

Register Saving Conventions

- **What about the remaining registers?**

Caller

```
yoo(…) {
  who();
}
```

Callee

```
who(…) {
  …
}
```

- ***"Caller Save"***
  - registers that the callee can overwrite
    (caller assumes value is not preserved across procedure calls)
  - Caller saves temporary values in its frame before the call

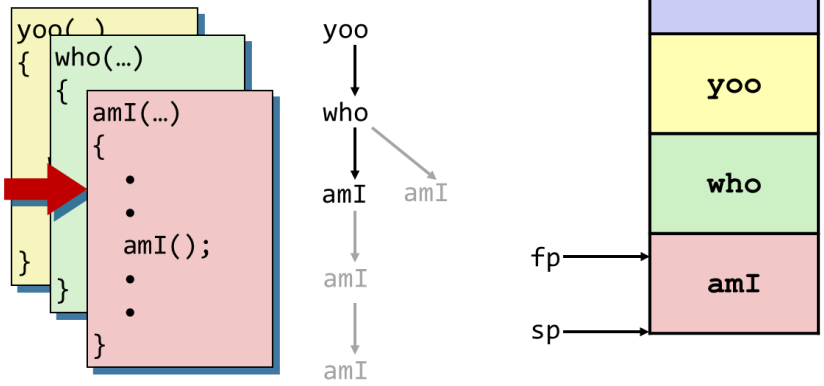
- ***"Callee Save"***
  - registers that the callee must preserve before overwriting with a new value
    (caller can reuse the value across procedure calls)
  - Callee saves temporary values in its frame before using

RISC-V Calling Convention

x0	hard-wired zero
ra (x1)	Caller
sp (x2)	Callee
gp (x3)	-
tp (x4)	-

t0-2 (x5-7)	Caller
s0,1 (x8,9)	Callee
a0-7 (x10-17)	Arguments, return value
s2-s11 (x18-27)	Callee
t3-6 (x28-31)	Caller

- Details: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc



The Runtime Stack and Stack-Based Language

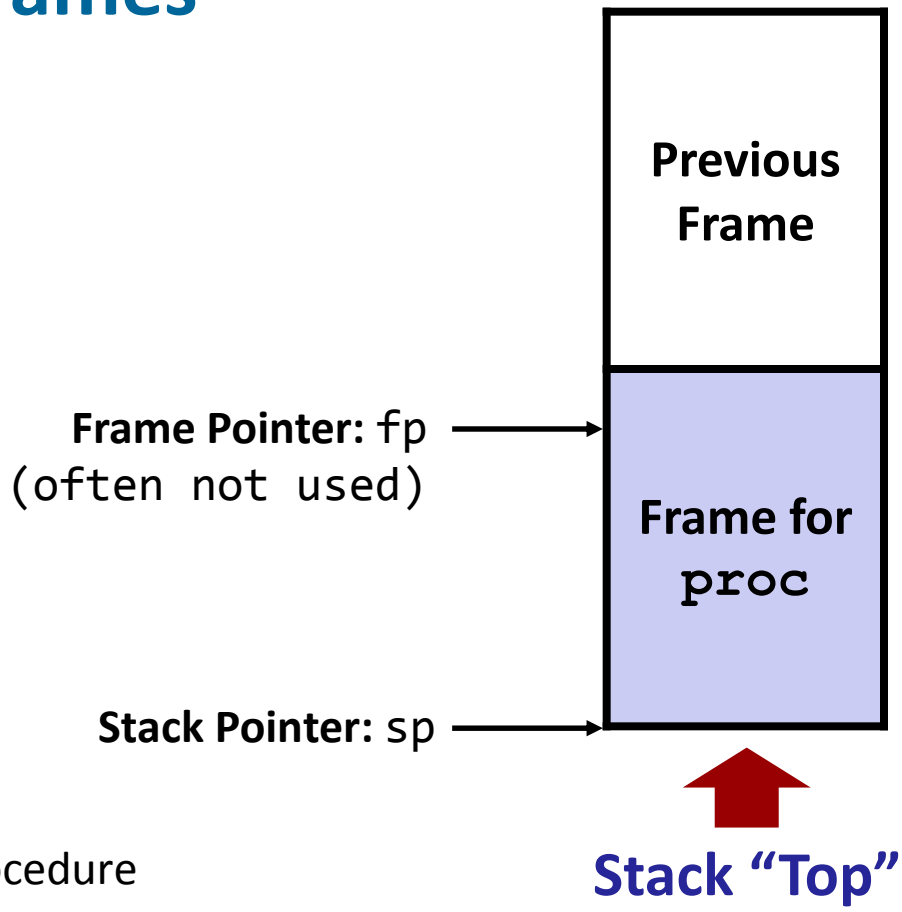
Stack-Based Languages

- **The runtime stack is a good match for stack-based language**
  - e.g., C, Pascal, Java
  - Code must be “reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in frames
  - state for single procedure instantiation

Procedure Activation Frames

- **aka "stack frames"**

- **Contents**
  - Local variables
  - Return information
  - Temporary space



- **Management**
  - Space allocated when entering a procedure
    - "Set-up" code, called *prologue*
  - Deallocated when returning to the caller
    - "Cleanup" code, called *epilogue*
  - Code automatically generated by the compiler

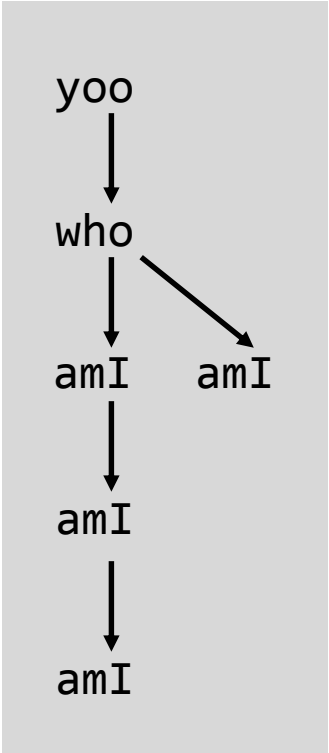
Call Chain Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

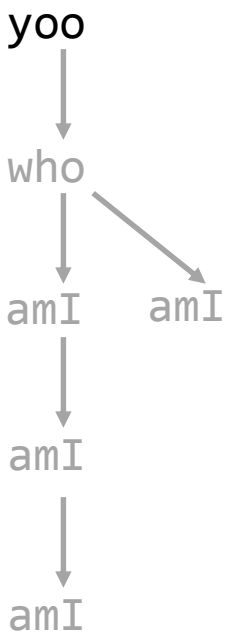
Example Call Chain

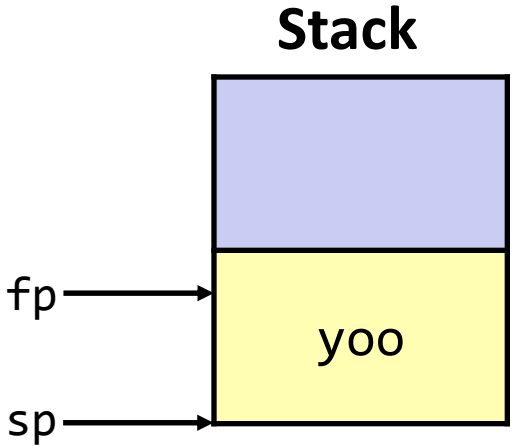


Procedure `amI()` is recursive

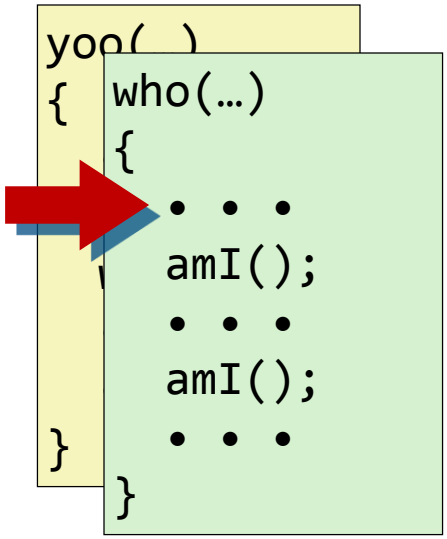
Example

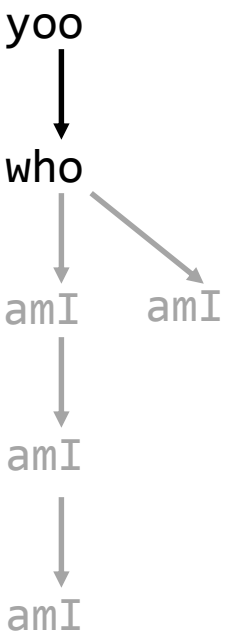
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

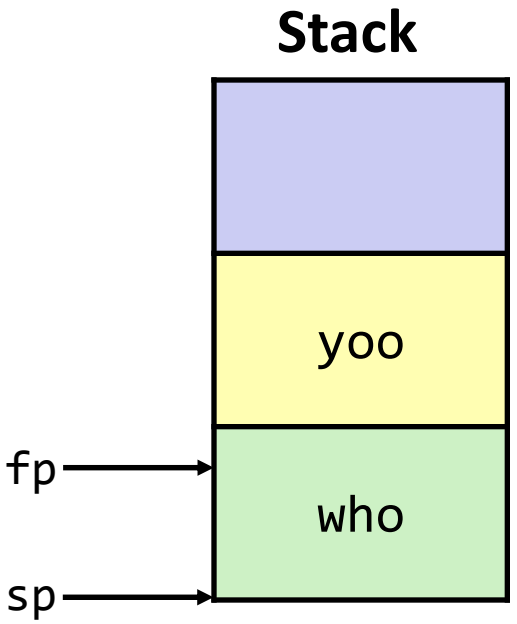




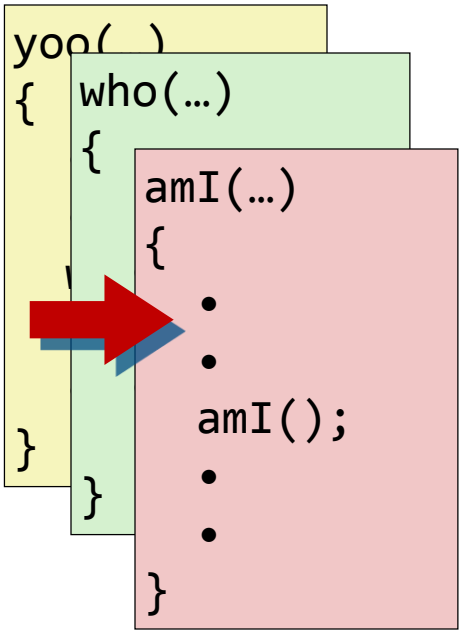
Example

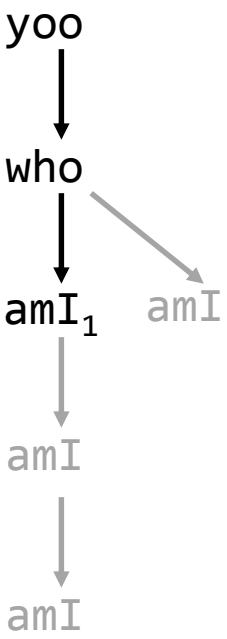


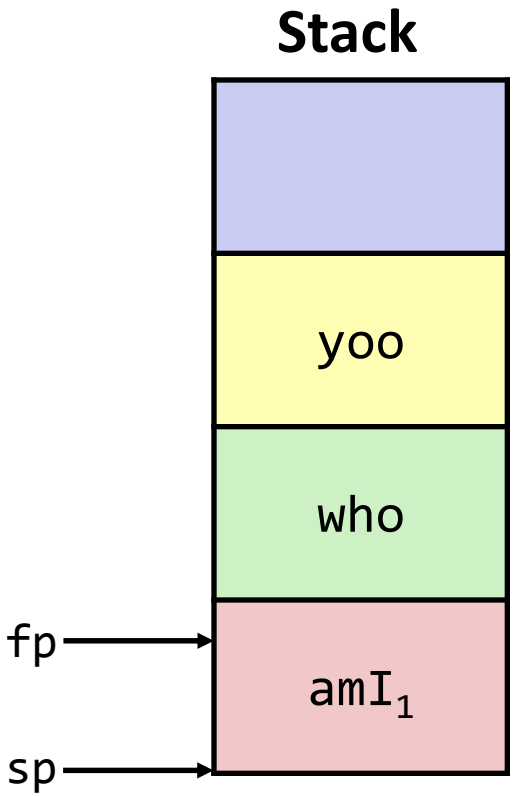




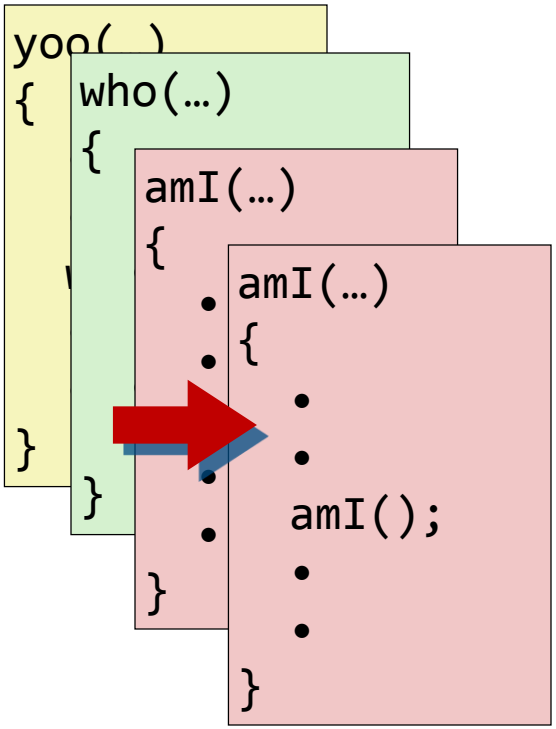
Example

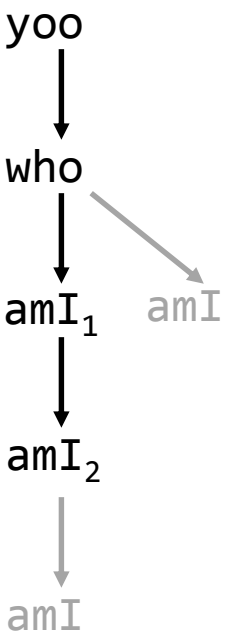


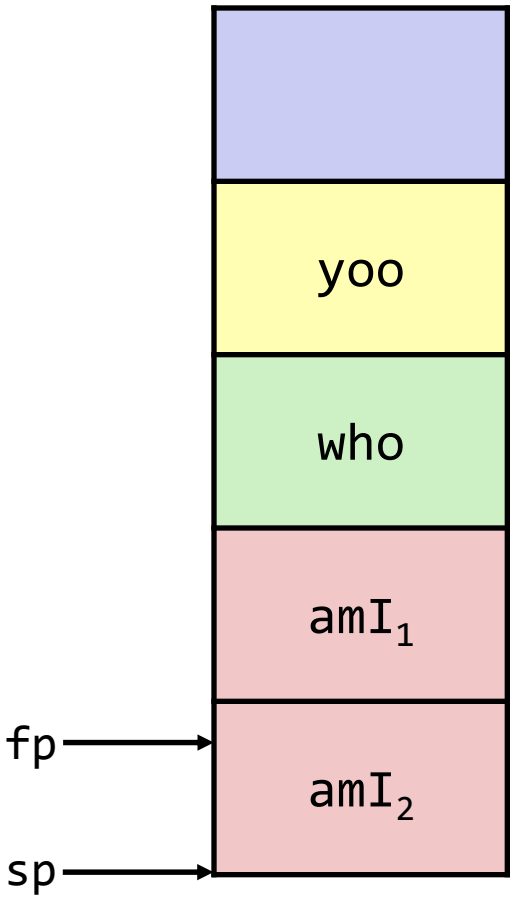




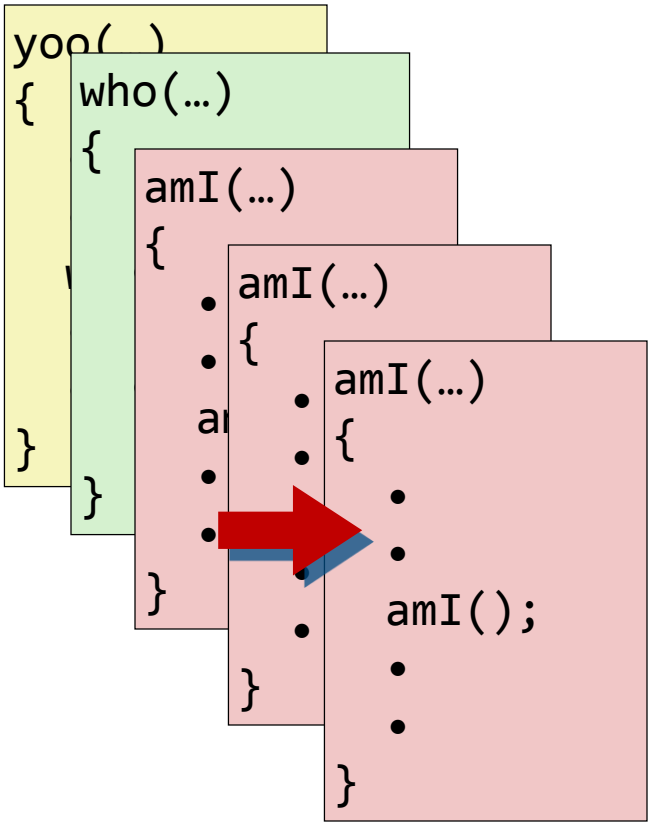
Example

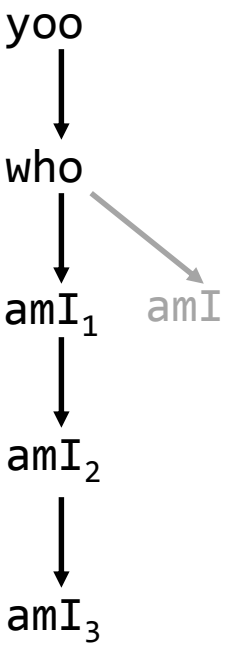


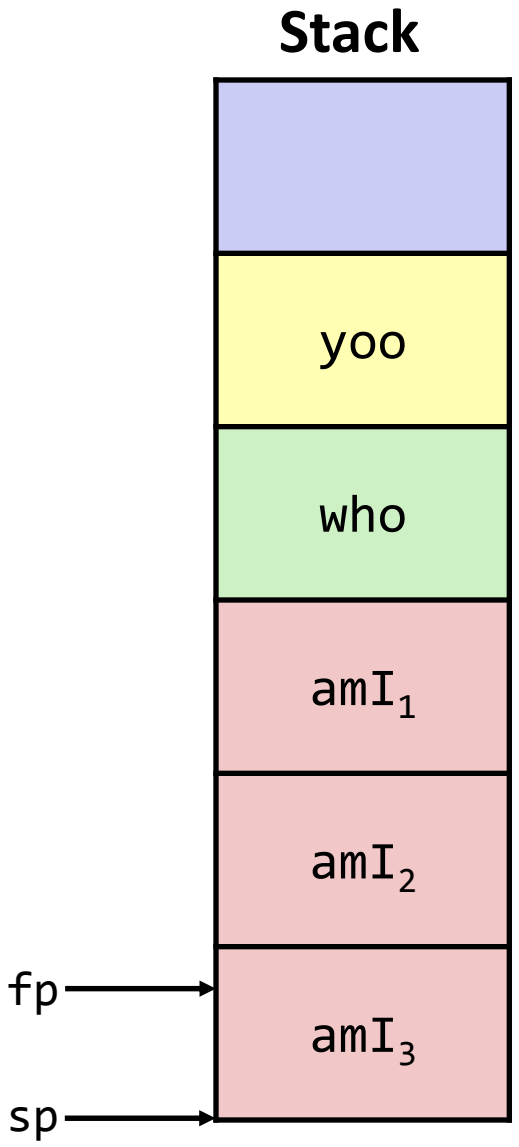




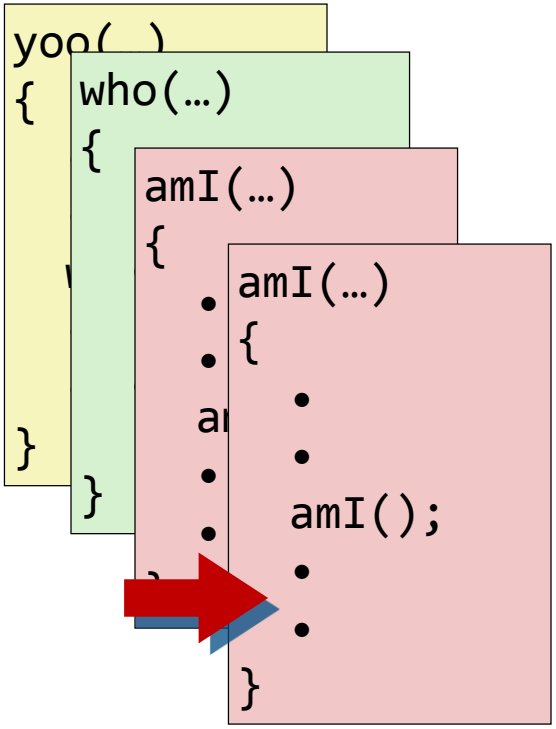
Example

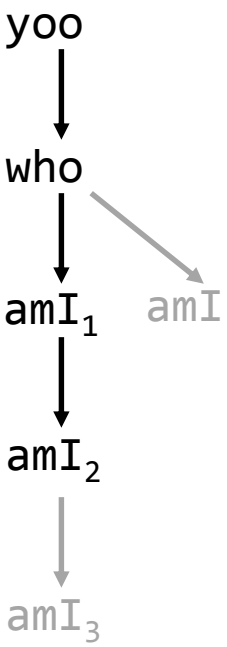


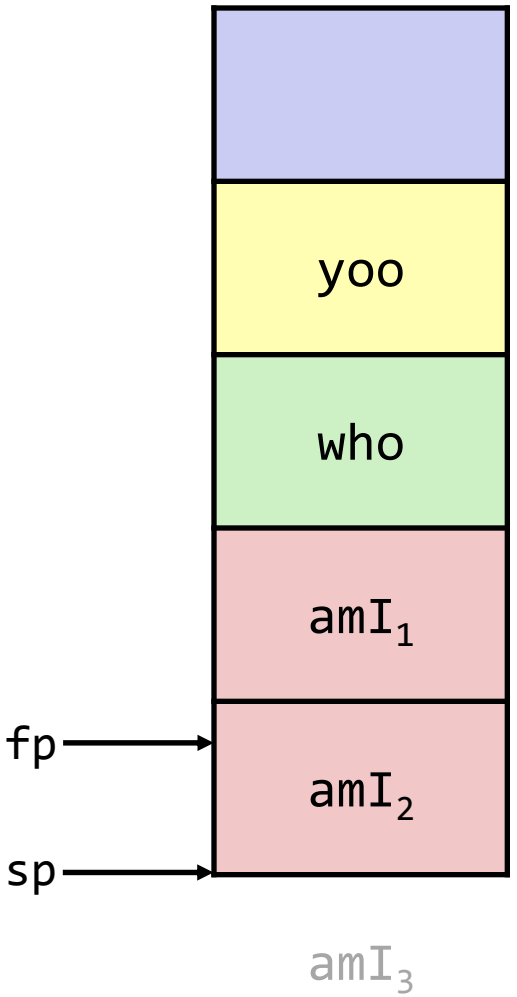




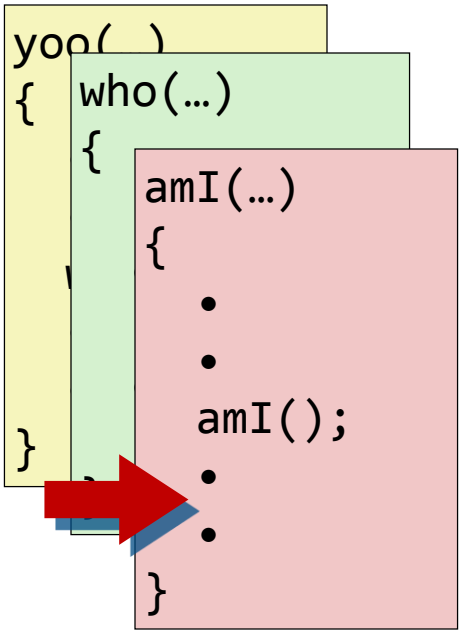
Example

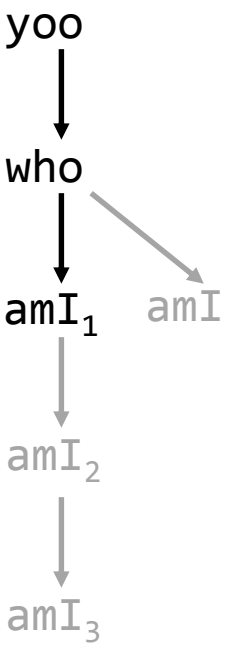


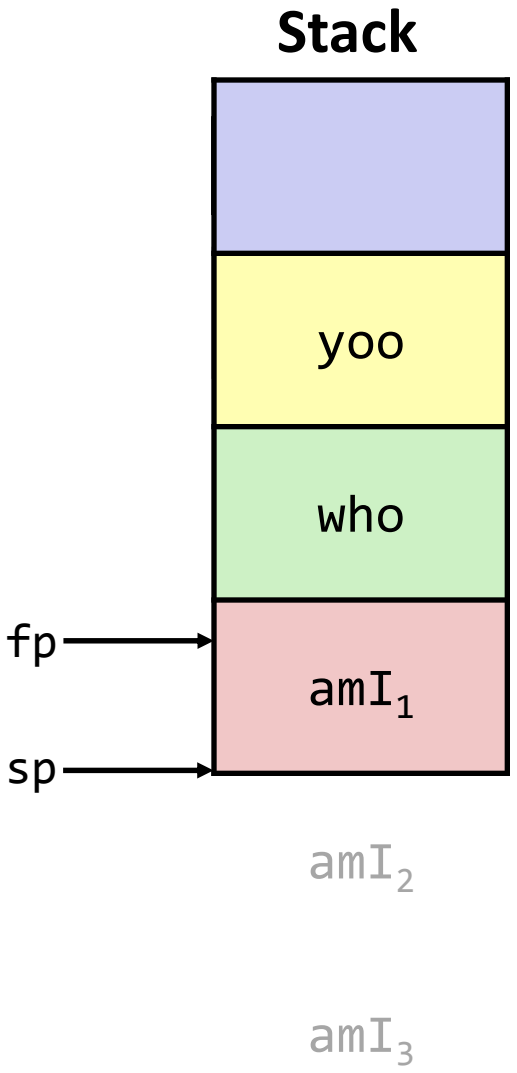




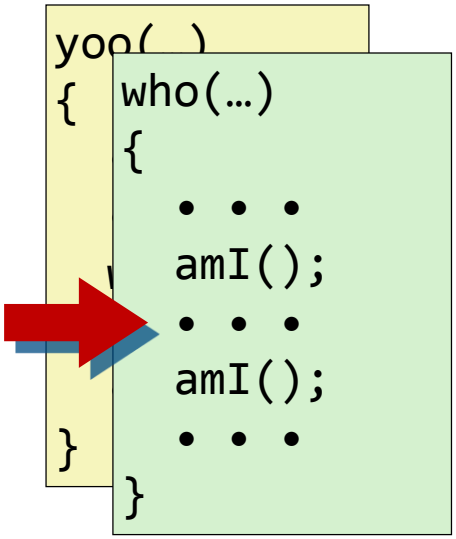
Example

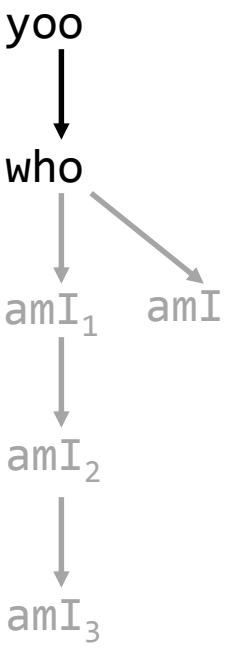


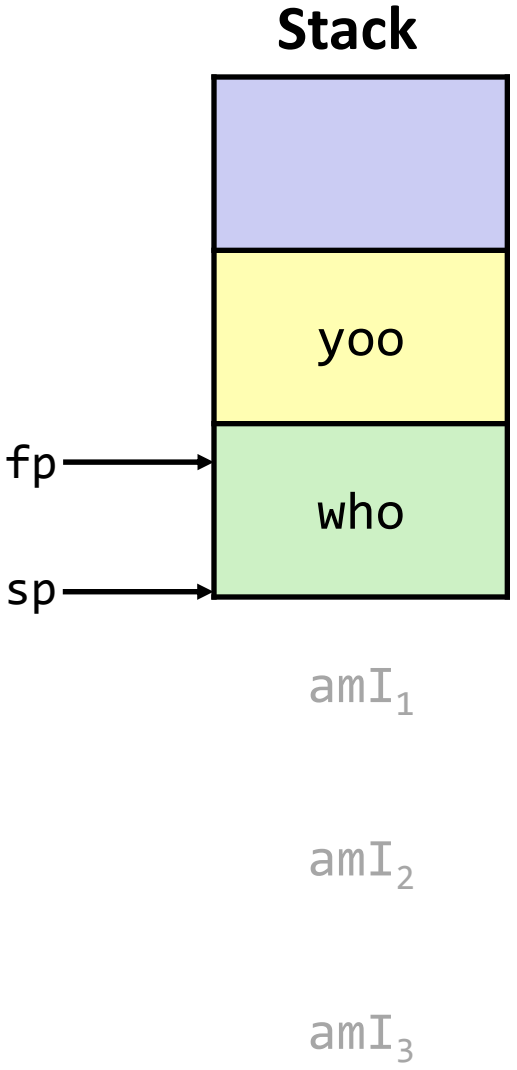




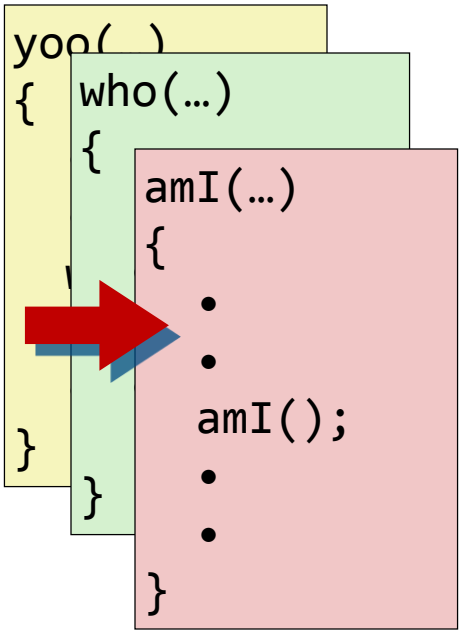
Example

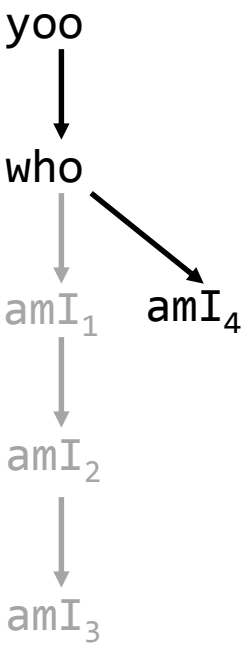


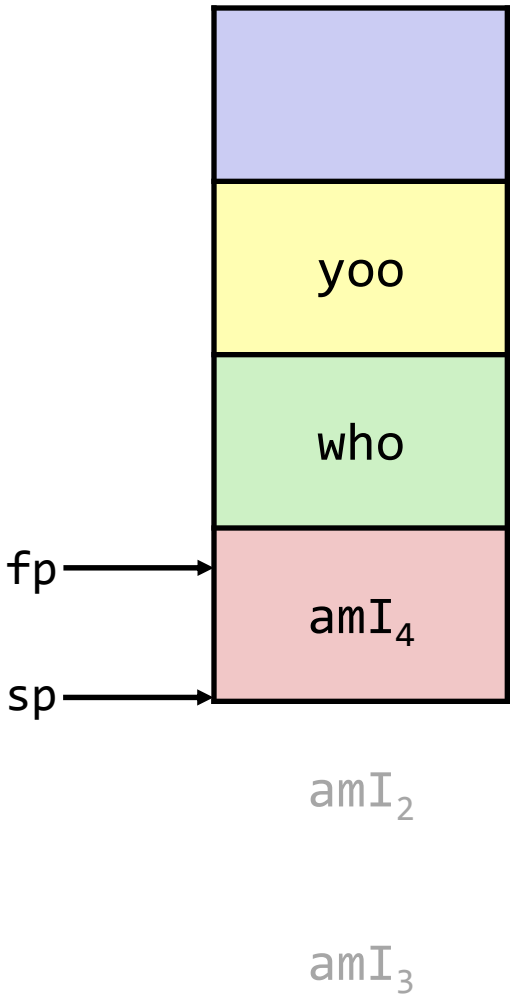




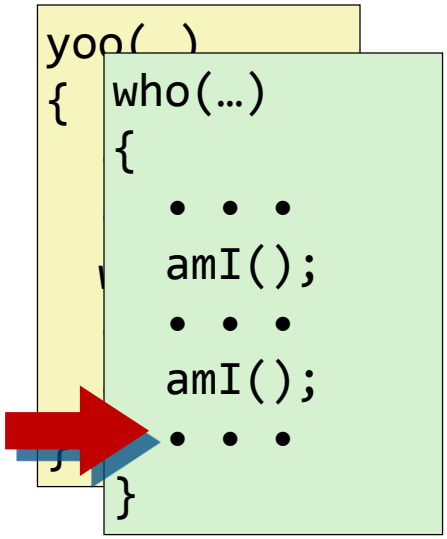
Example

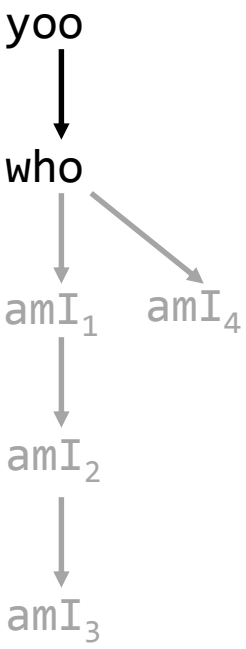


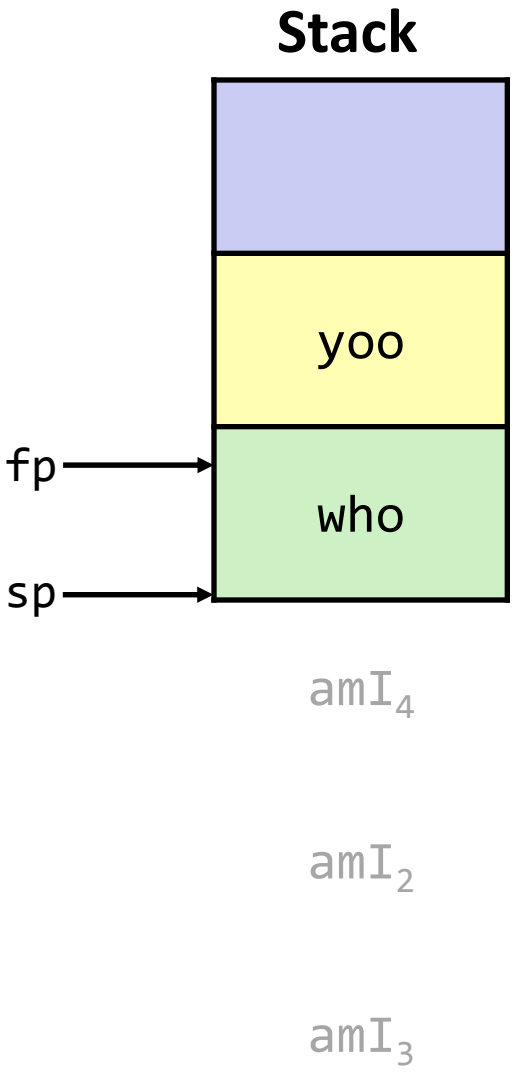




Example

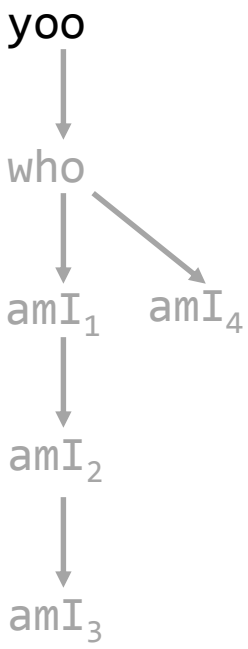


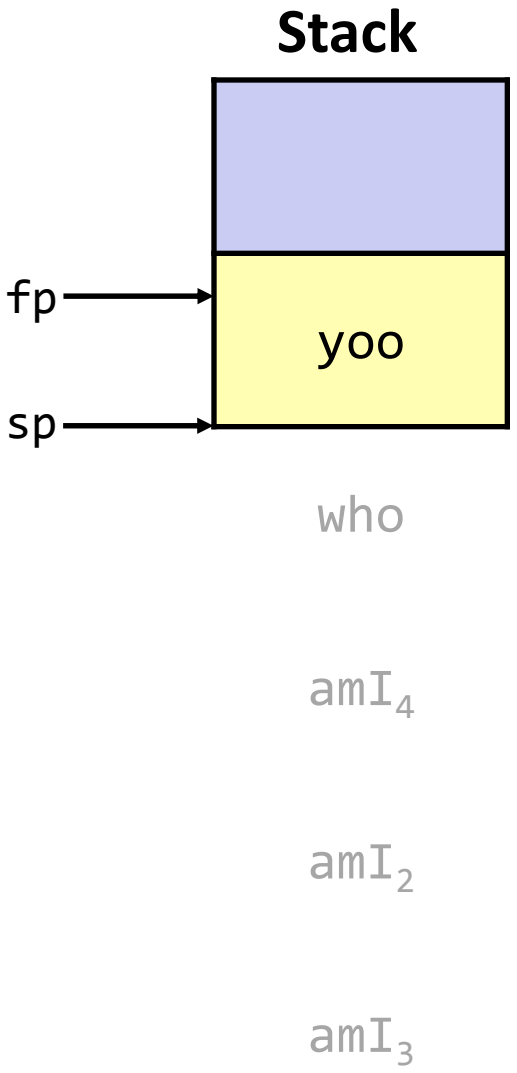




Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```





PAF Example

```
int foo(int *A, int a, int *p)
{
  int t1 = a;

  if (a + *p > 0) t1 += *p;

  if (a > 0) t1 += foo(A, a-1, p);

  A[a] = t1;

  return t1;
}
```

```
foo:
    addi    sp,sp,-32
    sd      s1,8(sp)
    lw      s1,0(a2)
    sd      s0,16(sp)
    sd      s2,0(sp)
    sd      ra,24(sp)
    addw    s1,s1,a1
    mv      s0,a1
    mv      s2,a0
    ble     s1,zero,.L7
.L2:
    ble     s0,zero,.L3
    addiw   a1,s0,-1
    mv      a0,s2
    call    foo
    addw    s1,s1,a0
.L3:
    slli    s0,s0,2
    add     s0,s2,s0
    sw      s1,0(s0)
    ld      ra,24(sp)
    ld      s0,16(sp)
    ld      s2,0(sp)
    mv      a0,s1
    ld      s1,8(sp)
    addi    sp,sp,32
    jr      ra
.L7:
    mv      s1,a1
    j       .L2
```

PAF Example

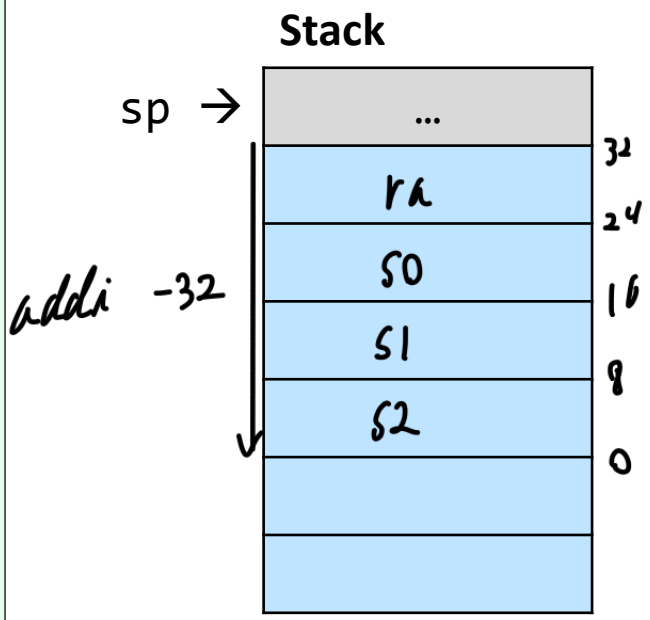
```
foo:
    addi    sp,sp,-32
    sd      s1,8(sp)
    lw      s1,0(a2)
    sd      s0,16(sp)
    sd      s2,0(sp)
    sd      ra,24(sp)
    addw    s1,s1,a1
    mv      s0,a1
    mv      s2,a0
    ble     s1,zero,.L7
.L2:
    ble     s0,zero,.L3
    addiw   a1,s0,-1
    mv      a0,s2
    call    foo
    addw    s1,s1,a0
.L3:
    slli    s0,s0,2
    add     s0,s2,s0
    sw      s1,0(s0)
    ld      ra,24(sp)
    ld      s0,16(sp)
    ld      s2,0(sp)
    mv      a0,s1
    ld      s1,8(sp)
    addi    sp,sp,32
    jr      ra
.L7:
    mv      s1,a1
    j       .L2
```

prologue

epilogue

return to origin

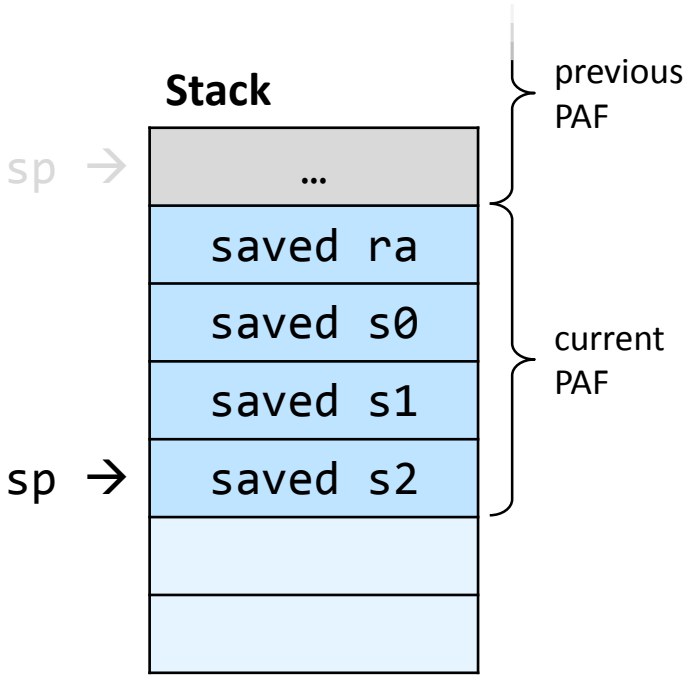
```
int foo(int *A, int a, int *p)
{
  int t1 = a;

  if (a + *p > 0) t1 += *p;
  if (a > 0) t1 += foo(A, a-1, p);
  A[a] = t1;

  return t1;
}
```

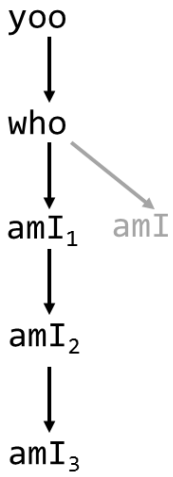


PAF Example

```
foo:
    addi    sp,sp,-32          }
    sd      s1,8(sp)           }
    lw      s1,0(a2)           }
    sd      s0,16(sp)          }  prologue
    sd      s2,0(sp)           }
    sd      ra,24(sp)          }
    addw    s1,s1,a1
    mv      s0,a1
    mv      s2,a0
    ble     s1,zero,.L7
.L2:
    ble     s0,zero,.L3
    addiw   a1,s0,-1
    mv      a0,s2
    call    foo
    addw    s1,s1,a0
.L3:
    slli    s0,s0,2
    add     s0,s2,s0
    sw      s1,0(s0)
    ld      ra,24(sp)          }
    ld      s0,16(sp)          }
    ld      s2,0(sp)           }  epilogue
    mv      a0,s1              }
    ld      s1,8(sp)           }
    addi    sp,sp,32           }
    jr      ra
.L7:
    mv      s1,a1
    j       .L2
```

```
int foo(int *A, int a, int *p)
{
  int t1 = a;

  if (a + *p > 0) t1 += *p;
  if (a > 0) t1 += foo(A, a-1, p);
  A[a] = t1;

  return t1;
}
```

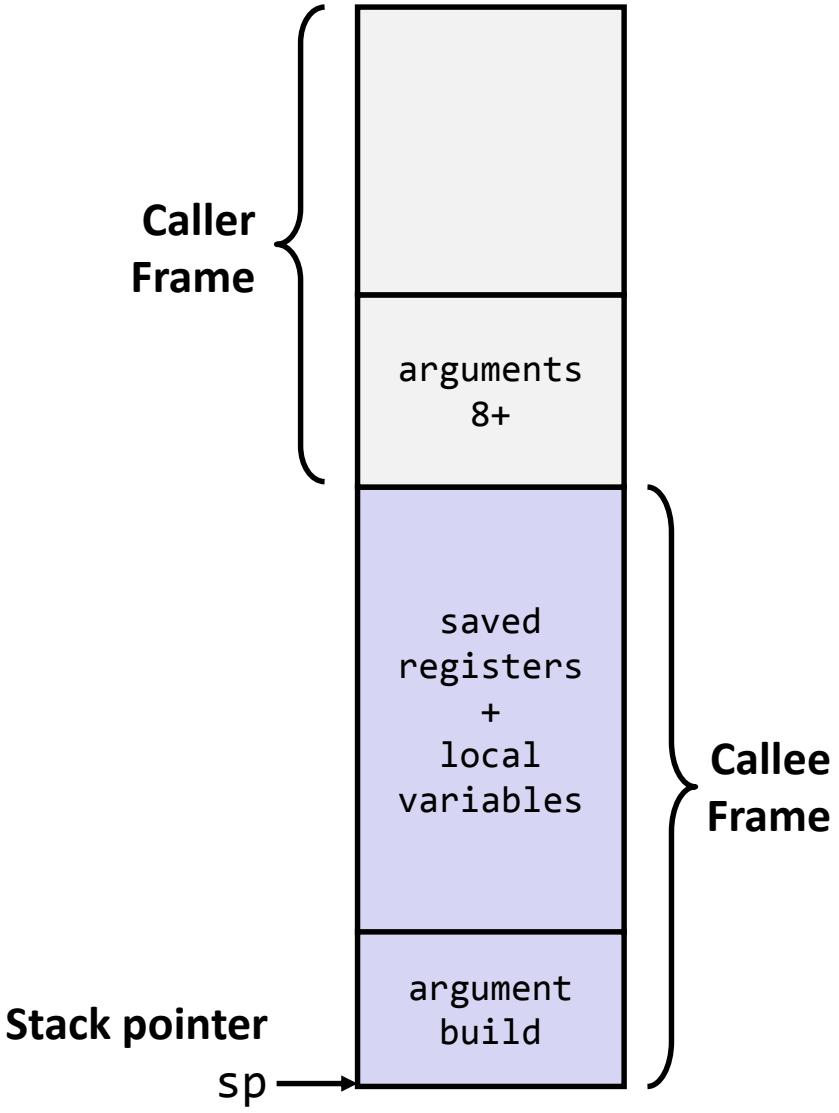




Module Summary

Module Summary

- **Procedures**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
  - Recursion (& mutual recursion) handled by normal calling conventions
    - Can safely store values in local stack frame and in callee-saved registers
    - Put function arguments at top of stack
    - Result return in `a0-1`
  - Pointers are addresses of values
    - On stack or global



Module Summary

- **Calling convention**
  - Up to 8 parameters passed in registers a0-a7 (x10-x17)
  - Return value passed in a0 or a0/a1 for scalars larger than a register
  - Callee saved registers: register values must be preserved across function calls
  - Caller saves registers: register values can be overwritten by callee
  - Note that, except for leaf procedures, all functions are both callee and caller!
  - Details: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc

x0	hard-wired zero
ra (x1)	Caller
sp (x2)	Callee
gp (x3)	-
tp (x4)	-

t0-2 (x5-7)	Caller
s0,1 (x8,9)	Callee
a0-7 (x10-17)	Arguments, return value
s2-s11 (x18-27)	Callee
t3-6 (x28-31)	Caller