

9/14/26



COMP 311: *Computer Organization!*

Lecture 7: Calling Convention, Stack Review



The University
of North Carolina
at Chapel Hill

<https://tinyurl.com/comp311-fa26>

Logistics Update



- You should have received a GitHub Organization invite over the weekend → accept! If you receive an error, please email me!
 - Lab is coming! Some technical issues have delayed us! 😊
- Practice problems and solutions posted
 - Will update solutions today!

Quiz Topics for Wednesday



RISC-V Control Flow

- Branch instructions (beq, bne, comparison/branch patterns)
- Labels and how branches change program execution
- Tracing loops and conditional code
- Translating simple C if statements and loops to/from RISC-V

Procedures & Function Calls

- jal, jalr
- Return address (ra)
- Tracing function calls in assembly
- Tracing recursion in assembly

The Stack

- Stack pointer (sp)
- Creating and removing a stack frame
- Saving/restoring registers with sw / lw
- Why values such as ra or arguments may need to be saved
- Understanding stack offsets

The Program Counter

- Determine the next PC after a normal instruction, branch, or jump
- Explain how branches/jumps affect control flow or the PC

Plan for Class Today



- 3:35-4:50: Stack Review
- 4:50-4:15: Calling Convention, Stack & Frame Pointer
- 4:15-4:20: BREAK
- 4:20-4:50: Stack Practice

Guiding Question 😊

*How do we manage
the complexity of
function calling in
assembly?*

Stack Concepts and the Application Binary Interface

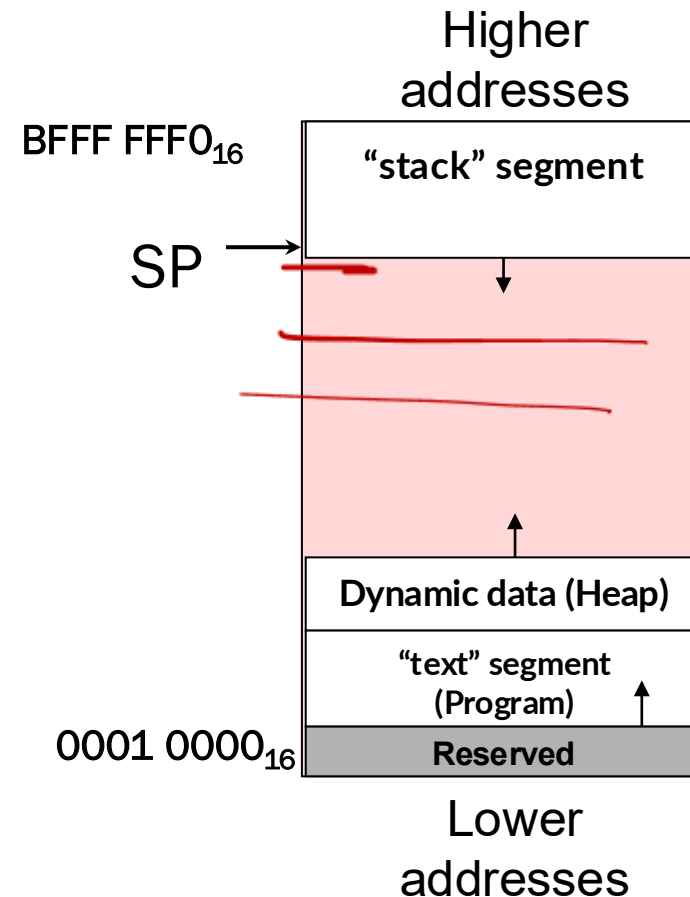
RISC-V Stack Convention



These conventions for allocating variables when calling fns →
part of the ABI

CONVENTIONS:

- Assign a register as the **Stack Pointer** ($sp = x2$).
- Stack grows **DOWN** (towards lower addresses) on *pushes* and *allocates*
- sp points to the last or **TOP** *used* location.
- Stack is placed far away from the program and its data.



Register use conventions

32 regs



By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages.

We'll use these names increasingly.

Why have such conventions?



x0/zero (always zero)
x1/ra (return address)
x2/sp (stack pointer)
x3/gp (global pointer)
x4/tp (thread pointer)
x5/t0 (temporary)
x6/t1 (temporary)
x7/t2 (temporary)
x8/fp (frame pointer)
x9/s1 (saved)
x10/a0 (argument/return value 1)
x11/a1 (argument/return value 2)
x12/a2 (argument)
x13/a3 (argument)
x14/a4 (argument)
x15/a5 (argument)

x16/a6 (argument)
x17/a7 (argument)
x18/s2 (saved)
x19/s3 (saved)
x20/s4 (saved)
x21/s5 (saved)
x22 (saved)
x23 (saved)
x24 (saved)
x25 (saved)
x26 (saved)
x27 (saved)
x28 (temporary)
x29 (temporary)
x30 (temporary)
x31 (temporary)

Prevents functions from accidentally overwriting each other's values

Allows independently compiled code to co-execute correctly!

Separately written fns can safely share CPU without corrupting each other's data

That was a little too easy...



```
int x = 5;
int y;

void main() {
    y = fact(x);
}

int fact(x) {
    if (x <= 1)
        return x;
    else
        return x*fact(x-1);
}
```

Q: What happens when we trace this?

```
main:    lw    a0,x
         jal   ra,fact
         sw    a0,y
*halt:   j     halt

x:       .word  2
y:       .word  0

fact:    ←addi   t0,x0,1
         bge    t0,a0,return
         addi    t0,x0,a0
         addi   a0,a0,-1
         ↗     jal   ra,fact
         mul    a0,a0,t0
return:  jalr   x0,ra
```

That was a little too easy...

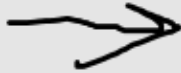


- The recursive call to fact() overwrites the saved value of x in t0.
- To make a bad thing worse, the ra is also overwritten </3

```
main:    lw    a0,x
         jal   ra,fact
         sw    a0,y
*halt:   j     halt

x:       .word  2
y:       .word  0

fact:    addi   t0,x0,1
         bge    t0,a0,return
         addi   t0,x0,a0
         addi   a0,a0,-1
         jal    ra,fact
         mul    a0,a0,t0
return:  jalr   x0,ra
```



Incorporating A Stack



```
int sqr(int x) {  
    if (x > 1)  
        x = sqr(x-1)+x+x-1;  
    return x;  
}
```

```
main()  
{  
    sqr(10);  
}
```



Every call **pushes** a frame (**prologue**) and **pops** it (**epilogue**), so nested/recursive calls don't overwrite each other's saved ra or saved registers

sqr:	<pre>addi sp, sp, -8 sw ra, 4(sp) sw s0, 0(sp) slti t0, a0, 2 bne t0, x0, return add s0, x0, a0 addi a0, a0, -1 jal ra, sqr add a0, a0, s0 add a0, a0, s0 addi a0, a0, -1</pre>	function prologue
return:	<pre>lw s0, 0(sp) lw ra, 4(sp) addi sp, sp, 8 jalr x0, ra</pre>	function epilogue
main:	<pre>addi sp, sp, -4 sw ra, (sp) addi a0, x0, 10 jal ra, sqr lw ra, (sp) addi sp, sp, 4 jalr x0, ra</pre>	

Stack Management Policies



ALLOCATE k: reserve k WORDS of stack
 $SP = SP - 4 * k$

```
addi  sp,sp,-4*k
```

DEALLOCATE k: release k WORDS of stack
 $SP = SP + 4 * k$

```
addi  sp,sp,4*k
```

PUSH x: push Reg[x] onto stack
 $Mem[SP - 4] = Rx$
 $SP = SP - 4$

```
addi  sp,sp,-4  
sw    rx,(sp)
```

POP x: pop the top of the stack into Reg[x]
 $Rx = Mem[SP]$
 $SP = SP + 4$

```
lw    rx,(sp)  
addi  sp,sp,4
```

Practice Time! 😊



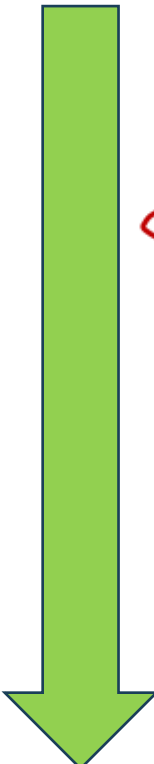
Stack Tracing. Following along w/ WS Code



saved ra: [0] Initial/Base Frame. Nothing is below this on the stack

Each stack entry stores the return address, which is the address of the next instruction after JAL

Stack Tracing. Following along w/ WS Code



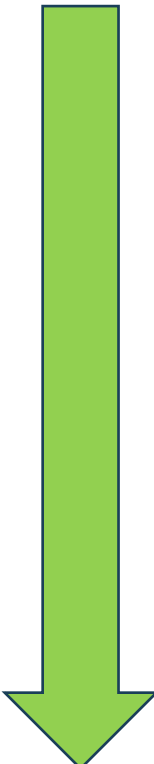
saved ra: [0] Initial/Base Frame. Nothing is below this on the stack

saved ra: [12] Main called from Reset

12
ra: PC + 4

addr 8: jal ra, main

Stack Tracing. Following along w/ WS Code

A large green arrow pointing downwards, indicating the direction of stack growth or the sequence of frames.

saved ra: [0] Initial/Base Frame. Nothing is below this on the stack

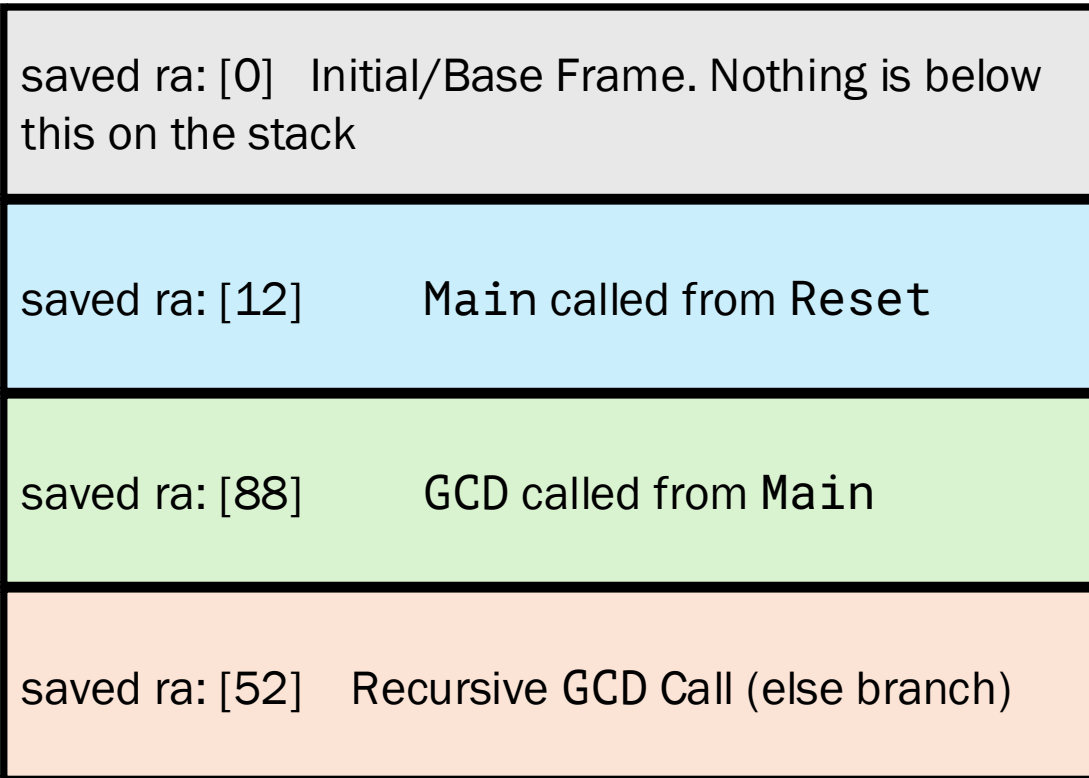
saved ra: [12] Main called from Reset

saved ra: [88] GCD called from Main

addr 8: jal ra, main

addr 84: jal ra, gcd

Stack Tracing. Following along w/ WS Code

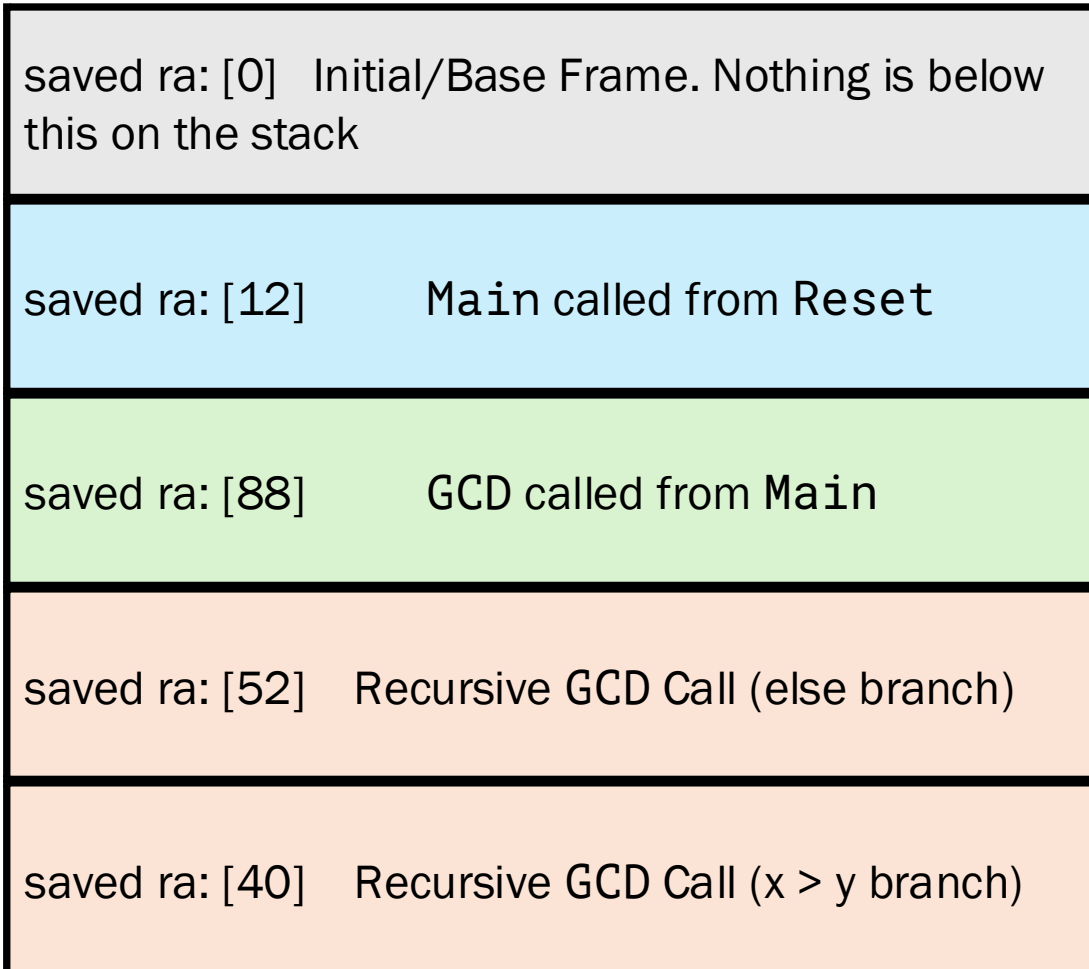


addr 8: jal ra, main

addr 84: jal ra, gcd

addr 48: jal ra, gcd

Stack Tracing. Following along w/ WS Code



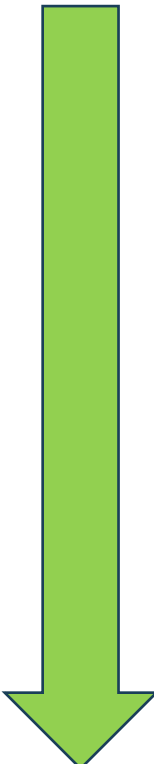
addr 8: jal ra, main

addr 84: jal ra, gcd

addr 48: jal ra, gcd

addr 36: jal ra, gcd

Stack Tracing. Following along w/ WS Code



saved ra: [0] Initial/Base Frame. Nothing is below this on the stack
saved ra: [12] Main called from Reset
saved ra: [88] GCD called from Main
saved ra: [52] Recursive GCD Call (else branch)
saved ra: [40] Recursive GCD Call (x > y branch)

Each stack entry stores the return address, which is the address of the next instruction after JAL

addr 8: jal ra, main

addr 84: jal ra, gcd

addr 48: jal ra, gcd

addr 36: jal ra, gcd

UNIT 1: PROGRAMMING THE MACHINE

[Back to the Worksheet! 😊](#)

RISC-V Debugging!

More Calling Convention Details



Procedure Linkage is Nontrivial

- The details can be overwhelming. How do we manage this complexity?
 - Abstraction: High-level languages hide the details
- There are great many implementation choices:
 - which variables are saved
 - who saves them
 - where are arguments stored?
- Solution: **CONTRACTS!**
 - Caller and Callee must agree on the details

Code Example



Set up stack frame,
save ra,
save original arg

{
sqr

```
addi sp, sp, -8  
sw    ra, 4(sp)  
sw    a0, 0(sp)
```

Check base case

{

```
slti t0, a0, 2  
bne  t0, x0, base
```

Make recursive call

{

```
addi a0, a0, -1  
jal  ra, sqr
```

Restore original arg
Compute
Skip to base
case/return

{

```
lw    a0, 0(sp)  
add   a0, a0, a0  
add   a0, a0, a0  
addi  a0, a0, -1  
j     done
```

Restore the caller's
saved return address,
give back 8 bytes of
stack space,
Jump back to caller

{

```
base:  
    # return n
```

```
done:  
    lw    ra, 4(sp)  
    addi  sp, sp, 8  
    jr    ra
```

Procedure Linkage: Caller Contract



The CALLER will:

- Save all caller-saved registers that it wants to survive subsequent calls in its stack frame
(t0-t6, a0-a7)
- Pass the first 8 arguments in registers a0-a7, and save subsequent arguments on stack,

Call procedure, using a **jal** instruction (places return address in ra).

- Access procedure's return values in a0-a1

Code Lawyer

- Our example is a CALLER. Let's make sure it obeys its contractual obligations

```
sqr:  addi sp, sp, -8
      sw   ra, 4(sp)
      sw   a0, 0(sp)
```

```
      slti t0, a0, 2
      bne  t0, x0, base
```

```
      addi a0, a0, -1
      jal  ra, sqr
```

```
      lw   a0, 0(sp)
      add  a0, a0, a0
      add  a0, a0, a0
      addi a0, a0, -1
      j    done
```

```
base:
      # return n
```

```
done:
      lw   ra, 4(sp)
      addi sp, sp, 8
      jr   ra
```



Code Lawyer

sqr:

```
addi sp, sp, -8
sw    ra, 4(sp)
sw    a0, 0(sp)
```

```
slti t0, a0, 2
bne  t0, x0, base
```

```
addi a0, a0, -1
```

```
jal  ra, sqr
```

```
lw    a0, 0(sp)
```

```
add  a0, a0, a0
add  a0, a0, a0
addi a0, a0, -1
j    done
```

```
base:
    # return n
```

```
done:
    lw    ra, 4(sp)
    addi sp, sp, 8
    jr    ra
```



- Our example is a CALLER. Let's make s contractual obligations

The CALLER will:

- Save all caller-saved registers that it wants to survive subsequent calls in its stack frame

(t0-t6, a0-a7)

- Pass the first 8 arguments in registers a0-a7, and save subsequent arguments on stack,

Call procedure, using a **jal** instruction (places return address in ra).

- Access procedure's return values in a0-a1

Procedure Linkage: Callee Contract



If needed the CALLEE will:

- 1) Allocate a stack frame with space for saved registers, local variables, and spilled args
- 2) Save any “preserved” registers used:
(s0-s11, and ra if we call another fn)
- 3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set fp (s0) to a fixed location in stack frame (optional)
- 4) EXECUTE procedure ~~47~~
- 5) Place return values in a0-a1
- 6) Restore saved registers
- 7) Restore sp to its original value
- 8) Return to CALLER with jr ra or ret

sqr:

Code Lawyer: w/ Callee this Time

- Our example is also a CALLEE. Are these contractual obligations satisfied?



If needed the CALLEE will:

- 1) Allocate a stack frame with space for saved registers, local variables, and spilled args
- 2) Save any “preserved” registers used:
(s0-s11, and ra if we call another fn)
- 3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set fp (s0) to a fixed location in stack frame (optional)
 - 4) EXECUTE procedure
 - 5) Place return values in a0-a1
 - 6) Restore saved registers
 - 7) Restore sp to its original value
 - 8) Return to CALLER with jr ra or ret

```
addi sp, sp, -8
sw    ra, 4(sp)
sw    a0, 0(sp)
```

```
slti t0, a0, 2
bne t0, x0, base
```

```
addi a0, a0, -1
jal  ra, sqr
```

```
lw    a0, 0(sp)
add   a0, a0, a0
add   a0, a0, a0
```

```
addi a0, a0, -1
j     done
```

```
base:
    # return n
```

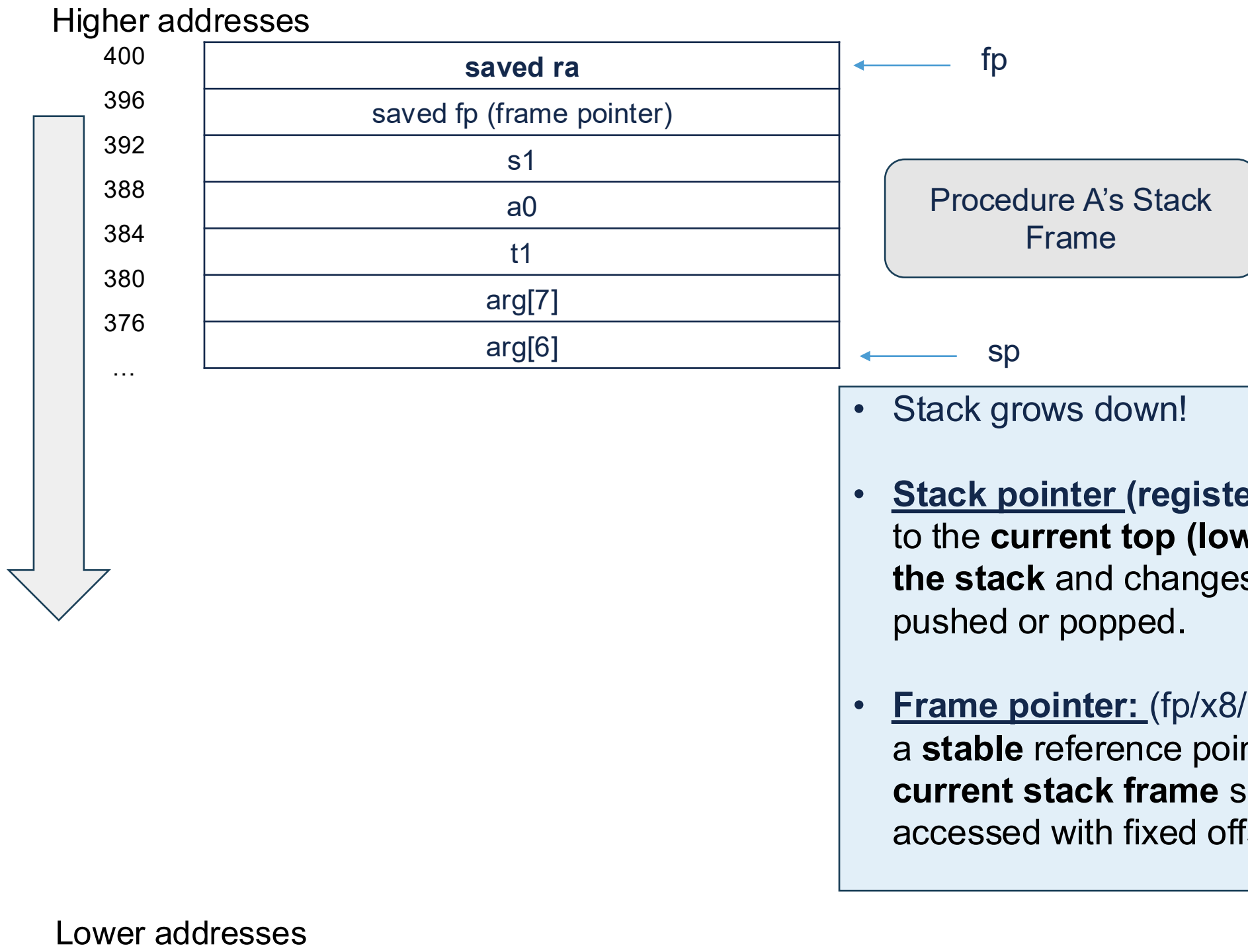
```
done:
    lw    ra, 4(sp)
    addi sp, sp, 8
    jr    ra
```

The Frame Pointer

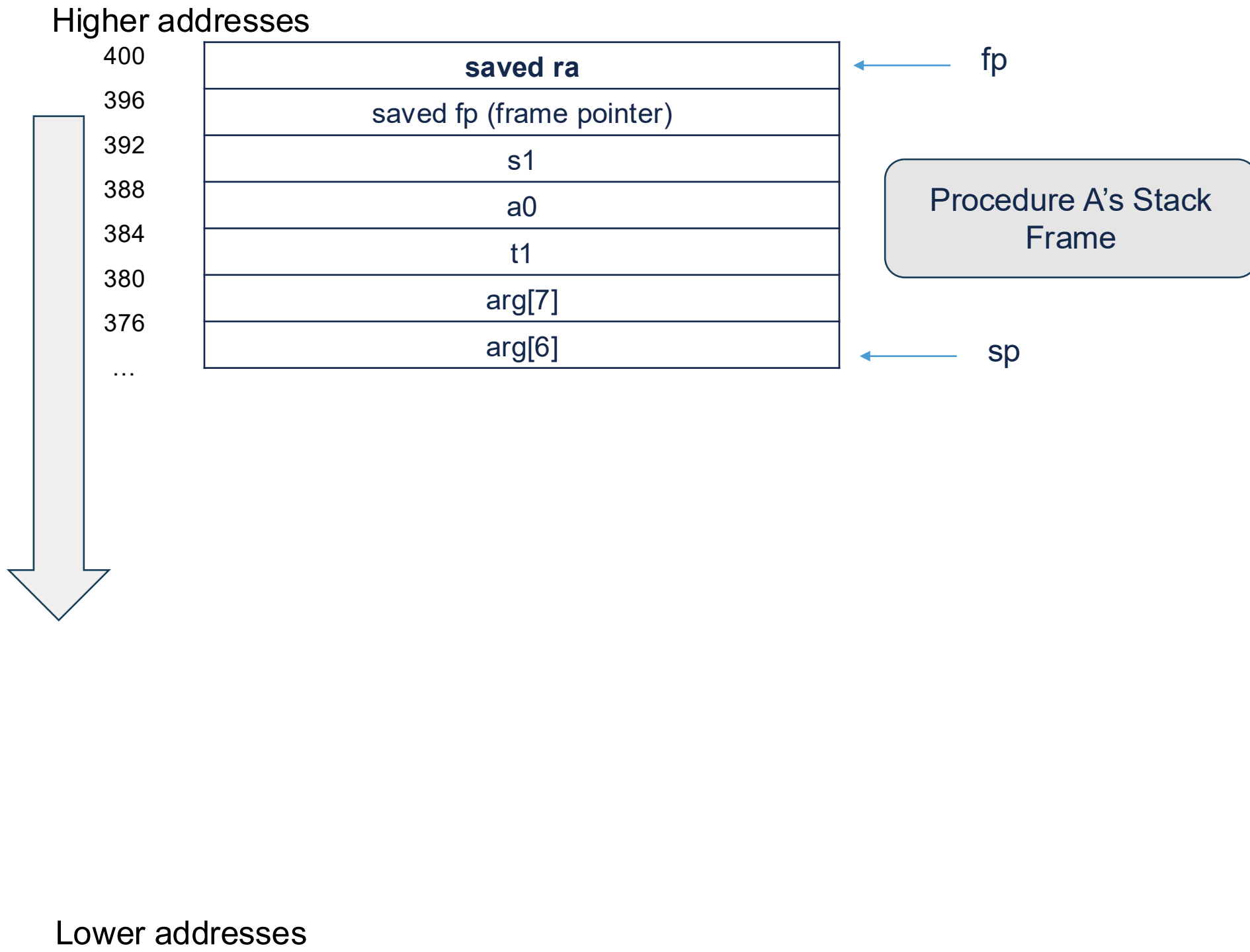


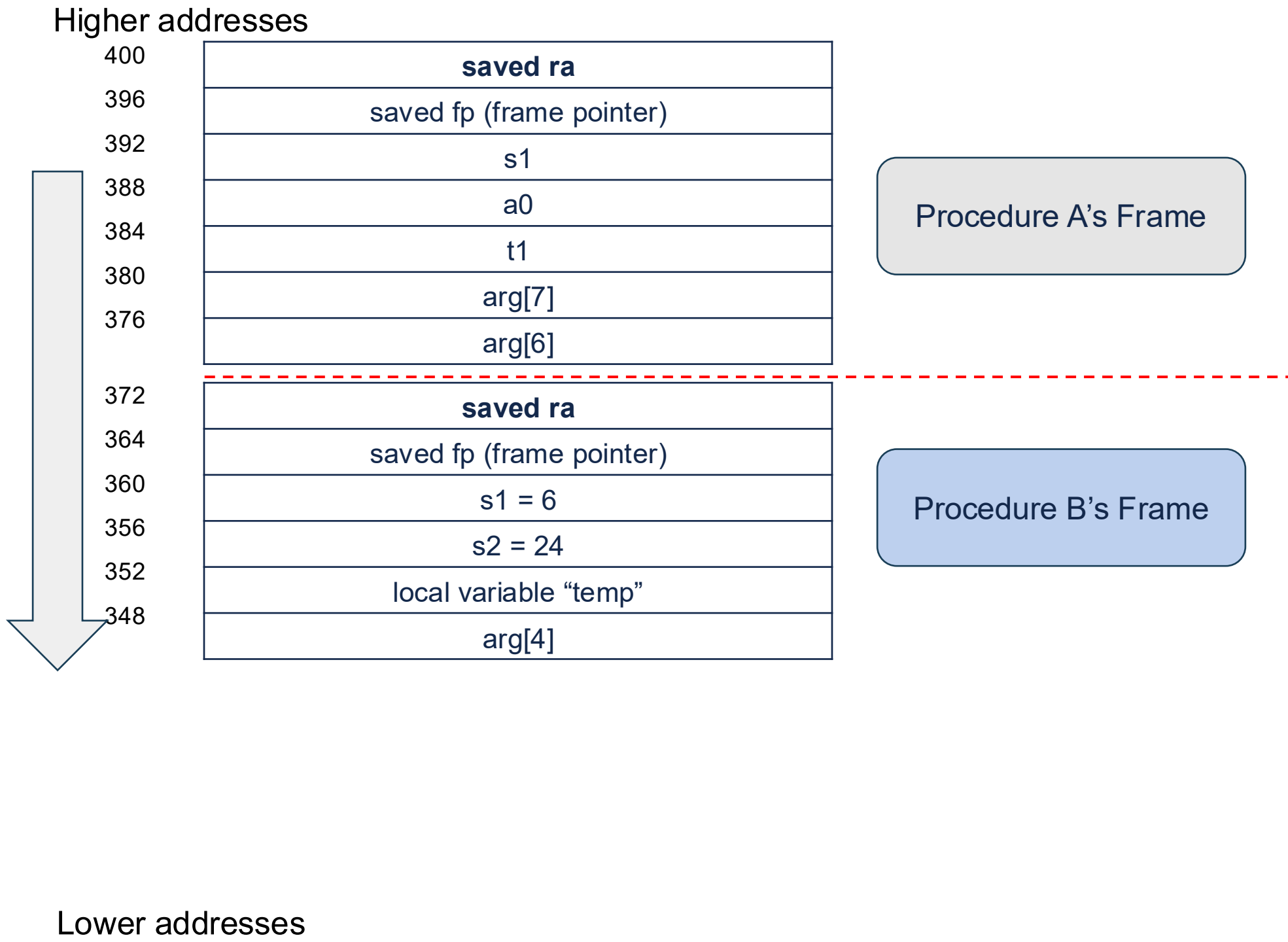
- Frame pointer (fp / s0) gives a stable reference point within the current stack frame.
- Useful because sp may move while a function is running.
- Local variables / saved values can be accessed at fixed offsets from fp.
- It is optional, many functions can use only sp!

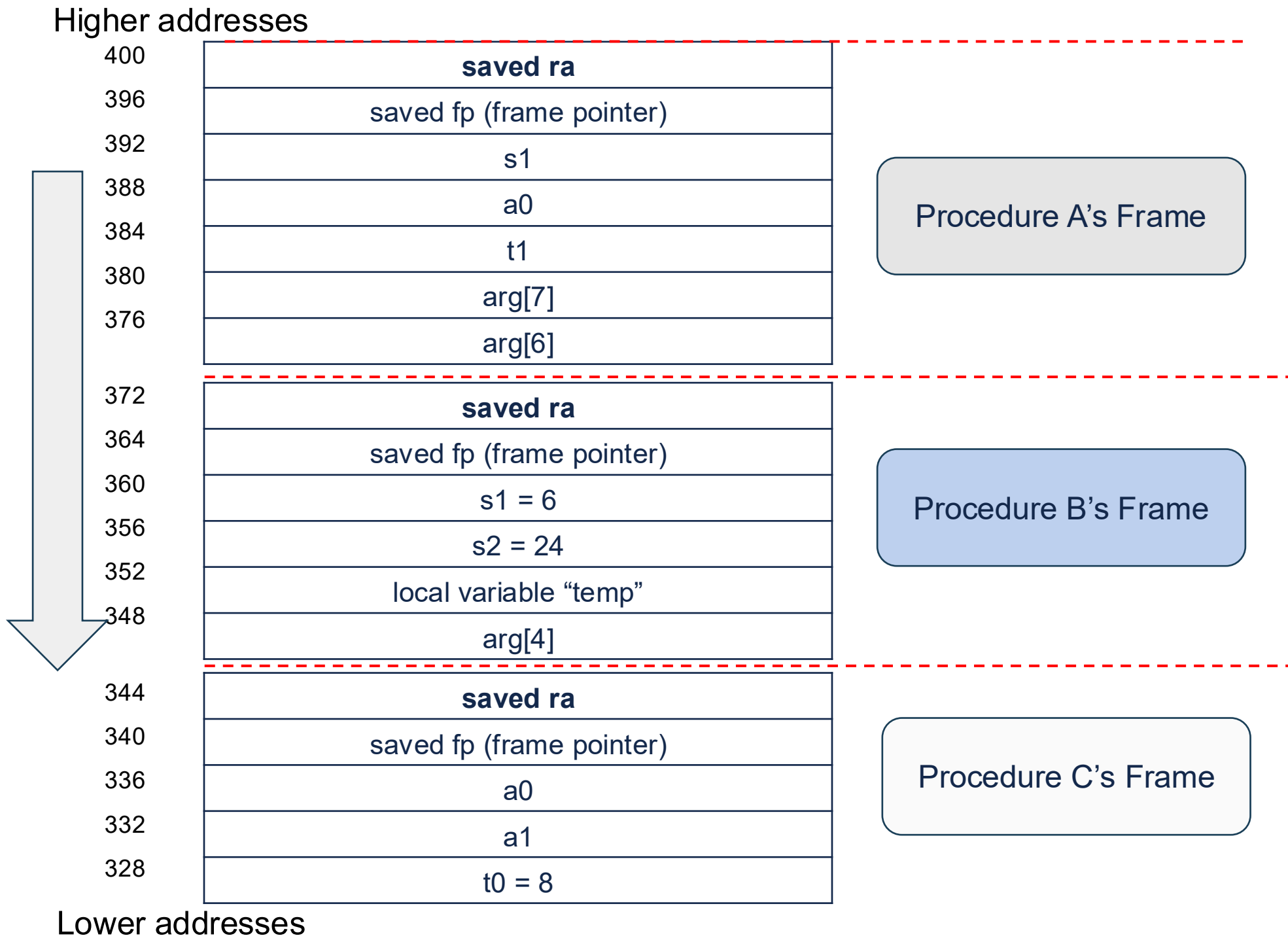
Think of sp as “where the stack is now”
and fp as “where this function’s frame started.”

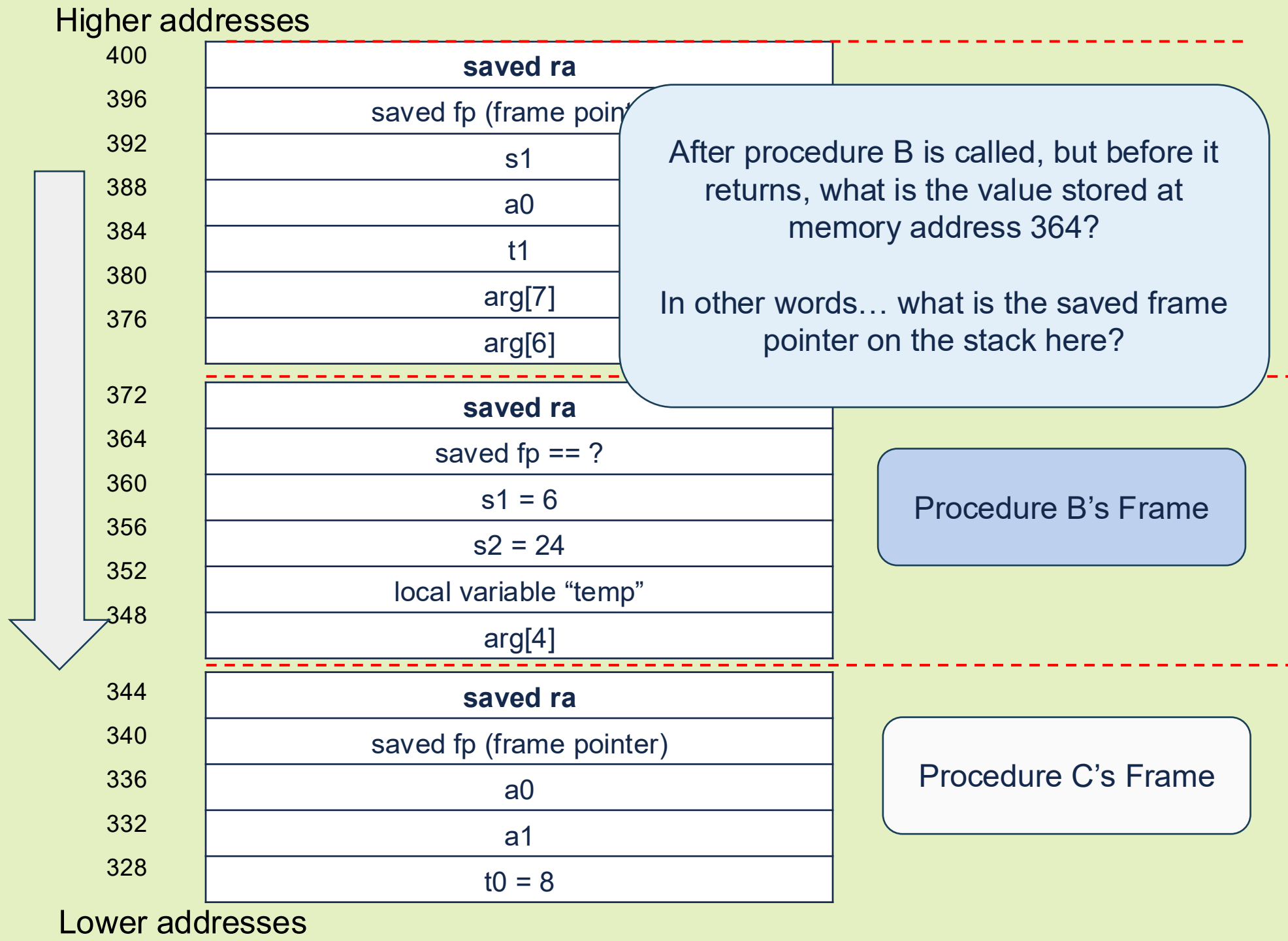


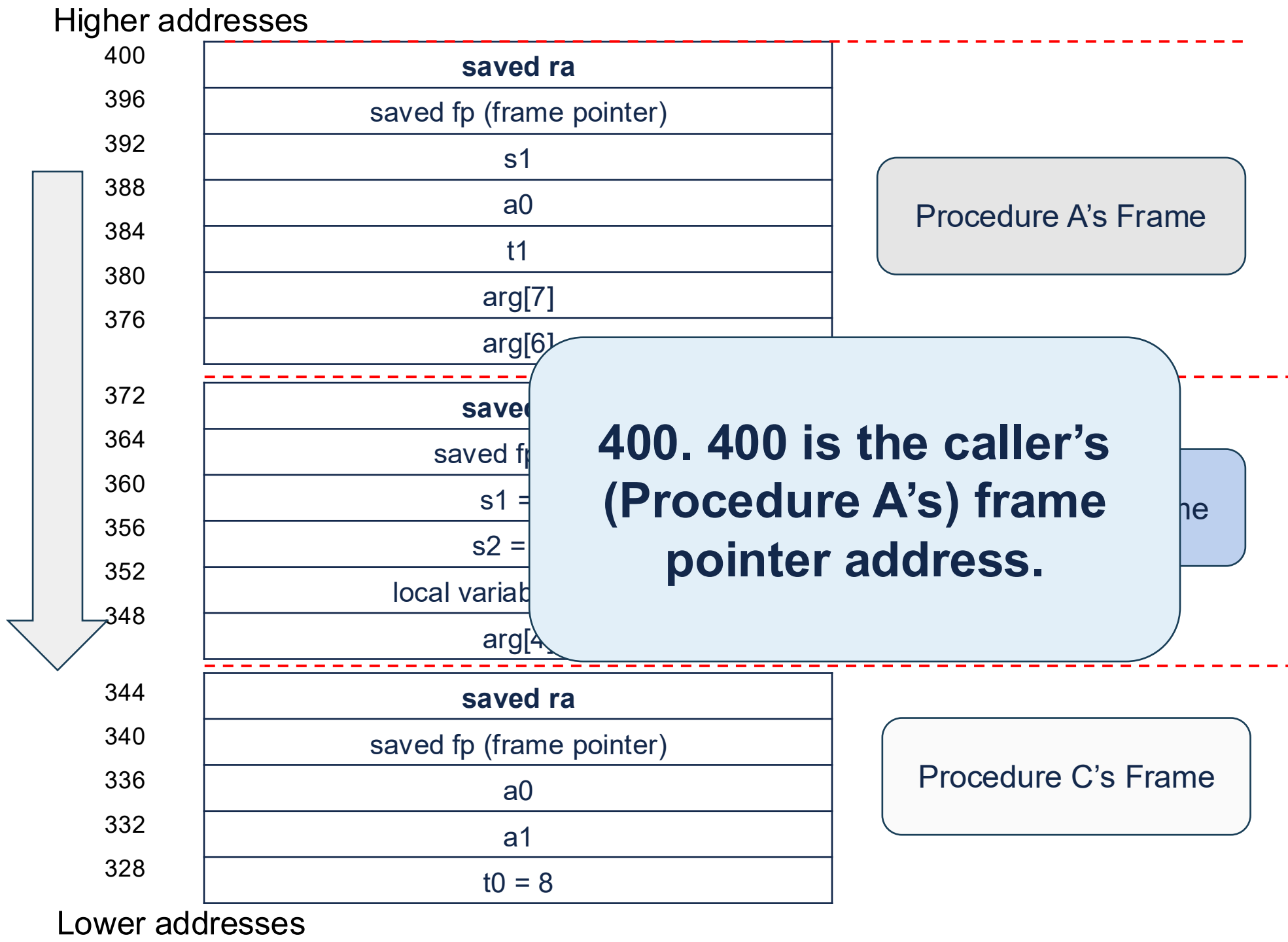
- Stack grows down!
- **Stack pointer (register sp/x2)**: points to the **current top (lowest address) of the stack** and changes as values are pushed or popped.
- **Frame pointer**: (fp/x8/s0) Provides a **stable** reference point within the **current stack frame** so data can be accessed with fixed offsets.









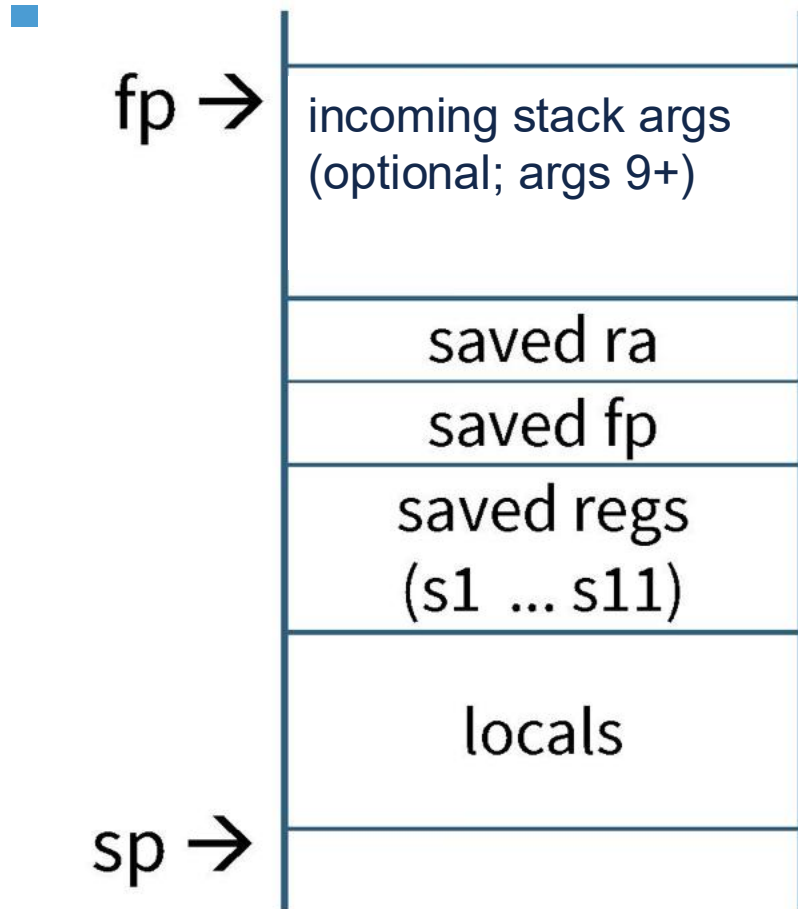


Code interacts with the stack in 3 main ways



- At the beginning of the function, it will “*push* a stack frame onto the call stack” by moving `sp` downward to make space for its own stack frame.
- During the execution of the function, it will use (positive) offsets from `sp` or `fp` to locate each of its local variables.
- At the end of the function, before it returns, it will “*pop* the stack frame off the call stack” by moving `sp` back up to wherever it used to be, “destroying” its stack frame.

RISC-V Calling Convention Cheat Sheet/Summary



- *first eight* args passed in registers `a0`, `a1`, ..., `a7`
- Space for args passed in *child's* stack frame
- return value (if any) in `a0`, `a1`
- stack frame at `sp`
 - contains `ra` (clobbered on JAL to sub-functions)
 - contains `fp`
 - contains *local vars* (possibly clobbered by sub-functions)
 - contains *space for incoming args*
- *Saved registers* (callee save regs) are *preserved*
- *Temporary registers* (caller save) regs are *not*

RISC-V Registers

- Return address: `x1` (`ra`)
- Stack pointer: `x2` (`sp`)
- Frame pointer: `x8` (`fp/s0`)
- First eight arguments: `x10-x17` (`a0-a7`)
- Return result: `x10-x11` (`a0-a1`)
- Callee-save free regs: `x18-x27` (`s2-s11`)
- Caller-save free regs: `x5-x7`, `x28-x31` (`t0-t6`)
- Global pointer: `x3` (`gp`)
- Thread pointer: `x4` (`tp`)

Source:

<https://www.cs.cornell.edu/courses/cs3410/2025fa/notes/asm-call.html>

Conclusions



- Need a convention (contract) between caller and callee
- Implement stack for storing each procedure's variables
- Procedure calls can now be arbitrarily nested
 - Recursion possible too
- FOLLOW the convention meticulously when programming!



UNIT 1: PROGRAMMING THE MACHINE

Stack Practice