CS2200
Systems and Networks
Spring 2022

Lecture 3: Processors (cont'ed)

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Leahy, Lively, Ramachandran of Georgia Tech

Announcements

- First lab tomorrow
  - Intro to CircuitSim
- Project 1 released on Friday
  - Duration: 3 weeks

Building an ISA – so far

Software	Hardware
Expressions & assignments	ALU instructions
Variable reuse	register addressing mode ld/st instructions
Data abstraction • struct • array	 base + offset addr mode base + index addr mode
Granularity of operands	ldb/ldh/ldw instructions addressability (byte, word)
Packing operands	Memory alignment (space/time tradeoff)
Endianness 0x11223344	Little (first byte is 0x44) / Big (first byte is 0x11)

What do we need for…

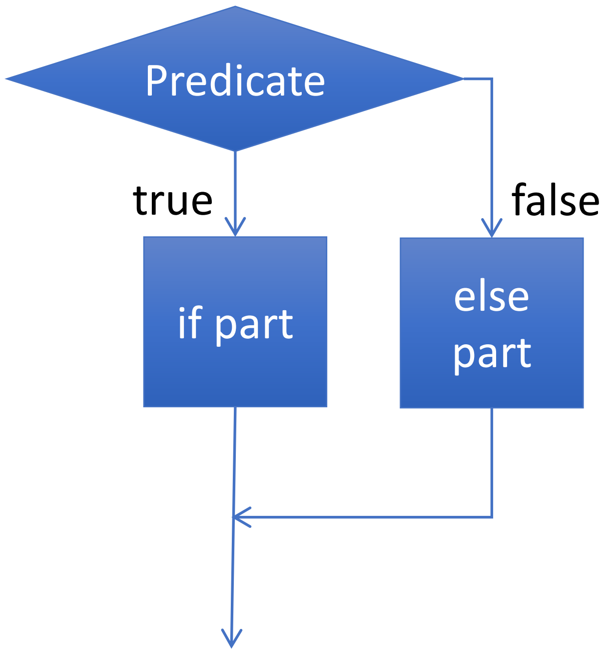
- Conditional statements
- Switch statements
- Loops
- Procedure calls
- Other considerations for ISA

Compiling Conditional Statements

- In what order are program statements normally executed?
- How do we know what instruction to execute next?
- How can we handle this high-level language construct:

```
if(x == y) z = 7;
```

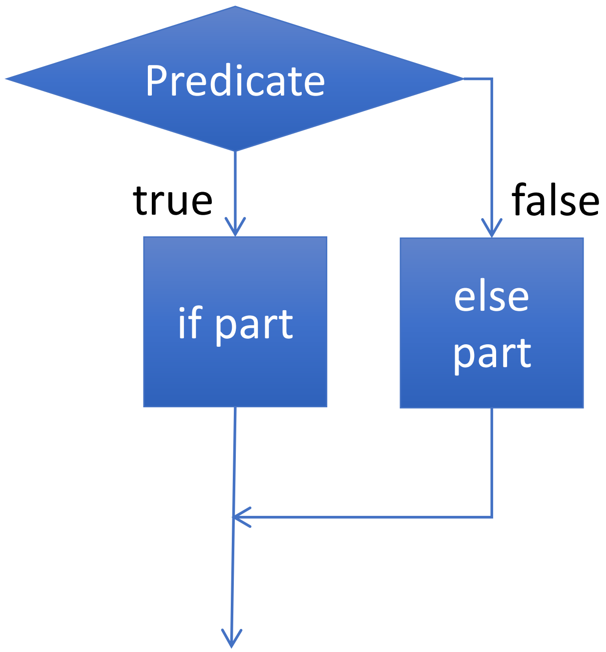
What Do We Need to Do?



- Evaluate predicate
- Break the sequential flow of instructions
- Rejoin control path

Implementing a Conditional

- Evaluate predicate
  - ALU Op
- Break sequential flow
  - Need to know where we are
    - ➔ PC
  - Need a new instruction
    - ➔ BEQ r1, r2, offset
    - ➔ if r1 == r2 then PC = PC + offset
      else do nothing
    - ➔ PC relative addressing mode!
- Rejoin control flow
  - ➔ need an unconditional jump



An Example

- C

```
if(a == b)
  c = d + e;
else
  c = f + g;
```

- Assembly

```
     beq r1, r2, then
     add r3, r6, r7
     beq r1, r1, skip*
then add r3, r4, r5
skip …
```

* Effectively an unconditional branch

Assuming
r1 = a
r2 = b
r3 = c
r4 = d
r5 = e
r6 = f
r7 = g

Outcome of Conditional Statements

- Introduction of PC
- One new instruction
  BEQ r_1, r_2, offset
- One new addressing mode: PC-relative
- (optional) an Unconditional Jump
  J r_n ; PC $\leftarrow$ r_n
- Do we really need an unconditional jump??

Compiling Switch Statements

```
if (n==0)
      x=a;
else if (n==1)
      x=b;
else if (n==2)
      x=c;
else
      x=d;
```

```
switch (n) {
      case 0:
            x=a;
            break;
      case 1:
            x=b;
            break;
      case 2:
            x=c;
            break;
      default:
            x=d;
}
```

Do these produce essentially equivalent assembly code?

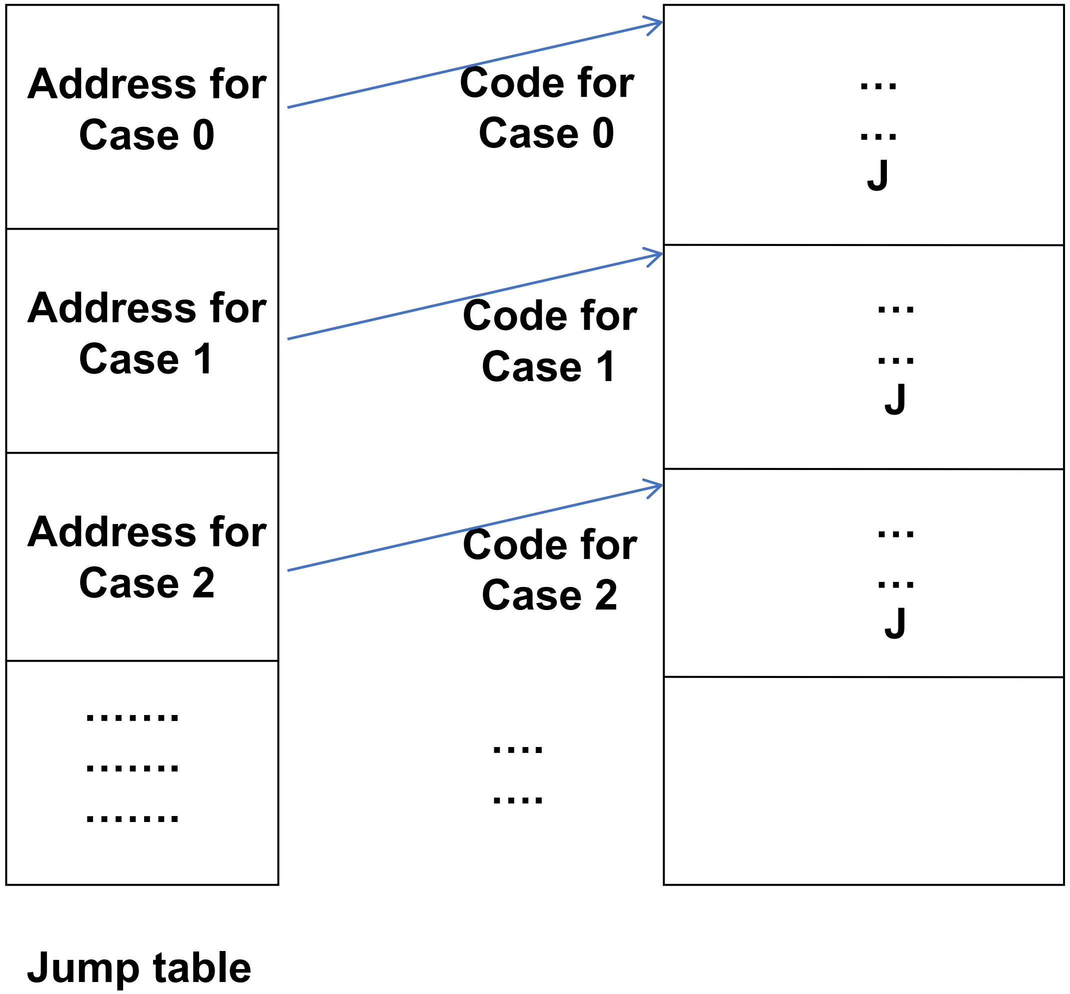
They can, but they don't have to!

Switch Can Use a Jump Table

- Think of a C array of pointers to the individual cases
- To do this we need an indirect addressing mode

  J @r_1

➔ PC ← Mem[r_1]



Loops

- Do we need anything new in the ISA?
- Not really.

Compiling Loops

- C

```
while(j ! = 0)
{
    /* loop body */
    t = t + a[j--];
}
```

- Assembly

```
loop   beq r1,r0,done
          ; loop body
          …
          beq r0, r0, loop
done  …
```

Summary

Software	Hardware
Expressions & assignments	ALU instructions, LD/ST instructions
Data abstraction • struct • array	register addr mode base + offset addr mode base + index addr mode
Conditionals & Loops	PC-relative addr mode branch/jump instruction (register or PC-relative) Indirect addr mode (optional)

How Do We Compile Function Calls?

State of Caller
Pass parameters
Remember return addr
Jump to procedure

```
int main()
{
   <decl local-variables>

   return-value = foo(actual-parms);
   /* continue upon
    * returning from foo
    */
}
```

Save the result
Continue the program

Caller

Allocate space for local vars

```
int foo(formal-parameters)
{
   <decl local-variables>

   /* code for function foo */
   return(<value>);
}
```

Pass result to caller
Return to caller

Callee

Remembering the Return Address

- Have we needed to do this before?

- Add a Jump & Link instruction
  - JALR r_{target}, r_{link} ; r_{link} <= PC, PC <= r_{target}

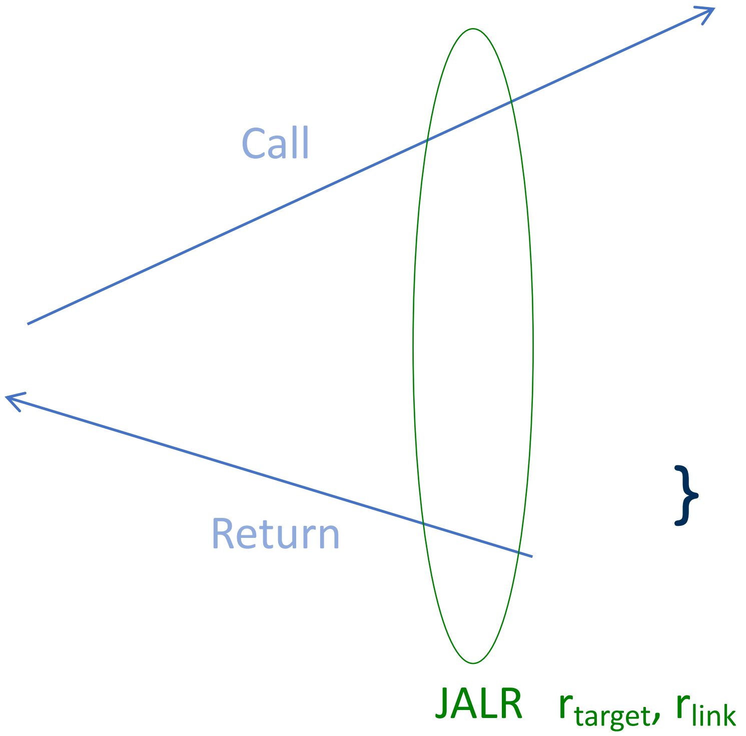
- Recall

  J r_{target} ; PC <= r_{target}

- Do we need this instruction anymore?

Control Flow

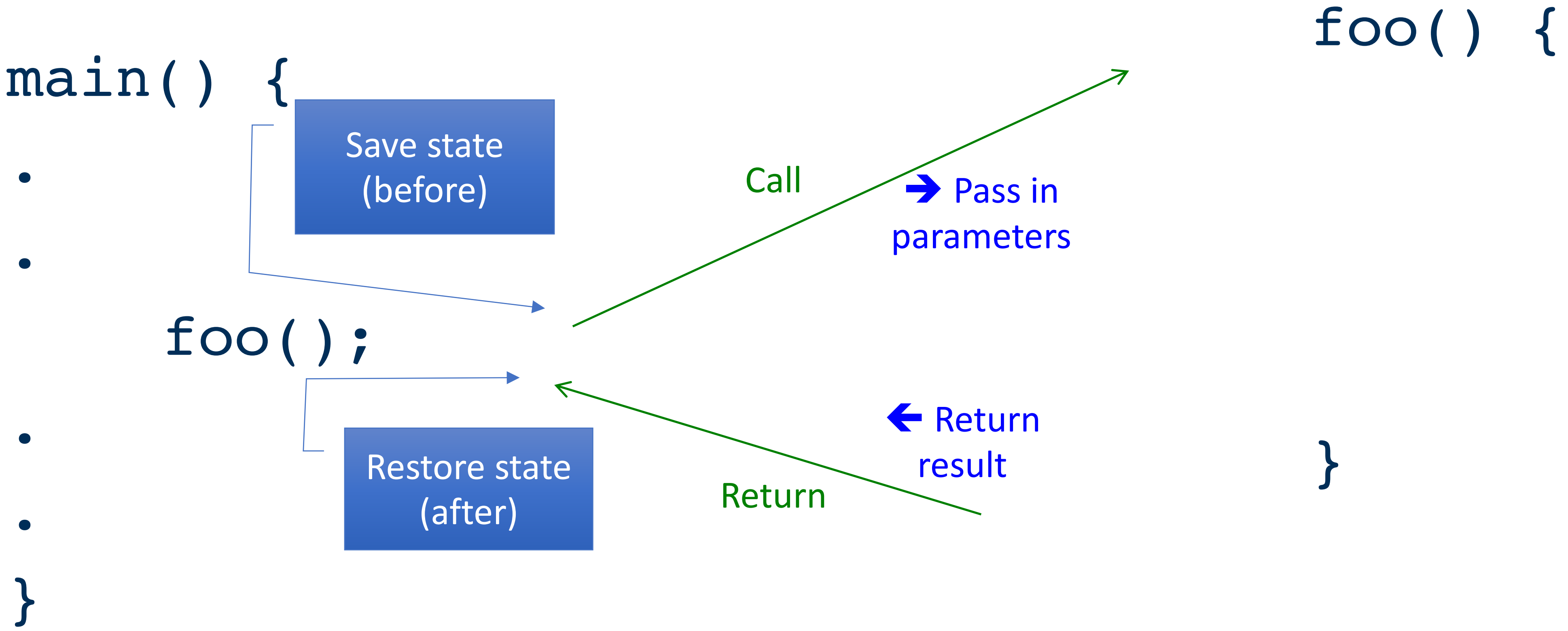
```
main() {
.
.
    foo();
.
.
}
```



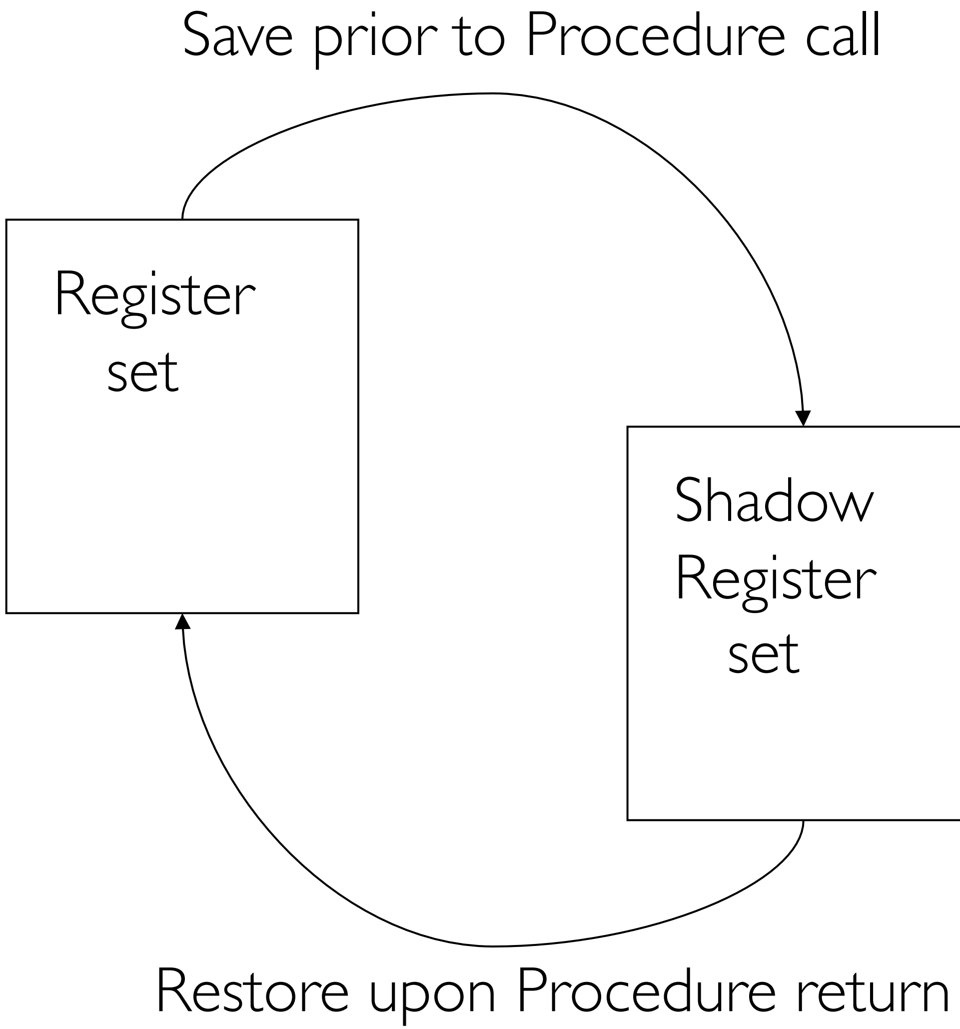

```
foo() {

}
```

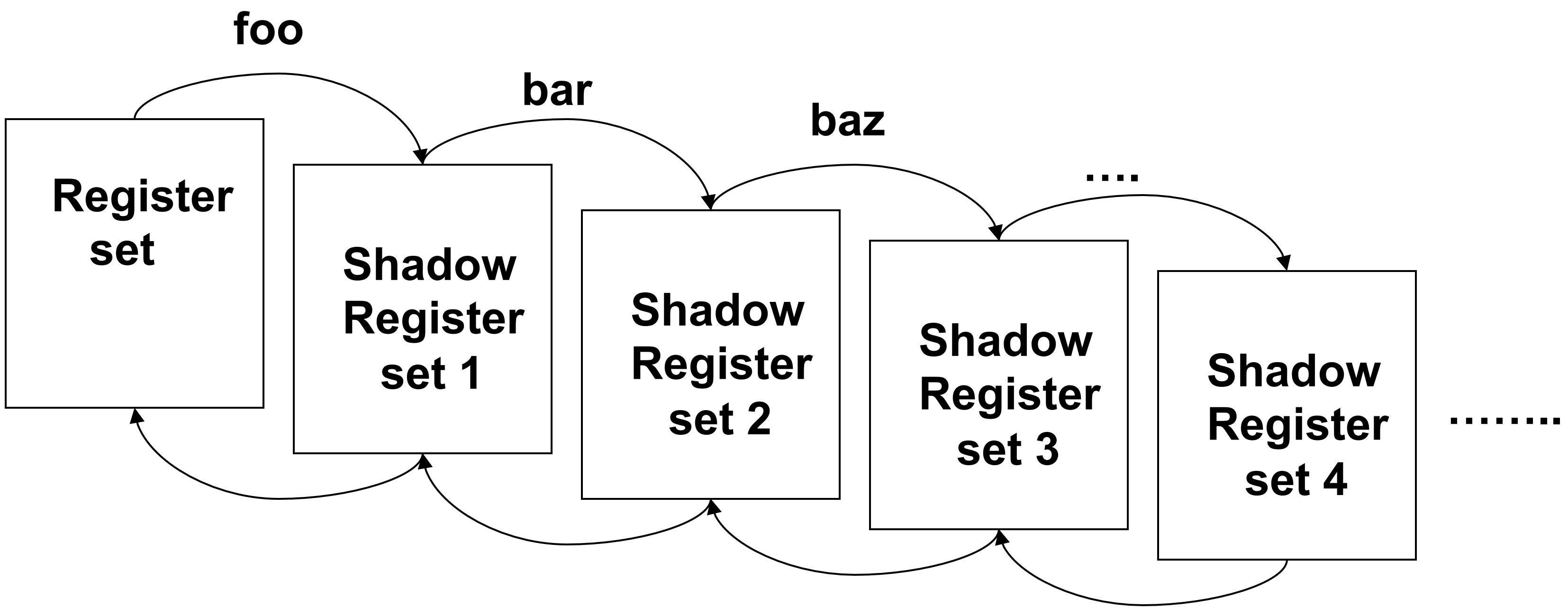
Control Flow



Another Way to Save State



Shadow Register Sets



- foo() calls bar() who calls baz(), etc.
- The Big Deal: No memory accesses!
  (but we need lots of extra registers)
- Another form of this is called **register renaming**

Saving State

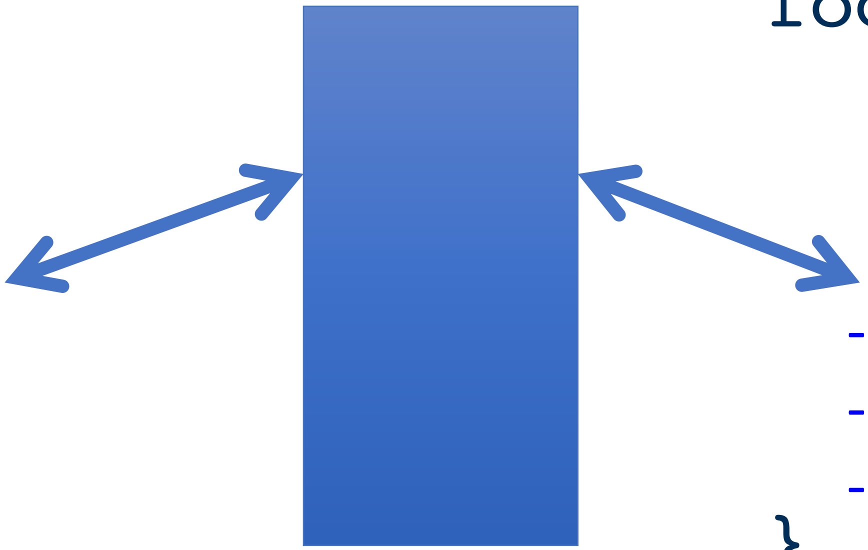
- If we don't have shadow registers, where are we going to save all that state?
- A stack

Where are we going to put the stack?

- In memory
- But in small cases, could we hold the state in a few extra registers? (another space/time tradeoff)

Use a Stack to Communicate

```
main() {
.
.
    foo();
.
.
}
```



```
foo() {

}
```

- Save/restore state
- Pass parameters
- Return results

- What else is needed in ISA?
- Nothing new..

Saving Registers During a Procedure

- We can have the **caller** save all the registers

  -or-

  We can have the **callee** save all the registers
- What's wrong with those choices?
  - Not everything needs to be saved every time…

Saving Registers During a Procedure

- If we split the assignment of the registers, then most of the time, the **caller** and **callee** can each save fewer registers based on what they actually need to use

- In the LC-2200 case, we'll functionally divide the working register set
  - **s0-s2** registers which the **callee** must preserve if it wants to use them
  - **t0-t2** registers which the **caller** must preserve if it wants their values to persist over a function call

- This division of responsibility saves memory accesses.

Stack as Communication Area

- Saving/restoring state over a procedure call

    Who does it?

    → Split between Caller and Callee

Stack as Communication Area

- Returning results

  Do we really need to put them on the stack?

  ➔ Use registers
  (We'll call this register v0)

Stack as Communication Area

- Parameter Passing

  Do we really need to put them on the stack?

  ➔ Use registers
  (We'll call these registers **a0-a2**)

Stack as Communication Area

- Will we need the stack at all for parameters and results?
- What if we run out of registers?
- We use the stack if we run out
- Here we're trading time for complexity

Moral of the Story

- Use the stack sparingly
  - LD/ST instructions are expensive
    (i.e., memory access is slow)

- Software calling convention
  - Used by the compiler to keep track of the use of the stack and registers
  - Better have one!

Software Convention for LC-2200

Use:
Program
Data

- Registers s0-s2 are the caller's saved registers
- Registers t0-t2 are the temporary registers
- Registers a0-a2 are the parameter passing registers
- Register v0 is used for return value

Use:
Bookkeeping

- Register ra is used for return address (r_{link})
- Register at is used for target address (r_{target})
- Register sp is used as a stack pointer

Session ID: dag2200

Review Question 1

Saving and restoring of registers on a procedure call…

22% A. Is always done by the caller.

22% B. Is always done by the callee.

22% C. Is never done explicitly since hardware magically takes care of it.

11% D. Is done on a need basis partly by the caller and partly by the callee.

22% E. What is a caller/callee?

Review Question 2

On the LC-2200, how are actual parameters passed to a function?

A. On the stack.
B. On the heap.
C. Up to 3 in registers, the rest on the stack.
D. Up to 6 in registers, the rest on the stack.
E. None of the above.

Session ID: dag2200

Review Question 3

We store some values in registers during a procedure call...

0% A. Because we like to mix things up – variety is good!

50% B. Because it reduces memory references.

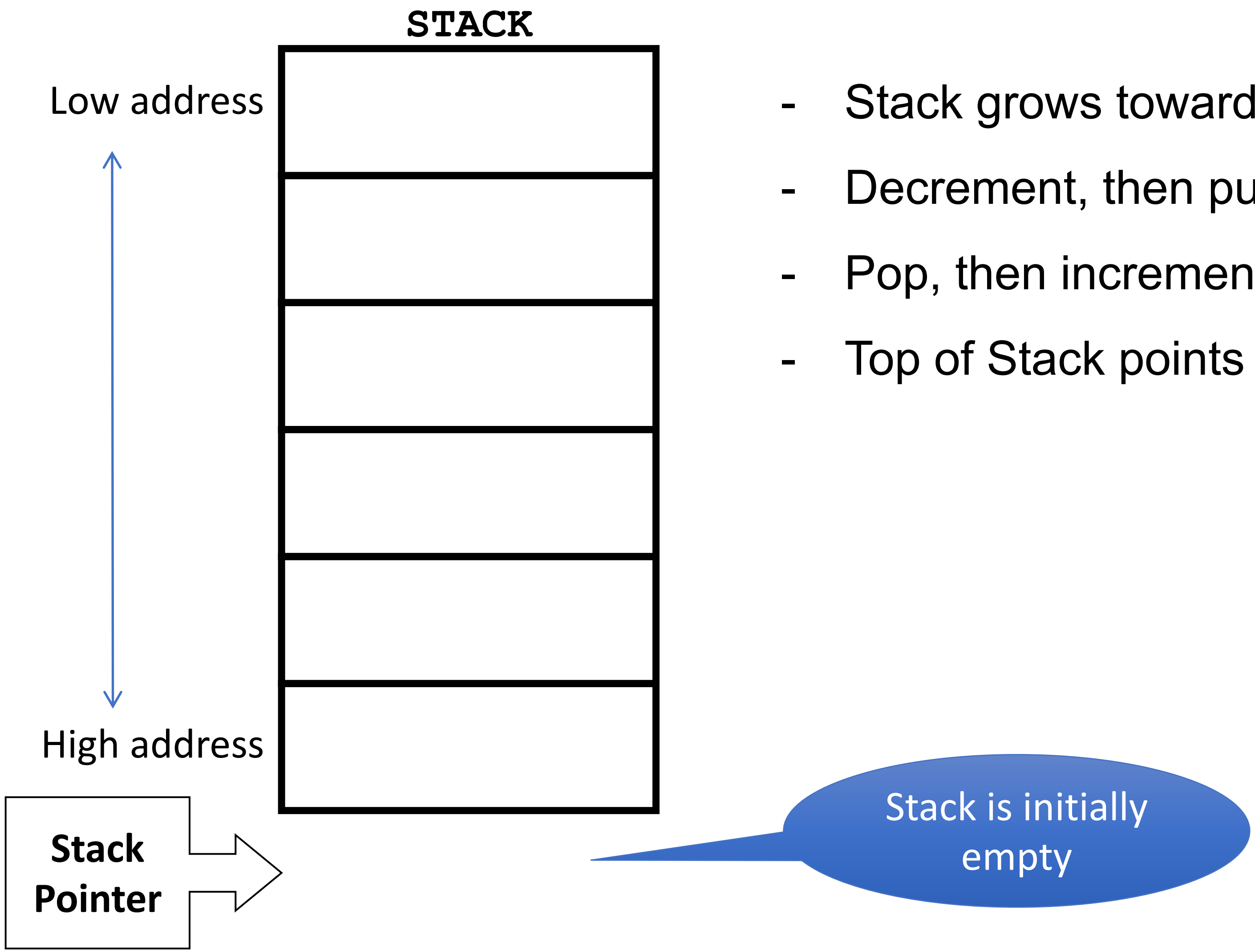
50% C. It makes the stack shorter so it reduces the danger of overflow.

0% D. It results in prettier code.

Activation Record

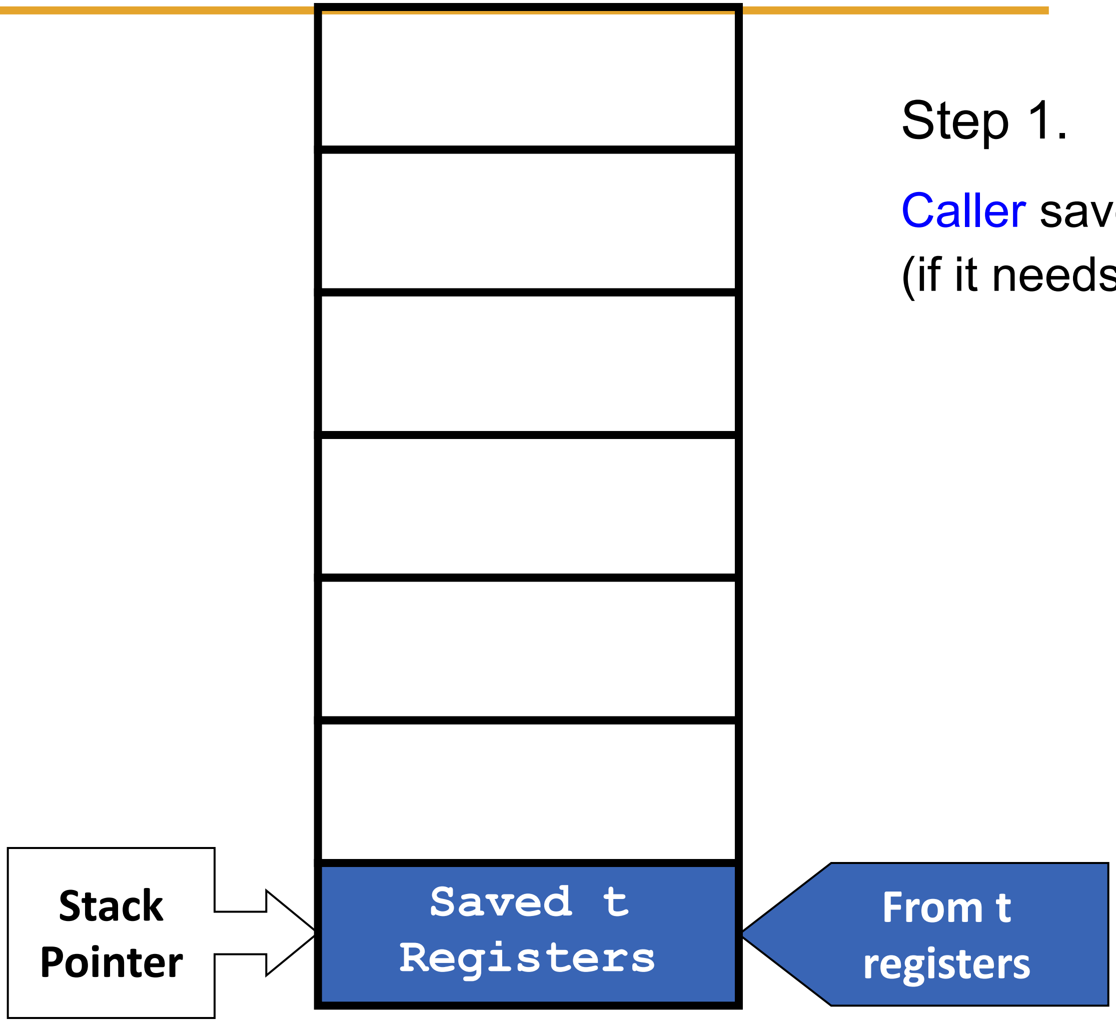
- Also known as a Stack Frame
- It's the space used by the caller and callee during the execution of a procedure call
- Used to store…
  - Caller saved registers
  - Additional parameters
  - Additional return values
  - Return address
  - Callee saved registers
  - Local variables

Stack Conventions



- Stack grows toward lower memory addresses
- Decrement, then push
- Pop, then increment
- Top of Stack points to last item placed in it

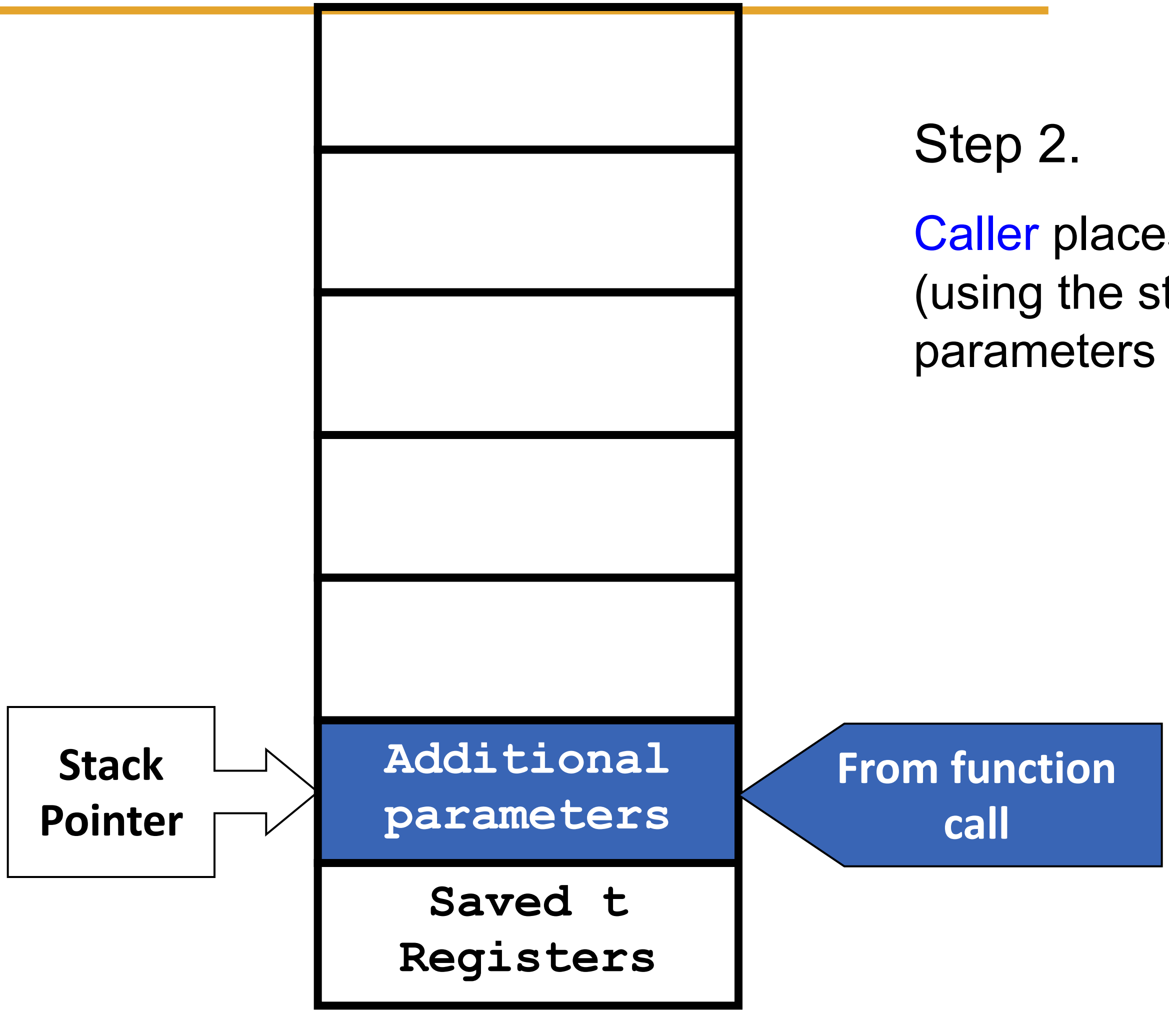
STACK



Step 1.

Caller saves any of registers t0-t2 on the stack (if it needs the values in them upon return).

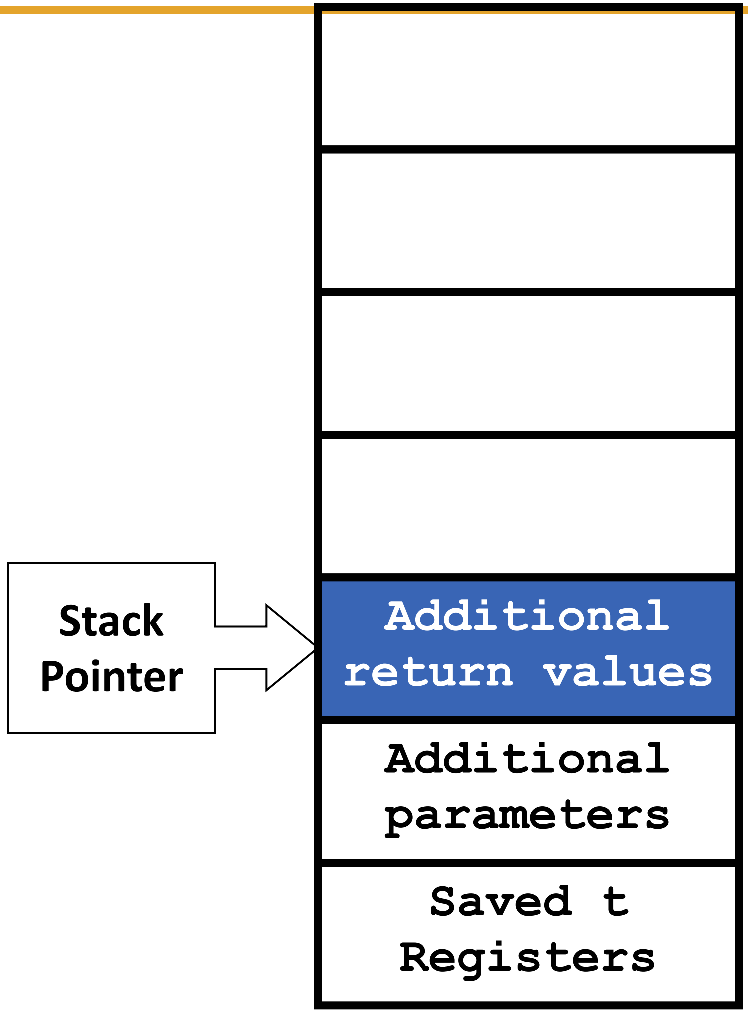
STACK



Step 2.

Caller places the parameters in a0-a2 (using the stack for additional parameters if needed).

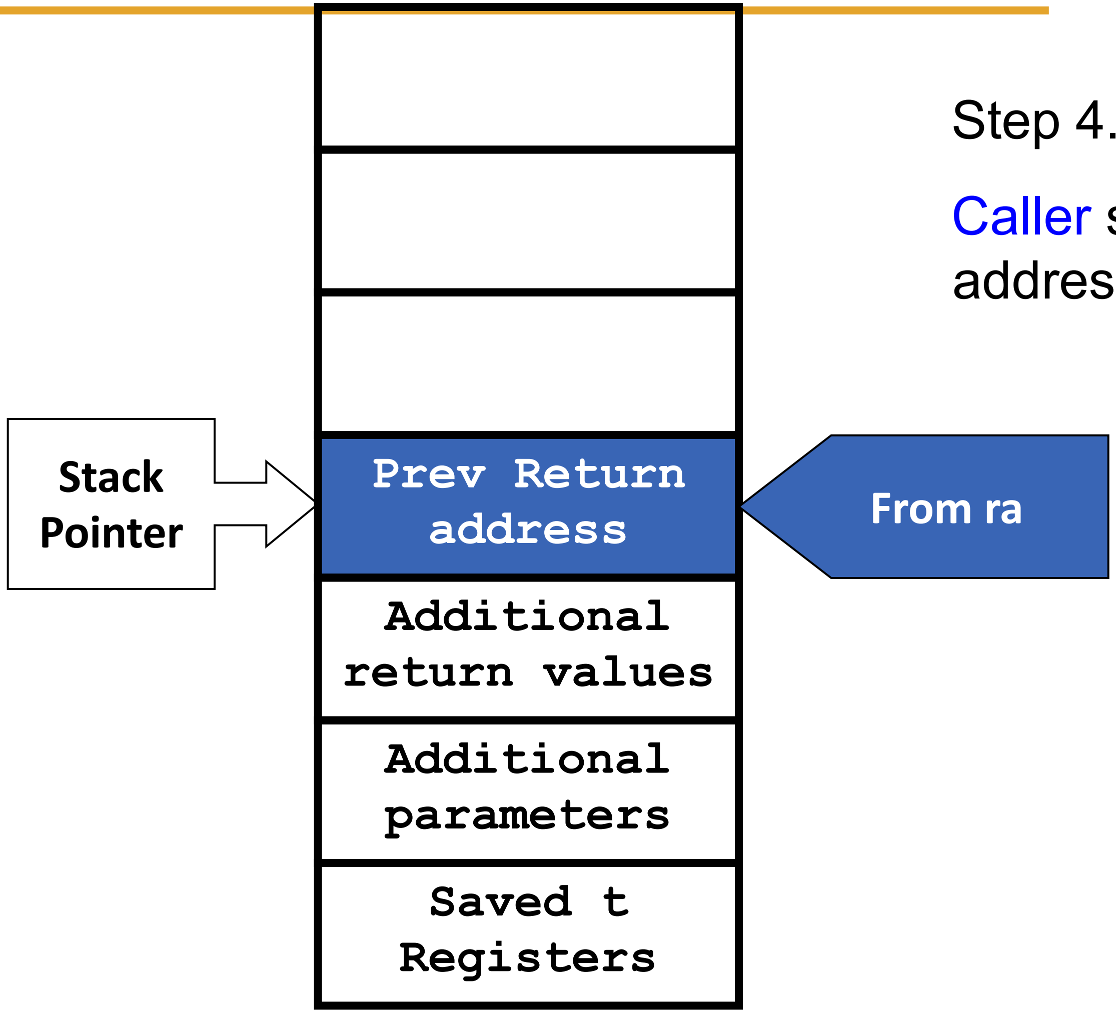
STACK



Step 3.

Caller allocates space for any additional return values on the stack

STACK

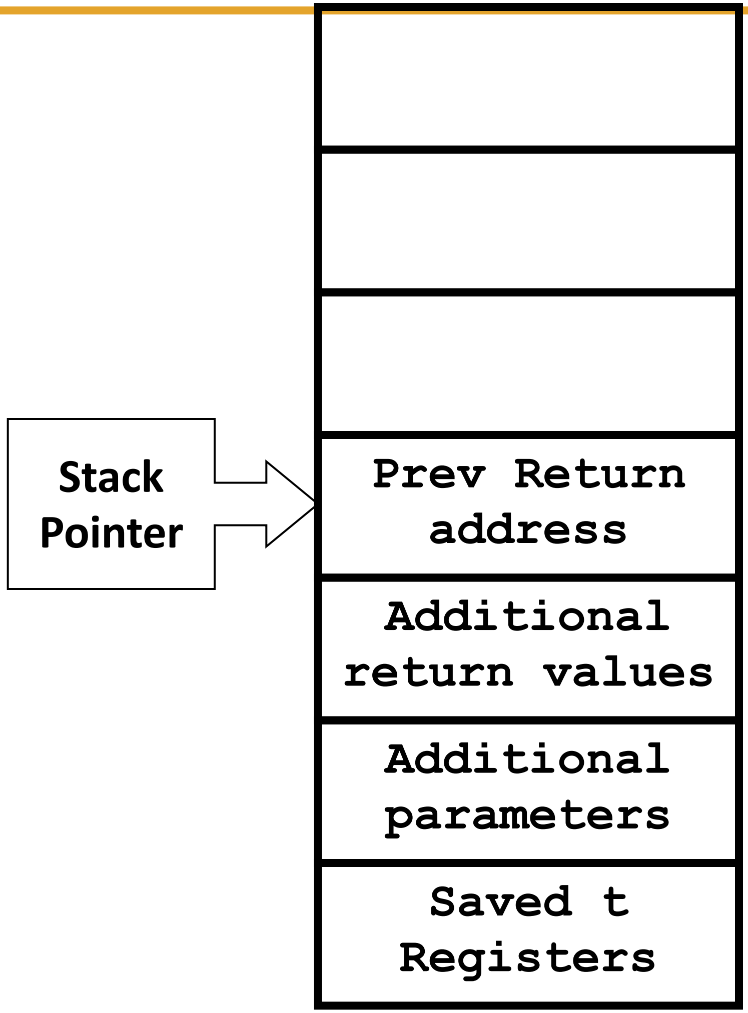


Step 4.

Caller saves previous return address currently in ra

Why?

STACK

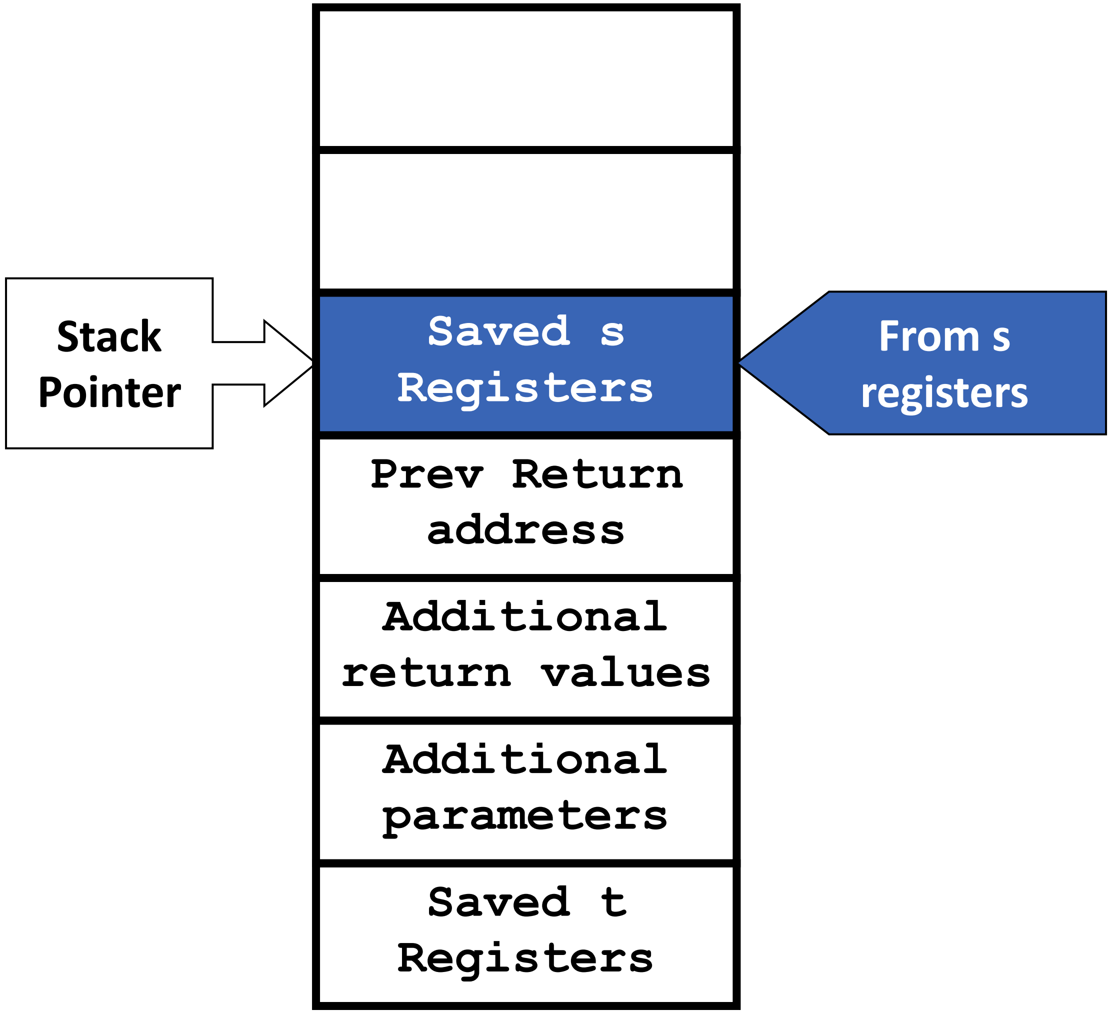


Step 5.

Caller executes JALR at, ra

(no effect on stack)

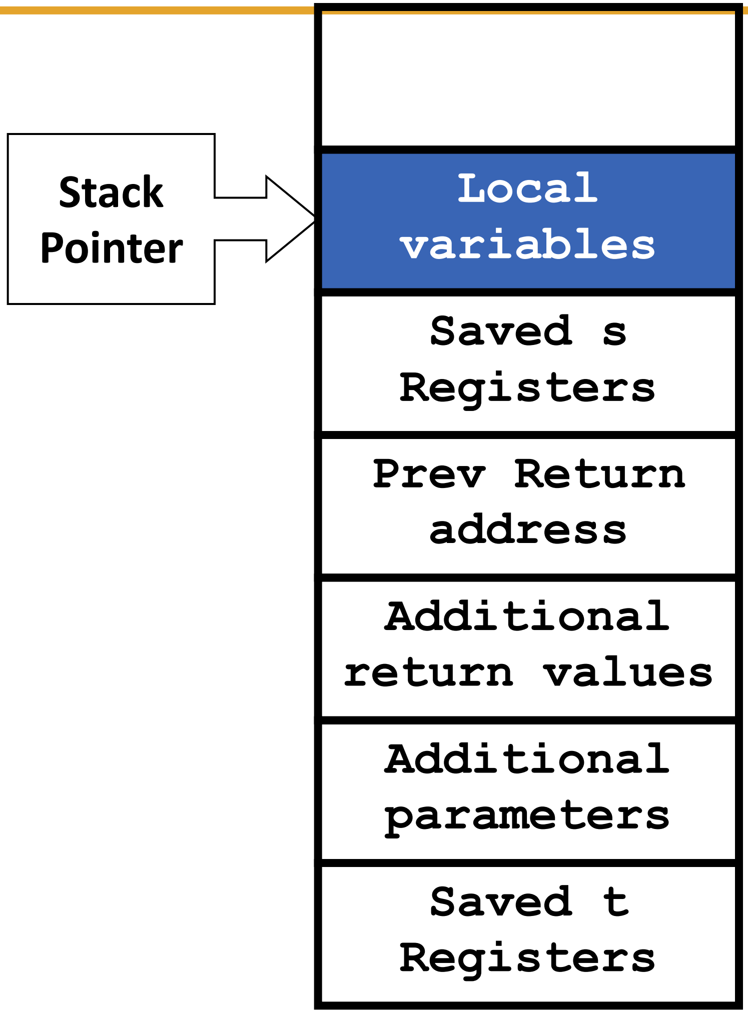
STACK



Step 6.

Callee saves any of registers s0-s2 that it plans to use during its execution on the stack.

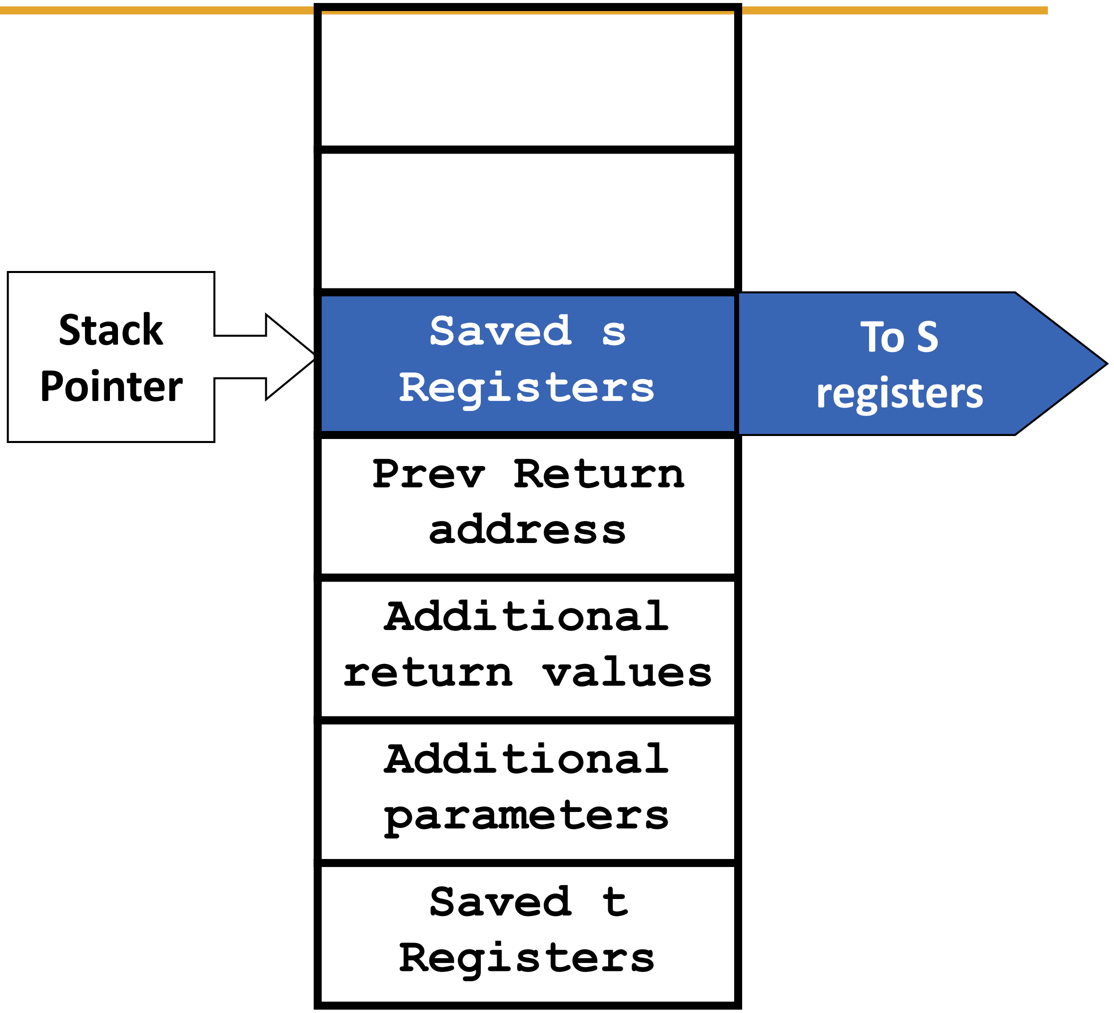
STACK



Step 7.

Callee allocates space for any local variables on the stack

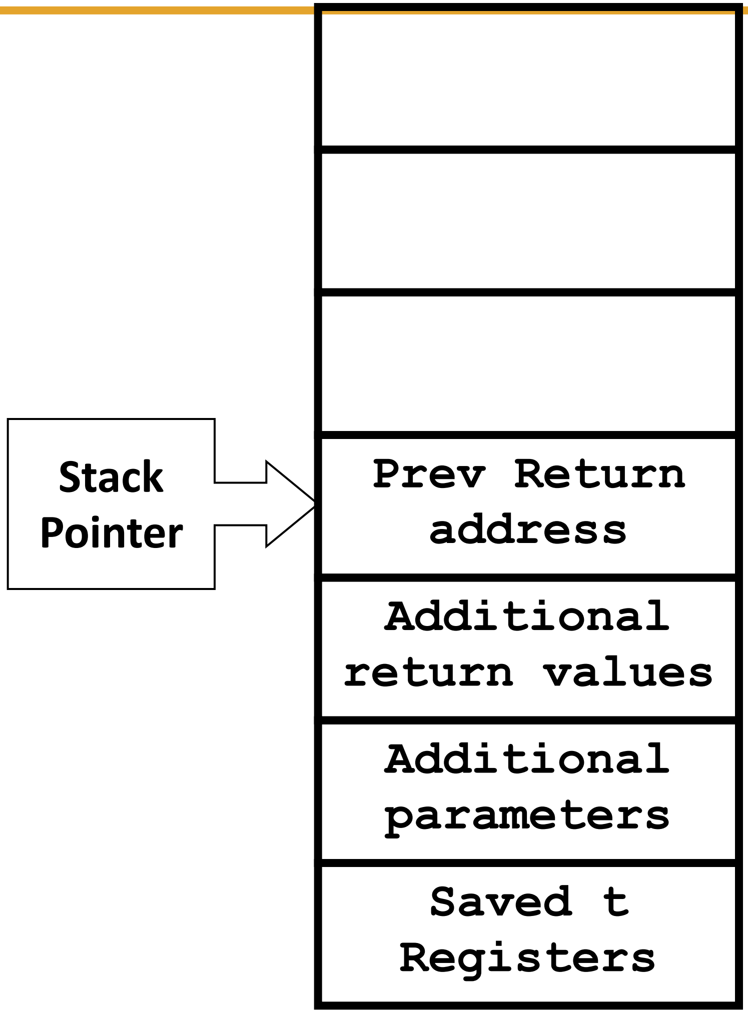
STACK



Step 8.

Prior to return, Callee restores any saved s0-s2 registers from the stack

STACK

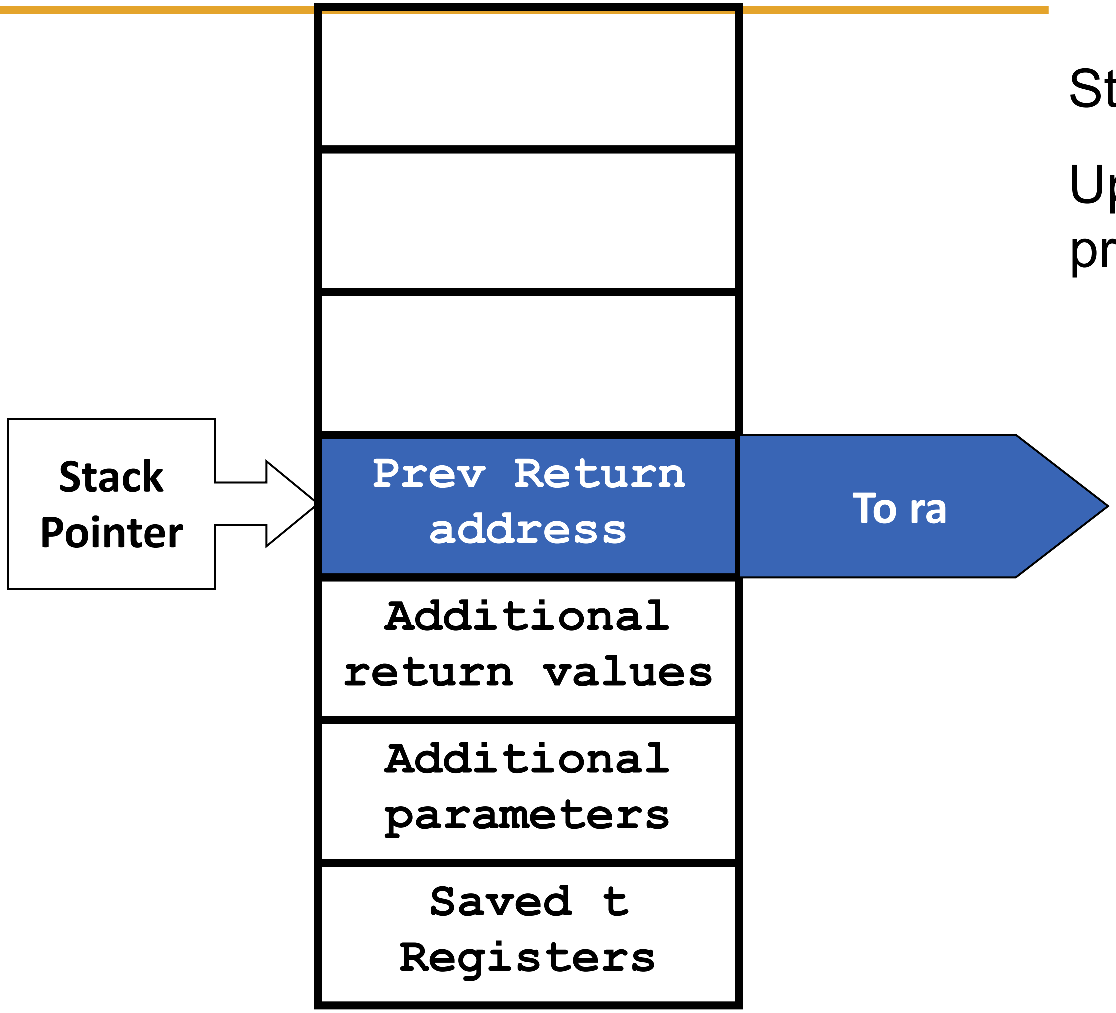


Step 9.

Callee executes jump to ra

No change to stack.

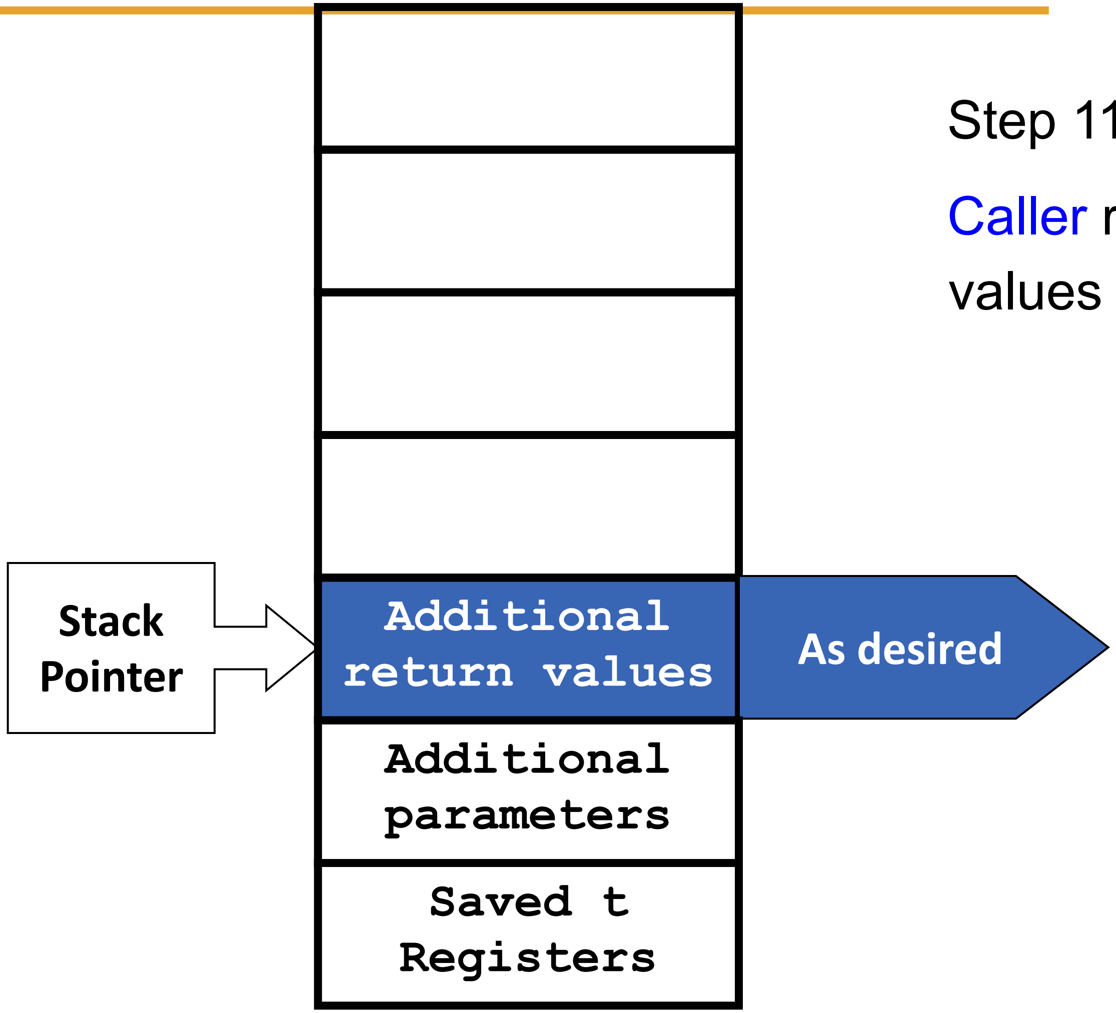
STACK



Step 10.

Upon return, Caller restores previous return address to ra

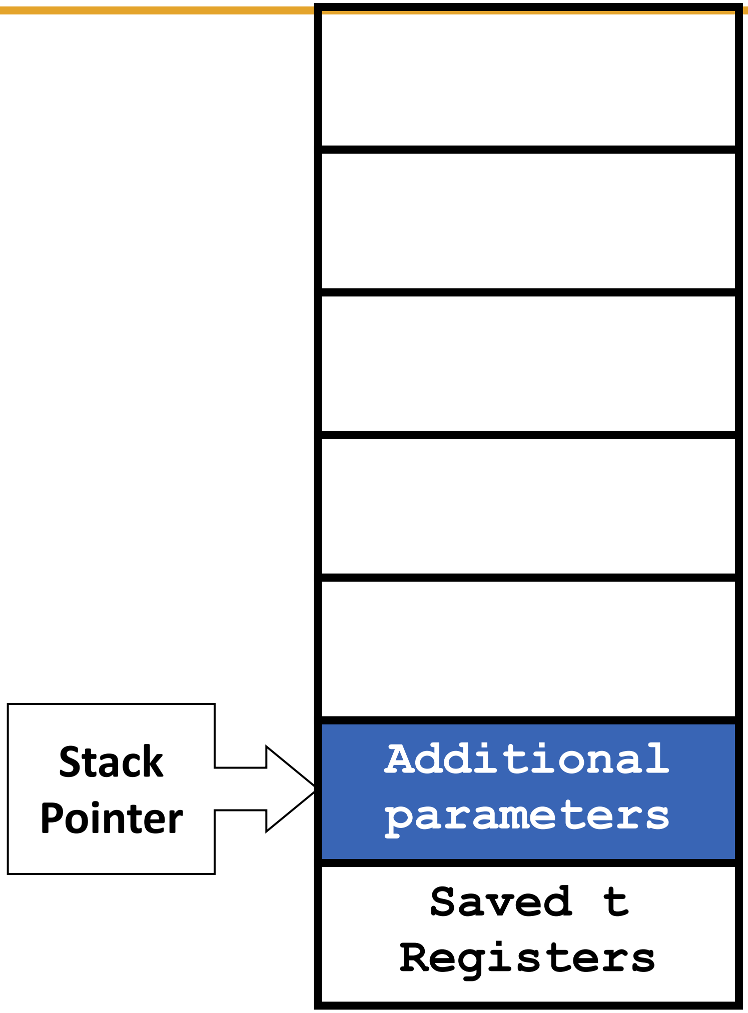
STACK



Step 11.

Caller reads additional return values as desired

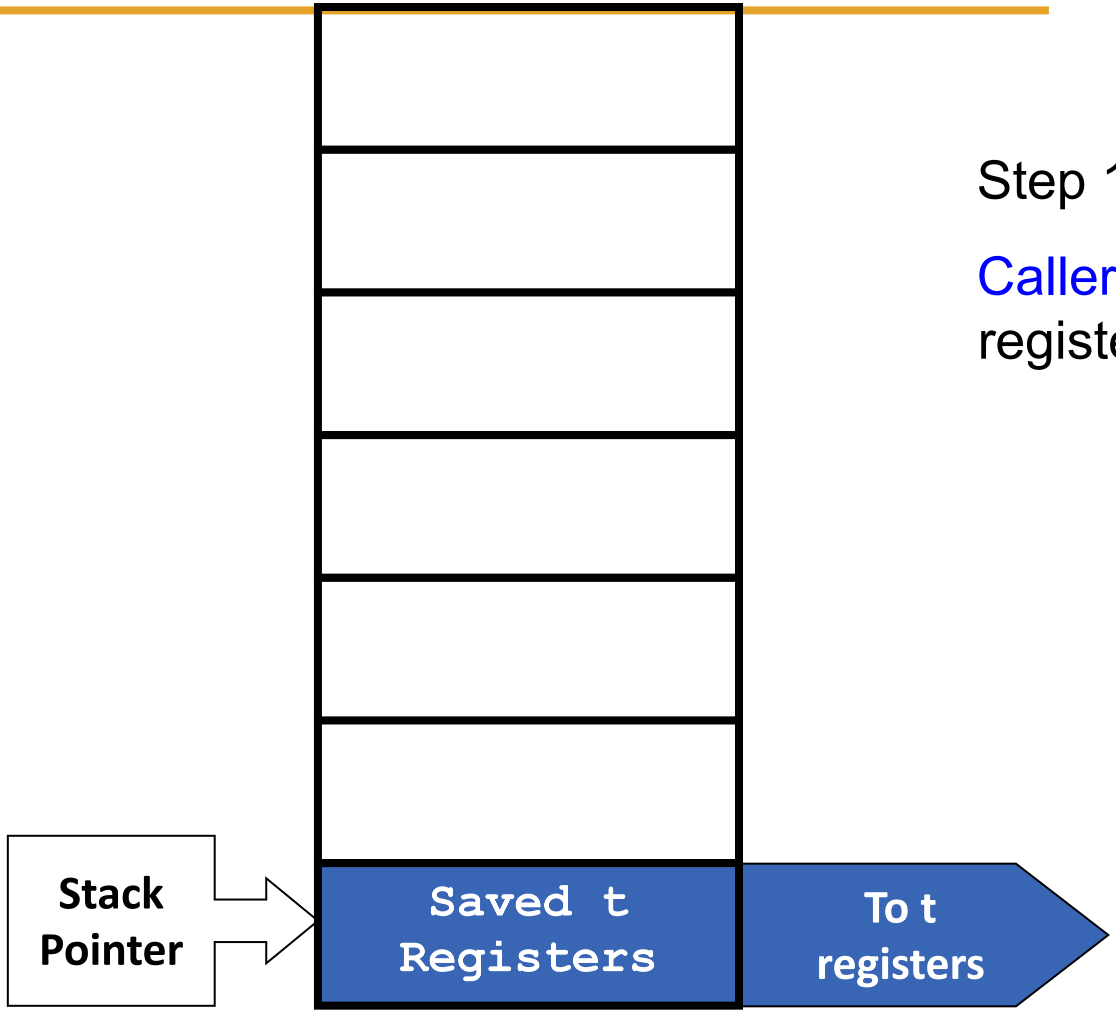
STACK



Step 12.

Caller moves stack pointer to discard additional parameters

STACK



Step 13.

Caller restores any saved t0-t2 registers from the stack

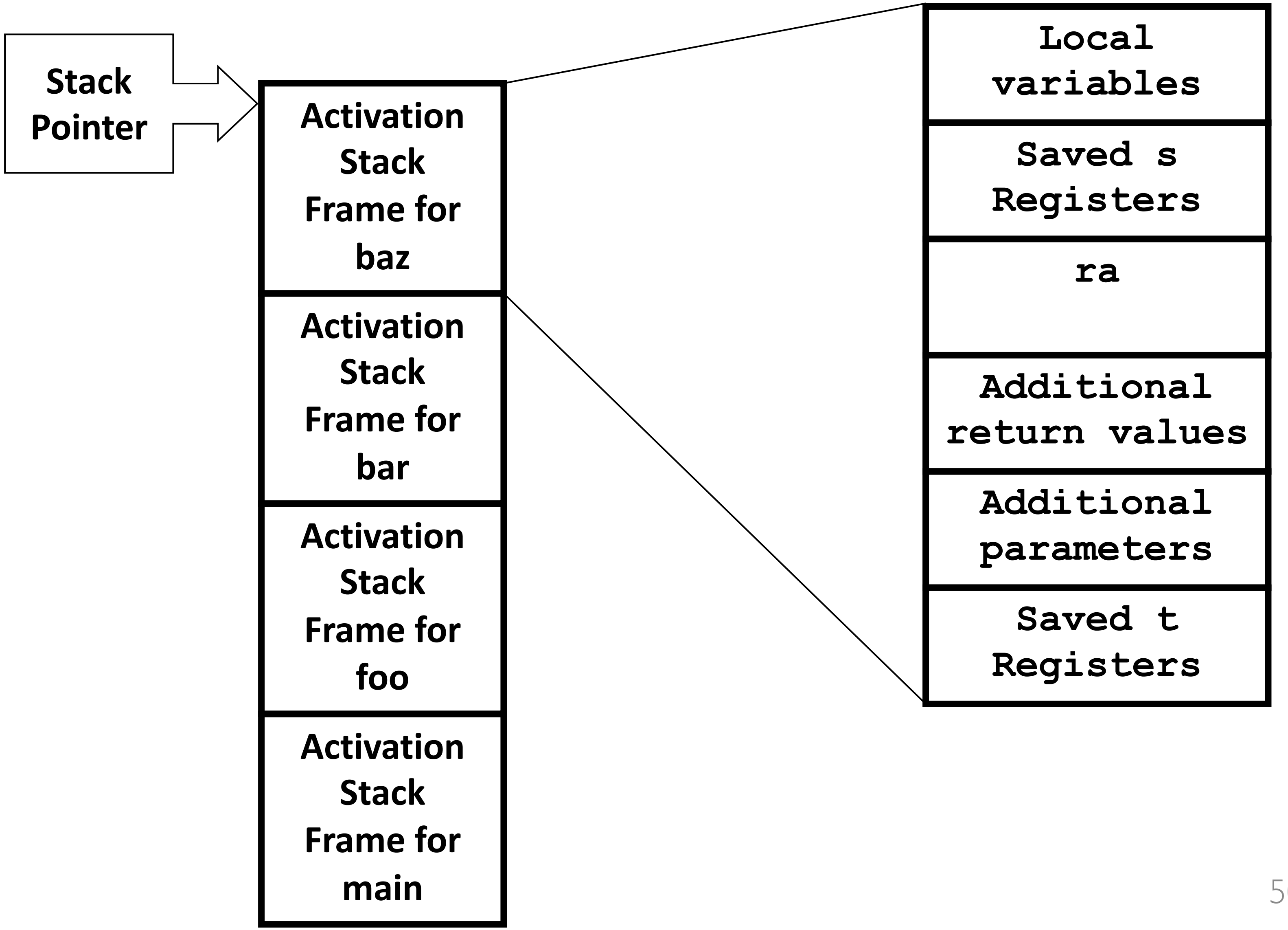
Session ID: dag2200

Local variables in a procedure...

- 20% A. Are usually allocated on the stack.
- 20% B. Are usually kept in processor registers.
- 20% C. Are usually kept in special hardware.
- 20% D. Are usually allocated in the heap space of the program.
- 20% E. None of the above

A Stack of Activation Records

main() ➔ foo() ➔ bar() ➔ baz()



Recursion

- Does recursion require any additional instruction set architecture items?

One More Thing: Frame Pointer

- During execution of given module it is possible for the stack pointer to move.
- Since the location of all items in a stack frame is based on the stack pointer it is useful to define a fixed point in each stack frame and maintain the address of this fixed point in a register called the frame pointer
- This necessitates storing the old frame pointer in each stack frame (i.e., caller's frame pointer)

Why Do We Need a Frame Pointer?

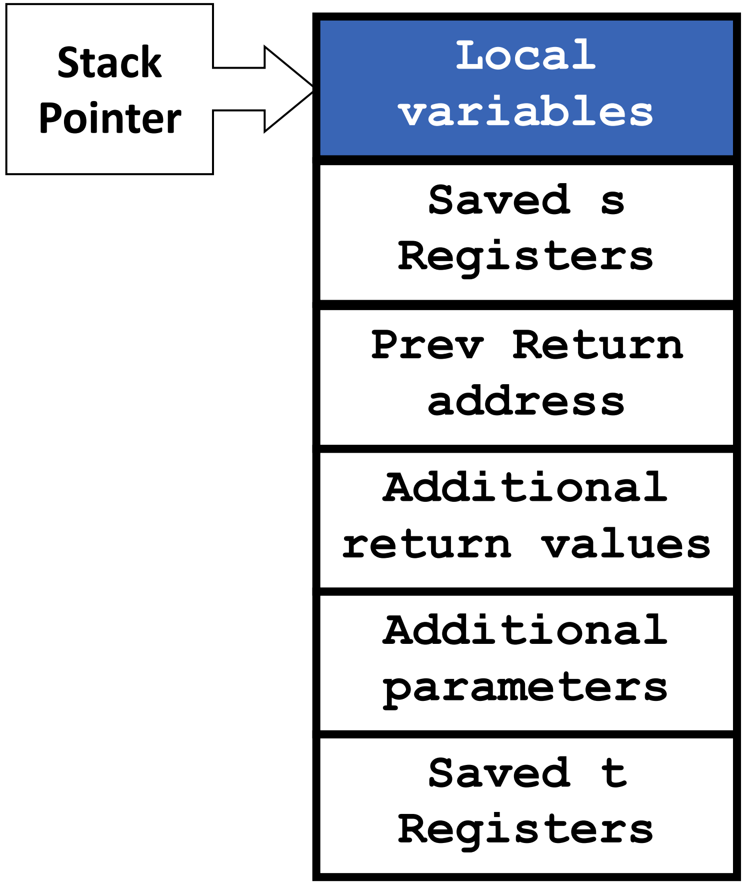
This code will cause us a problem:

```
foo(int p) {
      int a = 1, b = 3;
      if (a != p) {
            int c[p];
            c[p – 1] = b + a;
            …
      }
      b++; a++;
;     …
}
```

a and b are no longer at the same offset from SP

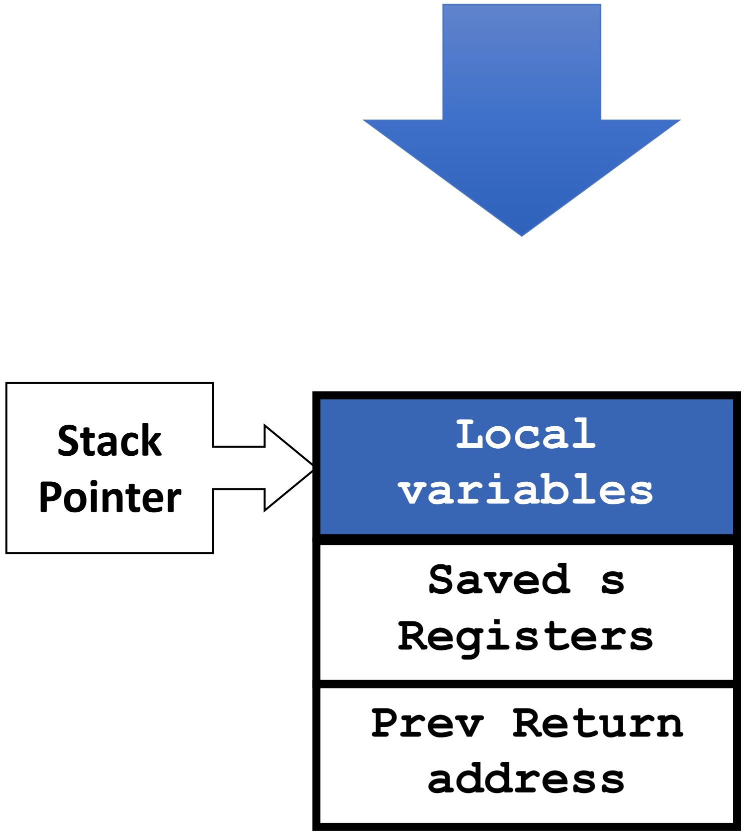
Let's look at the stack in detail

Let's Start at Step 7 To See What Our Function Does



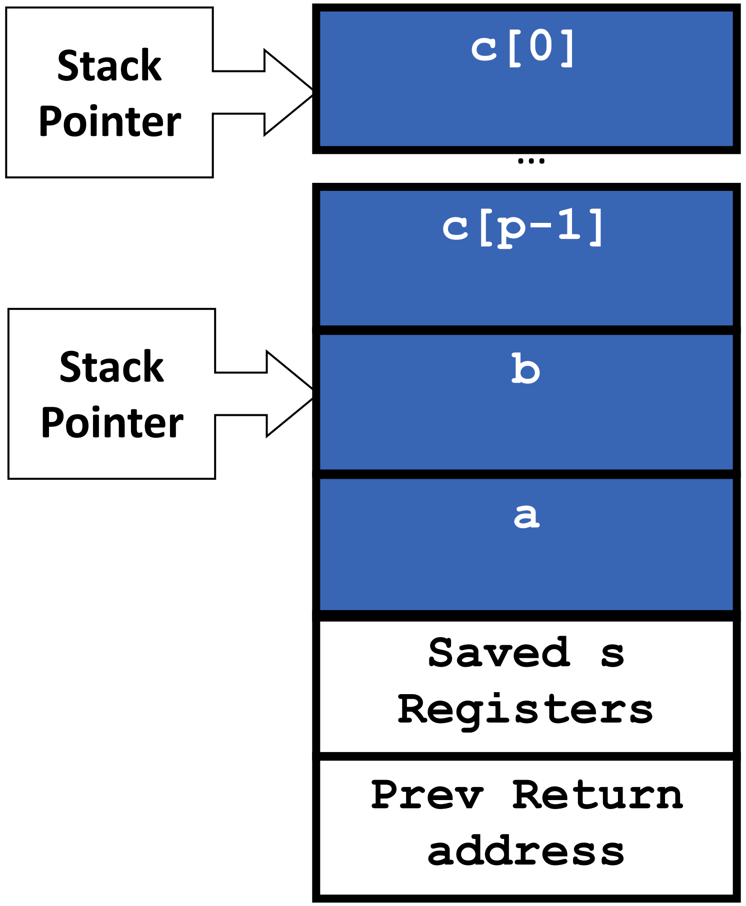

```
int a = 1, b = 3;
if (a != p) {
        int c[p];
        c[p-1] = b + a;
        …
        }
b++; a++;
```

Slide The Stack Diagram Down




```
int a = 1, b = 3;
if (a != p) {
        int c[p];
        c[p-1] = b + a;
        …
        }
b++; a++;
```

When Our Function Runs




```
➔ int a = 1, b = 3;
  if (a != p) {
          int c[p];
➔         c[p-1] = b + a;
          …
          }
➔ b++; a++;
```

This ”a” is at SP + 1

This "b" is at SP + 0

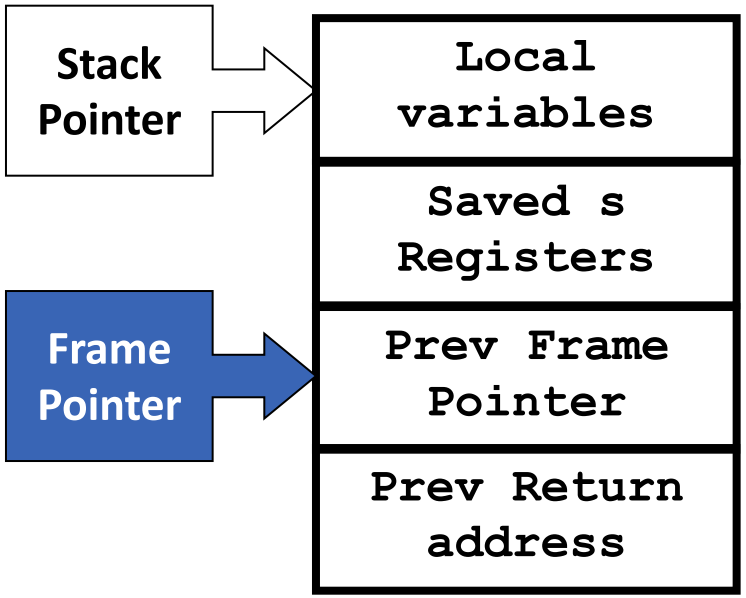
This “a” is at SP + p + 1

This "b" is at SP + p

This "b" is at SP + 0

This ”a" is at SP + 1

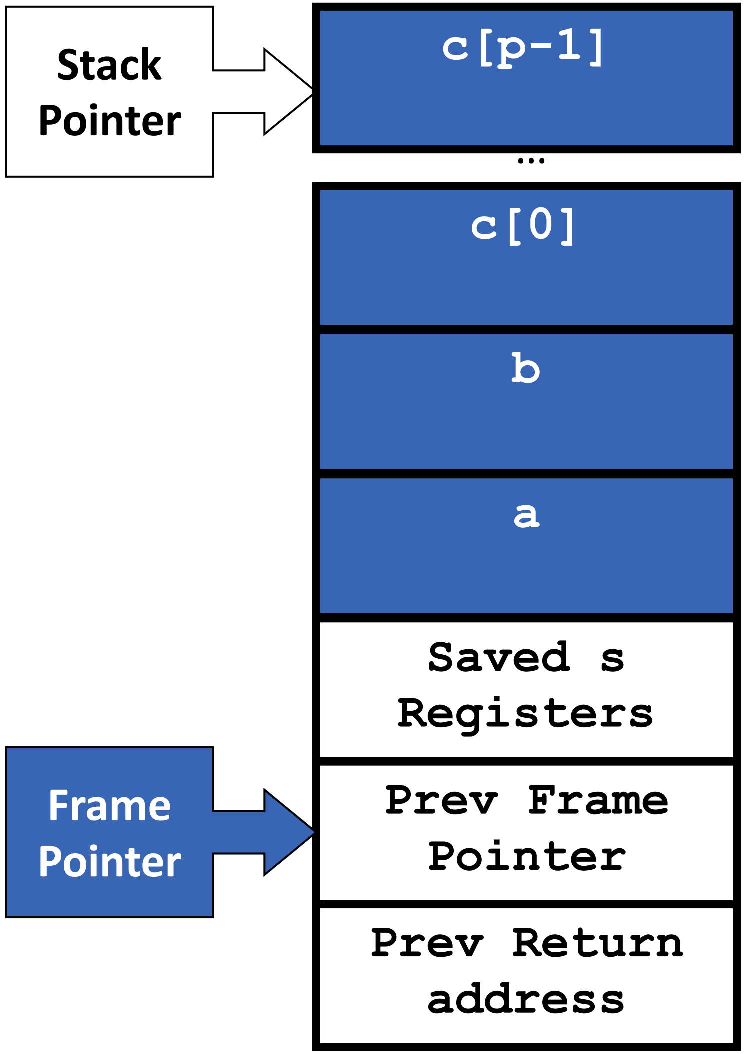
Let's Revise foo()'s Stack Frame



We're going to add one more item to the stack: Prev Frame Pointer because we'll need to save/restore our Frame Pointer register.

And that's where we'll point our Frame Pointer register.

Addressing Local Variables with FP



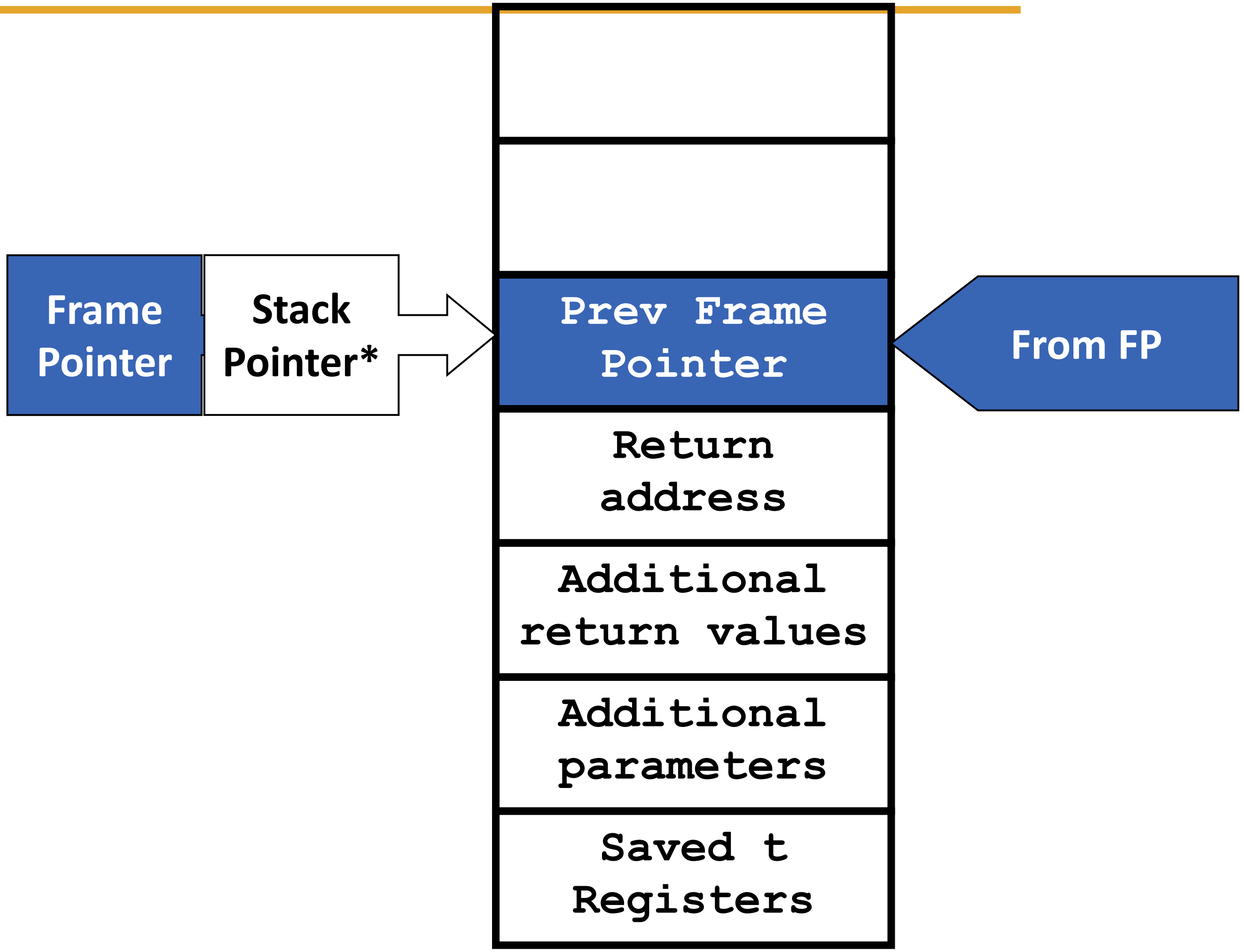
This "b" is at FP - 3

```
int a = 1, b = 3;
if (a != p) {
        int c[p];
        c[0] = b + a;
        …
        }
b++; a++;
```

This "b" is at FP - 3

This "b" is at FP - 3

STACK

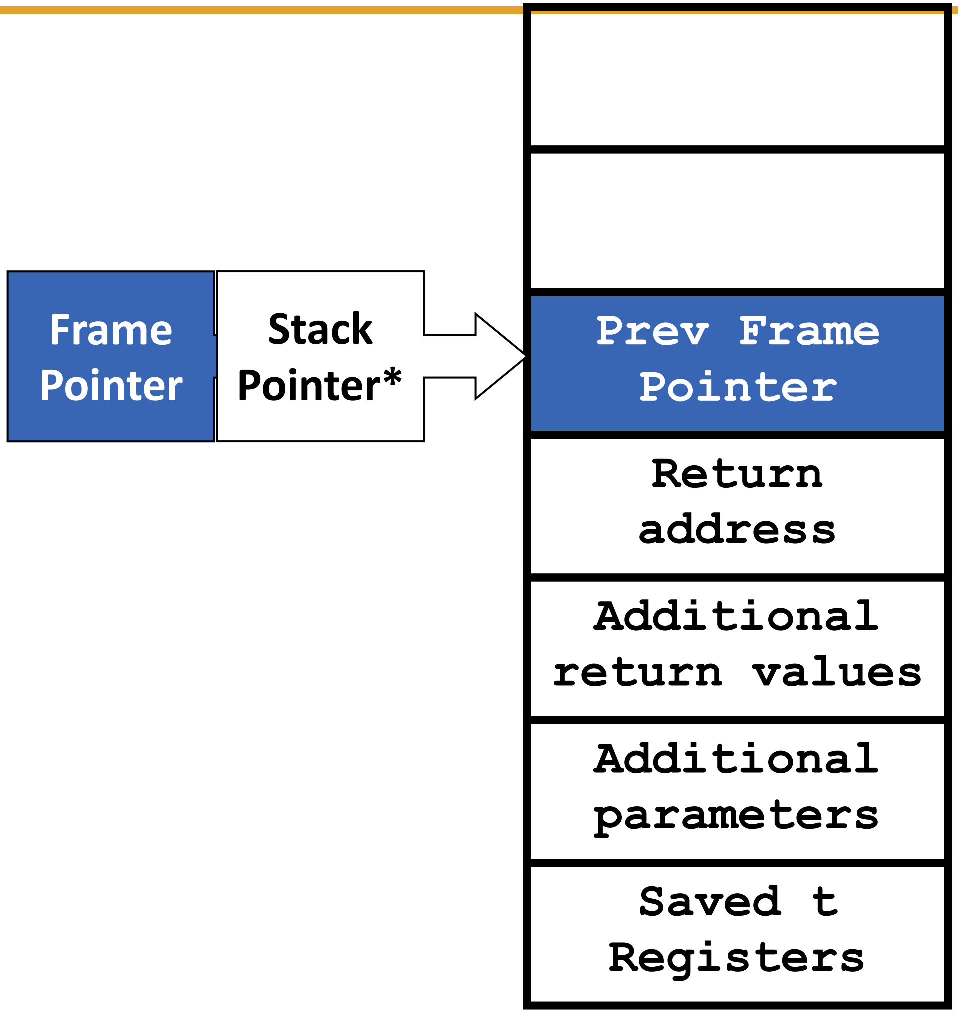


New Step 5.5.

Callee stores previous frame pointer then copies contents of stack pointer into frame pointer.

***Stack pointer may change during procedure execution**

STACK

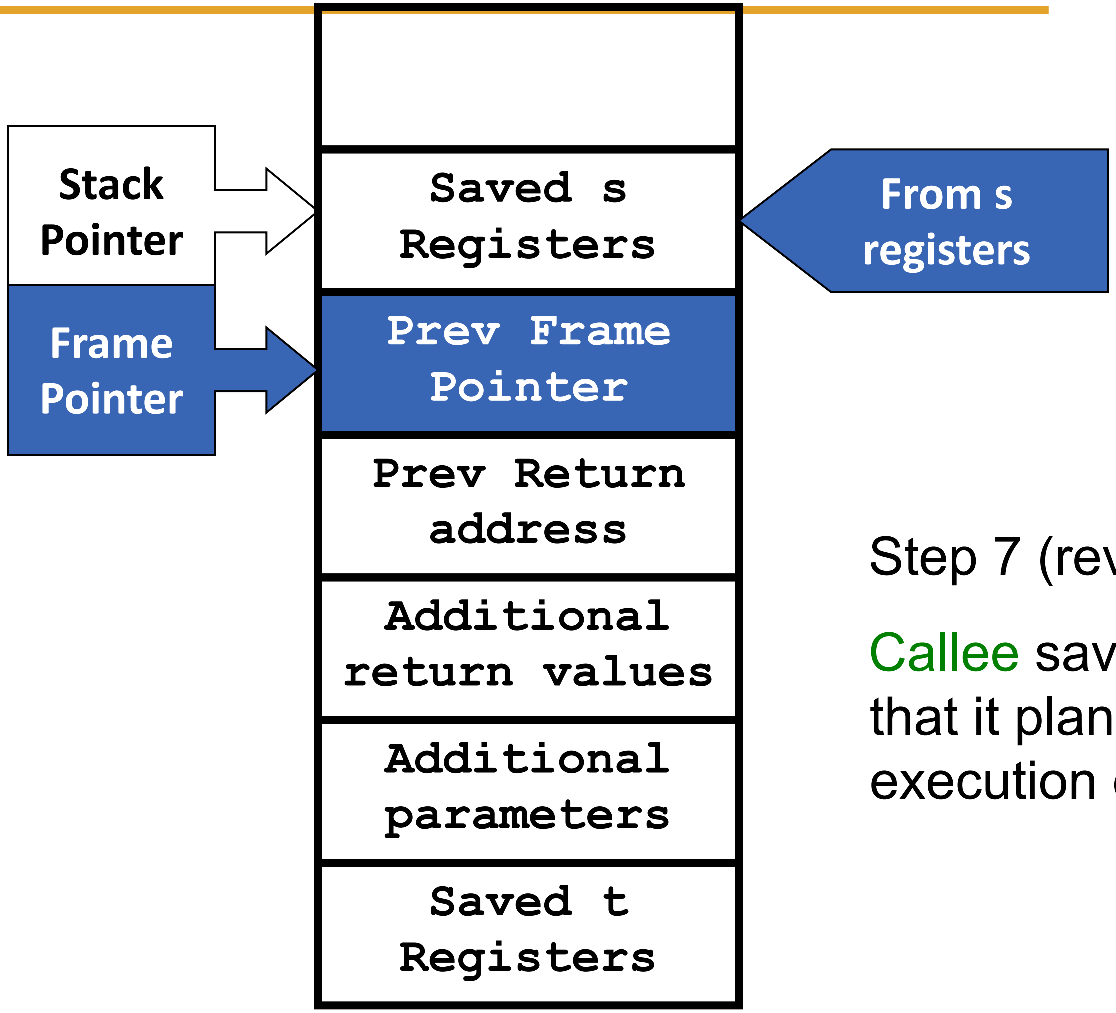


Step 6 (revised).

Callee saves Frame Pointer on the stack and sets Frame Pointer to the Stack Pointer

***Stack pointer may change during procedure execution**

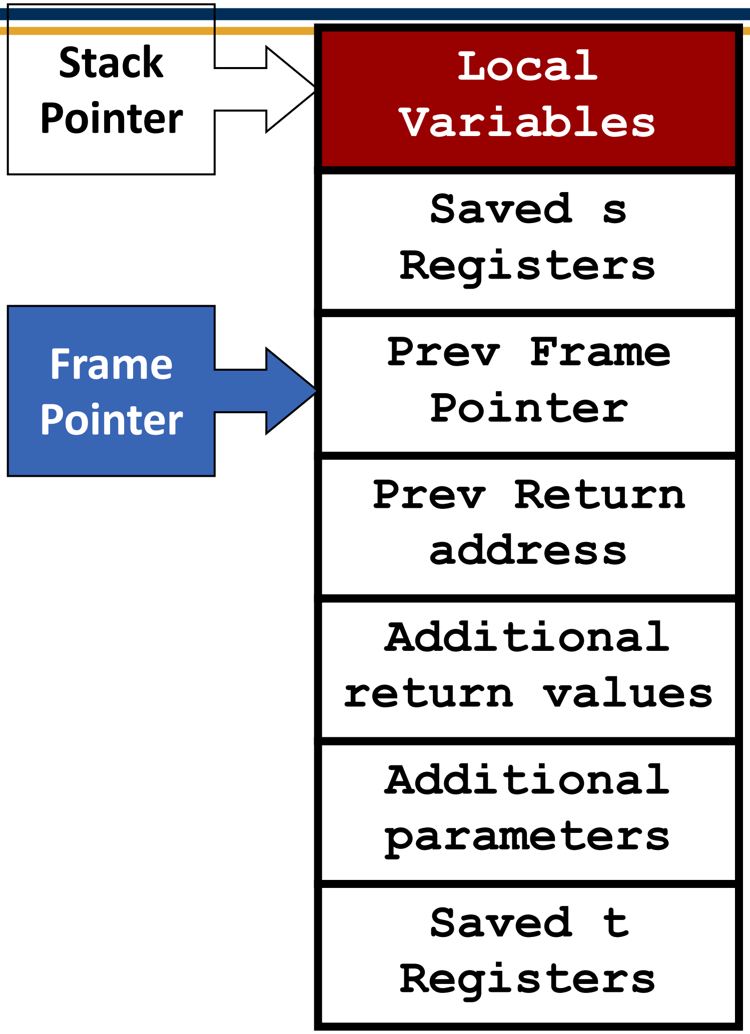
STACK



Step 7 (revised).

Callee saves any of registers s0-s2 that it plans to use during its execution on the stack.

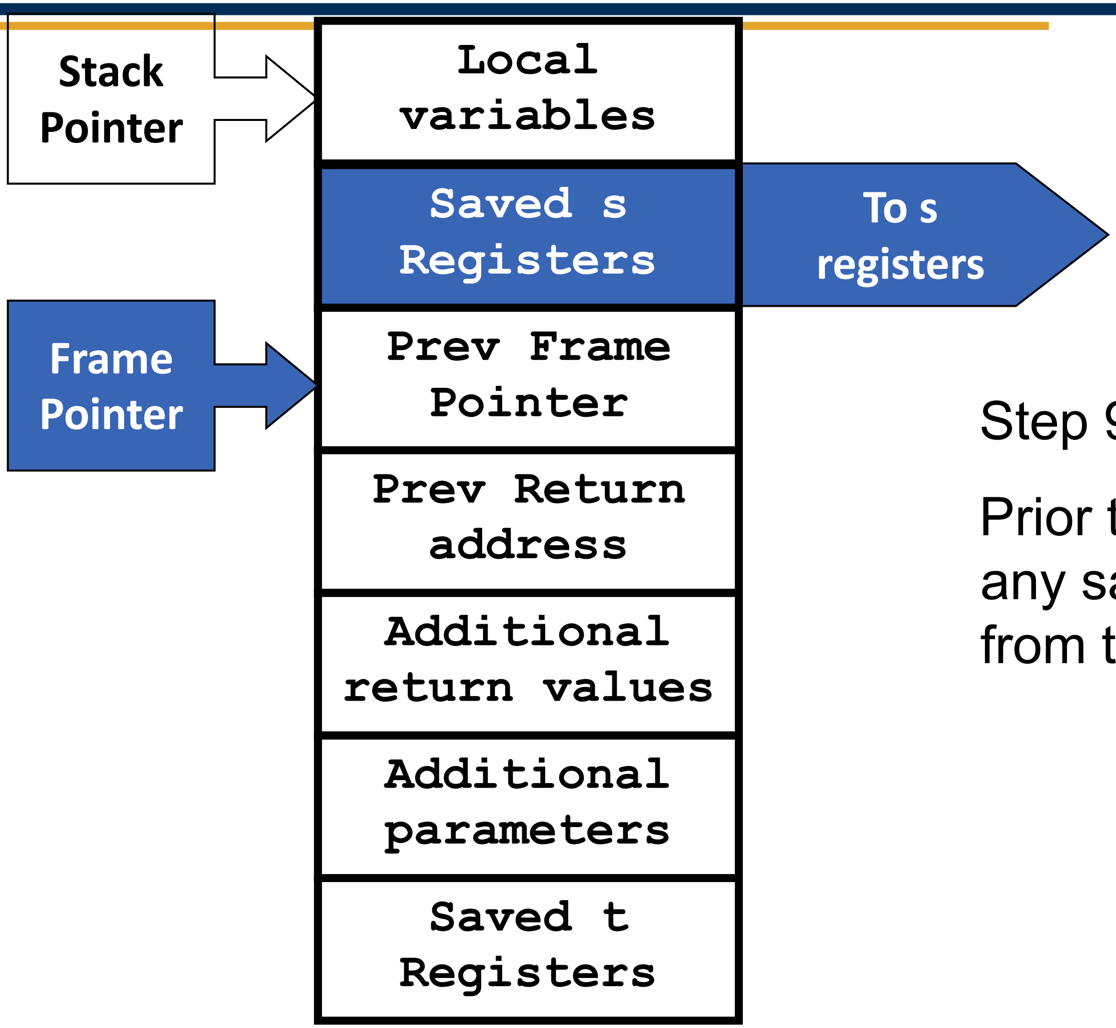
STACK



Step 8 (revised).

Callee allocates space for any local variables on the stack

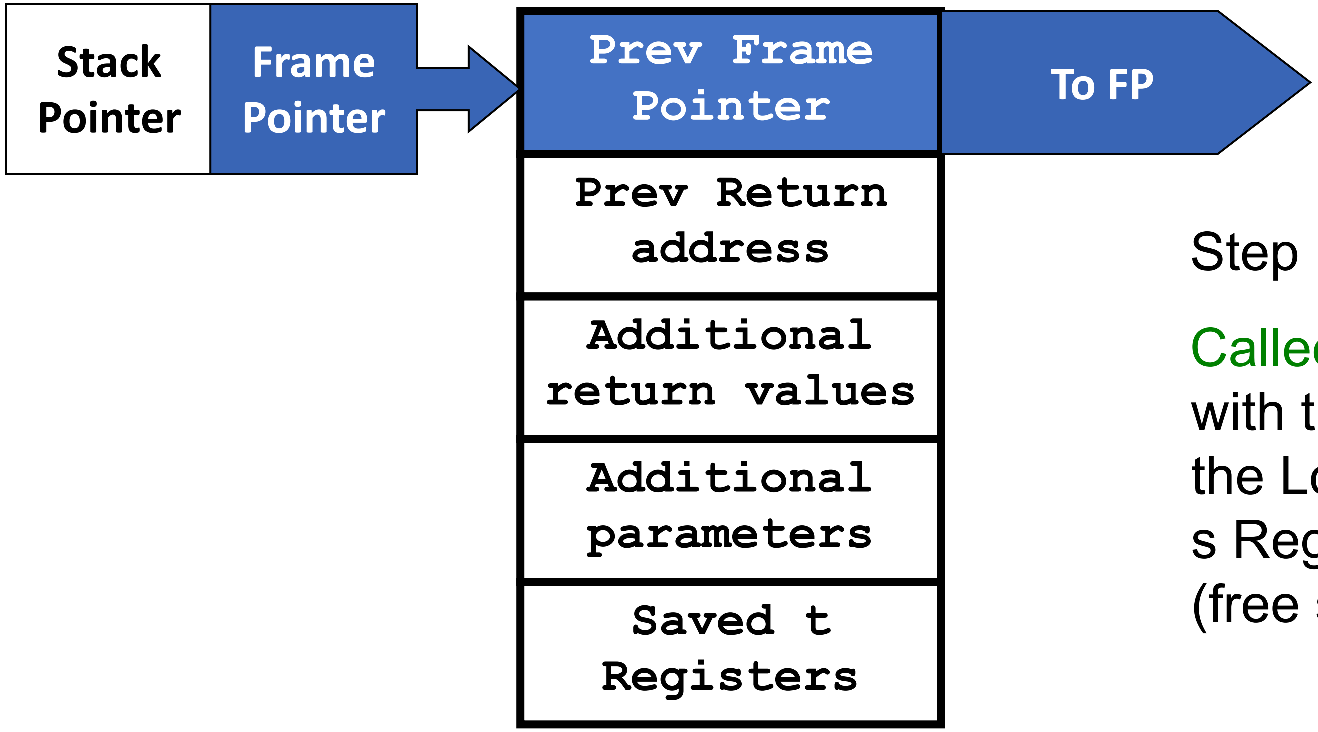
STACK



Step 9 (revised).

Prior to return, Callee restores any saved s0-s2 registers from the stack

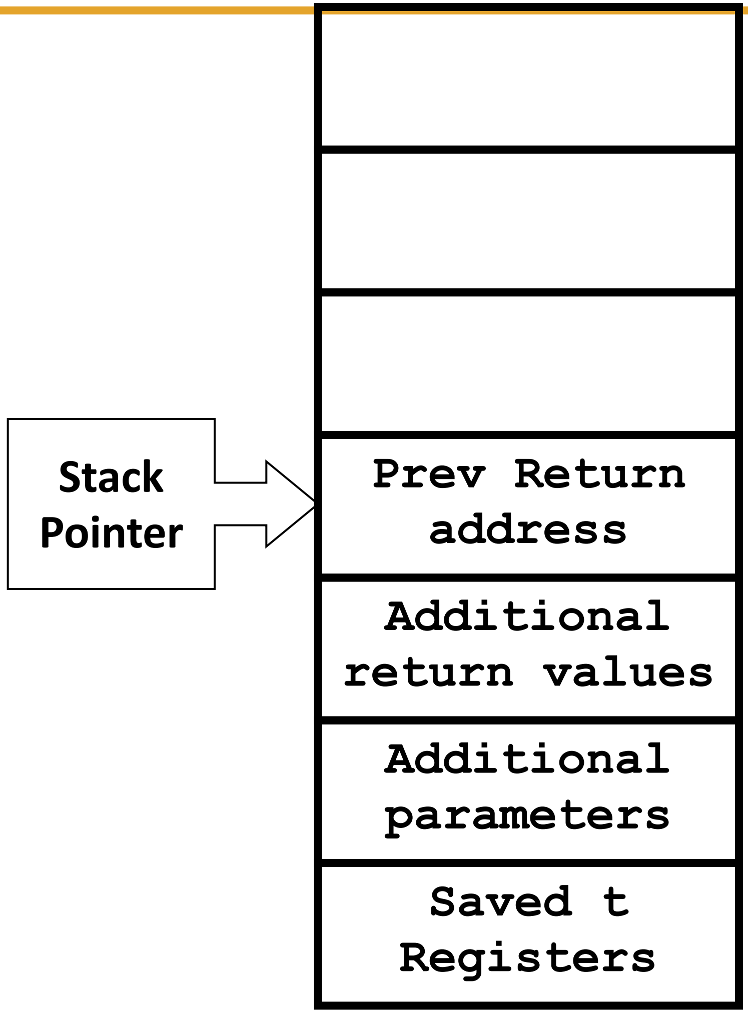
STACK



Step 10 (revised).

Callee replaces the value in SP with the value from FP to pop the Local Variables and Saved s Registers off the stack (free stack space)

STACK



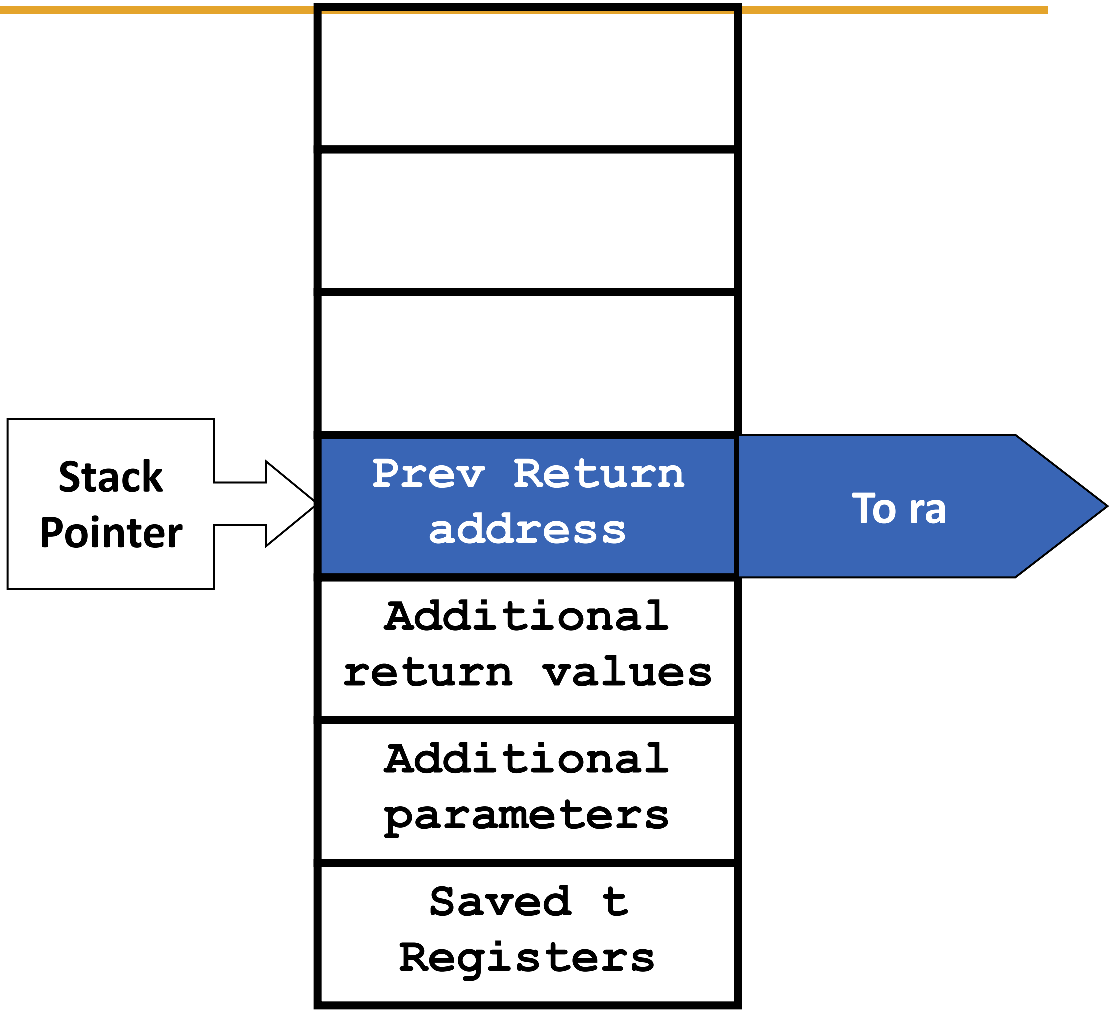
Step 11 (revised).

Callee executes jump to ra

No change to stack.

Note that Frame Pointer is now pointing to caller's activation record and we proceed as we did without a frame pointer

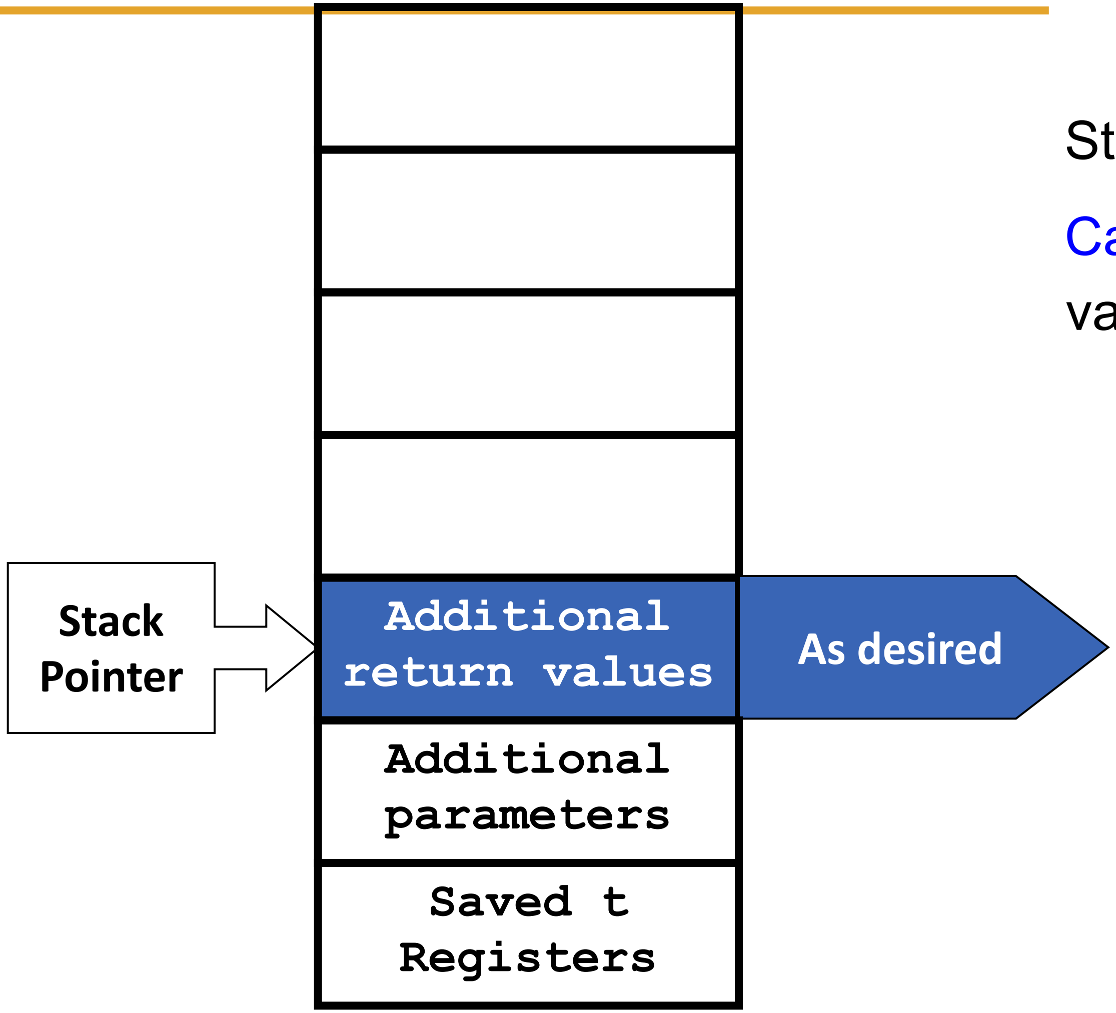
STACK



Step 12 (revised).

Upon return, Caller restores previous return address to ra

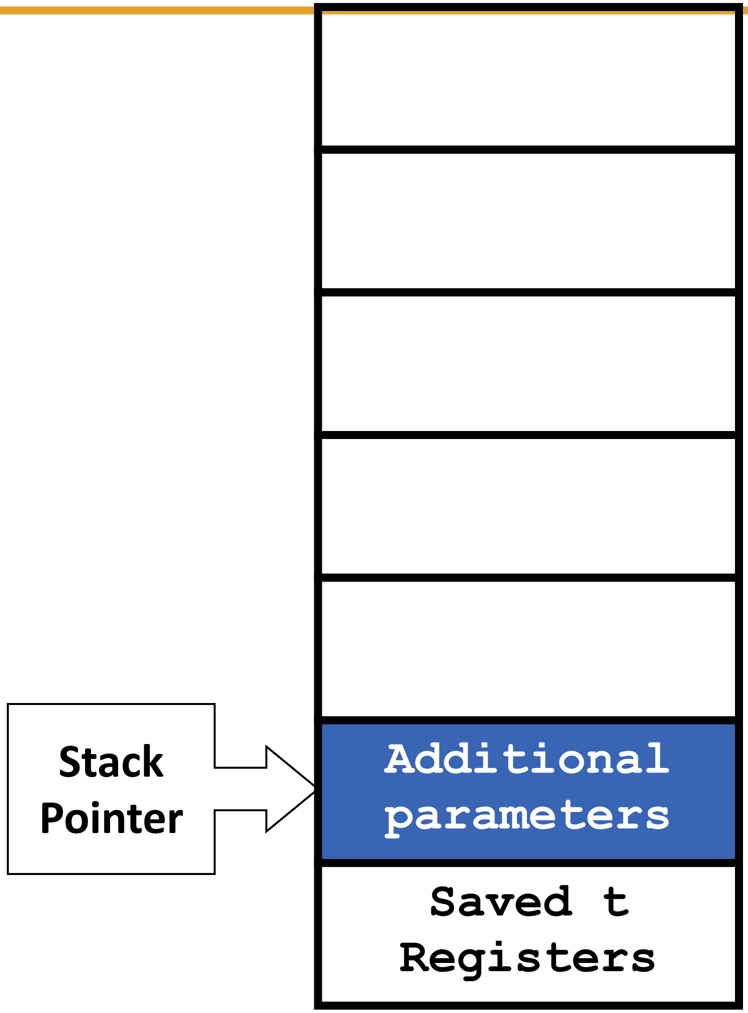
STACK



Step 13 (revised).

Caller stores additional return values as desired

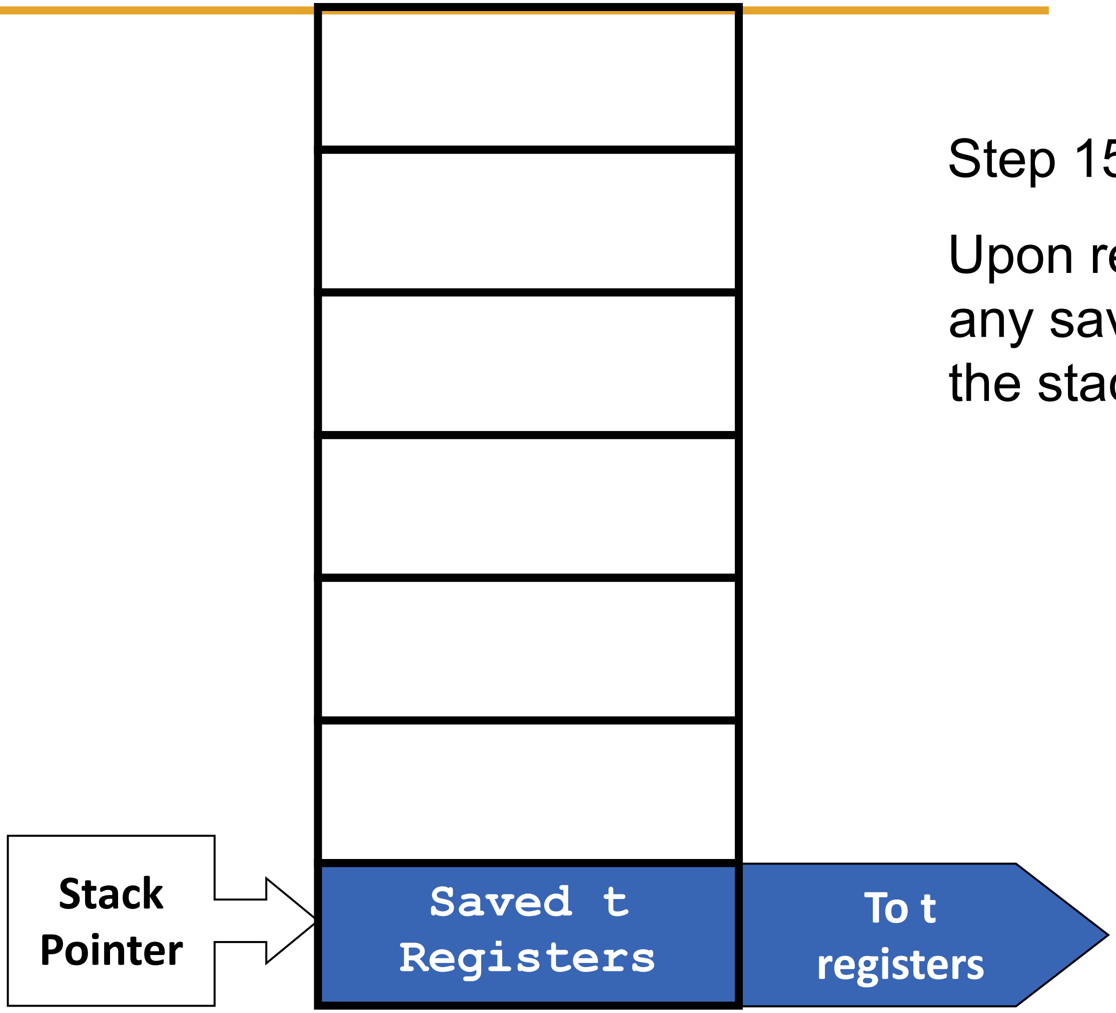
STACK



Step 14 (revised).

Upon return, Caller moves stack pointer to discard additional parameters

STACK



Step 15 (revised).

Upon return, Caller restores any saved t0-t2 registers from the stack

Effect of Stack Evolution

- The offset with respect to the stack pointer for referencing variables on the stack changes as the stack grows and shrinks

  ➔ A pain for the compiler writer
  ➔ Burdens the code with complicated local variable address calculations

- How to reduce this pain?

  ➔ Have a fixed harness on the stack for referencing local variables

    ➔ Frame Pointer (FP)

Session ID: dag2200

We keep track of a frame pointer because…

27% A. It's faster to access a variable through the frame pointer than it is to access through the stack pointer.

9% B. I can't explain why we waste one of our valuable register doing this.

55% C. We have to do it for legacy reasons.

9% D. It gives us a single, consistent, constant offset to reference the local variables in a stack frame.