Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2022

Lecture 4: Processors (final act)

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

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

Tuesday’s Recap

- Changing the control flow: conditional, switch statements, loops
  - PC, branch/jump instructions, PC-relative addressing mode, indirect addressing mode

- Procedure calls
  - JALR rt, at
  - Saving and restoring state in the stack
  - Stack pointer (sp), argument registers (a0-a2), return value register (v0)
  - Caller-callee convention: saved registers (s0-s2), temp registers (t0-t2)

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?

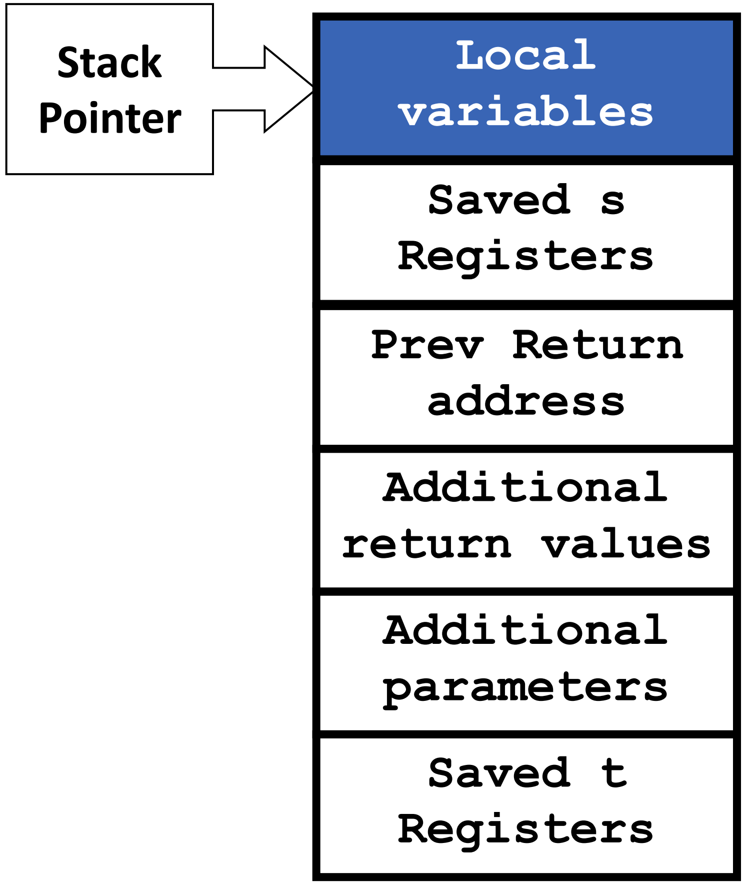
This code will cause us a problem:

```
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

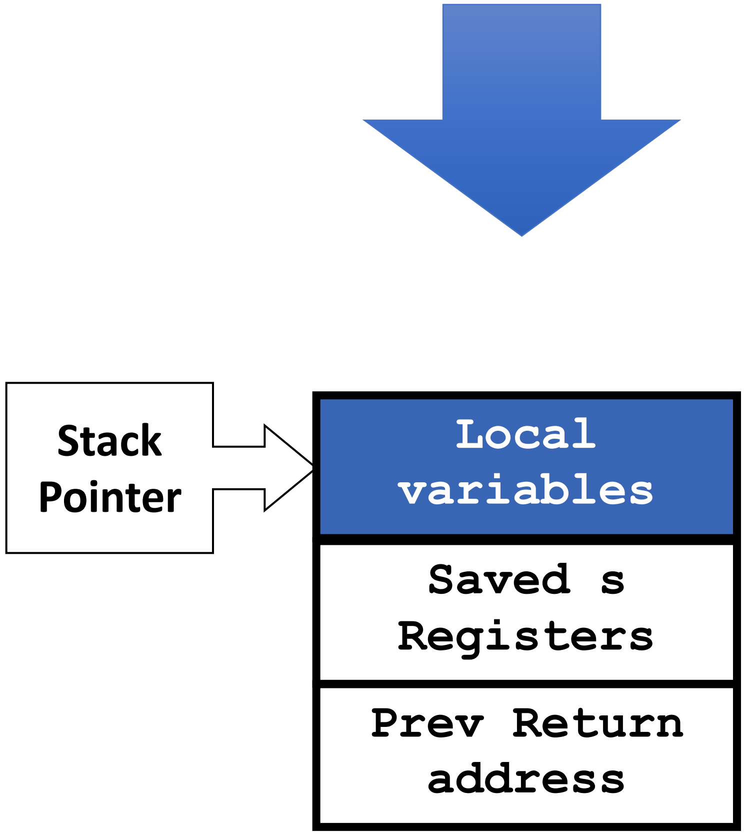
Let's look at the stack in detail

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



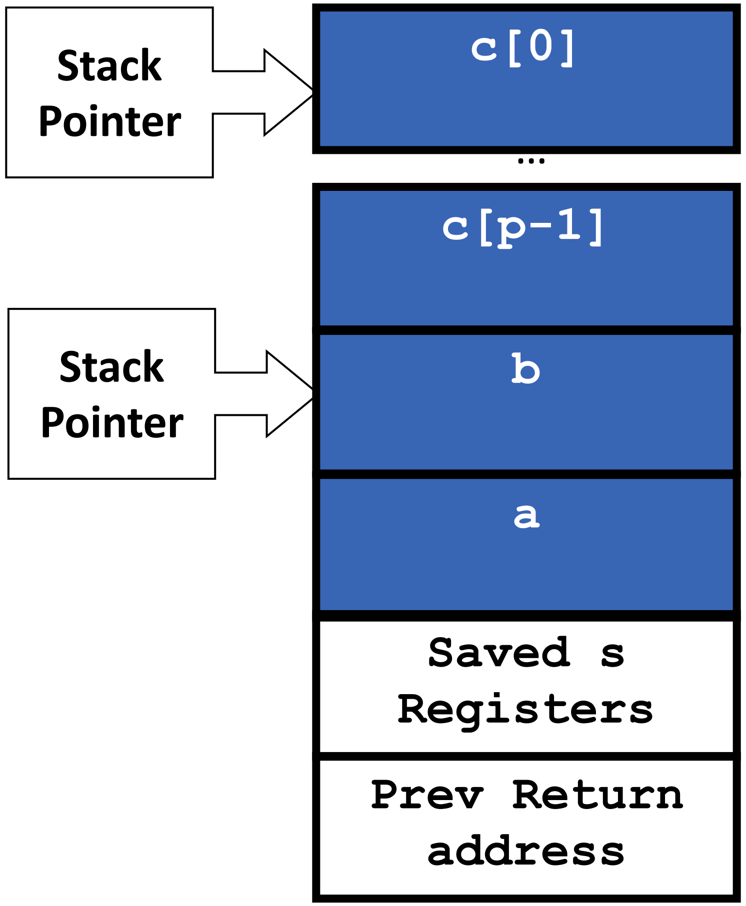

```
int a = 1, b = 3;
if (a != p) {
        int c[p];
        c[p-1] = b + a;
        …
        }
b++; a++;
```

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

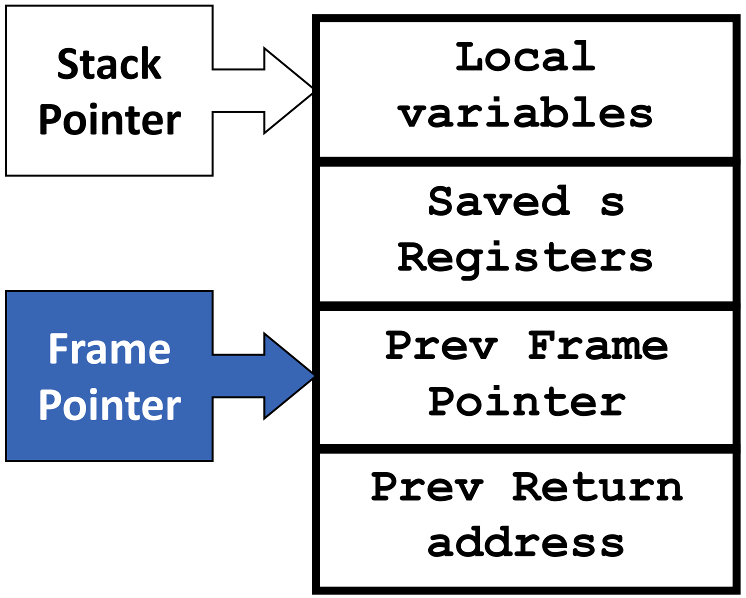
This “a” is at SP + p + 1

This "b" is at SP + p

This "b" is at SP + 0

This ”a" is at SP + 1

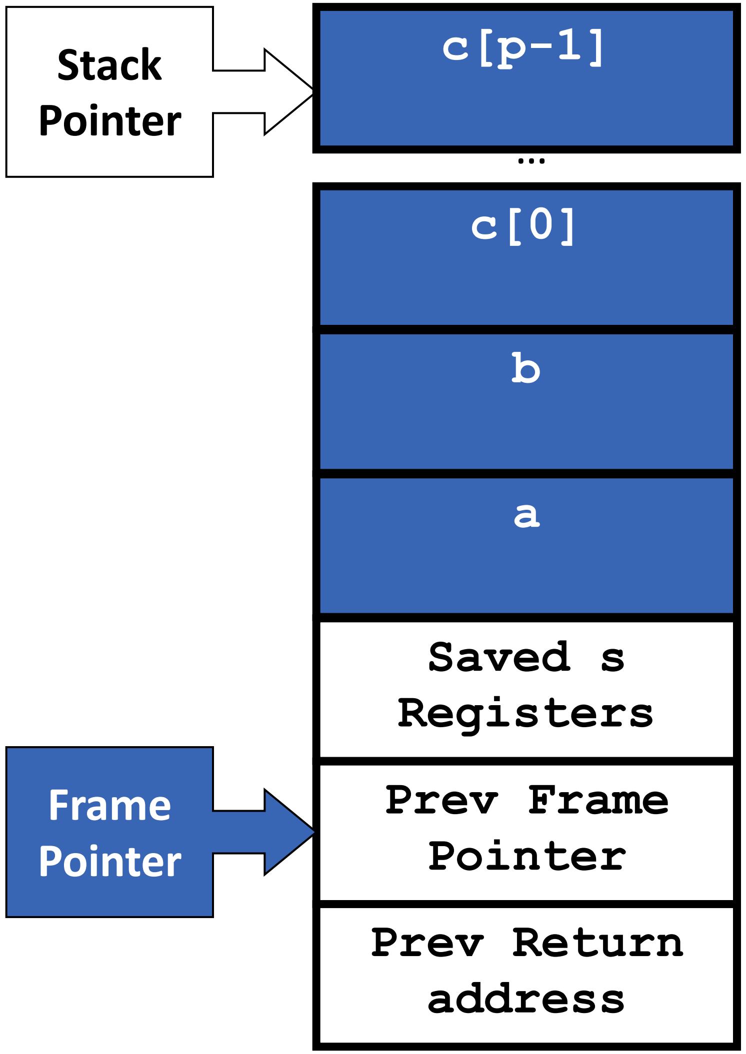
Let's Revise foo()'s Stack Frame



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



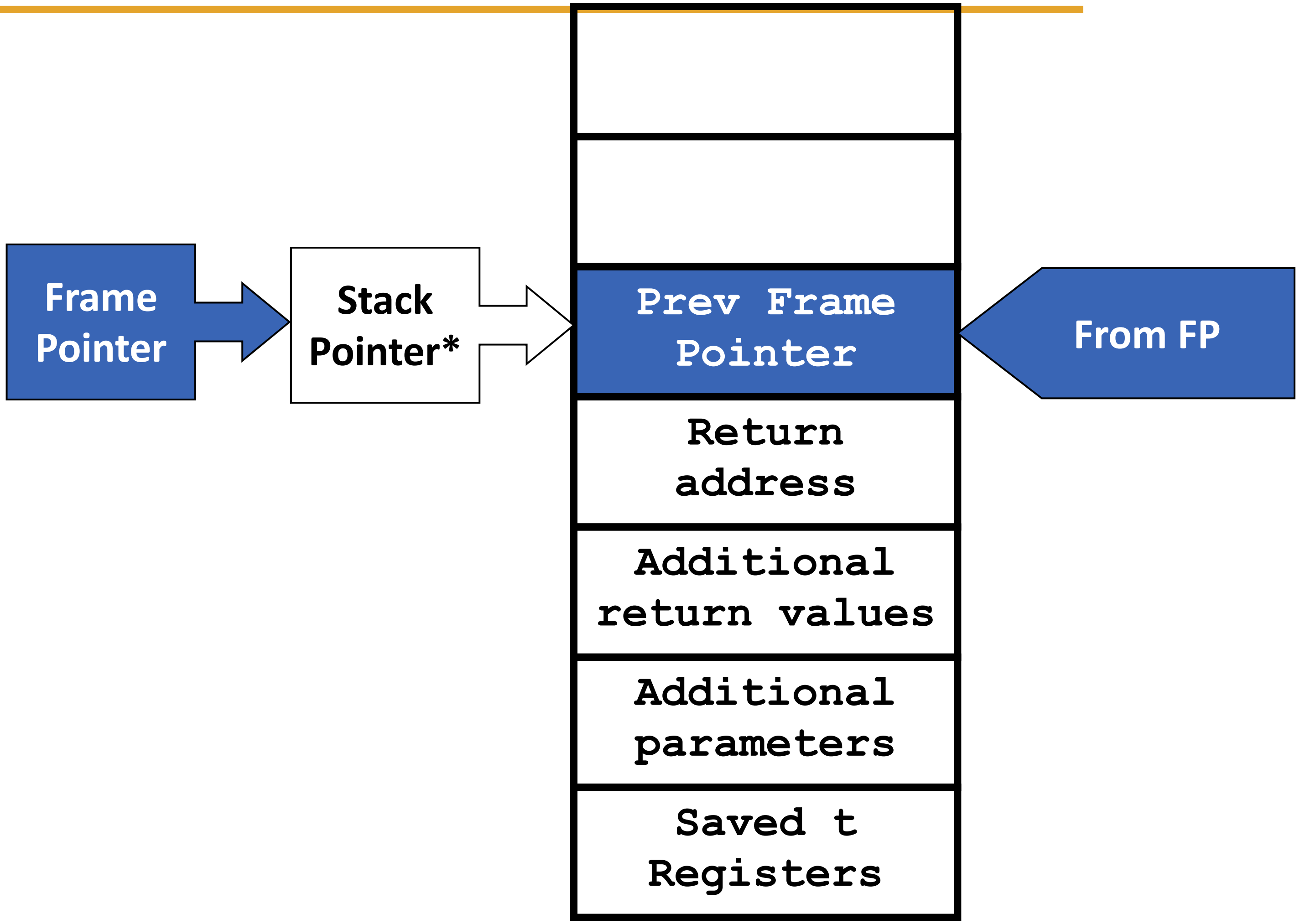
This "b" is at FP - 3

```
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

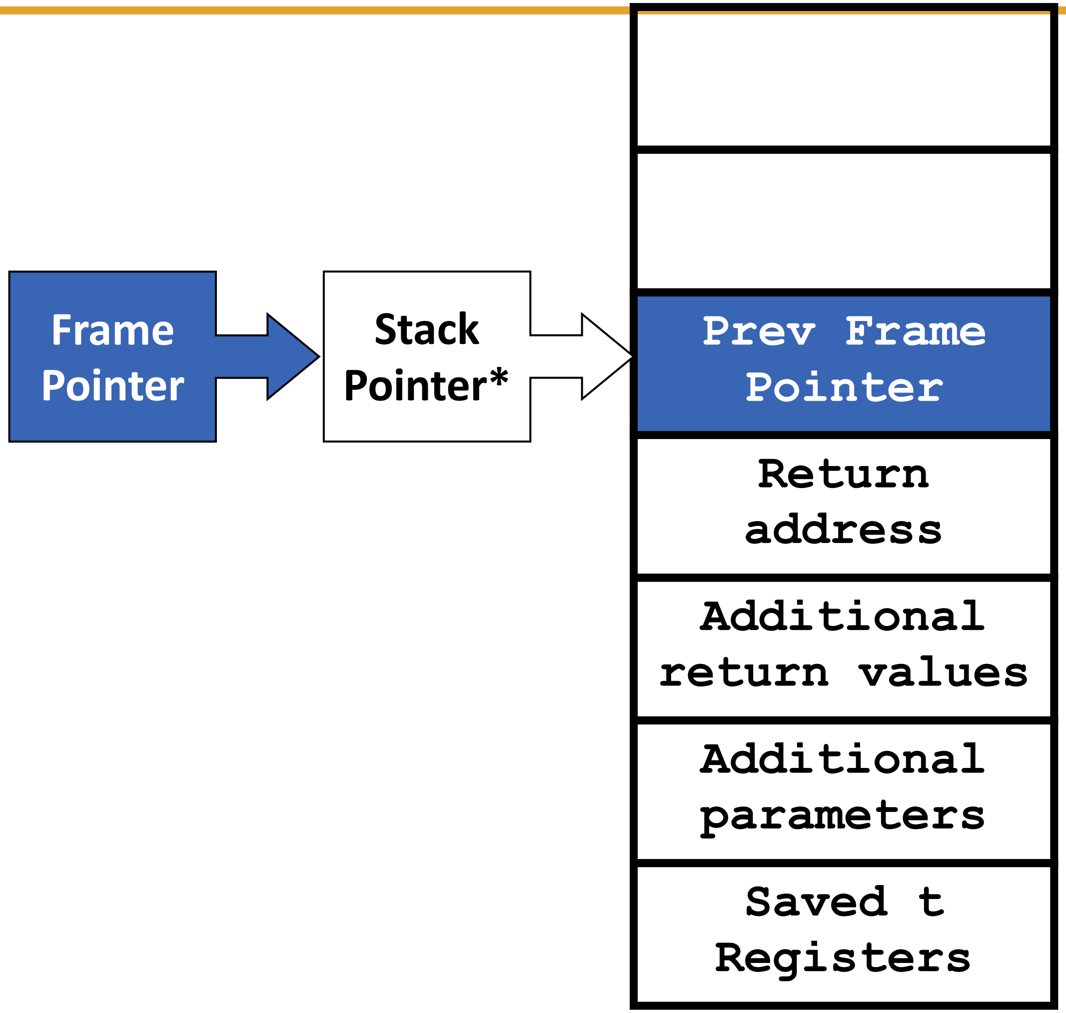


New Step 5.5.

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

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

STACK

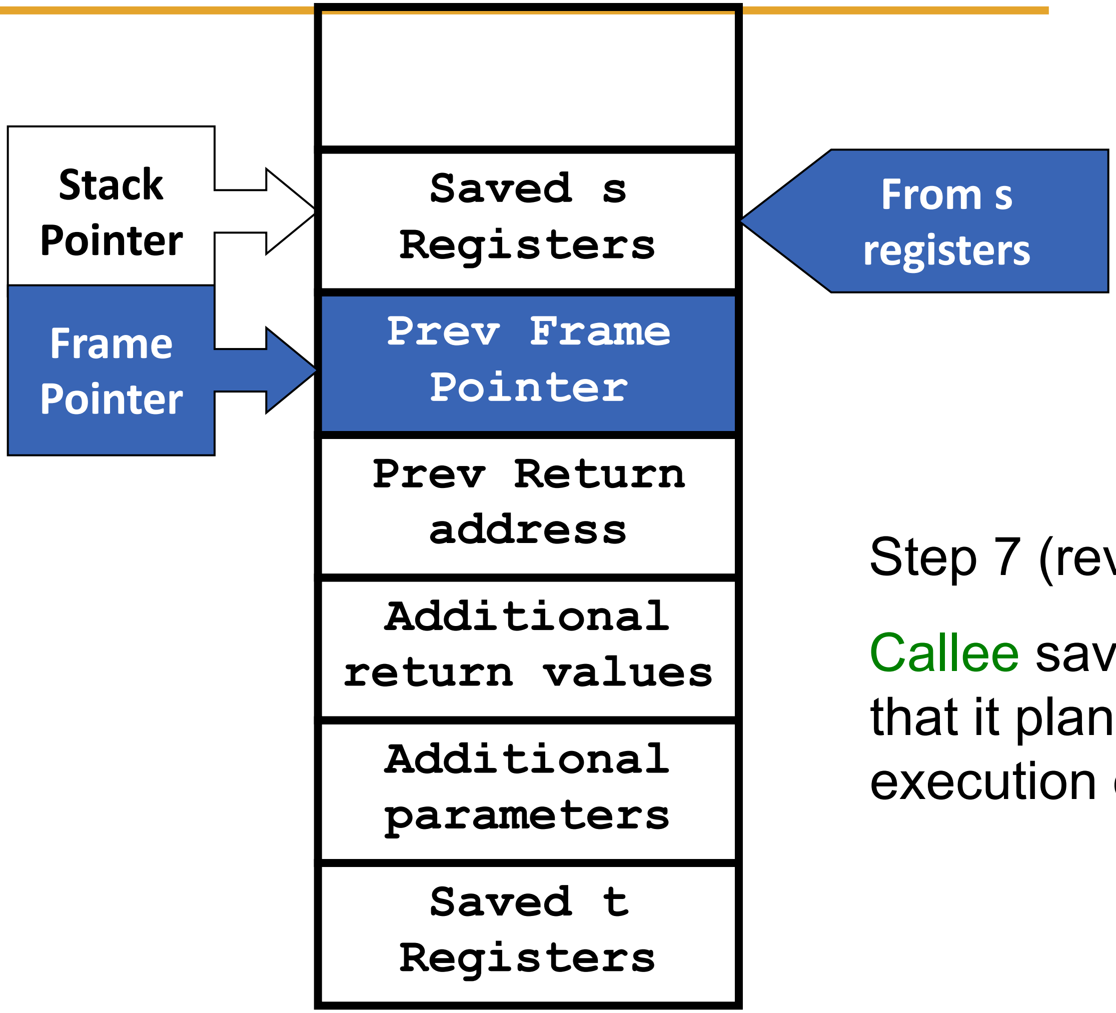


Step 6 (revised).

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

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

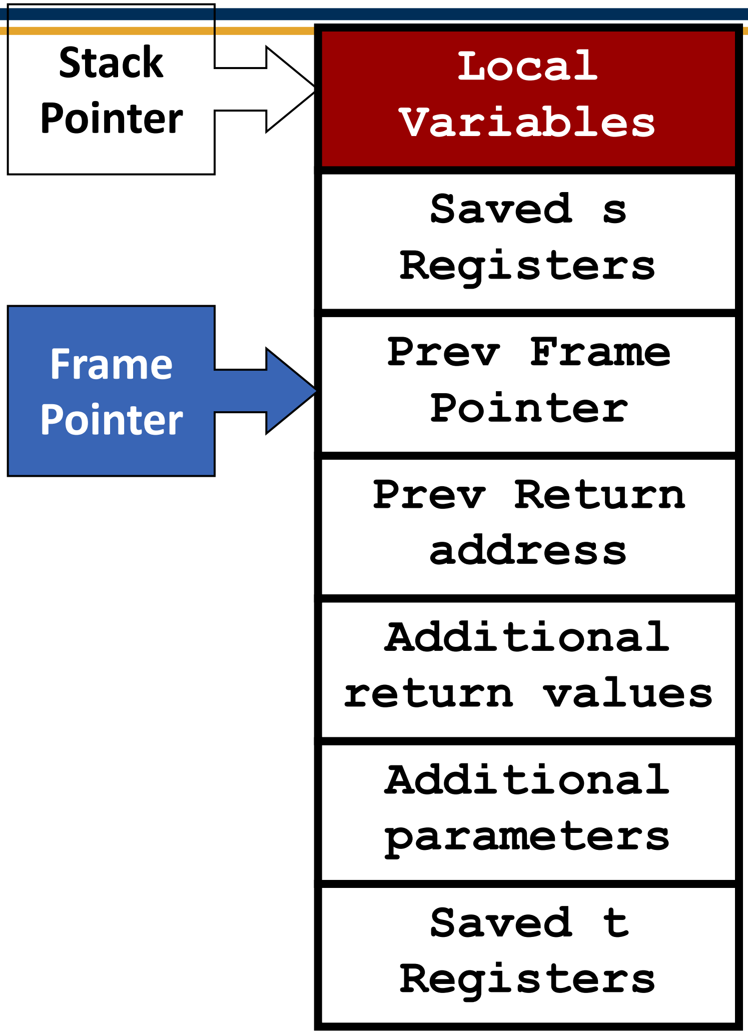
STACK



Step 7 (revised).

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

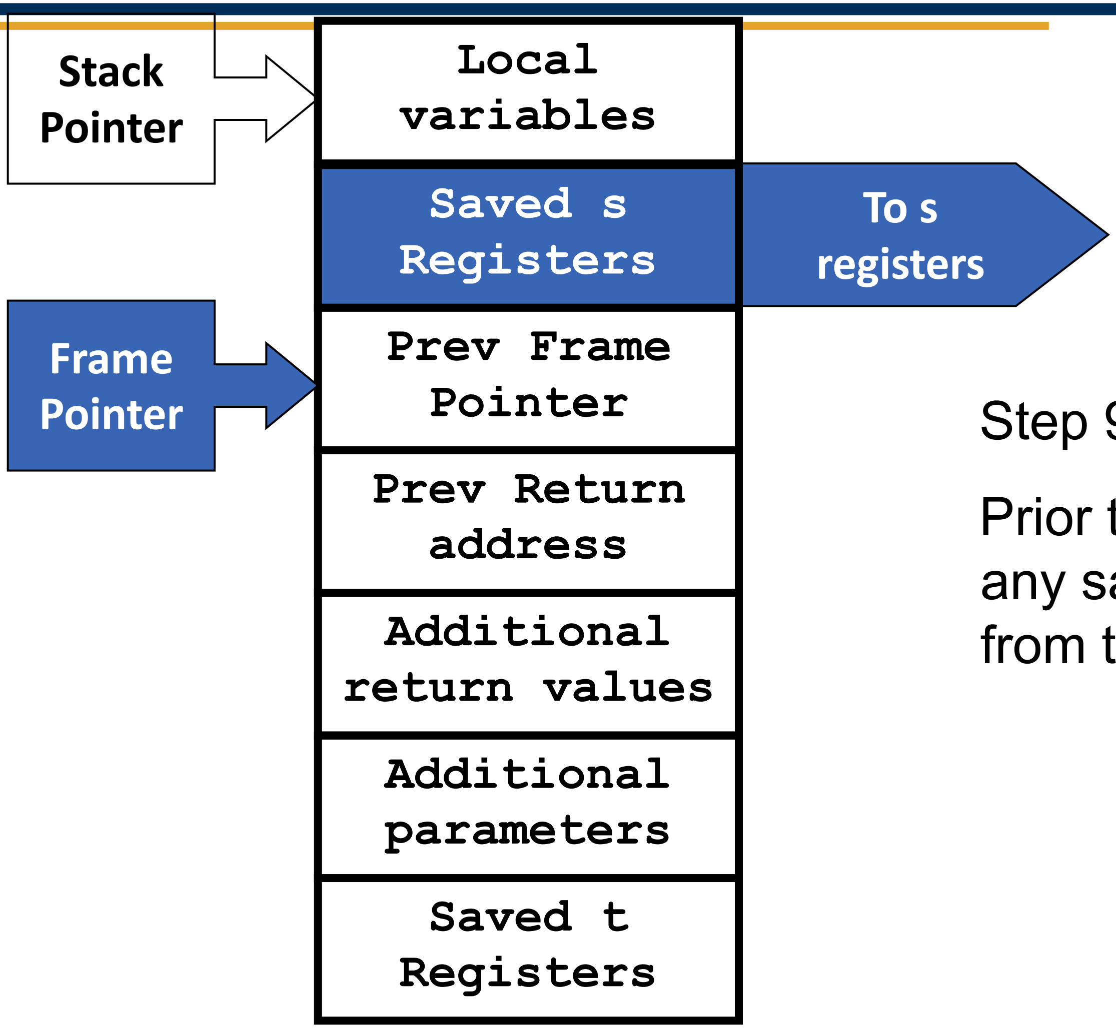
STACK



Step 8 (revised).

Callee allocates space for any local variables on the stack

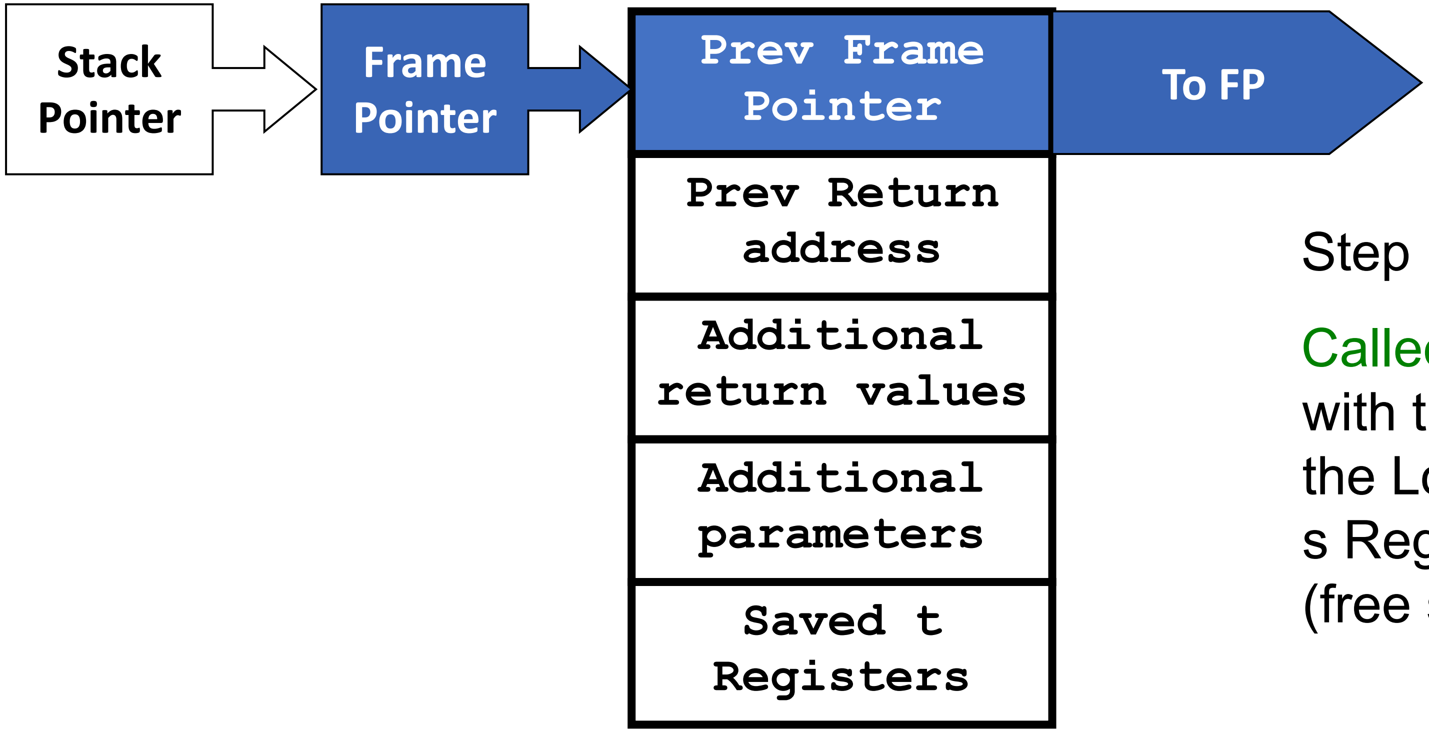
STACK



Step 9 (revised).

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

STACK

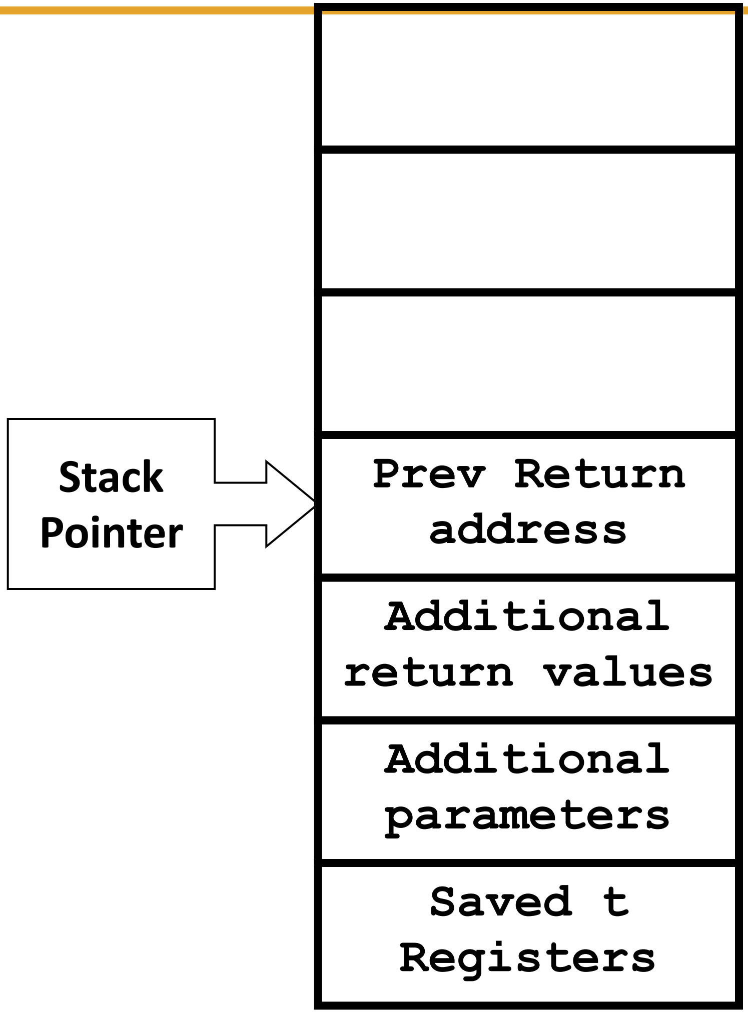


Step 10 (revised).

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)

Restore previous value of FP

STACK



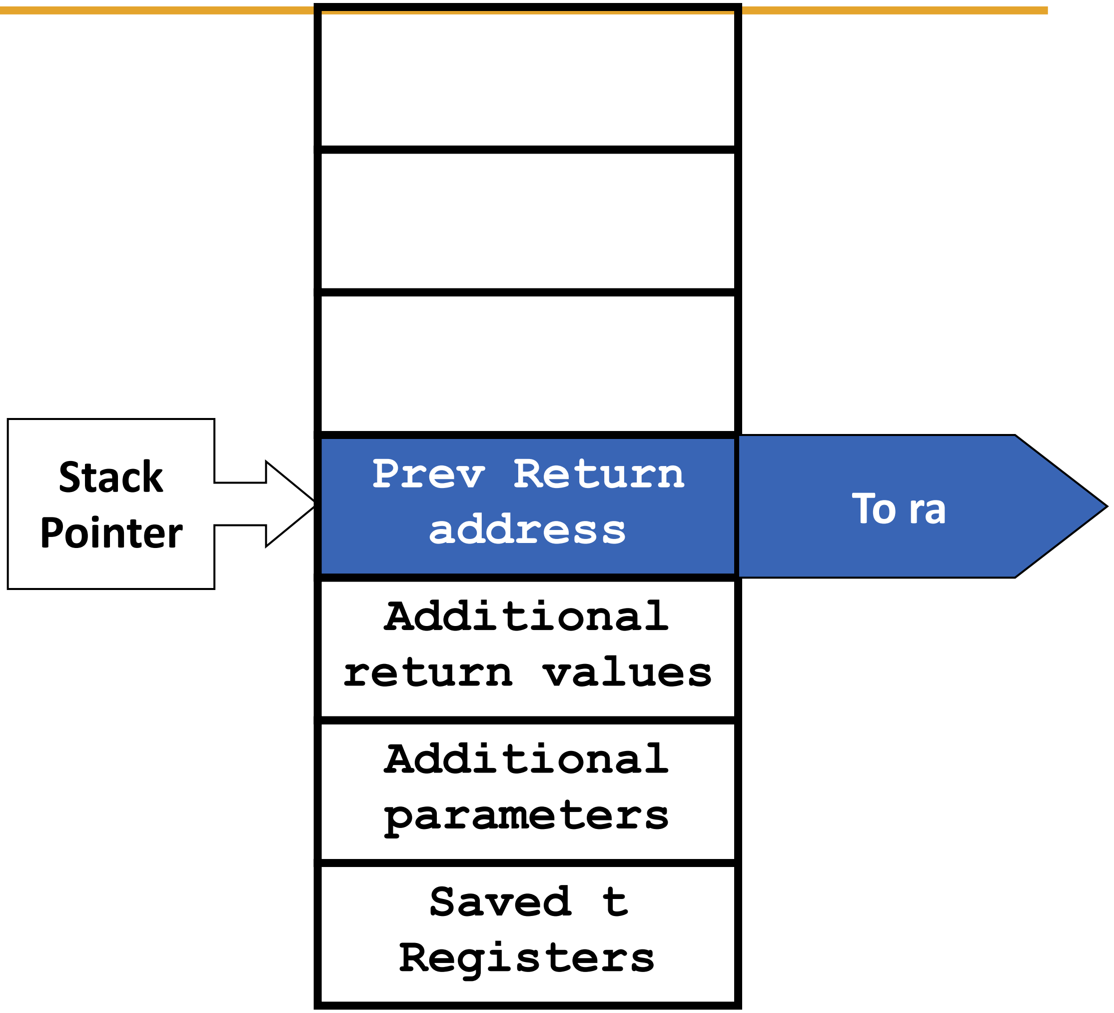
Step 11 (revised).

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 before introducing a frame pointer

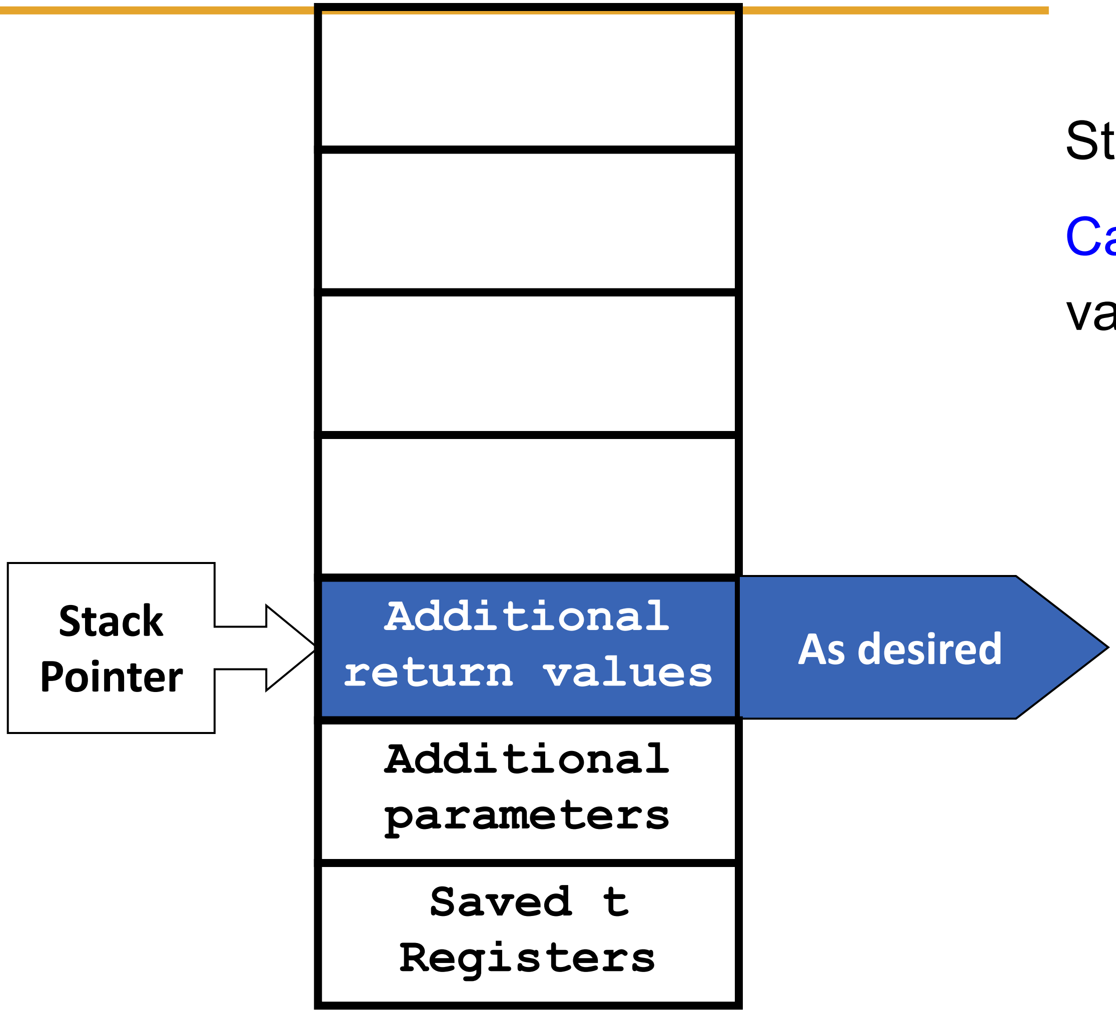
STACK



Step 12 (revised).

Upon return, Caller restores previous return address to ra

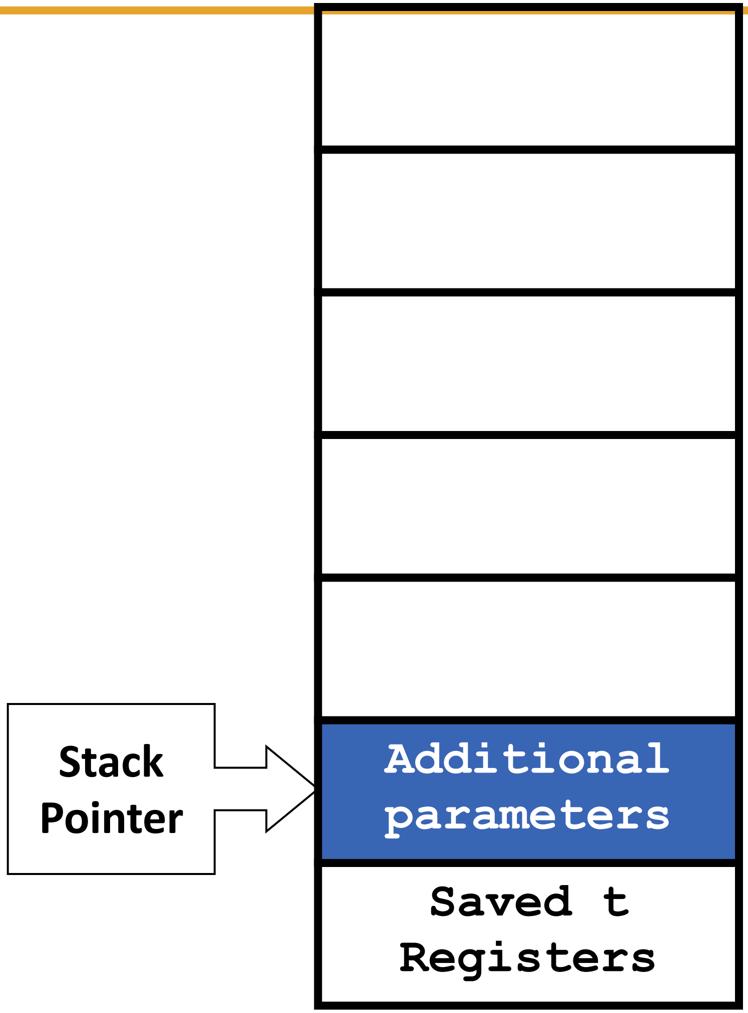
STACK



Step 13 (revised).

Caller stores additional return values as desired

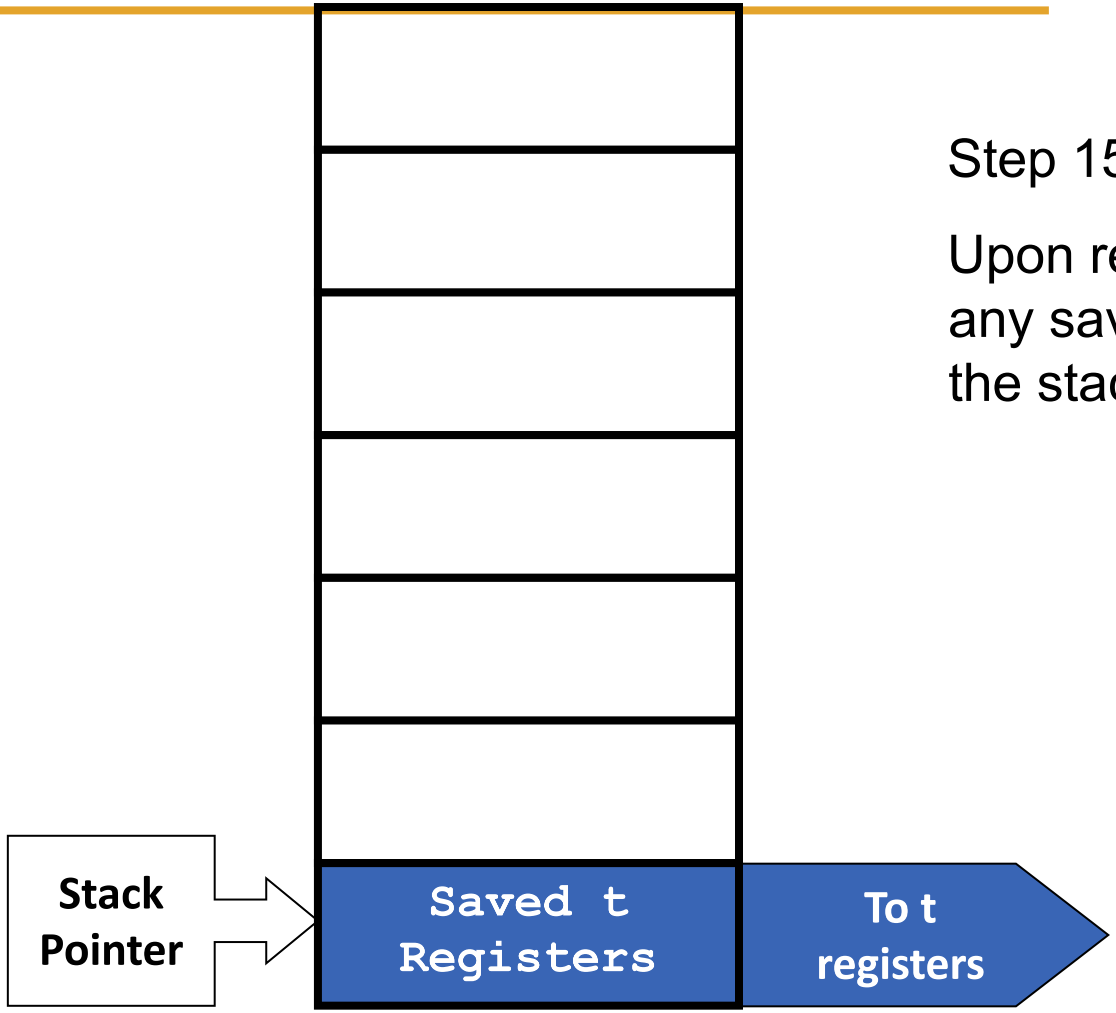
STACK



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.

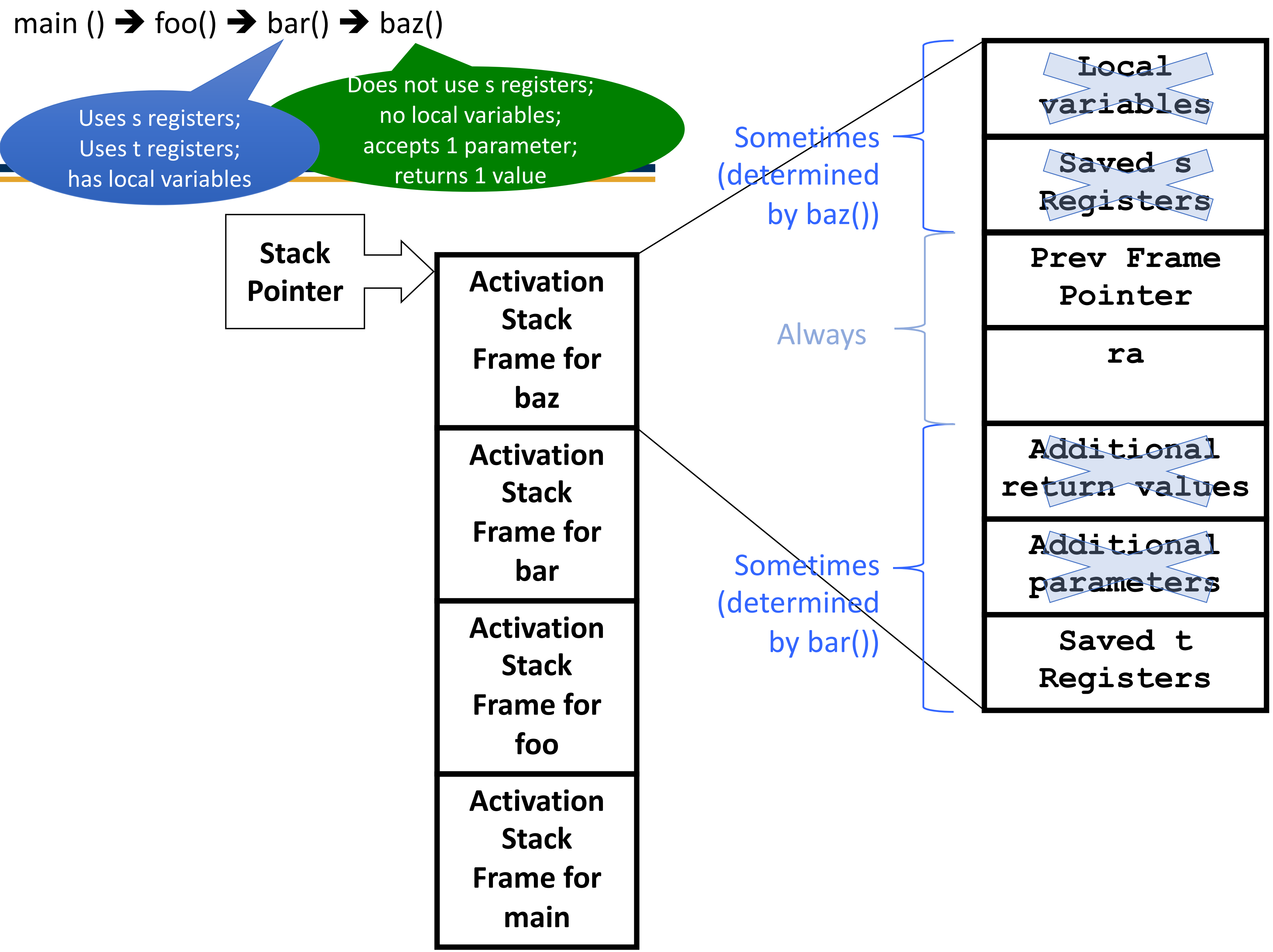
55% C. We have to do it for legacy reasons.

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

Example Stack Frames

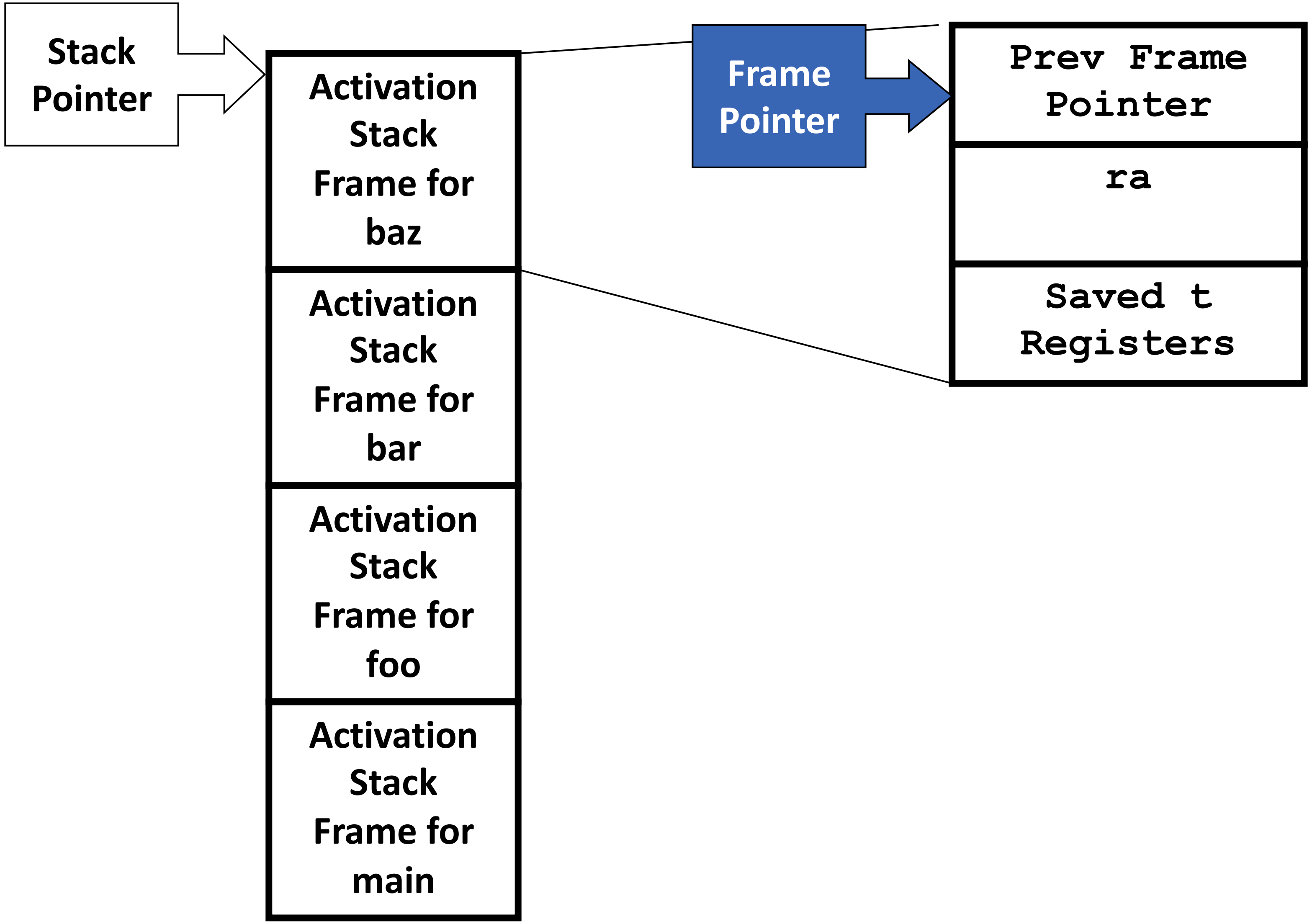
main () → foo() → bar() → baz()

- foo(): Does not use s registers; Uses t registers
- bar(): Uses s registers; Uses t registers; has local variables
- baz(): Does not use s registers; no local variables; accepts 1 parameter; returns 1 value

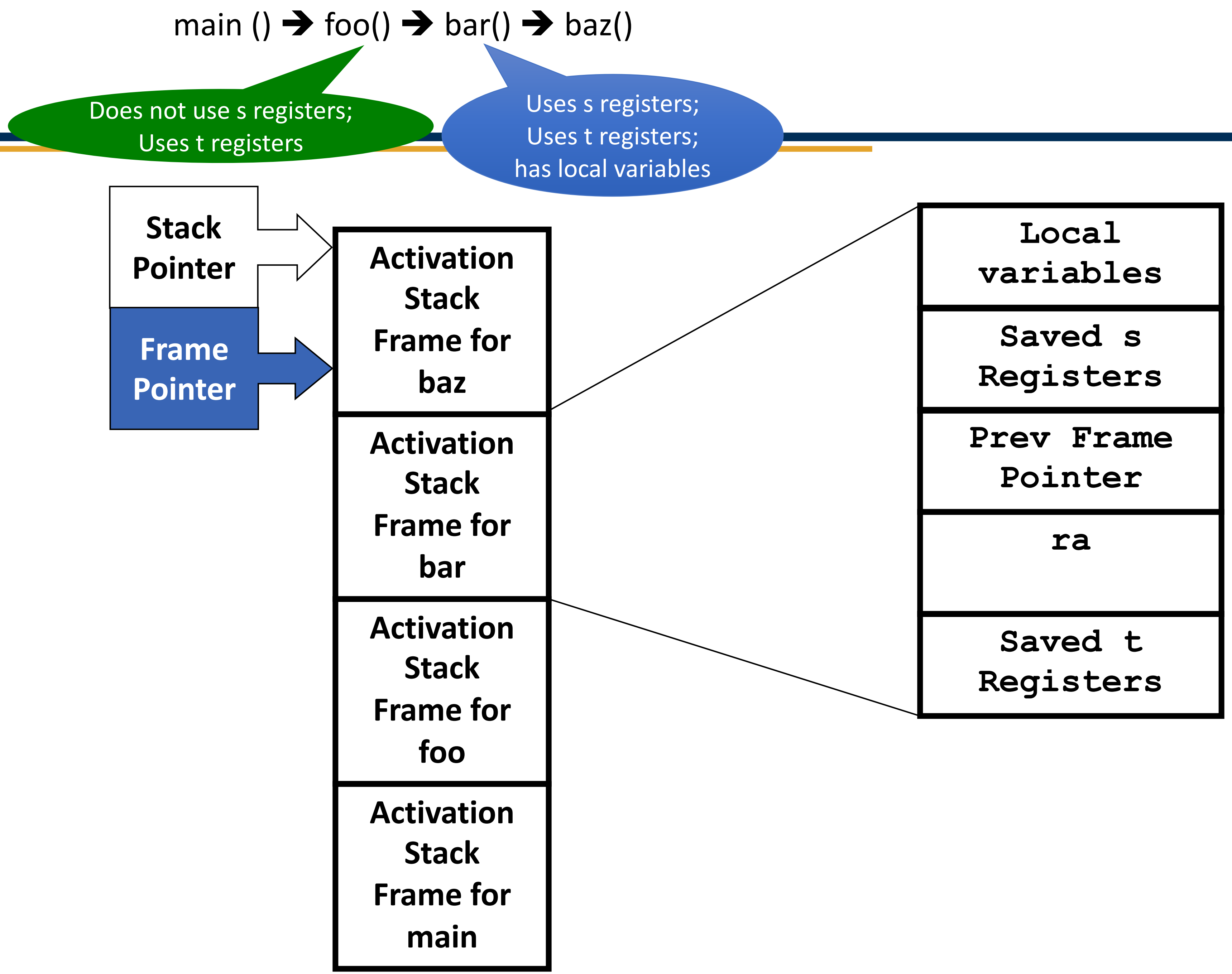

main () ➔ foo() ➔ bar() ➔ baz()
Uses s registers;
Uses t registers;
has local variables
Does not use s registers;
no local variables;
accepts 1 parameter;
returns 1 value
Stack Pointer
Activation Stack Frame for baz
Activation Stack Frame for bar
Activation Stack Frame for foo
Activation Stack Frame for main
Sometimes (determined by baz())
Always
Sometimes (determined by bar())
~~Local variables~~
~~Saved s Registers~~
Prev Frame Pointer
ra
~~Additional return values~~
~~Additional parameters~~
Saved t Registers

baz() Stack Frame

main () ➔ foo() ➔ bar() ➔ baz()



bar() Stack Frame



Session ID: dag2200

What do we do if there are no s registers to be saved during a procedure call?

25%	A.	Put a marker on the stack to indicate there are no saved registers.
25%	B.	Increment the stack pointer to leave room in case the s registers need to be saved later.
25%	C.	Push a word of zero for every s register not saved.
25%	D.	Nothing. Just don't use stack space.

Moving forward

- Making design choices
  - Additional instruction attributes
  - Addressing modes
  - Architecture styles
  - Instruction formats

- The LC-2200
  - Instructions
  - Registers
  - Stack frame

We Have to Make Many Choices…

- Specific set of arithmetic and logic instructions
- Addressing modes
- Architectural style
- Memory layout of the instruction (instruction format)
- Drivers of these decisions
  - Technology trends
  - Implementation feasibility
  - Goal of elegant/efficient support for high-level language constructs

More Addressing Modes

- All those we've seen
  - Register
  - PC-relative
  - Base+offset
  - Base+index
  - Indirect addressing (**`ld @ra)`**

- Pseudo-direct addressing
  - Address is formed from first 6 bits of PC and last 26 bits of instruction

Architecture Styles

- Accumulator oriented
  - Early digital computers
- Stack oriented
  - Burroughs
- Memory oriented
  - IBM s/360, DEC VAX, et al
- Register oriented
  - MIPS, Alpha, ARM, Power PC, CDC 6600
- Hybrid memory-register
  - PowerPC, x86

Instruction Formats

- Zero Operand Instructions
  - Halt, NOP
  - Stack machines: Add, Sub
- One Operand Instructions
  - Inc, Dec, Neg, Not
  - Accumulator machines: Load M, Add M
- Two Operand Instructions
  - Add r1, r2 (i.e., r1 = r1 + r2)
  - Mov r1, r2
- Three Operand Instructions
  - Add r1, r2, r3
  - Load rd, rb, offset

Instruction Format

Fixed Length Instructions

- Pros
  - Simplifies implementation
  - Can start decoding instructions immediately
- Cons
  - May waste space
  - May need additional logic in datapath
  - Limits instruction set designer

Variable Length Instructions

- Pros
  - No wasted space
  - Fewer constraints on designer
  - More flexibility with opcodes, addressing modes and operands
- Cons
  - Complicates implementation

Some History

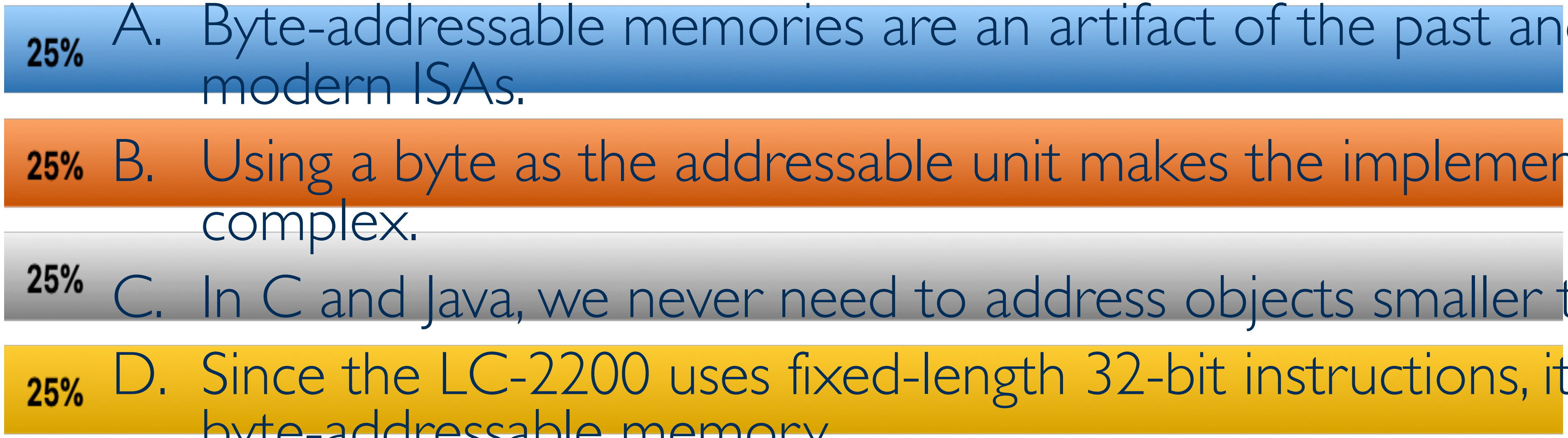
1940 – 1950	1960 – 1970	1980 – 1990
Hardware Expensive Memory Expensive	Hardware Cheaper Memory Expensive	Hardware Cheap Memory Cheap Microprocessors Compilers getting good
Accumulators (1-2) EDSAC IBM 701 UNIVAC I	Registers (8-16) Register-Memory IBM 360 DEC PDP-11 Univac 1108 Register-Register CDC 6600 Stack Burroughs B-5000	CISC DEC VAX Motorola 68000 Intel 80x86 RISC Berkeley RISC→Sparc Dave Patterson Stanford MIPS →SGI John Hennessy IBM 801 ARM

We've Made Some Choices

- The LC-2200
- RISC
- Register-register style
- Fixed-length, 32-bit, MIPS-like instructions
- 32-bit words, word addressable
- 16 registers
- Initially we define a very sparse set of instructions so there are still more choices to make

Session ID: dag2200

The LC-2200 is a 32-bit word-addressable ISA. Why is it not byte-addressable?



A. Byte-addressable memories are an artifact of the past and aren't used in modern ISAs.

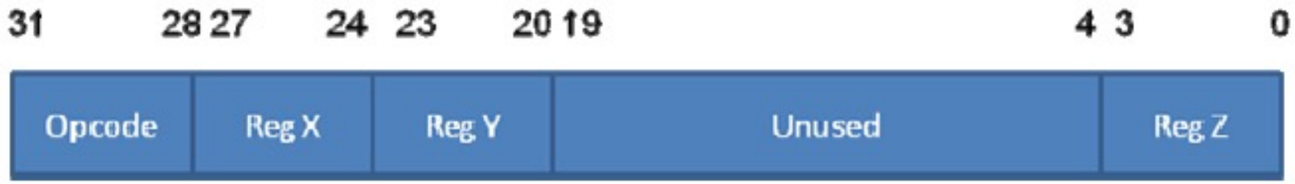
B. Using a byte as the addressable unit makes the implementation more complex.

C. In C and Java, we never need to address objects smaller than registers.

D. Since the LC-2200 uses fixed-length 32-bit instructions, it has no need for byte-addressable memory.

LC-2200 Instruction set

- R-type instructions (add, nand):



- I-type instructions (addi, lw, sw, beq):

- J-type instructions (jalr):

- O-type instructions (halt):

LC-2200 Register convention

Reg #	Name	Use	callee-save?
0	$zero	always zero (by hardware)	n.a.
1	$at	reserved for assembler	n.a.
2	$v0	return value	no
3-5	$a0-$a2	arguments	no
6-8	$t0-$t2	Temporaries	no
9-11	$s0-$s2	saved registers	YES
12	$k0	reserved for OS/traps	n.a.
13	$sp	stack pointer	no
14	$fp	frame pointer	YES
15	$ra	return address	no

Caller saves if you want to preserve them across a function call

Only if your function code will change them!

Always save

LC-2200 example mnemonics

Mnemonic Example	Format	Opcode	Action Register Transfer Language
add add $v0, $a0, $a1	R	0 0000_2	Add contents of reg Y with contents of reg Z, store results in reg X. RTL: $v0 ← $a0 + $a1
addi addi $v0, $a0, 25	I	2 0010_2	Add OFFSET to the contents of reg Y and store the result in reg X. RTL: $v0 ← $a0 + 25
lw lw $v0, 0x42($fp)	I	3 0011_2	Load reg X from memory. The memory address is formed by adding OFFSET to the contents of reg Y. RTL: $v0 ← MEM[$fp + 0x42]

beq
beq $a0, $a1,
    done
	I	5
		0101_2
			Compare the contents of reg X and reg Y. If they are the same, then branch to the address PC+1+OFFSET, where PC is the address of the beq instruction.
			RTL: if($a0 == $a1)
			        PC ← PC+1+OFFSET

Note: For programmer convenience (and implementer confusion), the assembler computes the OFFSET value from the number or symbol given in the instruction and the assemblers idea of the PC. In the example, the assembler stores done-(PC+1) in OFFSET so that the machine will branch to label "done" at run time.			

jalr jalr $at, $ra	J	6 0110_2	First store PC+1 into reg Y, where PC is the address of the jalr instruction. Then branch to the address now contained in reg X. Note that if reg X is the same as reg Y, the processor will first store PC+1 into that register, then end up branching to PC+1. RTL: $ra ← PC+1; PC ← $a0

Session ID: dag2200

What is true about register v0?

21% A. It can only be set and copied; it cannot be used for intermediate arithmetic operations nor can it store the return address during a JALR instruction.

11% B. It is stored in physical register number 0010_2.

21% C. Based on our calling convention, it holds the value being returned from a function unless the value is longer than 32 bits.

14% D. B and C only

18% E. A and B only

14% F. None of the above.

Issues Influencing Processor Design

- Instruction Set
- Applications
- Other
  - Operating system
  - Support for modern languages
  - Memory system
  - Parallelism
  - Debugging
  - Virtualization
  - Fault Tolerance
  - Security

Instruction Set

- Over-arching concern: Compiling high level language constructs into efficient machine code
- But other factors are in play
  - Market pressure
  - Performance
  - Technology workarounds

Influence of Applications on Instruction Set Design

- Number crunching requires efficient floating point
  - Development of floating point hardware
- Media applications deal with streaming data
  - Intel MMX extensions
- Gaming requires sophisticated graphic processing
  - Need GPU chips

A final note on Endianness