C to RISC-V II

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah	Ethan Weisberg	Maya Huizar
Angie Cao	Garrett O'Malley Kirsch	Meghana Vasireddy
August Fu	Hassan Rizwan	Perrie Quek
Caroline Begg	Iain Li	Sidharth Roy
Cathy Cao	Jerry Wang	Sydnie-Shea Cohen
Claire Lu	Juan Lopez	Vivi Li
Eric Sungwon Lee	Keith Mathe	Yousef AlRabiah

Poll Everywhere

pollev.com/cis2400

- How are you? Any Questions?

Logistics

- Please start HW9 as soon as you can.
- This one is much more time consuming than the others.
- There are only three weeks left…

Lecture Outline

- Application Binary Interface
  - X86, ARM, & RISC-V
- Register Convention
  - Frame Pointer and Return Address
  - Frame Records
  - Prologue & Epilogue
- Procedure Calling Convention
  - Argument Passing
  - Returning Values
- Linking and Loading
  - Absolute Addressing
  - Relative Addressing

Application Binary Interface

- Defines how programs and routines interact.
- 'Calling Convention'
  - Specifies how parameters are passed to functions and how arguments are received from different routines.

For example, you've never compiled the C standard libraries yourself, yet the functions you write can call them seamlessly, even though your code and the libraries ***were not compiled together***.

This means that if the ABI changes, we would need to recompile our code to ensure compatibility, ***even if no changes were made to the source code itself***.

Where do x86 and Arm fall?

- x86
  - Has only 15 General Purpose Registers + Instruction Pointer (***RIP***)
  - Passing Arguments and the Stack's general structure are not dissimilar to what we've seen in RISC-V but it is different. Particularly in how values are passed.

- Arm
  - There are 30 general-purpose registers but depending on the processors "Mode" we can access less or more.
  - The Frame Structure is also ***different.***
  - Procedure calling is also very different from RISC-V.
    - ***You can not*** save the Return Address in anything other than the 'Link Register' ***(lr)***

The Difference isn't just Hardware.

The differences between x86, ARM, and RISC-V go beyond just hardware or physical registers—they also involve ***how memory*** is organized, ***how functions*** are called, and ***how arguments*** are passed to routines.

Lecture Outline

- Application Binary Interface
  - X86, ARM, & RISC-V
- Register Convention
  - Frame Pointer and Return Address
  - Frame Records
  - Prologue & Epilogue
- Procedure Calling Convention
  - Argument Passing
  - Returning Values
- Linking and Loading
  - Absolute Addressing
  - Relative Addressing

Register Conventions

- Agreement on Register Usage:
  - Routines must know which registers are used for specific purposes
- Consistent Saving of Registers:
  - If registers need to be saved, **routines must agree on where to store them**.
    - How else would routines locate arguments/variables?
- **Provides a known location for return values.**
- **Allows routines to ‘reliably’ return to the caller.**

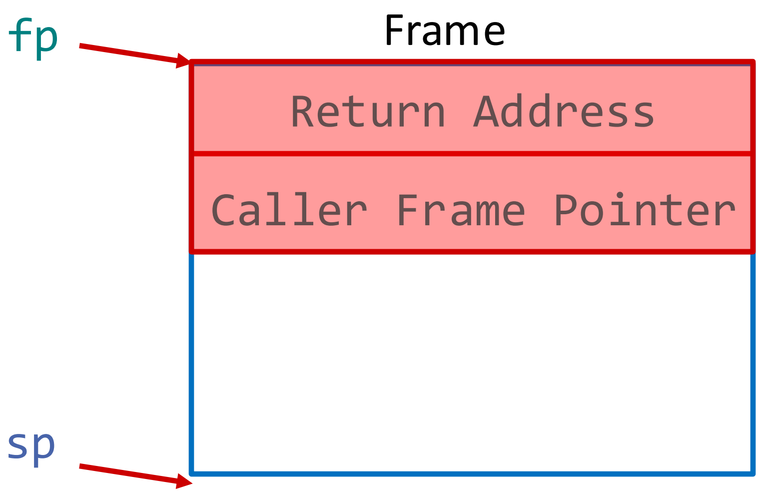
Registers

Yes, we don't care about these two.

x0/zero	x1/ra	x2/sp	x3/gp	x4/tp
x5/t0	x6/t1	x7/t2	x8/s0/fp	x9/s1
x10/a0	x11/a1	x12/a2	x13/a3	x14/a4
x15/a5	x16/a6	x17/a7	x18/s2	x19/s3
x20/s4	x21/s5	x22/s6	x23/s7	x24/s8
x25/s9	x26/s10	x27/s11	x28/t3	x29/t4
x30/t5	x31/t6			

Registers: The Agreement

x1/ra | x2/sp | x8/s0/fp



Frame Pointer Role:

- Points to the top of the bottom most stack frame

Frame Record Structure:

- First value: Holds the return address.
- Second value: Points to the previous frame.

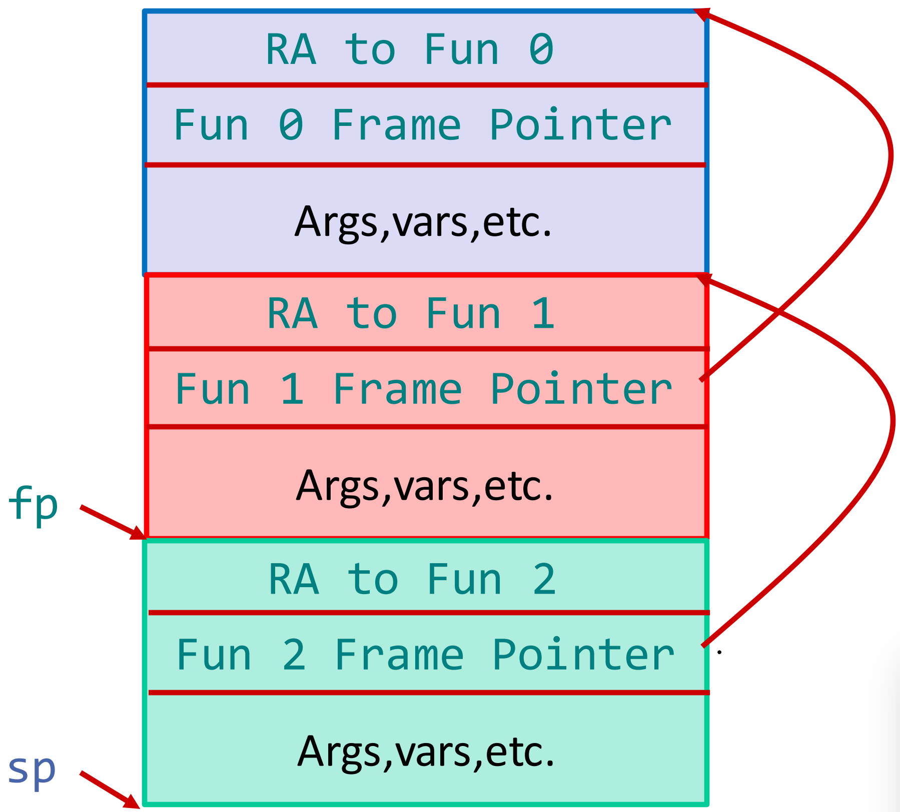
Always 4 bytes each.

Linked List of Frames:

- ***Each frame links back to its caller.***

Frame Record: Red Portion of Frame

Registers: The Agreement



Frame Pointer Role:

- Points to the bottom most stack frame, initiating a ***linked list*** of frames.
- **prev_frame = *(fp – 8)**

Linked List of Frames:

- Each frame links back to its caller.

Other than storing caller state, what else might this be useful for?

This is how a debugger figures out which functions called each other, letting you see the ***exact path*** that led to your 10-hour bug.

We say that the debugger "walks the stack".

Example of Walking the Stack: imessage

```
v 2.012s Thread_7611586   DispatchQueue_1: com.apple.main-thread  (serial)
 v 2.012s start  (in dyld) + 2476  [0x19a043154]
  v 2.012s ???  (in Messages)  load address 0x10049c000 + 0x1a68  [0x10049da68]
   v 2.012s UIApplicationMain  (in UIKitCore) + 148  [0x1c99637bc]
    v 2.012s UINSApplicationMain  (in UIKitMacHelper) + 972  [0x1b378bf50]
     v 2.012s _NSApplicationMainWithInfoDictionary  (in AppKit) + 24  [0x19df28654]
      v 2.012s NSApplicationMain  (in AppKit) + 880  [0x19dcd5240]
       v 2.007s -[NSApplication run]  (in AppKit) + 476  [0x19dcfdffc]
        v 2.001s -[NSApplication(NSEventRouting) _nextEventMatchingEventMask:untilDate:inMode:dequeue:]  (in AppKit) + 700  [0x19e5014
         v 2.001s _DPSNextEvent  (in AppKit) + 660  [0x19dd0acc8]
          v 2.001s _BlockUntilNextEventMatchingListInModeWithFilter  (in HIToolbox) + 76  [0x1a4c55d30]
           v 1.801s ReceiveNextEventCommon  (in HIToolbox) + 648  [0x1a4c55fd8]
            v 1.800s RunCurrentEventLoopInMode  (in HIToolbox) + 292  [0x1a4c5619c]
             v 1.796s CFRunLoopRunSpecific  (in CoreFoundation) + 608  [0x19a4ab434]
              v 1.041s __CFRunLoopRun  (in CoreFoundation) + 2244  [0x19a4ac350]
               v 1.041s __CFRunLoopDoSource1  (in CoreFoundation) + 524  [0x19a4ad98c]
                v 1.041s __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION__  (in CoreFoundation) + 60  [0x19a4ada6c]
                 v 1.041s __CFMachPortPerform  (in CoreFoundation) + 296  [0x19a4dcfc4]
                  v 1.041s display_timer_callback(__CFMachPort*, void*, long, void*)  (in QuartzCore) + 348  [0x1a26febf0]
                   v 1.013s CA::Display::DisplayLink::dispatch_items(unsigned long long, unsigned long long, unsigned long long)  (i
                    v 1.010s CA::Transaction::commit()  (in QuartzCore) + 648  [0x1a26a2864]
                     v 0.214s CA::Context::commit_transaction(CA::Transaction*, double, double*)  (in QuartzCore) + 808  [0x1a2844e
                      v 0.192s CA::Context::retain_all_contexts(bool, CA::Context**, unsigned long&, __CFArray const*)  (in QuartzC
                       v 0.137s _qsort  (in libsystem_c.dylib) + 1880  [0x19a291434]
                        v 0.094s _qsort  (in libsystem_c.dylib) + 1880  [0x19a291434]
                         v 0.042s _qsort  (in libsystem_c.dylib) + 1948  [0x19a291478]
                          v 0.016s _qsort  (in libsystem_c.dylib) + 1948  [0x19a291478]
                           v 0.011s _qsort  (in libsystem_c.dylib) + 1880  [0x19a291434]
                            v 0.004s _qsort  (in libsystem_c.dylib) + 1880  [0x19a291434]
                               0.003s _qsort  (in libsystem_c.dylib) + 1144,1172,...  [0x19a291154,0x19a291170,...]
                             v 0.001s _qsort  (in libsystem_c.dylib) + 1880  [0x19a291434]
                                0.001s _qsort  (in libsystem_c.dylib) + 1500  [0x19a2912b8]
```

} Aw a qsort!

Understanding the specific functions here isn't important; what matters is that each function is traced up the stack using the same method.

This is how a debugger figures out which functions called each other, letting you see the ***exact path*** that led to your 10-hour bug.

We say that the debugger "walks the stack".

Prologue and Epilogue

```
Foo:
    addi sp, sp, -VAL
    sw ra, (VAL – 4)(sp)
    sw s0, (VAL – 8)(sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, (VAL – 4)(sp)
    lw s0, (VAL – 8)(sp)
    addi sp, sp, VAL
    jalr x0, ra, 0
```

- Prologue
  - Setting Up Function's Frame and Storing Callee-Saved Registers

Initial Stack Allocation

If more memory is needed, the stack pointer will be lowered.

(VAL – 4) & (VAL – 8) put ra and s0 in the correct memory location always.

Prologue and Epilogue

```
Foo:
    addi sp, sp, -VAL
    sw ra, (VAL – 4)(sp)
    sw s0, (VAL – 8)((sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, (VAL – 4)(sp)
    lw s0, (VAL – 8)(sp)
    addi sp, sp, VAL
    jalr x0, ra, 0
```

- Prologue
  - Setting Up Function's Frame and Storing Callee-Saved Registers

If more memory is needed, the stack pointer will be lowered.

- Epilogue
  - Cleaning up Function's Frame and restoring Callee-Saved Registers

Poll Everywhere

pollev.com/cis2400

```
Foo:
    addi sp, sp, -8
    sw ra, VAL1(sp)
    sw s0, VAL2(sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, VAL1(sp)
    lw s0, VAL2(sp)
    addi sp, sp, 8
    jalr x0, ra, 0
```

What should the values of VAL1 and VAL2 be to set up the frame and exit the routine correctly?

A)
VAL1 = 0
VAL2 = 4

B)
VAL1 = 4
VAL2 = 0

C)
VAL1 = 8
VAL2 = 4

D)
VAL1 = 8
VAL2 = 4

Poll Everywhere

pollev.com/cis2400

```
Foo:
    addi sp, sp, -8
    sw ra, VAL1(sp)
    sw s0, VAL2(sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, VAL1(sp)
    lw s0, VAL2(sp)
    addi sp, sp, 8
    jalr x0, ra, 0
```

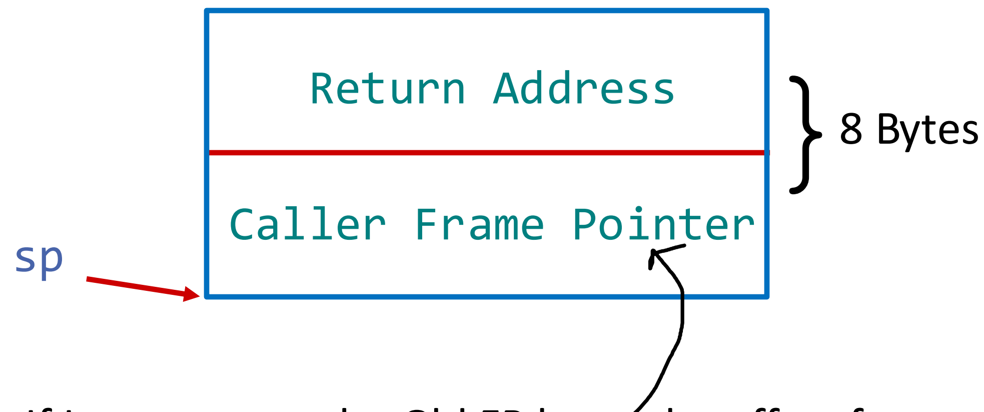
What should the values of VAL1 and VAL2 be to set up the frame and exit the routine correctly?

Poll Everywhere

pollev.com/cis2400

```
Foo:
    addi sp, sp, -8
    sw ra, VAL1(sp)
    sw s0, VAL2(sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, VAL1(sp)
    lw s0, VAL2(sp)
    addi sp, sp, 8
    jalr x0, ra, 0
```

What should the values of VAL1 and VAL2 be to set up the frame and exit the routine correctly?



If I want to put the Old FP here, the offset from SP must be 0.

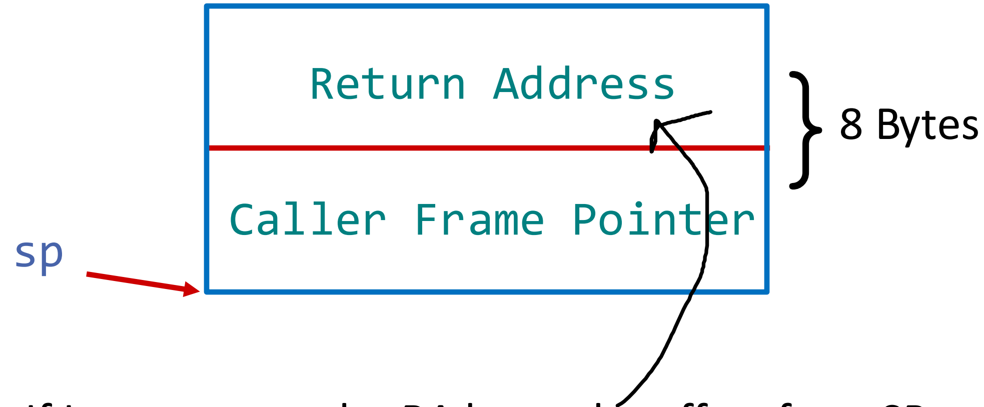
`VAL2 = 0`

Poll Everywhere

pollev.com/cis2400

```
Foo:
    addi sp, sp, -8
    sw ra, VAL1(sp)
    sw s0, VAL2(sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, VAL1(sp)
    lw s0, VAL2(sp)
    addi sp, sp, 8
    jalr x0, ra, 0
```

What should the values of VAL1 and VAL2 be to set up the frame and exit the routine correctly?



If I want to put the RA here, the offset from SP must be 4.

VAL1 = 4

Poll Everywhere

pollev.com/cis2400

```
Foo:
    addi sp, sp, -8
    sw ra, VAL1(sp)
    sw s0, VAL2(sp)
    addi s0, sp, VAL

    //awesome instructions
    //amazing instructions
    //even better instructions
    //meh instructions
    //ready to return!


    lw ra, VAL1(sp)
    lw s0, VAL2(sp)
    addi sp, sp, 8
    jalr x0, ra, 0
```

What should the values of VAL1 and VAL2 be to set up the frame and exit the routine correctly?

A)

```
VAL1 = 0
VAL2 = 4
```

B)

```
VAL1 = 4
VAL2 = 0
```

C)

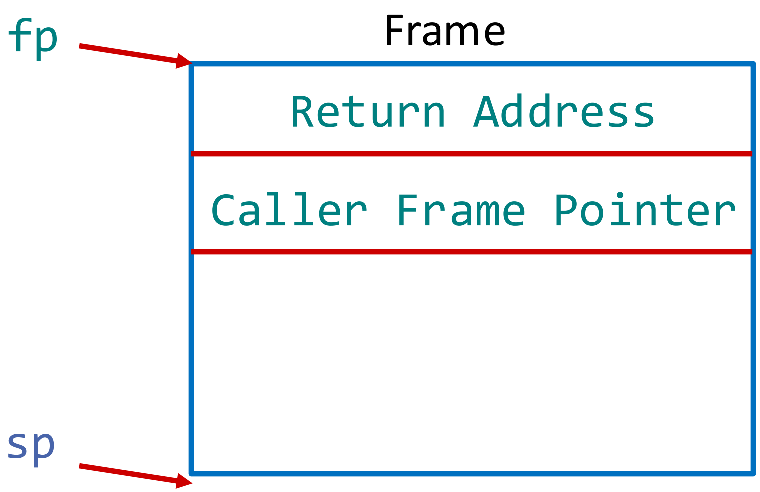
```
VAL1 = 8
VAL2 = 4
```

D)

```
VAL1 = 8
VAL2 = 4
```

RISC-V's Leniency

Official Verbiage: The frame pointer points to the *Canonical Frame Address* (CFA), ***which is the stack pointer value at the function's entry.***



The return address is located @ fp – 4

The previous frame pointer is @ fp – 8

What does *RISC-V* tell us?

It is left to the 'platform' to determine the level of conformance with this convention.

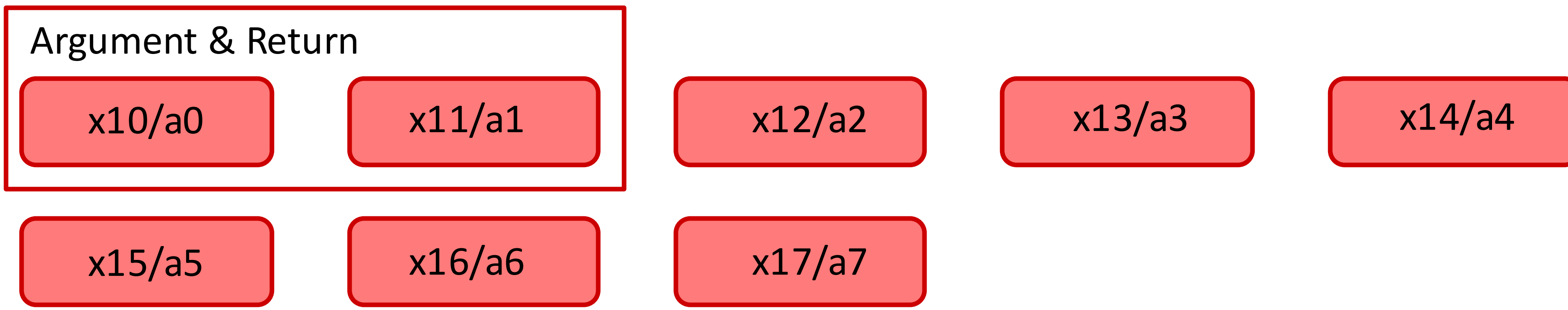
specific hardware environment or operating system

Lecture Outline

- Application Binary Interface
  - X86, ARM, & RISC-V
- Register Convention
  - Frame Pointer and Return Address
  - Frame Records
  - Prologue & Epilogue
- Procedure Calling Convention
  - Argument Passing
  - Returning Values
- Linking and Loading
  - Absolute Addressing
  - Relative Addressing

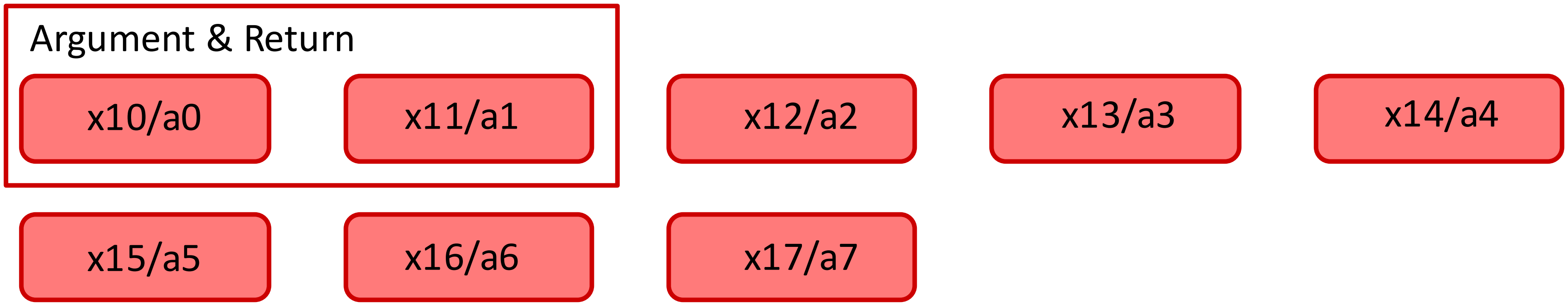
Argument Passing

- Argument Registers
  - These *pass* arguments to functions
  - Only ***two registers*** *'return'* values.



Registers

- What if Arguments don't fit across all of these?



Naïve Solution: Why not always just write values to the callee's frame from the start and then have the callee load them? This way, we never stress about this issue.

Registers are ***FAST.*** If arguments are already loaded into registers, we can do arithmetic operations immediately. We can spend less time waiting for the values to be loaded from memory.

Argument Passing: Just Enough

```
int function(int one, int two, int three, int four,
                  int five, int six, int seven, int eight){
        //do nothing
}
```

```
function:
    addi sp, sp, -48
    sw ra, 44(sp)
    sw s0, 40(sp)
    addi s0, sp, 48
    sw a0, -16(s0)
    sw a1, -20(s0)
    sw a2, -24(s0)
    sw a3, -28(s0)
    sw a4, -32(s0)
    sw a5, -36(s0)
    sw a6, -40(s0)
    sw a7, -44(s0)
```

When we use all 8 arguments, we use all 8 registers, a0 – a7

Argument Passing: Different Sizes

```
int function(char one, char two, char three, char four,
                 short five, short six, short seven, short eight){
        //do nothing
}
```

```
function:
    addi sp, sp, -32
    sw ra, 28(sp)
    sw s0, 24(sp)
    addi s0, sp, 32
    sb a0, -13(s0)
    sb a1, -14(s0)
    sb a2, -15(s0)
    sb a3, -16(s0)
    sh a4, -18(s0)
    sh a5, -20(s0)
    sh a6, -22(s0)
    sh a7, -24(s0)
```

When we use all 8 arguments, we use all 8 registers, a0 – a7.

Even if the arguments are different types!

The only things that change are the offsets and storing instructions.

Arguments up to 4 bytes in size are passed with registers, or on the stack by value if no registers are available.

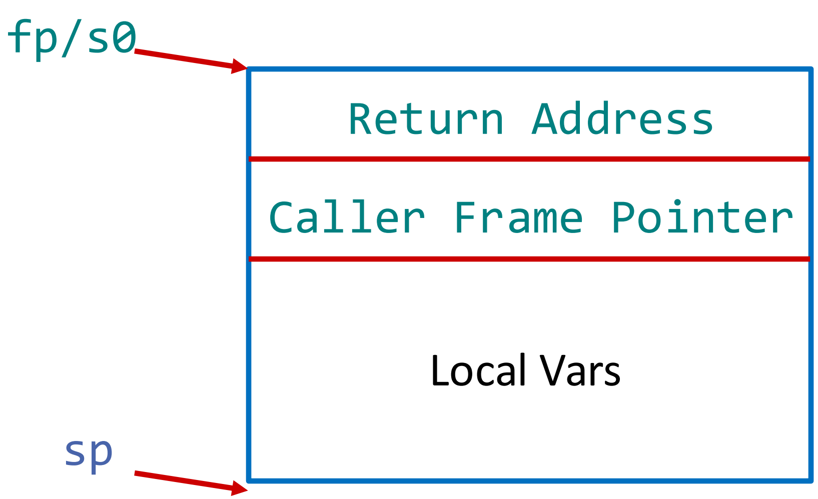
Argument Passing: One Register Short

```c
int function(int one, int two, int three, int four,
                 int five, int six, int seven, int eight, int nine){
        //do nothing
}
```

```
function:
    addi sp, sp, -48
    sw ra, 44(sp)
    sw s0, 40(sp)
    addi s0, sp, 48
    lw t0, 0(s0)
    sw a0, -12(s0)
    sw a1, -16(s0)
    sw a2, -20(s0)
    sw a3, -24(s0)
    sw a4, -28(s0)
    sw a5, -32(s0)
    sw a6, -36(s0)
    sw a7, -40(s0)
```

When we use 9 arguments, we use all 8 registers, a0 – a7, ***and the stack***.

`lw t0, 0(s0)` "Load the value above the frame"



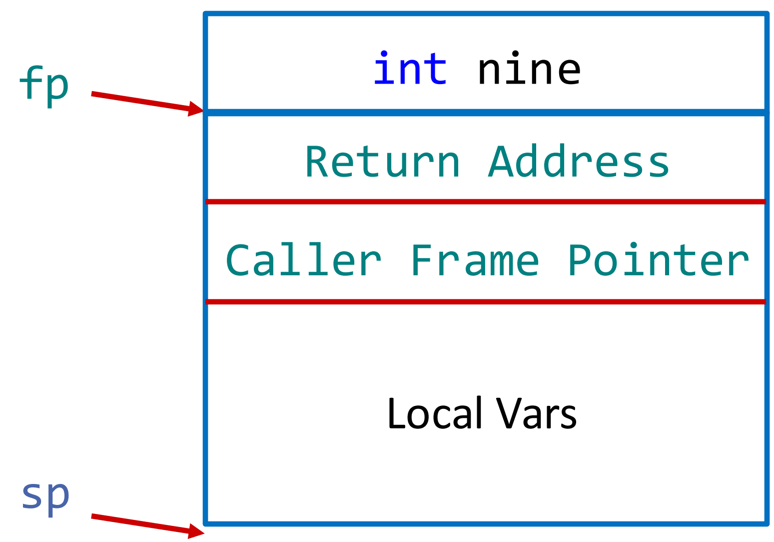
Argument Passing: One Register Short

```c
int function(int one, int two, int three, int four,
                  int five, int six, int seven, int eight, int nine){
        //do nothing
}
```

```
function:
    addi sp, sp, -48
    sw ra, 44(sp)
    sw s0, 40(sp)
    addi s0, sp, 48
    lw t0, 0(s0)
    sw a0, -12(s0)
    sw a1, -16(s0)
    sw a2, -20(s0)
    sw a3, -24(s0)
    sw a4, -28(s0)
    sw a5, -32(s0)
    sw a6, -36(s0)
    sw a7, -40(s0)
```

These already have the arguments

The caller places the arguments at the end of its own frame, allowing the callee to retrieve any arguments that don't fit in registers.



What about Structs?

```
typedef struct {
    int x;
    int y;
} pair;


pair make_struct(int x, int y){
        return (pair){x, y};
}
```

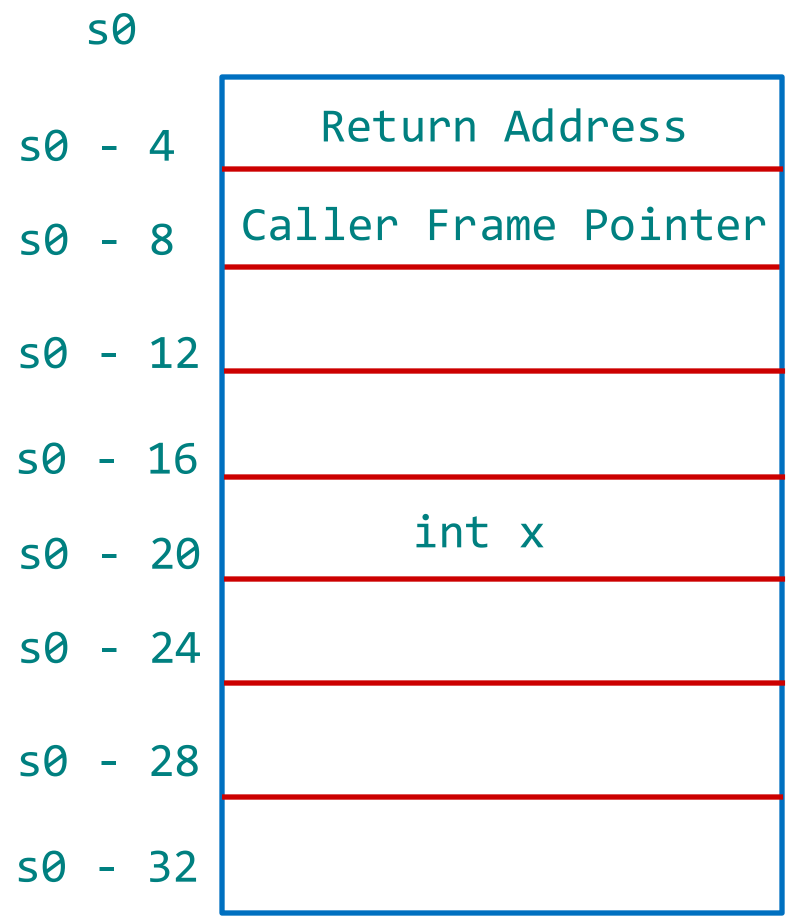
```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```

Registers and the Stack

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)   ←
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```

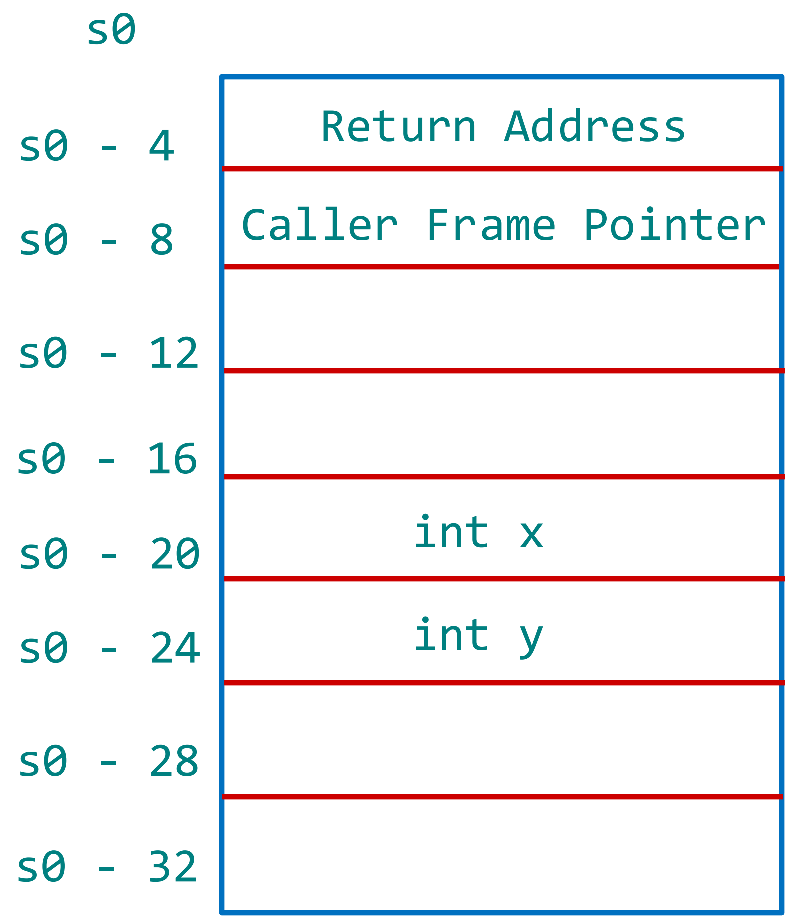
a0: int x

a1: int y



Registers and the Stack

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)   <-
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```

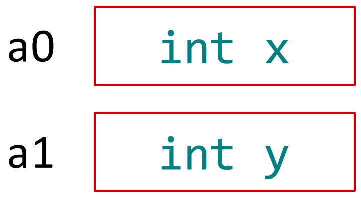


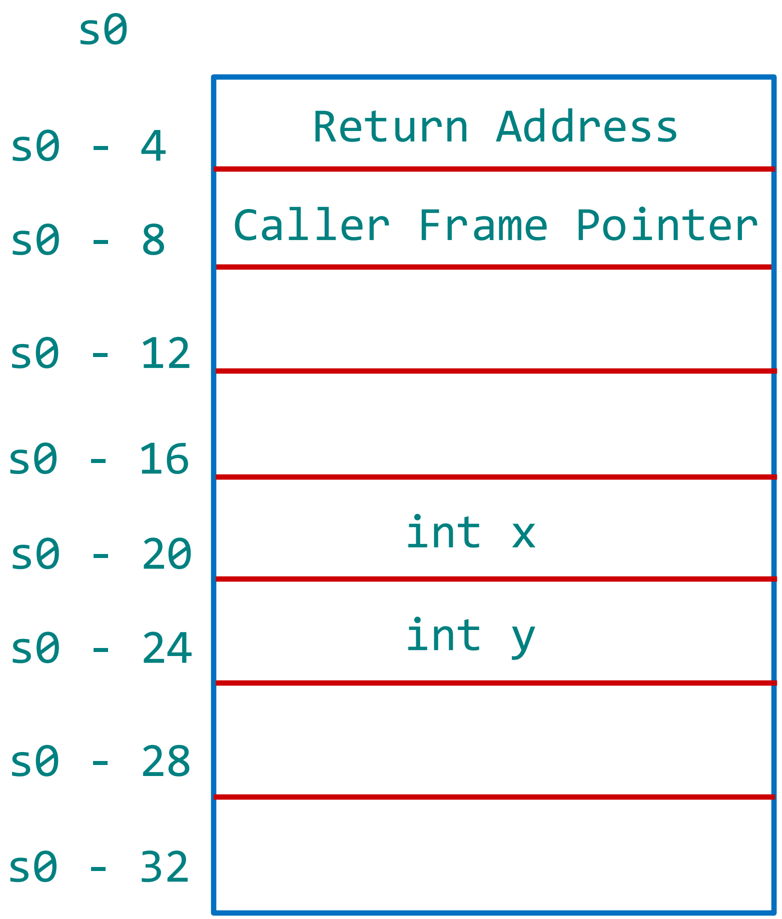
a0 | int x

a1 | int y

Registers and the Stack

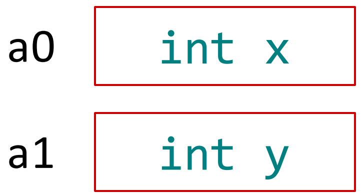
```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)   <--
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```

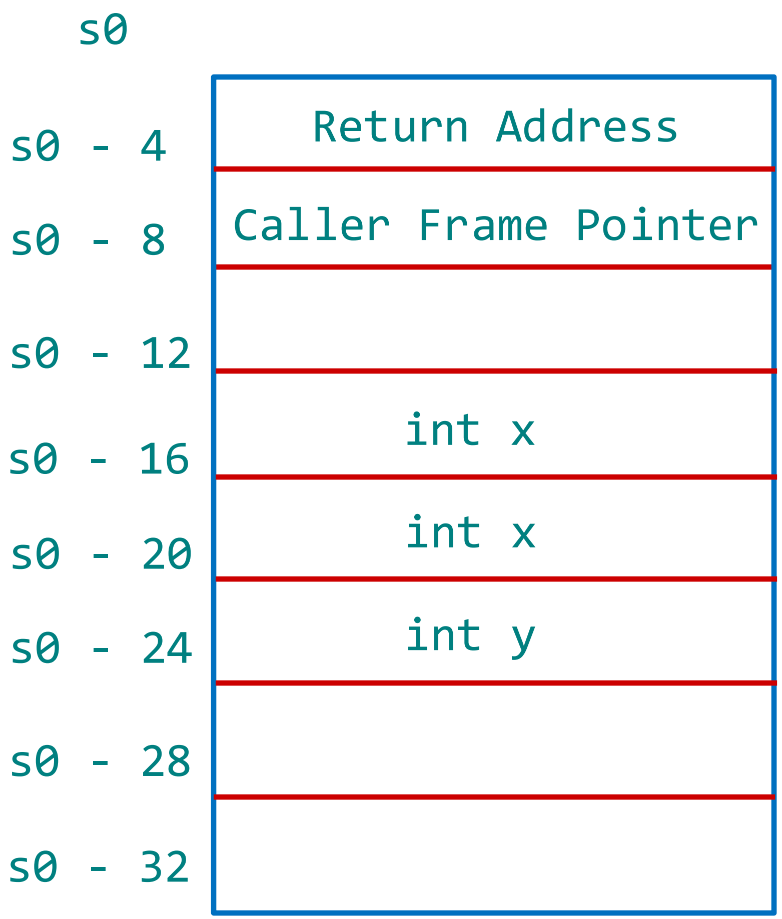




Registers and the Stack

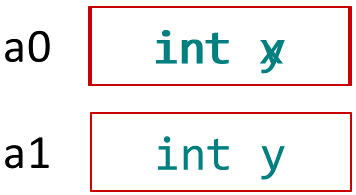
```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)   ←
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```

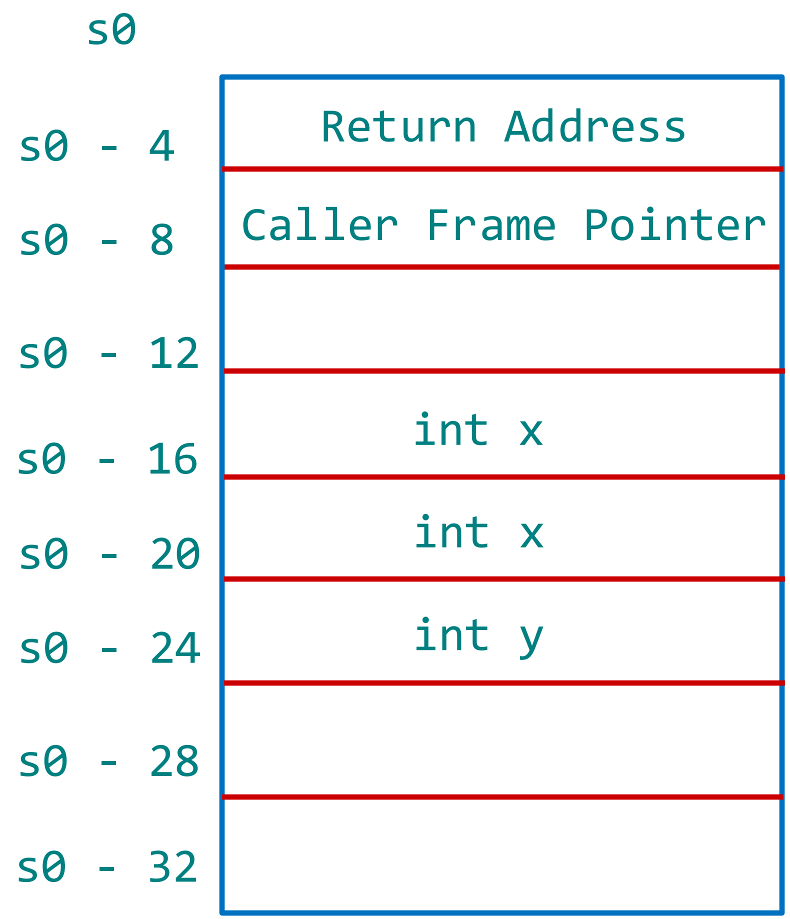




Registers and the Stack

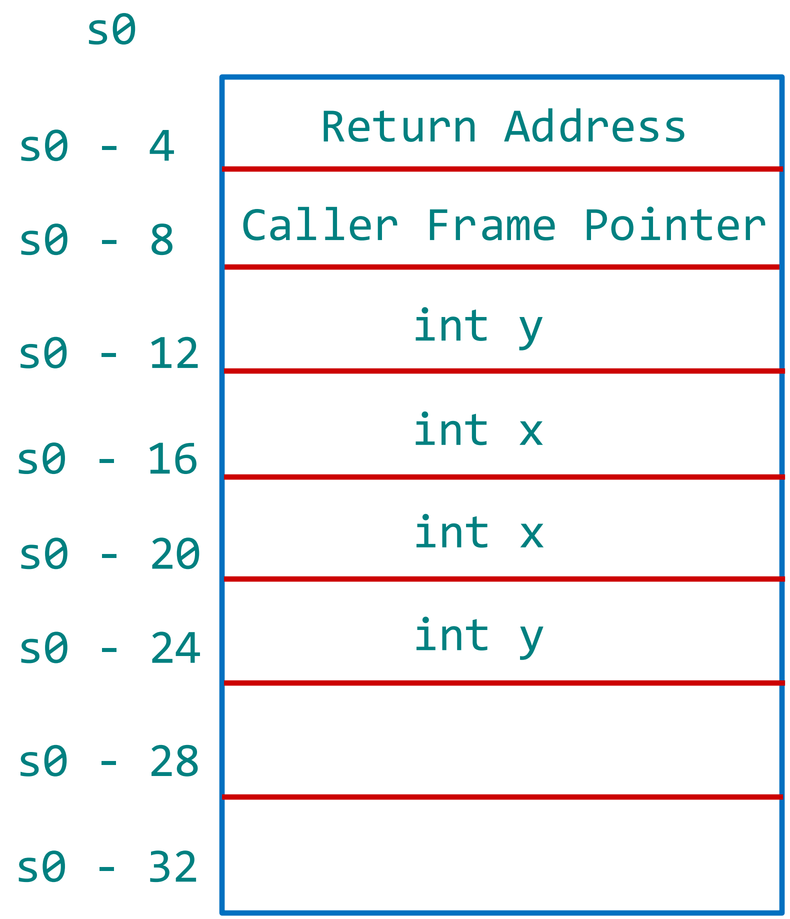
```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)   ←
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```





Registers and the Stack

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)   ←
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```



a0: int y

a1: int y

Registers and the Stack

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)   ←
    lw a1, -12(s0)

    //rest of epilogue omit..
```

a0	int x
a1	int y

s0	
s0 - 4	Return Address
s0 - 8	Caller Frame Pointer
s0 - 12	int y
s0 - 16	int x
s0 - 20	int x
s0 - 24	int y
s0 - 28	
s0 - 32	

Registers and the Stack

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)   ←

    //rest of epilogue omit..
```

a0	int x
a1	int y

This is the 'struct' being returned.

s0	
s0 - 4	Return Address
s0 - 8	Caller Frame Pointer
s0 - 12	int y
s0 - 16	int x
s0 - 20	int x
s0 - 24	int y
s0 - 28	
s0 - 32	

Registers and the Stack

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)  <-

    //rest of epilogue omit..
```

```
typedef struct {
    int x;
    int y;
} pair;


pair make_struct(int x, int y){
        return (pair){x, y};
}
```

Internally, this struct is treated as two separate integers to be returned to the caller, rather than ***as a single unit***.

a0	int x
a1	int y

This is the 'struct' being returned.

The Official Verbiage

```
makestruct:
    addi sp, sp, -32

    //rest of prologue omit..

    sw a0, -20(s0)
    sw a1, -24(s0)
    lw a0, -20(s0)
    sw a0, -16(s0)
    lw a0, -24(s0)
    sw a0, -12(s0)
    lw a0, -16(s0)
    lw a1, -12(s0)

    //rest of epilogue omit..
```

a0	int x
a1	int y

This is the 'struct' being returned.

Structs/Arrays up to 2 Words in size are passed across two registers.

If only one register is available, the first Word is passed in the register, and the remaining Word goes on the stack.

If no registers are available or if it's too large, the entire structure or array is passed on the stack.

This is called 'Register Spilling'

Poll Everywhere

pollev.com/cis2400

```
typedef struct {
    char filename[255];
} filestr;


filestr makestruct(){
        return (filestr){};
}
```

This function returns a 'filestr' struct without modifying the array. **According to the rules the *callee* should….**

A) Create a copy of the struct in the callee's frame, then copy it to the caller, even if the array is empty.

B) Allocate space for the struct in the caller's frame and then return.

C) Do nothing; the caller should have already allocated space for the struct, so the callee only needs to create it

D) When is lecture over?

Poll Everywhere

pollev.com/cis2400

```c
typedef struct {
    char filename[255];
} filestr;


filestr makestruct(){
        return (filestr){};
}
```

This function returns a ‘filestr’ struct without modifying the array. **According to the rules the *callee* should….**

A) Create a copy of the struct in the callee’s frame, then copy it to the caller, even if the array is empty.

This approach would be very inefficient. It involves unnecessary duplication and copying, which wastes both time and memory.

Imagine if the struct used an array of 1<< 11 bytes, not very good.

Poll Everywhere

pollev.com/cis2400

```c
typedef struct {
   char filename[255];
} filestr;


filestr makestruct(){
        return (filestr){};
}
```

This function returns a 'filestr' struct without modifying the array**. According to the rules the *callee* should….**

C) Allocate space for the struct in the caller's frame and then return.

If the callee were to allocate space for the caller, it would break the standard calling convention.

Additionally, ***the callee cannot directly modify the caller's stack pointer***, which would be necessary to allocate space in the caller's frame.

Poll Everywhere

pollev.com/cis2400

```c
typedef struct {
    char filename[255];
} filestr;


filestr makestruct(){
        return (filestr){};
}
```

This function returns a 'filestr' struct without modifying the array**. According to the rules the *callee* should….**

B) Not do anything, the caller had to have made space for the struct already. All we do is 'create' it.

If no registers are available or if it's too large, the entire structure or array is passed on the stack.

There is ***no way to 'pass' something back to the caller***, unless the caller already has allocated space for the struct or array ***it expects.***

Let’s see the compiled code

```
typedef struct {
    char filename[255];
} filestr;


filestr makestruct(){
        return (filestr){};
}
```

```
makestruct:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    //prologue over

    sw a0, -12(s0)
    li a1, 0
    li a2, 255
    call memset

    //epilogue start
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

???? → call memset

memset(void *b, int c, size_t len);

The **memset**() function writes len bytes of value c to the string b.

a0	void *b
a1	int c
a2	size_t len

This means a0 is the address of the filestr struct in the caller!

Registers and the Stack

```c
typedef struct {
   char filename[255];
} filestr;


filestr makestruct(){
        return (filestr){};
}
```

```
makestruct:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s0, 8(sp)
    addi s0, sp, 16
    //prologue over

    sw a0, -12(s0)
    li a1, 0
    li a2, 255
    call memset

    //epilogue start
    lw ra, 12(sp)
    lw s0, 8(sp)
    addi sp, sp, 16
    ret
```

The call to `memset` zeros out the structs char array in the caller. In this way, the function does "create" the struct by setting up its initial state, a zero'd out array.

Note: You weren't expected to know that a `memset` call would be used here. However, this behavior is defined in the C standard.

Registers and the Stack

```
typedef struct {
    char filename[255];
} filestr;


filestr makestruct(){
    return (filestr){};
}

int main() {
    mystruct cpy = makestruct();
}
```

```
makestruct:
    //prologue omit

    sw a0, -12(s0)
    li a1, 0
    li a2, 255
    call memset

    //epilogue omit

main:
    addi sp, sp, -272
    sw ra, 268(sp)
    sw s0, 264(sp)
    addi s0, sp, 272
    addi a0, s0, -263
    call makestruct
    li a0, 0
    lw ra, 268(sp)
    lw s0, 264(sp)
    addi sp, sp, 272
    ret
```

Look at this huge stack allocation.

Here we set `a0 = s0 - 263`

We ***pass in*** the address of the struct's location in the caller's memory implicitly. Even if the function doesn't take any arguments! wow.

More Practice

```
typedef struct {
    char a;
    char b;
    short size;
} smallstruct;


smallstruct make_smallstruct(){
        return (smallstruct){0,1,2};
}
```

total 4 bytes in size

```
make_smallstruct:
    //prologue omited

    li a0, 0
    sb a0, -12(s0)
    li a0, 1
    sb a0, -11(s0)
    li a0, 2
    sh a0, -10(s0)
    lhu a0, -10(s0)
    slli a0, a0, 16    }
    lhu a1, -12(s0)    }
    or a0, a0, a1      }

    //epilogue omited
```

tldr; this struct is forced into a 32 bit register.

“If it fits in a single register, put it in one”

You can see this here.

More Official RISC-V Rules

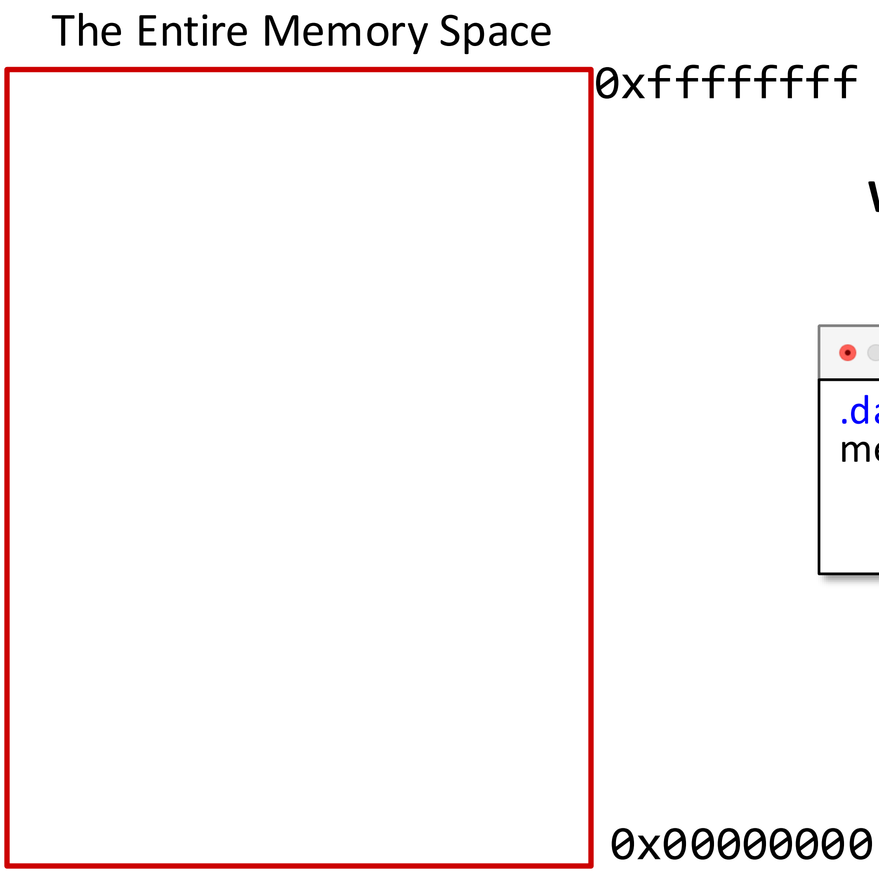
- Procedures should **not rely on stack-allocated data** below the current stack pointer, as it may not persist.
- The stack **grows downwards** (toward lower addresses).
- On procedure entry:
  - The **stack pointer** must be **aligned to a 128-bit boundary**.
  - The **first argument** passed on the stack is located at **offset zero** from the stack pointer, with subsequent arguments stored at higher addresses.
- **Registers s0 to s11** must be **preserved** across procedure calls.
- **Floating-point registers** are **not preserved** across calls (not important for us).

Lecture Outline

- Application Binary Interface
  - X86, ARM, & RISC-V
- Register Convention
  - Frame Pointer and Return Address
  - Frame Records
  - Prologue & Epilogue
- Procedure Calling Convention
  - Argument Passing
  - Returning Values
- Linking and Loading
  - Absolute Addressing
  - Relative Addressing

Loading: Absolute Addressing

- **What Are Absolute Addresses?**
  - Direct 32-bit addresses that refer to a fixed memory location.
  - Used to load data or access ***specific memory locations***.



We can then, technically, load *anything* if we have the correct address.

```
.data
message:
        .asciiz "Hello, World!"
```

If we want to use this string, we have to load the address of it first.

Loading: Absolute Addressing

What Are Absolute Addresses?

- Direct 32-bit addresses that refer to a fixed memory location.
- Used to load data or access ***specific memory locations***.

```
.data
message:
            .asciiz "Hello, World!"


.text
.globl main
main:
      lui a0, %hi(message)
      addi a0, a0, %lo(message)
      # Call printf with
      # 'message' address in a0

      call printf
```

lui a0, %hi(message)

- loads the high 20 bits of the message label's address into a0.

addi a0, a0, %lo(message)

- loads the lower 12 bits of the message label's address into a0.

These two instructions allow you to construct **any 32-bit address.** This is how programs load values from the data segment.

Loading: PC - Relative Addressing

What Are Relative Addresses?

- 32-bit addresses who’s location we only know ‘relative’ to the PC.
- Used to jump to labels/routines during runtime.

This allows the code to be ***position-independent***

```
.text

main:
        li a0, 1
        jal ra, foo
        li a1, 2
        ret

foo:
        li a0, 10
        addi a0, a0, 5
        jalr ra, ra, 0
```

If foo is always located below main, we can jump to it using a relative offset from the current program counter (PC).

We don’t need the exact address of foo; we just need to know its position relative to main.

Loading: PC - Relative Addressing

This allows the code to be ***position-independent***

```
.text

main:
        li a0, 1
        jal ra, foo
        li a1, 2
        ret

foo:
        li a0, 10
        addi a0, a0, 5
        jalr ra, ra, 0
```

If foo is always located below main, we can jump to it using a relative offset from the current program counter (PC).

We don't need the exact address of foo; we just need to know its position relative to main.

What does jal do? pc += se(imm20<<1)

This means jumps have a limit for 'how far' we can jump to a routine using **if we only use jal**.

This is approximately a ±1 MiB range relative to the PC.

If a routine is farther away, we can not just use a jal.

What is a routine is farther away?

```
lui t0, Imm
jalr ra, Imm(t0)
```

JALR (Jump and Link Register) is designed to allow a two-instruction sequence to jump to ***any 32-bit absolute address.***

lui (Load Upper Immediate) loads the **upper 20 bits** of the relative offset

jalr uses t0 as the base address and adds an offset to form the complete 32-bit offset.

This essentially allows us to jump almost ***anywhere.***

```
auipc t0, Imm
jalr ra, Imm(t0)
```

This is another way to to jump to ***any 32-bit absolute address.***

Unfortunately, the 'real functionality' is a bit more complex than what's shown in these slides.
However, the main idea remains the same.

Next time!

- More C to RISC-V with Travis!