C to RISC-V Wrap-Up

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

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

Poll Everywhere

pollev.com/tqm

- Any Questions?

Logistics

- HW 10 & 11
  - J Compiler
  - Last homework, but on the bigger side
  - You have to write a compiler for a fake language
    - Real compilers do a lot more, but it should help you understand some of what goes into the compilation process
  - Two parts:
    - HW10: tokenizer and basic assembly generation
    - HW11: Advanced assembly generation. Function calls, loops, ifs, etc

- Final Exam on December 16th @ 9am

J Compiler Demo

- If looking at the slides and want to see this, look at the recording

- There should also be a recitation on this soon

Lecture Outline

- **C to ASM Functions & Stack**
  - **Review**
  - **Growing the stack & Local Variables**
- If statements in ASM
- While loops in ASM

Review Questions

- Some of these questions are not good exam questions
- They may have some “traps”
- I don’t like multiple choice, but that is what PollEV has got

Poll Everywhere

pollev.com/tqm

- Which of these is true about the frame pointer?

A. The frame pointer grows as the stack frame grows

B. The frame pointer is the "base" of a stack frame

C. The frame pointer is saved by the caller of a function so that it can be restored as the function being called returns

D. The frame pointer is the same as the stack pointer

Poll Everywhere

pollev.com/tqm

- Which of these is true about the stack pointer?

A. Keeps track of the current stack frame. Specifically, where the previous function's frame ends and the current function's frame ends

B. When a function is invoked, the callee stores the previous stack pointer onto its own stack frame so it can be retrieved later

C. The stack pointer doesn't need to be saved explicitly. The callers stack frame bottom is just the top of the callees stack frame that gets popped off the stack

D. When we call a function, the function grows the stack at least enough to store the return address, return value, caller's frame pointer and caller's stack pointer

Poll Everywhere

pollev.com/tqm

- Which of these is true about the return address?

A. Return Address keeps track of where caller’s stack frame was so we can restore it when we return to the caller

B. Return Address is set by the JALR and JAL instructions and stored in x0

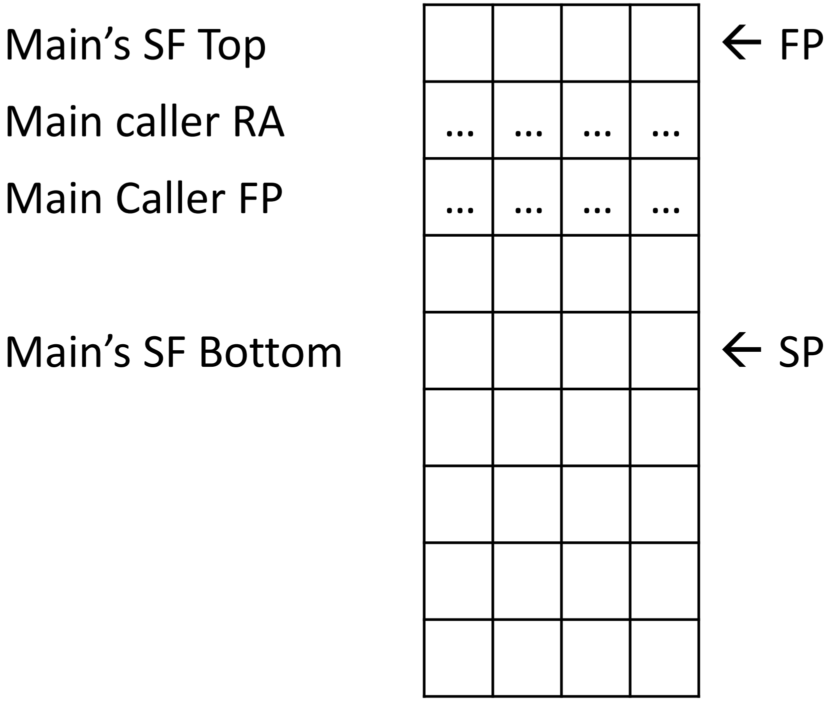
C. The Return Address keeps track of the instruction after JAL or JALR that was invoked by the caller.

D. The caller stores the return address on its stack frame so that it can be gotten later

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```



Main's SF Top ← FP

Main caller RA … … … …

Main Caller FP … … … …

Main's SF Bottom ← SP

Call foo…

```
foo(int, int):
            addi     sp, sp, -16
            sw       ra, 12(sp)
            sw       fp, 8(sp)
            addi     fp, sp, 16
            sw       a0, -12(fp)
            sw       a1, -16(fp)
            li       a0, 5
            lw       ra, 12(sp)
            lw       fp, 8(sp)
            addi     sp, sp, 16
            ret

main:
            addi     sp, sp, -16
            sw       ra, 12(sp)
            sw       fp, 8(sp)
            addi     fp, sp, 16
            li       a0, 3
            li       a1, 7
       →    call     foo(int, int)
            sw       a0, -12(fp)
            li       a0, 0
            lw       ra, 12(sp)
            lw       fp, 8(sp)
            addi     sp, sp, 16
            ret
```

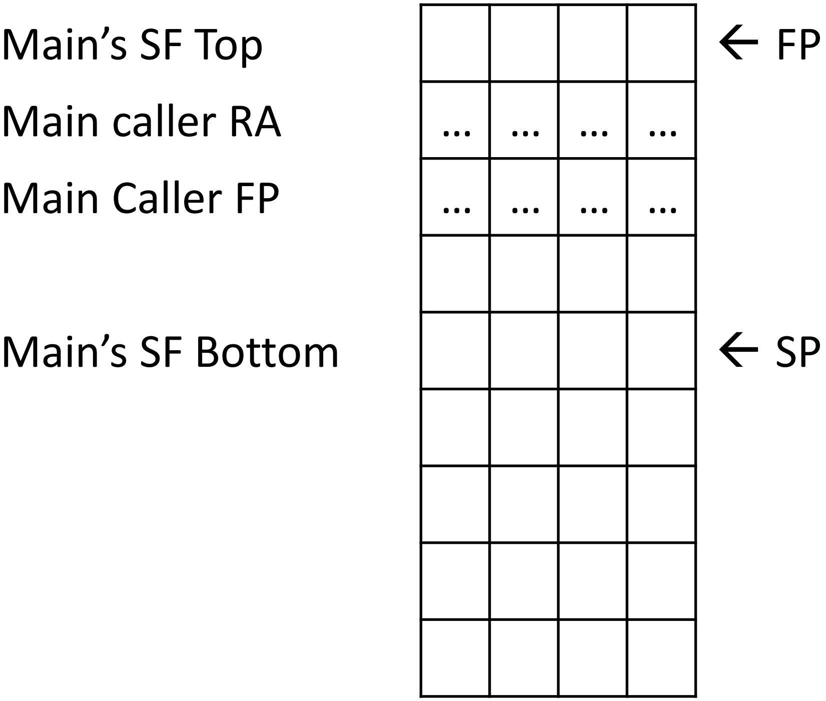
Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```c
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```

In foo, need to allocate stack frame

```
foo(int, int):
        →       addi      sp, sp, -16
                sw        ra, 12(sp)
                sw        fp, 8(sp)
                addi      fp, sp, 16
                sw        a0, -12(fp)
                sw        a1, -16(fp)
                li        a0, 5
                lw        ra, 12(sp)
                lw        fp, 8(sp)
                addi      sp, sp, 16
                ret

main:
                addi      sp, sp, -16
                sw        ra, 12(sp)
                sw        fp, 8(sp)
                addi      fp, sp, 16
                li        a0, 3
                li        a1, 7
                call      foo(int, int)
      ra  →     sw        a0, -12(fp)
                li        a0, 0
                lw        ra, 12(sp)
                lw        fp, 8(sp)
                addi      sp, sp, 16
                ret
```



Main's SF Top ← FP

Main caller RA … … … …

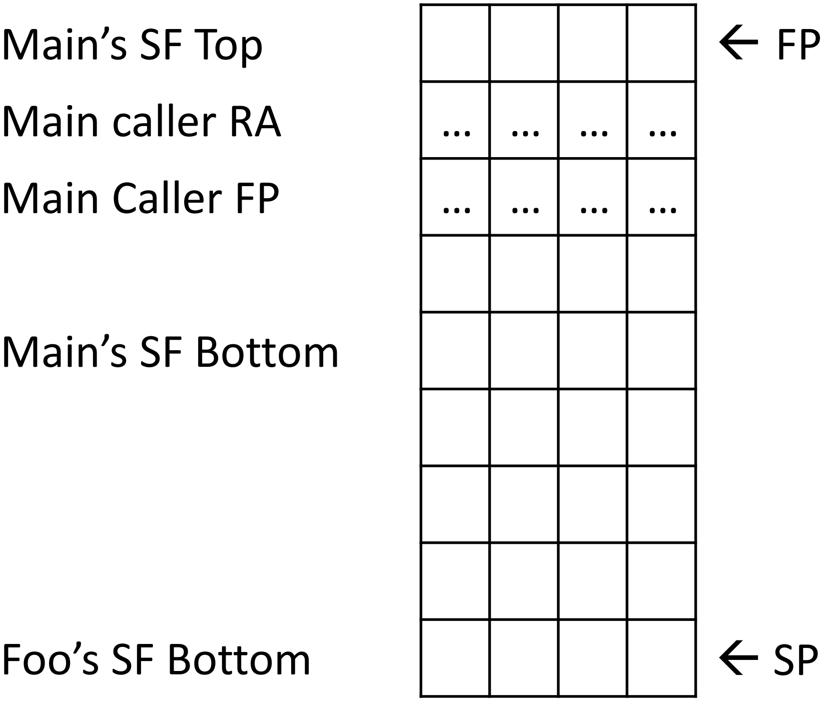
Main Caller FP … … … …

Main's SF Bottom ← SP

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```



Main's SF Top

Main caller RA

Main Caller FP

Main's SF Bottom

Foo's SF Bottom

← FP

← SP

…	…	…	…
…	…	…	…

Store copy of RA so we can return later

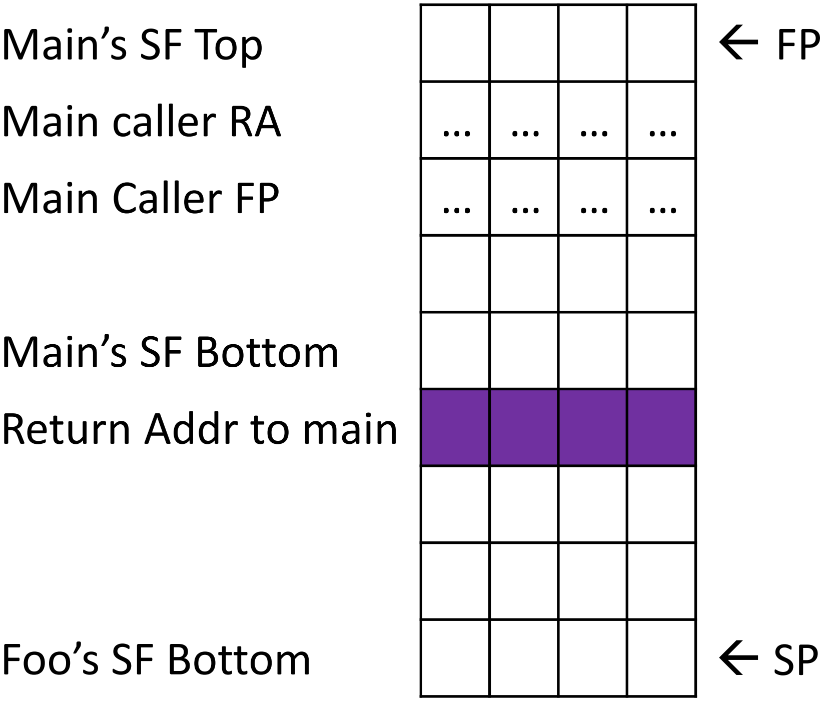
```
foo(int, int):
                addi     sp, sp, -16
        →       sw       ra, 12(sp)
                sw       fp, 8(sp)
                addi     fp, sp, 16
                sw       a0, -12(fp)
                sw       a1, -16(fp)
                li       a0, 5
                lw       ra, 12(sp)
                lw       fp, 8(sp)
                addi     sp, sp, 16
                ret

main:
                addi     sp, sp, -16
                sw       ra, 12(sp)
                sw       fp, 8(sp)
                addi     fp, sp, 16
                li       a0, 3
                li       a1, 7
                call     foo(int, int)
        ra  →   sw       a0, -12(fp)
                li       a0, 0
                lw       ra, 12(sp)
                lw       fp, 8(sp)
                addi     sp, sp, 16
                ret
```

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

Now store a copy of FP….

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```


Main's SF Top

Main caller RA

Main Caller FP

Main's SF Bottom

Return Addr to main

Foo's SF Bottom

…	…	…	…
…	…	…	…
▨	▨	▨	▨

```
foo(int, int):
                addi     sp, sp, -16
                sw       ra, 12(sp)
        →       sw       fp, 8(sp)
                addi     fp, sp, 16
                sw       a0, -12(fp)
                sw       a1, -16(fp)
                li       a0, 5
                lw       ra, 12(sp)
                lw       fp, 8(sp)
                addi     sp, sp, 16
                ret

main:
                addi     sp, sp, -16
                sw       ra, 12(sp)
                sw       fp, 8(sp)
                addi     fp, sp, 16
                li       a0, 3
                li       a1, 7
                call     foo(int, int)
      ra   →    sw       a0, -12(fp)
                li       a0, 0
                lw       ra, 12(sp)
                lw       fp, 8(sp)
                addi     sp, sp, 16
                ret
```

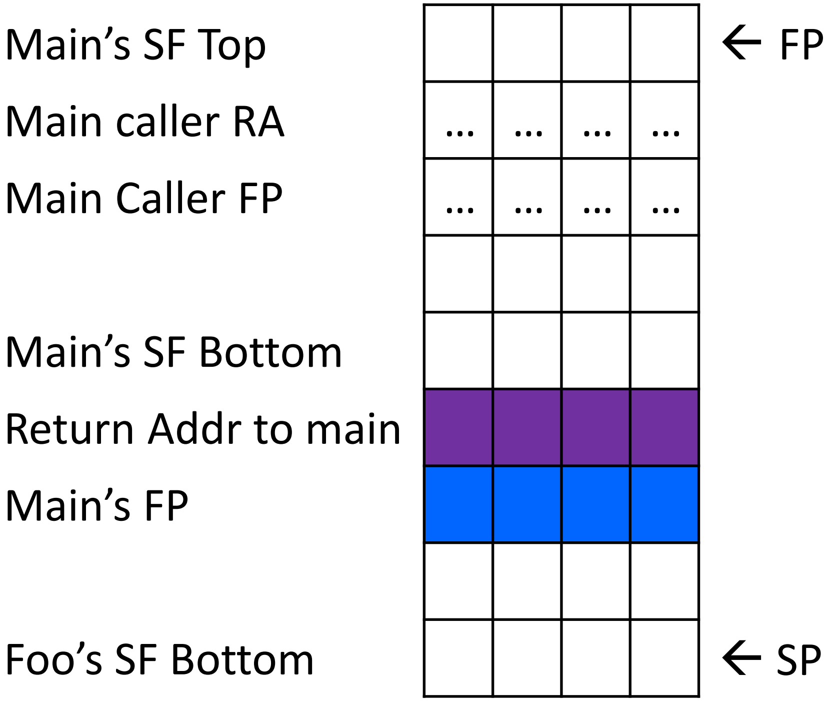
Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```

Now set our new FP. ("top of our frame")

```
foo(int, int):
            addi    sp, sp, -16
            sw      ra, 12(sp)
            sw      fp, 8(sp)
     →      addi    fp, sp, 16
            sw      a0, -12(fp)
            sw      a1, -16(fp)
            li      a0, 5
            lw      ra, 12(sp)
            lw      fp, 8(sp)
            addi    sp, sp, 16
            ret

main:
            addi    sp, sp, -16
            sw      ra, 12(sp)
            sw      fp, 8(sp)
            addi    fp, sp, 16
            li      a0, 3
            li      a1, 7
            call    foo(int, int)
   ra →     sw      a0, -12(fp)
            li      a0, 0
            lw      ra, 12(sp)
            lw      fp, 8(sp)
            addi    sp, sp, 16
            ret
```



Main's SF Top					← FP
Main caller RA	…	…	…	…	
Main Caller FP	…	…	…	…	
Main's SF Bottom					
Return Addr to main	🟪	🟪	🟪	🟪	
Main's FP	🟦	🟦	🟦	🟦	
Foo's SF Bottom					← SP

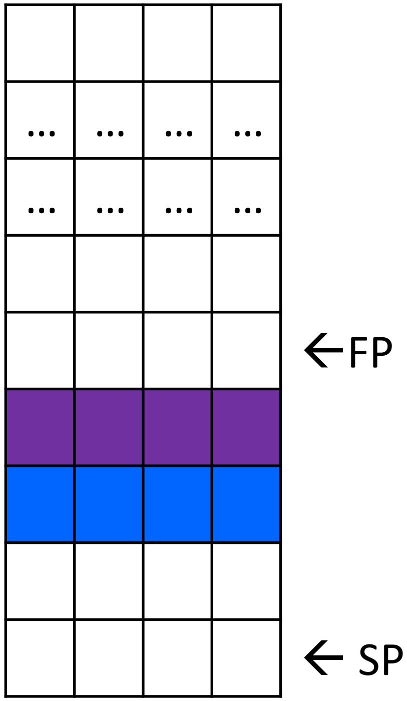
Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

Save copy of a0

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```

```
foo(int, int):
                addi    sp, sp, -16
                sw      ra, 12(sp)
                sw      fp, 8(sp)
                addi    fp, sp, 16
       →        sw      a0, -12(fp)
                sw      a1, -16(fp)
                li      a0, 5
                lw      ra, 12(sp)
                lw      fp, 8(sp)
                addi    sp, sp, 16
                ret

main:
                addi    sp, sp, -16
                sw      ra, 12(sp)
                sw      fp, 8(sp)
                addi    fp, sp, 16
                li      a0, 3
                li      a1, 7
                call    foo(int, int)
   ra  →        sw      a0, -12(fp)
                li      a0, 0
                lw      ra, 12(sp)
                lw      fp, 8(sp)
                addi    sp, sp, 16
                ret
```



Main's SF Top

Main caller RA

Main Caller FP

Main's SF Bottom

Return Addr to main

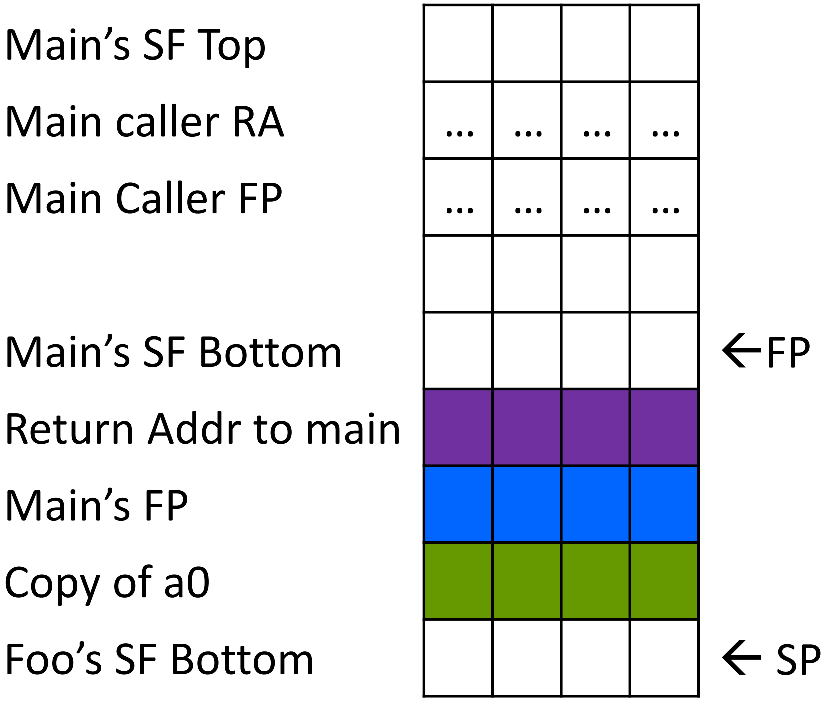
Main's FP

Foo's SF Bottom

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```



Main's SF Top

Main caller RA

Main Caller FP

Main's SF Bottom ←FP

Return Addr to main

Main's FP

Copy of a0

Foo's SF Bottom ← SP

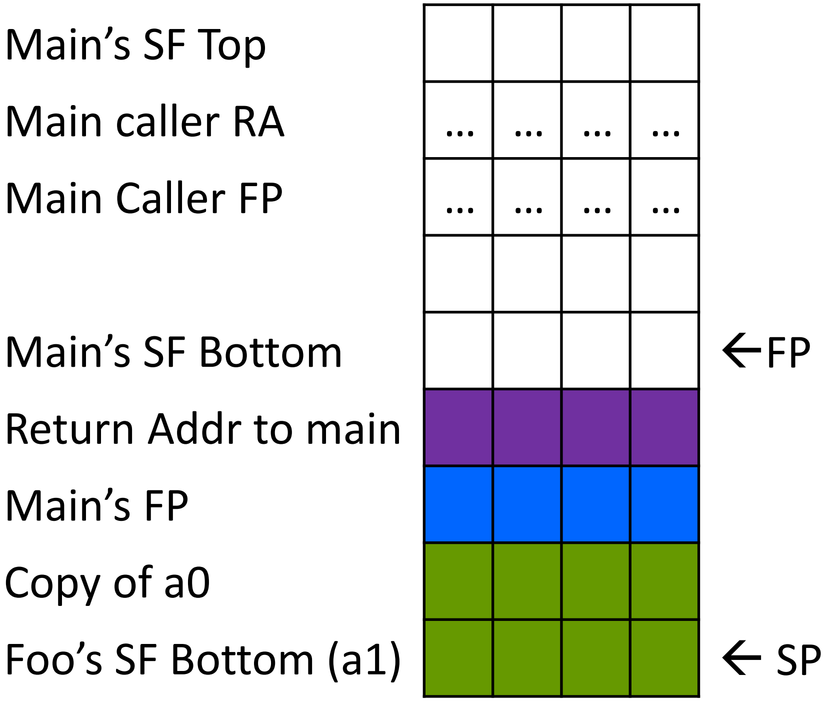
Save copy of a1

```
foo(int, int):
                addi    sp, sp, -16
                sw      ra, 12(sp)
                sw      fp, 8(sp)
                addi    fp, sp, 16
                sw      a0, -12(fp)
        →       sw      a1, -16(fp)
                li      a0, 5
                lw      ra, 12(sp)
                lw      fp, 8(sp)
                addi    sp, sp, 16
                ret

main:
                addi    sp, sp, -16
                sw      ra, 12(sp)
                sw      fp, 8(sp)
                addi    fp, sp, 16
                li      a0, 3
                li      a1, 7
                call    foo(int, int)
       ra   →   sw      a0, -12(fp)
                li      a0, 0
                lw      ra, 12(sp)
                lw      fp, 8(sp)
                addi    sp, sp, 16
                ret
```

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```c
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```



All things saved now…

```
foo(int, int):
              addi    sp, sp, -16
              sw      ra, 12(sp)
              sw      fp, 8(sp)
              addi    fp, sp, 16
              sw      a0, -12(fp)
              sw      a1, -16(fp)
      →       li      a0, 5
              lw      ra, 12(sp)
              lw      fp, 8(sp)
              addi    sp, sp, 16
              ret

main:
              addi    sp, sp, -16
              sw      ra, 12(sp)
              sw      fp, 8(sp)
              addi    fp, sp, 16
              li      a0, 3
              li      a1, 7
              call    foo(int, int)
     ra  →    sw      a0, -12(fp)
              li      a0, 0
              lw      ra, 12(sp)
              lw      fp, 8(sp)
              addi    sp, sp, 16
              ret
```

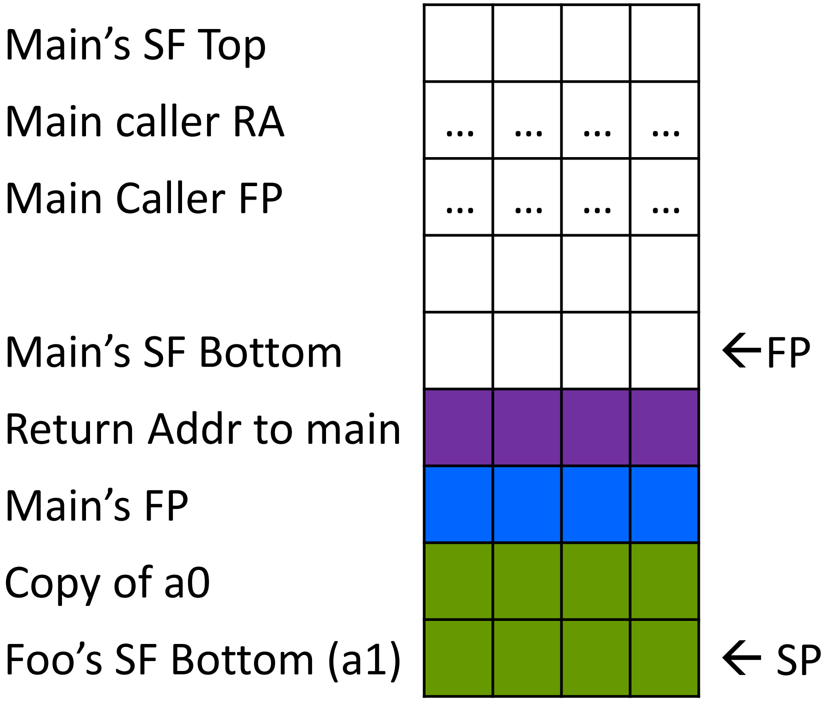
Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

Epilogue done!
we can now execute the function body….

Which just returns 5 in this case…

```
foo(int, int):
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        sw      a0, -12(fp)
        sw      a1, -16(fp)
        li      a0, 5
  →     lw      ra, 12(sp)
        lw      fp, 8(sp)
        addi    sp, sp, 16
        ret

main:
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        li      a0, 3
        li      a1, 7
        call    foo(int, int)
ra  →   sw      a0, -12(fp)
        li      a0, 0
        lw      ra, 12(sp)
        lw      fp, 8(sp)
        addi    sp, sp, 16
        ret
```



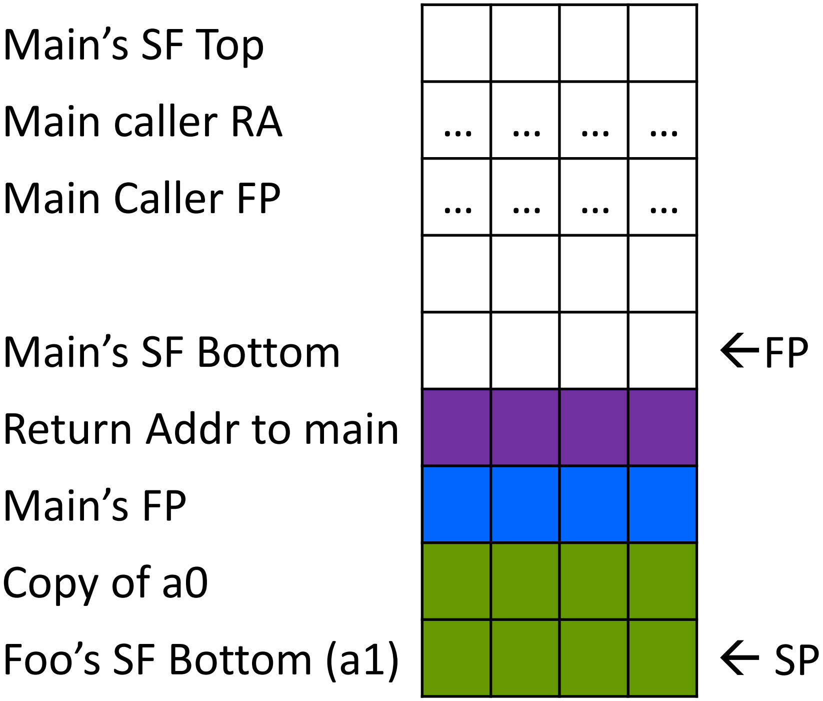
Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

Restore RA so we can go back to main
(RA register was not changed in this function, but we save/restore by default)

```
   return 5;
}

int main() {
   int x = foo(3, 7);
}
```

```
foo(int, int):
              addi    sp, sp, -16
              sw      ra, 12(sp)
              sw      fp, 8(sp)
              addi    fp, sp, 16
              sw      a0, -12(fp)
              sw      a1, -16(fp)
              li      a0, 5
              lw      ra, 12(sp)
      →       lw      fp, 8(sp)
              addi    sp, sp, 16
              ret

main:
              addi    sp, sp, -16
              sw      ra, 12(sp)
              sw      fp, 8(sp)
              addi    fp, sp, 16
              li      a0, 3
              li      a1, 7
              call    foo(int, int)
    ra →      sw      a0, -12(fp)
              li      a0, 0
              lw      ra, 12(sp)
              lw      fp, 8(sp)
              addi    sp, sp, 16
              ret
```



Main's SF Top

Main caller RA … … … …

Main Caller FP … … … …

Main's SF Bottom ←FP

Return Addr to main

Main's FP

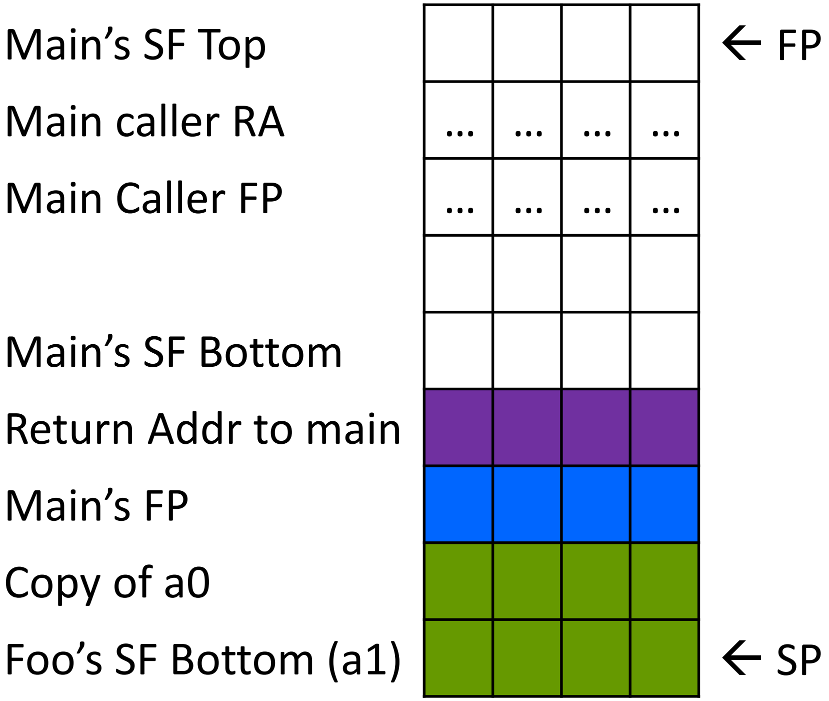
Copy of a0

Foo's SF Bottom (a1) ← SP

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```c
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```



Main's SF Top ← FP

Main caller RA … … … …

Main Caller FP … … … …

Main's SF Bottom

Return Addr to main

Main's FP

Copy of a0

Foo's SF Bottom (a1) ← SP

Restore main's fp…

```
foo(int, int):
                addi    sp, sp, -16
                sw      ra, 12(sp)
                sw      fp, 8(sp)
                addi    fp, sp, 16
                sw      a0, -12(fp)
                sw      a1, -16(fp)
                li      a0, 5
                lw      ra, 12(sp)
                lw      fp, 8(sp)
        →       addi    sp, sp, 16
                ret

main:
                addi    sp, sp, -16
                sw      ra, 12(sp)
                sw      fp, 8(sp)
                addi    fp, sp, 16
                li      a0, 3
                li      a1, 7
                call    foo(int, int)
      ra  →     sw      a0, -12(fp)
                li      a0, 0
                lw      ra, 12(sp)
                lw      fp, 8(sp)
                addi    sp, sp, 16
                ret
```

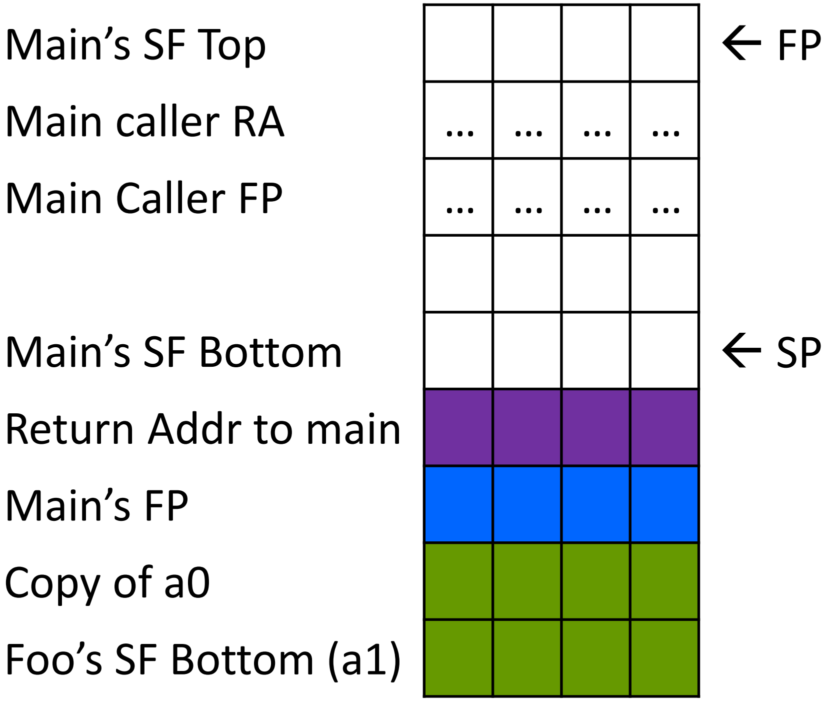
Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

Pop off this stack frame…

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```

```
foo(int, int):
            addi     sp, sp, -16
            sw       ra, 12(sp)
            sw       fp, 8(sp)
            addi     fp, sp, 16
            sw       a0, -12(fp)
            sw       a1, -16(fp)
            li       a0, 5
            lw       ra, 12(sp)
            lw       fp, 8(sp)
            addi     sp, sp, 16
       →    ret

main:
            addi     sp, sp, -16
            sw       ra, 12(sp)
            sw       fp, 8(sp)
            addi     fp, sp, 16
            li       a0, 3
            li       a1, 7
            call     foo(int, int)
   ra  →    sw       a0, -12(fp)
            li       a0, 0
            lw       ra, 12(sp)
            lw       fp, 8(sp)
            addi     sp, sp, 16
            ret
```



Main's SF Top ← FP

Main caller RA … … … …

Main Caller FP … … … …

Main's SF Bottom ← SP

Return Addr to main

Main's FP

Copy of a0

Foo's SF Bottom (a1)

Review: Simple Example:

Red Arrow is PC, the instruction we are about to execute

```
int foo(int input, int x) {
  return 5;
}

int main() {
  int x = foo(3, 7);
}
```

Main's SF Top

Main caller RA

Main Caller FP

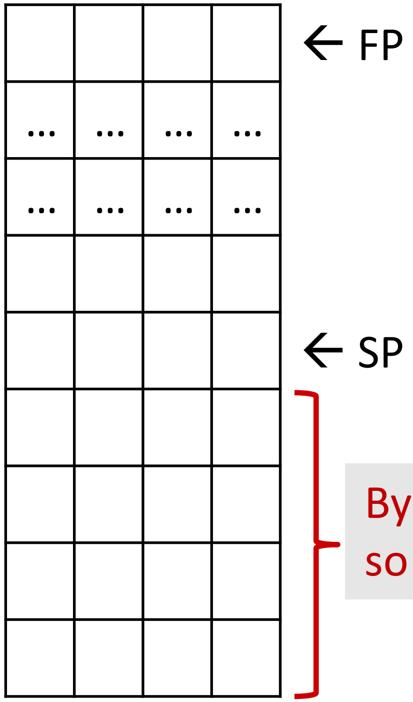
Main's SF Bottom

Return Addr to main

Main's FP

Copy of a0

Foo's SF Bottom (a1)



Returned to main!

```
foo(int, int):
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        sw      a0, -12(fp)
        sw      a1, -16(fp)
        li      a0, 5
        lw      ra, 12(sp)
        lw      fp, 8(sp)
        addi    sp, sp, 16
        ret

main:
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        li      a0, 3
        li      a1, 7
        call    foo(int, int)
  →     sw      a0, -12(fp)
        li      a0, 0
                ra, 12(sp)
                fp, 8(sp)
                sp, sp, 16
        ret
```

Bytes are not zeroed out… so some data may still be here

Poll Everywhere

pollev.com/tqm

- Given this prologue, which is the corresponding epilogue?

```
addi    sp, sp, -16
sw      ra, 12(sp)
sw      fp, 8(sp)
addi    fp, sp, 16
sw      a0, -12(fp)
sw      a1, -16(fp)
```

A.

```
lw      fp, -8(fp)
lw      ra, -4(fp)
addi    sp, sp, 16
ret
```

B.

```
lw      ra, 12(sp)
lw      fp, 8(sp)
addi    sp, sp, 16
ret
```

C.

```
lw      ra, -12(fp)
lw      fp, -8(fp)
addi    sp, sp, 16
ret
```

D.

```
lw      ra, 8(sp)
lw      fp, 12(sp)
addi    sp, sp, 16
ret
```

Poll Everywhere

pollev.com/tqm

- Given this function, how much do you think we decrement the stack pointer in the function’s prologue?
  - Assume integers are 32-bits (4-Bytes)

```c
bool foo(int input, int x) {
  int arr[5];

  // no other local vars…

  return true;
}
```

Poll Everywhere

pollev.com/tqm

- Given this function, how do we allocate the array after the prologue?
  - Assume integers are 32-bits (4-Bytes)

```
bool foo(int len, int x) {
  int arr[len];

  // no other local vars…

  return true;
}
```

```
foo(int, int):
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        sw      a0, -12(fp)
        sw      a1, -16(fp)


# TODO: Body of function.
  How do we allocate arr?


        lw      ra, 12(sp)
        lw      fp, 8(sp)
        addi    sp, sp, 16
        ret
```

Handling Growing Stacks

This may be useful for HW11

- If we have to extend the stack then how much do we add to the stack to pop the stack frame off?
  - What if we only grow the stack on some inputs?

```
foo(int, int):
          addi     sp, sp, -16
          sw       ra, 12(sp)
          sw       fp, 8(sp)
          addi     fp, sp, 16
          sw       a0, -12(fp)
          sw       a1, -16(fp)


         slli     a1, a0, 2
         sub      sp, sp, a1

          # uhhh, sp is not
          # in same spot anymore
          lw       ra, 12(sp)
          lw       fp, 8(sp)
          addi     sp, sp, 16
          ret
```

- Could keep track of how much stack grows
  - But we can do something simpler using FP. What could we do?

Lecture Outline

- C to ASM Functions & Stack
  - Review
  - Growing the stack & Local Variables
- **If statements in ASM**
- **While loops in ASM**

If & Loops in ASM

- Not all programming constructs have direct RISC-V instructions

- How would we implement

```
if (x10 >= 3)
   x11 = x10;
```

Note how we check for the inverse of the condition.
If the condition is NOT met, then skip the next section
by default asm just goes to the next instruction

```
START:
          li  t0, 3
          blt x10, t0, AFTER_IF
          mov x11, x10
AFTER_IF:
          ...
```

Poll Everywhere

pollev.com/tqm

- Is this translation correct?
  - Make sure you understand why

```
if (x10 == x12) {
  t0 = 1;
} else {
  t0 = x10 - x12;
}
```

```
START:
            bne x10, x12, ELSE
            li t0, 1
ELSE:
            sub t0, x10, x12
ENDIF:
            ...
```

If & Loops in ASM

- Not all programming constructs have direct LC4 instructions

- How would we implement

```
for (t0 = 0; t0 < t1; t0++) {
   // ...
}
```

```
        li    t0, 0
START_LOOP:
        bge   t0, t1, AFTER_LOOP
        # ...
        ADD   t0, t0, #1
        J     START_LOOP
AFTER_LOOP:
        # ...
```

Poll Everywhere

pollev.com/tqm

- Is this translation correct?
  - Make sure you understand why

```
while (x10 != x12)
  x10++;
}
```

```
WHILE:
          bne x10, x12, AFTER
          addi x10, x10, 1
          j WHILE
AFTER:
          ...
```

Poll Everywhere

pollev.com/tqm

- Is this translation correct?
  - Make sure you understand why

```
if (x10 == x12) {
  t0 = 1;
} else {
  if (t1 != 0){
    x10 += t1
  }
  t0 = x10 - x12;
}
```

```
START:
              bne  x10, x12, ELSE
              mov t0, x1
              J AFTER_IF
ELSE:
              beq t1, x0, ELSE
              add x10, x10, t1
ELSE:
              sub t0, x10, x12
AFTER_IF:
              ...
```

Unique Labels

- As part of HW11 you will need to generate assembly, this assembly can contain multiple if/else/endifs, while loops, etc.
  - Labels must be unique in assembly!
  - How can we enforce uniqueness?
    - Just number the labels:
      - IF1, ELSE1, ENDIF1
      - IF2, ELSE2, ENDIF2
      - Etc

- How do we handle nested structures? We need to keep track of ELSE we need to jump to at what time.
  - You will need recursion or a LIFO data structure (like a deque or stack) to do this.