Compiler Optimizations

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

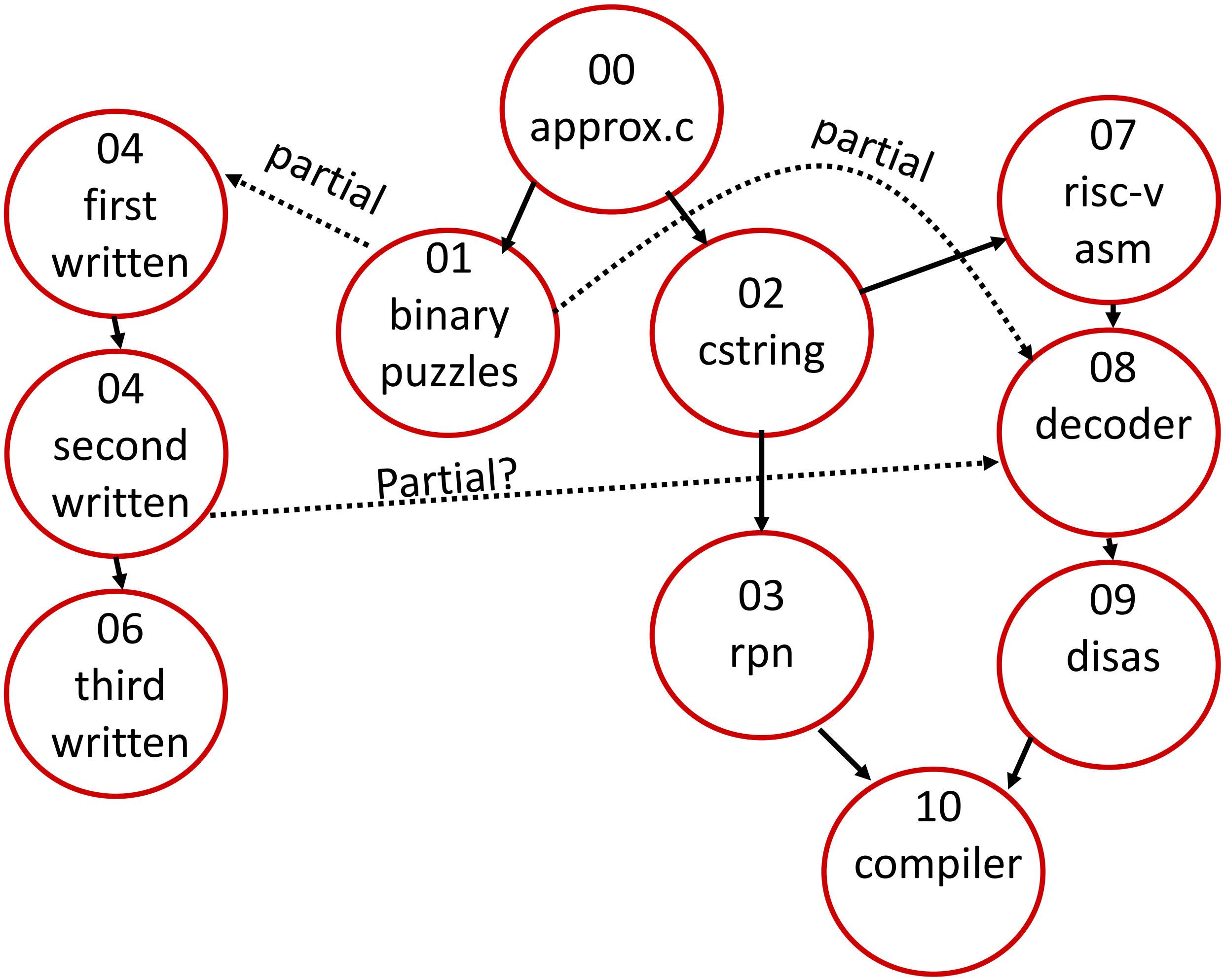
- Any Questions?

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 16^{th} @ 9am
- Check-in out tomorrow

HW DAG

- We suggest you doing HW's sequentially, but if you need to put one off till later, it is not the end of the world.



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

- Actually happening this lecture
- We updated **`penn-sim`** and **`as`**
- You will need to redownload the penn-sim.zip
  - If a file doesn't have executable permissions try doing something like:
    - chmod +x ./penn-sim
    - chmod +x ./as

Poll Everywhere

pollev.com/tqm

- Any Questions?

Lecture Outline

- **ASM Call Stack**
  - **Why all the loads and stores?**
- Optimizations
  - Role of the compiler
  - Optimized prologue / epilogue
  - Loop unrolling
  - Register Allocation
  - Linker Relaxation
  - General Optimizations

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					← SP

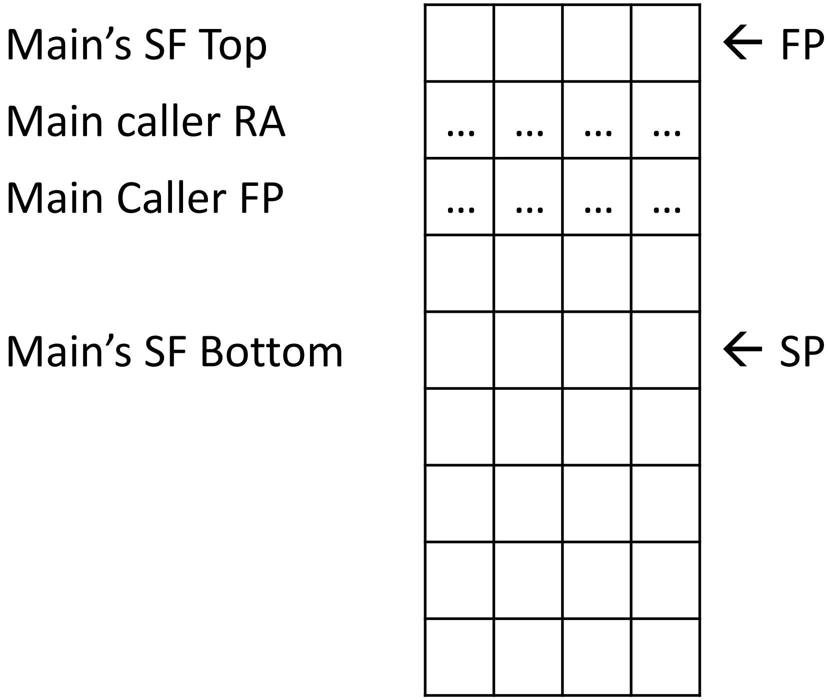
Call foo…

```
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);
}
```



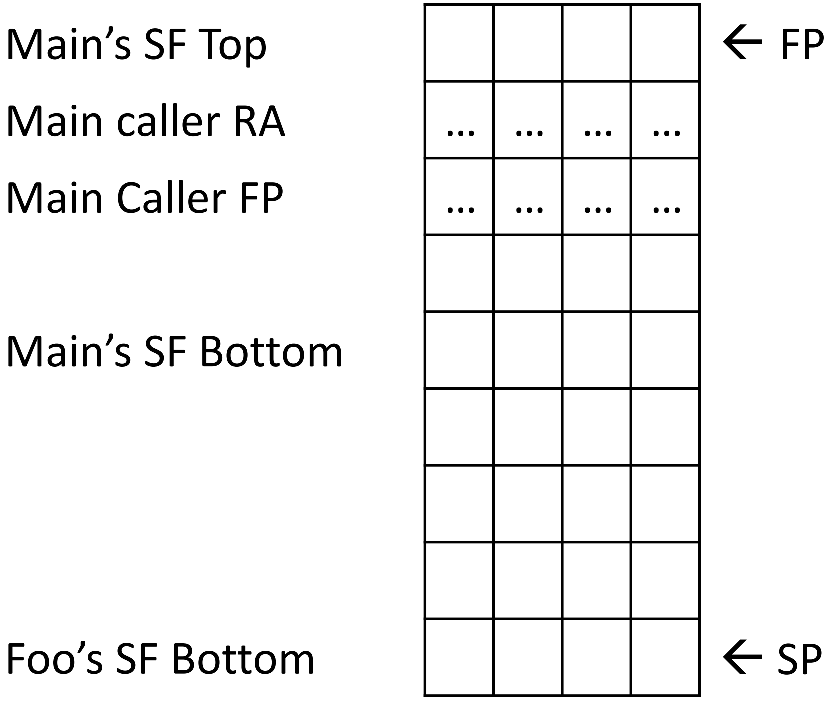
In foo, need to allocate stack frame

```
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
```

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);
}
```



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….

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

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

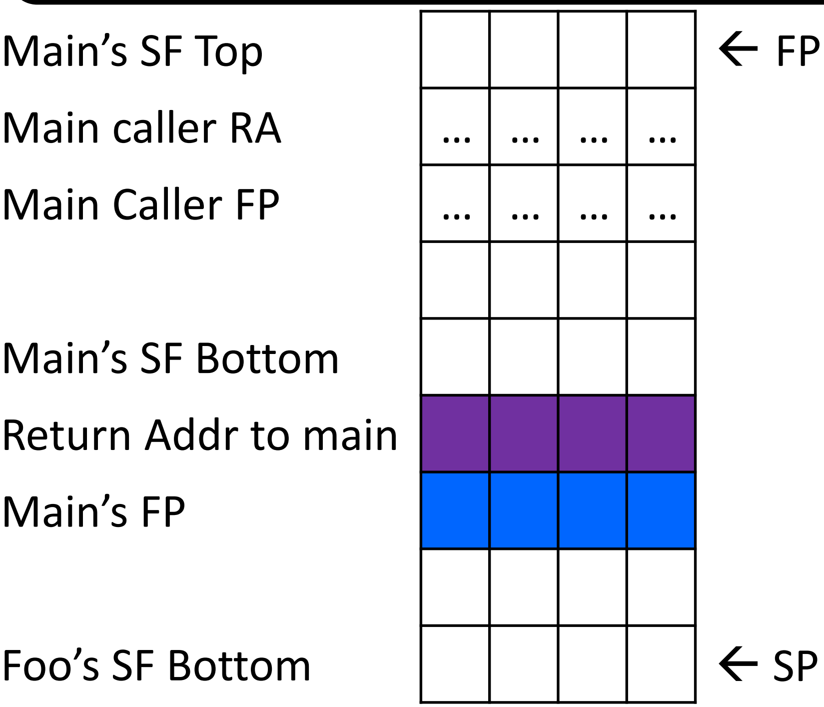
Main's SF Top					← FP
Main caller RA	…	…	…	…	
Main Caller FP	…	…	…	…	
Main's SF Bottom					
Return Addr to main	(shaded)	(shaded)	(shaded)	(shaded)	
Foo's SF Bottom					← SP

```
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);
}
```



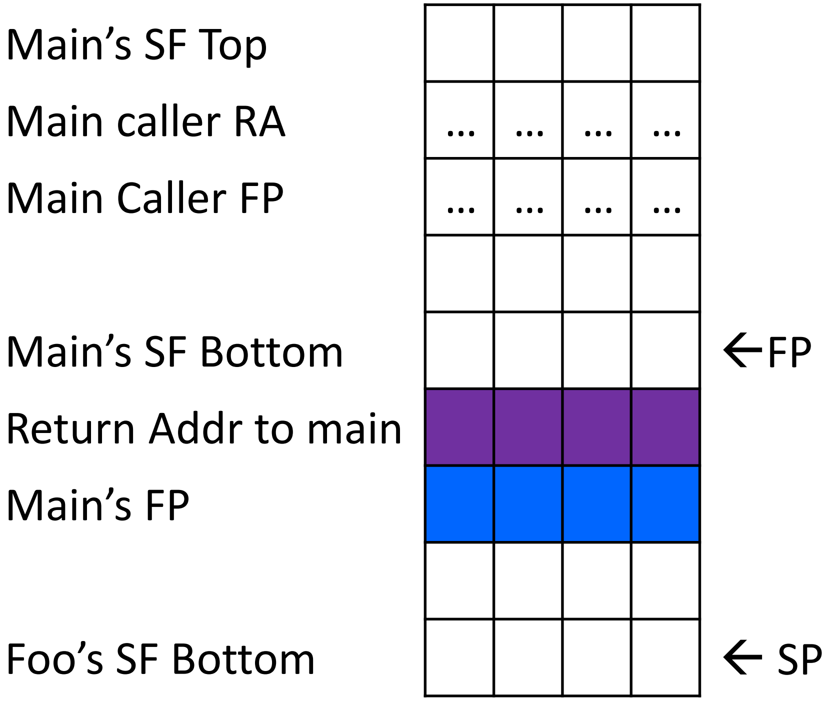
Now set our new FP. ("top of our frame")

```
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
```

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);
}
```



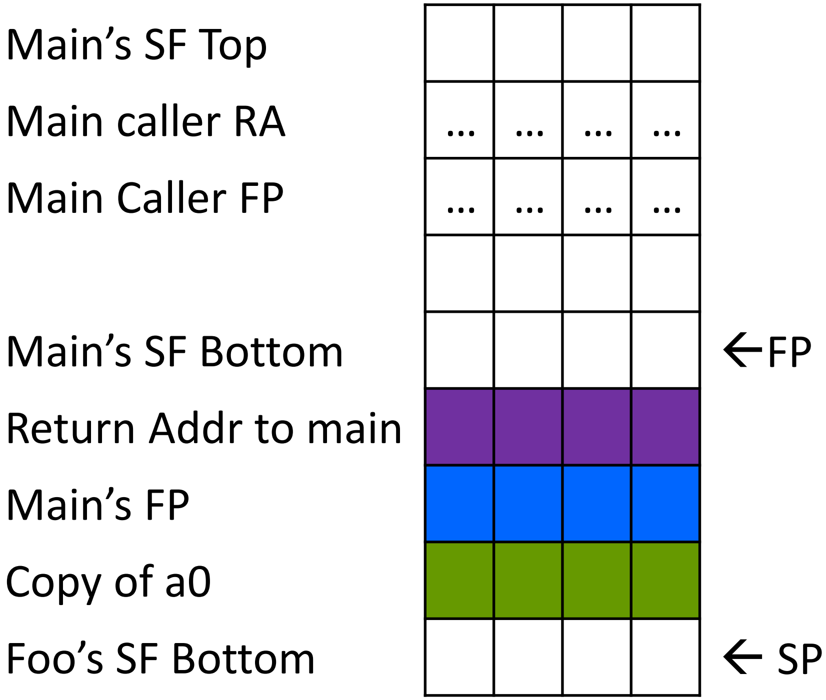
Save copy of a0

```
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);
}
```



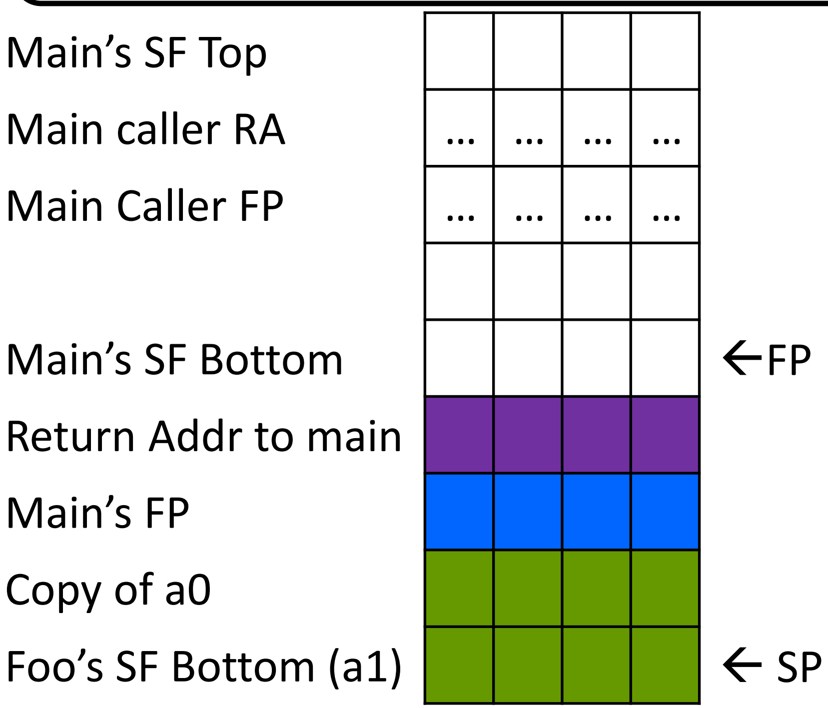
Save copy of a1

```
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

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

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



All things saved now…

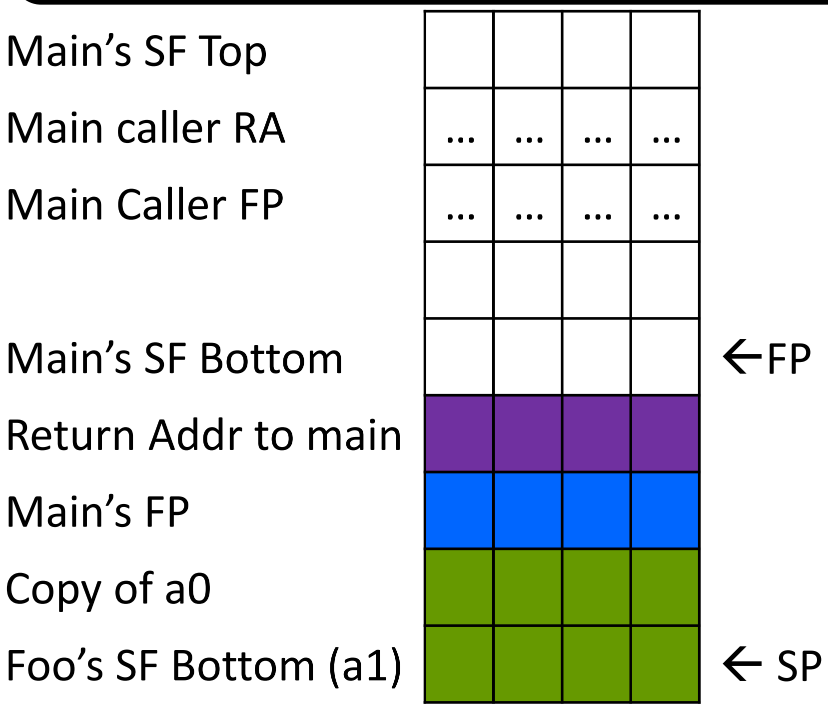
```
foo(int, int):
             addi    sp, sp, -16
             sw      ra, 12(sp)
             sw      fp, 8(sp)
             addi    fp, sp, 16
             sw      a0, -12(fp)
             sw      a1, -16(fp)
    →        li      a0, 5
             lw      ra, 12(sp)
             lw      fp, 8(sp)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



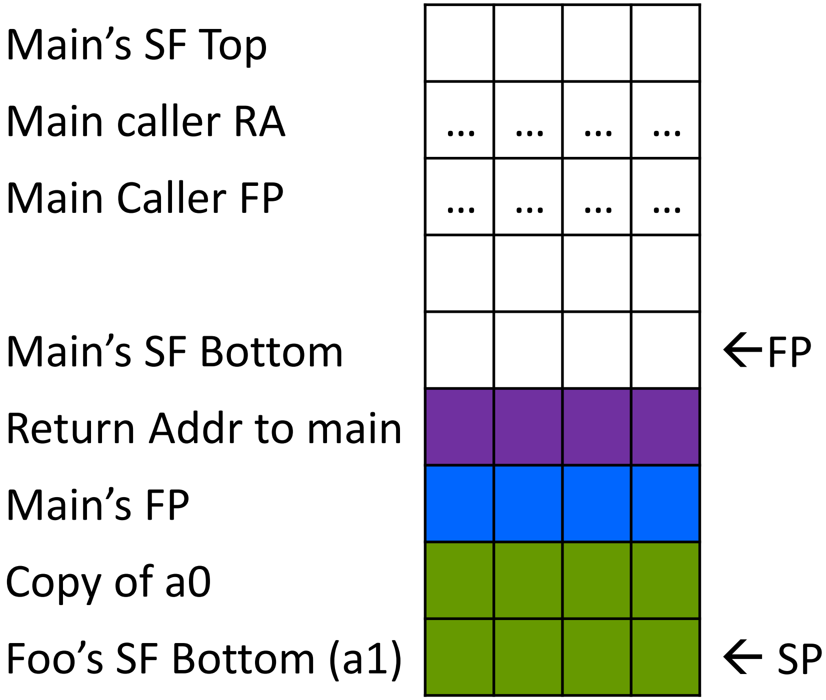

```
foo(int, int):
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        sw      a0, -12(fp)
        sw      a1, -16(fp)
        li      a0, 5
  →     lw      ra, 12(sp)
        lw      fp, 8(sp)
        addi    sp, sp, 16
        ret

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)

```c
  return 5;
}

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

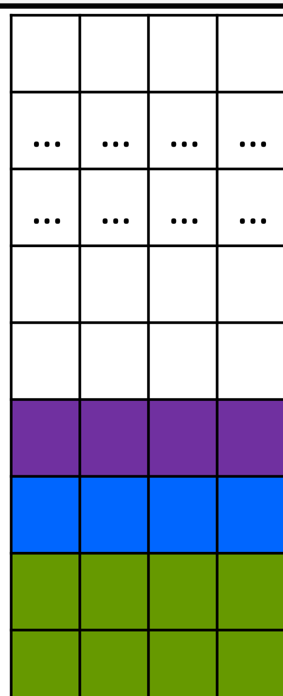



```
foo(int, int):
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      fp, 8(sp)
        addi    fp, sp, 16
        sw      a0, -12(fp)
        sw      a1, -16(fp)
        li      a0, 5
        lw      ra, 12(sp)
  →     lw      fp, 8(sp)
        addi    sp, sp, 16
        ret

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
```

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);
}
```



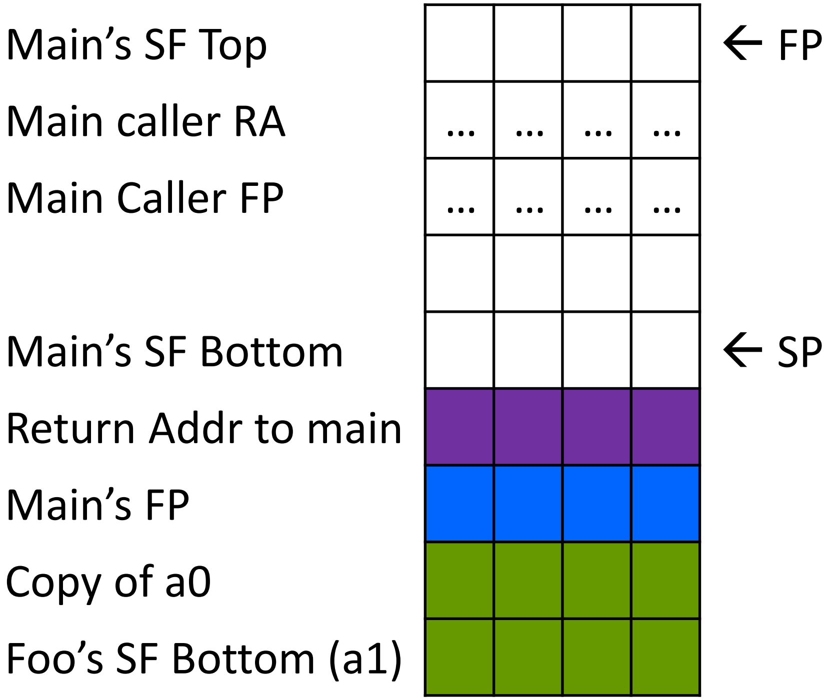
Restore main's fp…

```
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

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

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



Pop off this 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
```

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);
}
```

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
```

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)					

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

Why all the memory accesses?

- You may have noticed that we stored many values on the stack that go either unchanged or unused.
  - Return address
  - Frame Pointer
  - Arguments

- Some of these values are stored and immediately re-loaded? Why????????

- By default, the compiler will not look at everything the function does. It saves register values on the stack JUST IN CASE they are needed and to avoid needed values being overwritten

Poll Everywhere

pollev.com/tqm

- Consider this code:
  (We don't know what the function **`baz()`** does)

```
int baz(int x);

int foo(int input, int x) {
  input = baz(x)
  return x + input;
}
```

- Below are the values that would normally be saved by the function **`foo()`** in it's prologue. Which of the following values actually **NEED** to be saved by **`foo()`**?
  - Callers' frame pointer (fp)
  - Return address to foo's caller (ra)
  - input (a0)
  - x (a1)

Lecture Outline

- ASM Call Stack
  - Why all the loads and stores?
- **Optimizations**
  - **Role of the compiler**
  - **Optimized prologue / epilogue**
  - **Loop unrolling**
  - **Register Allocation**
  - **Linker Relaxation**
  - **General Optimizations**

Compiler Optimizations

- Q: Does your computer execute the program you wrote?

True	False
☐	☒

  - Stole this from one of Herb Sutter's Conference talks :P

Compiler Optimizations

- Compiler (or processor or cache) may say:
  - "No, it's much better to execute a different program. Hey, don't complain. It's for your won good. You really wouldn't want to execute that *dreck* you actually wrote."

- Lots of things may happen between your source code and actual execution. We will only look at the compiler today

- All transformations made should result in the equivalent thing happening (this is less clear when we start to use threads, which is outside this course.)

Compiler Optimizations: Rule of Thumb

- Less Instructions == Faster Code ☺
  - This isn't always true: We simplify when we say all instructions take the same amount of time.
  - We will say 1 instruction == 1 clock cycle

Optimizing the Prologue / Epilouge

- As we saw in previous lectures, sometimes the compiler can determine not everything in the prologue / epilogue is needed.

- If we can remove those instructions, our code is faster ☺

Optimization Questions

- Some practice problems to see if you can apply the stuff we have learned.

Poll Everywhere

pollev.com/tqm

- Which piece of code is likely faster?

```
int fact(int N){
  int res = 1;
  for (int i = 1; i <= N; i++)
    res *= N;
  }
  return res;
}
```

```
int fact(int N){
  if (N <= 2)  {
    return N
  }
  return N * fact(N - 1);
}
```

Poll Everywhere

pollev.com/tqm

- Which piece of code is likely faster?
  - Assume optimizations are turned off for this.

```
int increment(int N){
  return N + 1
}

int main() {
  int x = increment(5);
}
```

```
#define INCREMENT(N) ((N) + 1)

int main() {
  int x = INCREMENT(5);
}
```

Recursion

- When we call a function then we have to run ~8 or so instructions across the prologue and epilogue to that function

- Usually a loop is less overhead, so a loop is usually faster than recursion

- Tail recursion helps with this in some programming languages

Macros

- Macros avoid the overhead of calling functions, so can be quicker.

- Sometimes the compiler may “inline” a function for you to get the same performance boost.

Poll Everywhere

pollev.com/tqm

- Which piece of code is likely faster?
  - Assume optimizations are turned off for this.

```
for (int i = 0; i < 3; i++) {
  printf("Hello!\n");
}
```

```
printf("Hello!\n");
printf("Hello!\n");
printf("Hello!\n");
```

Poll Everywhere

pollev.com/tqm

- How could we optimize this?

```
int x = 5;
for (int i = 0; i < 5000; i++) {
  x += 1;
}
```

Poll Everywhere

pollev.com/tqm

- How could we optimize this?

```
int foo(int x) {
    x = x + 2 + 7;
    x = 42 * 5 * 6;
    return x;
}
```

General Optimizations

- Some code is redundant so may get removed
- In the previous example, the first line (x = x + 2 + 7) stores a value in X that is immediately overwritten. So why even do the addition?
- Some operations on constants can be done by the compiler and result loaded immediately to save time during execution.

```
int foo(int x) {
    x = x + 2 + 7;
    x = 42 * 5 * 6;
    return x;
}
```

Loop Unrolling

- If we know how many times a loop will execute, the compiler may “unroll” your loop (e.g just “paste” the loop body X times) to avoid the overhead of maintaining & checking the loop variables.

Register Allocation

- To avoid constantly storing values in memory, some variables may be held within a register for a prolonged period of time.

- If we maintain the variable in a register, then we don't have to SW and LW memory as often to access the value.

Linker Relaxation

- The call pseudo instruction is usually JAL or JALR + AUIPC.

- Why JALR and AUIPC together?
  - JALR can add a 12-bit immediate to a register and set PC to it
  - AUIPC can set the upper 20 buts of a register
  - Together these allow us to add any 32-bit number to PC to go to the function we want.

  - JAL only adds a 20-bit immediate to the PC. The immediate may not be big enough to get to the instruction we want to execute.

- At some point in the compilation process, a JALR + AUIPC may be replaced by a JAL if it would work.

Ideology:

- It is normal for a lot of focus is put on how to make code faster.

- However faster != better

Readability & Accessibility

- What does this code do?

```
const int main[] = {
    -443987883, 440, 113408, -1922629632,
    4149, 899584, 84869120, 15544,
    266023168, 1818576901, 1461743468, 1684828783,
    -1017312735
};
```

- https://jroweboy.github.io/c/asm/2015/01/26/when-is-main-not-a-function.html

Readability & Accessibility

- What does this code do?

```
int main() {
  write(STDOUT_FILENO, "Hello World!\n", 13);
}
```

- https://jroweboy.github.io/c/asm/2015/01/26/when-is-main-not-a-function.html

Readability:

- General rule of thumb:
  - Quick code is not necessarily unreadable
  - Prioritize readability until speed is an issue
    - Your code may still be in use 10 years later and some unfortunate person has to go back and fix/update it
    - That unfortunate person may be you!