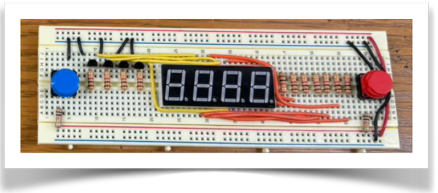
Admin

Lab 2

Please submit exit form!

Assign 2

Extension encouraged!



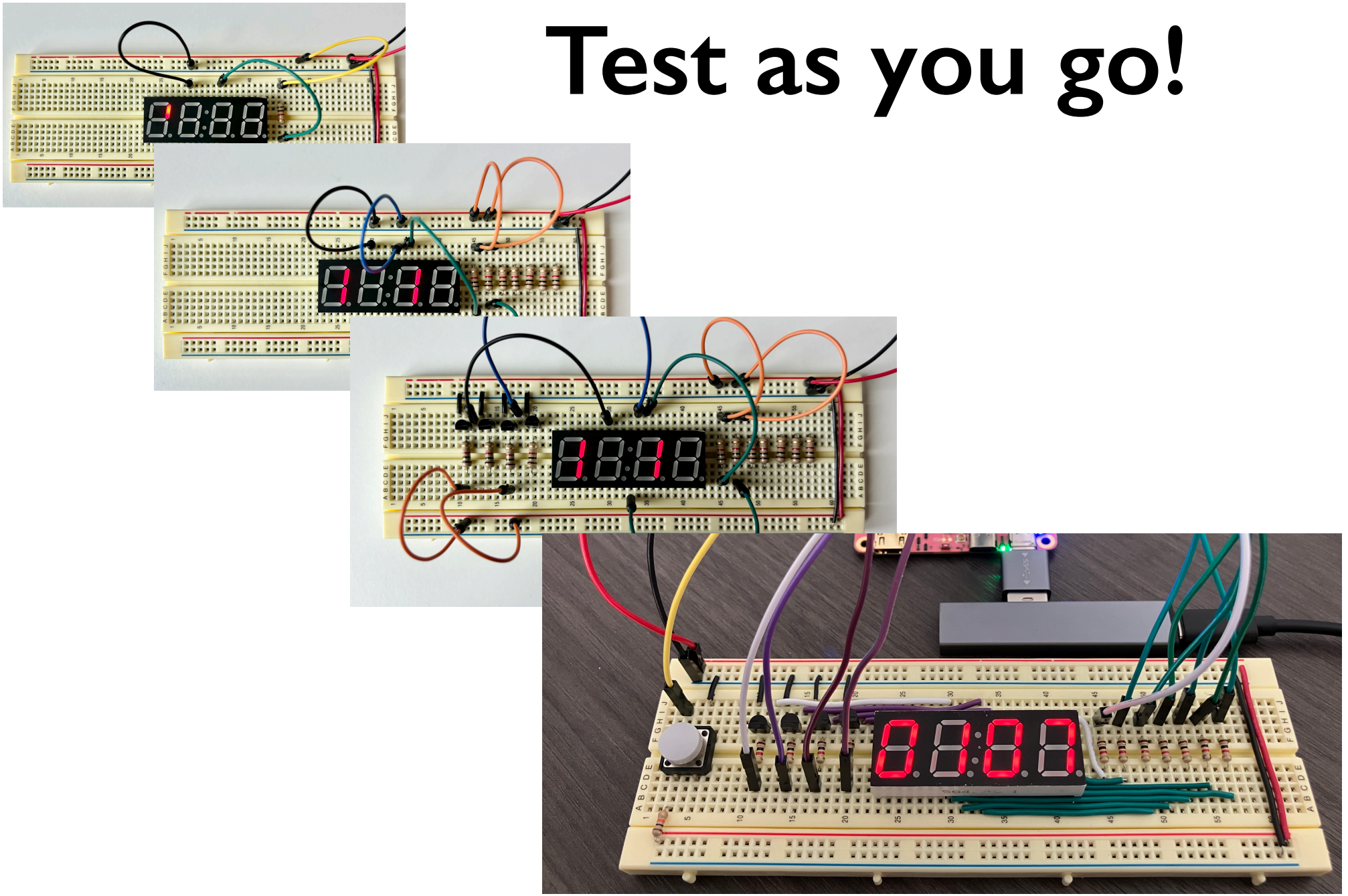
Today: C functions

Strategies for testing

Implementation of C function calls

Management of runtime stack, register use

Test as you go!



Biggest Rookie Mistake

The worse thing to do is to write a lot of code without testing it.

It will almost certainly not work.

Trust me. I've been writing code that doesn't work for 30 years. 🙃

Test-Driven Development

How to begin? **Bottom-up**

- identify simplest testable function, no depend on anything else
- write test cases to verify function behavior
- implement function and test
- only move on to next when all tests pass

What function to tackle next?

- another independent function, or one that builds on tested code
- rinse and repeat

Take baby steps from a known working state to a new working state.
When add new code, rerun all old tests. It is easy to accidentally break something that used to work. ("regression")

Tests are your friend!

Read function interface/documentation, this is "contract"

Write assertions to confirm a function meets specification. Assertions will clarify your understanding of how the function should work.

Part of the skill of software development is anticipating what can go wrong. Consider both desired outcomes as well as ensure no unwanted/wrong behavior.

Our starter code gives just a few tests, you will extend to be comprehensive!

Trick from Don Knuth: Develop "torture tests." The border cases, the bizarre cases. What would some maniacal person try to do to trip you up!?

From C to Assembly (cont'd)

Previously

- C variable ⇒ registers
- C arithmetic/logical expression ⇒ ALU instructions
- C control flow ⇒ branch instructions
- C pointer ⇒ memory address
- Read/write memory ⇒ load/store instructions
- Array/struct data layout ⇒ address arithmetic

Today

- Assembly implementation of function call/return, parameters, return value
- Conventions on register use, ABI
- Runtime stack, local variables

Excerpted from blink.s

```
loop:

    sw      a1,0x40(a0)
    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay

    sw      zero,0x40(a0)
    lui     a2,0x3f00
delay2:
    addi    a2,a2,-1
    bne     a2,zero,delay2

    j       loop
```

Repeated code,
would be nice to unify…

```
loop:

    sw    a1,0x40(a0)
    j     pause

    sw    zero,0x40(a0)
    j     pause

    j     loop
```

```
pause:
    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay

// but... where to go now?
```

```
loop:
    sw    a1,0x40(a0)
    jal   ra,pause

    sw    zero,0x40(a0)
    jal   ra,pause

    j     loop
```

Need to remember where we came from, so we can go back...

```
pause:
    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay
    jr      ra
```

```
loop:
    sw    a1,0x40(a0)
    lui   a2,0x3f00
    jal   ra,pause

    sw    zero,0x40(a0)
    lui   a2,0x3f00
    jal   ra,pause

    j     loop
```

How to communicate arguments to function?

```
pause:
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay
    jr      ra
```

Add'l RISC-V instructions

Jump and Link `jal`

Saves pc+4 into rd before jump to target (imm pc-relative offset)

```
jal rd,imm                  // rd = pc+4, pc = pc+imm
```

Jump and Link Register `jalr`

Saves pc+4 into rd before jump to target (register + offset)

```
jalr rd,imm(rs1)            // rd = pc+4, pc = rs1+imm
```

Add Upper Immediate to PC `auipc`

```
auipc   rd,imm              // rd = pc+imm<<12
```

Pseudo-instructions

```
call fn     ->    jal ra,fn
jr rs1      ->    jalr zero,0(rs1)
ret         ->    jalr zero,0(ra)
```

Anatomy of C function call

```
int factorial(int n)
{
   int result = 1;
   for (int i = n; i > 1; i--)
      result *= i;
   return result;
}
```

Call and return

Pass arguments

Local variables

Return value

Scratch/work space

Complication: nested function calls, recursion

Application binary interface

ABI specifies how code interoperates:

- Mechanism for call/return
- How parameters passed
- How return value communicated
- Use of registers (ownership/preservation)
- Stack management (up/down, alignment)

Mechanics of call/return

Caller stores up to 8 arguments in a0 - a7

call (jal) saves pc+4 to ra and jump to target

```
li a0,100
li a1,7
call fn
```

```
sum(100, 7);
```

Callee stores return value in a0

ret (jalr) jumps back to ra

```
add a0,a0,a1
ret
```

```
int sum(int a, int b) {
    return a + b;
}
```

Caller and Callee

caller: function doing the calling

callee: function being called

```
void main(void) {
   gpio_init();
   timer_init();
   config_clock();
   clock_run();
   ...
}

void config_clock(void) {
    gpio_set_output(…);
    ...
}

void gpio_set_output(gpio_id_t) {
  ...
```

`main` is caller of **`config_clock`**

`config_clock` is callee of **`main`**

`config_clock` is caller of **`gpio_set_output`**

`gpio_set_output` is callee of **`config_clock`**

Register ownership

`a0-a7, t0-t6` are **callee-owned** registers ("args", "temp")

- **Callee** can freely use/modify these registers
- **Caller** cedes to callee, has no expectation of register contents after call

`s0-s11` are **caller-owned** registers ("saved")

- **Caller** retains ownership, expects register contents to be same after call as it was before call
- **Callee** cannot use/modify these registers unless takes steps to preserve/restore values

To discuss…

1. If callee needs scratch space for an intermediate result, which type of register should it choose?

2. Why might a callee need to use a caller-owned register? What does callee have to do if using one?

3. What is the advantage in having some registers callee-owned and others caller-owned? Wouldn't it be simpler if all treated the same?

The stack to the rescue

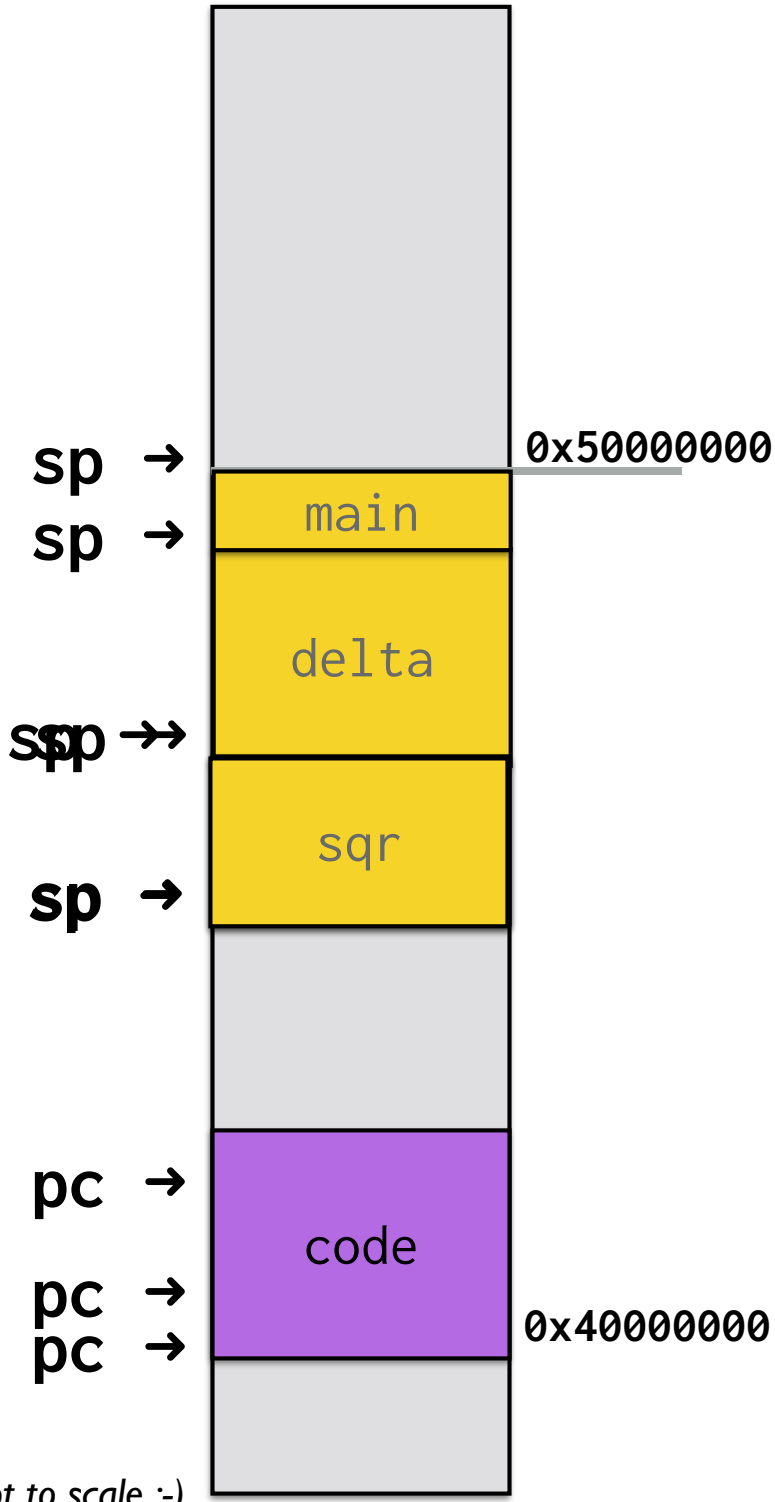
Reserve section of memory to store data for executing function

Stack frame allocated per function invocation
Can store local variables, scratch values, saved registers

- **sp** points to lastmost value pushed
- stack grows down
  - Decrement **sp** at function entry makes space for stack frame ("push")
  - Access frame variables using **sp**-relative offset
  - Increment **sp** at function exit to clean up frame ("pop")
- Call stack is LIFO, last frame pushed is first frame popped

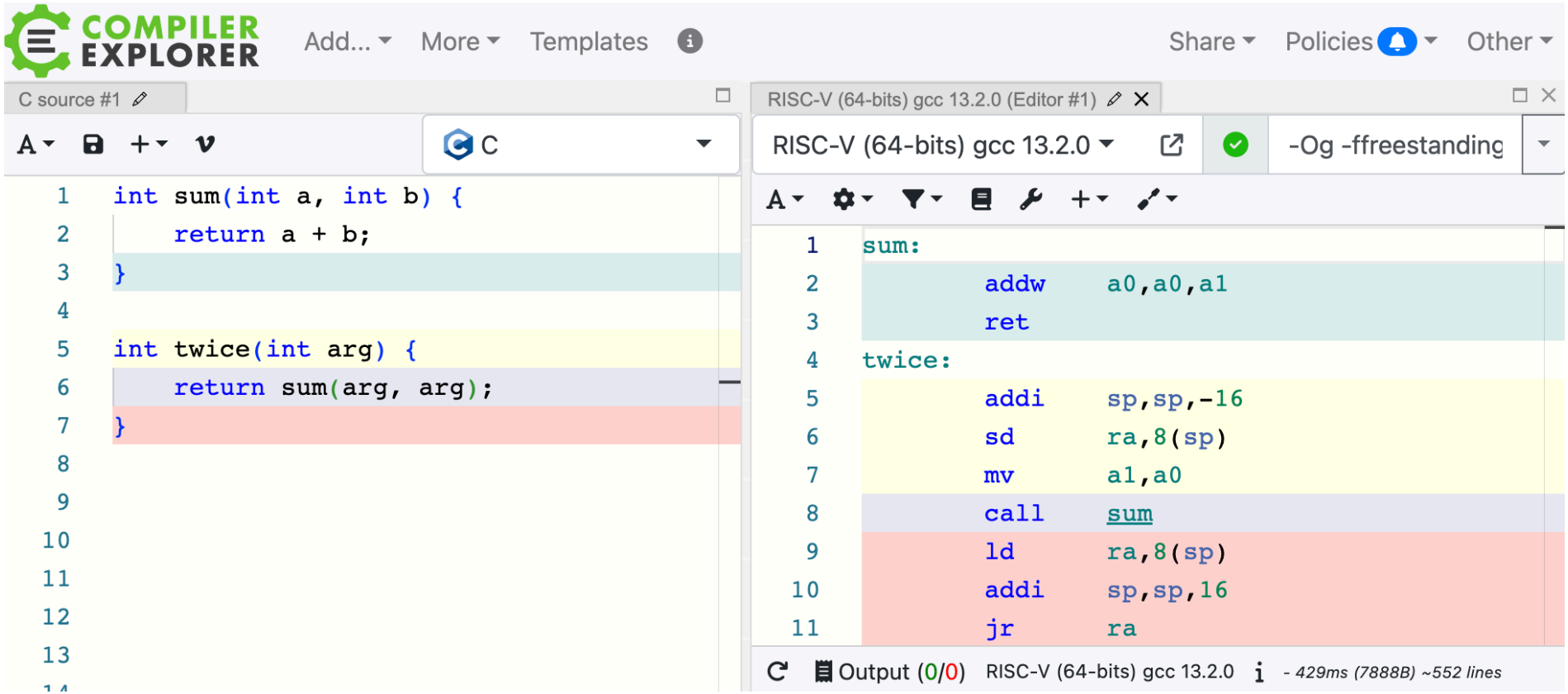
```
// start.s
lui   sp,0x5000
jal   ra,main
```

```
void main(void)
{
   delta(3, 7);
}

int delta(int a, int b)
{
  int diff = sqr(a) - sqr(b);
  return diff;
}

int sqr(int v)
{
  return v * v;
}
```



Memory diagram not to scale :-)

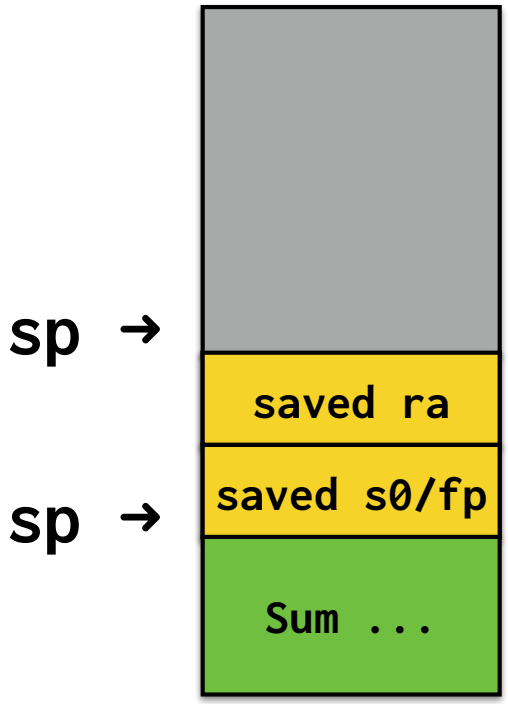
Compiler Explorer!



https://godbolt.org/

Stack operation

```
twice:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        addi    s0,sp,16
        mv      a1,a0
        call    sum
        ld      ra,8(sp)
        ld      s0,0(sp)
        addi    sp,sp,16
        jr      ra
```




```
int twice(int arg) {
    return sum(arg, arg);
}
```

C vs Assembly Smackdown

Why C?

Variable names, type system

Function decomposition, control flow

Portable abstractions

Consistent semantics

Compiler back-end doing heavy lifting - yay!

Why assembly?

Execution is always in asm, this is the real deal -- WYSIWYG

Ability to drop down and review/debug asm is key

Certain hardware features only accessible via asm

Hand-code in asm for optimization or obtain precise timing

Abstraction FTW!

good -> better -> best!

```
    lui     a0,0x2000
    addi    a1,zero,1
    sw      a1,0x30(a0)

loop:
    xori    a1,a1,1
    sw      a1,0x40(a0)

    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay

    j       loop
```

```
void pause(int count) {
    while (count-- != 0) {}
}

void main(void) {
    volatile unsigned int *PB_CFG0 = 0x2000030;
    volatile unsigned int *PB_DATA = 0x2000040;

    *PB_CFG0 = 1;   // config as output

    while (1) {
        *PB_DATA = 1;   // on
        pause(0x3f00000);
        *PB_DATA = 0;   // off
        pause(0x3f00000);
    }
}
```

```
void main(void) {
    timer_init();
    gpio_init();
    gpio_set_output(pin);

    while (1) {
        gpio_write(pin, 1);
        timer_delay_ms(SECOND);
        gpio_write(pin, 0);
        timer_delay_ms(SECOND);
    }
}
```