# Admin

*Assign 1*

A medal for your bare-metal mettle

Grading in-process, results publish to github

*Lab 2*

Breadboarding extraordinaire

*Assign 2*

Extension encouraged!

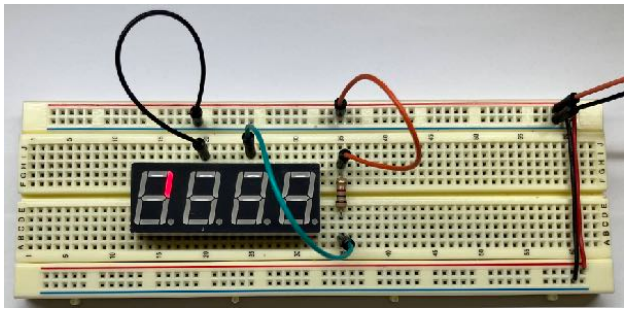
See me for input device

# Today: C functions

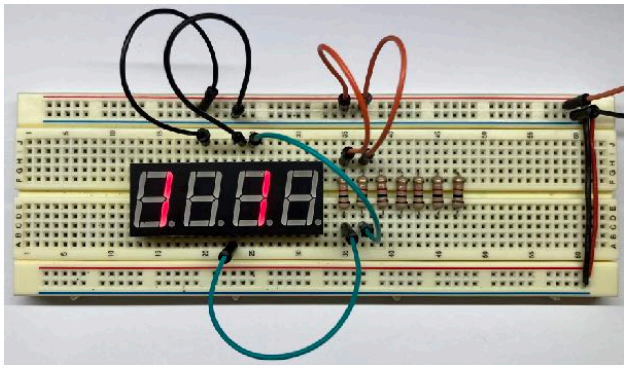
Strategies for testing

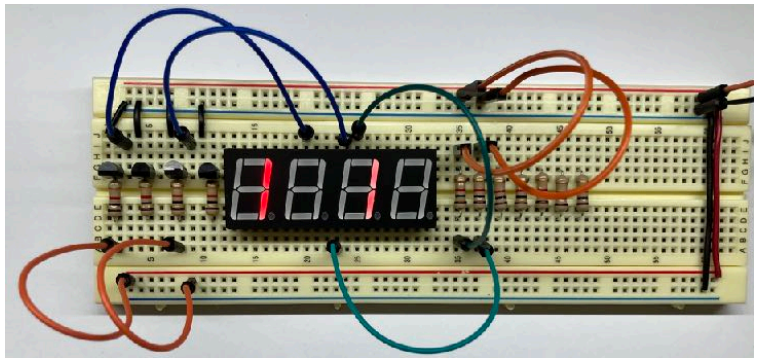
Implementation of C function calls

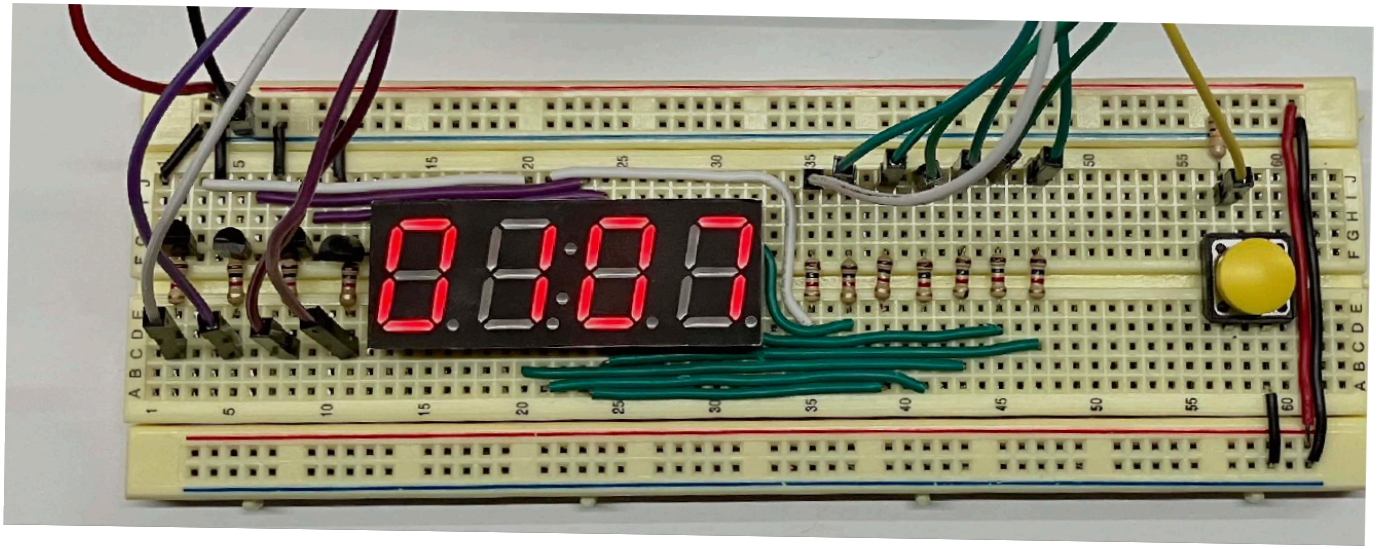
Coordinate of register use, runtime stack



# 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 feature, isolated, no dependencies
- write test cases first, run tests, all fail (as expected)
- implement function and test again, if not pass, investigate & fix
- iterate until all tests pass

What next?

- choose another independent function, or one that layers on already tested code
- rinse and repeat on testing cycle

Take baby steps from a known working state to a new working state. When add new code, always rerun all old tests. This detects if broke something that used to work. ("regression")

# Tests are your friend!

Function interface documents the "contract"

Write assertions to confirm function meets specification. Assertions will clarify your understanding of how the function is supposed to operate.

Consider what can go wrong. Confirm desired outcomes as well as ensure no unwanted/incorrect behavior.

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

*Tip from Don Knuth: Develop "torture tests." The border cases, the bizarre cases. What could a maniacal person do to try to trip up your code…?*

# 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

```
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
```

*How to unify repeated code? …*

```
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?
```

# RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

## Arithmetic Operation

| Mnemonic | Instruction | Type | Description |
|---|---|---|---|
| ADD rd, rs1, rs2 | Add | R | rd ← rs1 + rs2 |
| SUB rd, rs1, rs2 | Subtract | R | rd ← rs1 - rs2 |
| ADDI rd, rs1, imm12 | Add immediate | I | rd ← rs1 + imm12 |
| SLT rd, rs1, rs2 | Set less than | R | rd ← rs1 < rs2 ? 1 : 0 |
| SLTI rd, rs1, imm12 | Set less than immediate | I | rd ← rs1 < imm12 ? 1 : 0 |
| SLTU rd, rs1, rs2 | Set less than unsigned | R | rd ← rs1 < rs2 ? 1 : 0 |
| SLTIU rd, rs1, imm12 | Set less than immediate unsigned | I | rd ← rs1 < imm12 ? 1 : 0 |
| LUI rd, imm20 | Load upper immediate | U | rd ← imm20 << 12 |
| AUIP rd, imm20 | Add upper immediate to PC | U | rd ← PC + imm20 << 12 |
| XORI rd, rs1, imm12 | XOR immediate | I | rd ← rs1 ^ imm12 |
| SLL rd, rs1, rs2 | Shift left logical | R | rd ← rs1 << rs2 |
| SRL rd, rs1, rs2 | Shift right logical | R | rd ← rs1 >> rs2 |
| SRA rd, rs1, rs2 | Shift right arithmetic | R | rd ← rs1 >> rs2 |
| SLLI rd, rs1, shamt | Shift left logical immediate | I | rd ← rs1 << shamt |
| SRLI rd, rs1, shamt | Shift right logical imm. | I | rd ← rs1 >> shamt |
| SRAI rd, rs1, shamt | Shift right arithmetic immediate | I | rd ← rs1 >> shamt |

## Load / Store Operations

| Mnemonic | Instruction | Type | Description |
|---|---|---|---|
| LD rd, imm12(rs1) | Load doubleword | I | rd ← mem[rs1 + imm12] |
| LW rd, imm12(rs1) | Load word | I | rd ← mem[rs1 + imm12] |
| LH rd, imm12(rs1) | Load halfword | I | rd ← mem[rs1 + imm12] |
| LB rd, imm12(rs1) | Load byte | I | rd ← mem[rs1 + imm12] |
| LWU rd, imm12(rs1) | Load word unsigned | I | rd ← mem[rs1 + imm12] |
| LHU rd, imm12(rs1) | Load halfword unsigned | I | rd ← mem[rs1 + imm12] |
| LBU rd, imm12(rs1) | Load byte unsigned | I | rd ← mem[rs1 + imm12] |
| SD rs2, imm12(rs1) | Store doubleword | S | rs2 → mem[rs1 + imm12] |
| SW rs2, imm12(rs1) | Store word | S | rs2(31:0) → mem[rs1 + imm12] |
| SH rs2, imm12(rs1) | Store halfword | S | rs2(15:0) → mem[rs1 + imm12] |
| SB rs2, imm12(rs1) | Store byte | S | rs2(7:0) → mem[rs1 + imm12] |
| BLT rs1, rs2, imm12 | Branch less than | SB | if rs1 < rs2 pc ← pc + imm12 |
| BLTU rs1, rs2, imm12 | Branch less than unsigned | SB | if rs1 < rs2 pc ← pc + imm12 << 1 |
| JAL rd, imm20 | Jump and link | UJ | rd ← pc + 4<br>pc ← pc + imm20 |
| JALR rd, imm12(rs1) | Jump and link register | I | rd ← pc + 4<br>pc ← rs1 + imm12 |

## Pseudo Instructions

| Mnemonic | Instruction | Base instruction(s) |
|---|---|---|
| LI rd, imm12 | Load immediate (near) | ADDI rd, zero, imm12 |
| LI rd, imm | Load immediate (far) | LUI rd, imm[31:12]<br>ADDI rd, rd, imm[11:0] |
| LA rd, sym | Load address (far) | AUIPC rd, sym[31:12]<br>ADDI rd, rd, sym[11:0] |
| MV rd, rs | Copy register | ADDI rd, rs, 0 |
| NOT rd, rs | One's complement | XORI rd, rs, -1 |
| NEG rd, rs | Two's complement | SUB rd, zero, rs |
| BGT rs1, rs2, offset | Branch if rs1 > rs2 | BLT rs2, rs1, offset |
| BLE rs1, rs2, offset | Branch if rs1 ≤ rs2 | BGE rs2, rs1, offset |
| BGTU rs1, rs2, offset | Branch if rs1 > rs2 (unsigned) | BLTU rs2, rs1, offset |
| BLEU rs1, rs2, offset | Branch if rs1 ≤ rs2 (unsigned) | BGEU rs2, rs1, offset |
| CALL offset | Call subroutine (far) | AUIPC ra, offset[31:12]<br>JALR ra, ra, offset[11:0] |
| RET | Return from subroutine | JALR zero, 0(ra) |
| NOP | No operation | ADDI zero, zero, 0 |

JAL rd, imm20 | Jump and link | UJ | rd ← pc + 4<br>pc ← pc + imm20

JALR rd, imm12(rs1) | Jump and link register | I | rd ← pc + 4<br>pc ← rs1 + imm12

## 32-bit instruction format




## Register File

| | | | |
|---|---|---|---|
| r0 | r1 | r2 | r3 |
| r4 | r5 | r6 | r7 |
| r8 | r9 | r10 | r11 |
| r12 | r13 | r14 | r15 |
| r16 | r17 | r18 | r19 |
| r20 | r21 | r22 | r23 |
| r24 | r25 | r26 | r27 |
| r28 | r29 | r30 | r31 |

## Register Aliases

| | | | |
|---|---|---|---|
| zero | ra | sp | gp |
| tp | t0 | t1 | t2 |
| s0/fp | s1 | a0 | a1 |
| a2 | a3 | a4 | a5 |
| a6 | a7 | s2 | s3 |
| s4 | s5 | s6 | s7 |
| s8 | s9 | s10 | s11 |
| t3 | t4 | t5 | t6 |

**ra** - return address
**sp** - stack pointer
**gp** - global pointer
**tp** - thread pointer

**t0 - t6** - Temporary registers
**s0 - s11** - Saved by callee
**a0 - 17** - Function arguments
**a0 - a1** - Return value(s)

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

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

    j     loop
```

*Right before jump, "write down" address of next instruction, save that address in register* `ra`

*Effectively* `ra=pc+4`

*Return at instruction whose address saved in register* `ra`

*Effectively* `pc=ra`

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

*pseudo instruction expands to* `jalr zero,0(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

**`jal`** **jump and link**

Save pc+4 to rd, jump target (target val immediate)

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

**`jalr`** **jump and link register**

Save pc+4 to rd, jump target (target val in register, +offset)

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

**Pseudo-instruction conveniences**

```
call fn    ->    jal ra,fn
jr reg     ->    jalr zero,0(reg)
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;
}
```

**Control flow: call and resume**

**Mechanism to pass arguments, return value**

**Shared used of registers, scratch space**

**Storage for local variables**

*Complication: nested function calls, recursion*

# Application binary interface

ABI specifies low-level machine interface and conventions that must be followed

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

**Functions defined in separate modules/libraries must adhere to same ABI in order to successfully interoperate**

# Mechanics of call/return

Pass up to 8 args in registers **a0-a7**

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

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

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

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

Return value put into **a0**

**ret** (**jalr**) jumps back to **ra** (i.e. resume previous)

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

# 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**

# Shared use of registers

Only one set of shared registers, must coordinate on which code has "rights" to register during function call

Half of registers designated **callee-owned**

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

Other half designated **caller-owned**

> **Caller** retains ownership of these registers, register contents must be same after call as before call
>
> **Callee** can use these registers only if preserve value and restore (thus fulfilling caller's expectation that value is unchanged)

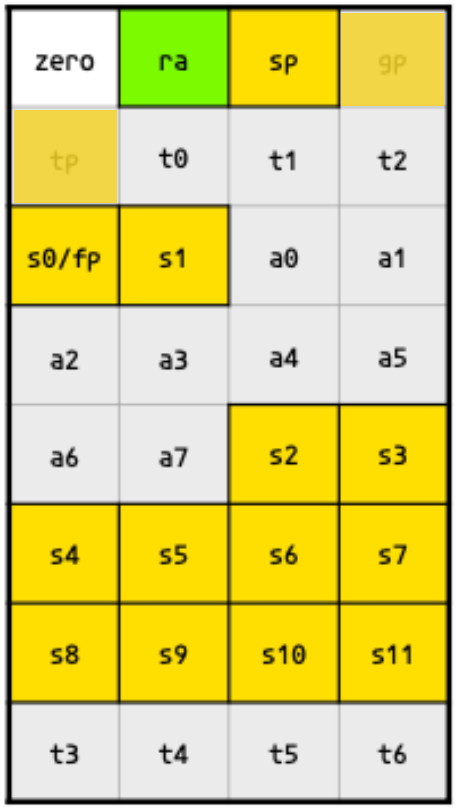
# Register ownership

**callee-owned** `a0-a7,t0-t6`

(a=arg, t=temp)

**caller-owned** `s0-s11,ra-tp`

(s=saved)



# The need for scratch space

Callee has free use of all callee-owned registers, but could need more than that. Why?

- has many local variables with overlapping live ranges

- if callee makes a call to another function, has to cede all callee-owned registers to it

- if callee needs to use a caller-owned register, has to preserve value somewhere so can restore later

**Where else can we store additional values and/or copy register values for safekeeping?**

# The stack to the rescue!

Reserve section of memory to store data for executing function

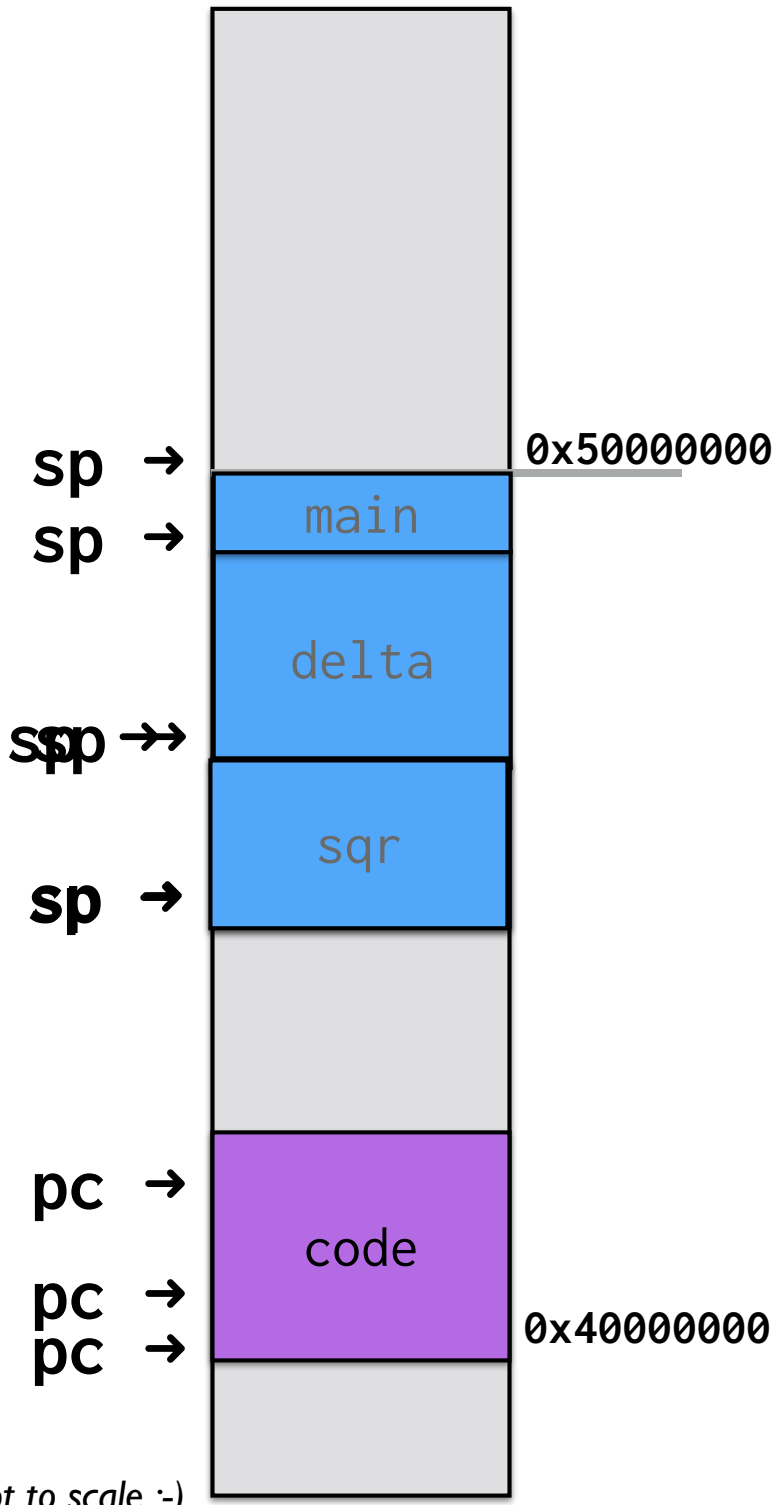
Stack frame allocated per function invocation
Can store local variables, scratch values, saved registers

**sp** points to location in memory of lastmost value pushed
Per function call:
- Function entry: decrement **sp** makes space for stack frame ("push")
  - During function: access data in frame using **sp**-relative offset
- Function exit: increment **sp** to clean up frame ("pop")

Stack is LIFO, last frame pushed is first frame popped
Stack memory grows down (inner frames at lower addresses)

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

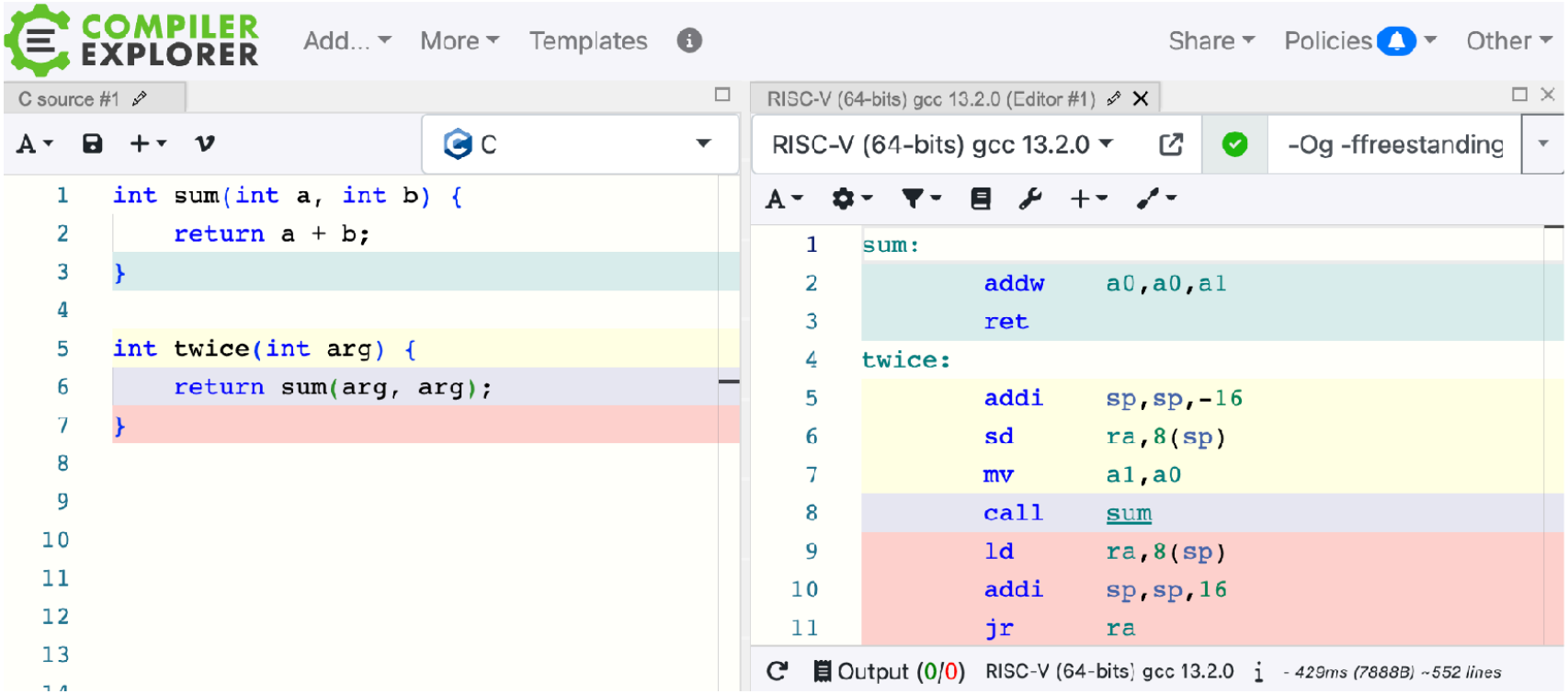
```
void main(void)
{
   delta(3, 7);
}

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

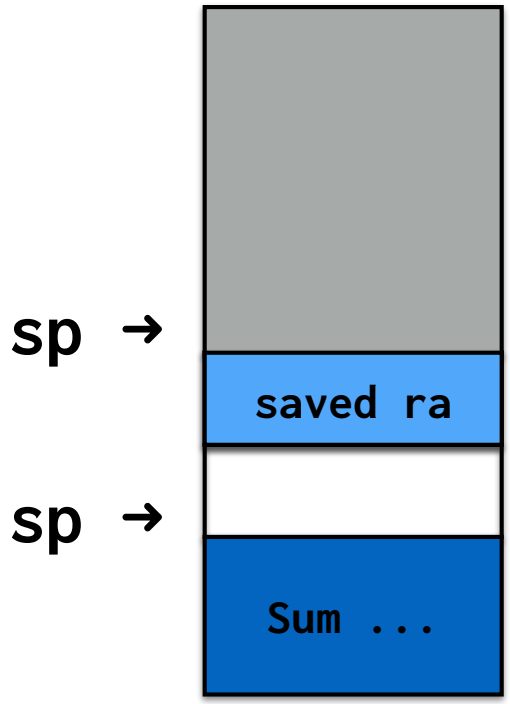
Let's use it to see function-call mechanisms and use of stack




https://godbolt.org/

# Stack operation

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

```
    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,de

    j       loop
```

better

```
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);
    }
}
```

best!

```
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);
    }
}
```