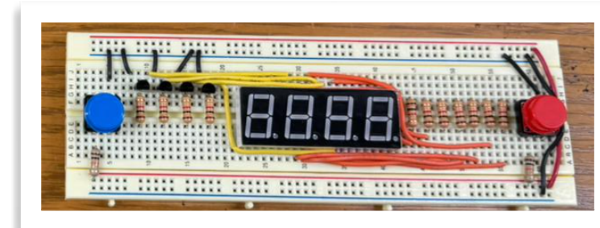
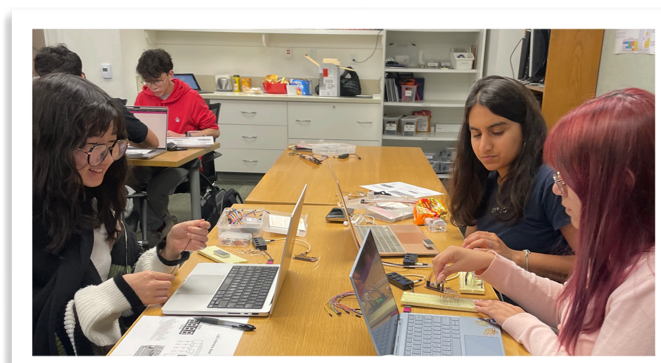
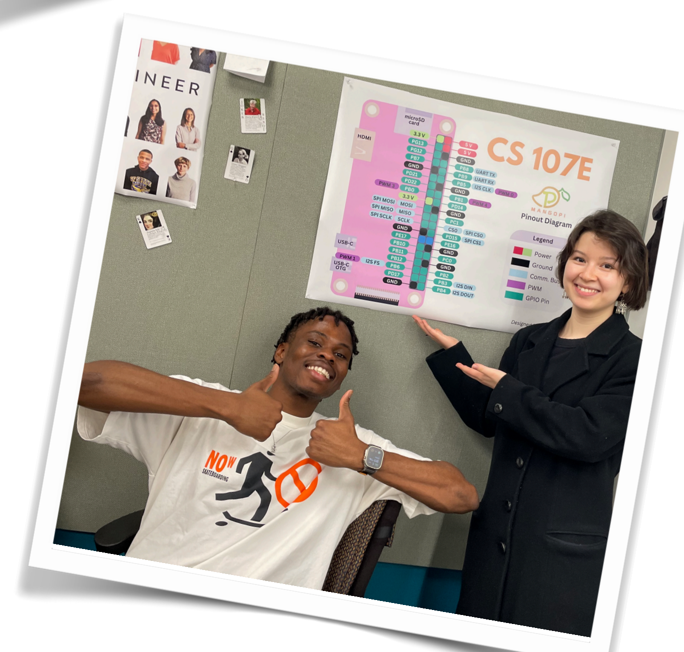
Admin

assign1 due Tuesday
lab2 Wednesday







Today: C functions

Implementation of C function calls

Management of runtime stack, register use

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
```

How to remember where we came from, so we can go back there…

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

How to communicate arguments to function?

```
loop:

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

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

  j    loop
```

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

New instructions

Jump and Link jal

Saves pc+4 into rd before jump to target (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

Also 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);
```

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

Callee stores return value in `a0`
ret (**jalr**) jumps back to ra

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

Caller and Callee

caller: function doing the calling

callee: function being called

main is caller of `range`

`range` is callee of `main`

`range` is caller of `abs`

```
void main(void) {
  range(13, 99);
}

int range(int a, int b) {
  return abs(a-b);
}

int abs(int v) {
  return v < 0 ? -v : v;
}
```

Register Ownership

a0-a7,t0-t6 are **callee-owned** registers

- **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

- **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

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,0x6000
call  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;
}
```

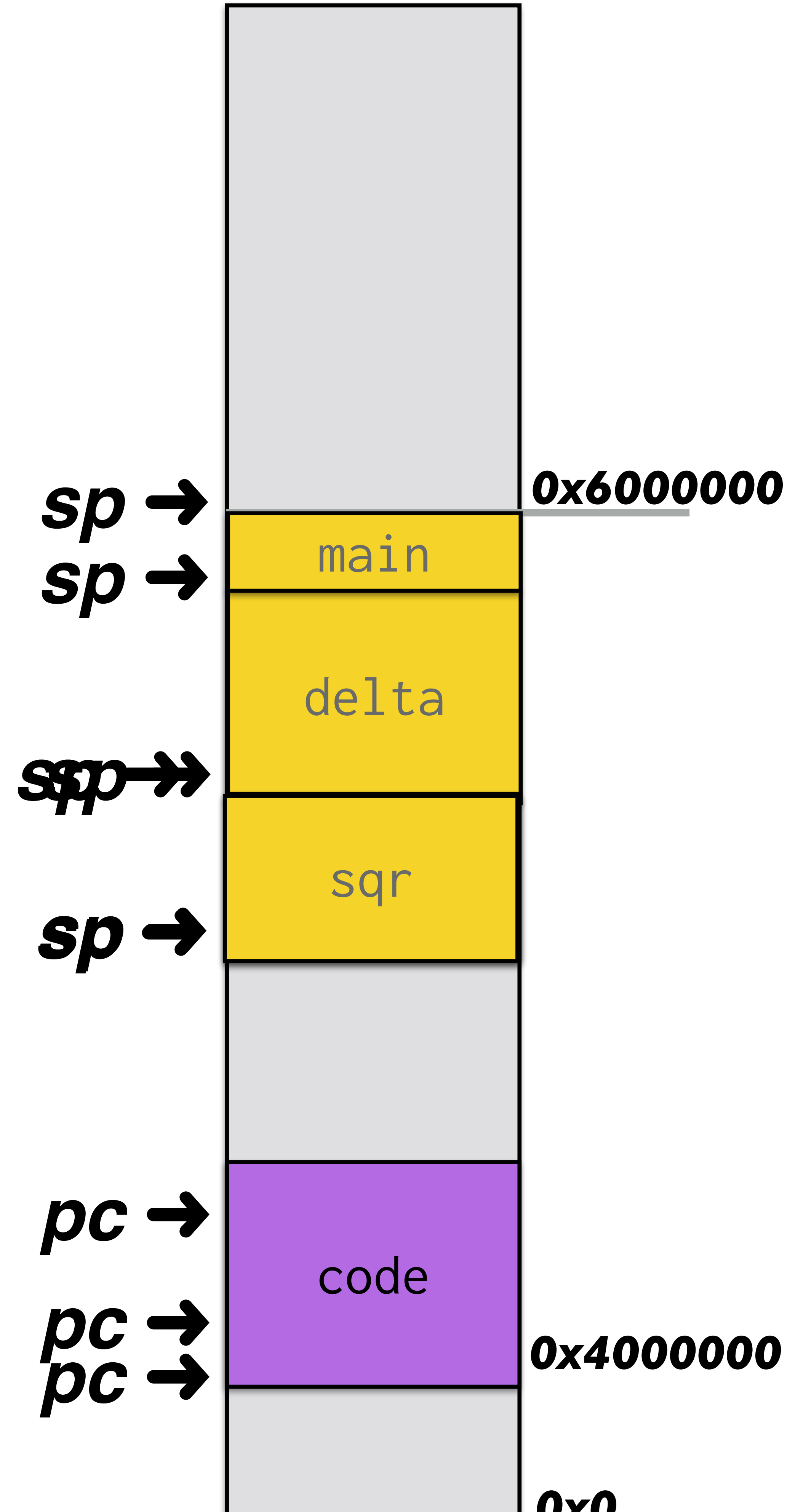
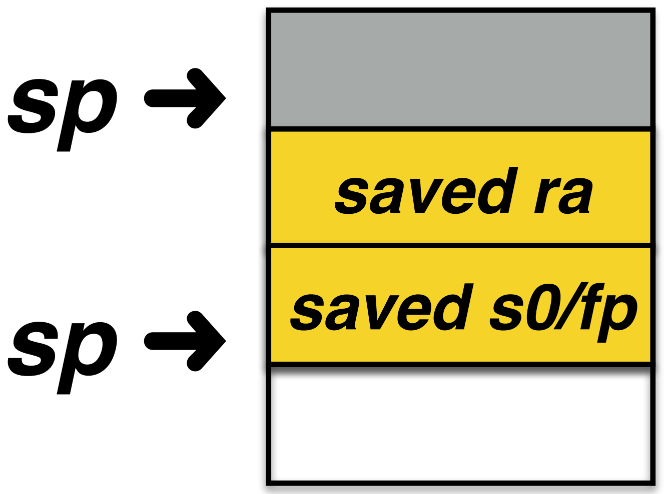


Diagram not to scale

Stack operation

```
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)
add   sp,sp,16
ret
```



Gdb debugger

Debugger is incredibly useful

Allows you to run your program in a monitored context

Can set breakpoints, examine state, change values, reroute control, and more

gdb has simulation mode where it pretends to be an RISC-V processor, running on your laptop 🙌

Pretty good approximation (not perfect, e.g. no peripherals)

Let's try it now!

Run under debugger and observe stack in action

```
$ riscv64-unknown-elf-gdb -q --command=$CS107E/
other/gdbsim.commands program.elf
(gdb) target sim
(gdb) load
```

👀 **Read our course guide on gdb!** 👀

http://cs107e.github.io/guides/gdb/

(this guide will be updated ASAP!)

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