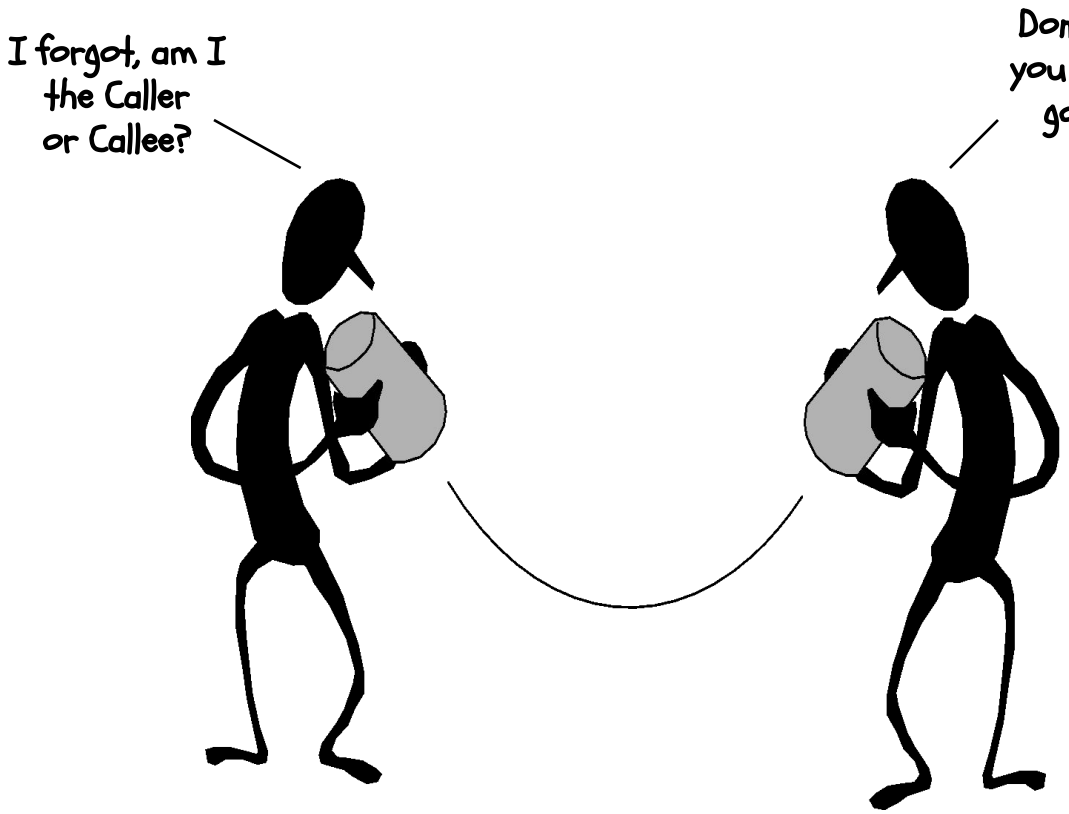
Stacks and Procedures



- Problem set #1 is due before midnight tonight
- Problem set #2 will become active this afternoon

Language support for modular code is an integral part of modern computer organization. In particular, support for subroutines, procedures, and functions.

Pseudo-instructions

The miniRISCV assembler/simulator supports many mnemonics that aren't actual instructions at all. They provide common shortcuts.

Pseudoinstruction	Translation
`neg rd,rs`	`sub rd,x0,rs`
`not rd,rs`	`xori rd,rs,-1`
`j label`	`jal x,label`
`mv rd,rs`	`addi rd,rs,0`
`ret`	`jalr x0,x1`
`bgt rs,rt,label`	`blt rt,rs,label`
`ble rs,rt,label`	`bge rt,rs,label`
`la  rd,label  # pc relative`	`auipc rd,label_hi` `addi  rd,rd,label_lo`
`li  rd,value`	`lui   rd,value_hi` `addi  rd,rd,value_lo`

Assembler Directives

There are also many assembler directives that provide hints to the assembler for allocating memory locations

Directive	Meaning
`.word v1,v2,v3, ...,vn`	Initialize sequential memory words with values v1, v2, etc
`.space N`	Allocate space for N words
`.string "anysizeoftext"`	Initialize sequential memory bytes based on the given string
`.text`	Place the following in the .text segment (usually instructions)
`.data`	Place the following in the .data segment (usually global or static data declarations)
`.align N`	N must be a power of 2. Adjusts the next address in the current segment such that address % N == 0

The Beauty of Procedures

- Reusable code fragments (modular design)

```
clear_screen();
    …       // code to draw a bunch of lines
clear_screen();
    …
```

- Parameterized procedures (variable behaviors)

```
line(x1,y1,x2,y2,color);
line(x2,y2,x3,y3,color);
    …
```

```
for (int i = 0; i < N-1; i++)
    line(x[i],y[i],x[i+1],y[i+1],color);
line(x[i],y[i],x[0],y[0],color);
```

- Functions (procedures that return values)

```
xMax = max(max(x1,x2),x3);
yMax = max(max(y1,y2),y3);
```

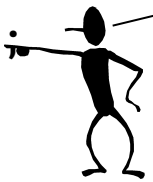
More Procedure Power

- Global vs. Local scope (Name Independence)

```
int x = 9;

int fee(int x) {
   return x+x-1;
}

int foo(int i) {
   int x = 0;
   while (i > 0) {
         x = x + fee(i);
         i = i - 1;
   }
  return x;
}

main() {
     fee(foo(x));
}
```

These are different "x"s

This is yet another "x"



That "fee( )" seems odd to me?
And, foo( )'s a little square.

How do we keep track of all these variables?

Recap: We defined ABI conventions

By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages. We'll use these names increasingly. Why have such conventions?

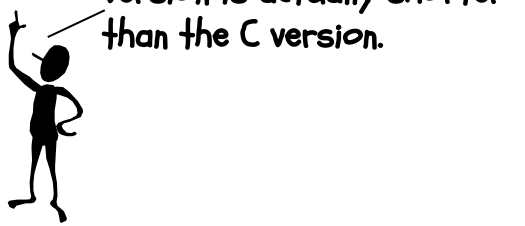
x0/zero (always zero)	x16/a6 (argument)
x1/ra (return address)	x17/a7 (argument)
x2/sp (stack pointer)	x18/s2 (saved)
x3/gp (global pointer)	x19/s3 (saved)
x4/tp (thread pointer)	x20/s4 (saved)
x5/t0 (temporary)	x21/s5 (saved)
x6/t1 (temporary)	x22 (saved)
x7/t2 (temporary)	x23 (saved)
x8/fp (frame pointer)	x24 (saved)
x9/s1 (saved)	x25 (saved)
x10/a0 (argument/return value 1)	x26 (saved)
x11/a1 (argument/return value 2)	x27 (saved)
x12/a2 (argument)	x28 (temporary)
x13/a3 (argument)	x29 (temporary)
x14/a4 (argument)	X30 (temporary)
x15/a5 (argument)	X31 (temporary)

A Function That Worked

```
main:   lw  a0,x
        lw  a1,y
        jal ra,gcd
        sw  a0,z

*halt:  j   halt

x:      .word   35
y:      .word   55
z:      .word   0
```

```
int x = 35;
int y = 55;
int z;

void main() {
    z = gcd(x, y);
}

int gcd(a,b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

```
gcd:    beq  a0,a1,return
        blt  a0,a1,else
        sub  a0,a0,a1
        beq  x0,x0,gcd
else:   sub  a1,a1,a0
        beq  x0,x0,gcd
return: jalr zero,(ra)
```



And one that Didn't

```
main:   lw  a0,x
        jal ra,fact
        sw  a0,y
*halt:  j   halt

x:      .word   2
y:      .word   0
```

```
fact:   addi  t0,x0,1
        bge   t0,a0,return
        addi  t0,x0,a0
        addi  a0,a0,-1
        jal   ra,fact
        mul   a0,a0,t0
return: jalr  x0,ra
```

```
int x = 5;
int y;

void main() {
    y = fact(x);
}

int fact(x) {
    if (x <= 1)
        return x;
    else
        return x*fact(x-1);
}
```

This time, things are really messed up.

The recursive call to fact( ) overwrites the daved value of x in t0.

To make a bad thing worse, the ra is also overwritten.

I knew there was a reason that I avoid recursion.

A Simple Case

```
x:      .word 9
```

Callee

```
fee:    add     a0,a0,a0
        addi    a0,a0,1
        jalr    x0,ra
```

The "jalr" instruction changes the PC to the contents of the specified register. Here it is used to return to the address after the one where "fee" was called.

Caller

```
main:   lw      a0,x
        jal     ra,fee
        jalr    x0,ra
```

Recall that the address of the next instruction is saved in the "linkage pointer", LP.

Works for cases where Callees need few resources and call no other functions.

This type of function (one that calls no other) is called a LEAF function.

But there are still a few issues:

- How does a Callee call functions?
- More than 4 arguments?
- Local variables?
- Where does main return to?

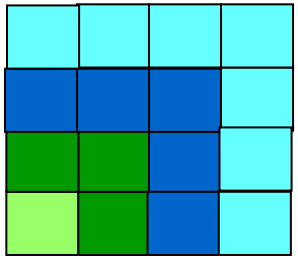
Let's consider the worst case of a Callee who is a Caller...

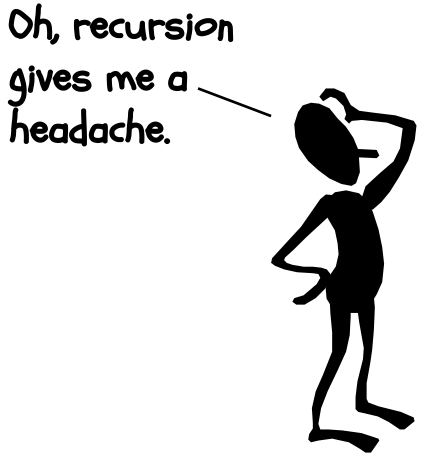
Callees who call themself!

```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}


main() {
  sqr(10);
}
```

How do we go about writing non-leaf procedures? Procedures that call other procedures, perhaps even themselves.

sqr(10) = sqr(9)+10+10-1 = 100
sqr(9) = sqr(8)+9+9-1 = 81
sqr(8) = sqr(7)+8+8-1 = 64
sqr(7) = sqr(6)+7+7-1 = 49
sqr(6) = sqr(5)+6+6-1 = 36
sqr(5) = sqr(4)+5+5-1 = 25
sqr(4) = sqr(3)+4+4-1 = 16
sqr(3) = sqr(2)+3+3-1 = 9
sqr(2) = sqr(1)+2+2-1 = 4
sqr(1) = 1
sqr(0) = 0



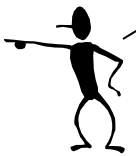


A First Try

OOPS!

```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}


main()
{
  sqr(10);
}
```

```
sqr:    slti    t0,a0,2
        bne     t0,x0,return
        add     s0,x0,a0
        addi    a0,a0,-1
        jal     ra,sqr
        add     a0,a0,s0
        add     a0,a0,s0
        addi    a0,a0,-1
return: jalr    x0,ra


main:   addi    a0,x0,10
        jal     ra,sqr
        jalr    x0,ra
```



Will saving "x" in memory rather than in a register help?

i.e. replace **`add s0,x0,a0`** with **`sw a0,x`** and adding **`lw s0,x`** after **`jal sqr`**

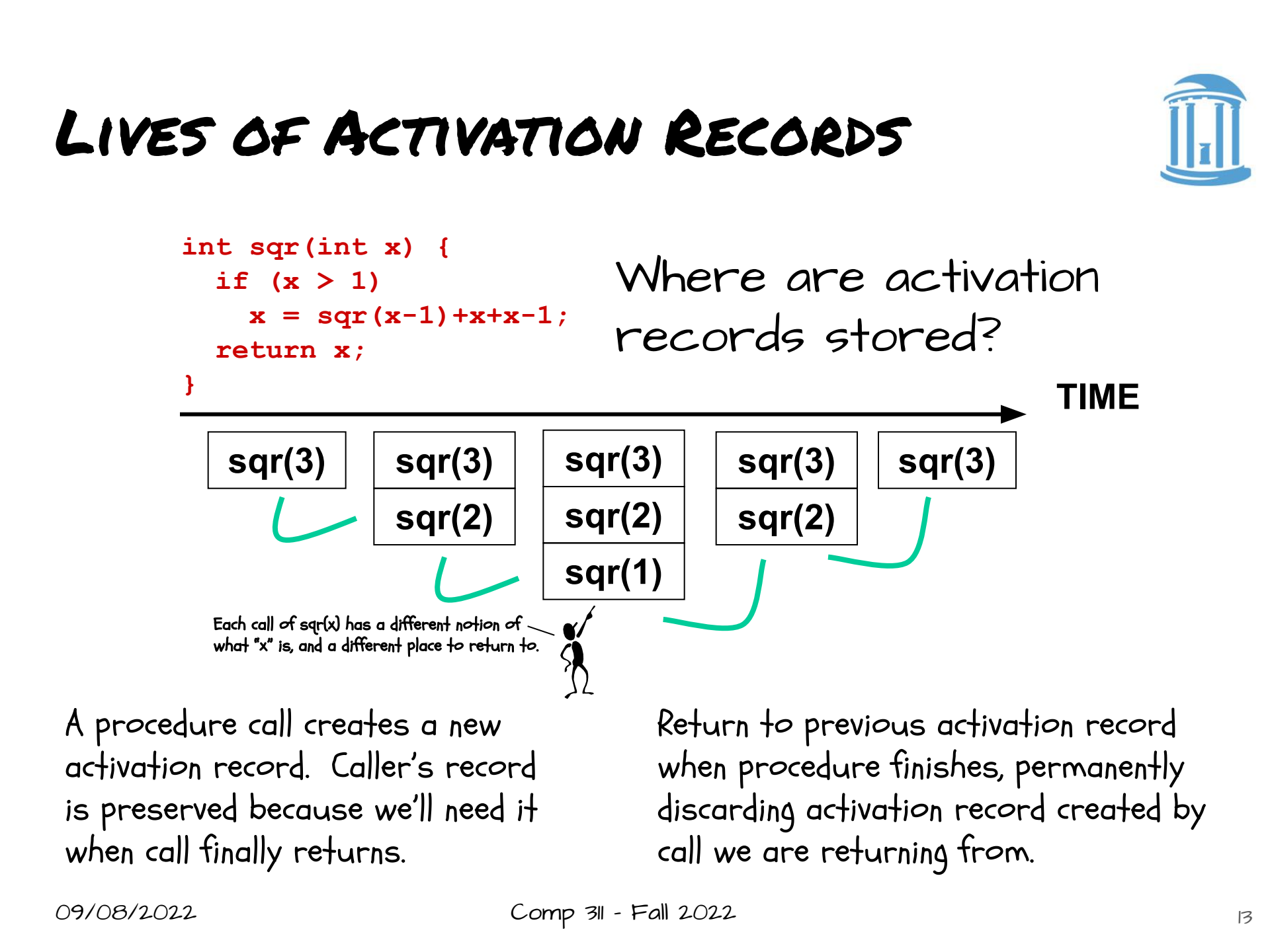
A Procedure's Storage Needs

- In addition to a conventions for using registers to pass in arguments and return results, we also need a means for **allocating new variables for each instance when a procedure is called.**
  The "Local variables" of the Callee:

```
...
{
  int x, y;
  ... x ... y ...;
}
```

- Local variables are specific to a "particular" invocation or ***activation*** of the Callee. Collectively, the arguments passed in, the return address, and the callee's local variables are its ***activation record***, or ***call frame***.

Lives of Activation Records



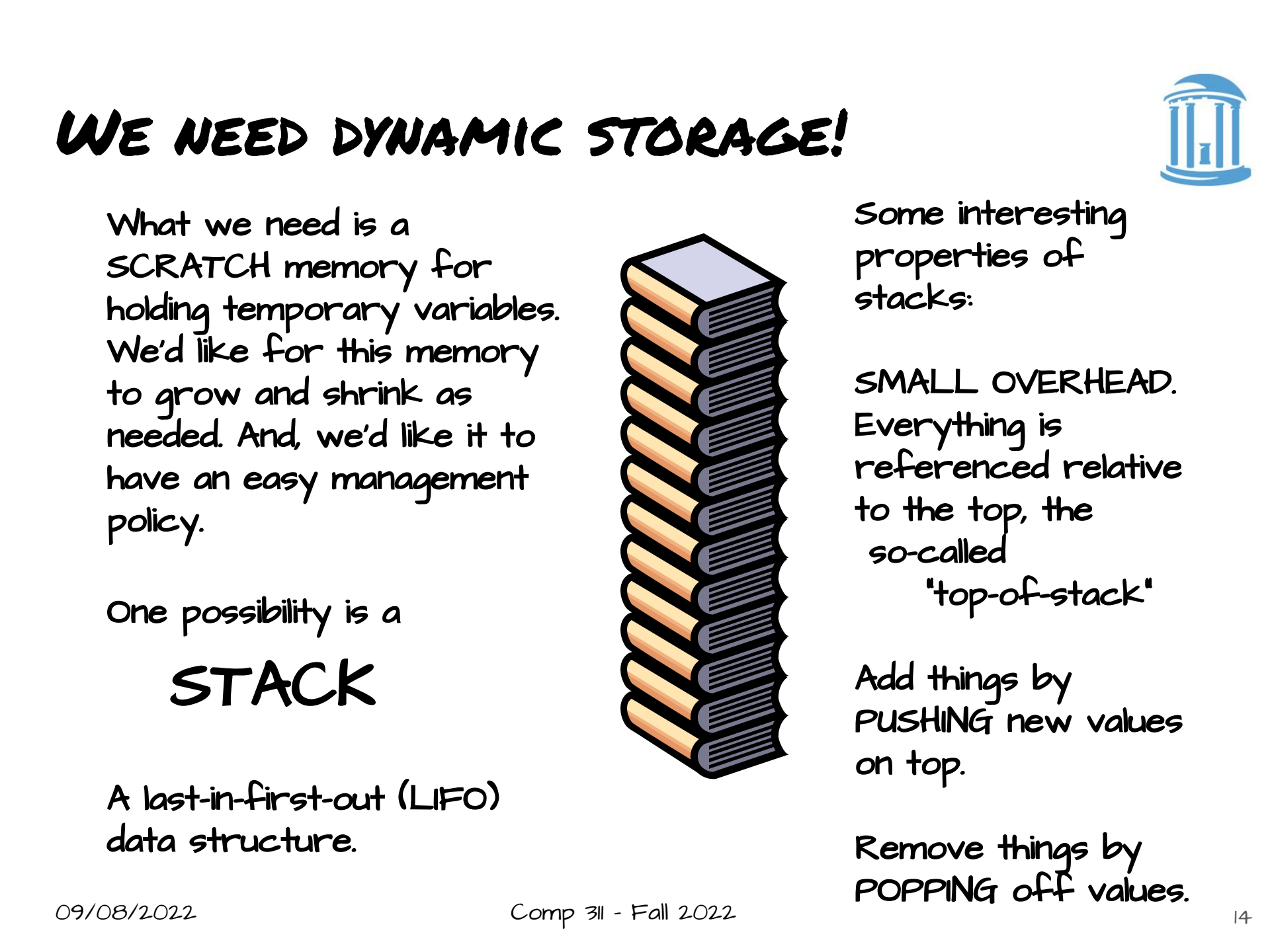
```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}
```

Where are activation records stored?

A procedure call creates a new activation record. Caller's record is preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

We need dynamic storage!



What we need is a SCRATCH memory for holding temporary variables. We'd like for this memory to grow and shrink as needed. And, we'd like it to have an easy management policy.

One possibility is a

STACK

A last-in-first-out (LIFO) data structure.

Some interesting properties of stacks:

SMALL OVERHEAD. Everything is referenced relative to the top, the so-called "top-of-stack"

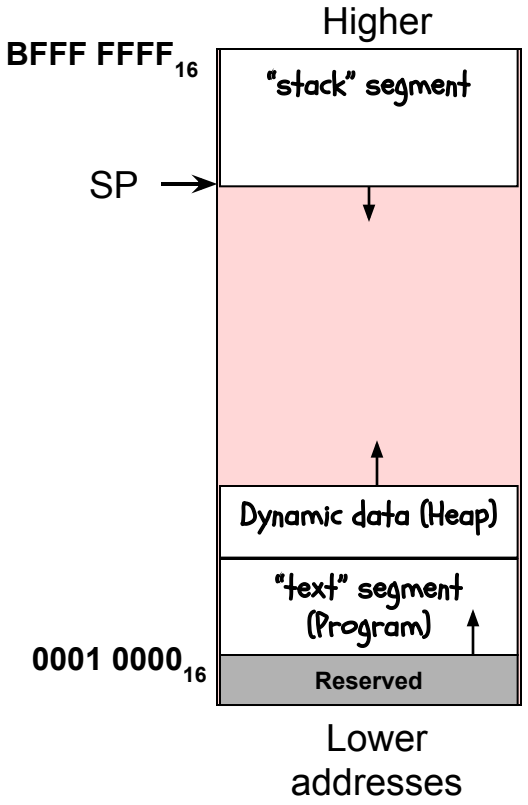
Add things by PUSHING new values on top.

Remove things by POPPING off values.

RISC-V Stack Convention

Conventions:

- Allocate a register for the Stack Pointer (SP = x2).
- Stack grows DOWN (towards lower addresses) on pushes and allocates
- SP points to the last or TOP *used* location.
- Stack is placed far away from the program and its data.



Stack Management

ALLOCATE k: reserve k WORDS of stack

SP = SP - 4*k

```
addi    sp,sp,-4*k
```

DEALLOCATE k: release k WORDS of stack

SP = SP + 4*k

```
addi   sp,sp,4*k
```

PUSH $x: push Reg[x] onto stack

Mem[SP - 4] = Rx

SP = SP - 4

```
addi     sp,sp,-4
sw       rx,(sp)
```

POP $x: pop the top of the stack into Reg[x]

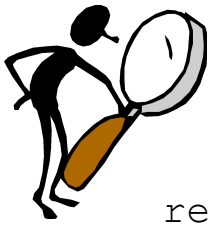
Rx = Mem[SP]

SP = SP + 4

```
lw       rx,(sp)
addi     sp,sp,4
```

Incorporating a Stack

```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}


main()
{
  sqr(10);
}
```



```
sqr:    addi    sp,sp,-8        function
        sw      ra,4(sp)        prologue
        sw      s0,0(sp)
        slti    t0,a0,2
        bne     t0,x0,return
        add     s0,x0,a0
        addi    a0,a0,-1
        jal     ra,sqr
        add     a0,a0,s0
        add     a0,a0,s0
        addi    a0,a0,-1
return: lw      s0,0(sp)        function
        lw      ra,4(sp)        epilogue
        addi    sp,sp,8
        jalr    x0,ra

main:   addi    sp,sp,-4
        sw      ra,(sp)
        addi    a0,x0,10
        jal     ra,sqr
        lw      ra,(sp)
        addi    sp,sp,4
        jalr    x0,ra
```

Next time

Still some loose ends to tie up

1. More than 8 arguments

```
foo(a,b,c,d,e,f,g,h,i)
```

2. Addresses of arguments

```
int fee(x) {
    int *y = &x;
}
```

3. Complex argument types

```
int a[10];
struct point { int x; int y; };
struct point p = { 3, 4 };

y = sum(a, &p);
```