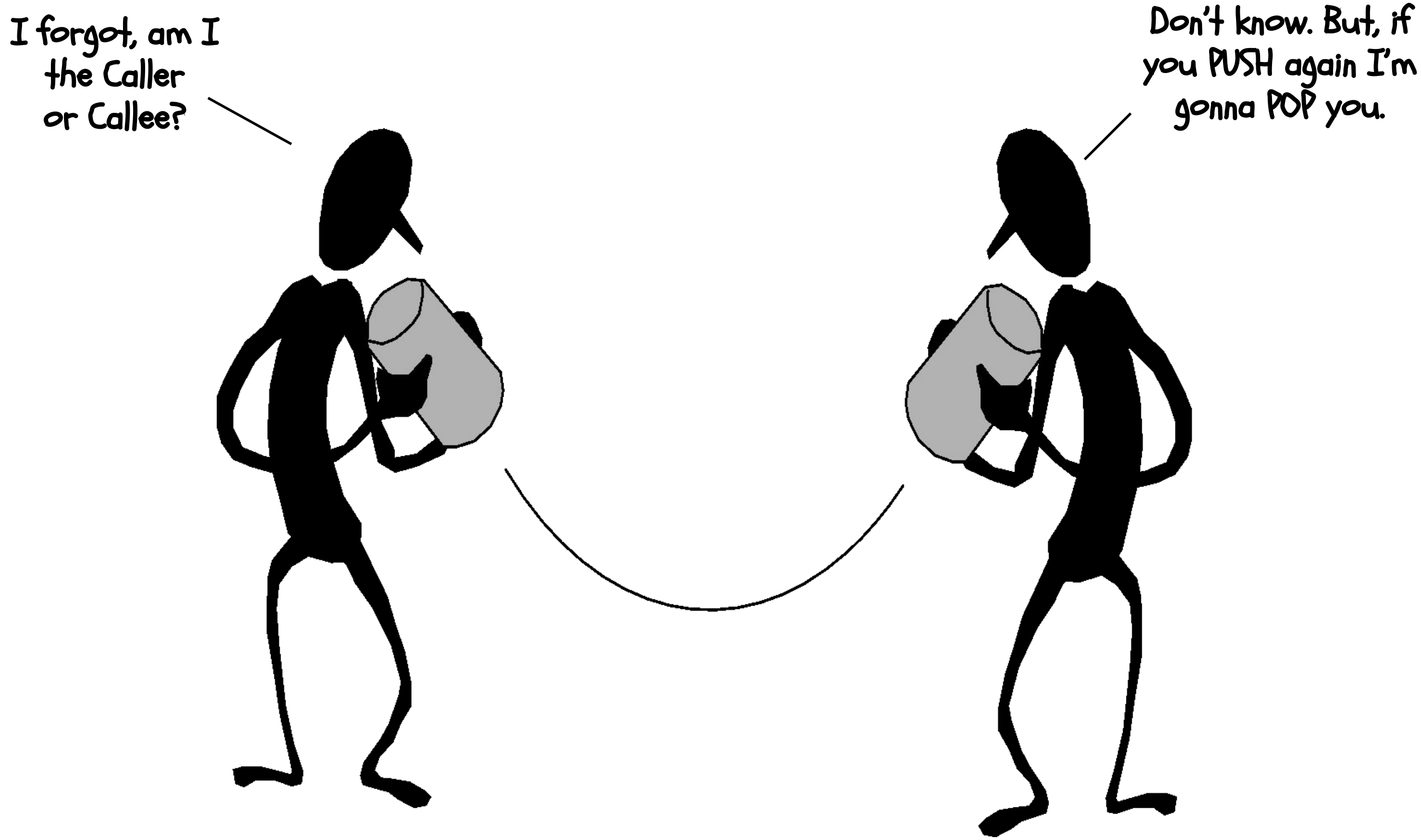
Stacks and Procedures



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

The Beauty of Procedures

- Reusable code fragments (modular design)

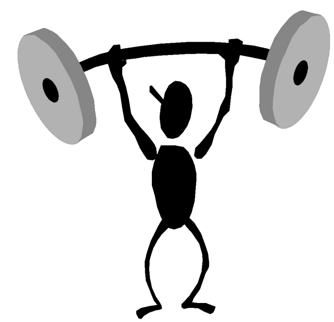
```
clear_screen();
    …       // code to draw a bunch of lines
clear_screen();
    …
```

- 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)

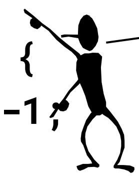


```
xMax = max(max(x1,x2),x3);
yMax = max(max(y1,y2),y3);
```

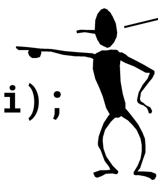
More Procedure Power

- 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"

How do we keep track of all these variables?

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

Using Procedures

- **A "calling" program (Caller) must:**
  - Provide the procedure's parameters. In other words, put arguments in a place where the procedure can access them
  - Transfer control to the procedure. "Branch" to it, and provide a "link" back
- **A "called" procedure (Callee) must:**
  - Acquire/create resources needed to perform the function (local variables, registers, etc.)
  - Perform the function
  - Place results in a place where the Caller can find them
  - Return control back to the Caller through the supplied link
- **Solution (a least a partial one):**
  - WE NEED CONVENTIONS, agreed upon standards for how arguments are passed in and how function results are retrieved
  - Solution part #1: Allocate registers for these specific functions

ARM Register Usage

Recall these conventions from last time

- Conventions designate registers for procedure arguments (R0-R3) and return values (R0-R3).
- The ISA designates a "linkage pointer" for calling procedures (R14)
- Transfer control to Callee using the **BL** instruction
- Return to Caller with the **BX LR** instruction

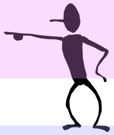
Register	Use
R0-R3	First 4 procedure arguments. Return values are placed in R0 and R1.
R4-R10	Saved registers. Must save before using and restore before returning.
R11	FP - Frame pointer (to access a procedure's local variables)
R12	IP - Temp register used by assembler
R13	SP - Stack pointer Points to next available word
R14	LR - Link Register (return address)
R15	PC - program counter

And it almost works!

```
x:          .word 9
```

Callee

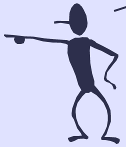
```
fee:        ADD     R0,R0,R0
            ADD     R0,R0,#1
            BX      LR
```



The "BX" 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:       LDR     R0,x
            BL      fee
            BX      LR
```



Recall that when the "L" suffix is appended to a branch instruction, it causes the address of the next instruction to be saved in the "linkage register", LR.

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?

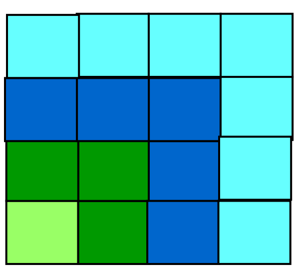
Let's consider the worst case of a Callee who is a Caller…

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



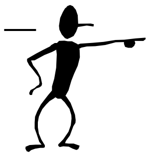
A First Try

OOPS!

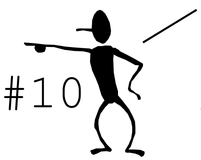
```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}

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

```
sqr:     CMP   R0,#1
         BLE   return
         MOV   R4,R0
         SUB   R0,R0,#1
         BL    SQR
         ADD   R0,R0,R4
         ADD   R0,R0,R4
         SUB   R0,R0,#1
return:  BX    LR

main:    MOV   R0,#10
         BL    sqr
         BX    LR
```

R4 is clobbered on successive calls.



We also clobber our return address, so there's no way back!



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

i.e. replace `MOV R4,R0` with `STR R0,x` and adding `LDR R4,x` after `BL 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:

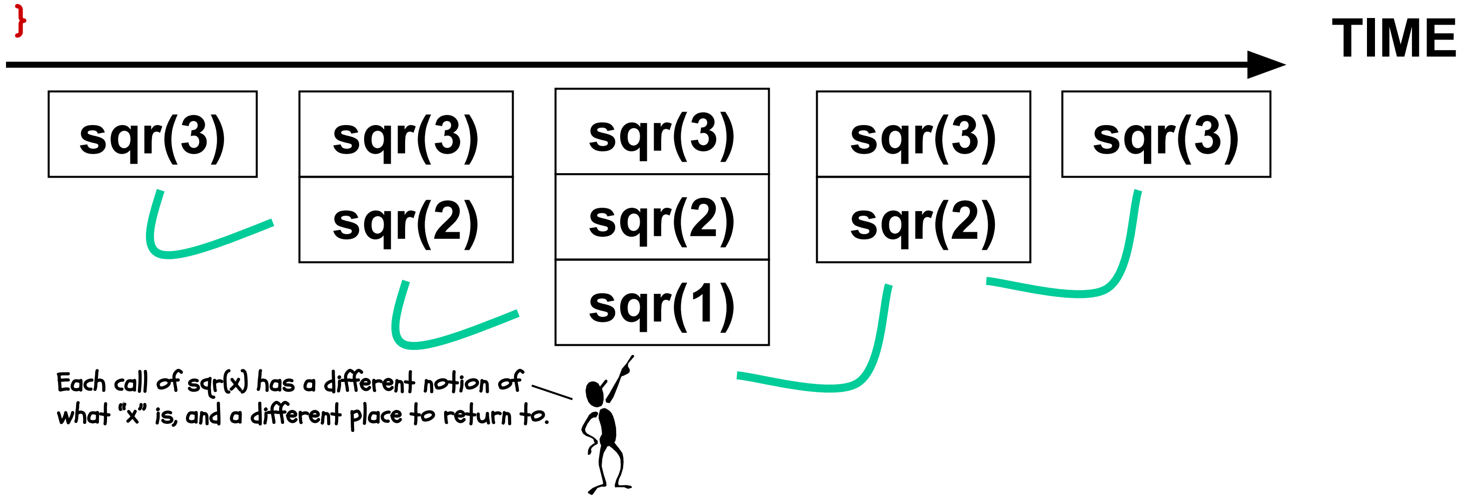
```
...
{
  int x, y;
  ... x ... y ...;
}
```

- 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.

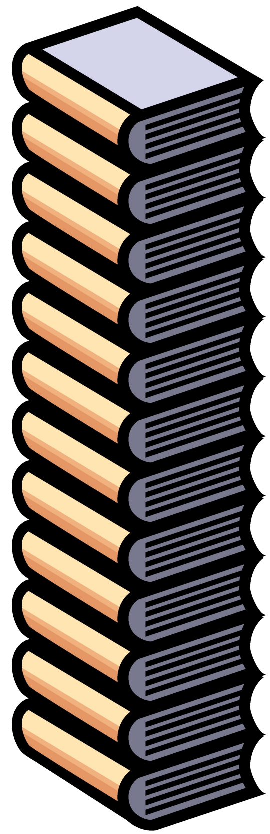
We need dynamic storage!

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"

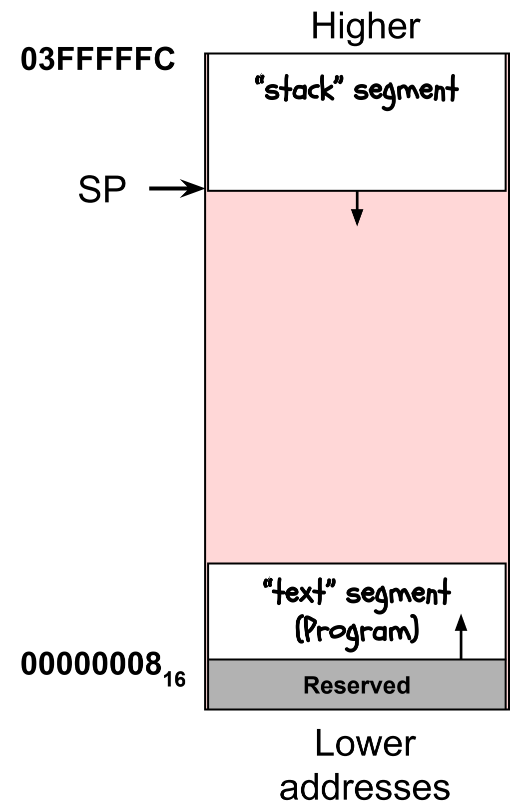
Add things by PUSHING new values on top.

Remove things by POPPING off values.

ARM Stack Convention

Conventions:

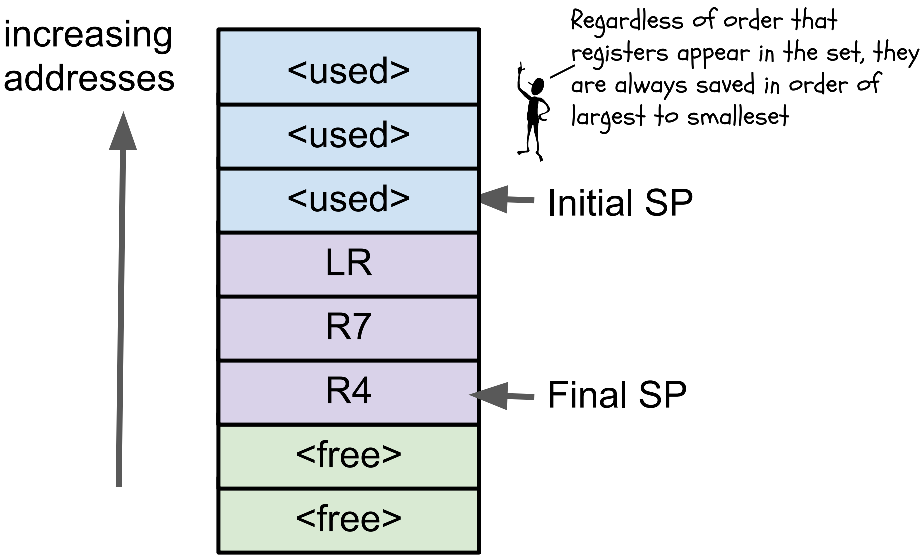
- Dedicate a register for the Stack Pointer (SP = 13).
- Stack grows DOWN (towards lower addresses) on pushes and allocates
- SP points to the last or TOP *used* location.
- Stack is placed far away from the program and its data.



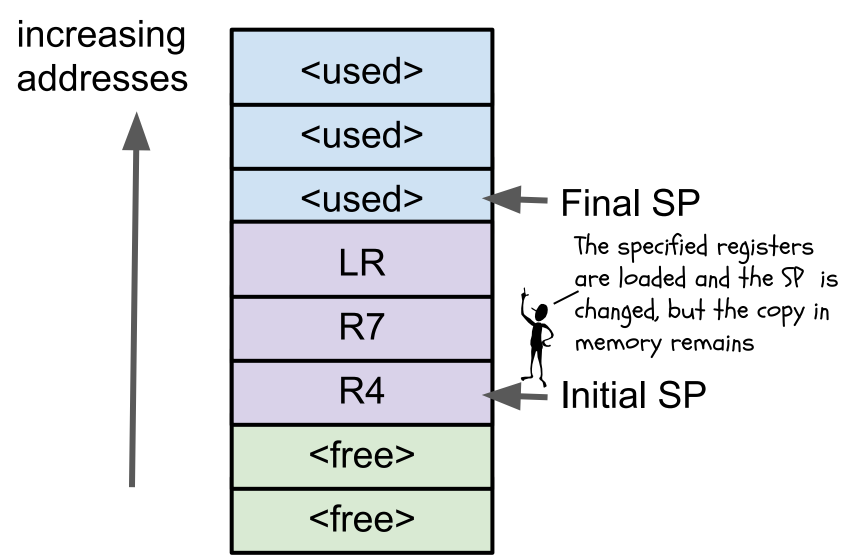
Turbo Stack Instructions

Recall ARM's block move instructions **LDMFD** and **STMFD** which are ideal for implementing our stack. The "M" means multiple, the "F" means full (i.e. the SP points to the last pushed entry, as opposed to "E" for empty, the next available entry), and the "D" stands for descending (growing towards lower addresses, vs. "A" for ascending).

```
STMFD  SP!,{r4,r7,LR}
```

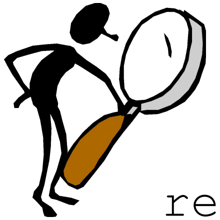



```
LRMFD  SP!,{r4,r7,LR}
```



Incorporating a Stack

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


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



```
sqr:        STMFD   SP!,{R4,LR}
            CMP     R0,#1
            BLE     return
            MOV     R4,R0
            SUB     R0,R0,#1
            BL      SQR
            ADD     R0,R0,R4
            ADD     R0,R0,R4
            SUB     R0,R0,#1
return:     LRMFD   SP!,{R4,LR}
            BX      LR

main:       MOV     R0,#10
            BL      sqr
            BX      LR
```

Revisiting Factorial

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

int x = 5;
int y;

y = fact(x);
```

```
main:   ldr     r0,x
        bl      fact
        str     r0,y
        bx      lr

x:      .word   5
y:      .word   0

fact:   stmfd   sp!,{r4,lr}
        cmp     r0,#1
        ble     return
        mov     r4,r0
        sub     r0,r0,#1
        bl      fact
        mul     r0,r0,r4
return: ldmfd   sp!,{r4,lr}
        bx      lr
```

miniARM

It works! And the changes are relatively small. Just saving r4 and lp on entry, and replacing them before returning

Missing Details

Thus far the stack has been only been used by callee's that are also callers (i.e. non-leaf procedures) to save resources that "they" and "their caller" expect to be preserved.

Our procedure calling convention works, but it has a few limitations...

1. Callee's are limited to 4 arguments
2. All arguments must "fit" into a single register
3. What if our argument is not a "value", but instead, an address of where to put a result (i.e. an array, an object, etc.)

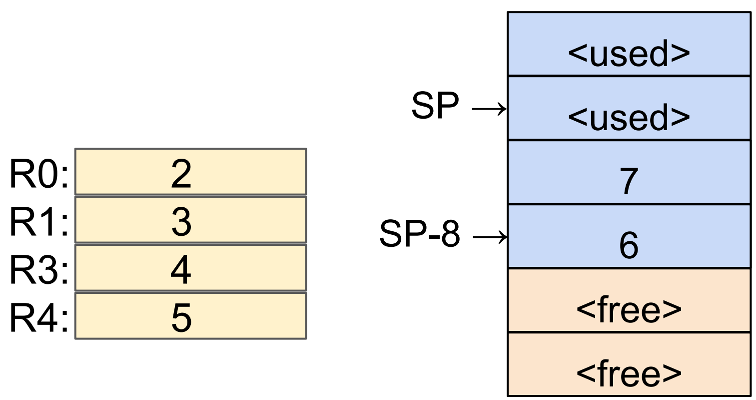
'Which brings us to my next point.'

Caller provided Storage

If a caller calls a function that requires more than 4 arguments, it must place these extra arguments on the stack, and remove them when the callee returns.

```
int sum6(int a, int b, int c, int d, int e, int f) {
    return a+b+c+d+e+f;
}

int main() {
    return sum6(2,3,4,5,6,7);
}
```

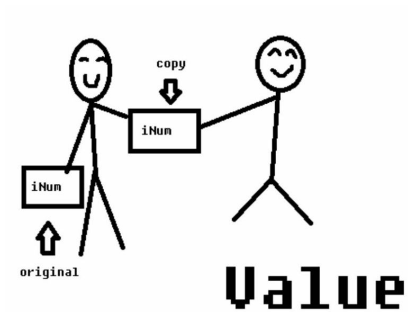
```
sum6:   add     r1,r0,r1        ; b=a+b
        add     r1,r1,r2        ; b=b+c
        add     r1,r1,r3        ; b=b+d
        ldr     r2,[sp, #0]
        add     r0,r1,r2        ; a=b+e
        ldr     r2,[sp, #4]
        add     r0,r0,r2        ; a=a+f
        bx      lr

main:   sub     sp,sp,#8       ; allocate extra args
        mov     r3,#6
        str     r3,[sp,#0]
        mov     r3,#7
        str     r3,[sp,#4]
        mov     r0,#2
        mov     r1,#3
        mov     r2,#4
        mov     r3,#5
        bl      sum6
        add     sp,sp,#8
halt:   b       halt
```

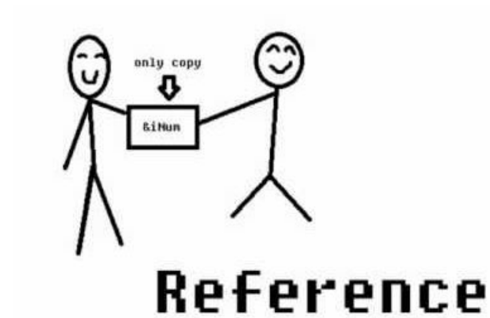


Complex Arguments

How do we pass arguments that don't fit in a register?

- Arrays
- Objects
- Dictionaries
- etc.





Rather than **copy** the complex arguments, we instead just send an "**address**" of where the complex argument is in memory.

Conundrum: Callees process "copies" of simple arguments, and thus any modifications they make don't affect the original. But, with complex arguments, the callee modifies the original version.

Next time

Special variable types for holding "addresses"

1. Pointers
2. Dereferencing
3. Addresses of pointers