**Basics of Calling**

```
LDR R0, x
LDR R1, y
BL  GCD
STR R0, z
```

```
halt: B  halt
```

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

Here the assembly language version is actually shorter than the C/Java version.
That was a little too easy

That was a little too easy

LDR R0, x
BL fact
STR R0, y
halt: B  halt

x:  .word 5
y:  .word 0

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

int x = 5;
int y;
y = fact(x);

This time, things are really messed up.
The recursive call to fact() overwrites the value of x that was saved in R1.

To make a bad thing worse, the LP is also overwritten.
I knew there was a reason that I avoid recursion.
Next Time

- Stacks
- Contracts
- Writing serious code
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);
  ...
  ```

- Functions (procedures that return values)
  ```
  xMax = max(max(x1,x2),x3);
  yMax = max(max(y1,y2),y3);
  ```

  ```
  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);
  ```
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));
  }
  ```

How do we keep track of all these variables?

These are different "x"s

This is yet another "x"

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

● A "calling" program (Caller) must:
  - Provide procedure 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 LP instruction.

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R3</td>
<td>First 4 procedure arguments. Return values are placed in R0 and R1.</td>
</tr>
<tr>
<td>R4-R10</td>
<td>Saved registers. Must save before using and restore before returning.</td>
</tr>
<tr>
<td>R11</td>
<td>FP - Frame pointer (to access a procedure's local variables)</td>
</tr>
<tr>
<td>R12</td>
<td>IP - Temp register used by assembler</td>
</tr>
<tr>
<td>R13</td>
<td>SP - Stack pointer Points to next available word</td>
</tr>
<tr>
<td>R14</td>
<td>LP - Link Pointer (return address)</td>
</tr>
<tr>
<td>R15</td>
<td>PC - program counter</td>
</tr>
</tbody>
</table>
And it almost works!

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...

```
x: .word 9

fee:
  ADD   R0, R0, R0
  ADD   R0, R0, #1
  BX    LP

main:
  LDR   R0, =x
  BL    fee
  BX    LP
```

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.

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 pointer", LP.
Callees who call themselves!

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

Oh, recursion gives me a headache.
A First Try

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     LP
main:                         MOV    R0,#10
                             BL     sqr
                             BX     LP

R4 is clobbered — We also on successive
calls. 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 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:

```c
...
{
    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.
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

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.

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

Humm… Why is that the TOP of the stack?

Reserved

"text" segment
(Program)

"stack" segment

Higher

00000008_{16}

Reserved

Lower addresses
Stack Management

**ALLOCATE k:** reserve k WORDS of stack

\[ SP = SP - 4*k \]

**DEALLOCATE k:** release k WORDS of stack

\[ SP = SP + 4*k \]

**PUSH \$x:** push Reg[x] onto stack

\[ \text{Mem}[SP - 4] = Rx \]
\[ SP = SP - 4 \]

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

\[ Rx = \text{Mem}[SP] \]
\[ SP = SP + 4 \]
Turbo Stack Instructions

Recall ARM’s block move instructions LDMFD and STMFD when used with the SP.

STMFD SP!, {r4, r7, LP}

LRMFD SP!, {r4, r7, LP}
**INTEGRATING A STACK**

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

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

```asm
sqr:
    STMFD SP!, {R4, LP}
    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, LP}
    BX     LP

main:
    MOV    R0, #10
    BL     sqr
    BX     LP
```
**Next Time**

Still some loose ends to tie up

1. More than 4 arguments
2. Addresses of arguments
3. Complex argument types

<table>
<thead>
<tr>
<th>D type:</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1110</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td>Imm12</td>
</tr>
</tbody>
</table>