12. Procedures & Stacks

6.004x Computation Structures
Part 2 – Computer Architecture

Procedures: A Software Abstraction

- Procedure: Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal parameters
  - Local storage
  - Returns control to the caller when finished
- Using multiple procedures enables abstraction and reuse
  - Compose large programs from collections of simple procedures

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

bool coprimes(int a, int b) {
  return gcd(a, b) == 1;
}

coprimes(5, 10); // false
coprimes(9, 10); // true
```

Implementing Procedures

- Option 1: Inlining
  - Compiler substitutes procedure call with body
  - Problems?
    - Code size
    - Recursion

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n - 1);
  } else {
    return 1;
  }
}
```

- Option 2: Linking
  - Produce separate code for each procedure
  - Caller evaluates input arguments, stores them and transfers control to the callee's entry point
  - Callee runs, stores result, transfers control to caller

Procedure Linkage: First Try

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n - 1);
  } else {
    return 1;
  }
}

fact(3);
```

```
fact(3) = 3*fact(2)
fact(2) = 2*fact(1)
fact(1) = 1*fact(0)
fact(0) = 1
```

- Need calling convention: Uniform way to transfer data and control between procedures
- Proposed convention:
  - Pass argument (value of n) in R1
  - Pass return address in R28
    - use `BR(fact,r28)` to call and `JMP(r28)` to return
  - Return result in R0

Procedure Linkage: First Try

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n - 1);
  } else {
    return 1;
  }
}

fact(3);
```

```
fact:
        CMPLEC(r1,0,r0)
        BT(r0,else)
        MOVE(r1,r2) // save n
        SUBC(r2,1,r1)
        BR(fact,r28)
        MUL(r0,r2,r0)
        BR(rtn)
else:   CMOVE(1,r0)
rtn:    JMP(r28)

main:   CMOVE(3,r1)
        BR(fact,r28)
        HALT()
```

OOPS!

- Proposed convention:
  - Pass argument (value of n) in R1
  - Pass return address in R28
  - Return result in R0

Procedure Storage Needs

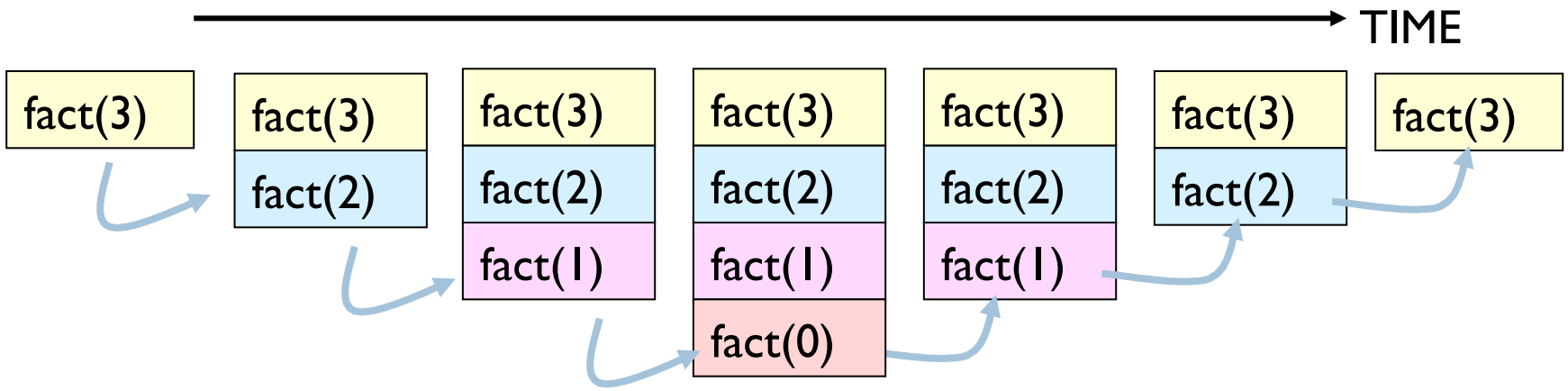
- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - Results

- Local storage:
  - Variables that compiler can't fit in registers
  - Space to save caller's register values for registers that we overwrite

Each of these is specific to a particular *activation* of a procedure. We call them the procedure's *activation record*

Activation Records

```
int fact(int n) {
  if (n > 0) return n*fact(n - 1);
  else return 1;
}
```

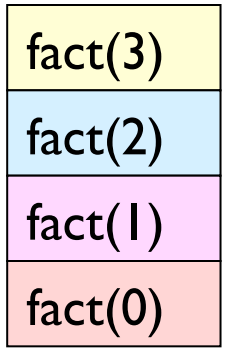


A procedure call creates a new activation record. Caller's record is preserved because we'll need it when callee finally returns.

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

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records
- Activation records are allocated and deallocated in last-in-first-out (LIFO) order
- Stack: push, pop, access to top element



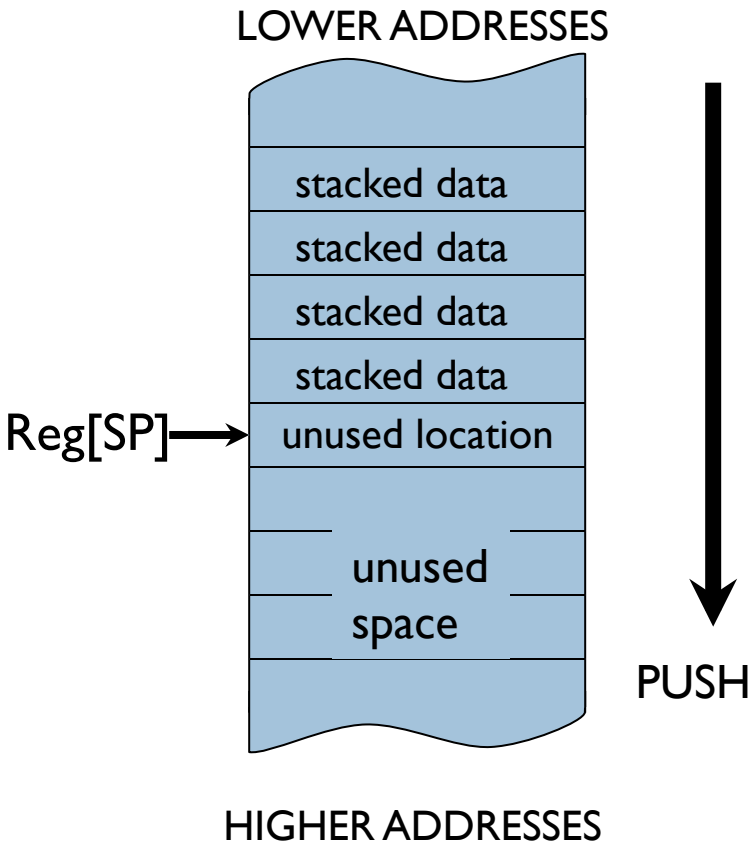
- For C, we only need to access to the activation record of the currently executing procedure

Stack Implementation

CONVENTIONS:

- Dedicate a register for the Stack Pointer (SP = R29).
- Builds *up* (towards higher addresses) on PUSH
- SP points to first UNUSED location; locations with addresses lower than SP have been previously allocated.
- Discipline: can use stack *at any time*; but leave it as you found it!
- Reserve a large block of memory well away from our program and its data

We use only *software conventions* to implement our stack (many architectures dedicate hardware)



Other possible implementations include stacks that grow "down", SP points to top of stack, etc.

Stack Management Macros

PUSH(RX): Push Reg[x] onto stack

```
Reg[SP] ← Reg[SP] + 4;
Mem[Reg[SP]-4] ← Reg[x]
```

```
ADDC(R29, 4, R29)
ST(RX,-4,R29)
```

POP(RX): Pop value on top of the stack into Reg[x]

```
Reg[x] ← Mem[Reg[SP]-4]
Reg[SP] ← Reg[SP] - 4;
```

```
LD(R29, -4, RX)
SUBC(R29,4,R29)
```

ALLOCATE(k): Reserve k WORDS of stack

```
Reg[SP] ← Reg[SP] + 4*k
```

```
ADDC(R29,4*k,R29)
```

DEALLOCATE(k): Release k WORDS of stack

```
Reg[SP] ← Reg[SP] - 4*k
```

```
SUBC(R29,4*k,R29)
```

Fun with Stacks

We can use stacks to save values we'll need later. For instance, the following code fragment can be inserted anywhere within a program.

```
        // Argh!!! I'm out of registers Scotty!!
        //
        PUSH(R0)              // Frees up R0
        PUSH(R1)              // Frees up R1
        LD(dilithum_xtals, R0)
        LD(seconds_til_explosion, R1)
suspense: SUBC(R1, 1, R1)
        BNE(R1, suspense)
        ST(R0, warp_engines)
        POP(R1)              // Restores R1
        POP(R0)              // Restores R0
```

Data is popped off the stack in the opposite order that it is pushed on

Next, we'll use show how to use stacks for activation records..

Solving Procedure Linkage Problems

Reminder: Procedure storage needs

1) We need a way to *pass arguments* to the procedure
2) Procedures need their own *LOCAL storage*
3) Procedures need to *call other procedures*; special case: recursive procedures that *call themselves*

Plan for caller:

- Push argument values onto stack *in reverse order* for use by callee
- Branch to callee, save return address in dedicated register (LP = R28)
- Clean up stack after callee return

C code:

proc(*expr*$_1$, …, *expr*$_n$)

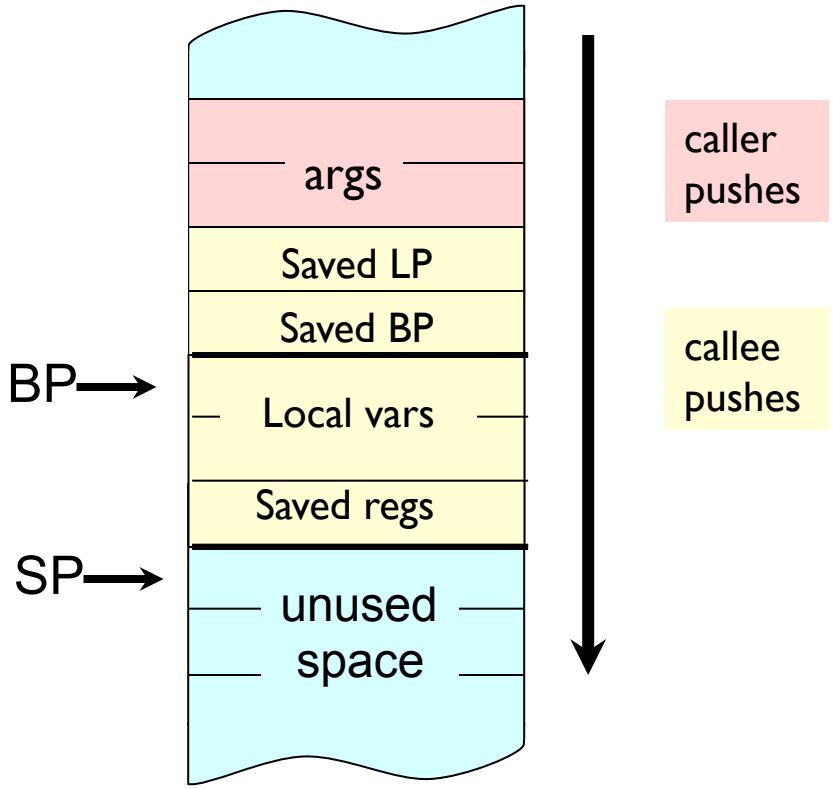
Beta assembly:

compile_expr(*expr*$_n$)⇒Rx
PUSH(rx)
…
compile_expr(*expr*$_1$)⇒Rx
PUSH(rx)
BR(*proc*,LP)
DEALLOCATE(*n*)

Stack Frames as Activation Records

CALLEE uses stack for all of the its storage needs:

1. Saving return address back to the caller (it's in LP)
2. Saving BP of caller (pointer to caller's activation record)
3. Allocating stack locations to hold local variables
4. Save any registers callee uses: "callee saves" convention



Dedicate another register (BP = R27) to hold address of the activation record. Use when accessing

- Arguments
- Other local storage

BP is a convenience

In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

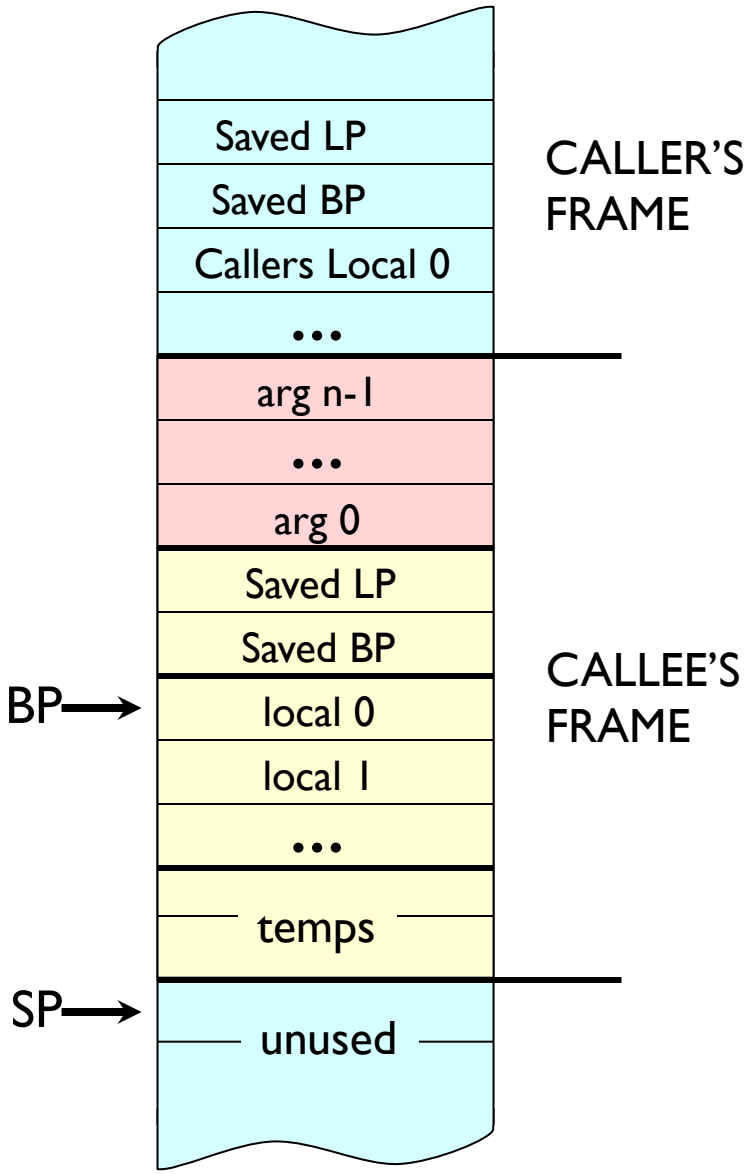
Stack Frame Details

CALLER passes arguments to CALLEE on the stack in *reverse* order

F(1,2,3,4) translates to:

```
CMOVE(4,R0)
PUSH(R0)
CMOVE(3,R0)
PUSH(R0)
CMOVE(2,R0)
PUSH(R0)
CMOVE(1,R0)
PUSH(R0)
BR(F, LP)
```

Why push args in REVERSE order?



Argument Order & BP usage

Why push args in reverse order? It allows the BP to serve double duties when accessing the local frame

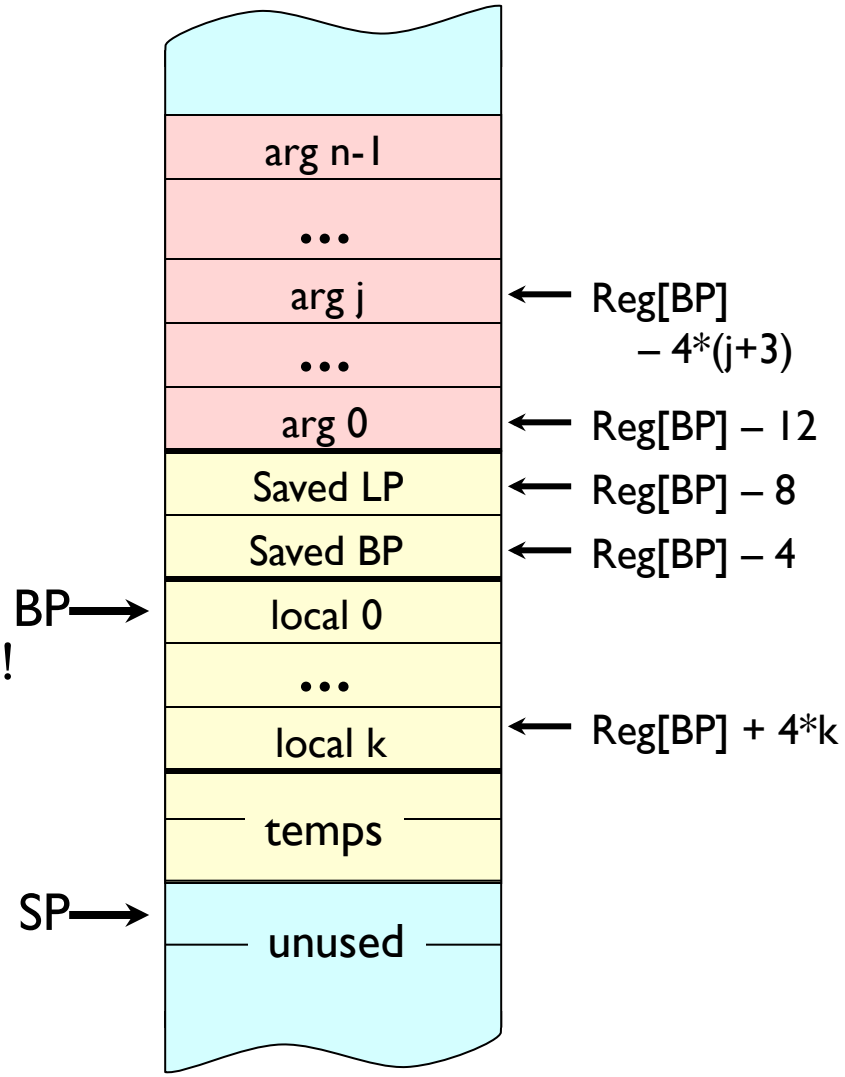
1) To access j^{th} argument ($j \geq 0$):

```
LD(BP, -4*(j+3), rx)
    or
ST(rx, -4*(j+3), BP)
```

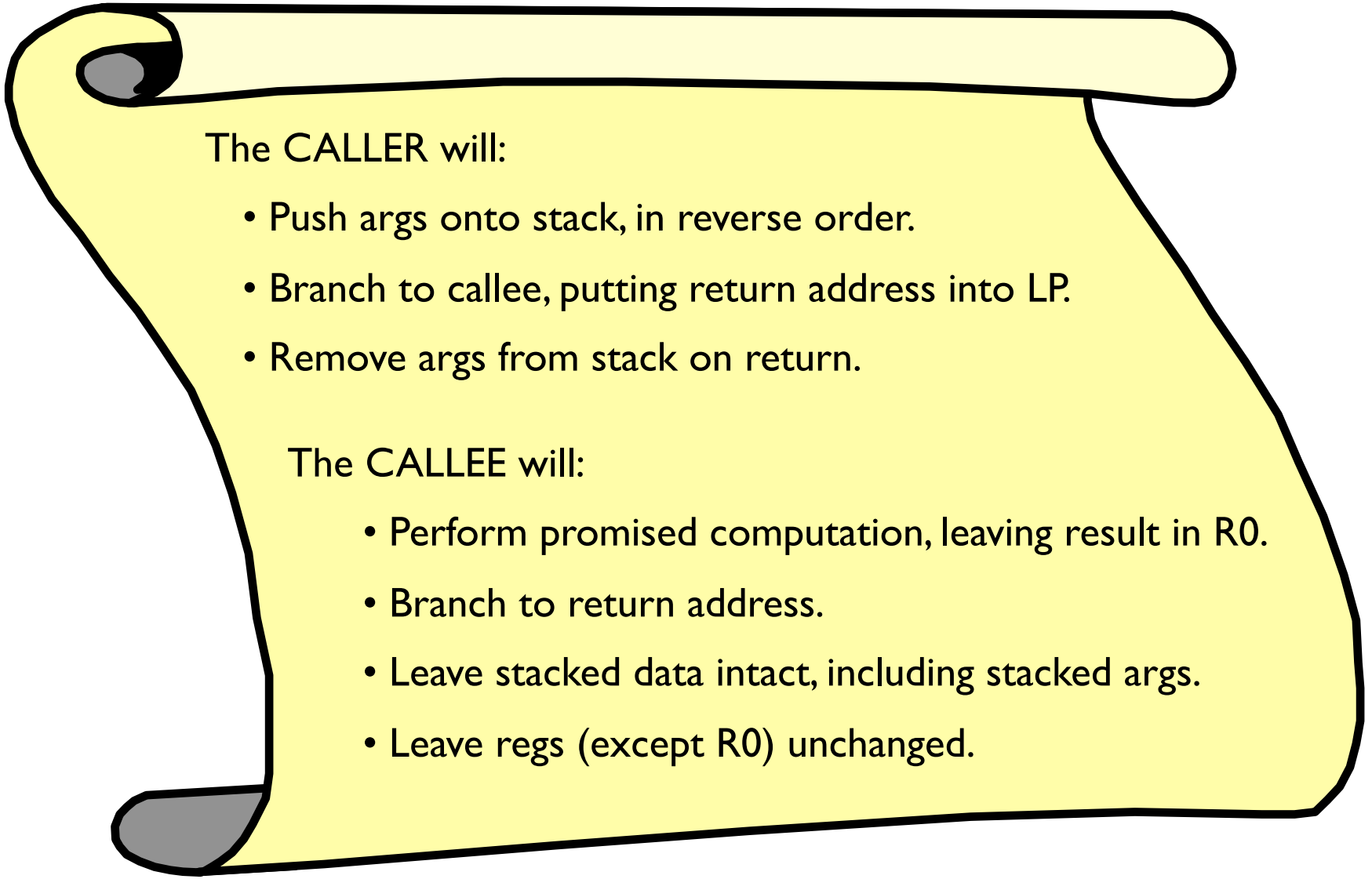
CALLEE can access the first few arguments without knowing how many arguments have been passed!

2) To access k^{th} local variable ($k \geq 0$)

```
LD(BP, k*4, rx)
    or
ST(rx, k*4, BP)
```



Procedure Linkage: The Contract



The CALLER will:

- Push args onto stack, in reverse order.
- Branch to callee, putting return address into LP.
- Remove args from stack on return.

The CALLEE will:

- Perform promised computation, leaving result in R0.
- Branch to return address.
- Leave stacked data intact, including stacked args.
- Leave regs (except R0) unchanged.

Procedure Linkage

typical "boilerplate" templates

Calling Sequence

```
PUSH(arg_n)                 // push args, last arg first
...
PUSH(arg_1)
BR(f, LP)                   // Call f.
DEALLOCATE(n)               // Clean up!
...                         // (f's return value in r0)
```

Entry Sequence

```
f:  PUSH(LP)                // Save LP and BP
    PUSH(BP)                // in case we make new calls.
    MOVE(SP,BP)             // set BP=frame base
    ALLOCATE(nlocals)       // allocate locals
    (push other regs)       // preserve any regs used
```

Exit Sequence

```
// return value in R0…
(pop other regs)            // restore regs
MOVE(BP,SP)                 // strip locals, etc
POP(BP)                     // restore CALLER's linkage
POP(LP)                     // (the return address)
JMP(LP)                     // return.
```

Why no DEALLOCATE?

Putting It All Together: Factorial

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n-1);
  } else {
    return 1;
  }
}
```

```
fact:   PUSH(LP)          // save linkages
        PUSH(BP)
        MOVE(SP,BP)       // new frame base
        PUSH(r1)          // preserve regs
        LD(BP,-12,r1)     // r1 ← n
        CMPLEC(r1,0,r0)   // if (n > 0)
        BT(r0,else)

        SUBC(r1,1,r1)     // r1 ← (n-1)
        PUSH(r1)          // push arg1
        BR(fact,LP)       // fact(n-1)
        DEALLOCATE(1)     // pop arg1
        LD(BP,-12,r1)     // r1 ← n
        MUL(r1,r0,r0)     // r0 ← n*fact(n-1)
        BR(rtn)

else:   CMOVE(1,r0)       // return 1

rtn:    POP(r1)           // restore regs
        MOVE(BP,SP)       // Why?
        POP(BP)           // restore links
        POP(LP)
        JMP(LP)           // return
```

Recursion?

Of course!

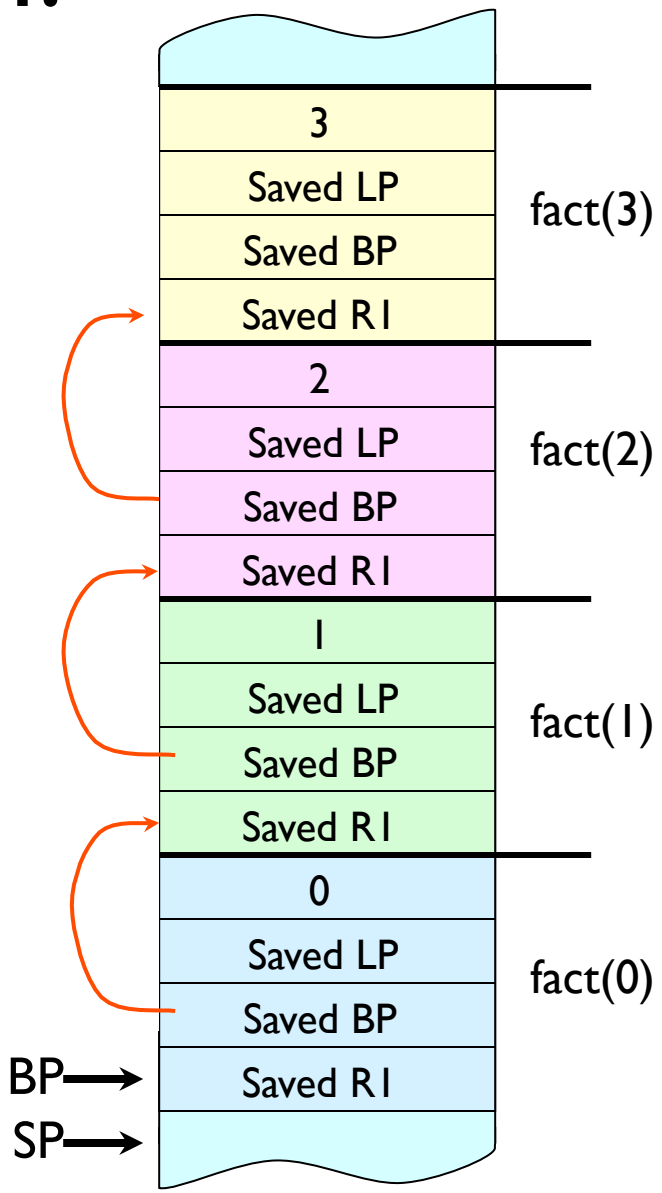
- Frames allocated for each recursive call…
- Deallocated (in inverse order) as recursive calls return

Debugging skill:

"stack crawling"

- Given code, stack snapshot – figure out what, where, how, who…
- Follow old <BP> links to parse frames
- Decode args, locals, return locations, etc etc etc

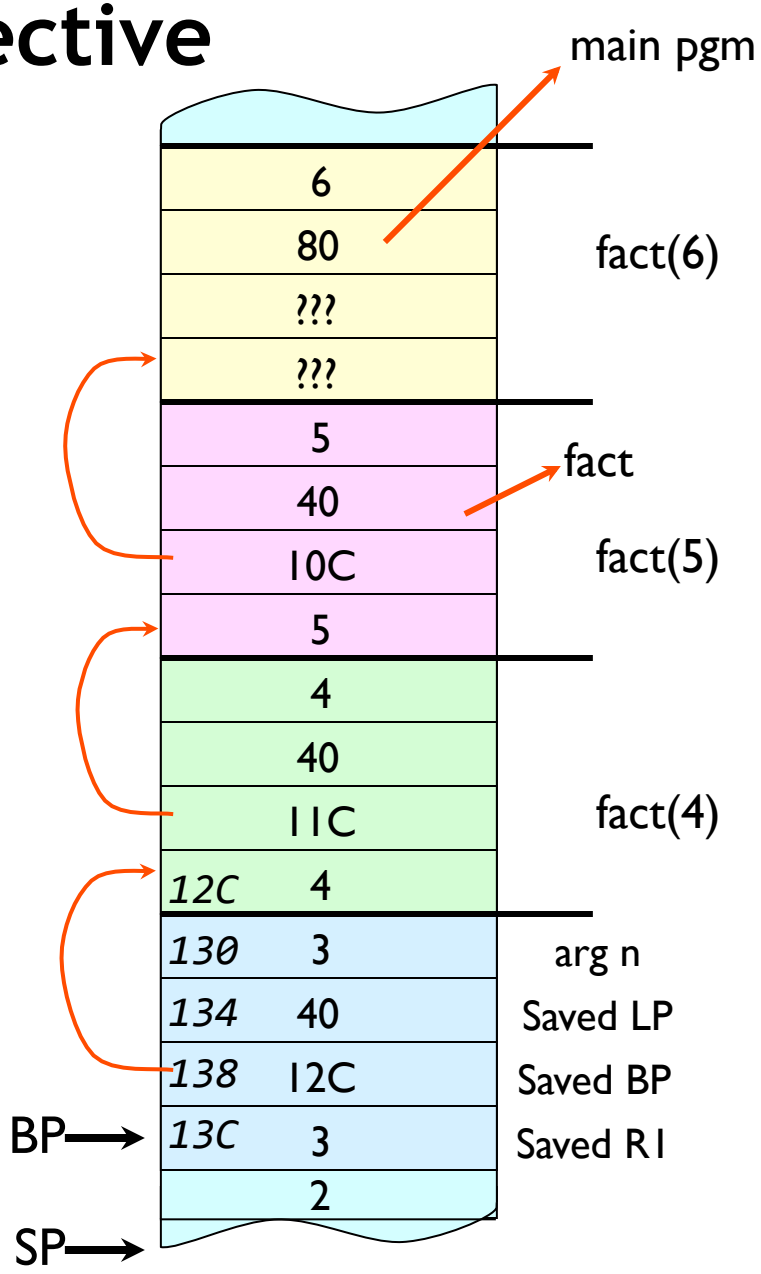
Particularly useful on 6.004 quizzes!



Stack Detective

`fact(n)` is called. During the calculation, the computer is stopped with the PC at 0x40; the stack contents are shown (in hex).

- What's the argument to the *active* call to fact? 3
- What's the argument to the *original* call to fact? 6
- What's the location of the original calling (BR) instruction? 80 – 4 = 7C
- What instruction is about to be executed? DEALLOCATE(1)
- What value is in BP? 13C
- What value is in SP? 13C+4+4=144
- What value is in R0? fact(2) = 2



Summary of Dedicated Registers

The Beta ISA has 32 registers. But we've dedicated several of them to serve a specific purpose:

- R31 is always zero [ISA]
- R30 … reserved for future use… [next lecture]
- R29 = SP, stack pointer [software convention]
- R28 = LP, linkage pointer [software convention]
- R27 = BP, base pointer [software convention]

Summary

- Each procedure invocation has an activation record
  - Created during procedure call/entry sequence
  - Discarded when procedure returns
  - Holds:
    - Argument values (in reverse order)
    - Saved LP, BP from caller (callee reuses those regs)
    - Storage for local variables (if any)
    - Other saved regs from caller (callee needs regs to use)
  - BP points to activation record of active call
    - Access arguments at offsets of -12, -16, -20, ..
    - Access local variables at offsets of 0, 4, 8, ...
- "Callee saves" convention: all reg values preserved
- Except for R0, which holds return value