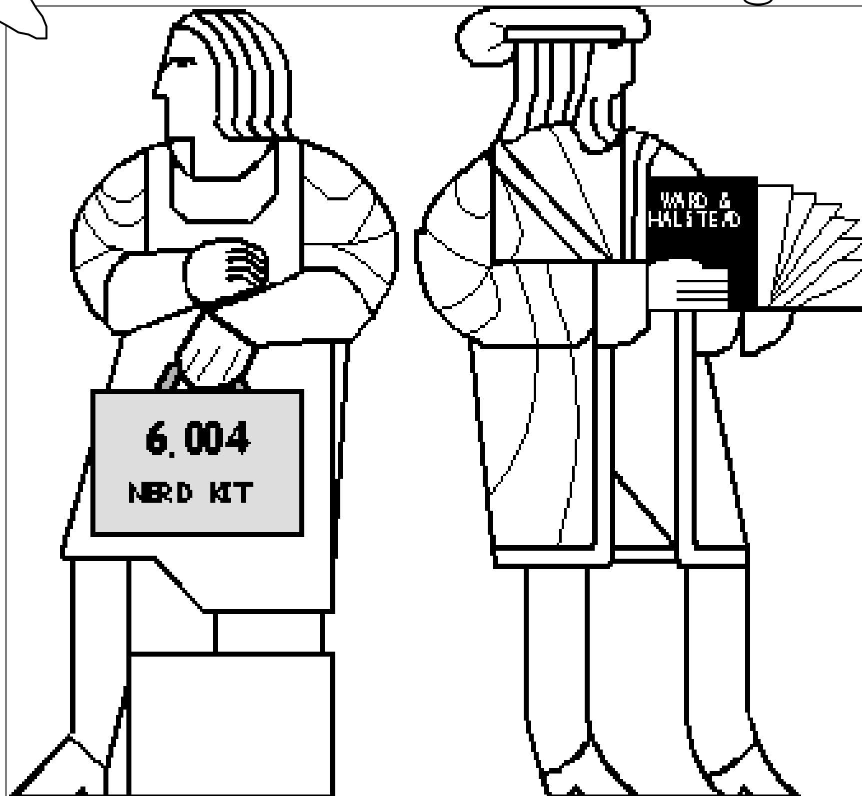
Processor Architecture: Stacks and Stack Frames



Source Language: Constructs to be Compiled

Variables - declaration, access:

```
int x, y, z, a[100];
     ...
     z = x + y;
```

Procedures - call, return

```
int f()
 {   z = x + y;
 }
```

Block structure: arguments, locals

```
int f(n)
 {   int a = 2*n-1;
     if (n>1)
       a = a+f(n-1);
     return a;
 }
```

Common problem: ____________________

REVIEW: Static Allocation

Given

```
int x, y, z;
      ...
      z = x + y;
```

Translate as follows:

1. Assign fixed (constant) addresses to variables x, y, z

2. Translate each variable reference to a direct read/write of the assigned memory location.

Storage allocated at ____________________ time!

Code Generation using Static Allocation

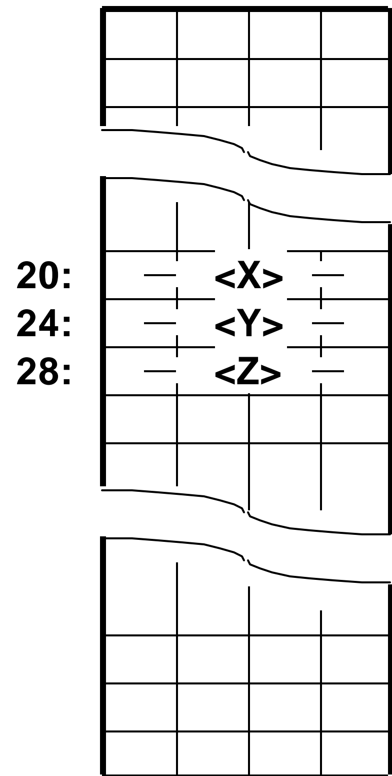
Memory:

Program:

```
int  x,  y,  z;
z  =  x + y;
```

might translate to:

```
LD(20,r0)
LD(24,r1)
ADD(r0,r1)
ST(r1,28)
```



28¬ <20> + <24>

STATIC allocations of program variables:

- x, y, z bound to locations 20, 24, 28 at compile time;
- "x" always translates to <20>, etc.

Static Allocation: Pros & Cons

> Static Allocation:
> Address of each variable is a *compile-time constant.*

Good things:

- **SIMPLE** code-generation discipline
- **Fast**, direct access to variables: build addresses into instruction stream.

Bad things:

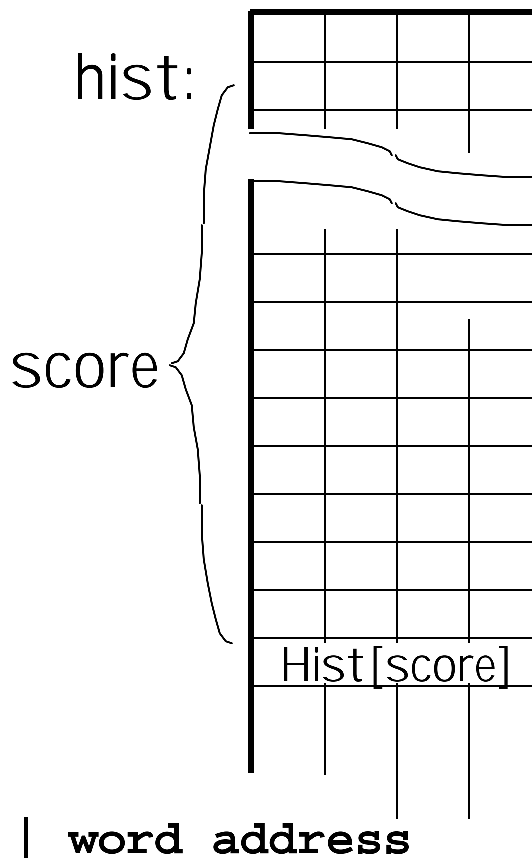
- **Not modular** (allocation is *global*)
- **Total storage bounded** at compile time; not suited to dynamically-varying storage needs.

Complication: aggregate data structures

Aggregate Storage I: Arrays

The C source code

Memory:




```
int Hist[100];
   ...
   Hist[score] =
     1+Hist[score];
```

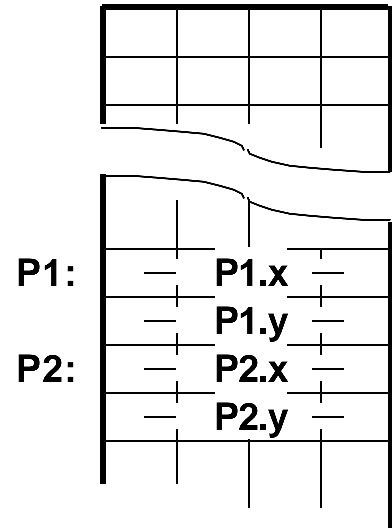
might translate to:

```
hist:  .=.+4*100
       ...
       <score in r1>
       MULC(r1,4,r2)     | word address
       LD(hist,r2,r0)    | hist[score]
       ADDC(r0,1,r0)     | increment
       ST(r0,hist,r2)    | hist[score]
```

Address: CONSTANT base address
+ VARIABLE offset from index.

Aggregate Storage II:
Records (a.k.a. Structures)

```
struct Point
  { int x, y;
  } P1, P2, *p;
   ...
   P1.x = 157;
   ...
   p = &P1;
   p->y = 157;
```



might translate to:

```
P1: .=.+8
P2: .=.+8
x=0              | Offset for x component
y=4              | Offset for y component
   ...
   ADDC(r31,157,r0)    | r0 <- 157
   ST(r0,P1+x)         | P1.x = 157
   ...
   <p in r3>
   ST(r0,y,r3)         | p->y = 157;
```

Address: VARIABLE base address
+ CONSTANT offset from element.

Other Storage Needs

Procedure/Function Stuff:

- Arguments
  `f(x,y,z)` or worse ... **`sin(a+b)`**
- Return Address in Calling routine
- Results passed back to caller.

Temporary Storage:

intermediate results during expression evaluation.
`(a+b) * (c+d)`

Local variables:

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

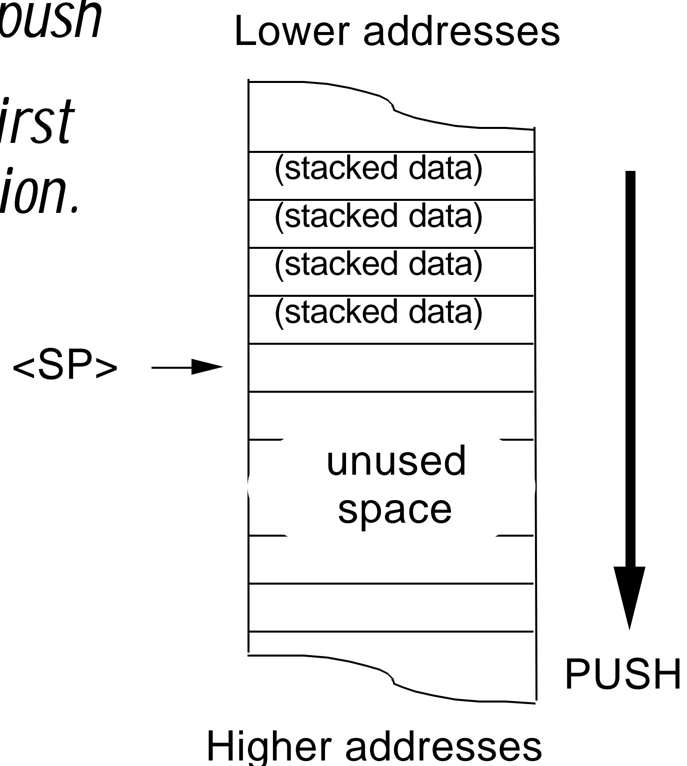
Each of these is specific to an *activation* of a procedure; collectively, they may be viewed as an *activation record.*

Stack Implementation

one of several possible implementations

Conventions:

- *Builds UP on push*
- *SP points to first UNUSED location.*



To push <x>:

  SP ¬ <SP> + 4;

  <SP>-4 ¬ <x>

To pop() a value into x:

  x ¬ <<SP>-4>

  SP ¬ <SP> - 4;

Stack Frames: Our (SOFTWARE) convention

RESERVED REGISTERS:

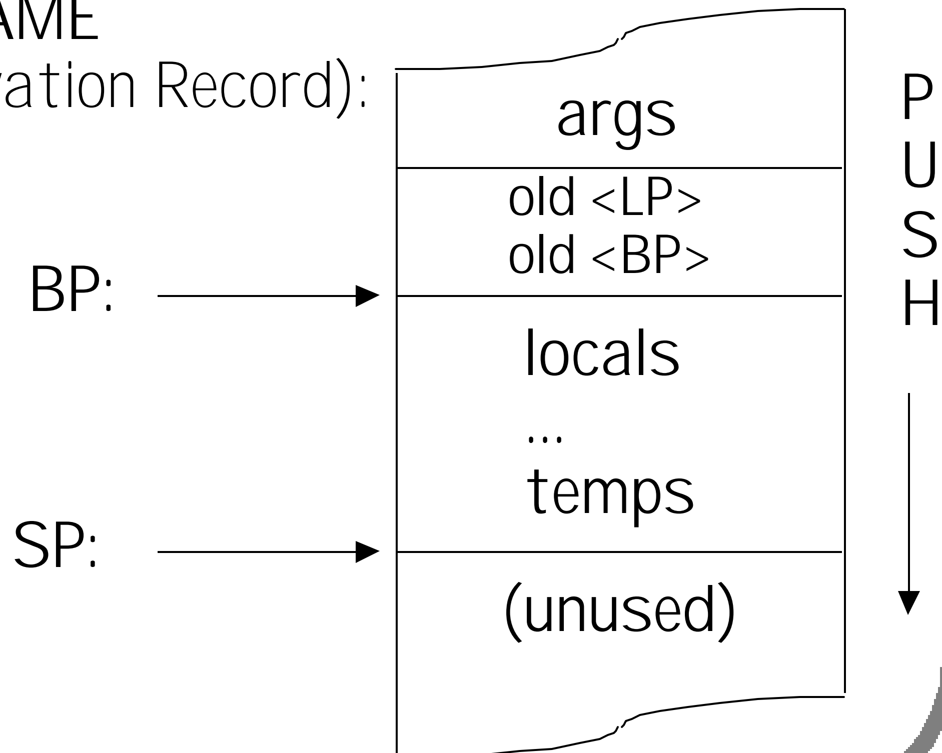
r27 = **BP**. Base ptr, points to 1st local.

r28 = **LP**. Linkage pointer, saved <PC>.

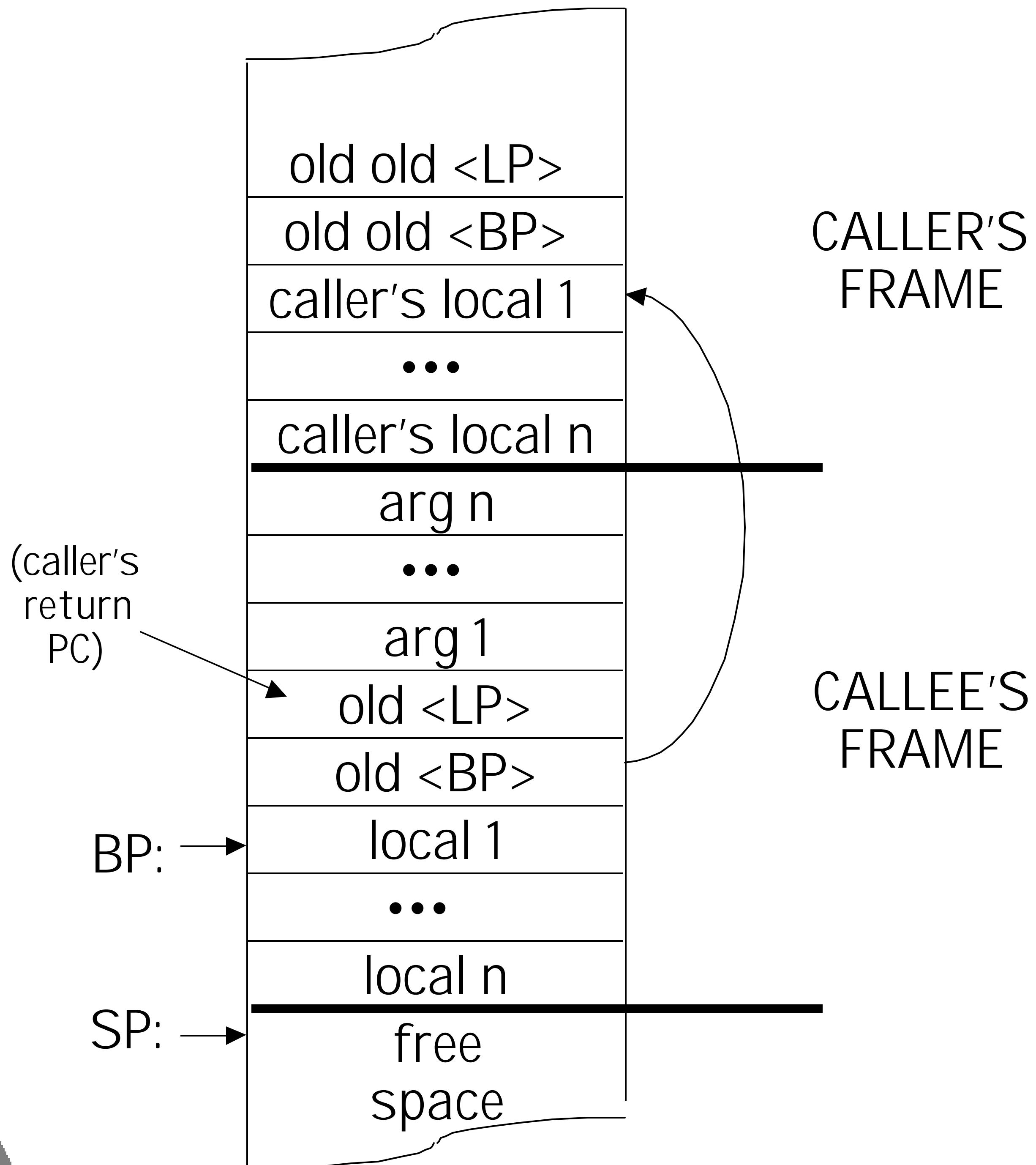
r29 = **SP**. Stack ptr, points to 1st unused word.

r30 = **XP**. Exception pointer, saved <PC>

STACK FRAME
(aka Activation Record):



Stack Frame Detail



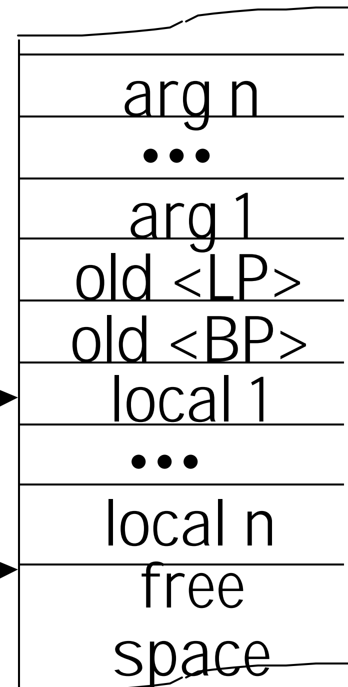
Access to Local Frame

To access jth local variable ($j \geq 1$):

```
LD(BP, (j-1)*4, rx)
```

or

```
ST(rx, (j-1)*4, BP)
```



To access jth argument ($j \geq 1$):

```
LD(BP, -4*(j+2), rx)
```

or

ST(rx, -4*(j+2), BP)

QUESTION: Why push args in REVERSE order???

Stack Management: some handy macros

PUSH(rx) – pushes 32-bit value onto stack.

```
ADDC(SP,4,SP)
ST(rx,-4,SP)
```

POP(rx) – pops 32-bit value into rx.

```
LD(SP,-4,rx)
ADDC(SP,-4,SP)
```

ALLOCATE(k) – reserve k WORDS of stack

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

DEALLOCATE(k) – give back k WORDS

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

Procedure Linkage: The Contract

The ______________ will:

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

The _________________ will:

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

Procedure Linkage

Calling Sequence:

```
        PUSH(arg_n)             | push args, in
        ...                     | RIGHT-TO-LEFT
        PUSH(arg_1)             | order!
        BR(f, LP)               | Call f.
        DEALLOCATE(n)           | Clean up!
        ...                     | (value in r0)
```

Entry Sequence:

```
f:      PUSH(LP)                | Save <LP>, <BP>
        PUSH(BP)                |   for new calls.
        MOVE(SP,BP)             | set BP=frame base
        ALLOCATE(locals)        | allocate locals
        (push other regs)       | preserve regs used
        ...
```

Return Sequence:

```
        (pop other regs)        | restore regs
        MOVE(val, R0)           | set return value
        MOVE(BP,SP)             | strip locals, etc
        POP(BP)                 | restore linkage
        POP(LP)                 | (the return <PC>)
        JMP(LP)                 | return.
```

See if you recognize THIS...

```
| int fact(n)
|  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
       BNE(r1, big)       | if n>0
       ADDC(r31,1,r0)     | else return 1;
       BR(rtn)

big:   SUBC(r1,1,r1)      | r1 ¬  (n-1)
       PUSH(r1)           | arg1
       BR(fact, LP)       | fact(n-1)
       DEALLOCATE(1)      | (pop arg)
       LD(BP, -12, r1)    | r0 ¬  n
       MUL(r1,r0,r0)      | r0 ¬  n*fact(n-1)

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

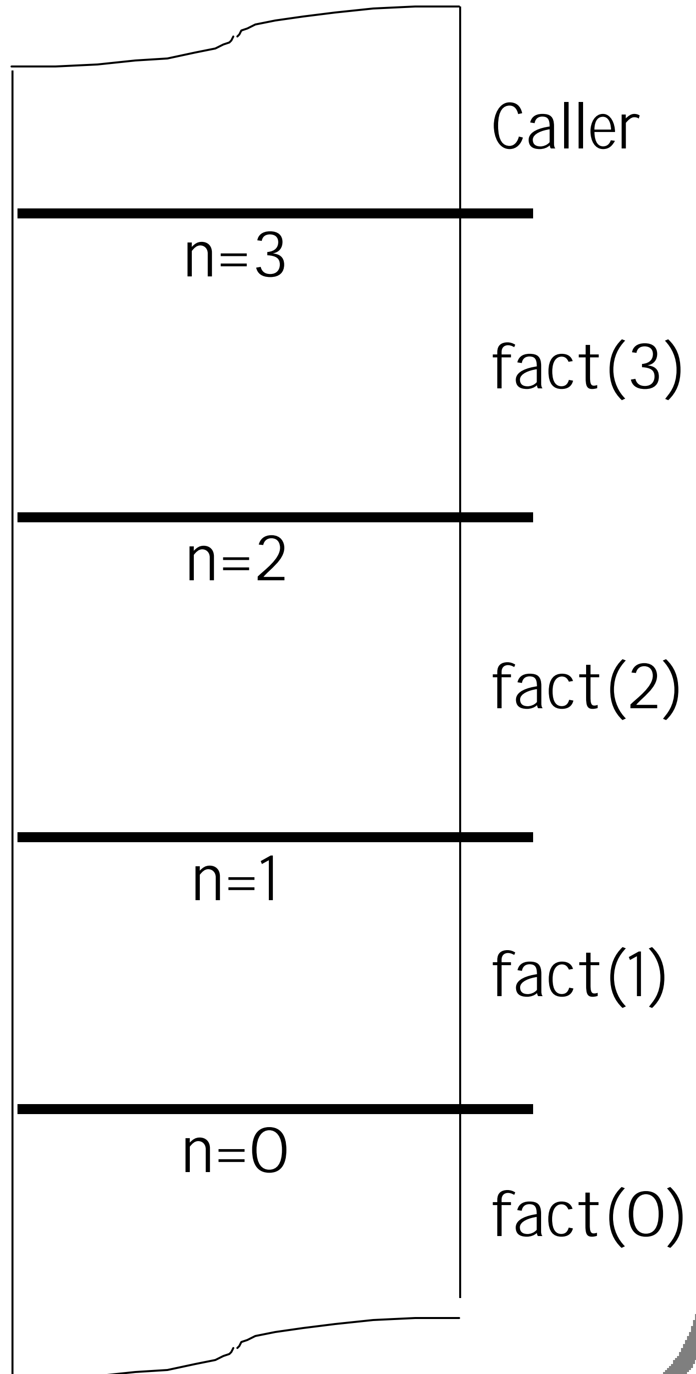
Recursion...

fact(3) ...

lower
addresses

STACK
BUILDS
THIS
WAY

higher
addresses



Other uses of processor stack:

Nested calls: f⇒ g ⇒ h

OBEYS Stack Discipline:

each call leaves stack as it found it.

Temporaries: Z = SQRT(X)+SQRT(Y);

push[SQRT(X)];
Z ← SQRT(Y);
Z ← <Z> + pop[];

Preserving (& restoring) processor state:

push[R1];
...
(use R1)
...
R1 ← pop[];

e.g., during execution of procedure body.

Storage of DYNAMICALLY ALLOCATED variables...

... giving rise to ______________________________.

Man vs Machine:

a programming challenge

Here's a C program which was fed to the C compiler*. Can you generate code as good as it did?

```
int
ack (int i, int j)
{ if (i == 0) return 2*j;
  if (j == 0) return i+1;
  return ack (i-1, ack (i, j-1));
}
```

* GCC Port courtesy of Cotton Seed & Pat LoPresti; available on Athena

```
Athena% attach 6.004
Athena% gcc-beta -S -O2 file.c
```

Traps, Exceptions, and Interrupts

"implicit" procedure calls

Traps: calls to Operating System

SVC(2) = WRCHAR()

write a character on console display.

EFFECT: **BR(H_SVC, xp)**

Faults: Handlers for exceptions & errors

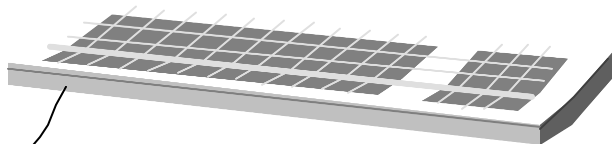
SUBC(r31, 1, r1)
JMP(r1)

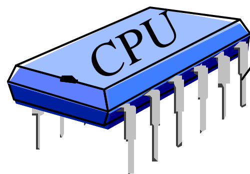
EFFECT: **BR(H_MEMERR, xp)**

Addressing Error at FFFFFFFF
You Lose, Bunky!

Interrupts: Handlers for I/O events

Interrupt REQ line raised by KBD





EFFECT:
BR(H_KBD, xp)

Coding of Handlers

(early glimpse)

H_SVC: Called by SVC(n)

- Fetch SVC instr via XP pointer
- Decode SVC, perform function
- JMP(XP) to return

H_XXXERR: Invoked by BR(H_XXXERR, xp)

- Fetch faulting instr, diagnose.

EITHER

- Fix, return via JMP(XP)
- Report unrecoverable error

H_KBD: Invoked by BR(H_KBD, xp)

- Service I/O device (read char, stuff into buffer)
- Return via JMP(XP)

NB: Unlike SVCs, faults & interrupts are

______________________________!

Tough Problems

1. NON-LOCAL variable access, particularly in nested procedure *definitions*.

   "FUNARG" problem of LISP.

   Conventional solution: "static links" in stack frames, pointing to frames of statically enclosing blocks. This allows a run-time discipline which correctly accesses variables in enclosing blocks.
   ANALOG: LISP Environments, closures.
   [Optional reading: text 14.8, p. 400]

   (C avoids this problem by outlawing nested procedure declarations!)

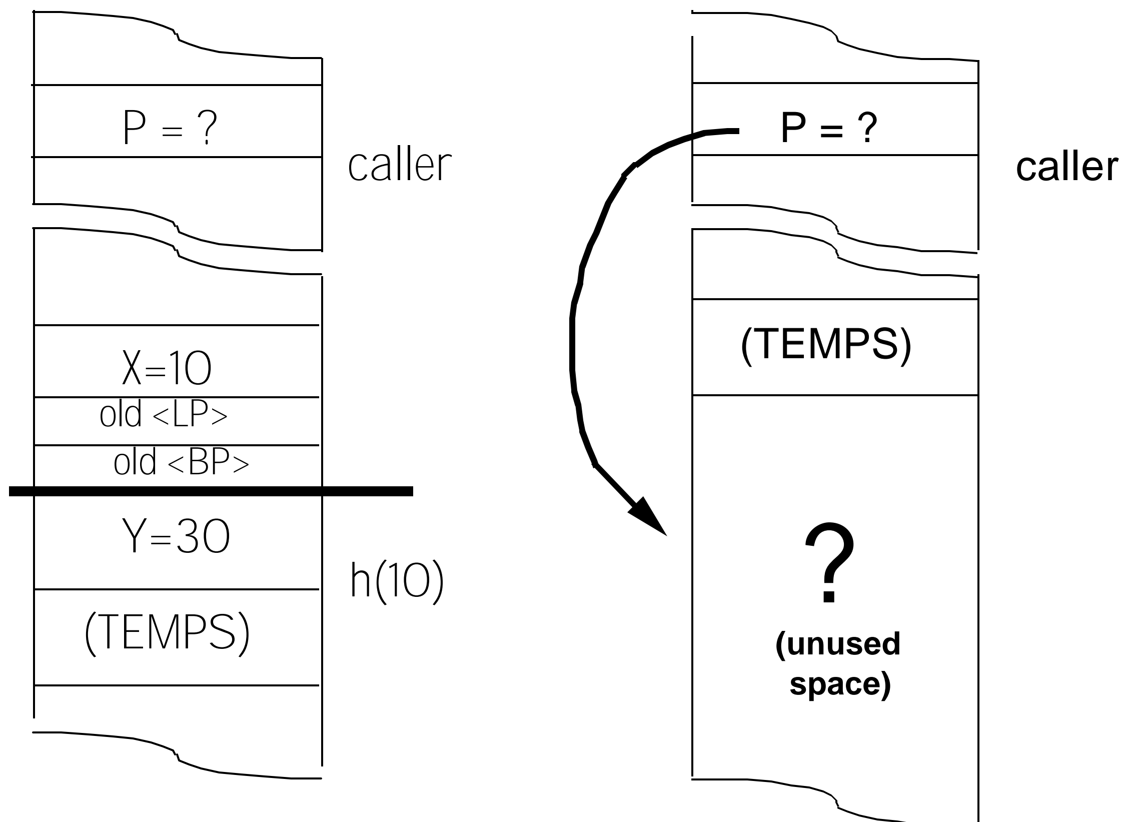
2. "Dangling References" - - -

Dangling References

```
int *p;                     /* POINTER */

int h(x)
 { int y = x*3;
   p = &y;                  /* adr of y */
   return 37;
 }

h(10);
print(*p);
```



_______________________ WILL HAPPEN!

Clever Ways Around the Dangling Reference Problem

PASCAL: kiddie scissors only.

No "ADDRESS OF" operator: language restrictions *forbid* constructs which could lead to dangling references.

C: real tools, real dangers.

"you get what you deserve".

SCHEME/LISP: throw cycles at it.

Activation records allocated from a HEAP, reclaimed transparently by garbage collector (at considerable cost).

"You get what you pay for"

Of course, there's a stack hiding there somewhere..*

* **you get what you deserve.**

COMING ATTRACTIONS:

NEXT WEEK: Building a Beta

```
ack:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        PUSH (R1)
        PUSH (R2)
        LD (BP, -12, R2)
        LD (BP, -16, R0)
_36:    BNE (R2, _34)
        SHLC (R0, 1, R0)
        BR (_37)
_34:    BEQ (R0, _35)
        SUBC (R2, 1, R1)
        SUBC (R0, 1, R0)
        PUSH (R0)
        PUSH (R2)
        BR (ack, LP)
        MOVE (R1, R2)
        SUBC (SP, 8, SP)
        BR (_36)
_35:    ADDC (R2, 1, R0)
_37:    POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)
```