15-213

"The course that gives CMU its Zip!"

Machine-Level Programming III: Procedures Sept. 15, 1998

Topics

- Stack-based languages
- Stack frames
- Register saving conventions
- Optimizing to avoid stack frame
- Creating pointers to local variables

Procedure Control Flow

Maintain Return Address in Designated Register (`$26`)

Procedure call:

- **`bsr $26, label`** Save return addr in **`$26`**, branch to ***`label`***
- **`jsr $26, (Ra)`** Save return addr in **`$26`**, jump to address in **`Ra`**

Return Address Value

- **Address of calling instruction + 4**

```
0x100: bsr $26, label
0x104: addq $0, $16, $0   # Following instruction
```

- **Example `bsr` will set `$26` to `0x104`**

Procedure return:

- **`ret $31, ($26)`** Jump to address in **`$26`**

Call & Return Code

C Code

```
long int caller()
{
  • • •
  return callee();
}

long int callee()
{
  return 5L;
}
```

Annotated Assembly

```
caller:
          • • •
# save return addr (0x804) in $26
# branch to callee (0x918)
0x800     bsr $26,callee
          • • •
callee:
# return value = 5
0x918     bis $31,5,$0
# jump to addr in $26
0x91c     ret $31,($26),1
```

For Given Procedure Call

- Procedure in which call is made is the "*Caller*"
- Procedure being called is the "*Callee*"

Stack-Based Languages

Languages that Support Recursion

- **e.g., C, Pascal, Java**
- **Code must be "*Reentrant*"**
  - Multiple simultaneous instantiations of single procedure
- **Need some place to store state of each instantiation**
  - Arguments
  - Local variables
  - Return pointer

Stack Discipline

- **State for given procedure needed for limited time**
  - From when called to when return
- **Callee returns before caller does**

Stack Allocated in *Frames*

- **state for single procedure instantiation**

Call Chain Example

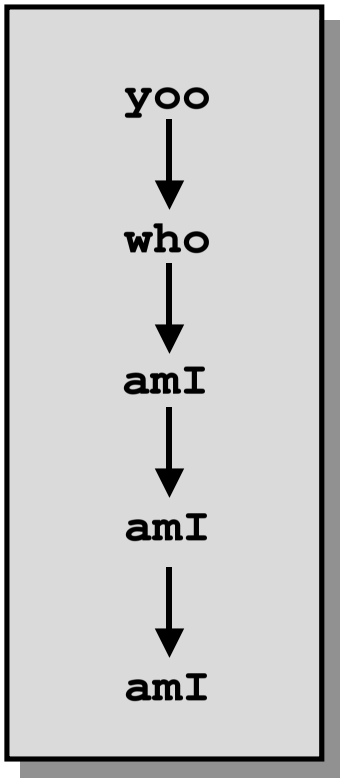
Code Structure

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    •
    •
    amI();
    •
    •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain



- Procedure `amI` recursive

Alpha Stack Structure

Stack Growth

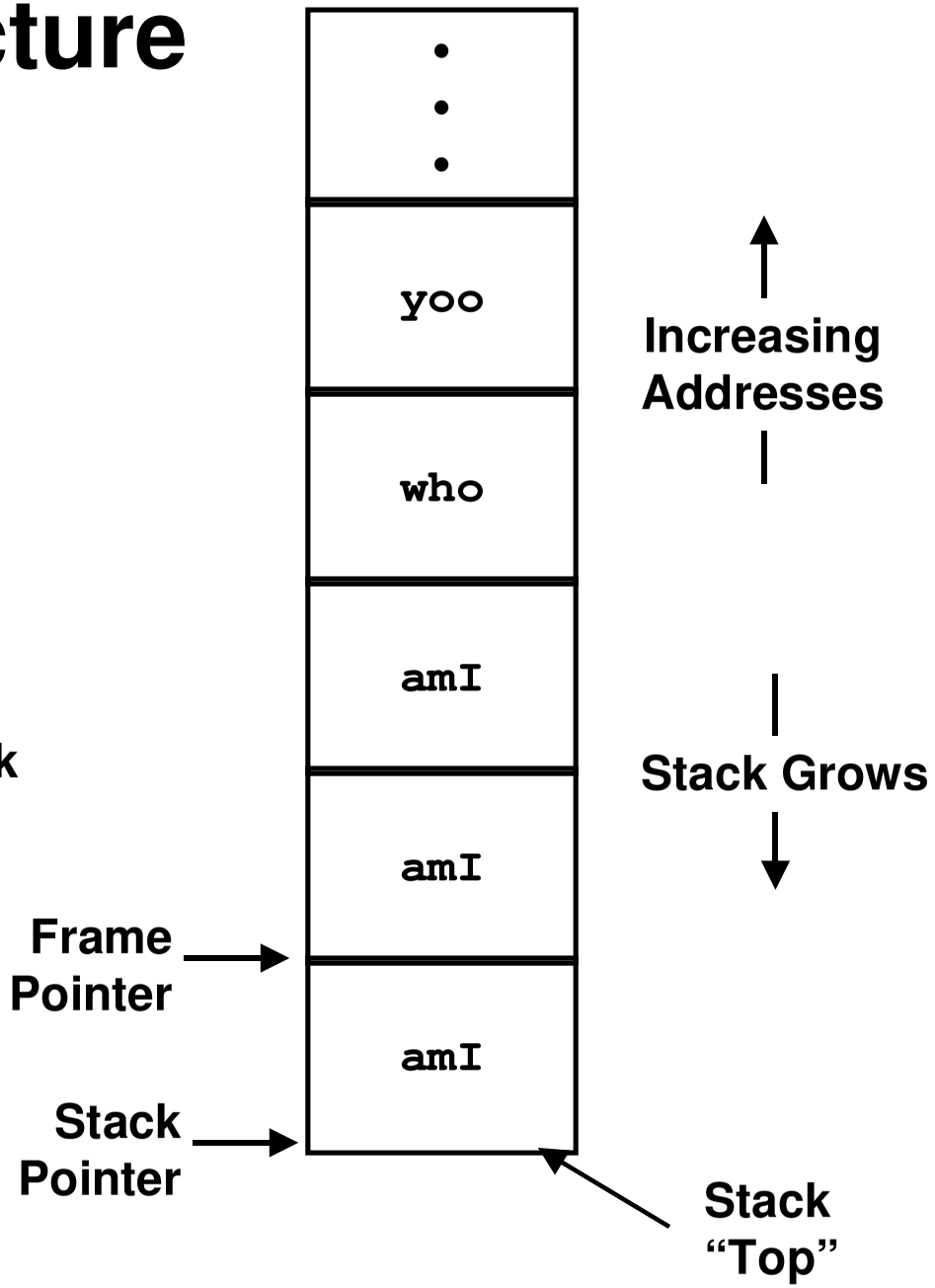
- **Toward lower addresses**
- **Drawn with "top" at bottom**

Stack Pointer

- **Address of top element on stack**
- **Use register $30**

Frame Pointer

- **Top element from caller stack frame**
- **Usually "virtual"**
  - Do not allocate explicit register
  - Computed as offset relative to stack pointer



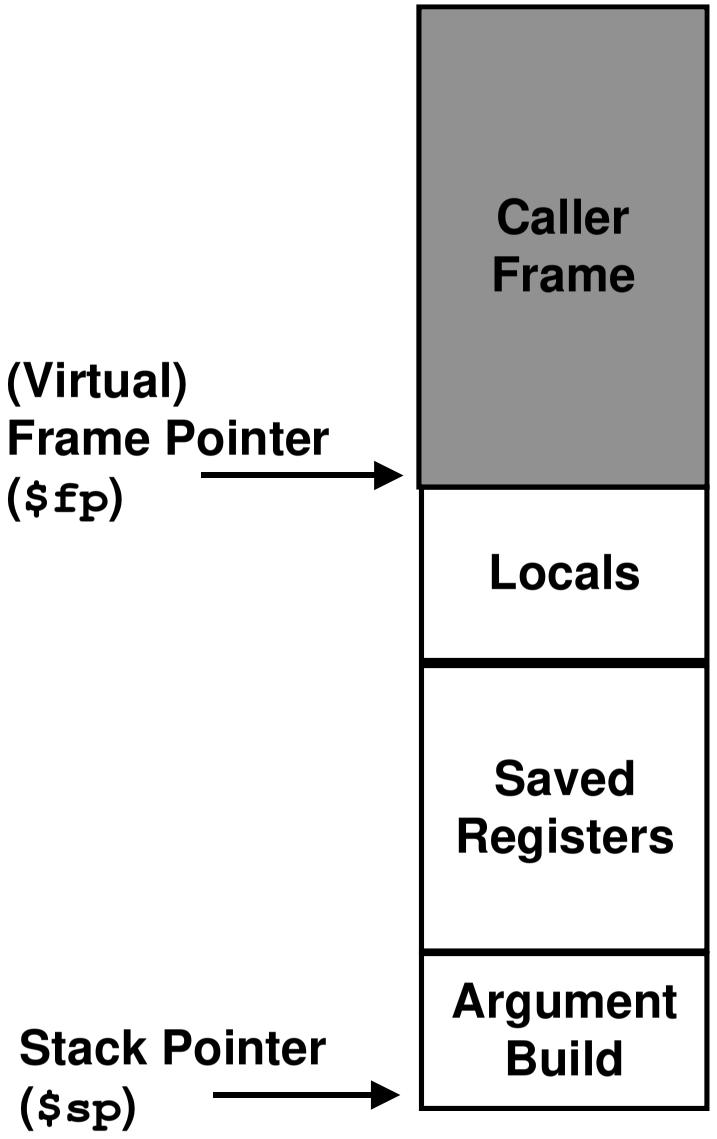
Alpha Stack Frame

Stack Frame ("Top" to Bottom)

- **Parameters to called functions**
  - Only if too many for registers
- **Saved register context**
- **Storage for local variables**

Size Requirements

- **Saved state area must be multiple of 16 bytes**
- **Local area must be multiple of 16 bytes**



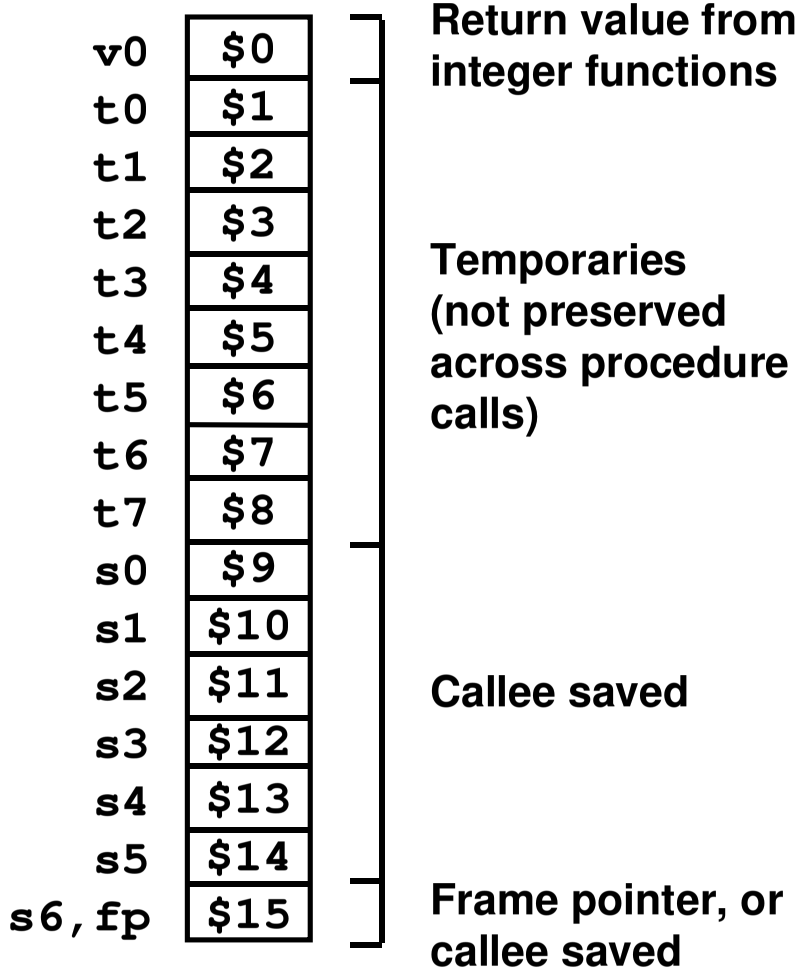
Alpha Register Convention

General Purpose Registers

- 32 total
- Store integers and pointers
- Fast access: 2 reads, 1 write in single cycle

Usage Conventions

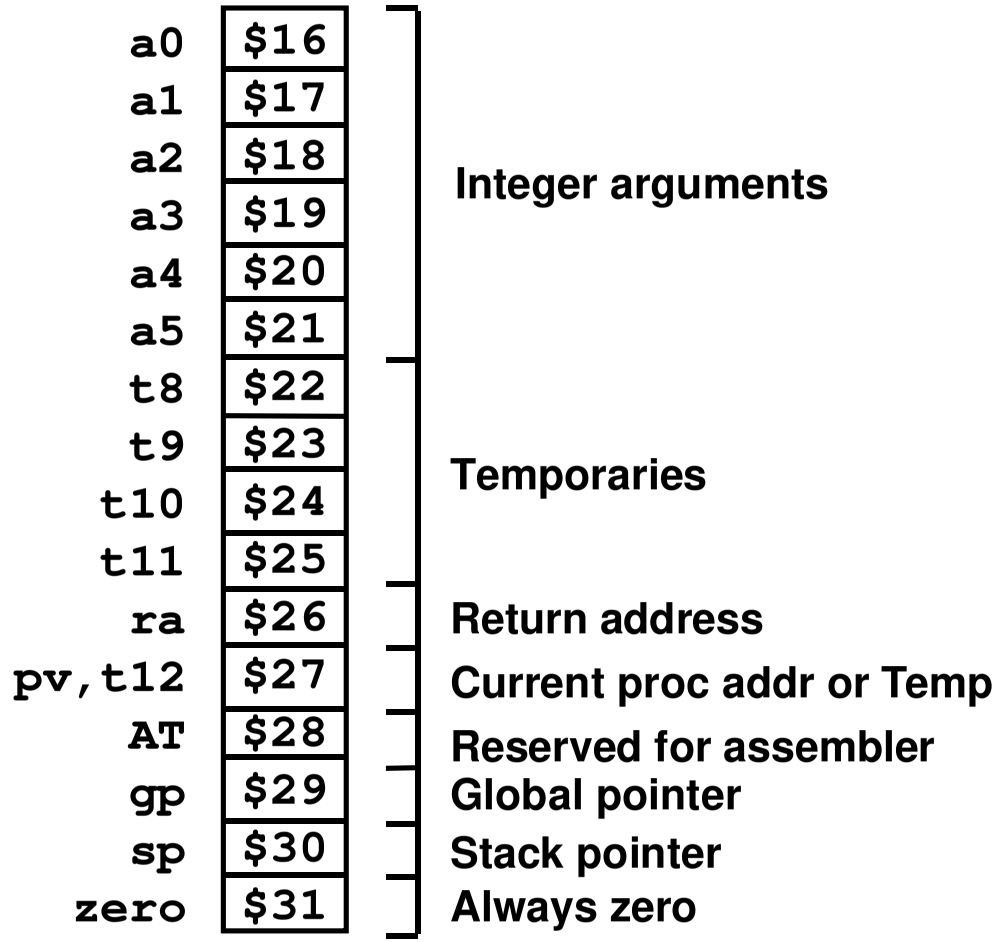
- Established as part of architecture
- Used by all compilers, programs, and libraries



Registers (cont.)

Important Ones for Now

- `$0` **Return Value**
  - Also used as temp.
- `$1` **Temporary**
  - Unsafe
- `$9...$11` **Callee Save**
  - "Safe" temporaries
- `$16...$17` **Arguments**
- `$26` **Return address**
- `$29` **Global Pointer**
  - Access to global vars.
- `$30` **Stack Pointer**
- `$31` **Constant 0**



Register Saving Conventions

When procedure `yoo` calls `who`:

- **`yoo` is the *caller*, `who` is the *callee***

Can Register be Used for Temporary Storage?

```
yoo:
    bis $31, 17, $1
    bsr $26, who
    addq $1, $0, $0
    ret $31, ($26)
```

```
who:
    bis $31, 255, $1
    s4addq $1, $1, $0
    ret $31, ($26)
```

- **Contents of registers `$1` and `$26` overwritten by `who`**

Conventions

- **"Caller Save"**
  - Caller saves temporary in its frame before calling
- **"Callee Save"**
  - Callee saves temporary in its frame before using

Caller Save Version

"Caller Save" Temporary Registers:

- not guaranteed to be preserved across procedure calls
- can be immediately overwritten by a procedure without first saving
- if `yoo` wants to preserve a caller-save register across a call to `who`:
  - save it on the stack before calling `who`
  - restore after `who` returns

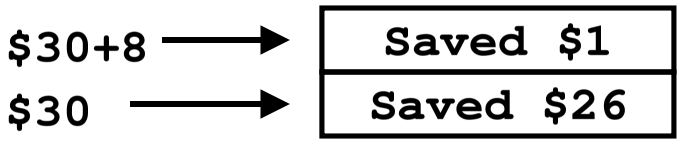
```
yoo:
    • • • # make frame
    bis $31, 17, $1
    # save $26, $1
    stq $26, 0($sp)
    stq $1, 8($sp)
    bsr $26, who
    # restore $26, $1
    ldq $1, 8($sp)
    ldq $26, 0($sp)
    addq $1, $0, $0
    • • • # remove frame
    ret $31, ($26)
```

```
who:
    bis $31, 255, $1
    s4addq $1, $1, $0
    ret $31, ($26)
```

Stack Frame for Caller Save Version

Stack Frame for Procedure `yoo`

- 16 bytes total



- Allocated by decrementing stack pointer

  `lda $sp, -16($sp)`

- Deallocated by incrementing stack pointer

  `addq $sp, 16($sp)`

```
yoo:
    # make frame
    lda $sp, -16($sp)

    . . .

    # remove frame
    addq $sp, 16, $sp
    ret $31, ($26)
```

Stack Frame for Procedure `who`

- None required
- “Leaf” procedure

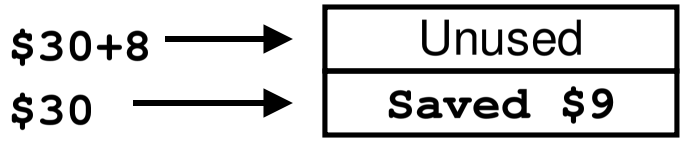
Callee Save Version

Callee Save

- **Must be preserved across procedure calls**
- **If `who` wants to use a callee-save register**
  - Save it on stack upon procedure entry
  - Restore when returning

Stack Frame for Procedure `who`

- 16 bytes total
- To meet size requirements

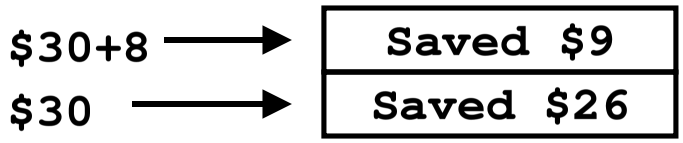



```
who:
    # make frame
    lda $sp, -16($sp)
    # save $9
    stq $9, 0($sp)
    bis $31, 255, $9
    s4addq $9, $9, $0
    # restore $9
    ldq $9, 0($sp)
    # remove frame
    addq $sp, 16, $sp
    ret $31, ($26)
```

Callee Save Version (Cont.)

Stack Frame for Procedure `yoo`

- 16 bytes total



- Must save **$9**
  - Preserve for *its* caller
- Must save **$26**
  - Will get overwritten by **bsr** instruction

```
yoo:
    # make frame
    lda $sp, 16($sp)
    # save $9, $26
    stq $26, 0($sp)
    stq $9, 8($sp)
    bis $31, 17, $9
    bsr $26, who
    addq $9, $0, $0
    # restore $9, $26
    ldq $26, 0($sp)
    ldq $9, 8($sp)
    # remove frame
    addq $sp, 16($sp)
    ret $31, ($26)
```

Actual Form

Procedure `yoo` uses callee save

```
yoo:
    # make frame
    lda $sp, 16($sp)
    # save $9, $26
    stq $26, 0($sp)
    stq $9, 8($sp)
    bis $31, 17, $9
    bsr $26, who
    addq $9, $0, $0
    # restore $9, $26
    ldq $26, 0($sp)
    ldq $9, 8($sp)
    # remove frame
    addq $sp, 16($sp)
    ret $31, ($26)
```

Procedure `who` uses only caller save

- Doesn't need to save anything since makes no calls
- Doesn't need any stack frame

```
who:
    bis $31, 255, $1
    s4addq $1, $1, $0
    ret $31, ($26)
```

Procedure Categories

Leaf procedures that do not use stack

- Do not call other procedures
- Can fit all temporaries in caller-save registers

Leaf procedures that use stack

- Do not call other procedures
- Need stack for temporaries

Non-leaf procedures

- Must use stack
- at the very least, to save the return address `$26`
- Also when need to store array

Recursive Factorial

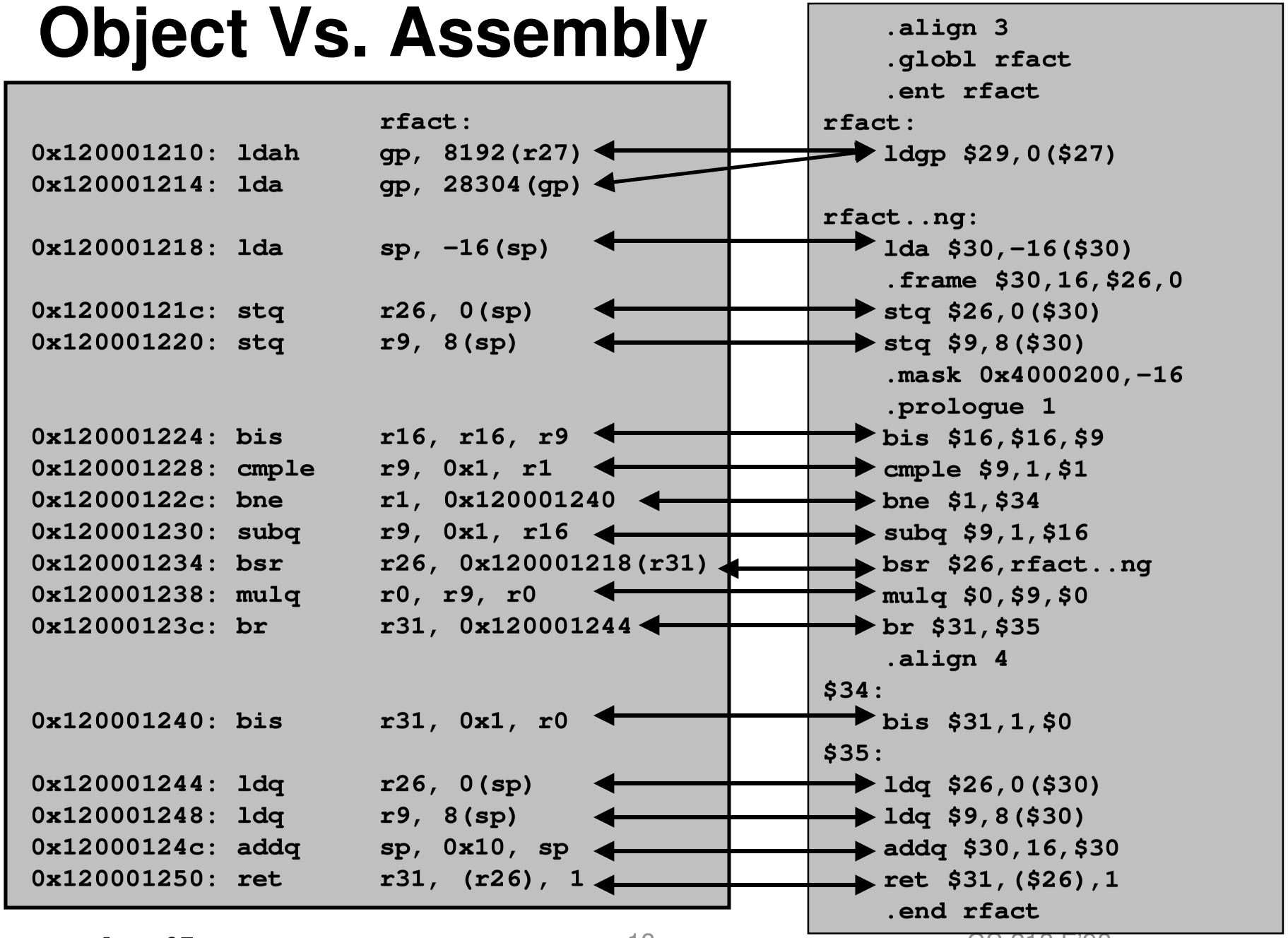
```
long int rfact
   (long int x)
{
   long int rval;
   if (x <= 1)
     return 1;
   rval = rfact(x-1);
   return rval * x;
}
```

Complete Assembly

- **Assembler directives**
  - Lines beginning with "."
  - Not of concern to us
- **Labels**
  - **$XX**
- **Actual instructions**

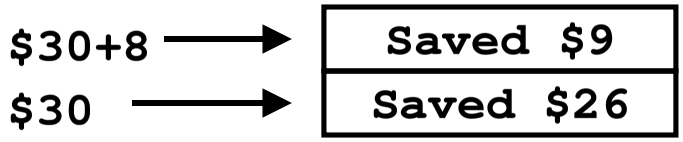
```
      .align 3
      .globl rfact
      .ent rfact
rfact:
      ldgp $29,0($27)
rfact..ng:
      lda $30,-16($30)
      .frame $30,16,$26,0
      stq $26,0($30)
      stq $9,8($30)
      .mask 0x4000200,-16
      .prologue 1
      bis $16,$16,$9
      cmple $9,1,$1
      bne $1,$34
      subq $9,1,$16
      bsr $26,rfact..ng
      mulq $0,$9,$0
      br $31,$35
      .align 4
$34:
      bis $31,1,$0
$35:
      ldq $26,0($30)
      ldq $9,8($30)
      addq $30,16,$30
      ret $31,($26),1
      .end rfact
```

Object Vs. Assembly



Rfact Stack

Stack Frame



`$26`

– Return pointer

`$9`

– Callee save

Global Pointer

- **Set on first entry to `rfact`**
- **Recursive calls don't need to change**
  - Enter at **`rfact..ng`**

Prologue: Set Up Frame

```
rfact:
    ldgp $29,0($27)
rfact..ng:
    lda $30,-16($30)
    stq $26,0($30)
    stq $9,8($30)
```

Epilogue: Undo Frame

```
$35:
    ldq $26,0($30)
    ldq $9,8($30)
    addq $30,16,$30
    ret $31,($26),1
```

Rfact Body

```
  • • •                 # Prologue
  bis $16,$16,$9        #  $9 = x
  cmple $9,1,$1         #  if (x <= 1)
  bne $1,$34            #   then goto R1
  subq $9,1,$16         #  $16 = x-1
  bsr $26,rfact..ng     #  rfact(x-1)
  mulq $0,$9,$0         #  rval * x
  br $31,$35            #  goto Epilogue
$34:                    # R1:
  bis $31,1,$0          #  Return 1
$35:                    # Epilogue:
```

Registers

$0

- **rval** from recursive call
- Return value from this call

$16

- Initially argument **x**
- Must set to **x-1** when making call

$9

- Hold copy of **x**
- To use after recursive call returns

```
long int rfact
  (long int x)
{
  long int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

Tail Recursion

Tail Recursive Procedure

```
long int t_helper
   (long int x, long int val)
{
  if (x <= 1)
    return val;
  return
    t_helper(x-1, val*x);
}
```

General Form

```
t_helper(x, val)
{
    • • •
  return
   t_helper(Xexpr,  Vexpr)
}
```

Top-Level Call

```
long int tfact(long int x)
{
  return t_helper(x, 1);
}
```

Form

- Directly return value returned by recursive call

Consequence

- Can convert into loop

Removing Tail Recursion

Optimized General Form

```
t_helper(x, val)
{
 start:
   • • •
  val = Vexpr;
  x = Xexpr;
  goto start;
}
```

Resulting Code

```
long int t_helper
  (long int x, long int val)
{
 start:
  if (x <= 1)
    return val;
  val = val*x;
  x = x-1;
  goto start;
}
```

Effect of Optimization

- **Turn recursive chain into single procedure**
- **No stack frame needed**
- **Constant space requirement**
  - Vs. linear for recursive version

Generated Code for Tail Recursive Proc.

Optimized Form

```
long int t_helper
   (long int x,
    long int val)
{
 start:
  if (x <= 1)
    return val;
  val = val*x;
  x = x-1;
  goto start;
}
```

Assembly

```
t_helper:
    bis $17,$17,$0    #  $0 = val
$39:                  # start:
    cmple $16,1,$1    #  if x<=1
    bne $1,$37        #  goto done
    mulq $0,$16,$1    #  temp = val*x
    subq $16,1,$16    #  x--
    bis $1,$1,$0      #  val = temp
    br $31,$39        #  goto start
$37:                  # done:
    ret $31,($26),1   #  Return
```

Multi-Way Recursion

```
long int r_prod
   (long int from, long int to)
{
  long int middle;
  long int prodA, prodB;
  if (from >= to)
    return from;
  middle = (from + to) >> 1;
  prodA = r_prod(from, middle);
  prodB = r_prod(middle+1, to);
  return prodA * prodB;
}
```

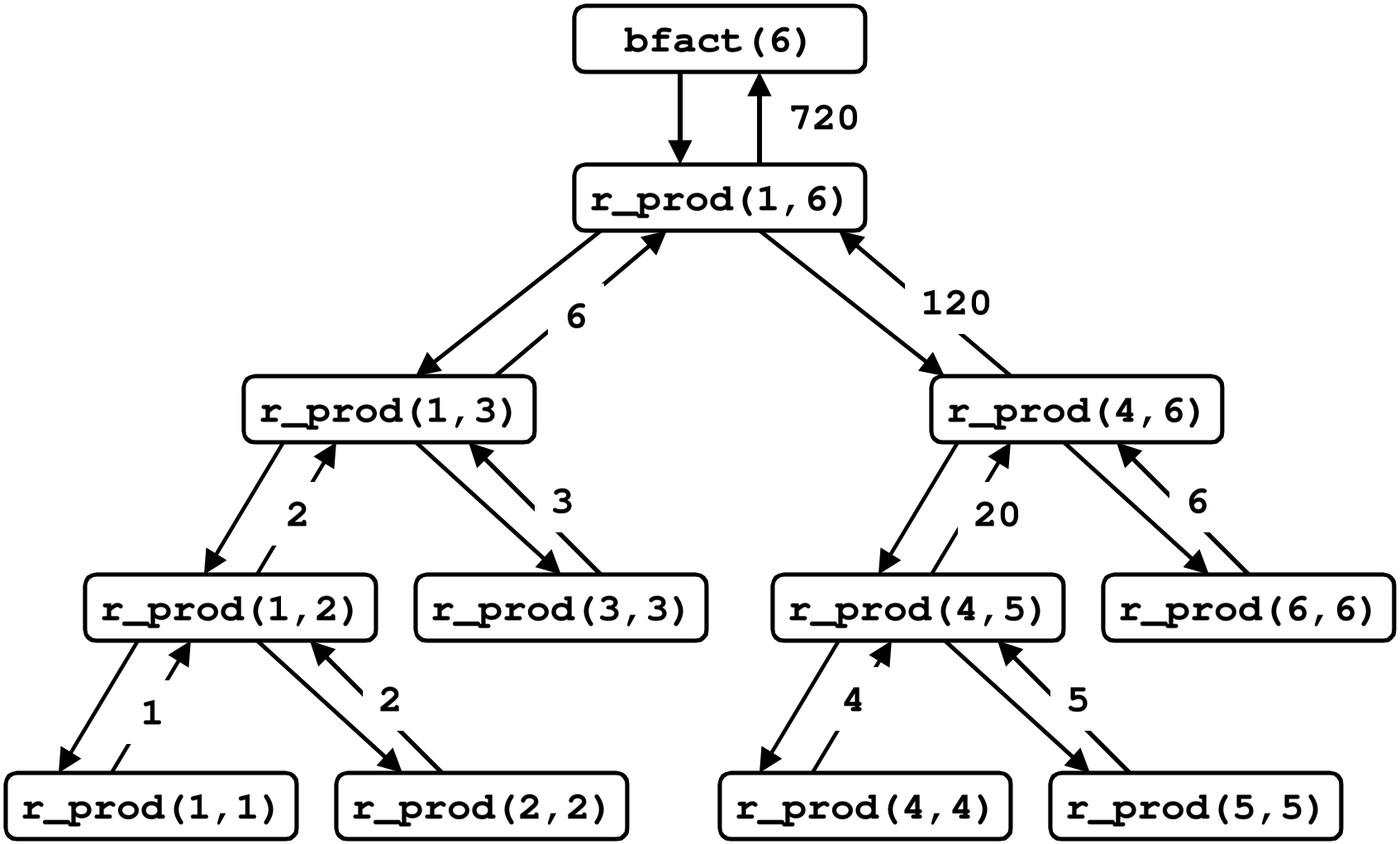
Top-Level Call

```
long int bfact
   (long int x)
{
  return r_prod(1,x);
}
```

- **Compute product** $x * (x+1) * \ldots * (y-1) * y$
- **Split into two ranges:**
  - **Left:** $x * (x+1) * \ldots * (m-1) * m$
  - **Right:** $(m+1) * \ldots * (y-1) * y$

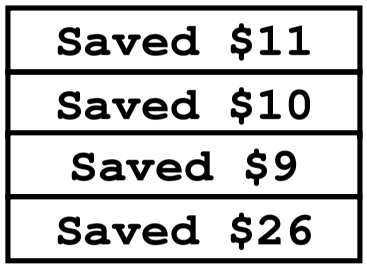
  $m = \lfloor (x+y)/2 \rfloor$
- **No real advantage algorithmically**

Binary Splitting Example



Multi-Way Recursive Code

Stack Frame



$26
- Return pointer

$9
- Callee save
- Used to hold **middle**

$10
- Callee save
- Used to hold **prodA**

$11
- Callee save
- Used to hold **to**

```
r_prod..ng:
  • • •                 # Prologue
  bis $16,$16,$0        # $0 = from
  bis $17,$17,$11       # $11 = to
  # if !(from<to) goto Epilogue
  cmplt $0,$11,$1
  beq $1,$47
  addq $0,$11,$9
  sra $9,1,$9           # $9 = middle
  bis $9,$9,$17
  # r_prod(from, middle)
  bsr $26,r_prod..ng
  bis $0,$0,$10         # prodA
  addq $9,1,$16         # middle+1
  bis $11,$11,$17
  # r_prod(middle+1, to)
  bsr $26,r_prod..ng
  mulq $10,$0,$0        # prodA*prodB
$47:  • • •             # Epilogue
```

Pointer Code

Recursive Procedure

```
void s_helper
  (long int x,
   long int *accum)
{
  if (x <= 1)
    return;
  else {
    *accum *= x;
    s_helper
        (x-1, accum);
  }
}
```

Top-Level Call

```
long int sfact
  (long int x)
{
  long int val = 1;
  s_helper(x, &val);
  return val;
}
```

- **Pass pointer to update location**
- **Uses tail recursion**
  - But GCC only partially optimizes it

Creating Pointer

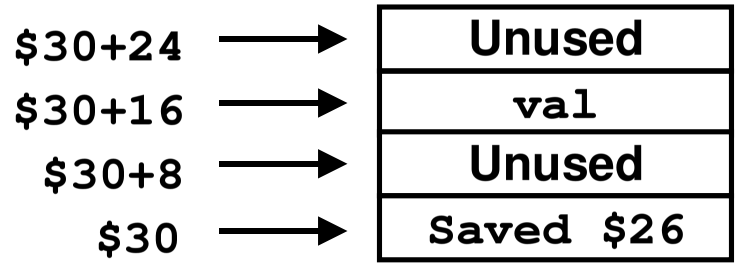
```
long int sfact
   (long int x)
{
   long int val = 1;
   s_helper(x, &val);
   return val;
}
```

Body for `sfact`

```
   • • •                # Prologue
   bis $31,1,$1
   stq $1,16($30)       # val = 1
   addq $30,16,$17      # $17 = &val
   bsr $26,s_helper..ng
   ldq $0,16($30)       # return val
   • • •                # Epilogue
```

Using Stack for Local Variable

- **Variable `val` must be stored on stack**
  - Need to create pointer to it
- **Compute pointer as `$sp+16`**
- **Store in `$17` to pass as second argument**



Using Pointer

```
void s_helper
   (long int x,
    long int *accum)
{
   • • •
   *accum *= x;
   • • •
}
```

```
   • • •
   ldq $1,0($17)    # $1 = *accum
   mulq $16,$1,$1   # *accum * x
   stq $1,0($17)    # Update *accum
   • • •
```

- Register `$16` holds `x`
- Register `$17` holds `accum`

Mutual Recursion

Top-Level Call

```
long int lrfact
   (long int x)
{
  long int left = 1;
  return
   left_prod(&left, &x);
}
```

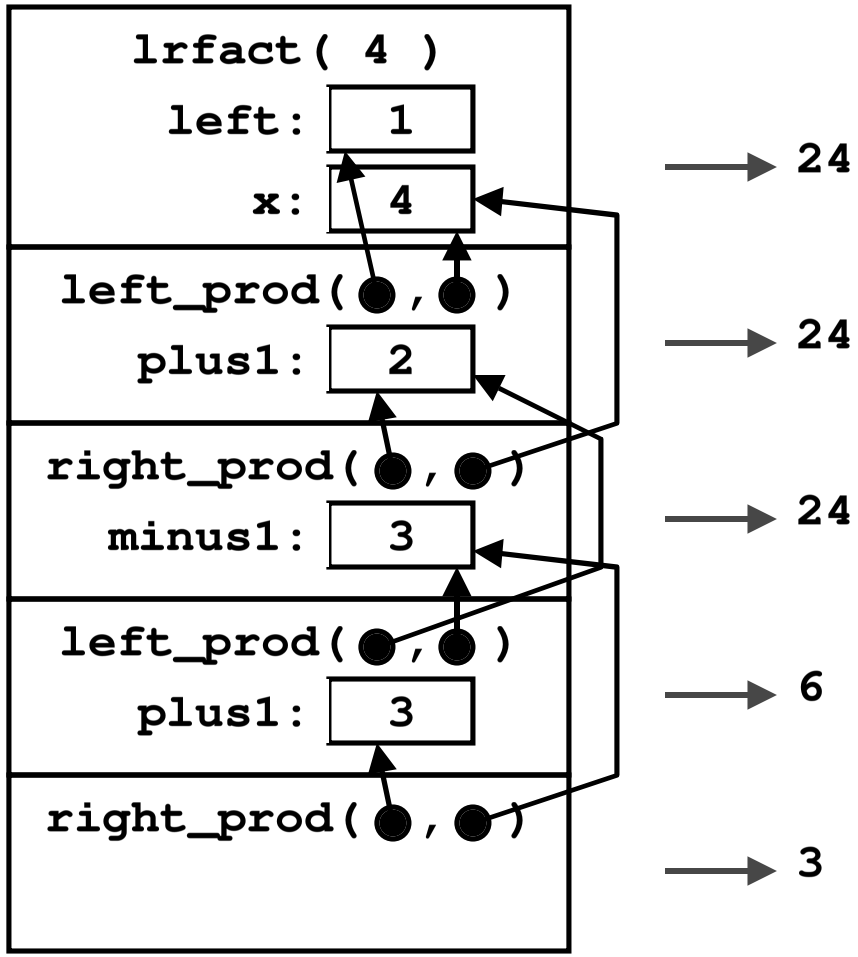
```
long int left_prod
(long int *leftp, long int *rightp)
{
  long int left = *leftp;
  if (left >= *rightp)
    return left;
  else {
    long int plus1 = left+1;
    return left *
      right_prod(&plus1, rightp);
  }
}
```

```
long int right_prod
(long int *leftp, long int *rightp)
{
  long int right = *rightp;
  if (*leftp == right)
    return right;
  else {
    long int minus1 = right-1;
    return right *
      left_prod(leftp, &minus1);
  }
}
```

Mutually Recursive Execution Example

Calling

- **Recursive routines pass two arguments**
  - Pointer to own local variable
  - Pointer to caller's local variable

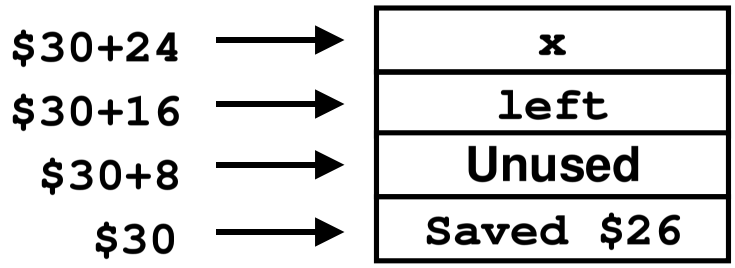


Implementation of `lrfact`

Call to Recursive Routine

```
long int left = 1;
return left_prod(&left, &x);
```

Stack Frame



Code for Call

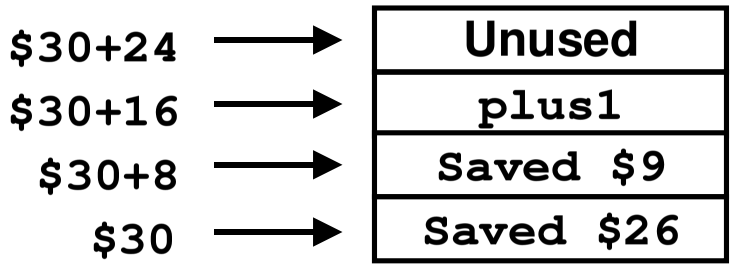
```
stq $16,24($30)   # x on stack
bis $31,1,$1
stq $1,16($30)    # left on stack
addq $30,16,$16   # &left
addq $30,24,$17   # &x
bsr $26,left_prod..ng
```

Implementation of `left_prod`

Call to Recursive Routine

```
long int plus1 = left+1;
return left * right_prod(&plus1, rightp);
```

Stack Frame



Code for Call

```
ldq $9,0($16)    # left = *leftp
 • • •
addq $9,1,$1     # left+1
stq $1,16($30)   # plus1 on stack
addq $30,16,$16  # &plus1
 # rightp still in $17
jsr $26,right_prod
```

Main Ideas

Can Sometimes Avoid Allocating Stack Frame

- Leaf procedures
- Tail recursion converted into loop

Stack Provides Storage for Procedure Instantiation

- Save state
- Local variables
- Any variable for which must create pointer

Assembly Code Must Manage Stack

- Allocate / deallocate by decrementing / incrementing stack pointer
- Saving / restoring register state

Stack Adequate for All Forms of Recursion

- Multi-way
- Mutual