

# Imperative Programming

## The Case of FORTRAN

ICOM 4036

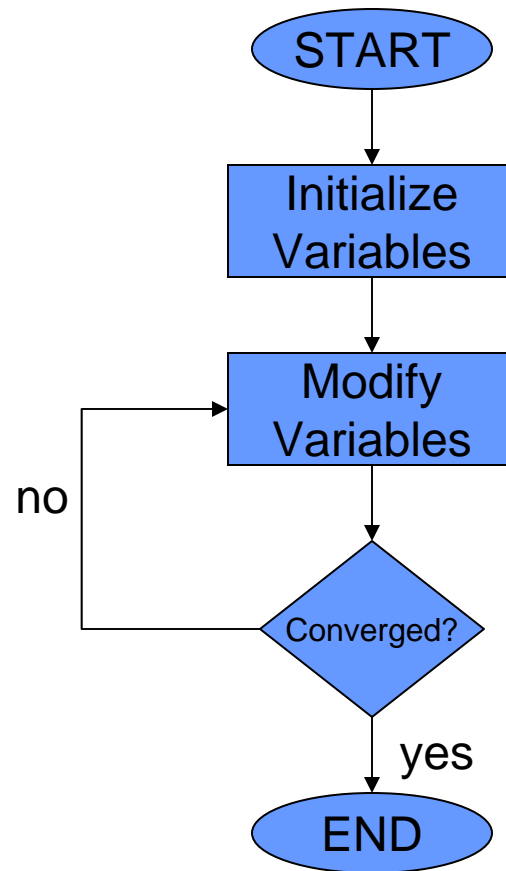
Lecture 5

# The Imperative Paradigm

- Computer Model consists of bunch of variables
- A program is a sequence of state modifications or assignment statements that converge to an answer
- PL provides multiple tools for structuring and organizing these steps
  - E.g. Loops, procedures

This is what you have been doing since INGE 3016!

# A Generic Imperative Program



# Imperative Fibonacci Numbers (C)

```
int fibonacci(int f0, int f1, int n) {  
    // Returns the nth element of the Fibonacci sequence  
    int fn = f0;  
    for (int i=0; i<n; i++) {  
        fn = f0 + f1;  
        f0 = f1;  
        f1 = fn;  
    }  
    return fn;  
}
```

## Examples of (Important) Imperative Languages

- FORTRAN (J. Backus IBM late 50's)
- Pascal (N. Wirth 70's)
- C (Kernigham & Ritchie AT&T late 70's)
- C++ (Stroustrup AT&T 80's)
- Java (Sun Microsystems late 90's)
- C# (Microsoft 00's)

# FORTRAN Highlights

- For High Level Programming Language ever implemented
- First compiler developed by IBM for the IBM 704 computer
- Project Leader: John Backus
- Technology-driven design
  - Batch processing, punched cards, small memory, simple I/O, GUI's not invented yet

# Some Online References

- Professional Programmer's Guide to FORTRAN
- Getting Started with G77

Links available on course web site

# Structure of a FORTRAN program

```
PROGRAM <name>
    <program_body>
END

SUBROUTINE <name> (args)
    <subroutine_body>
END

FUNCTION <name> (args)
    <function_body>
END
...
```



# Lexical/Syntactic Structure

- One statement per line
- First 6 columns reserved
- Identifiers no longer than 6 symbols
- Flow control uses numeric labels
- Unstructured programs possible

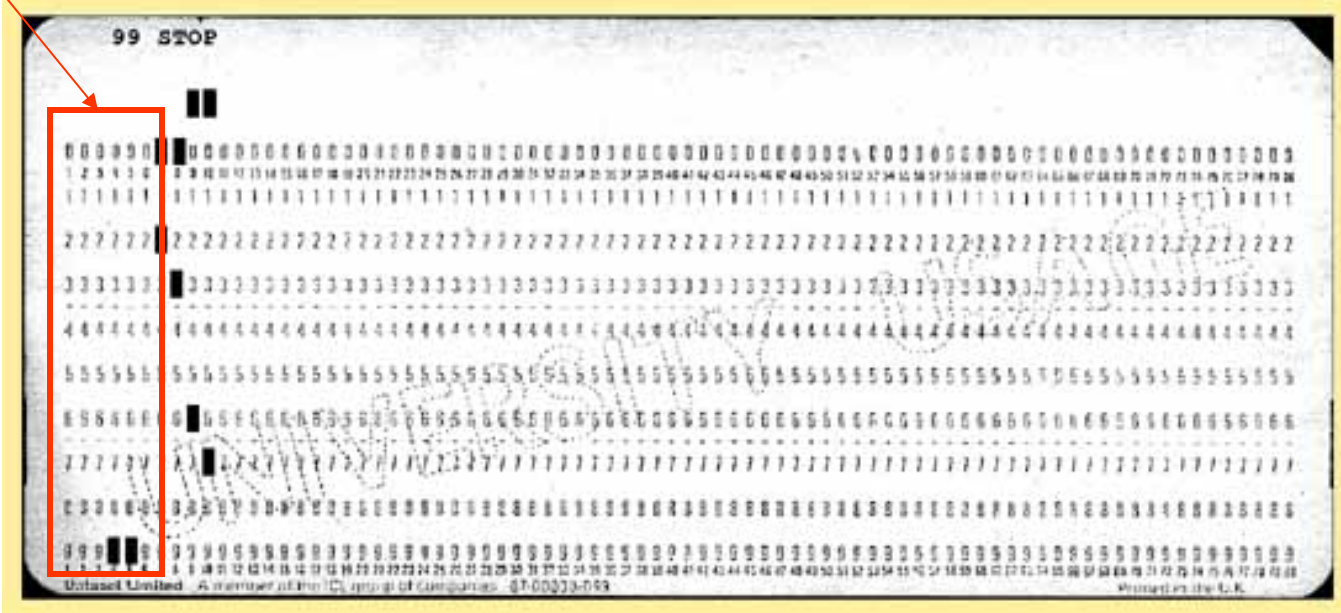
# Hello World in Fortran

```
PROGRAM TINY
  WRITE(UNIT=*, FMT=*) 'Hello, world'
END
```

One Statement Per line

First 6 columns Reserved

Designed with the Punched Card in Mind



# FORTRAN By Example 2

```
PROGRAM LOAN
  WRITE(UNIT=*, FMT=*) 'Enter amount, % rate, years'
  READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
  RATE = PCRATE / 100.0
  REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
  WRITE(UNIT=*, FMT=*) 'Annual repayments are ', REPAY
END
```

Implicitly Defined Variables  
Type determined by initial letter  
I-M ~ INTEGER  
A-H, O-Z FLOAT

# FORTRAN By Example 2

```
PROGRAM LOAN
  WRITE(UNIT=*, FMT=*) 'Enter amount, % rate, years'
  READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
  RATE = PCRATE / 100.0
  REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
  WRITE(UNIT=*, FMT=*) 'Annual repayments are ', REPAY
END
```

FORTRAN's Version  
of  
Standard Output Device

# FORTRAN By Example 2

```
PROGRAM LOAN
  WRITE(UNIT=*, FMT=*) 'Enter amount, % rate, years'
  READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
  RATE = PCRATE / 100.0
  REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
  WRITE(UNIT=*, FMT=*) 'Annual repayments are ', REPAY
END
```

FORTRAN's Version  
of  
Default Format

# FORTRAN By Example 3

```
PROGRAM REDUCE
WRITE(UNIT=*, FMT=*) 'Enter amount, % rate, years'
READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
RATE = PCRATE / 100.0
REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
WRITE(UNIT=*, FMT=*) 'Annual repayments are ', REPAY
WRITE(UNIT=*, FMT=*) 'End of Year Balance'
DO 15, IYEAR = 1, NYEARS, 1
    AMOUNT = AMOUNT + (AMOUNT * RATE) - REPAY
    WRITE(UNIT=*, FMT=*) IYEAR, AMOUNT
15 CONTINUE
END
```

A loop consists of two  
separate statements  
-> Easy to construct  
unstructured programs

# FORTRAN Do Loops

```
PROGRAM REDUCE
WRITE(UNIT=*, FMT=*) 'Enter amount, % rate, years'
READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
RATE = PCRATE / 100.0
REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
WRITE(UNIT=*, FMT=*) 'Annual repayments are ', REPAY
WRITE(UNIT=*, FMT=*) 'End of Year Balance'
DO 15, IYEAR = 1, NYEARS, 1
    AMOUNT = AMOUNT + (AMOUNT * RATE) - REPAY
    WRITE(UNIT=*, FMT=*) IYEAR, AMOUNT
15 CONTINUE
END
```

```
Enter amount, % rate, years
2000, 9.5, 5
Annual repayments are 520.8728
End of Year Balance
  1 1669.127
  2 1306.822
  3  910.0968
  4  475.6832
  5  2.9800416E-04
```

A loop consists of two separate statements

-> Easy to construct unstructured programs

# FORTRAN Do Loops

```
PROGRAM REDUCE
WRITE(UNIT=*, FMT=*) 'Enter amount, % rate, years'
READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
RATE = PCRATE / 100.0
REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
WRITE(UNIT=*, FMT=*) 'Annual repayments are ', REPAY
WRITE(UNIT=*, FMT=*) 'End of Year Balance'
DO 15, IYEAR = 1, NYEARS, 1
    AMOUNT = AMOUNT + (AMOUNT * RATE) - REPAY
    WRITE(UNIT=*, FMT=*) IYEAR, AMOUNT
15 CONTINUE
END
```

```
Enter amount, % rate, years
2000, 9.5, 5
Annual repayments are 520.8728
End of Year Balance
  1 1669.127
  2 1306.822
  3  910.0968
  4  475.6832
  5  2.9800416E-04
```

- optional increment (can be negative)
- final value of index variable
- index variable and initial value
- end label



# FORTRAN Functions

```
PROGRAM TRIANG
  WRITE(UNIT=*,FMT=*)'Enter lengths of three sides:'
  READ(UNIT=*,FMT=*) SIDEA, SIDEB, SIDEC
  WRITE(UNIT=*,FMT=*)'Area is ', AREA3(SIDEA,SIDEB,SIDEC)
END

FUNCTION AREA3(A, B, C)
* Computes the area of a triangle from lengths of sides
  S = (A + B + C)/2.0
  AREA3 = SQRT(S * (S-A) * (S-B) * (S-C))
END
```

- No recursion
- Parameters passed by reference only
- Arrays allowed as parameters
- No nested procedure definitions – Only two scopes
- Procedural arguments allowed
- No procedural return values

Think: why do you think FORTRAN designers made each of these choices?

# FORTRAN IF-THEN-ELSE

```
REAL FUNCTION AREA3(A, B, C)
*      Computes the area of a triangle from lengths of its sides.
*      If arguments are invalid issues error message and returns
*      zero.
REAL A, B, C
S = (A + B + C)/2.0
FACTOR = S * (S-A) * (S-B) * (S-C)
IF(FACTOR .LE. 0.0) THEN
    STOP 'Impossible triangle'
ELSE
    AREA3 = SQRT(FACTOR)
END IF
END
```

NO RECURSION ALLOWED IN FORTRAN77 !!!

# FORTRAN ARRAYS

```
SUBROUTINE MEANSD(X, NPTS, AVG, SD)
  INTEGER NPTS
  REAL X(NPTS), AVG, SD
  SUM = 0.0
  SUMSQ = 0.0
  DO 15, I = 1, NPTS
    SUM = SUM + X(I)
    SUMSQ = SUMSQ + X(I)**2
15  CONTINUE
  AVG = SUM / NPTS
  SD = SQRT(SUMSQ - NPTS * AVG**2) / (NPTS-1)
END
```

Subroutines are analogous  
to void functions in C

Parameters are passed by reference

```
subroutine checksum(buffer,length,sum32)
```

```
C Calculate a 32-bit 1's complement checksum of the input buffer, adding  
C it to the value of sum32. This algorithm assumes that the buffer  
C length is a multiple of 4 bytes.
```

```
C a double precision value (which has at least 48 bits of precision)  
C is used to accumulate the checksum because standard Fortran does not  
C support an unsigned integer datatype.
```

```
C buffer - integer buffer to be summed  
C length - number of bytes in the buffer (must be multiple of 4)  
C sum32 - double precision checksum value (The calculated checksum  
C is added to the input value of sum32 to produce the  
C output value of sum32)
```

```
integer buffer(*),length,i,hibits  
double precision sum32,word32  
parameter (word32=4.294967296D+09)  
C (word32 is equal to 2**32)
```

```
C LENGTH must be less than 2**15, otherwise precision may be lost  
C in the sum  
if (length .gt. 32768)then  
print *, 'Error: size of block to sum is too large'  
return  
end if
```

```
do i=1,length/4  
if (buffer(i) .ge. 0)then  
sum32=sum32+buffer(i)  
else  
C sign bit is set, so add the equivalent unsigned value  
sum32=sum32+(word32+buffer(i))  
end if  
end do
```

```
C fold any overflow bits beyond 32 back into the word  
10 hibits=sum32/word32  
if (hibits .gt. 0)then  
sum32=sum32-(hibits*word32)+hibits  
go to 10  
end if  
end
```

# WhiteBoard Exercises

- Computing machine precision
- Computing the integral of a function
- Solving a linear system of equations

FORTRAN Heavily used in scientific computing applications

# Implementing Procedures

- Why procedures?
  - Abstraction
  - Modularity
  - Code re-use
- Initial Goal
  - Write segments of assembly code that can be re-used, or “called” from different points in the main program.
  - KISS: Keep It Simple Stupid:
    - no parameters, no recursion, no locals, no return values

# Procedure Linkage

## Approach I

- Problem
  - procedure must determine where to return after servicing the call
- Solution: Architecture Support
  - Add a jump instruction that saves the return address in some place known to callee
    - MIPS: `jal` instruction saves return address in register `$ra`
  - Add an instruction that can jump to return address
    - MIPS: `jr` instruction jumps to the address contained in its argument register

# Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
    a = 12;
    b = 5;
    res = 0;
    div();
    printf("Res
}
void div(void
    while (a >=
        a = a - b;
        res ++;
    }
}
```

```
.data
x:      .word 0
y:      .word 0
res:    .word 0
pf1:    .ascii "Result = "
pf2:    .ascii "Remainder = "
        .globl main
        .text
main:
        # int main() {
        # assumes registers sx unused
        la      $s0, x
        li      $s1, 12
        sw      $s1, 0($s0)
        la      $s0, y
        li      $s2, 5
        sw      $s2, 0($s0)
        la      $s0, res
        li      $s3, 0
        # res = 0;
        sw      $s3, 0($s0)
        jal     d
        # div();
        lw      $s3, 0($s0)
        la      $a0, pf1
        li      $v0, 4
        # printf("Result = %d \n");
        # //system call to print_str
        syscall
        move    $a0, $s3
        li      $v0, 1
        # //system call to print_int
        syscall
        la      $a0, pf2
        li      $v0, 4
        # printf("Remainder = %d \n");
        # //system call to print_str
        syscall
        move    $a0, $s1
        li      $v0, 1
        # //system call to print_int
        syscall
        jr      $ra
        # return // TO Operating System
```

**Function  
Call**

C++

MIPS

Assembly Language



# Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
    a = 12;
    b = 5;
    res = 0;
    div();
    printf("Res = %d\n", res);
}

void div(void) {
    while (a >= b) {
        a = a - b;
        res ++;
    }
}
```

```
# div function
# PROBLEM: Must save args and registers before using them
d:
    # void d(void) {
    # // Allocate registers for globals
    # // x in $s1
    la    $s0, x
    lw    $s1, 0($s0)
    la    $s0, y
    lw    $s2, 0($s0)
    la    $s0, res
    lw    $s3, 0($s0)
    # // res in $s3
while:   bgt    $s2, $s1, ewhile
        sub    $s1, $s1, $s2
        addi   $s3, $s3, 1
        j     while
    # while (x <= y) {
    #     x = x - y
    #     res ++
    # }
    # // Update variables in memory
ewhile:  la    $s0, x
        sw    $s1, 0($s0)
        la    $s0, y
        sw    $s2, 0($s0)
        la    $s0, res
        sw    $s3, 0($s0)
    # return;
enddiv:  jr    $ra
    # }
```

C++

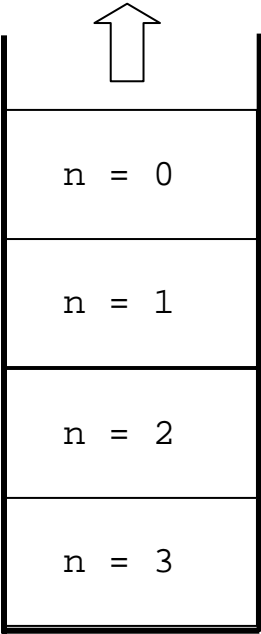
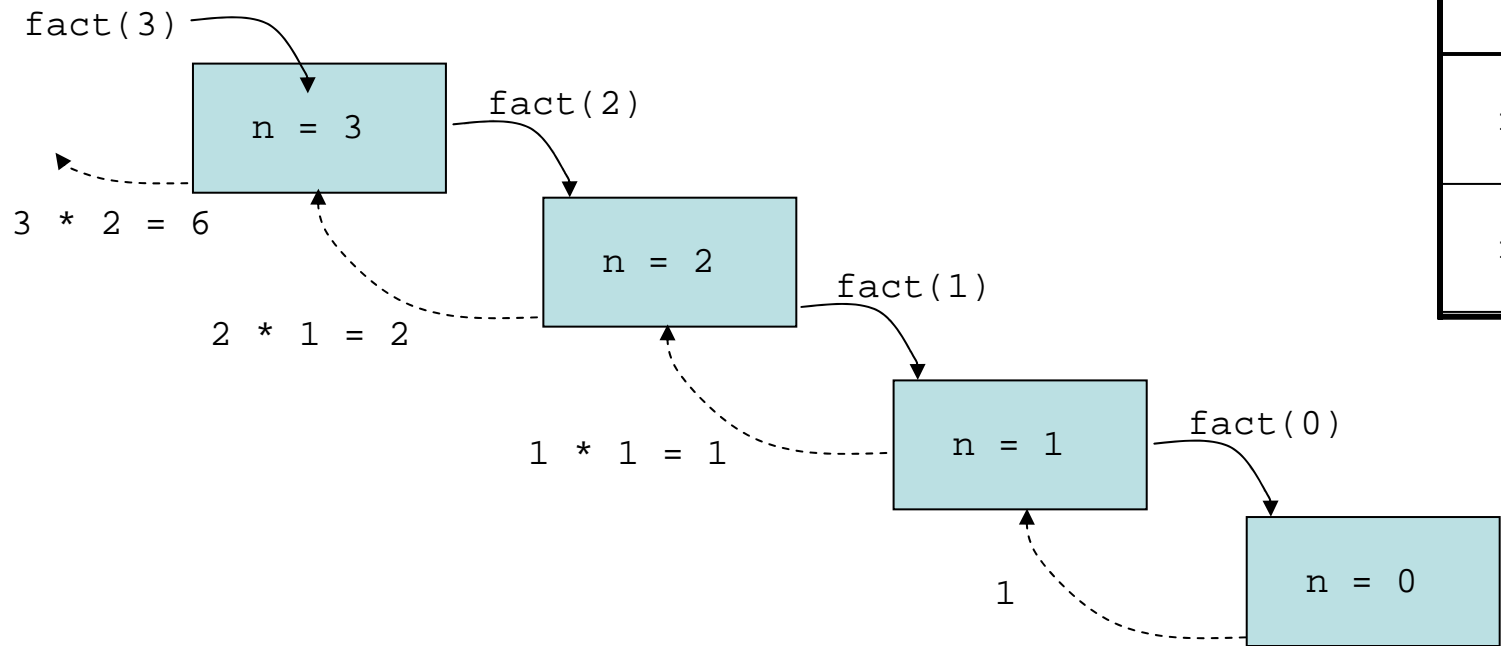
MIPS  
Assembly Language

# Pending Problems With Linkage Approach I

- Registers shared by all procedures
  - procedures must save/restore registers (use stack)
- Procedures should be able to call other procedures
  - save multiple return addresses (use stack)
- Lack of parameters forces access to globals
  - pass parameters in registers
- Recursion requires multiple copies of local data
  - store multiple procedure activation records (use stack)
- Need a convention for returning function

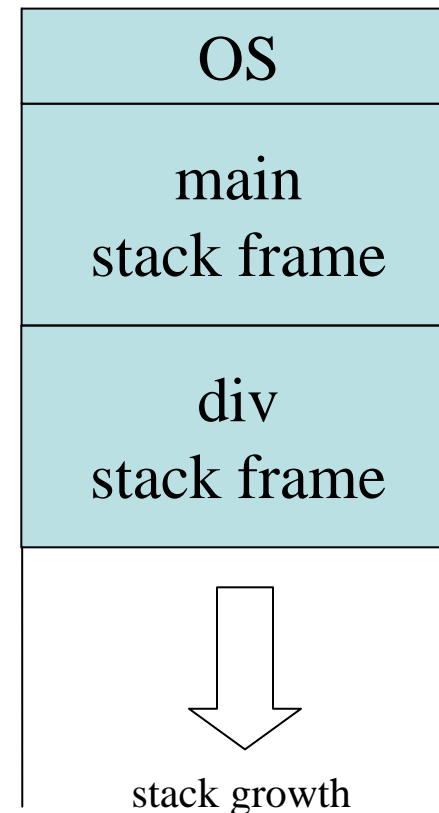
# Recursion Basics

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

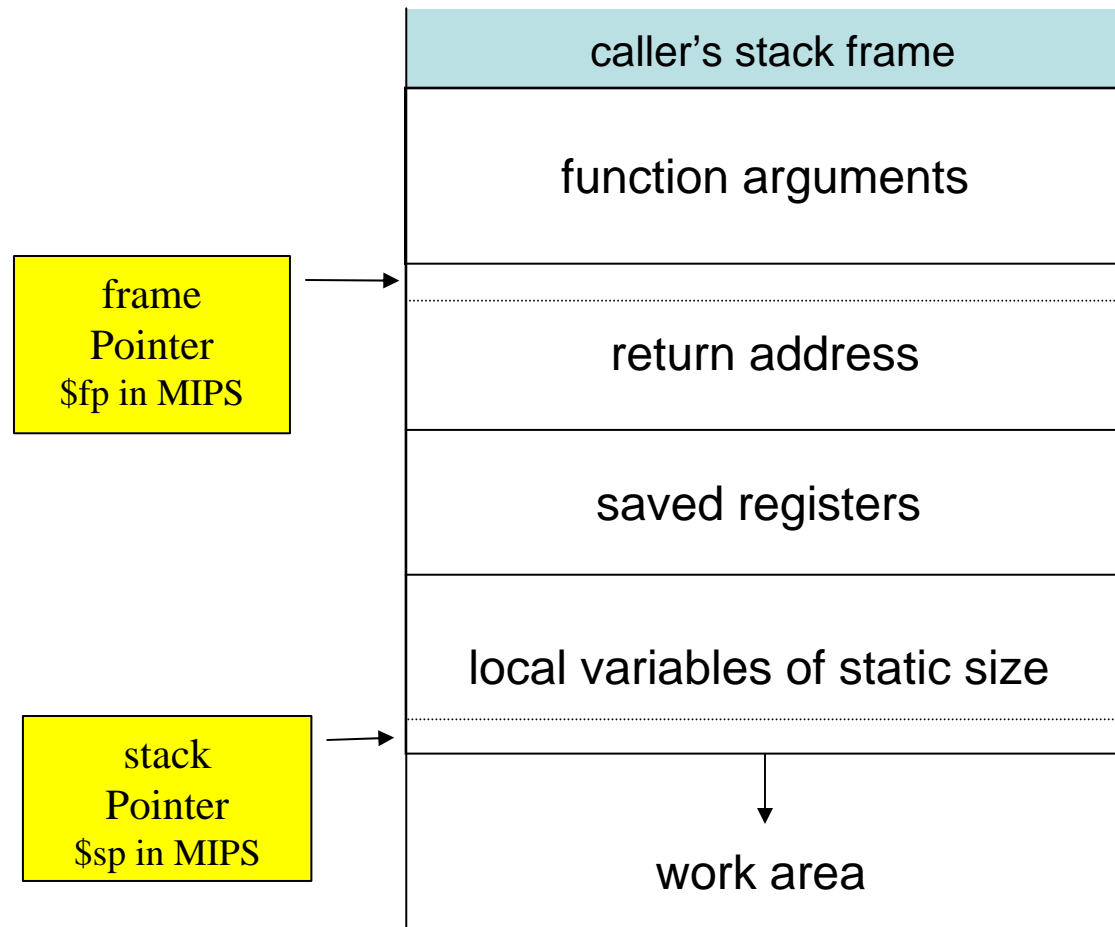


# Solution: Use Stacks of Procedure Frames

- Stack frame contains:
  - Saved arguments
  - Saved registers
  - Return address
  - Local variables



# Anatomy of a Stack Frame



Contract: Every function must leave the stack the way it found it

# Example: Function Linkage using Stack Frames

```
int x = 0;
int y = 0;
int res = 0;
main () {
    x = 12;
    y = 5;
    res = div(x,y);
    printf("Res = %d",res);
}
int div(int a,int b) {
    int res = 0;
    if (a >= b) {
        res = div(a-b,b) + 1;
    }
    else {
        res = 0;
    }
    return res;
}
```

- Add return values
- Add parameters
- Add recursion
- Add local variables

# Example: Function Linkage using Stack Frames

```
div:      sub      $sp, $sp, 28      # Alloc space for 28 byte stack frame
          sw       $a0, 24($sp)     # Save argument registers
          sw       $a1, 20($sp)     # a in $a0
          sw       $ra, 16($sp)     # Save other registers as needed
          sw       $s1, 12($sp)     # Save callee saved registers ($sx)
          sw       $s2, 8($sp)
          sw       $s3, 4($sp)     # No need to save $s4, since not used
          li       $s3, 0
          sw       $s3, 0($sp)     # int res = 0;
          # Allocate registers for locals
          lw       $s1, 24($sp)     # a in $s1
          lw       $s2, 20($sp)     # b in $s2
          lw       $s3, 0($sp)     # res in $s3

if:       bgt     $s2, $s1, else     # if (a >= b) {
          sub     $a0, $s1, $s2     #
          move    $a1, $s2
          jal     div
          addi   $s3, $v0, 1        # res = div(a-b, b) + 1;
          j      endif
else:     li      $s3, 0            # else { res = 0; }
endif:

          sw     $s1, 32($sp)       # deallocate a from $s1
          sw     $s2, 28($sp)       # deallocate b from $s2
          sw     $s3, 0($sp)        # deallocate res from $s3
          move   $v0, $s3           # return res

          lw     $a0, 24($sp)       # Restore saved registers
          lw     $a1, 20($sp)       # a in $a0
          lw     $ra, 16($sp)       # Save other registers as needed
          lw     $s1, 12($sp)       # Save callee saved registers ($sx)
          lw     $s2, 8($sp)
          lw     $s3, 4($sp)       # No need to save $s4, since not used
          addu   $sp, $sp, 28       # pop stack frame
enddiv:   jr      $ra              # return;
#
```

# MIPS: Procedure Linkage Summary

- First 4 arguments passed in \$a0-\$a3
- Other arguments passed on the stack
- Return address passed in \$ra
- Return value(s) returned in \$v0-\$v1
- Sx registers saved by callee
- Tx registers saved by caller