Imperative Programming
The Case of FORTRAN

ICOM 4036

Lecture 5

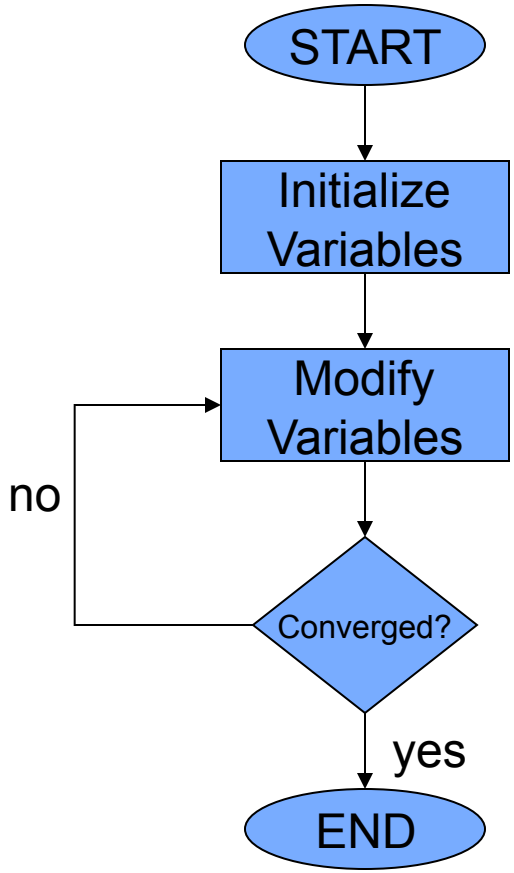
READINGS: PLP Chapters 6 and 8

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

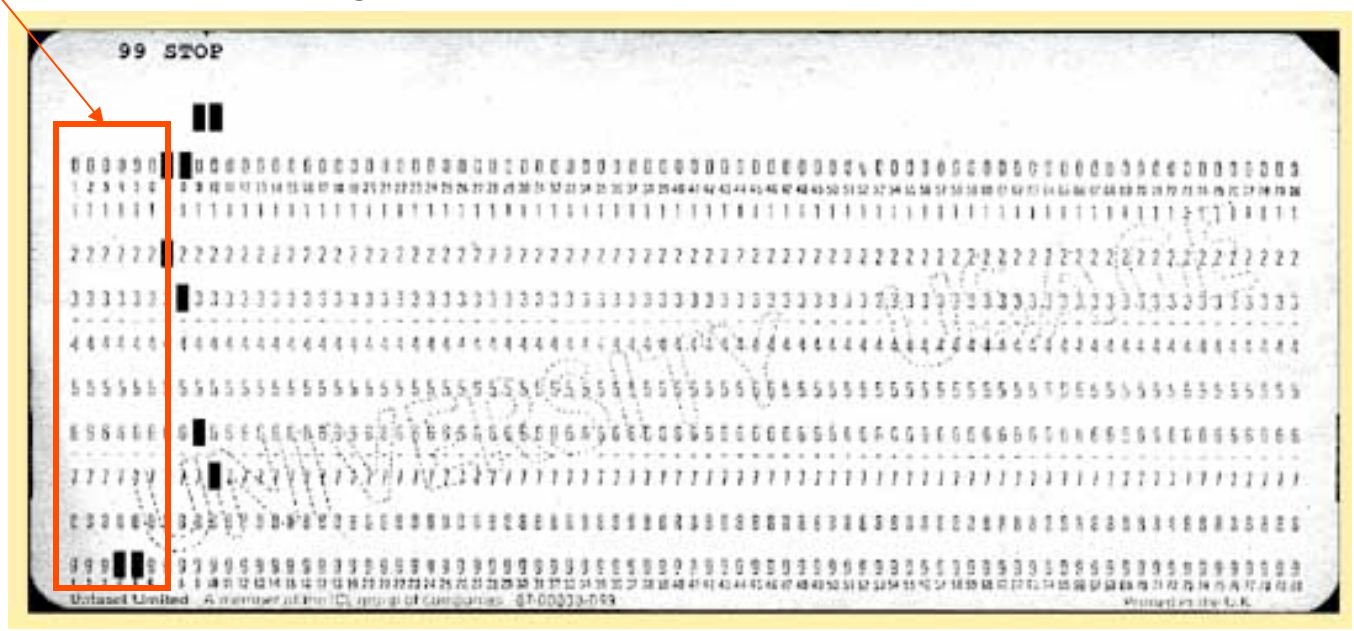
Hello World in Fortran

```
      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)/(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
```

Appendix B
From
Original
Fortan I
Manual
(IBM)

APPENDIX B. TABLE OF FORTRAN STATEMENTS

STATEMENT	NORMAL SEQUENCING
a = b	Next executable statement
GO TO n	Statement n
GO TO n, (n_1,n_2,. . .,n_m)	Statement last assigned
ASSIGN i TO n	Next executable statement
GO TO (n_1,n_2,. . .,n_m), i	Statement n_i
IF (a) n_1,n_2,n_3	Statement n_1,n_2,n_3 as a less than, =, or greater than 0
SENSE LIGHT i	Next executable statement
IF (SENSE LIGHT i) n_1,n_2	Statement n_1,n_2 as Sense Light i ON or OFF
IF (SENSE SWITCH i) n_1,n_2	" " " as Sense Switch i DOWN or UP
IF ACCUMULATOR OVERFLOW n_1,n_2	" " " as Accumulator Overflow trigger ON or OFF
IF QUOTIENT OVERFLOW n_1,n_2	" " " as MQ Overflow trigger ON or OFF
IF DIVIDE CHECK n_1,n_2	" " " as Divide Check trigger ON or OFF
PAUSE or PAUSE n	Next executable statement
STOP or STOP n	Terminates program
DO n i = m_1,m_2 or DO n i = m_1,m_2,m_3	Next executable statement
CONTINUE	" " "
FORMAT (Specification)	Not executed
READ n, List	Next executable statement
READ INPUT TAPE i, n, List	" " "
PUNCH n, List	" " "
PRINT n, List	" " "
WRITE OUTPUT TAPE i, n, List	" " "
READ TAPE i, List	" " "
READ DRUM i, j, List	" " "
WRITE TAPE i, List	" " "
WRITE DRUM i, j, List	" " "
END FILE i	" " "
REWIND i	" " "
BACKSPACE i	" " "
DIMENSION v, v, v, ...	Not executed
EQUIVALENCE (a,b,c,. .), (d,e,f,. .), . . .	" "
FREQUENCY n(i,j,. .), m(k,l,. .), . . .	" "

WhiteBoard Exercises

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

FORTRAN Heavily used in scientific computing applications

Chapter 6:: Control Flow

Programming Language Pragmatics

Michael L. Scott

Control Flow

- Basic paradigms for control flow:
  - Sequencing (e.g. Begin … End)
  - Selection
  - Iteration
  - Subroutines, recursion (and related control abstractions, e.g. iterators)
  - Nondeterminacy
  - Concurrency

Expression Evaluation

- Infix, prefix operators
- Precedence, associativity (see Figure 6.1)
  - C has 15 levels - too many to remember
  - Pascal has 3 levels - too few for good semantics
  - Fortran has 8
  - Ada has 6
    - Ada puts *and & or* at same level
  - **Lesson**: when unsure, use parentheses!

Expression Evaluation

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if . . . then . . . else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Figure 6.1: **Operator precedence levels in Fortran, Pascal, C, and Ada.** The operators at the top of the figure group most tightly.

Expression Evaluation

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
  - distinguish between commutativity, and (assumed to be safe)
  - associativity (known to be dangerous)

`(a + b) + c` works if `a~=maxint` and `b~=minint` and `c<0`

`a + (b + c)` does not

  - inviolability of parentheses

Expression Evaluation

- Short-circuiting
  - Consider `(a < b) && (b < c)`:
    - If `a >= b` there is no point evaluating whether `b < c` because `(a < b) && (b < c)` is automatically false
  - Other similar situations

```
if (b != 0 && a/b == c) ...
if (*p && p->foo) ...
if (f || messy()) ...
```

Expression Evaluation

- Variables as values vs. variables as references
  - value-oriented languages
    - C, Pascal, Ada
  - reference-oriented languages
    - most functional languages (Lisp, Scheme, ML)
    - Clu, Smalltalk
  - Algol-68 kinda halfway in-between
  - Java deliberately in-between
    - built-in types are values
    - user-defined types are objects - references

Expression Evaluation

- Expression-oriented vs. statement-oriented languages
  - expression-oriented:
    - functional languages (Lisp, Scheme, ML)
    - Algol-68
  - statement-oriented:
    - most imperative languages
  - C kinda halfway in-between (distinguishes)
    - allows expression to appear instead of statement

Expression Evaluation

- Orthogonality
  - Features that can be used in any combination
    - Meaning is consistent

```
if (if b != 0 then a/b == c else false)
  then ...
if (if f then true else messy()) then ...
```

- Initialization
  - Pascal has no initialization facility (assign)
- Aggregates

Expression Evaluation

- Assignment
  - statement (or expression) executed for its side effect
  - assignment operators (+=, -=, etc)
    - handy
    - avoid redundant work (or need for optimization)
    - perform side effects exactly once
  - C --, ++
    - postfix form

Expression Evaluation

- Side Effects
  - often discussed in the context of functions
  - a side effect is some permanent state change caused by execution of function
    - some noticeable effect of call other than return value
    - in a more general sense, assignment statements provide the ultimate example of side effects
      - they change the value of a variable

Expression Evaluation

- SIDE EFFECTS ARE FUNDAMENTAL TO THE WHOLE VON NEUMANN MODEL OF COMPUTING

- In (pure) functional, logic, and dataflow languages, there are no such changes
  - These languages are called SINGLE-ASSIGNMENT languages

Expression Evaluation

- Several languages outlaw side effects for functions
  - easier to prove things about programs
  - closer to Mathematical intuition
  - easier to optimize
  - (often) easier to understand
- But side effects can be nice
  - consider rand()

Expression Evaluation

- Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
  - It's nice not to specify an order, because it makes it easier to optimize
  - Fortran says it's OK to have side effects
    - they aren't allowed to change other parts of the expression containing the function call
    - Unfortunately, compilers can't check this completely, and most don't at all

Sequencing

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neuman

Selection

- Selection
  - sequential if statements

```
if ... then ... else
if ... then ... elsif ... else
(cond
      (C1) (E1)
      (C2) (E2)
      ...
      (Cn) (En)
      (T)  (Et)
)
```

Selection

- Selection
  - Fortran computed gotos
  - jump code
    - for selection and logically-controlled loops
    - no point in computing a Boolean value into a register, then testing it
    - instead of passing register containing Boolean out of expression as a synthesized attribute, pass inherited attributes INTO expression indicating where to jump to if true, and where to jump to if false

Selection

- Jump is especially useful in the presence of short-circuiting
- **Example** (section 6.4.1 of book):

```
if ((A > B) and (C > D)) or (E <> F) then
   then_clause
  else
   else_clause
```

Selection

- Code generated w/o short-circuiting (Pascal)

```
            r1 := A                  -- load
            r2 := B
            r1 := r1 > r2
            r2 := C
            r3 := D
            r2 := r2 > r3
            r1 := r1 & r2
            r2 := E
            r3 := F
            r2 := r2 $<>$ r3
            r1 := r1 $|$ r2
            if r1 = 0 goto L2
L1:         then_clause    -- label not actually used
            goto L3
L2:         else_clause
L3:
```

Selection

- Code generated w/ short-circuiting (C)

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1
L4:     r1 := E
        r2 := F
        if r1 = r2 goto L2
L1:     then_clause
        goto L3
L2:     else_clause
L3:
```

Iteration

- Enumeration-controlled
  - Pascal or Fortran-style for loops
    - scope of control variable
    - changes to bounds within loop
    - changes to loop variable within loop
    - value after the loop

Iteration

- The `goto` controversy
  - *assertion*: `gotos` are needed almost exclusively to cope with lack of one-and-a-half loops
  - early return from procedure
  - exceptions
  - in many years of programming, I can't remember using one for any other purpose
    - except maybe complicated conditions that can be separated into a single if-then-else because of the need for short-circuiting

Recursion

- Recursion
  - equally powerful to iteration
  - mechanical transformations back and forth
  - often more intuitive (sometimes less)
  - *naïve* implementation less efficient
    - no special syntax required
    - fundamental to functional languages like Scheme

Recursion

- Tail recursion
  - No computation follows recursive call

```
        /* assume a, b > 0 */
int gcd (int a, int b) {
   if (a == b) return a;
   else if (a > b) return gcd (a - b, b);
   else return gcd (a, b - a);
}
```

Chapter 8 :: Subroutines and Control Abstraction

Programming Language Pragmatics

Michael L. Scott

The MIPS Architecture
ISA at a Glance

- Reduced Instruction Set Computer (RISC)
- 32 general purpose 32-bit registers
- Load-store architecture: Operands in registers
- Byte Addressable
- 32-bit address space

The MIPS Architecture

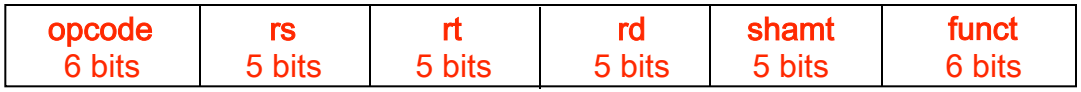
32 Register Set (32-bit registers)

Register #	Reg Name	Function
r0	r0	Zero constant
r4-r7	a0-a3	Function arguments
r1	at	Reserved for Operating Systems
r30	fp	Frame pointer
r28	gp	Global memory pointer
r26-r27	k0-k1	Reserved for OS Kernel
r31	ra	Function return address
r16-r23	s0-s7	Callee saved registers
r29	sp	Stack pointer
r8-r15	t0-t7	Temporary variables
r24-r25	t8-t9	Temporary variables
r2-r3	v0-v1	Function return values

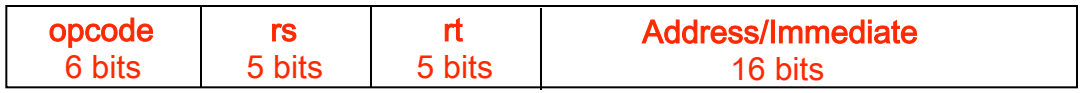
The MIPS Architecture
Main Instruction Formats

Simple and uniform 32-bit 3-operand instruction formats

–R Format: Arithmetic/Logic operations on registers



–I Format: Branches, loads and stores



–J Format: Jump Instruction

MIPS Data Paths

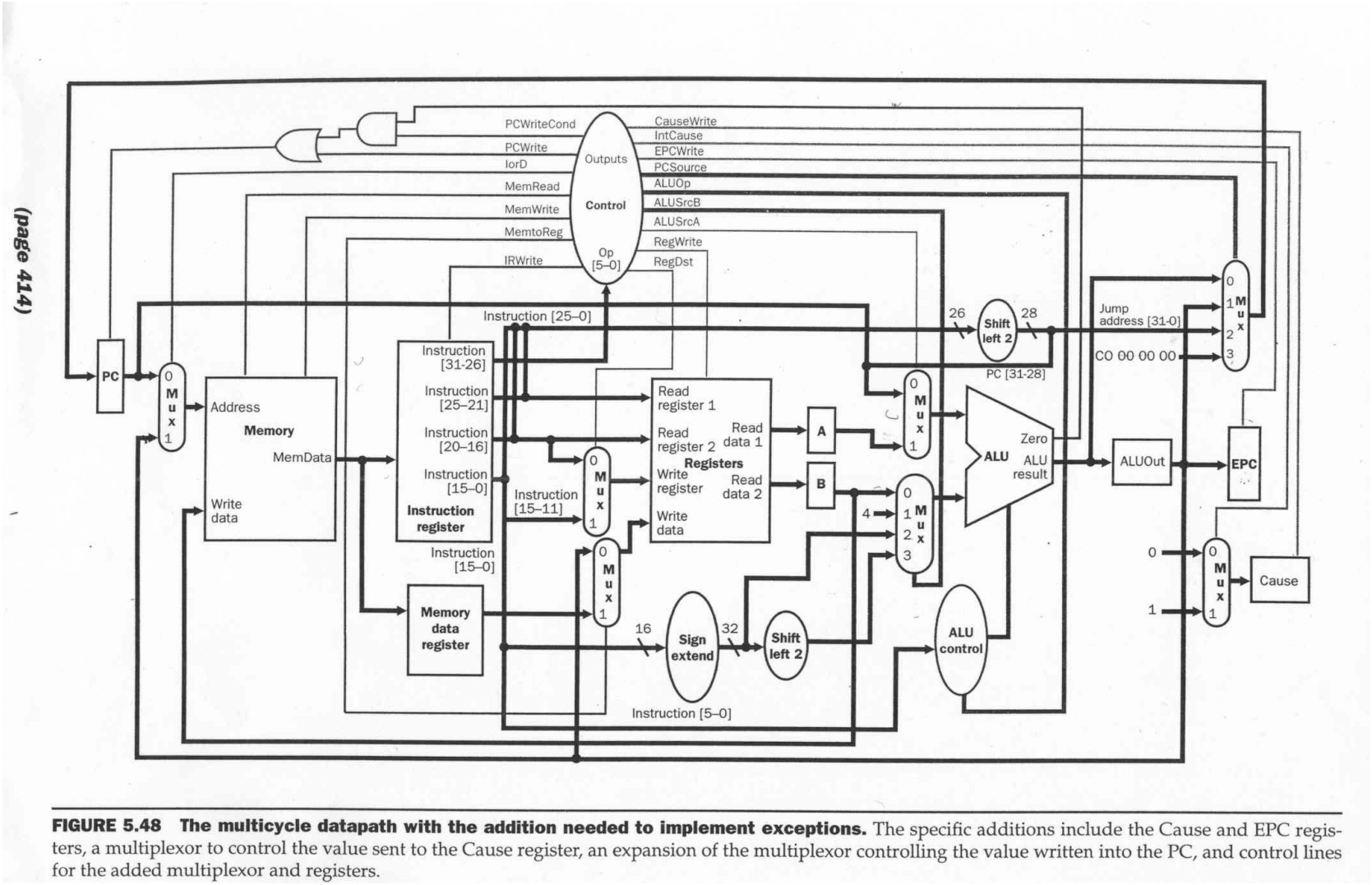
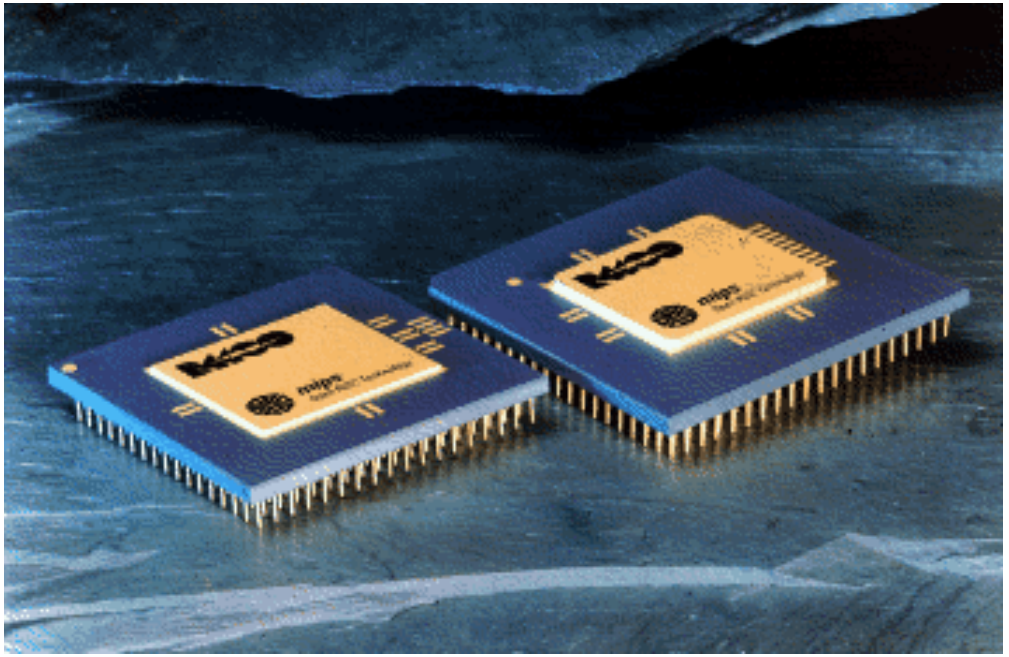


FIGURE 5.48 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers.

Mips Packaging



The MIPS Architecture

Examples of Native Instruction Set

Instruction Group	Instruction	Function
Arithmetic/ Logic	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
	`addi $s1,$s2,K`	`$s1 = $s2 + K`
Load/Store	`lw $s1,K($s2)`	`$s1 = MEM[$s2+K]`
	`sw $s1,K($s2)`	`MEM[$s2+K] = $s1`
Jumps and Conditional Branches	`beq $s1,$s2,K`	`if ($s1=$s2) goto PC + 4 + K`
	`slt $s1,$s2,$s3`	`if ($s2<$s3) $s1=1 else $s1=0`
	`j K`	`goto K`
Procedures	`jal K`	`$ra = PC + 4; goto K`
	`jr $ra`	`goto $ra`

The SPIM Assembler

Examples of Pseudo-Instruction Set

Instruction Group	Syntax	Translates to:
Arithmetic/ Logic	`neg $s1, $s2`	`sub $s1, $r0, $s2`
	`not $s1, $s2`	`nor $17, $18, $0`
Load/Store	`li $s1, K`	`ori $s1, $0, K`
	`la $s1, K`	`lui $at, 152` `ori $s1, $at, -27008`
	`move $s1, $s2`	
Jumps and Conditional Branches	`bgt $s1, $s2, K`	`slt $at, $s1, $s2` `bne $at, $0, K`

	`sge $s1, $s2, $s3`	`bne $s3, $s2, foo`
		`ori $s1, $0, 1`
		`beq $0, $0, bar`
		`foo: slt $s1, $s3, $s2`
		`bar:`

Pseudo Instructions: translated to native instructions by Assembler

The SPIM Assembler

Examples of Assembler Directives

Group	Directive	Function
Memory Segmentation	.data <addr>	Data Segment starting at
	.text <addr>	Text (program) Segment
	.stack <addr>	Stack Segment
	.ktext <addr>	Kernel Text Segment
	.kdata <addr>	Kernel Data Segment
Data Allocation	x: .word <value>	Allocates 32-bit variable
	x: .byte <value>	Allocates 8-bit variable
	x: .ascii "hello"	Allocates 8-bit cell array
Other	.globl x	x is external symbol

Assembler Directives: Provide assembler additional info to generate machine code

Handy MIPS ISA References

- Appendix A: Patterson & Hennessy
- SPIM ISA Summary on class website
- Patterson & Hennessy Back Cover

Computing Integer Division
Iterative C++ Version

MIPS/SPIM Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
   while (a >= b) {
     a = a - b;
     result ++;
   }
 }
}
```

C++

MIPS Assembly Language

```
                .data                       # Use HLL program as a comment
x:              .word   12                  # int x = 12;
y:              .word   4                   # int y = 4;
res:            .word   0                   # int res = 0;
                .globl  main
                .text
main:           la      $s0, x              # Allocate registers for globals
                lw      $s1, 0($s0)         #   x in $s1
                lw      $s2, 4($s0)         #   y in $s2
                lw      $s3, 8($s0)         #   res in $s3
while:          bgt     $s2, $s1, endwhile  # while (x >= y) {
                sub     $s1, $s1, $s2       #   x = x - y;
                addi    $s3, $s3, 1         #   res ++;
                j       while               # }
endwhile:
                la      $s0, x              # Update variables in memory
                sw      $s1, 0($s0)
                sw      $s2, 4($s0)
                sw      $s3, 8($s0)
```

Computing Integer Division
Iterative C++ Version

MIPS/SPIM Version
Input/Output in SPIM

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b) {
      a = a - b;
      result ++;
    }
  }
  print C++ ult = %d
}
```

C++

MIPS
Assembly Language

```
                .data                           # Use HLL program as a comment
x:              .word   12                      # int x = 12;
y:              .word   4                       # int y = 4;
res:            .word   0                       # int res = 0;
pf1:            .asciiz "Result = "
                .globl  main
                .text
main:           la      $s0, x                  # Allocate registers for globals
                lw      $s1, 0($s0)             #   x in $s1
                lw      $s2, 4($s0)             #   y in $s2
                lw      $s3, 8($s0)             #   res in $s3
while:          bgt     $s2, $s1, endwhile      # while (x >= y) {
                sub     $s1, $s1, $s2   #   x = x - y;
                addi    $s3, $s3, 1             #   res ++;
                j       while                   # }
endwhile:
                la      $a0, pf1                # printf("Result = %d \n");
                li      $v0, 4                  # //system call to print_str
                syscall
                move    $a0, $s3
                li      $v0, 1                  # //system call to print_int
                syscall
                la      $s0, x                  # Update variables in memory
                sw      $s1, 0($s0)
                sw      $s2, 4($s0)
                sw      $s3, 8($s0)
```

SPIM Assembler Abstractions

- Symbolic Labels
  - Instruction addresses and memory locations
- Assembler Directives
  - Memory allocation
  - Memory segments
- Pseudo-Instructions
  - Extend native instruction set without complicating arquitecture
- Macros

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(voi
  while (a >=
    a = a - b
    res ++;
  }
}
```

C++

MIPS Assembly Language

```
            .data
x:          .word   0
y:          .word   0
res:        .word   0
pf1:        .asciiz "Result = "
pf2:        .asciiz "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            #   res = 0;
            li          $s3, 0
            sw          $s3, 0($s0)
            jal         div                 #   div();
            lw          $s3, 0($s0)
            la          $a0, pf1            #   printf("Result = %d \n");
            li          $v0, 4              #   //system call to print_str
            syscall
            move        $a0, $s3
            li          $v0, 1              #   //system call to print_int
            syscall
            la          $a0, pf2            #   printf("Remainder = %d \n");
            li          $v0, 4              #   //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

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

C++

MIPS
Assembly Language

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

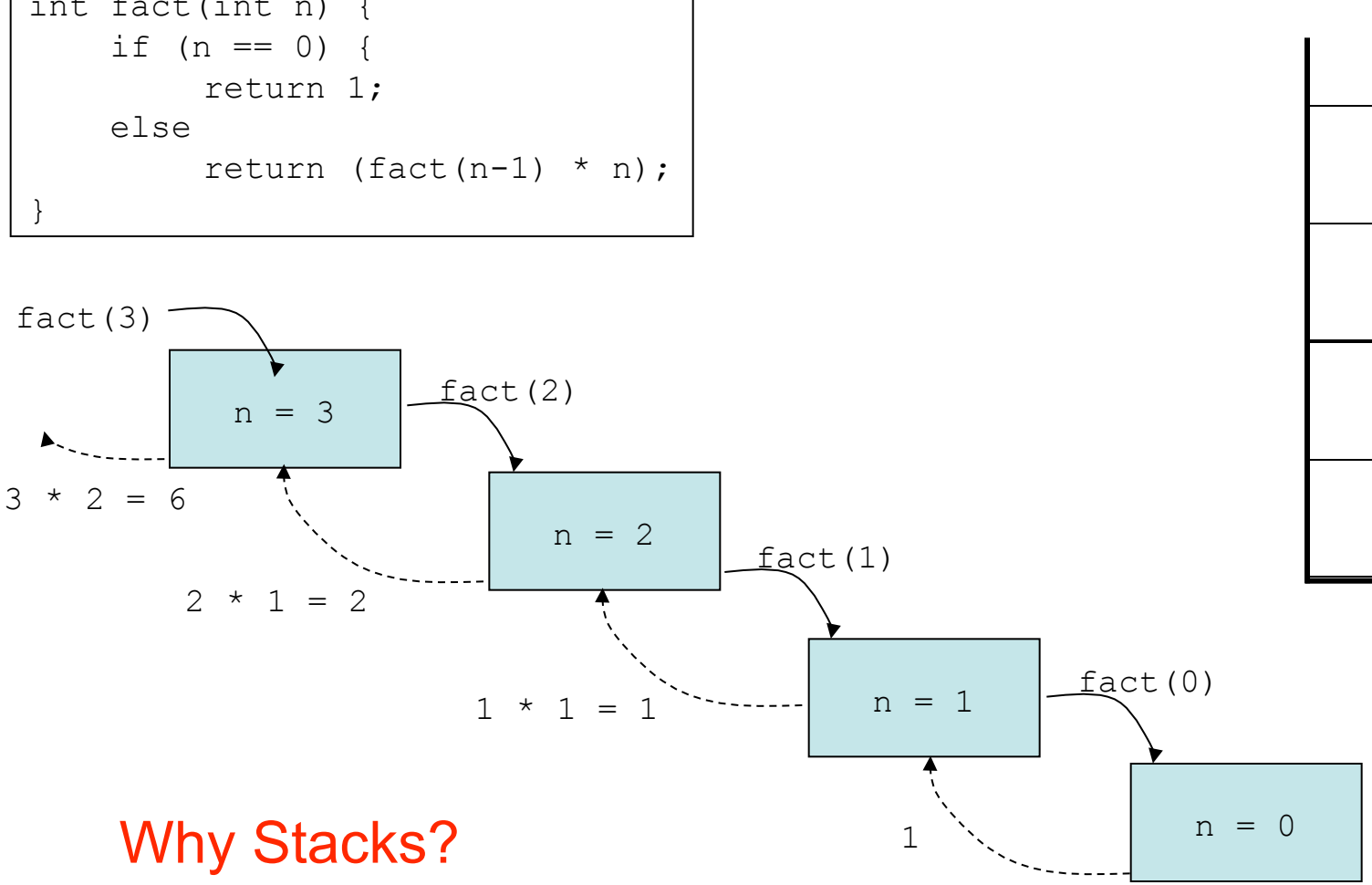
Function Return

Pending Problems With Linkage Approach I

- Registers shared by all procedures
  - procedures may overwrite each others registers
  - Solution?
- Procedures should be able to call other procedures
  - Procedures overwrite return address register
  - Solution?
- Lack of parameters forces access to globals
  - callee must know where parameters are stored
  - Solution?
- Need a convention for returning function values
  - Caller must know where return value is?
  - Solution?
- Recursion requires multiple copies of local data
  - Solution?

Recursion Basics

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

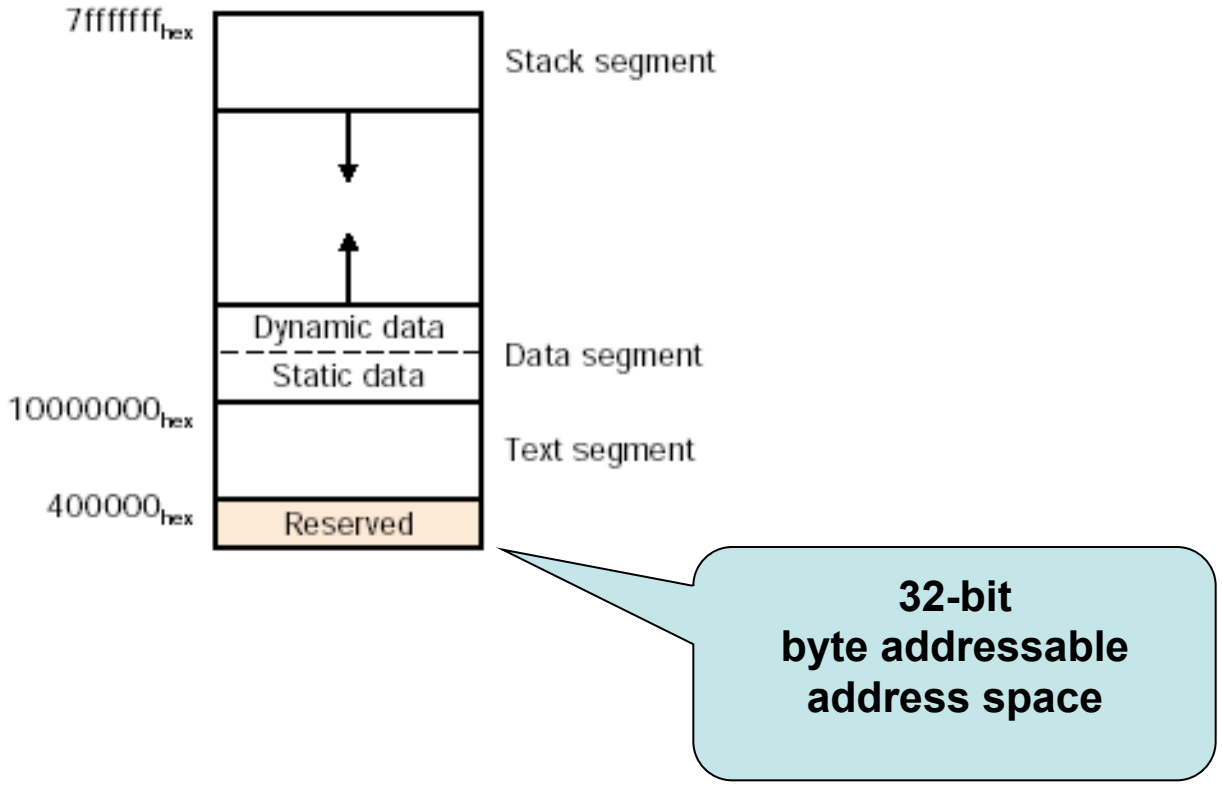


Why Stacks?

Pending Problems With Linkage Approach I

- Registers shared by all procedures
  - procedures may overwrite each others registers
  - procedures must save/restore registers in stack
- Procedures should be able to call other procedures
  - Procedures overwrite return address register
  - save multiple return addresses in stack
- Lack of parameters forces access to globals
  - callee must know where parameters are stored
  - pass parameters in registers and/or stack
- Need a convention for returning function values
  - Caller must know where return value is?
  - return values in registers
- Recursion requires multiple copies of local data
  - store multiple procedure activation records How many?

The MIPS Architecture
Memory Model



Review Of Stack Layout

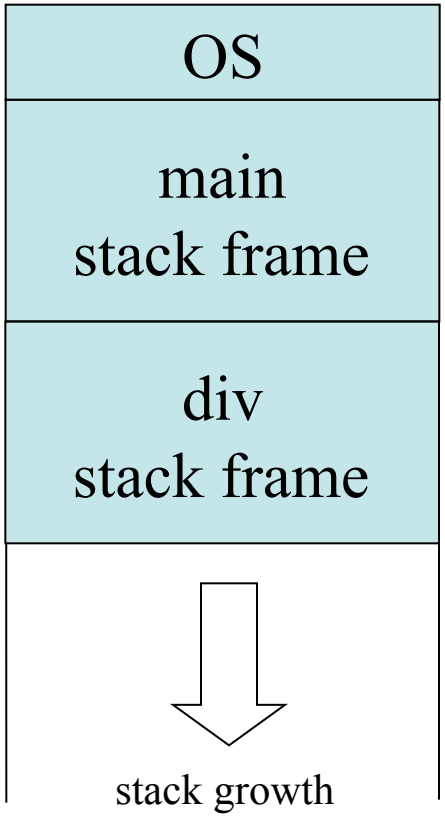
- Allocation strategies
  - Static
    - Code
    - Globals
    - *Own* variables
    - Explicit constants (including strings, sets, other aggregates)
    - Small scalars may be stored in the instructions themselves

Review Of Stack Layout

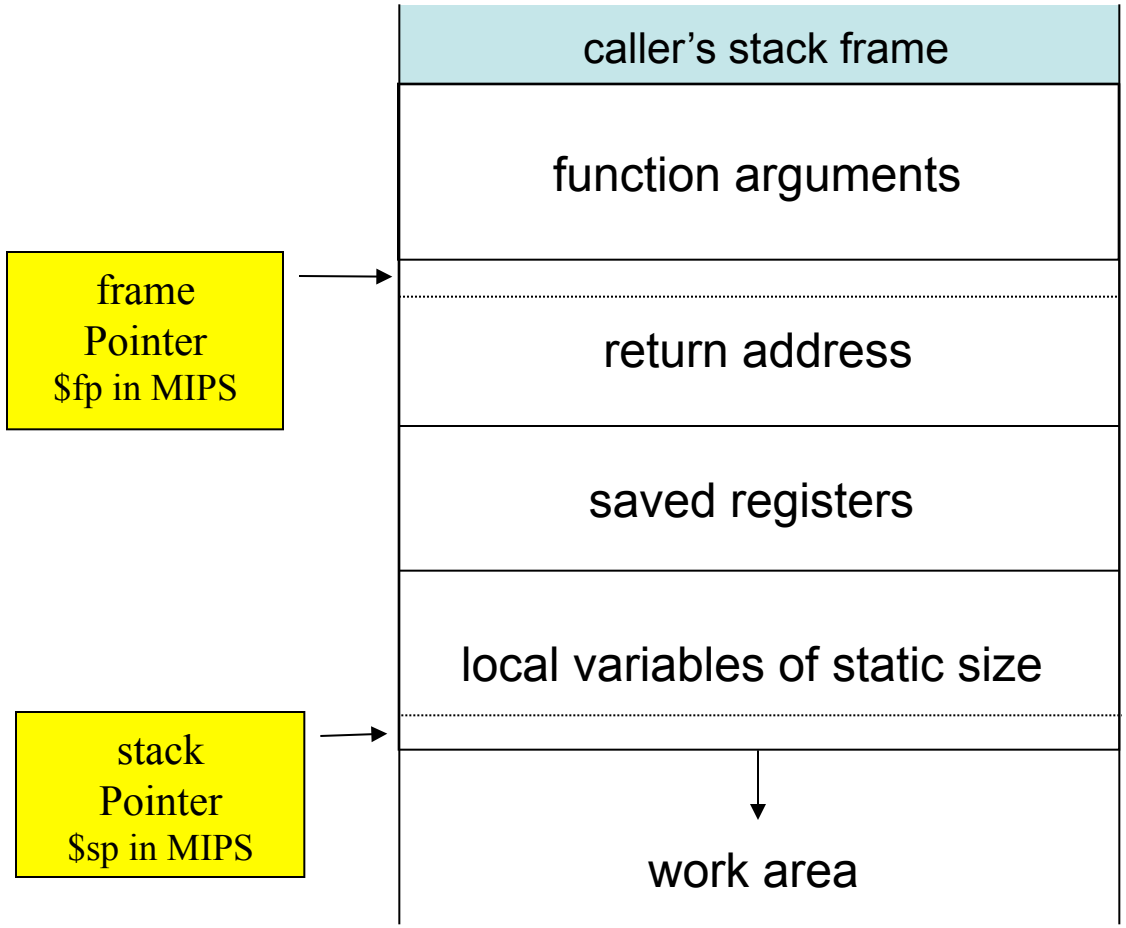
- Allocation strategies (2)
  - Stack
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation

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

Review Of Stack Layout

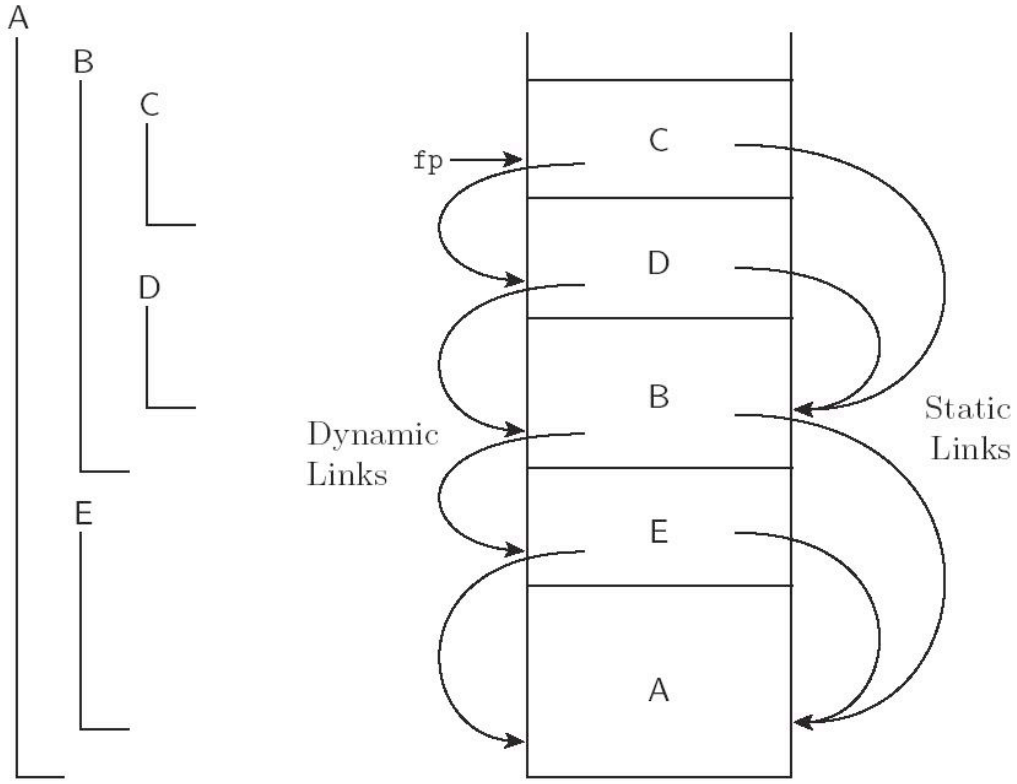


Figure 8.1: **Example of subroutine nesting, taken from Figure 3.5.** Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

Review Of Stack Layout

- Contents of a stack frame
  - bookkeeping
    - return PC (dynamic link)
    - saved registers
    - line number
    - saved display entries
    - static link
  - arguments and returns
  - local variables
  - temporaries

Calling Sequences

- Maintenance of stack is responsibility of *calling sequence* and *subroutine prolog* and *epilog* – discussed in Chapter 3
  - space is saved by putting as much in the prolog and epilog as possible
  - time *may* be saved by putting stuff in the caller instead, where more information may be known
    - e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee

Calling Sequences

- Common strategy is to divide registers into *caller-saves* and *callee-saves* sets
  - caller uses the "callee-saves" registers first
  - "caller-saves" registers if necessary
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
  - some storage layouts use a separate arguments pointer
  - the VAX architecture encouraged this

Calling Sequences

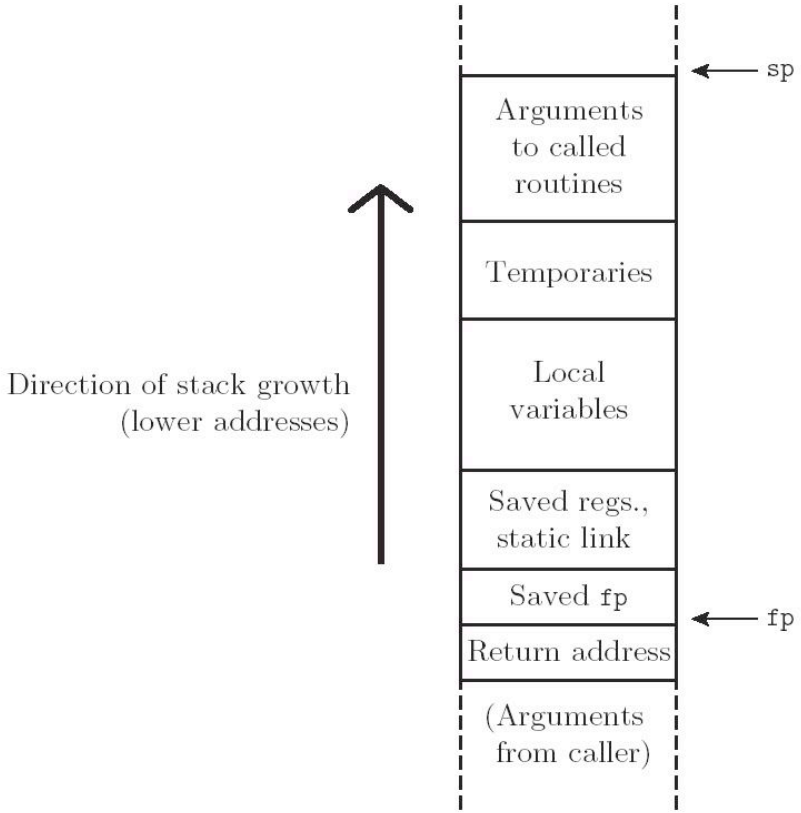


Figure 8.2: **A typical stack frame.** Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the `fp`. Local variables and temporaries are accessed at negative offsets from the `fp`. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the `sp`.

Calling Sequences (C on MIPS)

- Caller
  - saves into the temporaries and locals area any caller-saves registers whose values will be needed after the call
  - puts up to 4 small arguments into registers $4-$7 (a0-a3)
    - it depends on the types of the parameters and the order in which they appear in the argument list
  - puts the rest of the arguments into the arg build area at the top of the stack frame
  - does jal, which puts return address into register ra and branches
    - note that jal, like all branches, has a delay slot

Calling Sequences (C on MIPS)

- In prolog, Callee
  - subtracts framesize from sp
  - saves callee-saves registers used anywhere inside callee
  - copies sp to fp
- In epilog, Callee
  - puts return value into registers (mem if large)
  - copies fp into sp (see below for rationale)
  - restores saved registers using sp as base
  - adds to sp to deallocate frame
  - does jra

Calling Sequences (C on MIPS)

- After call, Caller
  - moves return value from register to wherever it's needed (if appropriate)
  - restores caller-saves registers lazily over time, as their values are needed
- All arguments have space in the stack, whether passed in registers or not
- The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory

Calling Sequences (C on MIPS)

- This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope

Calling Sequences (C on MIPS)

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  - particularly LEAF routines (those that don't call other routines)
    - leaving things out saves time
    - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast

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          $s1, 24($sp)            #   deallocate a from $s1
            sw          $s2, 20($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

Parameter Passing

- Parameter passing mechanisms have three basic implementations
  - *value*
  - *value/result* (copying)
  - *reference* (aliasing)
  - *closure/name*
- Many languages (e.g., Pascal) provide value and reference directly

Parameter Passing

- C/C++: functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x,int y){*x = *x+y } …
proc(&a,b);
```

  - or directly passed by reference (C++)

```
void proc(int& x, int y) {x = x + y }
proc(a,b);
```

Parameter Passing

- Ada goes for semantics: who can do what
  - *In*: callee reads only
  - *Out*: callee writes and can then read (formal not initialized); actual modified
  - *In out*: callee reads and writes; actual modified
- Ada in/out is always implemented as
  - value/result for scalars, and either
  - value/result or reference for structured objects

Parameter Passing

- In a language with a reference model of variables (Lisp, Clu), pass by reference (*sharing*) is the obvious approach
- It's also the only option in Fortran
  - If you pass a constant, the compiler creates a temporary location to hold it
  - If you modify the temporary, who cares?
- Call-by name is an old Algol technique
  - Think of it as call by textual substitution (procedure with all name parameters works like macro) - what you pass are hidden procedures called THUNKS

Parameter Passing

	implementation mechanism	permissible operations	change to actual?	alias?
value	value	read, write	no	no
`in`, `const`	value or reference	read only	no	maybe
`out` (Ada)	value or reference	write only	yes	maybe
value/result	value	read, write	yes	no
`var`, `ref`	reference	read, write	yes	yes
sharing	value or reference	read, write	yes	yes
`in out` (Ada)	value or reference	read, write	yes	maybe
name (Algol 60)	closure (thunk)	read, write	yes	yes

Figure 8.3: **Parameter passing modes.** Column 1 indicates common names for modes. Column 2 indicates implementation via passing of values, references, or closures. Column 3 indicates whether the callee can read or write the formal parameter. Column 4 indicates whether changes to the formal parameter affect the actual parameter. Column 5 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other.

Generic Subroutines and Modules

- Generic modules or classes are particularly valuable for creating *containers*: data abstractions that hold a collection of objects
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right

What is an Exception?

- Something that should not happen under normal or typical circunstances
- Programmer knows that it can happen
- Programmer cannot predict when will it happen
- Program MUST be prepared to handle it

Exception Handling Principles

Some sources of exceptions:

- Wrong user behavior (intentional or not)
- Non-existent input files
- Badly formatted input files
- Overflow conditions

Are programmer mistakes exceptions?

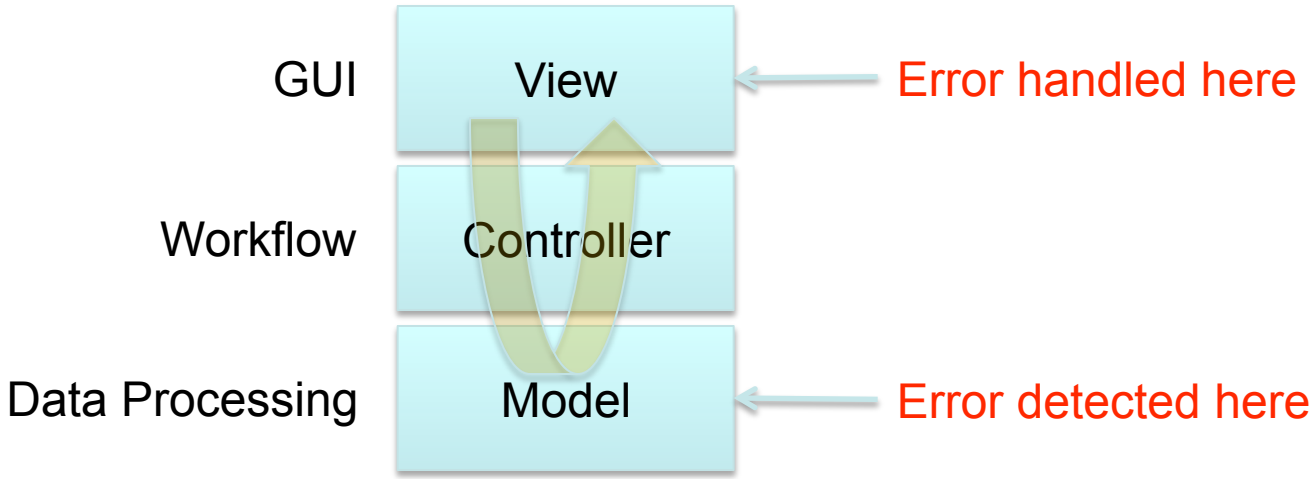
Exception Handling Principles

Approaches to exception handling:

- Without language support
  - Set global variable
  - Error condition reference parameters
  - Error return values
  - Error handling subroutines
- With language support
  - Exception classes (C++, Java)

Exception Handling Principles

Challenge of exception handling:



Java: unhandled exceptions are automatically propagated up the procedure call chain

Exception Handling Principles

Advantages of language supported exception handling:

- Keep exceptional and normal code separate
- Uniform mechanism
- Automatically propagate exception to place where it can best be handled

Exception Handling

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors

Catching Exceptions

- Example:

```
try
{
   String filename = . . .;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader); String input =
         in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception
- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions
  - explicitly in the program everywhere they might occur

Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name
- Coroutines can be used to implement
  - iterators (Section 6.5.3)
  - threads (to be discussed in Chapter 12)
- Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Coroutines

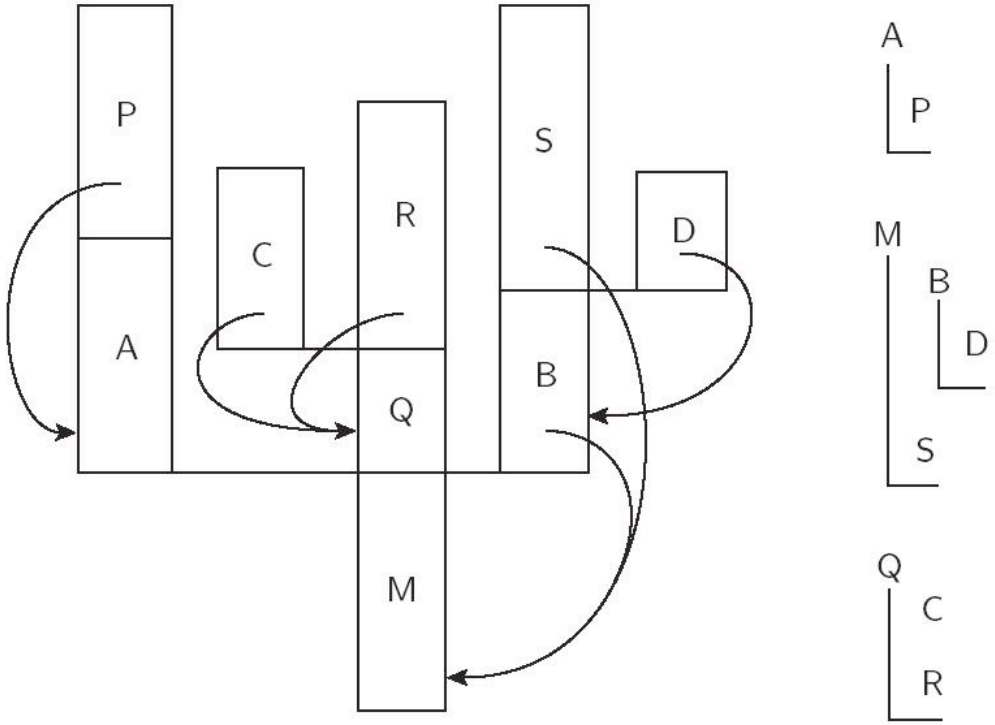


Figure 8.5: **A cactus stack.** Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it.