Procedures and the Stack

Chapter 4

S. Dandamudi

Outline

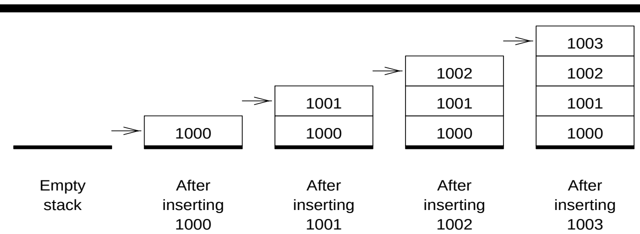
- What is stack?
- Pentium implementation of stack
- Pentium stack instructions
- Uses of stack
- Procedures
  - Assembler directives
  - Pentium instructions
- Parameter passing
  - Register method
  - Stack method
- Examples
  - Call-by-value
  - Call-by-reference
  - Bubble sort
- Procedures with variable number of parameters
- Local variables
- Multiple source program modules
- Performance: Procedure overheads

What is a Stack?

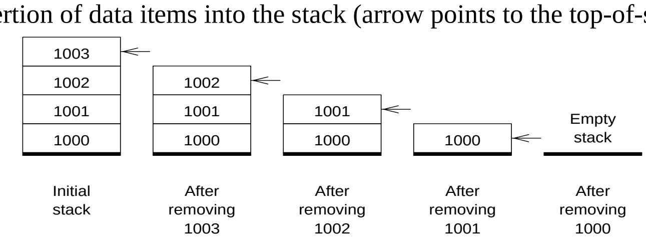
- Stack is a last-in-first-out (LIFO) data structure
- If we view the stack as a linear array of elements, both insertion and deletion operations are restricted to one end of the array
- Only the element at the top-of-stack (TOS) is directly accessible
- Two basic stack operations:
  - push (insertion)
  - pop (deletion)

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Stack Example



Insertion of data items into the stack (arrow points to the top-of-stack)



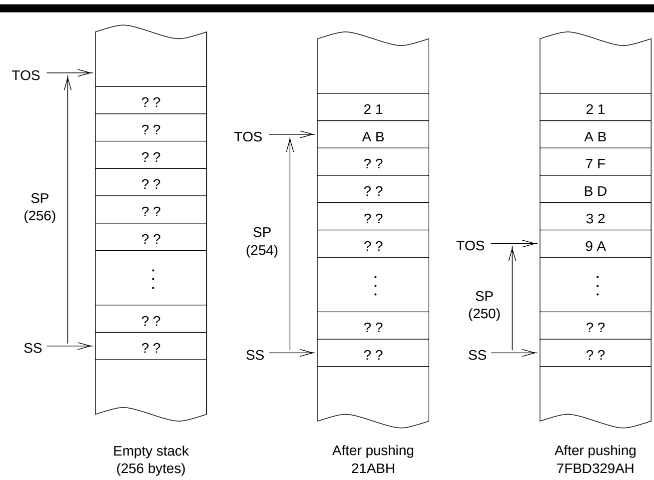
Deletion of data items from the stack (arrow points to the top-of-stack)

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Pentium Implementation of the Stack

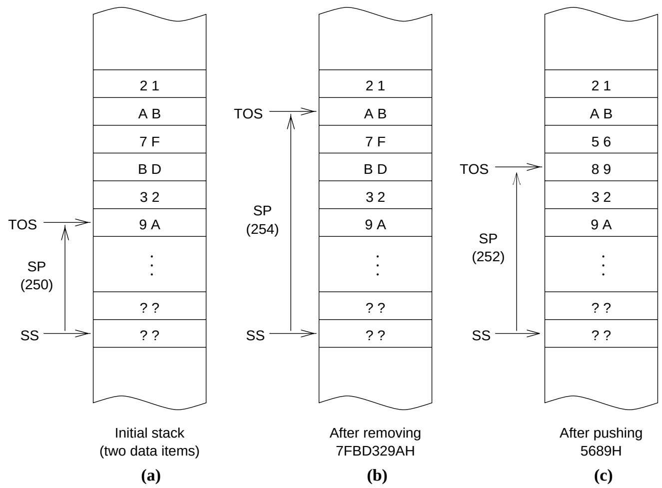
- Stack segment is used to implement the stack
  - Registers SS and (E)SP are used
  - SS:(E)SP represents the top-of-stack
- Pentium stack implementation characteristics are:
  - Only words (i.e., 16-bit data) or doublewords (i.e., 32-bit data) are saved on the stack, never a single byte
  - Stack grows toward lower memory addresses (i.e., stack grows "downward")
  - Top-of-stack (TOS) always points to the last data item placed on the stack

Pentium Stack Example - 1



(a) (b) (c)

Pentium Stack Example - 2



Pentium Stack Instructions

- Pentium provides two *basic* instructions:

```
push    source
pop     destination
```

- **`source`** and **`destination`** can be a
  - 16- or 32-bit general register
  - a segment register
  - a word or doubleword in memory
- **`source`** of **`push`** can also be an *immediate operand* of size 8, 16, or 32 bits

Pentium Stack Instructions: Examples

- On an empty stack created by

```
.STACK 100H
```

the following sequence of **push** instructions

```
push    21ABH
push    7FBD329AH
```

results in the stack state shown in (a) in the last figure

- On this stack, executing

```
pop     EBX
```

results in the stack state shown in (b) in the last figure and the register EBX gets the value 7FBD329AH

Additional Pentium Stack Instructions

Stack Operations on Flags

- **push** and **pop** instructions cannot be used with the Flags register
- Two special instructions for this purpose are

```
pushf (push 16-bit flags)
popf  (pop 16-bit flags)
```

- No operands are required
- Use **pushfd** and **popfd** for 32-bit flags (EFLAGS)

Additional Pentium Stack Instructions (cont'd)

Stack Operations on 8 General-Purpose Registers

- **`pusha`** and **`popa`** instructions can be used to save and restore the eight general-purpose registers

  AX, CX, DX, BX, SP, BP, SI, and DI
- **`pusha`** pushes these eight registers in the above order (AX first and DI last)
- **`popa`** restores these registers except that SP value is not loaded into the SP register
- Use **`pushad`** and **`popad`** for saving and restoring 32-bit registers

Uses of the Stack

- Three main uses
  - » Temporary storage of data
  - » Transfer of control
  - » Parameter passing

Temporary Storage of Data

Example: Exchanging **`value1`** and **`value2`** can be done by using the stack to temporarily hold data

```
push    value1
push    value2
pop     value1
pop     value2
```

Uses of the Stack (cont'd)

- Often used to free a set of registers

```
;save EBX & ECX registers on the stack
     push    EBX
     push    ECX
     . . . . . .
     <<EBX and ECX can now be used>>
     . . . . . .
;restore EBX & ECX from the stack
     pop     ECX
     pop     EBX
```


Uses of the Stack (cont'd)

Transfer of Control

- In procedure calls and interrupts, the return address is stored on the stack
- Our discussion on procedure calls clarifies this particular use of the stack

Parameter Passing

- Stack is extensively used for parameter passing
- Our discussion later on parameter passing describes how the stack is used for this purpose

Assembler Directives for Procedures

- Assembler provides two directives to define procedures: PROC and ENDP
- To define a NEAR procedure, use

```
proc-name    PROC   NEAR
```

  - ∗ In a NEAR procedure, both calling and called procedures are in the same code segment
- A FAR procedure can be defined by

```
proc-name    PROC   FAR
```

  - ∗ Called and calling procedures are in two different segments in a FAR procedure

Assembler Directives for Procedures (cont'd)

- If FAR or NEAR is not specified, NEAR is assumed (i.e., NEAR is the default)
- We focus on NEAR procedures
- A typical NAER procedure definition

```
proc-name    PROC
    . . . . .
<procedure body>
    . . . . .
proc-name    ENDP
```

proc-name should match in PROC and ENDP

Pentium Instructions for Procedures

- Pentium provides two instructions: **`call`** and **`ret`**
- **`call`** instruction is used to invoke a procedure
- The format is

```
call    proc-name
```

`proc-name` is the procedure name

- Actions taken during a near procedure call:

```
SP := SP - 2           ; push return address
(SS:SP) := IP          ;       onto the stack
IP := IP + relative displacement    ; update IP
                       ; to point to the procedure
```


Pentium Instructions for Procedures (cont'd)

- **`ret`** instruction is used to transfer control back to the calling procedure
- How will the processor know where to return?
  - Uses the return address pushed onto the stack as part of executing the **`call`** instruction
  - Important that TOS points to this return address when **`ret`** instruction is executed
- Actions taken during the execution of **`ret`** are:

```
IP := (SS:SP)          ; pop return address
SP := SP + 2           ;       from the stack
```

Pentium Instructions for Procedures (cont'd)

- We can specify an optional integer in the **`ret`** instruction
  - ∗ The format is

    ```
    ret    optional-integer
    ```
  - ∗ Example:

    ```
    ret 6
    ```
- Actions taken on **`ret`** with optional-integer are:

  IP := (SS:SP)
  SP := SP + 2 + optional-integer

How Is Program Control Transferred?

```
Offset(hex)  machine code(hex)
                             main     PROC
                             . . . . . .
cs:000A      E8000C          call     sum
cs:000D      8BD8            mov      BX,AX
                             . . . . . .
                             main     ENDP
-----------------------------------------------
                             sum      PROC
cs:0019      55              push     BP
                             . . . . . .
                             sum      ENDP
-----------------------------------------------
                             avg      PROC
                             . . . . . .
cs:0028      E8FFEE          call     sum
cs:002B      8BD0            mov      DX,AX
                             . . . . . .
                             avg      ENDP
```

Parameter Passing

- Parameter passing is different and complicated than in a high-level language
- In assembly language
  - » You should first place all required parameters in a mutually accessible storage area
  - » Then call the procedure
- Type of storage area used
  - » Registers (general-purpose registers are used)
  - » Memory (stack is used)
- Two common methods of parameter passing:
  - » Register method
  - » Stack method

Parameter Passing: Register Method

- Calling procedure places the necessary parameters in the general-purpose registers before invoking the procedure through the `call` instruction
- Examples:
  - * **PROCEX1.ASM**
    - » call-by-value using the register method
    - » a simple sum procedure
  - * **PROCEX2.ASM**
    - » call-by-reference using the register method
    - » string length procedure

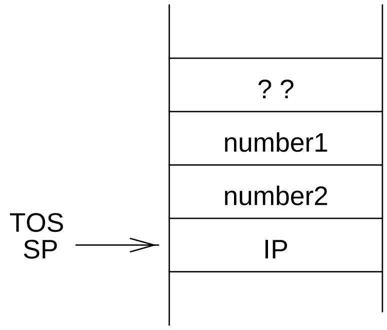
Pros and Cons of the Register Method

- Advantages
  - ∗ Convenient and easier
  - ∗ Faster
- Disadvantages
  - ∗ Only a few parameters can be passed using the register method
    - – Only a small number of registers are available
  - ∗ Often these registers are not free
    - – freeing them by pushing their values onto the stack negates the second advantage

Parameter Passing: Stack Method

- All parameter values are pushed onto the stack before calling the procedure
- Example:

```
push    number1
push    number2
call    sum
```



Accessing Parameters on the Stack

- Parameter values are buried inside the stack
- We cannot use

```
mov    BX,[SP+2]  ;illegal
```

  to access **number2** in the previous example

- We can use

```
mov    BX,[ESP+2]  ;valid
```

  Problem: The ESP value changes with **push** and **pop** operations

  » Relative offset depends of the stack operations performed

  » Not desirable

Accessing Parameters on the Stack (cont'd)

- We can also use

```
add    SP,2
mov    BX,[SP]    ;valid
```

  Problem: cumbersome

  » We have to remember to update SP to point to the return address on the stack before the end of the procedure

- Is there a better alternative?
  ∗ Use the BP register to access parameters on the stack

Using BP Register to Access Parameters

- Preferred method of accessing parameters on the stack is

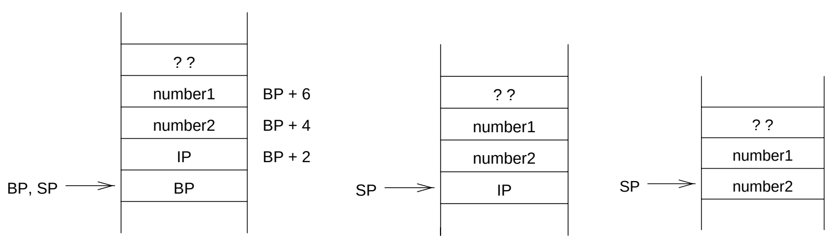
```
mov    BP,SP
mov    BX,[BP+2]
```

  to access **number2** in the previous example

- Problem: BP contents are lost!
  - We have to preserve the contents of BP
  - Use the stack (caution: offset value changes)

```
push   BP
mov    BP,SP
```


Clearing the Stack Parameters



Stack state after pushing BP

Stack state after pop BP

Stack state after executing **ret**

Clearing the Stack Parameters (cont'd)

- Two ways of clearing the unwanted parameters on the stack:
  - ∗ Use the optional-integer in the **ret** instruction
    - » Use

```
ret    4
```

in the previous example

  - ∗ Add the constant to SP in calling procedure (C uses this method)

```
push    number1
push    number2
call    sum
add    SP,4
```


Housekeeping Issues

- Who should clean up the stack of unwanted parameters?
  - ∗ Calling procedure
    - » Need to update SP with every procedure call
    - » Not really needed if procedures use fixed number of parameters
    - » C uses this method because C allows variable number of parameters
  - ∗ Called procedure
    - » Code becomes modular (parameter clearing is done in only one place)
    - » Cannot be used with variable number of parameters

Housekeeping Issues (cont'd)

- Need to preserve the state (contents of the registers) of the calling procedure across a procedure call.
      » Stack is used for this purpose
- Which registers should be saved?
   * Save those registers that are used by the calling procedure but are modified by the called procedure
      » Might cause problems as the set of registers used by the calling and called procedures changes over time
   * Save all registers (brute force method) by using **`pusha`**
      » Increased overhead (**`pusha`** takes 5 clocks as opposed 1 to save a register)

Housekeeping Issues (cont'd)

- Who should preserve the state of the calling procedure?
   * Calling procedure
      » Need to know the registers used by the called procedure
      » Need to include instructions to save and restore registers with every procedure call
      » Causes program maintenance problems
   * Called procedure
      » Preferred method as the code becomes modular (state preservation is done only once and in one place)
      » Avoids the program maintenance problems mentioned

A Typical Procedure Template

```
proc-name    PROC
             push    BP
             mov     BP,SP
               . . . . . .
             <procedure body>
               . . . . . .
             pop     BP
             ret     integer-value
proc-name    ENDP
```


Stack Parameter Passing: Examples

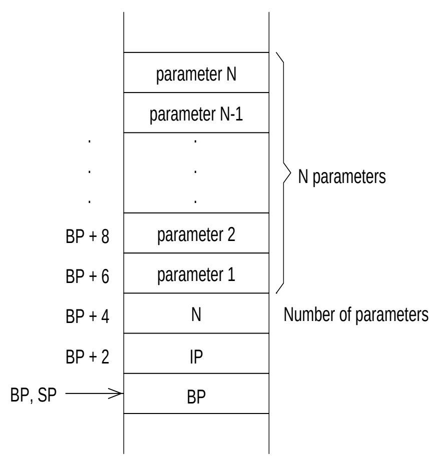
- **PROCEX3.ASM**
  - call-by-value using the stack method
  - a simple sum procedure
- **PROCSWAP.ASM**
  - call-by-reference using the stack method
  - first two characters of the input string are swapped
- **BBLSORT.ASM**
  - implements bubble sort algorithm
  - uses **pusha** and **popa** to save and restore registers

Variable Number of Parameters

- For most procedures, the number of parameters is fixed (i.e., every time the procedure is called, the same number of parameter values are passed)
- In procedures that can have variable number of parameters, with each procedure call, the number of parameter values passed can be different
- C supports procedures with variable number of parameters
- Easy to support variable number of parameters using the stack method

Variable Number of Parameters (cont'd)

- To implement variable number of parameter passing:
  * Parameter count should be one of the parameters passed onto the called procedure
  * This count should be the last parameter pushed onto the stack so that it is just below IP independent of the number of parameters passed



To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

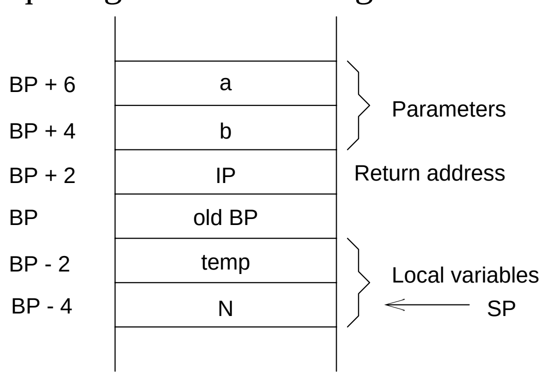
Local Variables

- Local variables are dynamic in nature
  - ∗ Local variables of a procedure come into existence when the procedure is invoked and disappear when the procedure terminates.
- Cannot reserve space for these variable in the data segment for two reasons:
  - » Such space allocation is static (remains active even when the procedure is not)
  - » It does not work with recursive procedures
- For these reasons, space for local variables is reserved on the stack

Local Variables (cont'd)

Example

- Assume that **N** and **temp** of two local variables, each requiring 16 bits of storage



Local Variables (cont'd)

- The information stored in the stack
  - » parameters
  - » returns address
  - » old BP value
  - » local variables

  is collectively called *stack frame*
- In high-level languages, stack frame is also referred to as the *activation record*
  - » Because each procedure activation requires all this information
- The BP value is referred to as the *frame pointer*
  - » Once the BP value is known, we can access all the data in the stack frame

Local Variables: Examples

- **`PROCFIB1.ASM`**
  - ∗ For simple procedures, registers can also be used for local variable storage
  - ∗ Uses registers for local variable storage
  - ∗ Outputs the largest Fibonacci number that is less than the given input number
- **`PROCFIB2.ASM`**
  - ∗ Uses the stack for local variable storage
  - ∗ Performance implications of using registers versus stack are discussed later

Multiple Module Programs

- In multi-module programs, a single program is split into multiple source files
- Advantages
  - » If a module is modified, only that module needs to be reassembled (not the whole program)
  - » Several programmers can share the work
  - » Making modifications is easier with several short files
  - » Unintended modifications can be avoided
- To facilitate separate assembly, two assembler directives are provided:
  - » PUBLIC and EXTRN

PUBLIC Assembler Directive

- The PUBLIC directive makes the associated labels public
  - » Makes these labels available for other modules of the program
- The format is

```
PUBLIC    label1, label2, . . .
```

- Almost any label can be made public including
  - » procedure names
  - » variable names
  - » equated labels
- In the PUBLIC statement, it is not necessary to specify the type of label

Example: PUBLIC Assembler Directive

```
                . . . . .
PUBLIC    error_msg, total, sample
                . . . . .
.DATA
error_msg    DB    "Out of range!",0
total        DW    0
                . . . . .
.CODE
                . . . . .
sample    PROC
                . . . . .
sample    ENDP
                . . . . .
```


EXTRN Assembler Directive

- The EXTRN directive tells the assembler that certain labels are not defined in the current module
- The assembler leaves "holes" in the OBJ file for the linker to fill in later on
- The format is

  ```
  EXTRN    label:type
  ```

  where **`label`** is a label made public by a PUBLIC directive in some other module and **`type`** is the type of the label

EXTRN Assembler Directive (cont'd)

Type	Description
UNKNOWN	Undetermined or unknown type
BYTE	Data variable (size is 8 bits)
WORD	Data variable (size is 16 bits)
DWORD	Data variable (size is 32 bits)
QWORD	Data variable (size is 64 bits)
FWORD	Data variable (size is 6 bytes)
TBYTE	Data variable (size is 10 bytes)
PROC	A procedure name (NEAR or FAR according to .MODEL)
NAER	A near procedure name
FAR	A far procedure name

EXTRN Assembler Directive (cont'd)

Example

```
.MODEL   SMALL
         . . . .
EXTRN    error_msg:BYTE, total:WORD
EXTRN    sample:PROC
         . . . .
```

Note: EXTRN (not EXTERN)

Example

- `module1.asm` (main procedure)
- `module2.asm` (string length procedure)

Performance: Procedure Overheads

Stack versus Registers

- *AL-original* (AX is not preserved)

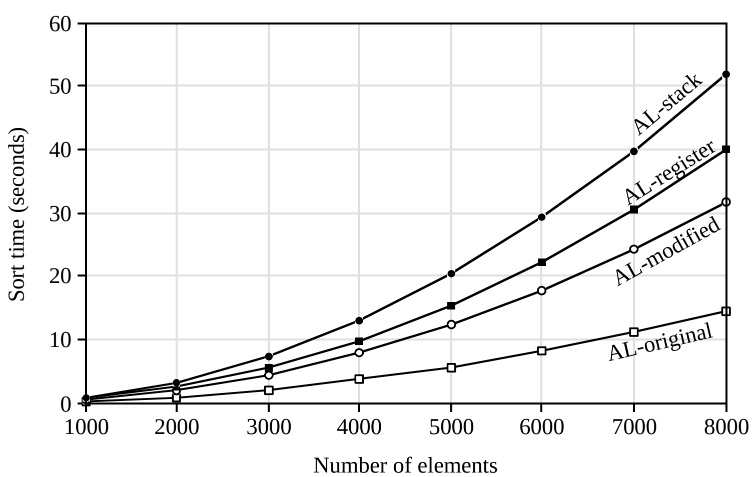
```
;AX contains the element pointed to by SI
xchg   AX,[SI+2]
mov    [SI],AX
```

- *AL-modified* (AX is preserved)

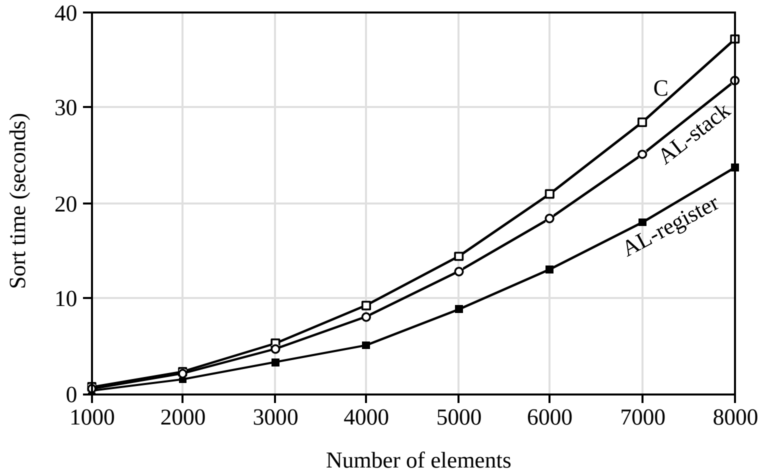
```
xchg   AX,[SI+2]
xchg   AX,[SI]
xchg   AX,[SI+2]
```

- Separate swap procedure
  - *AL-register* (register method of parameter passing)
  - *AL-stack* (stack method of parameter passing)

Performance: Procedure Overheads (cont'd)



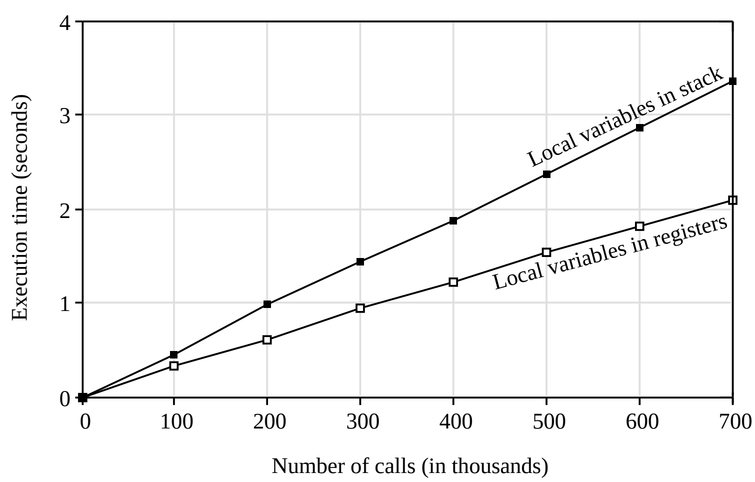
Performance: C versus Assembly



1998 © S. Dandamudi Procedures: Page 49

To be used with S. Dandamudi, “Introduction to Assembly Language Programming,” Springer-Verlag, 1998.

Performance: Local Variable Overhead



1998 © S. Dandamudi Procedures: Page 50

To be used with S. Dandamudi, “Introduction to Assembly Language Programming,” Springer-Verlag, 1998.