Procedures and the Stack

Chapter 4

S. Dandamudi

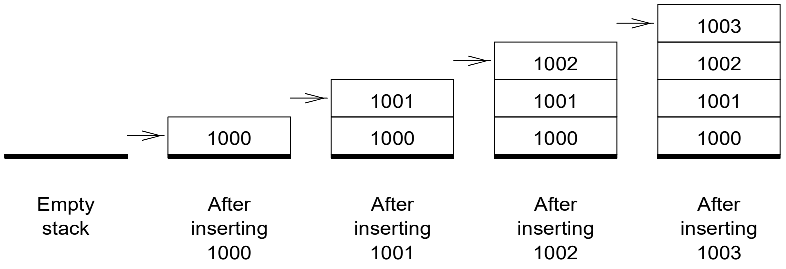
Outline

- 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

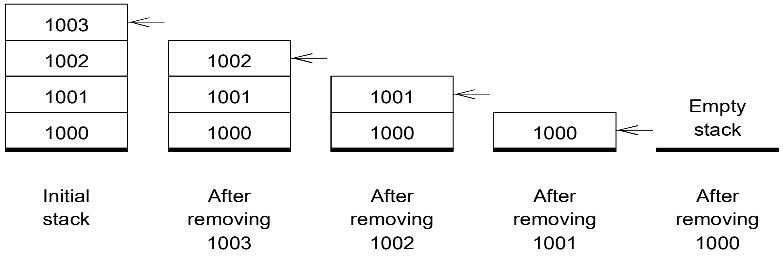
What is a Stack?

- 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)

Stack Example



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

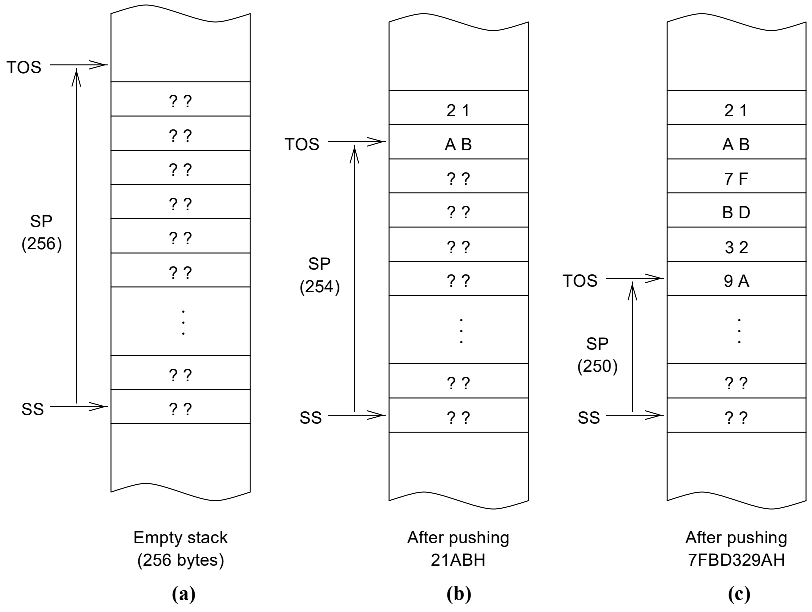


Deletion of data items from the stack (arrow points to the top-of-stack)

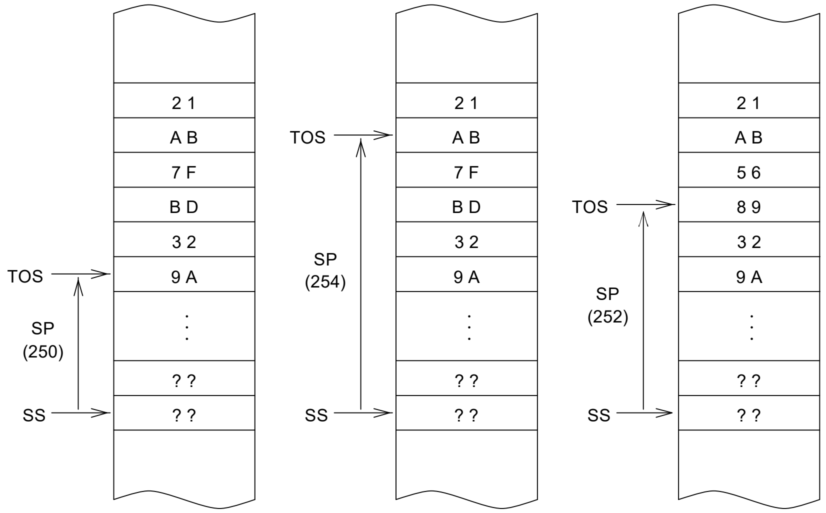
Pentium Implementation of the Stack

- 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



Pentium Stack Example - 2



Initial stack
(two data items)
(a)

After removing
7FBD329AH
(b)

After pushing
5689H
(c)

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