The Stack and Procedures

Assembly Language Programming

What Is A Stack?

- LIFO data structure
  - supports PUSH and POP operations
- 8086 Architecture Stack
  - Array based implementation
  - Dedicated register tracks TOS
  - Stack addressing modes use the BP register as an offset into the stack
    - allows random access of stack contents

Why Have a Stack?

- The 8086 processor has stack instructions
- The processor uses the stack when interrupts strike
- Procedure calls use the stack for return addresses
- It is convenient to have one around for temporary storage

Where Is The Stack?

- All executables must define a stack segment
  - The stack is an array of bytes accessed via the stack segment register and an offset
- SS points to the beginning of this memory area
- SP is the offset to the top of the stack
  - The loader sets these registers before execution begins

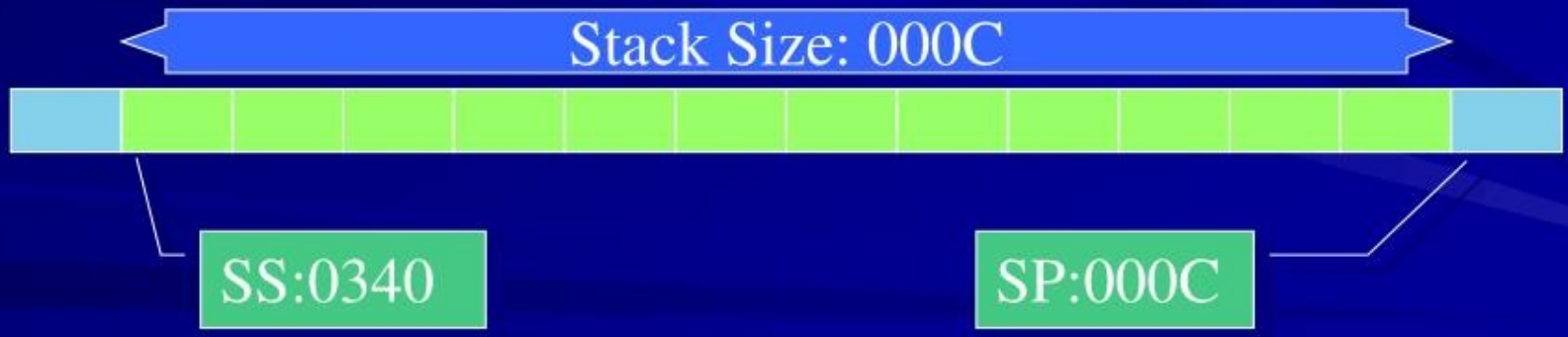
Stack Initialization

- The .stack directive hides an array allocation statement that looks like this
  - `The_Stack  DB  Stack_Size dup (?)`
- On program load…
  - SS is set to a segment address containing this array (usually `The_Stack` starts at offset 0)
  - SP is set to the offset of `The_Stack+Stack_Size` which is one byte past the end of the stack array
    - This is the condition for an empty stack

Initial Stack Configuration

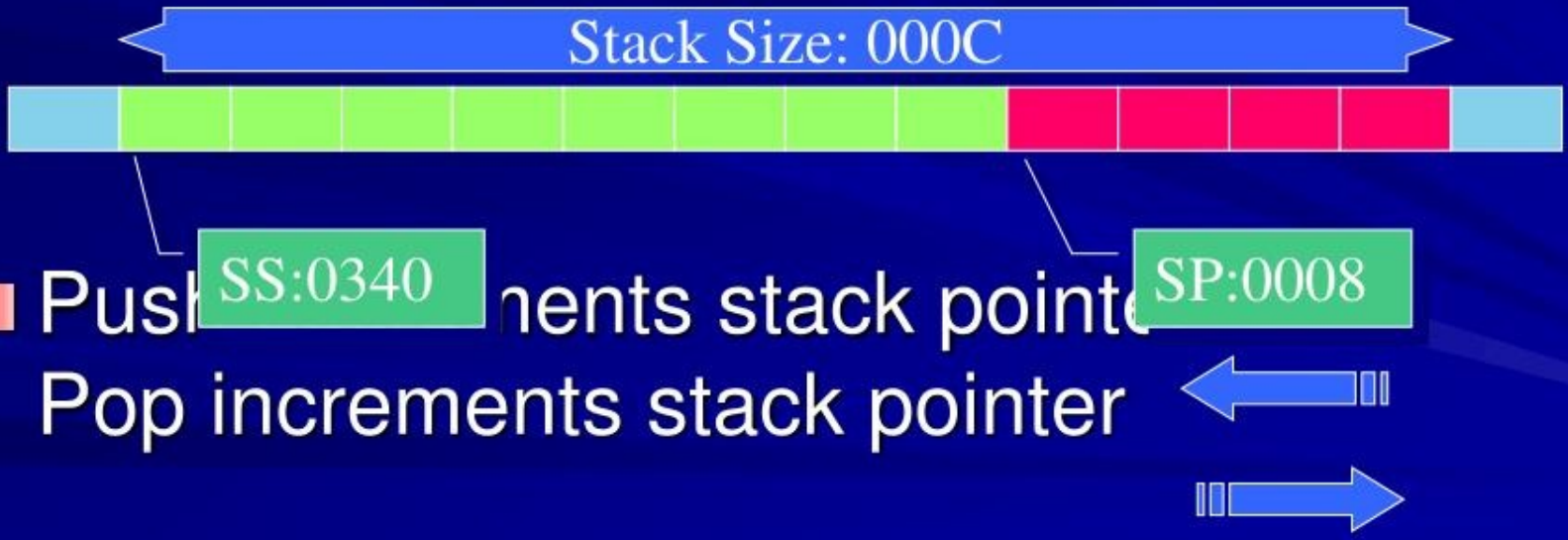
.stack 12 ;Reserve space for the stack

- Loader determines actual segment address for the start of the stack
  - This is an empty stack



How Does The Stack Work?

- The stack grows backwards through memory towards the start of the stack segment

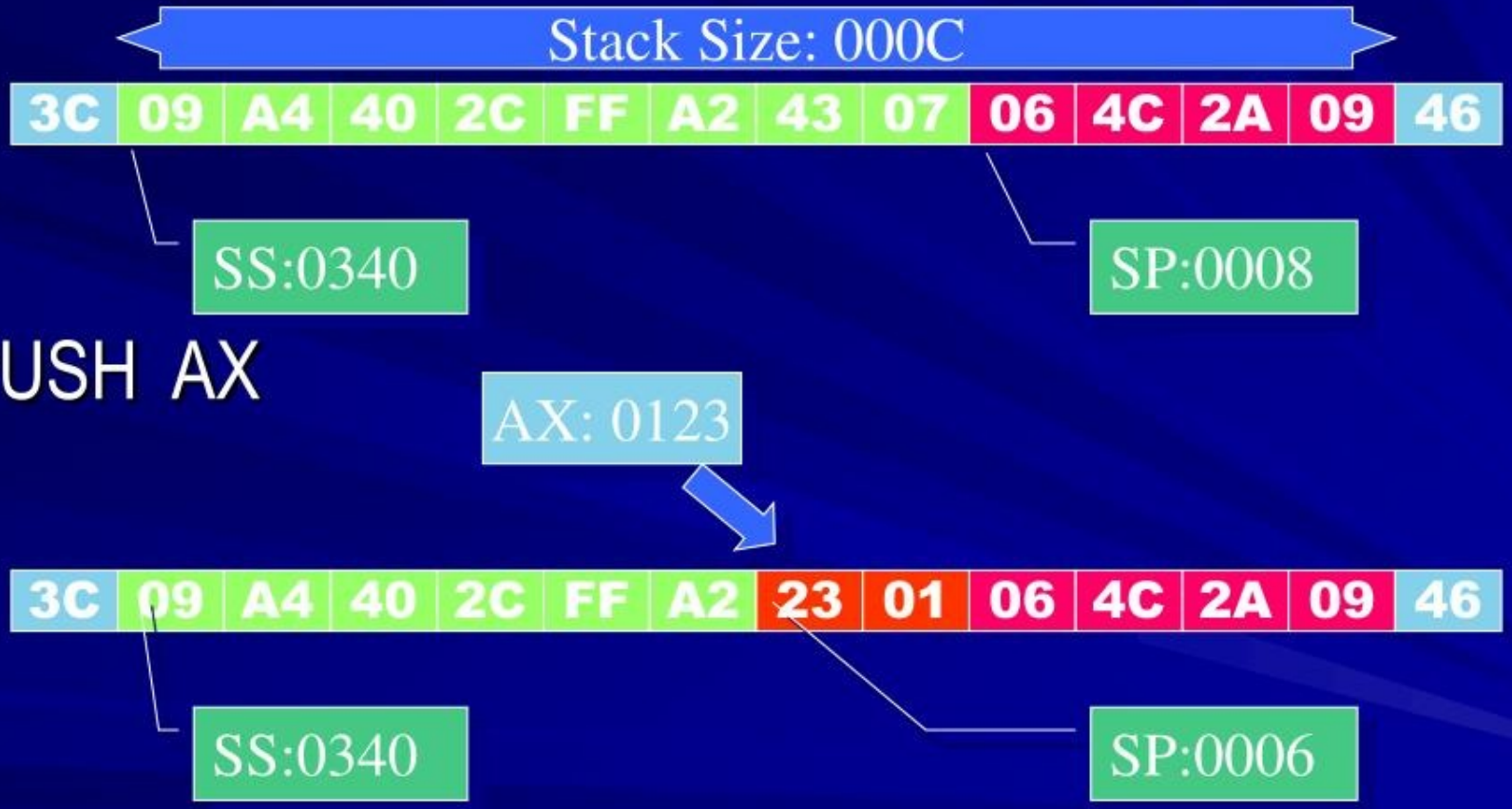


- Push decrements stack pointer
  Pop increments stack pointer

PUSH

- PUSH *source*
  - source is (almost) any 16/32-bit general or segment register or the address of a word or doubleword
- PUSHF or PUSHFD
  - Pushes the FLAGS register onto the stack
- A PUSH instruction subtracts 2/4 from SP and then stores the *source* data at SS:SP

PUSH Illustration



POP

- POP *destination*
  - destination is (almost) any 16/32-bit general or segment register or the address of a word or doubleword
- POPF or POPFD
  - Pops the top of stack to the FLAGS register
- A POP instruction copies the data at SS:SP to *destination*, then adds 2/4 to SP

POP Illustration

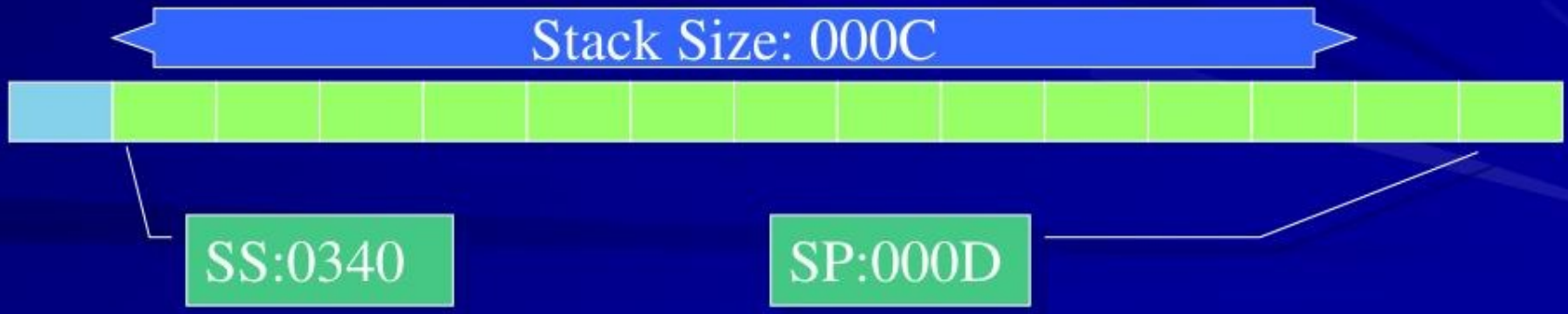
POP ES

Stack Over/Underflow

- The processor does not check for either illegal condition
  - Programs may include code to check for stack errors
  - Overflow occurs when SP is smaller than the address of the start of the stack array
    - Usually this means SP is decremented past 0!
  - Underflow occurs if SP gets bigger than its starting value

Out Of Bounds!

- Stack Overflow
- Stack Underflow



Procedures

```
proc_name     PROC      type
   ;procedure body
   RET     ;to return to caller
proc_name     ENDP
```

- *type* is NEAR or FAR
  - the default is NEAR (small model)
- Procedures may have one or more RET's

Procedure Calls and Returns

- Invoke a procedure (NEAR)

  CALL *proc_name*
  - push IP onto stack
  - copy address of *proc_name* into IP
- Return from a procedure (NEAR)

  RET *[n]*
  - pop top of stack into IP
  - add *n* to SP (this is optional)

Far Procedures

- Invoke a procedure (FAR)

  CALL *proc_name*
  - push CS, then IP onto stack
  - copy far address of *proc_name* into CS:IP
- Return from a procedure (FAR)

  RET *[n]*
  - pop top of stack into IP then pop into CS
  - add *n* to SP (this is optional)

Inter-Procedure Communication

- Shared storage
  - The data segment is accessible to all procedures in the current program
- Registers
  - Load registers with arguments (or argument addresses)
  - Store return values in registers
- Place argument information on the stack

Interrupts

- Interrupts are special procedure calls
  - These are always FAR calls since the interrupt routines are probably not accessible from your code segment
  - The Flags register must be preserved
- INT *interrupt_type*
  - Flags register is pushed, then TF and IF are cleared
  - CS is pushed, then IP is pushed
  - CS:IP set to address of interrupt vector implied by *interrupt_type*

Returning From an Interrupt

- IRET
- This instruction causes the following actions
  - Top of stack popped into IP
  - Top of stack popped into CS
  - Top of stack popped into Flags register
- This allows interrupted program to resume as if nothing had happened

Homework

- Page 158+
  - #1-7