ECE291

Lecture 5

Let's get stacked

Lecture outline

- Program stack
- Pushing and popping
- Program organization
- Debugging hints
- Assignments

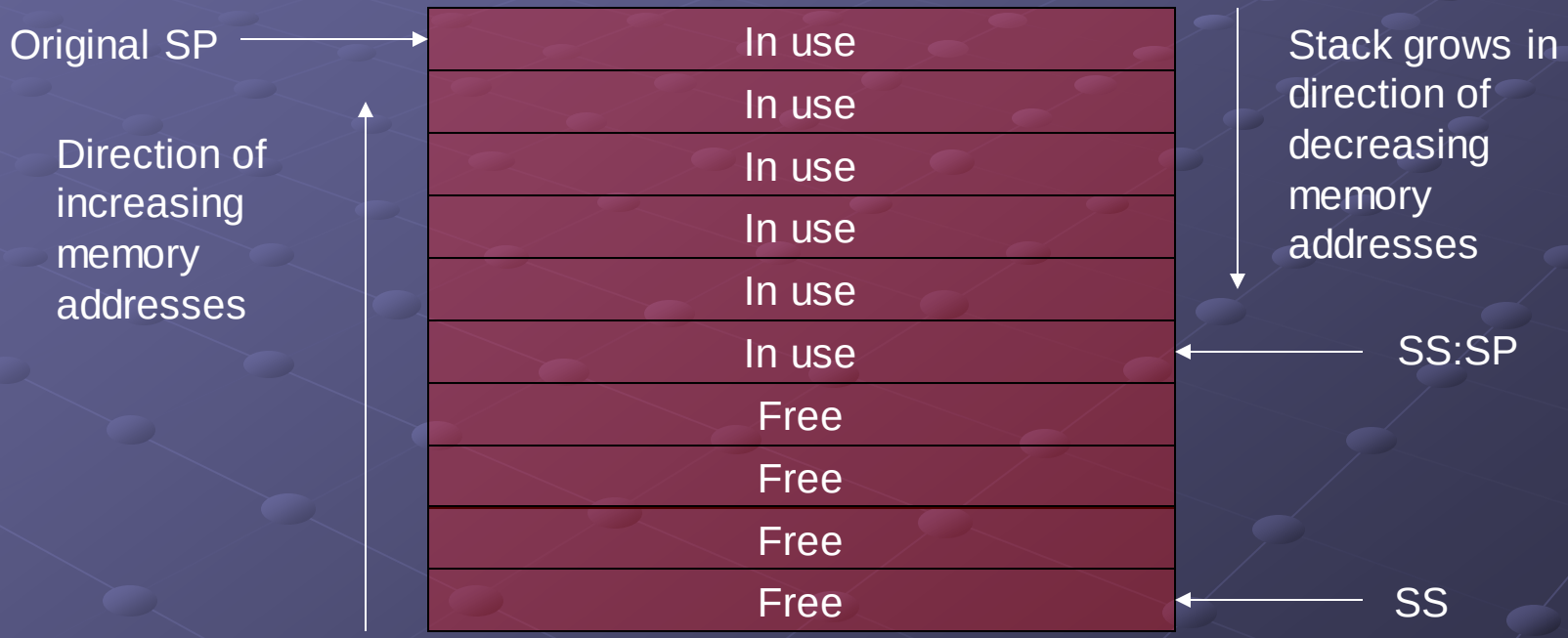
Program stack

Key characteristics

- Stores temporary data during program execution
- One point of access – the top of the stack
- Last-in-first-out (LIFO) storage
  - Data is retrieved in the reverse order from which it was stored
- Instructions that directly manipulate the stack
  - PUSH – places data on top of stack
  - POP – removes data from top of stack

Program stack

Implementation in memory

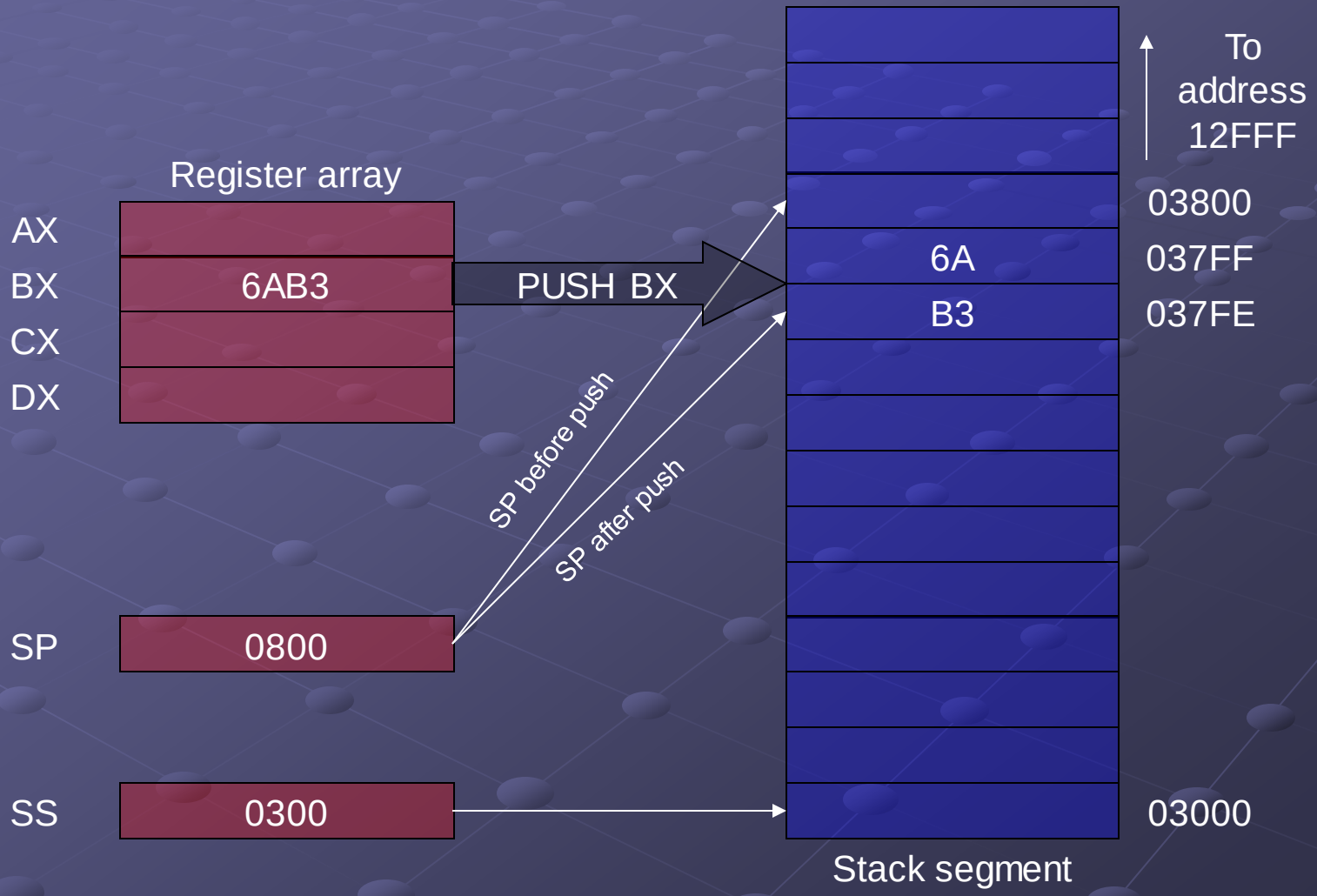


Program stack

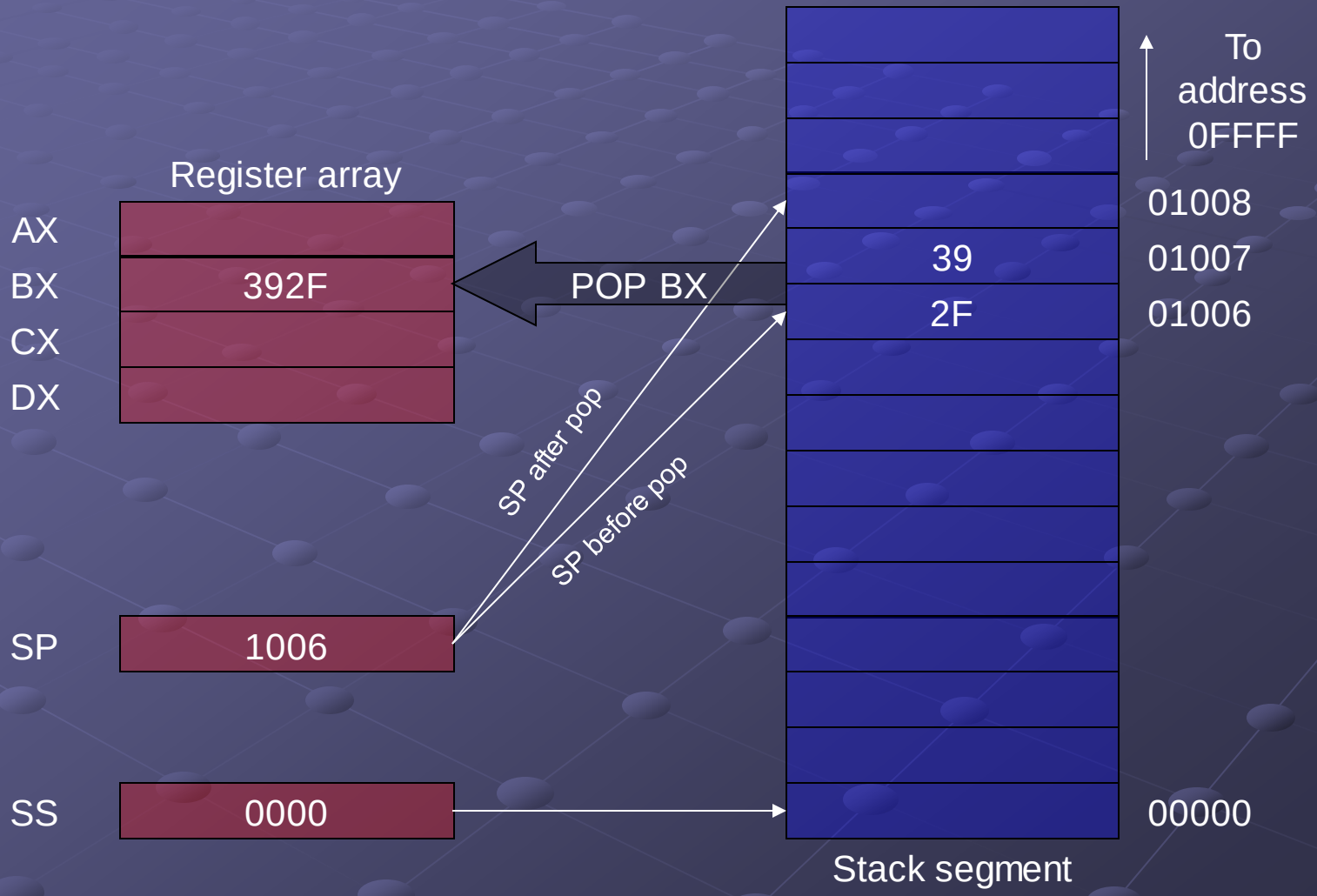
Implementation in memory

- SS – Stack segment
- SP – Stack pointer always points to the top of the stack
  - SP initially points to top of stack (high memory address)
  - SP decreases as data is PUSHed
    - PUSH AX → SUB SP, 2; MOV[SS:SP], AX
  - SP increases as data is POPed
    - POP AX → MOV AX, [SS:SP]; ADD SP, 2
- BP – Base pointer can point to any element on the stack

Push example



Pop example


Register array
AX
BX
CX
DX
392F
POP BX
39
2F
To address 0FFFF
01008
01007
01006
SP after pop
SP before pop
SP
1006
SS
0000
00000
Stack segment

Push and pop

- PUSH and POP always store or retrieve words of data (never bytes)
- 80386 and above allow words or double words to be transferred to and from the stack
- Source of data for PUSH
  - Any internal 16-bit register, immediate data, any segment register, or any **two bytes of memory**
- Pop places data into
  - Internal register, segment register (except CS), or a **memory location**

Pusha and popa

- 80286 and later include PUSHA and POPA to store and retrieve the contents internal register set (AX, BX, CX, DX, SP, BP, SI, DI)

Stack initialization example

- Assume that the stack segment resides in memory locations 10000h – 1FFFFh
- The stack segment is loaded with 1000h
- The SP is loaded with 0000h
  - This makes the stack 64KB
  - First PUSH does 0000h – 0002h = FFFEh, storing data in 1FFFEh and 1FFFFh

Stack use

- To store
  - Registers
  - Return address information while procedures are executing
  - Local variables that procedures may require
  - Dynamically allocated memory
- To pass
  - Parameters to procedures (function arguments)

Temporary register storage

PUSH and POP registers to preserve their values

```
PUSH AX             ; Place AX on the stack
PUSH BX             ; Place BX on the stack

…

< modify contents of AX and BX >

…

POP BX              ; Restore original value of BX
POP AX              ; Restore original value of AX
```

Temporary register storage

- Why would you want to backup and restore registers?
- Because registers are themselves temporary storage for instruction operands or memory addresses
- You might need to do a task that modifies registers, but you might then need the original contents of those registers later
- Any data that is an end result more than likely should go into a memory location (a variable)

The stack and procedures

- `call proc_name`
  - Pushes the instruction pointer and sometimes the code segment register onto the stack
  - Performs an unconditional jump to the label `proc_name`
- `ret`
  - Pops saved IP and if necessary saved CS from the stack and back into the IP and CS registers
  - This causes the instruction following the call statement to be executed next
- More on all of this procedure stuff tomorrow

Program organization

- Create block structure and/or pseudocode on paper to get a clear concept of program control flow and data structures
- Break the total program into logical procedures/macros
- Use jumps, loops, etc. where appropriate
- Use descriptive names for variables
  - Noun_type for types
  - Nouns for variables
  - Verbs for procedures/macros

Program organization

- Good program organization helps with debugging
- Programs do not work the first time
- Strategy to find problems
  - Use TD and set breakpoints to check program progress
  - Use comments to temporarily remove sections of code
  - Use “print” statements to announce milestones in the progrm
- Test values and test cases
  - Try forcing registers or variables to test output of a procedure
  - Use “print” statements to display critical data
- Double-check your logic
- Try a different algorithm if all else fails

NASM directives

- Includes — %include “drive:\path\filename”
- Definitions
  - DB/RESB — define/reserve byte (8 bits)
  - DW/RESW — define/reserve word (16 bits)
  - DD/RESD — define/reserve doubleword (32 bits)
  - EQU — names a constant (has no effect on memory)
- Labels
  - “.” prefixed — “local” to previous non-dotted label
  - “:” suffix — not required and doesn’t change label
  - Global.local — can access dotted local labels anywhere by prepending previous non-dotted label

NASM directives

- Procedures
  - Labeled sections of code you can jump to and return from any point in your program
  - Procedures begin with merely a non-dotted label
  - Use dotted labels inside procedures to keep them local to the procedure
  - Helps keep labels unique program-wide

NASM directives

- Macros
  - Procedures require some overhead in memory and execution time
  - NASM replaces macro call with the macro code itself
  - Advantages
    - Faster (no call instruction)
    - Readability – easier to understand program function
  - Drawbacks
    - Using the macro multiple times duplicates code
    - Tricky to debug sometimes (especially when you have nested macros)

NASM directives

- References to procedures
  - EXTERN *name* – gives you access to procedures and variables in other files (such as library files)
  - GLOBAL *name* – makes your procedures and variables available to other files (as if you were creating a library)
- Segment definition
  - SEGMENT *name*
  - Examples: SEGMENT STACK
    SEGMENT CODE

Example program structure

```
; ECE291:MPXXX
; In this MP you will develop a program which take input
; from the keyboard

;====== Constants ==========================================
;ASCII values for common characters
CR      EQU 13          ; EQU's have no effect on memory
LF      EQU 10          ; They are preprocessor directives only
ESCKEY  EQU 27          ; LF gets replace with 10 when assembled

;====== Externals ==========================================
; -- LIB291 Routines
extern dspmsg, dspout, kbdin
extern rsave, rrest, binasc
```

Example program structure

```
;==== LIBMPXXX Routines (Your code will replace calls to these
   ;functions)
extern LibKbdHandler
extern LibMouseHandler
extern LibDisplayResult
extern MPXXXXIT

;====== Stack ================================================
stkseg  segment STACK          ; *** STACK SEGMENT ***
        resb 64*8              ; 64*8 = 512 Bytes of Stack
stacktop:

;====== Begin Code/Data ======================================
codeseg  segment CODE          ; *** CODE SEGMENT ***
```

Example program structure

```
;====== Variables ========================================
inputValid      db  0           ; 0: InputBuffer is not ready
                                ; 1: InputBuffer is ready
                                ;-1: Esc key pressed

operandsStr     db  'Operands: ','$'
OutputBuffer    16 times db 0 ; Contains formatted output
                db '$'        ; (Should be terminated with '$')
MAXBUFLENGTH    EQU 24
InputBuffer     MAXBUFLENGTH times db 0
                              ; Contains one line of user input
                db '$'
graphData       %include "graphData.dat"      ; data

GLOBAL outputBuffer, inputValid, operandsStr
GLOBAL graphData
```

Example program structure

```
;====== Procedures ========================================
   KbdHandler
        <Your code here>
   MouseHandler
        <Your code here>
   DisplayResult
        <Your code here>
;====== Program Initialization ============================
..start:
        mov ax, cs              ; Use common code & data segment
        mov ds, ax
        mov sp, stacktop        ; Initialize top of stack
```

Example program structure

```
;====== Main Procedure ========================================
MAIN:
        MOV     AX, 0B800h ;Use extra segment to access video
        MOV     ES, AX
                <here comes your main procedure>

        CALL    MPXXXXIT                    ; Exit to DOS
```