AUBURN

UNIVERSITY

# Designing Robust Procedures

# Module Overview

## Objectives

- Learn and understand the runtime stack
- Understand, define and use procedures
- Learn how to design robust procedures.
- Introduce the mechanism of interrupt
- Examine the machine code of an assembly program

## Requirements

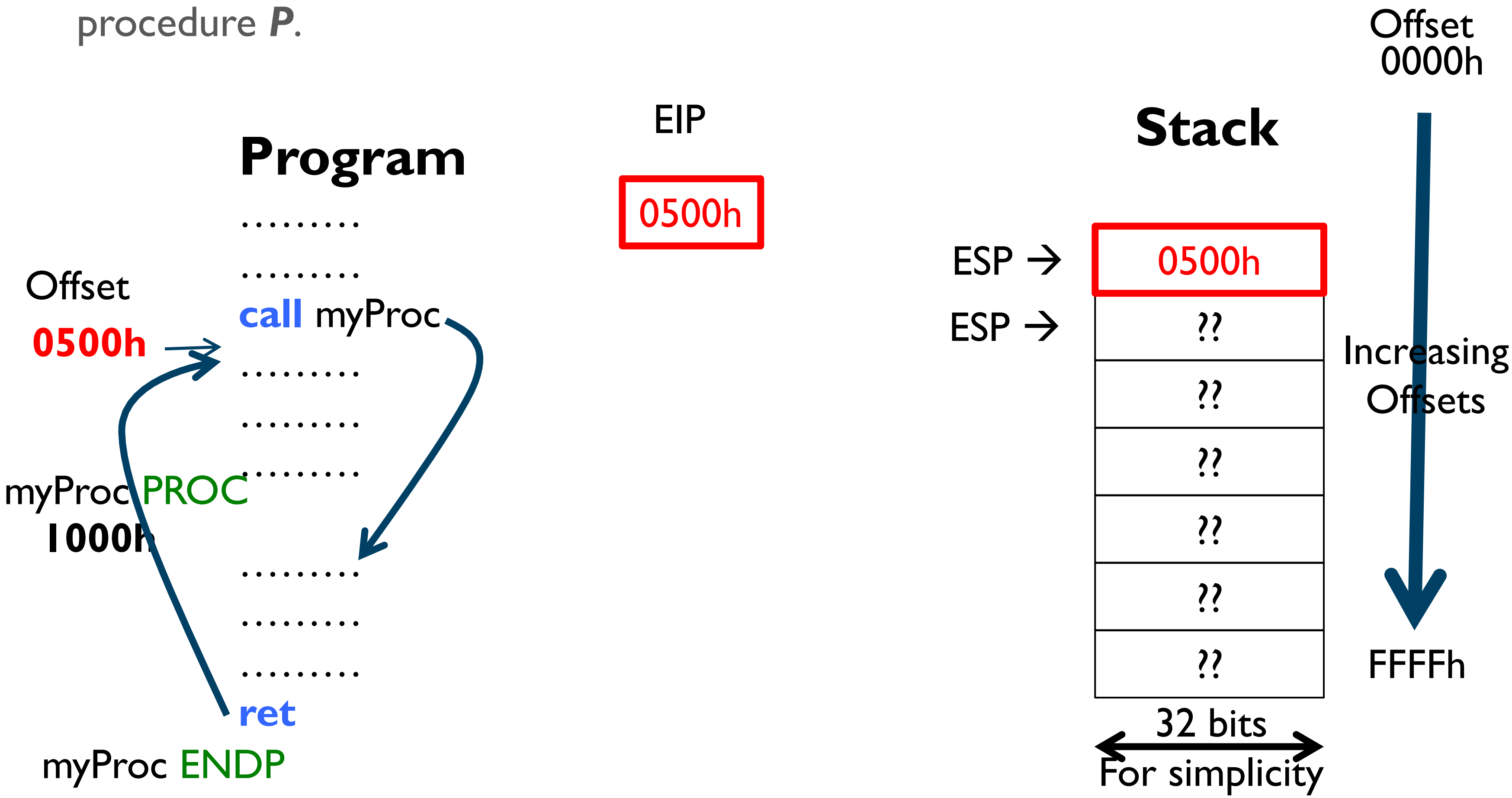
- Know the registers EIP and ESP
- Know the addressing modes (Indirect)
- Know the instruction set (CALL and RET)

## The Runtime Stack

- Why is the stack important for procedures?
- Implicit Use of Stack:
  - Call and ret instructions
- Explicit Use of Stack:
  - PUSH and POP Instructions

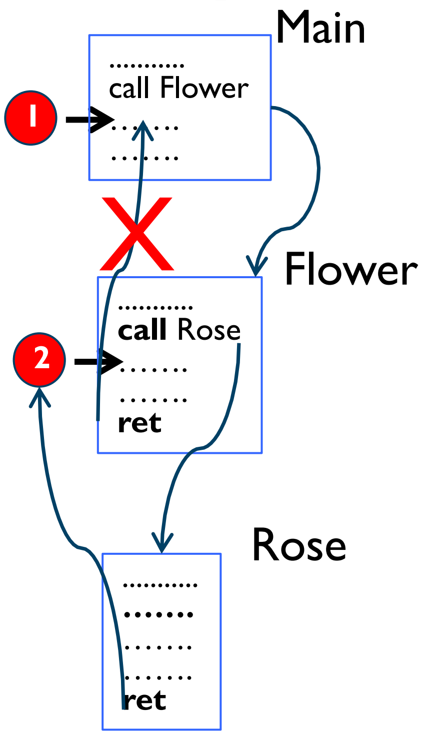
# Why is the stack important for procedures?

- **Short answer**: when calling a procedure ***P***, the CPU saves on the stack the register EIP. The CPU will use that saved value to return after the execution of the procedure ***P***.



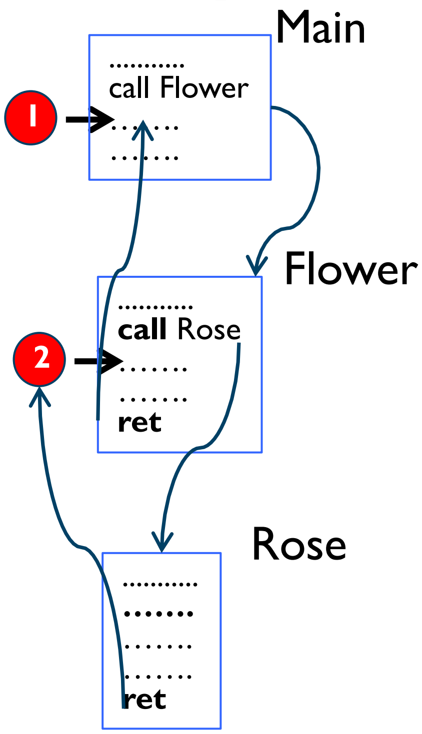
# But, why use the stack instead of a register…? (1/2)

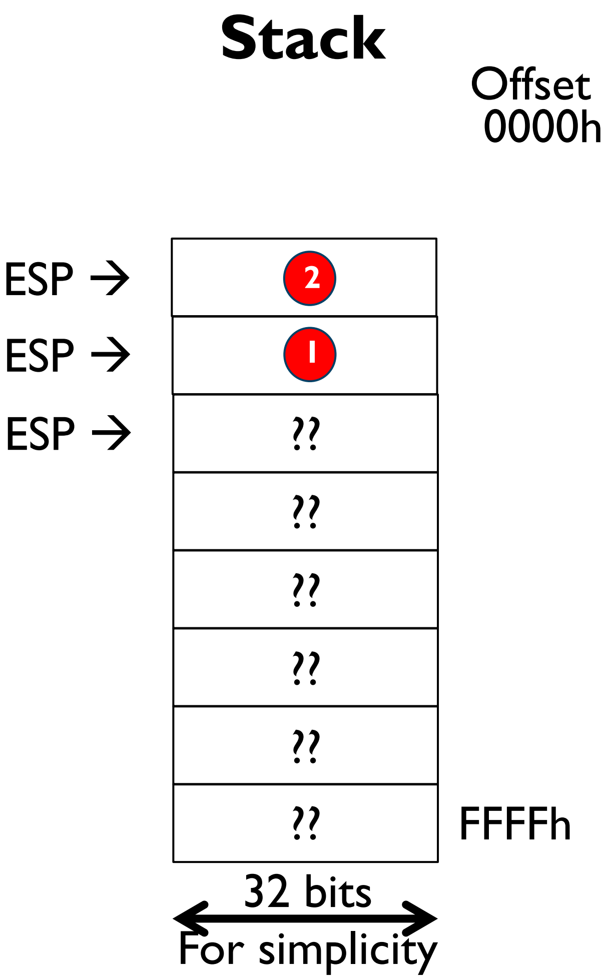
- Answer: a procedure can call itself another procedure which calls itself another one and so on ...




# But, why use the stack instead of a register…? (2/2)

- Answer: **nested** procedure calls. A procedure can call itself another procedure which calls itself another one and so on ...



# The Stack Can Be Used by the Programmer Too

- Stack is critical for the programmer, especially when using procedures:
- Two instructions can be used:
  - **PUSH** (save onto the stack)
  - **POP** (pull from the stack)

# PUSH Instruction: PUSH

**PUSH** *operand*

- **Mnemonic**: PUSH
- **Operand**:
  - *Operand* can be a 1) register or a memory of size 16 bits or 32 bits, or 2) a 32-bit immediate
- **Function**: (two actions)
  1. ESP ← ESP – **A** **(Decrement ESP)**
  2. [ESP] ← Operand

| Operand (Size) | A |
|---|---|
| 16 bits | 2 |
| 32 bits | 4 |

# POP Instruction: POP

**POP** *operand*

- **Mnemonic**: POP
- **Operand**:
  - *Operand* can be a register or a memory of size 16 bits or 32 bits.
- **Function**: (two actions)
  1. Operand ← [ESP]
  2. ESP ← ESP + **A** **(Increment A)**

| Operand (Size) | A |
|---|---|
| 16 bits | 2 |
| 32 bits | 4 |

## Defining and Using Procedures

- **PROC** and **ENDP** Directives
- **CALL** and **RET** Instructions

# PROC and ENDP Directives

- Recall, directives are hints to the assembler, NOT CPU instructions
- A procedure must be defined/declared between **PROC** and **ENDP** directives:
- **Examples**: main procedure, myProc procedure (main and myProc are the respective names of the two procedures).

```
Main PROC
…..........
.............
.............
Main ENDP
```

```
myProc PROC
…..........
.............
    ret
myProc ENDP
```

- **Important**: good design recommends that a procedure be **closed** with only one exit door: 1) a **call** to another procedure or 2) the **ret** instruction to return from the its call..
- **Consequence (of the closeness):** only labels defined inside a procedure are visible only within a procedure. This "limitation" is in fact a good protection against ill designed code that leaves procedures through jumps, rather than call or ret instructions.

# (Unconditional) Procedure Call : CALL (Second Visit)

- **Syntax** : **CALL** *Destination*
  - **Mnemonic**: CALL
  - **Operand**: *Destination* is a label
  - **Function**:
    1. EIP value ***pushed*** onto the stack (recall EIP contains the offset of the next instrution
       1. ESP ← ESP – 4
       2. [ESP] ← EIP (Recall that EIP contains the offset of the instruction just after the call instruction)
    2. EIP ← Offset indicated by *Destination*
    - **In words**
    1. Saves the value of EIP onto the stack (to mark where to return after the execution of the procedure)
    2. stores in EIP the offset indicated by *Destination*. Therefore, the next instruction to be executed is the instruction "marked" by the label Destination

# (Unconditional) Procedure Return: RET (Second Visit)

- **Syntax** : **RET**
  - **Mnemonic**: RET
  - **Operand**: ***NO*** *Operand*
  - **Function**: EIP ← Top of Stack
    1. EIP value is ***popped*** out of the stack
        1. EIP ← [ESP]
        2. ESP ← ESP + 4

    - **In words**

    Pops value from stack and stores it in EIP. Therefore, the next instruction to be executed is the instruction "marked" by the value that was previously *pushed* (saved) onto the stack (during CALL).

  - **Example:**

    See next slide for the use of CALL and RET.

# (Example) Procedure

main PROC ; PROC is a directive indicating the beginning of the *main* procedure

...........

...........

**call** MyRoutine ; 1) EIP value ***pushed*** onto the stack, 2) EIP ← Offset indicated by MyRoutine

mov eax,edx
add eacx,eax

............

............

MyRoutine PROC; PROC indicating the beginning of *MyRoutine* procedure
mov edx, 1

.............

**ret** ; EIP value is ***popped*** out of the stack

MyRoutine ENDP; ENDP is a directive indicating the end of *MyRoutine* procedure

........

........

main ENDP ; ENDP is a directive indicating the end of *main* procedure

# Design Robust Procedures

- **Step 1: Characterize** the role of the registers in a procedure:
  - *parameter* register
  - *intermediary* register
  - *result* register
- **Step 2: Comment** Procedures
- **Step 3: Save/Restore** (parameter and intermediary) registers

# Step 1: Characterize the Role of Procedure Registers

- Good design starts by **characterizing** the role of all procedure registers as parameter, intermediary, or result.
- A **parameter** register is a register that contains a parameter that the procedure uses to carry its operations. It is an ***input*** for the procedure.
- An **intermediary** register is a register used by the procedure to compute intermediary results inside the procedure. These intermediary results are not directly useful or needed after the procedure returns.
- A **result** register is a register that contains a result that will be returned to the caller of the procedure.

  - See example on the next slides

# Example: Compare Two Arrays (1/2)

**Procedure Description:**

1) Compare two 8-bytes arrays stored starting at offsets contained in ESI and EDI

2) If they are the same, return 0 in Register DL, otherwise return 1 in DL.

```
stringCompare PROC
              mov ecx, 8 ; 8 elements to compare at most
              mov dl,0
myLoop :      mov al, [esi] ; al ← current element of mySource
              cmp[edi], al ; compare[esi] and al
              jz keepGoing ; if ([esi] == 2Ch)jump to KeepGoing
              inc dl ; found a difference then DL ←DL + 1
              jmp Done   ; Done! no need to keep comparing
keepGoing:    inc esi ; update esi to refer the next
              inc edi ; update esi to refer the next
              loop myLoop
Done :        ret
stringCompare ENDP
```

**Register List**: ecx, dl, al, esi, edi

**Objective: Identify the parameter, intermediary, and result registers.**

# Example: Characterize the Registers

**Procedure Description:**

1) Compare two 8-bytes arrays stored at offsets contained in ESI and EDI
2) If they are the same, return 0 in Register DL, otherwise return 1 in DL.

- **Parameter** Registers: this procedure needs the registers ESI and EDI that specify the offset of the arrays to compare. These two registers contain all the information needed to carry on the comparison. So, **ESI** and **EDI** are parameter registers.

- **Result** Register: based on the description of the procedure, the result of the comparison is noted in the register **DL** (0 if equal and 1 otherwise). DL is a result register

- **Intermediary** Registers: these are all registers used by the procedure which are neither parameters nor results. Based on the code of this procedure, registers **ecx** and **al** are intermediary registers.

# Step 2: Comment the Procedures

- Comment separately each category of register at the beginning of the procedure.

```
; Result Registers
; --- DL: 0 if arrays equal, 1 otherwise
; Parameter Registers
; --- ESI: offset of the first array
; --- EDI: offset of the second array
; Intermediary Registers
; --- ecx : loop counter contains initially the size of of the arrays
; --- al : contains current array element to compare
 stringCompare PROC
               mov ecx, 8 ; 8 elements to compare at most
               mov dl,0
myLoop :     mov al, [esi] ; al ← current element of mySource
               cmp[edi], al ; compare[esi] and al
               jz keepGoing ; if ([esi] == 2Ch)jump to KeepGoing
               inc dl ; found a difference then DL ←DL + 1
               jmp Done   ; Done! no need to keep comparing
keepGoing:   inc esi ; update esi to refer the next
               inc edi ; update esi to refer the next
               loop myLoop
Done :         ret
stringCompare ENDP
```

# Step 3: Save/Restore parameter and intermediary registers

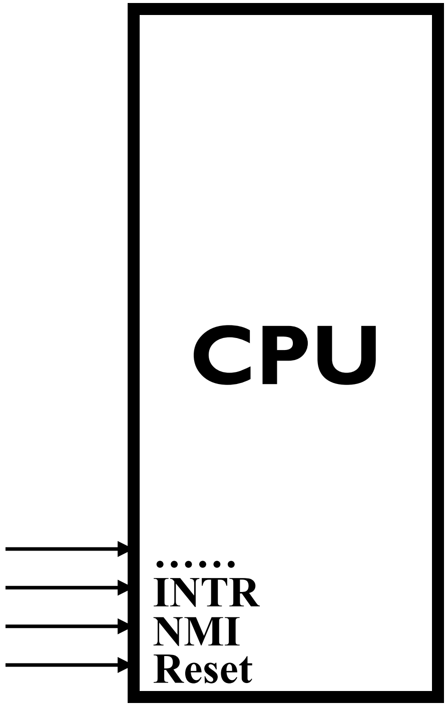
```
; Result Registers
; --- DL: 0 if arrays equal, 1 otherwise
; Parameter Registers
; --- ESI: offset of the first array
; --- EDI: offset of the second array
; Intermediary Registers
; --- ecx : loop counter contains initially the size of of the arrays
; --- al : contains current array element to compare
 stringCompare PROC
               PUSH esi ; save parameter and intermediary registers
               PUSH edi
               PUSH ecx
               PUSH ax
               mov ecx, 8 ; 8 elements to compare at most
               mov dl,0
myLoop :      mov al, [esi] ; al ← current element of mySource
               cmp[edi], al ; compare[esi] and al
               jz keepGoing ; if ([esi] == 2Ch)jump to KeepGoing
               inc dl ; found a difference then DL ←DL + 1
               jmp Done   ; Done! no need to keep comparing
keepGoing:    inc esi ; update esi to refer the next
               inc edi ; update esi to refer the next
               loop myLoop
Done :         POP ax ; save parameter and intermediary registers
               POP ecx
               POP edi
               POP esi
               ret
stringCompare ENDP
```

# Interrupts

- What is an interrupt?
- Why use interrupts?
- How does a CPU Handle Interrupts?
- How to use interrupts?

## **What** is an interrupt?

- ***Interrupt*** is a mechanism that allows some event **E** (external or internal) to :
    - 1) Stop the current thread of execution
    - 2) trigger immediately the execution of some routine (procedure) **R** to handle Event E.
    - 3) Resume, after completion of R, the interrupted thread of execution.




*R is called an **interrupt routine handler (irh)** or **interrupt service routine (isr)***

# **Why** do we use interrupts?

- Most events require immediate "attention":
  - **input** devices have limited memory (buffer) to store events (**keystrokes** for a keyboard, **frames** for a network card, **clicks** on a mouse, **hits** on a touch screen….)
- Polling is inefficient and costly
  - Waste of computing time/energy
- Programming with interrupts is more convenient

# How Does a CPU Handle an Interrupt?

1) Event occurs (rising or falling edge or pulse on a pin of CPU)
2) CPU **completes execution** of current instruction
3) CPU **saves program counter (EIP), state register (EFLAGS),** and possibly other general purpose registers on stack
4) CPU **determines the address** $\mathbf{A}_{irh}$ of the interrupt handler routine (**irh**).
5) CPU **writes** $\mathbf{A}_{irh}$ **in EIP**.
6) Interrupt handler routine is executed
7) The CPU **restores** state register (EFLAGS), EIP … from the stack. Therefore, the CPU returns to the *normal* execution.

# How to Use Interrupts?

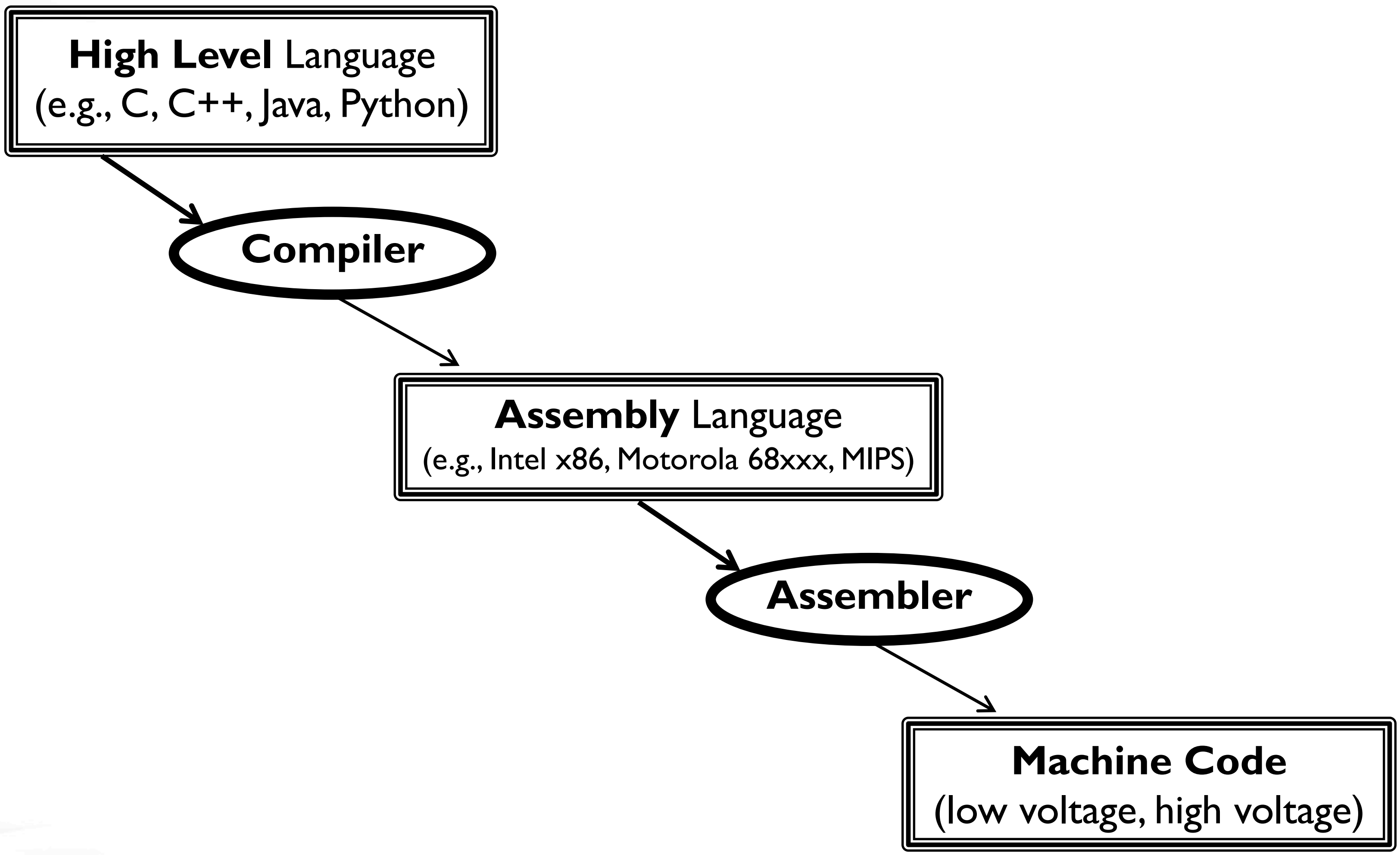
- The programmer must:
  1. Write an interrupt routine handler (i.e., the procedure that must be executed when the event happens)
  2. Load the procedure at some address $A_{irh}$
  3. Initialize the system to associate a specific event with  the address $A_{irh}$: this way, the CPU will know the routine to call when the event occurs.

- **NOTE**: on x86 processors (CPU), you must leave the interrupt routine handler with the **IRET** instruction instead of RET instruction.

# Machine Code

- From High-Level Program to Machine Code
- Example:
  1. An assembly program main.asm
  2. Its machine code: main.obj

# From High-Level Language to Machine Code



# Machine Code ←Assembly Program

Command Prompt - more Project.lst

```
                                .386
                                .model flat,stdcall
main.asm(5) : warning A4011:multiple .MODEL directives found : .MODEL ignored
                                .stack 4096
                                ExitProcess proto,dwExitCode:dword

                                .data
                                abyte BYTE 12h
                                aWord WORD 3456h
                                aDouble DWORD 789ABCDh
                                myString BYTE "Hello World",0dh,0ah



                                .code
                                main proc
                                        mov al, aByte
                                        add al, al
                                        mov ax, aWord

                                        add ax,ax
                                        mov eax, aDouble
                                        add eax, eax
                                        mov edx,OFFSET myString
                                        call WriteString
                                        ; Start writing your code for your assignment

                                        invoke ExitProcess,0
                                        push   +000000000h
-- More (5%) --
```

# Module Wrap Up

- Learn and understand the runtime stack
  - Relationship procedures ←→ Stack
  - CALL and RET use implicitly the stack and ESP
  - PUSH and POP instructions
  - Nested procedures
- Understand, define and use procedures
  - Definition directives : PROC and ENDP
  - CALL and RET
- Learn how to design robust procedures.
  - Characterize registers as parameter, intermediary, or result
  - Comment properly procedures: spell out all used registers with their meanings
  - Save parameter and intermediary registers at the beginning of the procedure
  - Restore parameter and intermediary registers just before the ret instruction
- Introduce the mechanism of interrupt
  - What is an interrupt?
  - How does the CPU handle an interrupt?
  - How to use interrupts?
- Examine the machine code of an assembly program
  - Relationship high-level language, assembly language, and machine code
  - Machine code is ultimately just 0s and 1s (low voltage, high voltage)