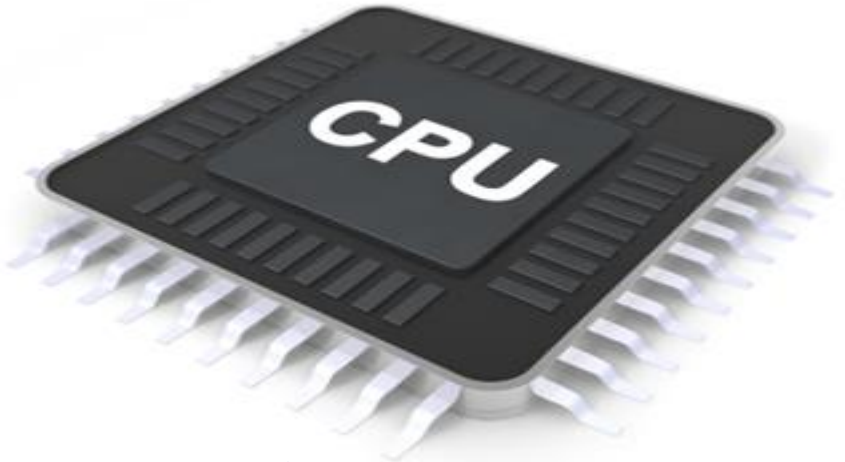


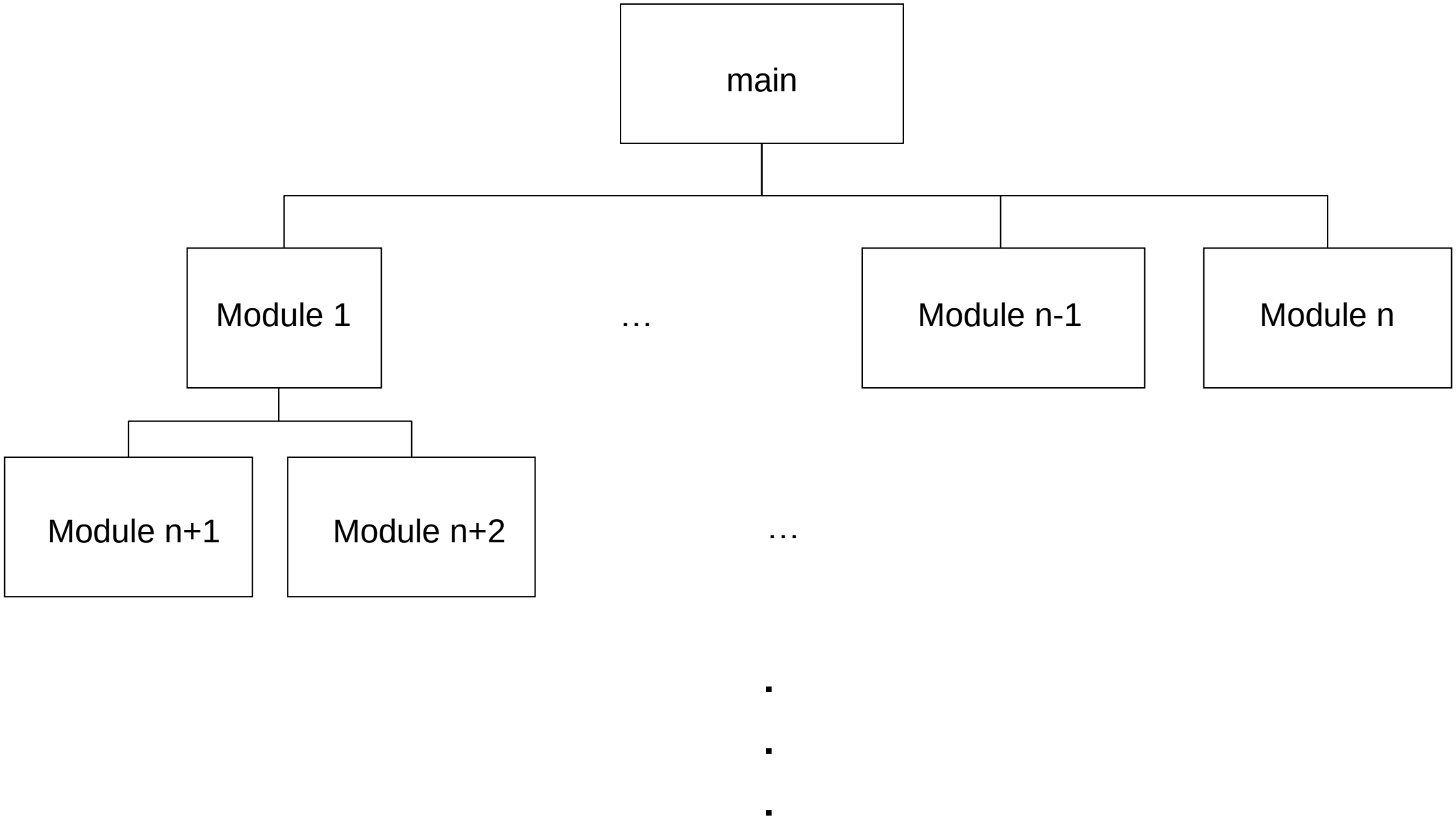
Subroutines

ENEE 3582

Microp

Modular Programming

- Top-Down Design
  - aka functional decomposition
  - design your program before starting to code
  - break large tasks into smaller ones
  - use a hierarchical structure based on subroutine calls
- MAIN module: used to organize the entire program
- Sub-modules can be called by MAIN and other modules


main
Module 1
…
Module n-1
Module n
Module n+1
Module n+2
…

Advantages of Modular Programming

- Faster development
  - Task organized code
  - Different individuals can be code in parallel
  - Testing/debugging of modules done individually
- Reusable modules
- Disadvantage of modular approach: Slower program execution

Subroutine

- aka Function, Procedure
- User-defined program module
  - Not part of the “main” program
  - Each subroutine has a user specified name
- CALL is used to jump to subroutines
  - Unconditional jump
  - Stores PC in the stack
  - Modifies PC to go to the beginning of the subroutine
- RET is used to return from subroutine
  - Retrieves the stored PC from stack
  - Modifies PC to go to the line below the CALL

CALL

- Jumps to subroutine in PM
  - PUSHes PC+1 (return address)
  - Modifies PC = address of subroutine
- `CALL`: Call address
  - Format: `CALL k ;PC = k. PUSH PC+1. SP=SP-3`
  - k is 22 bits: 0 to <4M
  - Best used when program memory > 8KB
- `ICALL`: Indirect call using Z
  - Format: `ICALL ;PC=mem[Z]. PUSH PC+1. SP=SP-3`
- `RCALL`: Relative call
  - Format: `RCALL k ;PC = PC+1+k. PUSH PC+1. SP=SP-3`
  - k is 11 bits: -2K to <2K
  - Best used when PM < 8KB

RET

- Returns from subroutine to calling module
  - POP return address from stack
  - Modifies PC = return address
- RET
  - Format: `RET ;PC = mem[SP]; SP=SP+3.`

Code Template with Subroutines

```
            .DSEG
data_var:   .BYTE  size
            .CSEG
PM_const:   .type  value(s)

main:                                   ;main program
            RCALL sub_name      ;use CALL if program is >8KB

main_end:   RJMP main_end       ;don't leave

sub_name:
            RET                 ;return to calling module
```

Microchip Studio

- When debugging use “Step Into” ↓ (F11) to debug procedure
- Using “Step Over” F10 will execute the procedure in 1 click.

CALL Example

PC	Instruction		Notes
0x00000	Main:	…	
0x00010		RCALL sub1	PUSH 0x00011; PC = 0x00030
0x00020	EndMain:		
…	…		
0x00030	sub1:	…	
0x00040		RCALL sub2	PUSH 0x00041; PC = 0x00050
0x00041		RET	
…	…		
0x00050	sub2:	…	
0x00060		RCALL sub3	PUSH 0x00061; PC = 0x00070
0x00061		RET	
…	…		
0x00070	sub3:	…	
0x00080		RET	

RET Example

PC	Instruction		Notes
0x00000	Main:	…	
0x00010		RCALL sub1	
0x00020	EndMain:		
…	…		
0x00030	sub1:	…	
0x00040		RCALL sub2	
0x00041		RET	PC = POP 3 BYTES
…	…		
0x00050	sub2:	…	
0x00060		RCALL sub3	
0x00061		RET	PC = POP 3 BYTES
…	…		
0x00070	sub3:	…	
0x00080		RET	PC = POP 3 BYTES

Coding Exercise

- Write a subroutine that adds 2 words in R2:R1 and R4:R3 and returns their word sum in R6:R5.
- Write a subroutine that adds 2 words. R2:R1=address of 1st word and R4:R3 = address of 2nd word. Returns their word sum in R6:R5.
- Write a subroutine that adds 2 words. R2:R1=address of 1st word and R4:R3 = address of 2nd word, R6:R5 = address of sum.
- Write a subroutine that adds the elements of a byte array. The PM address of the array is in Z, length of the array in r16. The byte sum is returned in R0.

Subroutine Arguments

- Before the CALL
  - All argument needed to be passed to/from a subroutine must be prepared.
- Inside the subroutine:
  - Any registers used inside a subroutine should be saved using PUSH
  - Don’t push arguments
- Before RET
  - Use POP to restore registers used inside subroutine
  - Don’t pop arguments

Coding Example

- Write a subroutine that adds 2 words. R2:R1=data mem address of 1st word and R4:R3 = data mem address of 2nd word, R6:R5 = data mem address of sum.