CPE 323
Intro to Embedded Computer Systems
Assembly Language Programming (Subroutines)

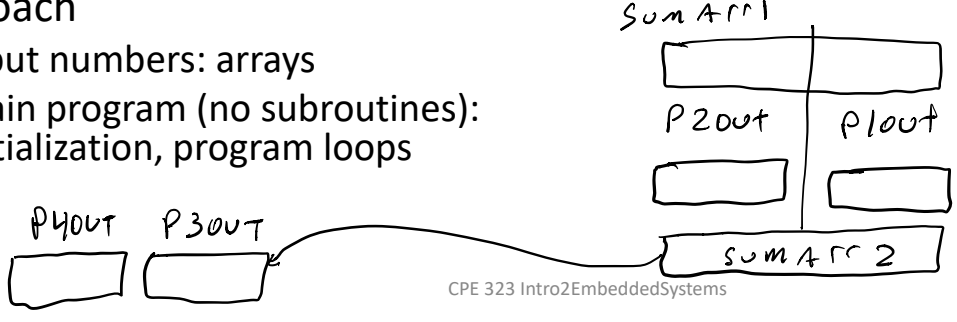
Aleksandar Milenkovic

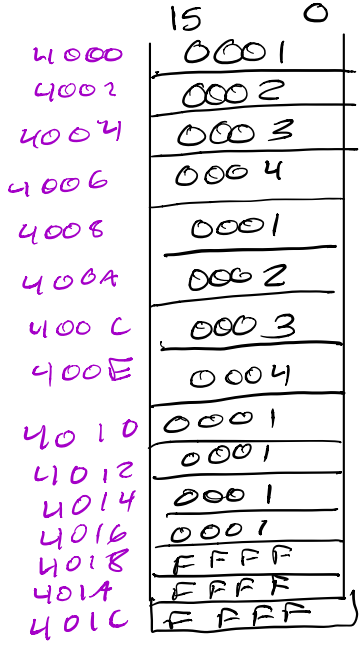
milenka@uah.edu

Admin

The Case for Subroutines: An Example

- Problem
  - Sum up elements of two integer arrays
  - Display results on P2OUT&P1OUT and P4OUT&P3OUT
- Example
  - arr1 .int 1, 2, 3, 4, 1, 2, 3, 4 ; the first array
  - arr2 .int 1, 1, 1, 1, -1, -1, -1 ; the second array
  - Results
    - P2OUT&P1OUT=0x000A, P4OUT&P3OUT=0x0001
- Approach
  - Input numbers: arrays
  - Main program (no subroutines): initialization, program loops





Sum Up Two Integer Arrays (ver1)

```
;-------------------------------------------------------------------------------
; File       : Lab5_D1.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of two integer arrays
; Description: The program initializes ports,
;              sums up elements of two integer arrays and
;              display sums on parallel ports
; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
; Output     : P1OUT&P2OU displays sum of arr1, P3OUT&P4OUT displays sum of arr2
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
                                            ; and retain current section.
            .retainrefs                     ; And retain any sections that have
                                            ; references to current section.

;-------------------------------------------------------------------------------
RESET:      mov.w   #__STACK_END,SP         ; Initialize stack pointer
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
```

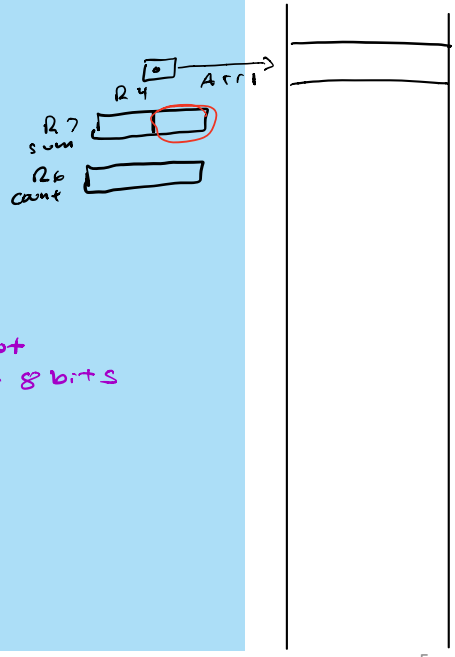
just a template...

Sum up two integer arrays (ver1)

```
;-------------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------------
main:   bis.b   #0xFF, P1DIR
        bis.b   #0xFF, P2DIR
        bis.b   #0xFF, P3DIR
        bis.b   #0xFF, P4DIR
        mov.w   #arr1, R4
        clr.w   R7
        mov.w   #8, R6
isum:   add.w   @R4+, R7;
        dec.w   R6
        jnz     isum
        mov.b   R7, P1OUT
        swpb    R7
        mov.b   R7, P2OUT
```

Configuring output ports.

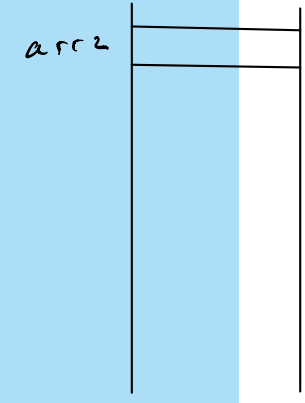
R7 ← R7 + m[R4]
R6 ← R6 - 1
jump not zero.
Move lower 8 bits to P1OUT
Lower 8 bits become upper 8 bits
Move upper 8 bits to P2OUT



Sum up two integer arrays (ver1)

```
        mov.w   #arr2, R4
        clr     R7
        mov.w   #7   R6
lsum2   add.w   @R4+, R7
        dec.w   R6
        jn2     lsum2
        mov.b   R7, P3out
        swpb    R7
        mov.b   R7, P4out
```

R7 ← R7 + m[R4] ; R4 ← R4 + 2



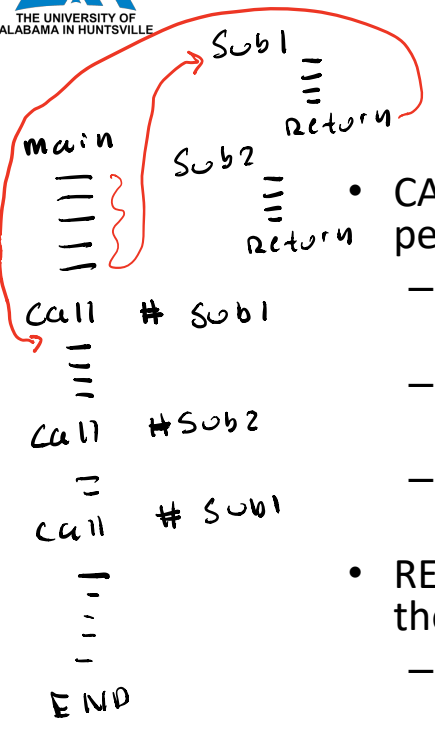

```
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
            .end
```

Subroutines

- A particular sub-task is performed many times on different data values
- Frequently used subtasks are known as subroutines
- Subroutines: How do they work?
  - Only one copy of the instructions that constitute the subroutine is placed in memory
  - Any program that requires the use of the subroutine simply branches to its starting location in memory
  - Upon completion of the task in the subroutine, the execution continues at the next instruction in the calling program

RA -> return address.

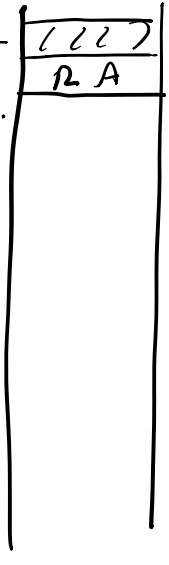
Subroutines (cont'd)



- CALL instruction: pushes RA to TOS
  perform the branch to subroutines
  - SP <= SP – 2 ; allocate a word on the stack for return address — Determines Start address
  - M[SP] <= PC ; push the return address (current PC) onto the stack
  - PC <= TargetAddress ; the starting address of the subroutine is moved into PC
- RET instruction:
  the last instruction in the subroutine
  - PC <= M[SP] ; pop the return address from the stack
  - SP <= SP + 2 ; release the stack space

1. SP ← SP-2
2. M[SP] ← PC
3. PC ← Target add.

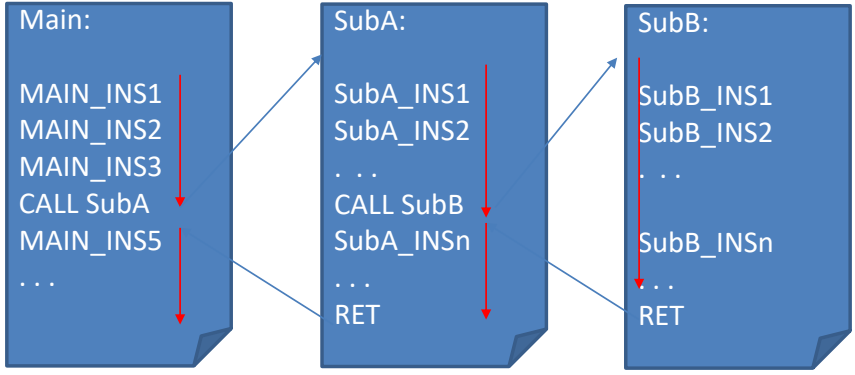
SP TOS



• RET ≡ mov.w @SP+, PC

1. PC ← M[SP]
2. SP ← SP+2

Subroutine Nesting



If you are using registers inside subroutines, do not destroy the subroutine.

Mechanisms for Passing Parameters

- Through registers
- Through stack
  - By value
    - Actual parameter is transferred
    - If the parameter is modified by the subroutine, the “new value” does not affect the “old value”
  - By reference
    - The address of the parameter is passed
    - There is only one copy of parameter
    - If parameter is modified, it is modified globally

Subroutine: SUMA_RP

- Subroutine for summing up elements of an integer array
- Passing parameters through registers
  - R12 - starting address of the array
  - R13 - array length
  - R14 - display id
    (0 for P2&P1, 1 for P4&P3)

R14 - keeps sum

Subroutine: SUMA_RP

```
;-------------------------------------------------------------------------------
; File       : Lab5_D2_RP.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of an input integer array
; Description: suma_rp is a subroutine that sums elements of an integer array
; Input      : The input parameters are:
;                   R12 -- array starting address input
;                   R13 -- the number of elements (>= 1) input
;                   R14 -- display ID (0 for P1&P2 and 1 for P3&P4) sum
; Output     : No output
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"      ; Include device header file

            .def suma_rp

            .text
```

Subroutine: SUMA_RP

```
suma_rp:
            push.w  R7              ; save the register R7 on the stack
            clr.w   R7              ; clear register R7 (keeps the sum)
lnext:      add.w   @R12+, R7       ; add a new element
            dec.w   R13             ; decrement step counter
            jnz     lnext           ; jump if not finished
            bit.w   #1, R14         ; test display ID
            jnz     lp34            ; jump on lp34 if display ID=1
            mov.b   R7, P1OUT       ; display lower 8-bits of the sum on P1OUT
            swpb    R7              ; swap bytes
            mov.b   R7, P2OUT       ; display upper 8-bits of the sum on P2OUT
            jmp     lend            ; skip to end
lp34:       mov.b   R7, P3OUT       ; display lower 8-bits of the sum on P3OUT
            swpb    R7              ; swap bytes
            mov.b   R7, P4OUT       ; display upper 8-bits of the sum on P4OUT
lend:       pop     R7              ; restore R7
            ret                     ; return from subroutine

            .end
```

```
suma_rp:
                clr.w   R14
Isum            add.w   @R12+,R14
                dec.w   R13
                jnz     Isum
                ret
```

R14 ← 0

R14 ← R14 + M[R12]

R12 ← R12 + 2

R12 - starting address

R13 - num of elements

R14 - sum

Main (ver2): Call suma_rp

```
;-------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------
main    bis.b   # 0xFF, P1DIR
        bis.b   # 0xFF, P2DIR
        bis.b   # 0xFF, P3DIR
        bis.b   # 0xFF, P4DIR
        mov.w   #arr1, R12
        mov.w   #8, R13
        CALL    #suma_rp
        mov.b   R14 P1OUT
        swpb    R14
        mov.b   R14, P2OUT
        mov.w   #arr2, R12
        mov.w   #7,  R13
        call    #suma_rp
        mov.b   R14, P3OUT
        swpb    R14
        mov.b   R14  P4OUT
        jump    $
```

// sum from page above

```
arr1 .int 1, 2, 3, 4, 1, 2, 3, 4
arr2 .int  111, -1, -1, -1, -1,
.end
```

Subroutine: SUMA_SP

- Subroutine for summing up elements of an integer array
- Passing parameters through the stack
  - The calling program prepares input parameters on the stack

Main (ver3): Call suma_sp (Pass Through Stack)

```
;-------------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------------
main:       bis.b   #0xFF,&P1DIR            ; configure P1.x as output
            bis.b   #0xFF,&P2DIR            ; configure P2.x as output
            bis.b   #0xFF,&P3DIR            ; configure P3.x as output
            bis.b   #0xFF,&P4DIR            ; configure P4.x as output
            push.w  #arr1
            push.w  #8
            sub.w   #2,SP                   ; making room for sum
            CALL    #suma_rp
            mov.b   @SP P1OUT
            mov.b   1(SP) P2OUT             ;SP+1
            add.w   #6, SP                  ;free the stack
            push.w  #arr2
            push.w  #7
            sub.b   #2, SP
            CALL    #SUMA_rp
            mov.b   @SP, P3OUT
            mov.b   1(SP), P4OUT
            add.w   #6, SP
            jmp     $
arr1:       .int    1, 2, 3, 4, 1, 2, 3, 4  ; the first array
arr2:       .int    1, 1, 1, 1, -1, -1, -1  ; the second array
```

Address	Stack
0x4000 →	xxxxxxxx TOS
0x3FFE	#arr1
0x3FFC	#8
0x3FFA	?Sum
0x3FF8	RA

Subroutine: SUMA_SP

```
;-------------------------------------------------------------------------------
; File       : Lab5_D3_SP.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of an input integer array
; Description: suma_sp is a subroutine that sums elements of an integer array
; Input      : The input parameters are on the stack pushed as follows:
;                 starting addrress of the array
;                 array length
;                 display id
; Output     : No output
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

            .def    suma_sp

            .text
```

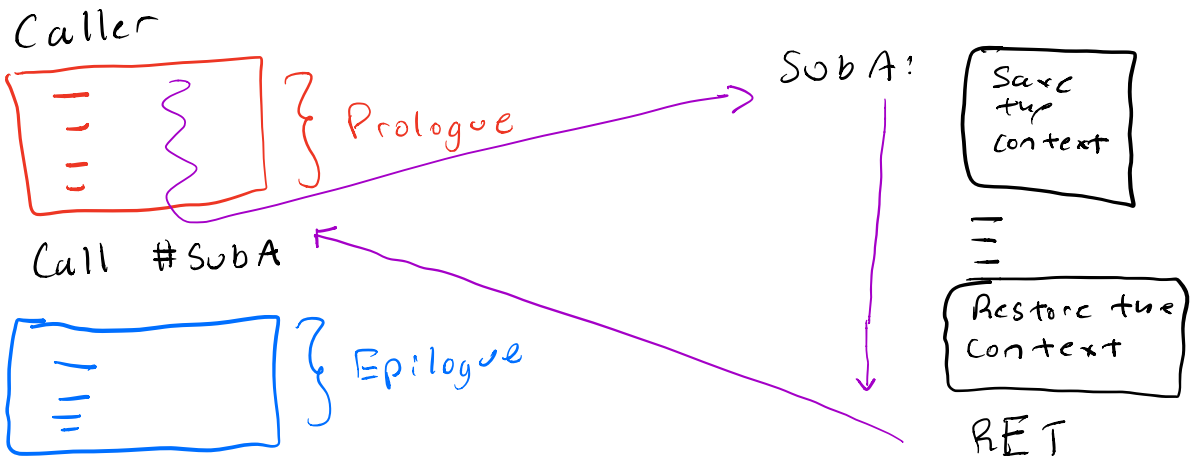
Subroutine: SUMA_SP (cont'd)

```
suma_sp
        .
        .
        .
        push.w  R7   }
        push.w  R6   }  Saving the
        push.w  R4   }  context
        mov.w   12(SP), R4;
        mov.w   10(SP), R6;
        clr.w   R7
lsum:   add.w   @R4+, R7
        dec     R6
        jnz     lsum
        mov.w   R7, 8(SP)
        pop.w   R4
        pop.w   R6
        pop.w   R7
        RET
```

Address	Stack
0x0800	OTOS
0x07FE	#arr1
0x07FC	0008
0x07FA	0000
0x07F8	Ret. Addr.
0x07F6	(R7)
0x07F4	(R6)
0x07F2	(R4)

The Stack and Local Variables

- Subroutines often need local workspace
- We can use a fixed block of memory space – static allocation – but:
  - The code will not be relocatable
  - The code will not be reentrant
  - The code will not be able to be called recursively
- Better solution: dynamic allocation
  - Allocate all local variables on the stack
  - STACK FRAME = a block of memory allocated by a subroutine to be used for local variables
  - FRAME POINTER = an address register used to point to the stack frame




```
int     suma(int *arr, int size)
{
   Local variables
}
```

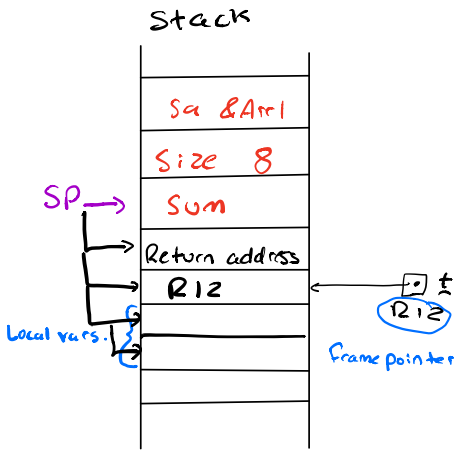
```
suma_sps:
push.w  R12
mov.w   SP, R12
sub.w   #4, SP
```

```
push.w  #arr1
push.w  #8
sub.w   #2, SP
```

Prologue

```
Call    #suma_sps
RA→
mov.w   0(SP), PAOUT
```

Epilogue




```
pop
pop
mov.w   R12, SP
pop     R12
RET
```

Subroutine: SUMA_SPSF

```
;-------------------------------------------------------------------------------
; File       : Lab5_D4_SPSF.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of an input integer array
; Description: suma_spsf is a subroutine that sums elements of an integer array.
;              The subroutine allocates local variables on the stack:
;                  counter (SFP+2)
;                  sum (SFP+4)
; Input      : The input parameters are on the stack pushed as follows:
;                  starting address of the array
;                  array length
;                  display id
; Output     : No output
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
           .cdecls C,LIST,"msp430.h"       ; Include device header file

           .def    suma_spsf

            .text
```

Subroutine: SUMA_SPSF (cont'd)

```
suma_spsf:
        ; save the registers on the stack
        push    R12             ; save R12 - R12 is stack frame pointer
        mov.w   SP, R12         ; R12 points on the bottom of the stack frame
        sub.w   #4, SP          ; allocate 4 bytes for local variables
        push    R4              ; pointer register
        clr.w   -4(R12)         ; clear sum, sum=0
        mov.w    6(R12), -2(R12) ; get array length
        mov.w   8(R12), R4      ; R4 points to the array starting address
lnext:  add.w   @R4+, -4(R12)   ; add next element
        dec.w   -2(R12)         ; decrement counter
        jnz     lnext           ; repeat if not done
        bit.w   #1, 4(R12)      ; test display id
        jnz     lp34            ; jump to lp34 if display id = 1
        mov.b   -4(R12), P1OUT  ; lower 8 bits of the sum to P1OUT
        mov.b   -3(R12), P2OUT  ; upper 8 bits of the sume to P2OUT
        jmp     lend            ; skip to lend
lp34:   mov.b   -4(R12), P3OUT  ; lower 8 bits of the sum to P3OUT
        mov.b   -3(R12), P4OUT  ; upper 8 bits of the sume to P4OUT
lend:   pop     R4              ; restore R4
        add.w   #4, SP          ; collapse the stack frame
        pop     R12             ; restore stack frame pointer
        ret                     ; return
        .end
```

Address	Stack
0x0800	OTOS
0x07FE	#arr1
0x07FC	0008
0x07FA	0000
0x07F8	Ret. Addr.
0x07F6	(R12)
0x07F4	counter
0x07F2	sum
0x0731	(R4)

R12 → 0x07F6 (R12)

SP → 0x0731 (R4)

Performance

- Execution time

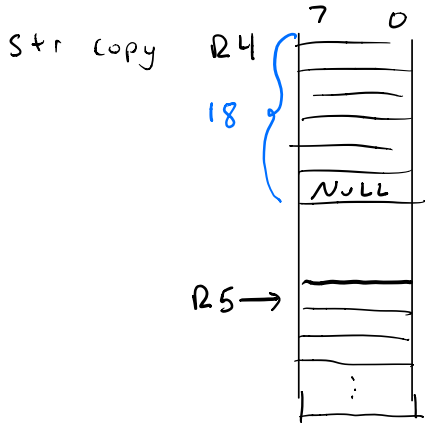
$$ET = IC \times CPI \times CCT = \frac{IC \times CPI}{CCF}$$

ET	≡	Execution Time	Seconds
IC	≡	Instruction Count	Instructions/program
CPI	≡	Cycles Per Instruction	Cycles/instruction
CCT	≡	Clock Cycle Time	time/cycle
CCF	≡	Clock Frequency	

Clock Freq: 2^{20} Hz

Clock cycle time = $1/2^{20}$

Practice:




```
Str copy: nop                          (1)
Inext:  mov.b  @R4+, 0(R5)             4
        tst.b  @R4+                    3
        jz     Iout                    2
        inc.w  R5                      1
        jmp    Inext                   2
Iout:   ret                            5
```

Number of instructions: $1 + 17 \times 5 + 3 + 1 = 90$

(1 = Str nop; 17 = String Chars; 3 = NULL; 1 = ret)

Clock Cycles: $1 + 17(4+3+2+1+2) + (4+3+2) + 5 = 219$

$$ET = \#N \cdot CCT$$
$$= 219 \cdot 1/2^{20} \text{ [s]}$$

$$CPI = \frac{219}{90}$$

MIPS millions of instructions per second

IC, ET → Sometimes considered meaningless.

$$MIPS = \frac{IC}{10^6 \cdot ET} = \frac{IC}{10^6 \cdot \frac{IC \times CPI}{CC\ F}} = \frac{CC\ F}{10^6 \times CPI}$$

FLOPS floating point operations per second