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

Aleksandar Milenkovic

milenka@uah.edu

Admin

→ HW. 3 due date is Friday

→ Quiz. 03 MSP430

→ Sample Exams

→ Misc

Autoincr. Absolute

MOV. W @R5+, &MyVar

2 words

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

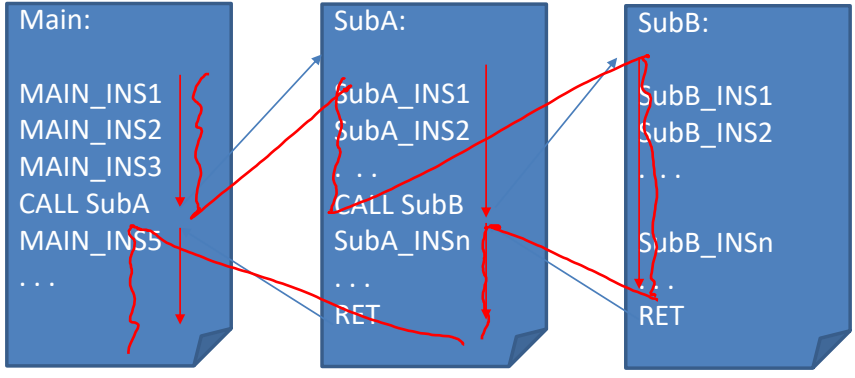
Subroutines

- 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

Subroutines (cont'd)

- CALL instruction:
  perform the branch to subroutines
  - SP <= SP – 2 ; allocate a word on the stack for return 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

Subroutine Nesting



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

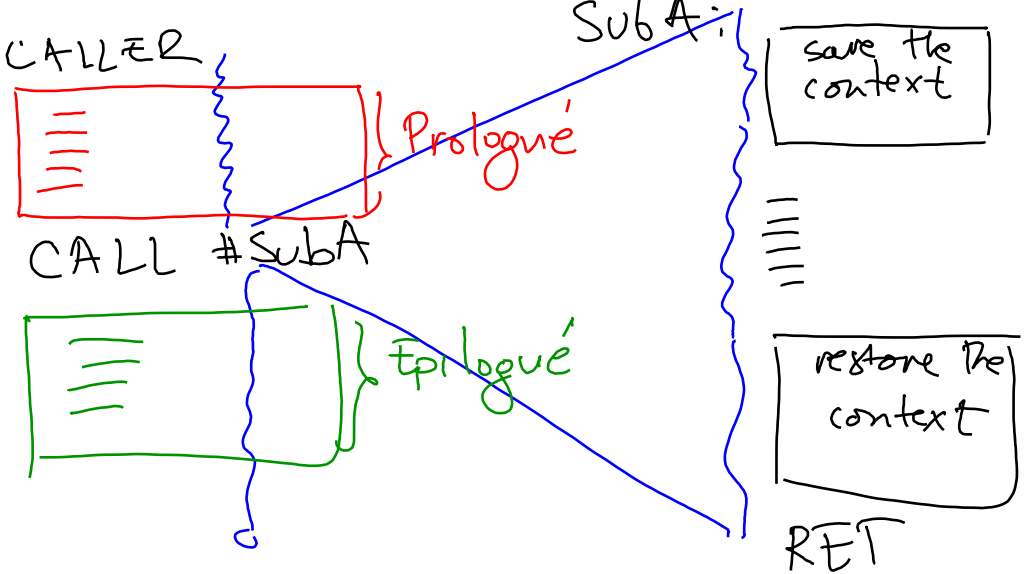
- Subroutine for summing up elements of an integer array
- Passing parameters through registers
  - `R12 - starting address of the array`
  - `R13 - array length`
  - `R14 - returns the sum`

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

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 *aa, int size) {
             local variables

}
```

Prologue

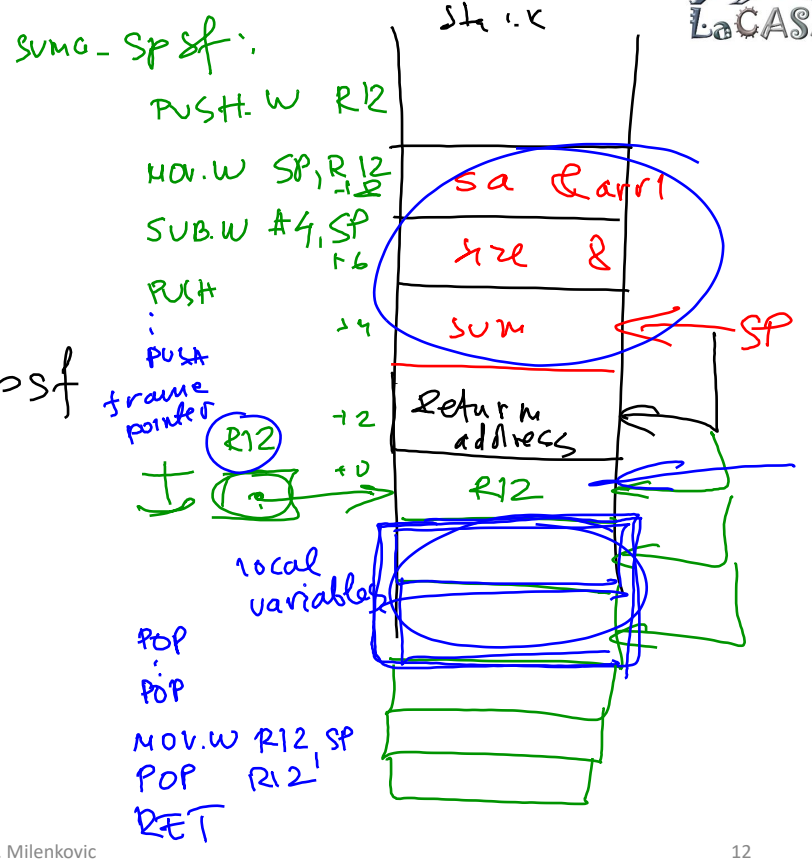
```
PUSH.W  #arr1
PUSH.W  #8
SUB.W   #2, SP
```

```
CALL    #suma_spsf
```

RA →

Epilogue

```
MOV.W   0(SP), PAOUT
ADD     #6, SP
```



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
;                  return 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_spsf

            .text
```

Subroutine: SUMA_SPSF (cont'd)

```
suma_spsf:
        ; save the registers on the stack
        push.w  R12
        mov.w   SP, R12
        sub.w   #4, SP      ; allocate local storage
        push.w  R4          ; store R4 onto the stack
        clr.w   -4(R12)     ; sum = 0
        mov.w   +6(R12), -2(R12) ; counter = 8
        mov.w   +8(R12), R4
Lnext:  add.w   @R4+, -4(R12)
        dec.w   -2(R12)
        jnz     Lnext
        mov.w   -4(R12), +4(R12)
        pop.w   R4
        mov.w   R12, SP
        pop.w   R12
```

SFP-2 points to counter

SFP-4 points to sum

Address	Stack	
0x0800	OTOS	
0x07FE	#arr1	+8
0x07FC	0008	+6
0x07FA	0000	+4
0x07F8	Ret. Addr.	+2
0x07F6	R12	+0
0x07F4		← counter
0x07F2		← sum
0x07F0	R4	← SP
0x07EE		
0x07EC		
0x07EA		
0x07E8		

R12 ⇒ 0x07F6

SP ⇒ 0x07F0

ret

Performance

Instruction / Program × Cycles / Instruction × Time / Cycle

- Execution time

Execution time

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

IC - Instruction Count

CPI - Cycles Per Instruction

CCT - Clock Cycle Time

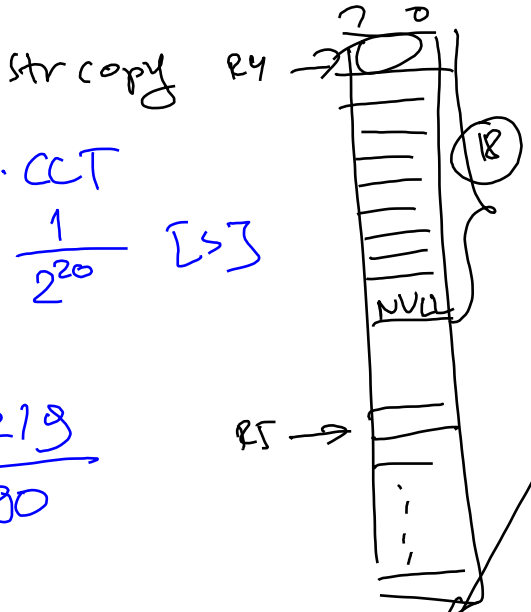
CCF - Clock Frequency

$ET = \#N * CCT$

Clock freq.: 2^{20} Hz

Clock cycle time = $\frac{1}{2^{20}}$

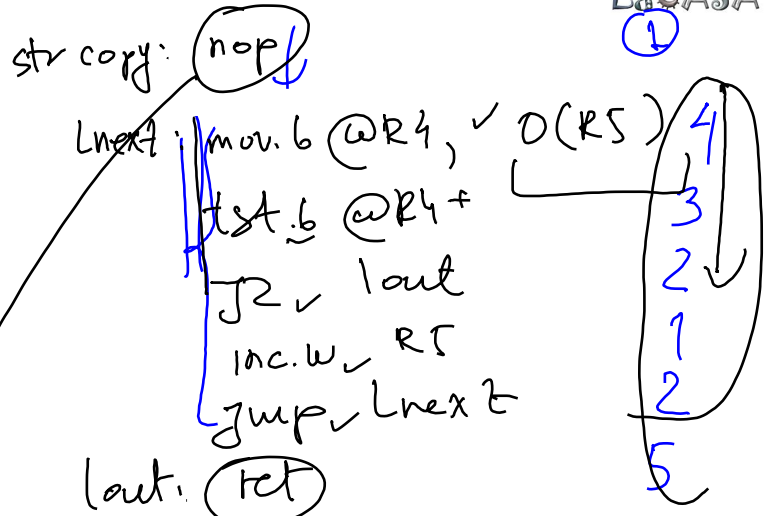
$= \frac{1}{1,042,86}$



$ET = \#N \cdot CCT$

$= 219 \cdot \frac{1}{2^{20}}$ [s]

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



IC = Number of instr.: $1 + (17 \times 5 + 3) + 1 = 90$

(17 x 5 + 3: 5 other characters, NULL; + 1: ret)

Clock cycles: $1 + 17(4+3+2+1+2) + (4+3+2) + 5$

$= 1 + 17 \times 12 + 9 + 5 = 219$

MIPS

Million of Instructions Per Second

IC, ET

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

FLOPS

$$ET = IC \times CPI \times CCT$$