Lecture 16

Subroutines IV

Nested Subroutine Calls

Recursions

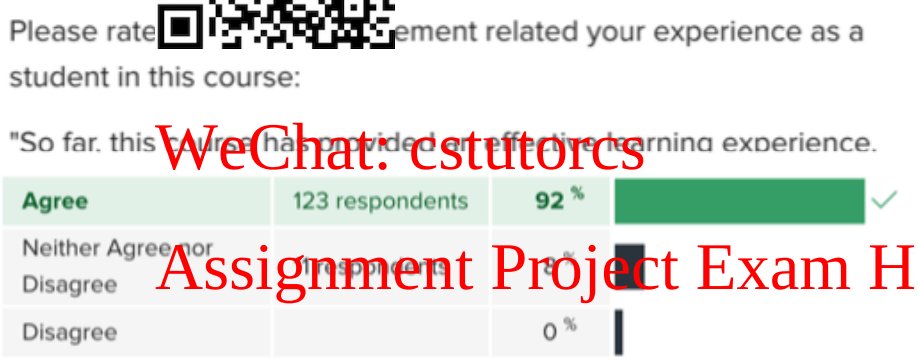
Division

Class Feedback

Thank you for the feedback you have provided on Carmen

I have read every comme[illegible]n and considered it

Where are we?



Not bad at all!

- Mostly happy with the structure of class, especially in class coding

⇒ Will continue to do in class coding, but we have to cover material too!

- Mostly happy with the lecture materials (i.e., slides)
- Mostly happy with assignments

⇒ Will try to add more optional challenges

Class Feedback

Things that we will have to live with:

- Class time – I am not a [illegible] pm class on Friday either
- Class time – only 2 lec[illegible]eek needs to be spent between lecture and in-class coding
- Slides posted before class – I prepare them right before class

What can be improved?

- Issues with audio in class and in recordings ⇒ asked ETS for help
- More practice problems ⇒ I will try to post more ideas in lecture slides

- Issues with CCS ⇒ use the Discord channel

- Having a code dictionary ⇒ already posted to Carmen

- Mandatory attendance ???

Looking Ahead

Topics to be covered:

- Finish subroutines
- General Purpose Input/Output (GPIO)
  - Two LEDs and two switches on the Launchpad
  - Other HW modules will have to wait for ECE 3567 MCU Lab
- Interrupts
- Timers

Assignments:

- Project – Working with CCS tools (import & visualize data) and Q-format
- Possibly a small quiz preparing for Midterm
- Midterm #2 – GPIO Interrupts
- Final Exam – Take home, over Final's week

Solution to Midterm #1

Memory allocation

- mad in RAM
- samples in FRAM

```
                .data
                .retain
                .retainrefs

mad:            .space 2

                .text
                .retain                         ; Assemble into program memory.
                .retainrefs                     ; Override ELF conditional linking
                                                ; And retain any sections that have

samples:        .word 46, 84, 11, 20, 39, 91, 57, 17, 71, 27, 63, 4, 36, 88, 62, 52
LENGTH:         .set 32                         ; length of array in bytes
```

Solution to Midterm #1

Computing the mean

```
; Compute the mea
            clr.w                                ; Index
            clr.w                                ; Mean

accumulate:
            add.w   samples(R4), R5
            incd.w  R4
            cmp.w   #LENGTH, R4
            jlo     accumulate

; R5 = sum(a) -- divide by 16 to find mean
            rra.w   R5
            rra.w   R5
            rra.w   R5
            rra.w   R5                           ; R5 = mean = 48
```

Solution to Midterm #1

Computing the mad – Version 1

```
; Compute the MAD
            clr.w   R4                      ; Index
            clr.w   R7                      ; MAD

read_next:  mov.w   array(R4), R6
            sub.w   mean, R6                ; R6 = deviation from mean
            jge     positive

negative:   inv.w   R6                      ; abs(R6) = -R6 if R6 < 0
            inc.w   R6                      ; 2's complement of R6

positive:   add.w   R6, R7

            incd.w  R4
            cmp.w   #LENGTH, R4
            jlo     read_next

; R7 = sum(abs(a-mean(a))) -- Divide R7 by 16 to find MAD
            rra.w   R7
            rra.w   R7
            rra.w   R7
            rra.w   R7                      ; R7 = MAD

            mov.w   R7, mad                 ; MAD = 23

main:       jmp     main
            nop
```

Solution to Midterm #1

Computing the mad – Version 2

```
; Compute the MAD
            clr.w   R4                  ; Index
            clr.w   R7                  ; MAD

read_next:  mov.w   array(R4), R6
            sub.w   mean, R6            ; R6 = deviation from mean
            jn      negative

positive:   add.w   R6, R7              ; abs(R6) = R6 if R6 >= 0
            jmp     proceed_to_next

negative:   sub.w   R6, R7              ; abs(R6) = -R6 if R6 < 0

proceed_to_next:
            incd.w  R4
            cmp.w   #LENGTH, R4
            jlo     read_next

; R7 = sum(abs(a-mean(a))) - Divide R7 by 16 to find MAD
            rra.w   R7
            rra.w   R7
            rra.w   R7
            rra.w   R7                  ; R7 = MAD = 23

            mov.w   R7, mad

main:       jmp     main
            nop
```

Quiz 5

Task: Write a subroutine that checks whether a given unsigned integer is prime or composite

```
;-------------------------------------------------------------------
; Subroutine: is_pri
; Inputs: unsigned v                6 -- returned unchanged
;
; Output: binary value in R13 -- R13 = 1 if n is prime
;                                R13 = 0 if n is composite
;
; All other core registers in R4-R15 unchanged
; Subroutine does not access addressed memory locations
;-------------------------------------------------------------------
```


Hint: A prime number is divisible only by 1 and itself

Use subroutine `is_divisible` from Quiz_4 to check this condition

Also: Always good practice to think about efficiency when writing code:

How can you improve **execution time** and/or memory usage?

Solution to Quiz 4

```
;-------------------------------------------------------------------------
; Subroutine: is_divisible
; Inputs: unsigned 16-bit integer x in R5 -- returned unmodified
;         unsigned 16-bit integer y in R6 -- returned unmodified
;
; Output: binary value                    = 1 if x|y
;                                         = 0 otherwise
;
; All other core regist                   unchanged
; Subroutine does not access addressed memory locations
;-------------------------------------------------------------------------
is_divisible:
            push    R6                  ; save R6 on stack

            clr.w   R12                 ; assume not divisible

check:      cmp.w   R5, R6              ; if R5 > R6
            jlo     found_remainder     ; R6 holds the remainder

            sub.w   R5, R6              ; if R6 >= R5 subtract R5 from R6
            jhs     check

found_remainder:
            tst.w   R6                  ; R6 is the remainder, check if zero
            jnz     ret_from_is_divisible   ; if not zero => not divisible

            mov.w   #1, R12             ; here R6=0 => divisible

ret_from_is_divisible:
            pop     R6                  ; restore R6 from stack
            ret
```

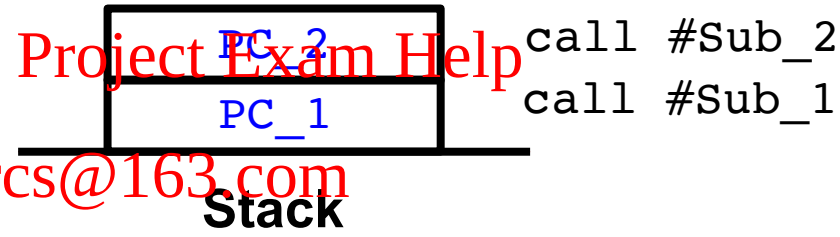
Subroutines Calling Subroutines

A subroutine can call another subroutine

```
;-------------------------
; Main loop here
;-------------------------
            ; do things
            call    #Sub_1
            ; do more things

Loop:   jmp Loop

;-------------------------------
; Subroutine: Sub_1
;-------------------------------
Sub_1:
            call    #Sub_2
            ret

;-------------------------------
; Subroutine: Sub_2
;-------------------------------
Sub_2:
            ; do something
            ret
```

Is there a limit to nesting subroutines?

Every subroutine call reserves at least 2 bytes on stack until returned



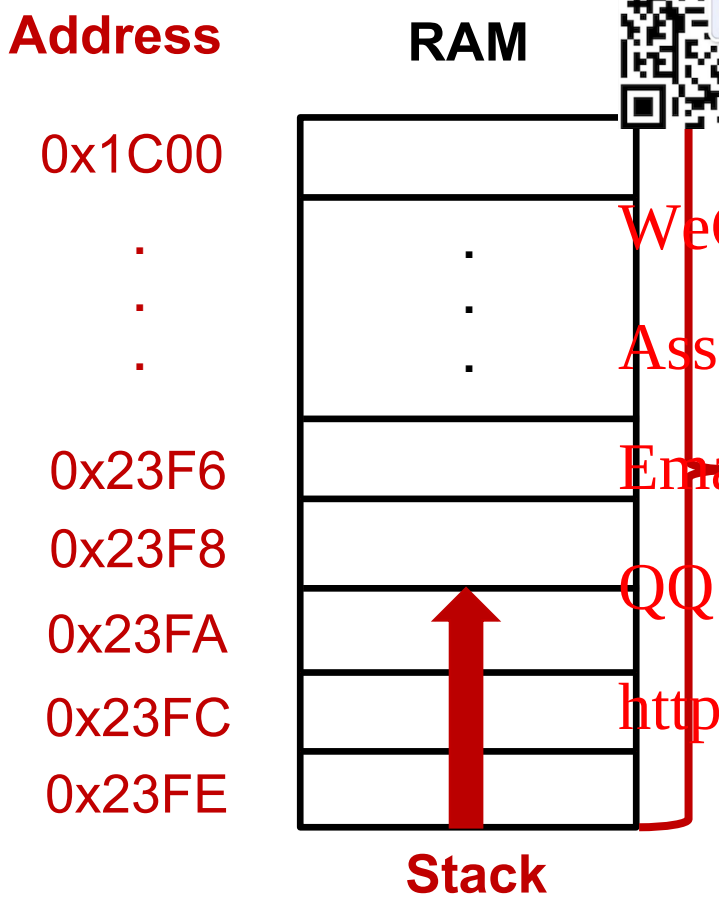
You cannot nest arbitrarily many subroutine calls.
What is the limit?

At most 1024 – often less!

Size of the Stack

How much data can we push onto the stack?

Max. 1024 words – less if [illegible]llocated .data at the beginning of program



Size of stack at any point:

0x2400 - SP in bytes

If we push too much data onto the stack it will start to overwrite the data allocated at the top of RAM

⇒ **Stack overflow**

When stack pointer crosses top of RAM

⇒ **Crash**

Avoid at all cost!

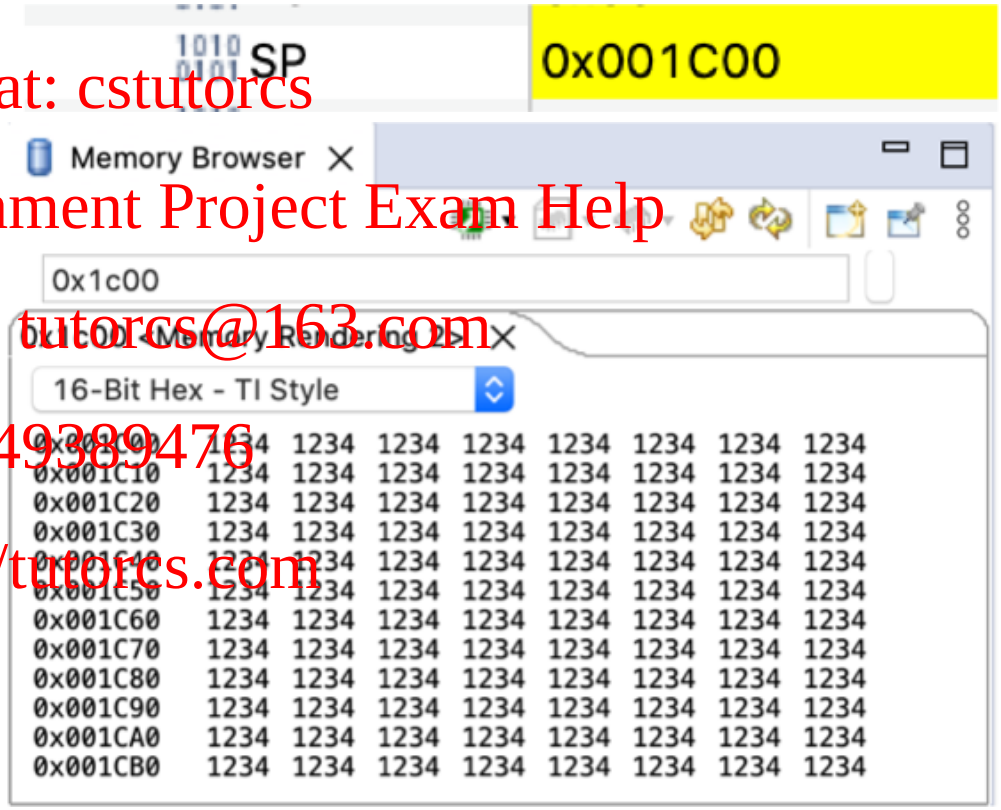
Size of the Stack

Question: Does CCS help preventing stack overflow?

Smashing the Stack for [illegible] Profit)

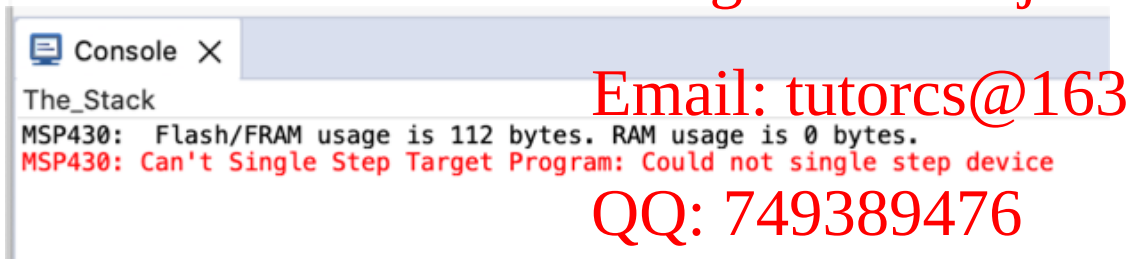
fter executing 1024 push instructions

```
;-------------------------------
; Main loop here
;-------------------------------
            mov.w   #1024, R5

Repeat:     push.w  #0x1234
            dec.w   R5
            jnz     Repeat

            push.w  #0x1234

Inf_Loop:   jmp     Inf_Loop
            nop
```



Size of the Stack


```
;-------------------------------
; Main loop here
;-------------------------------
            mov.w   #1024, R

Repeat:     push.w  #0x1234
            dec.w   R5
            jnz     Repeat

            push.w  #0x1234

Inf_Loop:   jmp     Inf_Loop
            nop
```



After executing the 1025th push

Question: Does CCS help preventing stack overflow?

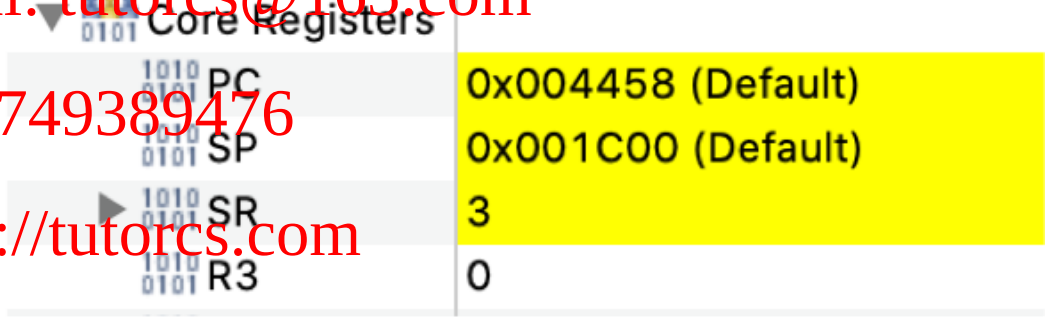
Answer: **No!**

Recursion with Subroutines

Recursion is a great programming trick **BUT** be careful when doing recursions with limited stack size

```
gcd:
            cmp.w   R5, R6          ensures that the larger value is in the cc
            ;more lines
            call    #gcd            ;recursively calls GCD again until GCD is found
End:
            ;mov.w  R5, R6
            ret                     ;end of subroutine after recursive call, all inst
```


Why not find gcd(1024, 1)



What about gcd(**1025**, 1)? **Crash! Boom! Bang!**

Recursion with Subroutines

Recursion is a great programming trick **BUT** be careful when using it!

Easy fix:

Instead of a call use a ju


```
gcd:
        cmp.w   R5, R6
        ;more lines
        call    #gcd

End:
        ;mov.w  R5, R6
        ret
```

```
gcd:
        cmp.w   R5, R6
        ; more lines
        jmp gcd

End:
        ;mov.w  R5, R6
        ret
```

Last Time

We wrote a subroutine to multiply 2 bytes in R5 and R6 and return the result in R12

```
;-------------------------------------------------------------
; Subroutine:
; Inputs: unsi                in R5 -- returned unchanged
;         unsigned byte y in R6 -- returned unchanged
;
; Output: unsigned number in R12 -- R12 = R5 * R6
;
; This time implement the long multiplication algorithm
;
; All other co                  R4-R10 unchanged
;-------------------------------------------------------------
x_times_y:
```


Main idea was to

- test the bits of one of the numbers (`R5`) one by one using `bit.w`
- add a left shifted version of the other number (`R6`) if a bit is 1

Binary Long Multiplication v.2

Alternative way of sequentially testing bits

```
rra.w   R5
```

shift/roll right arithmetic

right most bit in R5 → C status bit

use **jc** or **jnc** to control the flow

Does not require a bitmask to test the bits of R5

However, unlike bit test **bit.w**, roll right arithmetic **rra.w** **modifies R5**

No big deal! We know how to fix it.

Makes a great practice problem!

Binary Long Multiplication v.2


```
x_Times_y:
            push.w  R5
            push.w  R6
            push.w  R10

            clr.w   R12

            mov.w   #8, R10                 ; R10 will count through the bits

Repeat2:    rra.w   R5
            jnc     Next_bit2

            add.w   R6, R12

Next_bit2:  rla.w   R6
            dec.w   R10
            jne     Repeat2

            pop.w   R10
            pop.w   R6
            pop.w   R5

            ret
```

Division by a Power of Two

Subroutine that divides x by 2^p

```
;-------------------------------------------------------------
; Subroutine: x_d                                        P
;
; Inputs: signed                                in R5 -- returned unchanged
;         unsigned number p in R6 -- returned unchanged
;
; Output: signed number in R12 -- R12 = Floor(R5 / 2^p)
;
; All other core registers in R0-R6 unchanged
;-------------------------------------------------------------
```


How do we solve this problem?

One Solution


```
;-------------------------------------------------------------------
; Subroutine: x_div_2powerP
;
; Inputs: signed R5 -- returned unchanged
;         unsigned R6 -- returned unchanged
;
; Output: signed R12 = Floor(R5 / 2^p)
;
; All other core registers in R4-R15 unchanged
;-------------------------------------------------------------------
x_div_2powerP:
            push    R6

; Start with x in R12
            mov.w   R5, R12

; Shift x in R12 R6 times to the right
; Make a loop with R6 as counter
_repeat:
            tst.w   R6              ; Possible to have R6=p=0
            jz      _end            ; corresponding to dividing by 1

            rra.w   R12             ; shift R12 once
            dec.w   R6              ; account for the shift
            jnz     _repeat

_end:       pop     R6
            ret
```