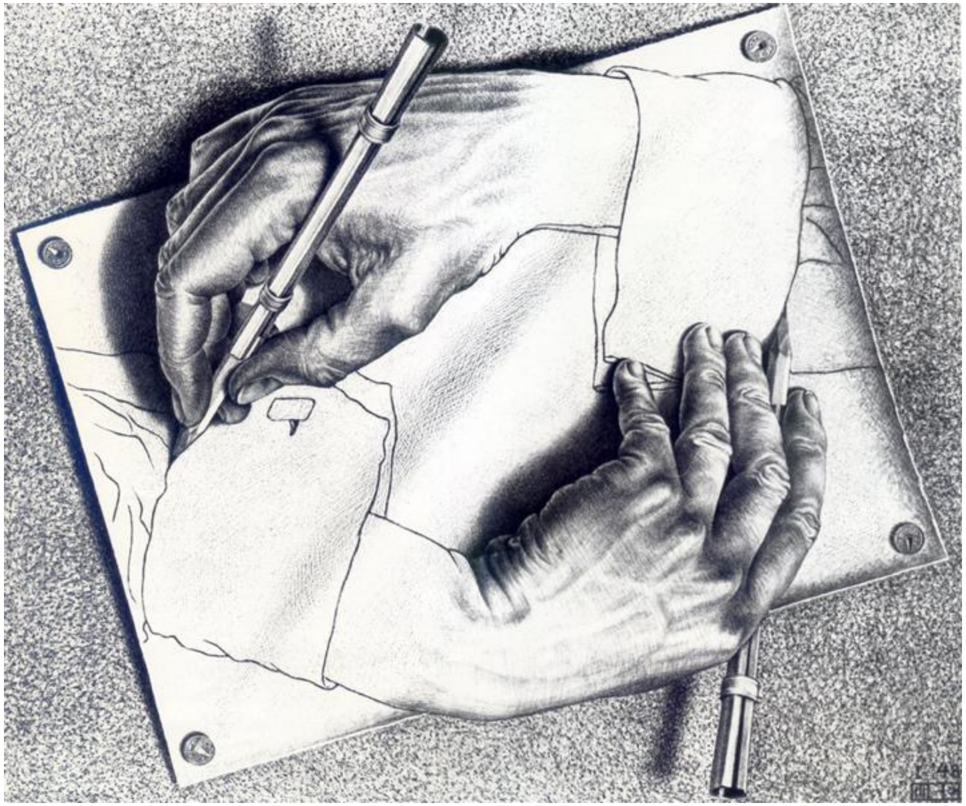
Lecture 16

Subroutines IV



Nested Subroutines

Recursions

Division

Stack Frames

Subroutines Calling Subroutines

A subroutine can call another subroutine

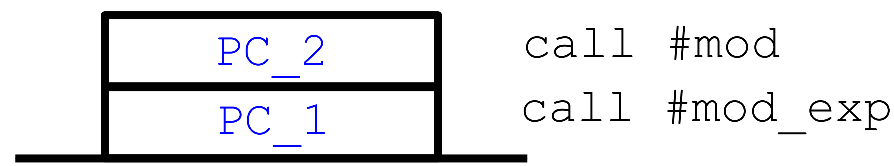
```
;-------------------------------------------
; Main loop here
;-------------------------------------------
            ; prepare the input
            call    #mod_exp
            ; record the output

Done:       jmp     Done

;-------------------------------------------
; Subroutine: mod_exp
;-------------------------------------------
mod_exp:
            ; do things
            call    #mod_exp
            ret
;-------------------------------------------
; Subroutine: mod
;-------------------------------------------
mod:
            ; find x % N
            ret
```

Is there a limit to nesting subroutines?

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



Stack

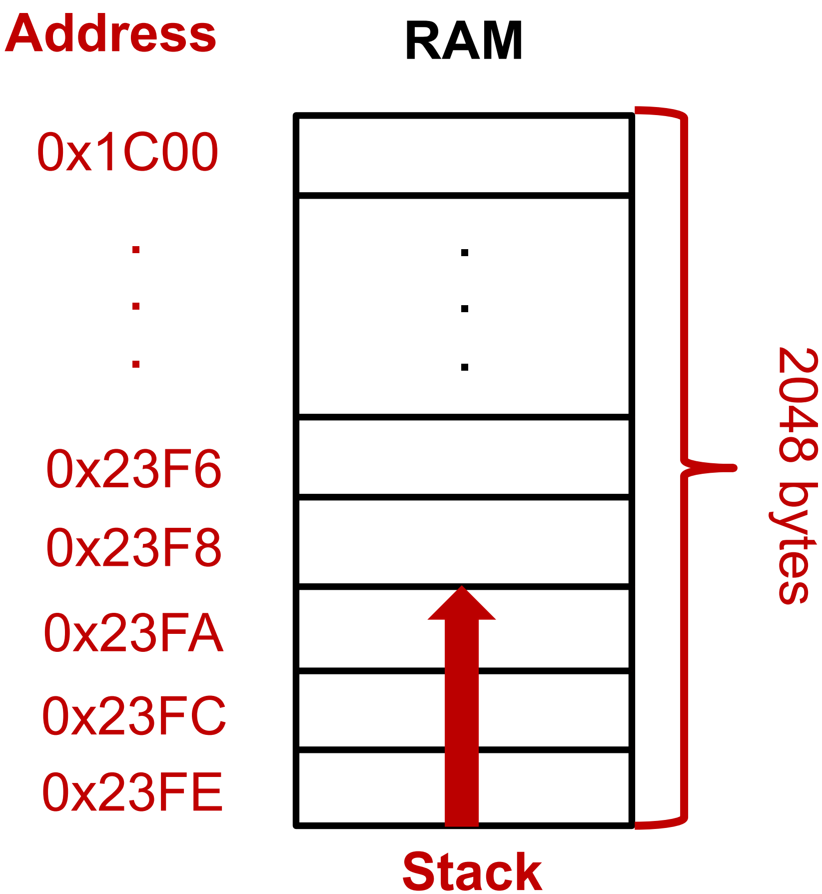
You cannot nest arbitrarily many subroutine calls
What is the limit?

At most 1024 – often less!

Size of the Stack

How much data can we push onto the stack?

Max. 1024 words – less if we have allocated .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!

Recursion with Subroutines

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

```
gcd:
            cmp.w   R5, R6      ;Makes 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)?

Core Registers	
PC	0x004458 (Default)
SP	0x001C00 (Default)
SR	3
R3	0

Console

The_Stack

```
MSP430:  Flash/FRAM usage is 112 bytes. RAM usage is 0 bytes.
MSP430: Can't Single Step Target Program: Could not single step device
```

Crash!

Recursion with Subroutines

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

Easy fix:

Instead of a call use a jump!

```
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
```

Integer Division

The integers $\mathbb{Z}$ form a ring: you can add, subtract, multiply but *not* divide

When a (non-negative) integer $\boldsymbol{x}$ is divided by (positive) integer $\boldsymbol{N}$ the result is a quotient $\boldsymbol{q}$ and a remainder $\boldsymbol{r}$ with $\boldsymbol{0} \leq \boldsymbol{r} < \boldsymbol{N}$ s.t.

$$x = q\,N + r$$

We write:

$$x \,/\, N = q$$
$$x \,\%\, N = r$$

You are asked to implement the modulus operator $\boldsymbol{x \,\%\, N}$ for Quiz 5

Let's do the division part in class

Coding: Integer Division

Task: Write a subroutine that does integer division with following contract

```
;-------------------------------------------------------------------------------
; Subroutine: x_div_N
; input: R5 unsigned 16-bit integer x -- returned unchanged
;        R6 unsigned 16-bit nonzero integer N -- returned unchanged
;           you can assume that N in R6 is nonzero, no need to check
;
; output: R12 unsigned 16-bit integer y
;           y = floor(x / N)
;           i.e., y is the integer part when x is divided by N
;
; All other core registers in R4-R15 unchanged
;-------------------------------------------------------------------------------
```

How do we do this?

What can we do? There is no instruction to do division

What is division? Repeated subtraction

How does division work?

When you have the simplest instructions – as is the case in assembly you have to break down a task into its simplest steps!!!

Get your hands dirty: Start with an example (on paper or tablet)

e.g., ***x = 17*** and ***N = 5***

To find ***17 / 5*** we repeatedly subtract ***5*** from ***17***

How many times? We do not know – that is the result we are looking for

When do we stop subtracting?

This we know – until we no longer can subtract because the result is < 5

Let's do it

17 - 5 = 12

12 - 5 = 7

7 - 5 = 2

We are done since ***2 < 5***

We were able to subtract 3 times

Hence

17 / 5 = 3

BONUS: We have found 17 % 5 = 2 too!

Example to Pseudocode

We have a great starting point to compute `y = x / N`

1. Initialize `y = 0`
2. Subtract `N` from `x` until we can no longer subtract
3. For every subtraction increase `y` by 1

Are we good? Is there anything that can go wrong?

Details !!!

Well, what about ***4 / 5?***

We have to account for the case when `x < N` to start with

Easy:

1. Initialize `y = 0`
2. Check if `x < N`
3. If not `x < N` subtract `N` from `x` and increase `y` by 1
4. If `x < N` we are done

More Details

We have an algorithms that should work – at least in theory

But theory ≠ practice

1. Initialize `y = 0`
2. Check if `x < N`
3. If not `x < N` subtract `N` from `x` and increase `y` by 1
4. If `x < N` we are done

Practice is 16 bits – always watch for overflow

- This time we are safe – we are reducing the number

Is there any danger of an infinite loop?

- Always – if we do not handle the stopping condition well

What about the contract?

- `x`, `y`, and `N` all unsigned 16-bit integers – **use the correct jumps !!!**
- `y` will be in `R12` – easy, we are free to change it
- We will need to change `x` too `x←x-N`

push & pop

A Word About `mod_exp`

Task: Write a subroutine that computes `y = x^e mod N`

What to do? Follow the same steps as before

First, assess the problem:

Exponentiation is repeated multiplication – multiply 1 `e`-times by `x`

Big issue here is **OVERFLOW -- `x^e` can be anything !!!**

But if we take mod after each multiplication, the result will always be `<N`

Get your hands dirty: Compute ***y = 5^3 mod 7***

1. Start with ***y = 1***
2. Multiply by ***5***, take ***mod 7*** ***y = y*5 mod 7*** ***1*5 = 5 mod 7 = 5***
3. Multiply by ***5***, take ***mod 7*** ***y = y*5 mod 7*** ***y*5 = 25 mod 7 = 4***
4. Multiply by ***5***, take ***mod 7*** ***y = y*5 mod 7*** ***y*5 = 20 mod 7 = 6***
5. We are done: we have multiplied ***e=3*** times

A Word About `mod_exp`

At this point you have a good starting algorithm:

1. Initialize `y = 1`
2. Multiply by `x (call #x_times_y)`
3. Reduce mod `N (call #mod)`
4. Decrease `e` by 1 to account for one multiplication by `x`
5. Stop when `e` hits zero – we have multiplied `e` times

Then of course there are the details

- What can go wrong?
- Contracts of subroutines – how to handle the input and output?
- **`push`** and **`pop`**
- **Test !!!**