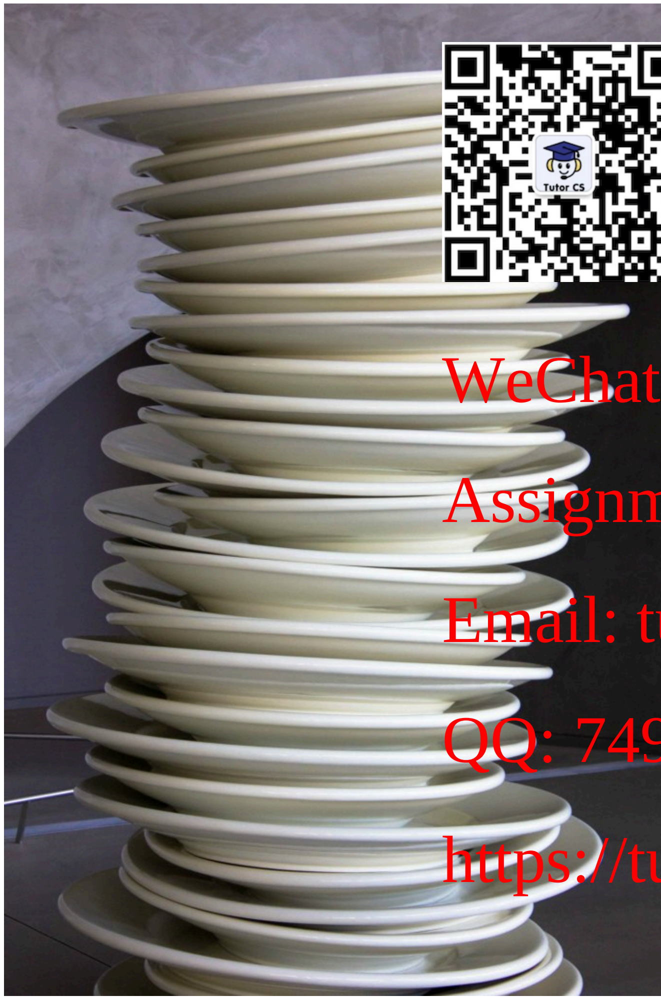


Lecture 17

Subroutines V

Stack Frames

Passing Data over the Stack

A Corny Joke

What happens when you push corn onto the stack?

corn

stack

popcorn

Last Time: The Stack

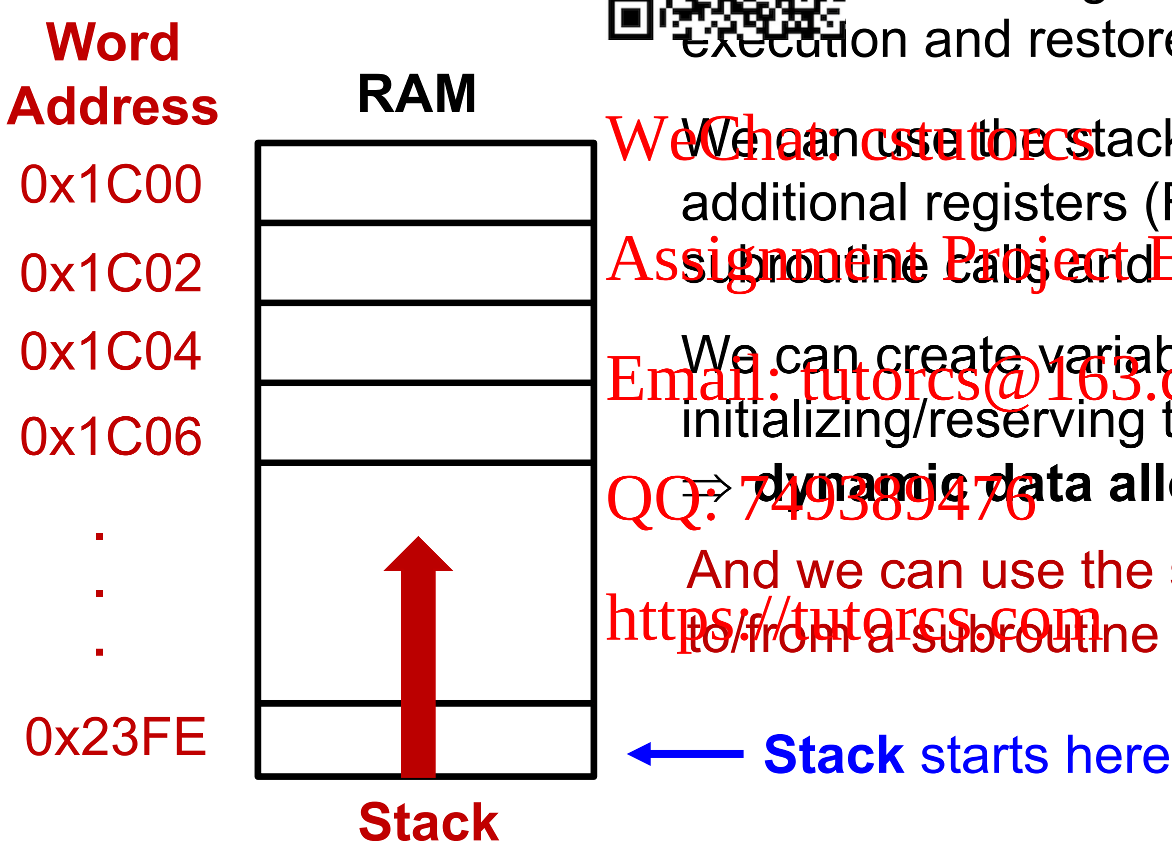
The **stack** is a data structure that is managed at the end of the RAM managed using **SP = R1**… d **pop**

- …tine calls and interrupts use the stack … critical registers (PC and SR) before execution and restore these with `ret/reti`

- We can use the stack to save/restore additional registers (R4 – R15) during subroutine calls and interrupts

- We can create variables during runtime without initializing/reserving them at compile time
  ⇒ **dynamic data allocation**

- And we can use the stack to pass input/output to/from a subroutine



Passing Data to Subroutines

A **subroutine** is a sequence of instructions that performs one specific task

Most subroutines take so rguments and return some output

Input/output should **not** b ed – otherwise subroutine cannot be reused for different sets of input

e.g., subroutine to multiply two unsigned bytes

When cal **outine**

```
;--------------------------------------------------------------
; Subroutine: x_Times_y
; Inputs: unsigned byte x in R5 -- returned unchanged
;         unsigned byte y in R6 -- returned unchanged
;
;
; Output: unsigned number in R12 -- R12 = R5 * R6
;
```


When the **caller**

In these examples the input/output is passed over core registers

- How data is passed is specified in the contract

Passing Data over the Stack

Another way to pass data between caller and subroutine is to use the **stack**

When calling the subroutine we need to pass `x` and `y` to the subroutine

- We place `x` and `y` on the stack where the subroutine can find it

When the subroutine returns it needs to pass the output to the caller

- Subroutine places `x*y` on the stack where we can find it

In both cases, not an absolute address, but relative to the stack pointer `SP`

We define a stack frame with fields for input, output and return address

- caller places **input(s)** into the stack frame using `push`
- with the call the return address (`PC`) is placed into the stack frame
- subroutine places its **output(s)** into the stack frame
- returning from the subroutine (`ret`) removes the saved `PC` from stack
- caller cleans up the rest of the stack

Example Stack Frame

The subroutine contract specifies the structure of the **stack frame**

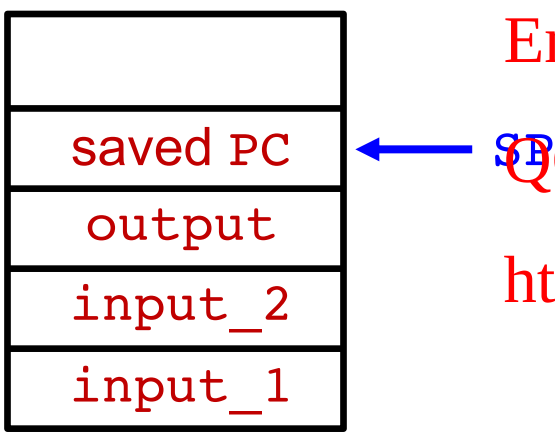
subroutine will see when it is first called

e.g., a stack frame with tw alues and one output value

- Caller pushes `input_1`, then `input_2`
- With the subroutine call `PC` is placed onto the stack

Subroutine

- reads `input_1` and `input_2`
- computes and writes `output` into the stack frame



- `ret` from subroutine removes `PC` from stack

Caller

- reads `output` from stack frame
- cleans up the rest of the stack

Example Stack Frame

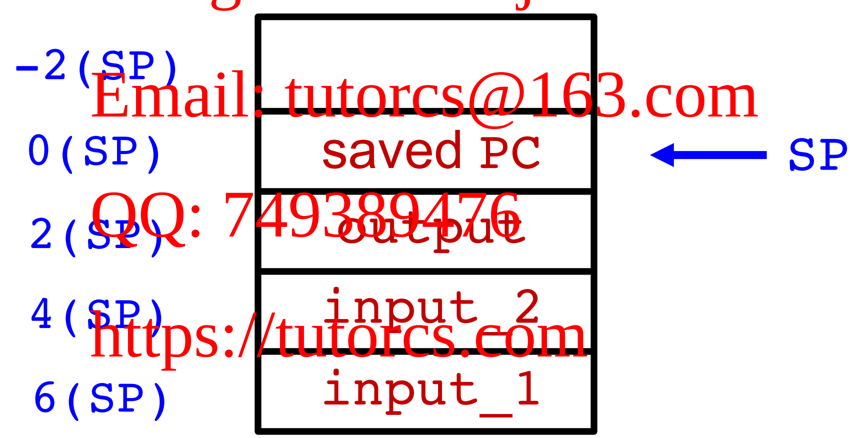
Subroutine

- reads `input_1` and i
- computes and writes c o the stack frame

How does the subroutir s these elements of the stack frame?

Indexed mode of addressing

even this works!



Indexed Mode of Addressing


```
x:  .word   0x0100,   0x0200,   0x0300
```

```
        mov.w   x(R4), R5
```

x is an address – a number e.g., 0x1C00

(R4) is another number – e.g., 2

same as

```
        mov.w   &0x1C02, R5
```


Works for any label (or number) an core register

e.g.,


```
        mov.w   0(SP), R5
        add.w   2(SP), 4(SP)
```

Putting Everything Together

A subroutine `almost_fib` that

- reads `x` and `y`
- returns `x+y`

from to stack with followin

stack frame

0(SP)	saved PC	← SP
2(SP)	x+y	
4(SP)	x	
6(SP)	y	

Caller function prepares stack frame

```
push    #y
push    #x
push    #0
```

Inside `almost_fib`

```
mov.w   4(SP), 2(SP)
add.w   6(SP), 2(SP)
ret
```

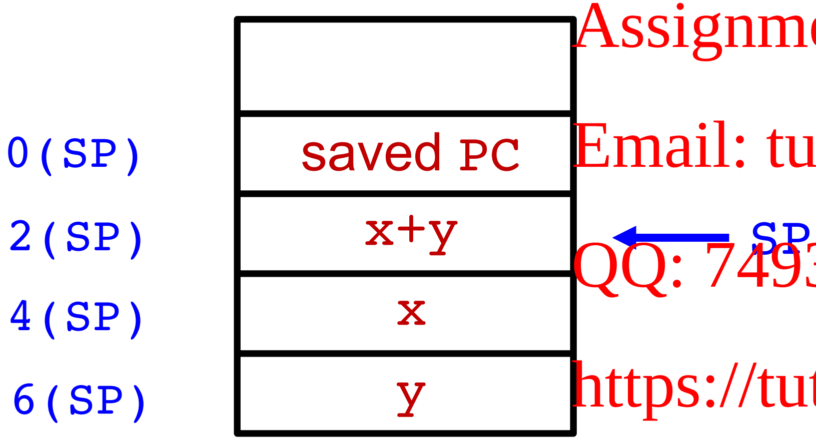
Putting Everything Together

A subroutine `almost_fib` that

- reads `x` and `y`
- returns `x+y`

from to stack with followi

stack frame



After returning from `almost_fib` the stack pointer changes!!!

Inside caller function

```
pop    R5        ; x+y -> R5

add.w  #4, SP    ; restores SP
                 ; we do not care
                 ; for x or y
```

Today's In Class Coding

When there is `almost_fib` there must be a `fib`

Subroutine f

x

using followi

```
;-------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------
            ; number of iterations
            push    #10         ; n=10
            ; initial conditions for Fibonacci
            push    #1          ; x[n-2] here n=0 -> x[-2]=1
            push    #1          ; x[n-1] here n=0 -> x[-1]=1
            ; last element is the output
            push    #0          ; reserve space for x[n]

            call    #fib

            pop     R5          ; n'th number -> R5
            add.w   #6, SP      ; clear up rest of stack !!!!

main:       jmp     main
```

0(SP)	
2(SP)	
4(SP)	
6(SP)	x[n-2]
8(SP)	n

Today’s In Class Coding


```
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------

            ; number of iterations
            push    #10         ;
            ; initial conditions f
            push    #1          ;                0 -> x[-2]=1
            push    #1          ;                0 -> x[-1]=1
            ; last element is the output
            push    #0          ; reserve space for x[n]

            call    #fib

            pop     R5          ; nth number -> R5
            add.w   #6, SP      ; clear up rest of stack!!!

main:       jmp     main

fib:
                ; add up previous two numbers in 2(SP)
                mov.w   4(SP), 2(SP)    ; 2(SP) = x[n-1]
                add.w   6(SP), 2(SP)    ; 2(SP) = x[n-1] + x[n-2]

                ; shift numbers for next iteration
                mov.w   4(SP), 6(SP)    ; x[n-1] becomes x[n-2], x[n-2] dropped
                mov.w   2(SP), 4(SP)    ; x[n] becomes x[n-1]

                dec.w   8(SP)           ; one more iteration complete
                jne     fib

                ret                     ; do not forget the return
```