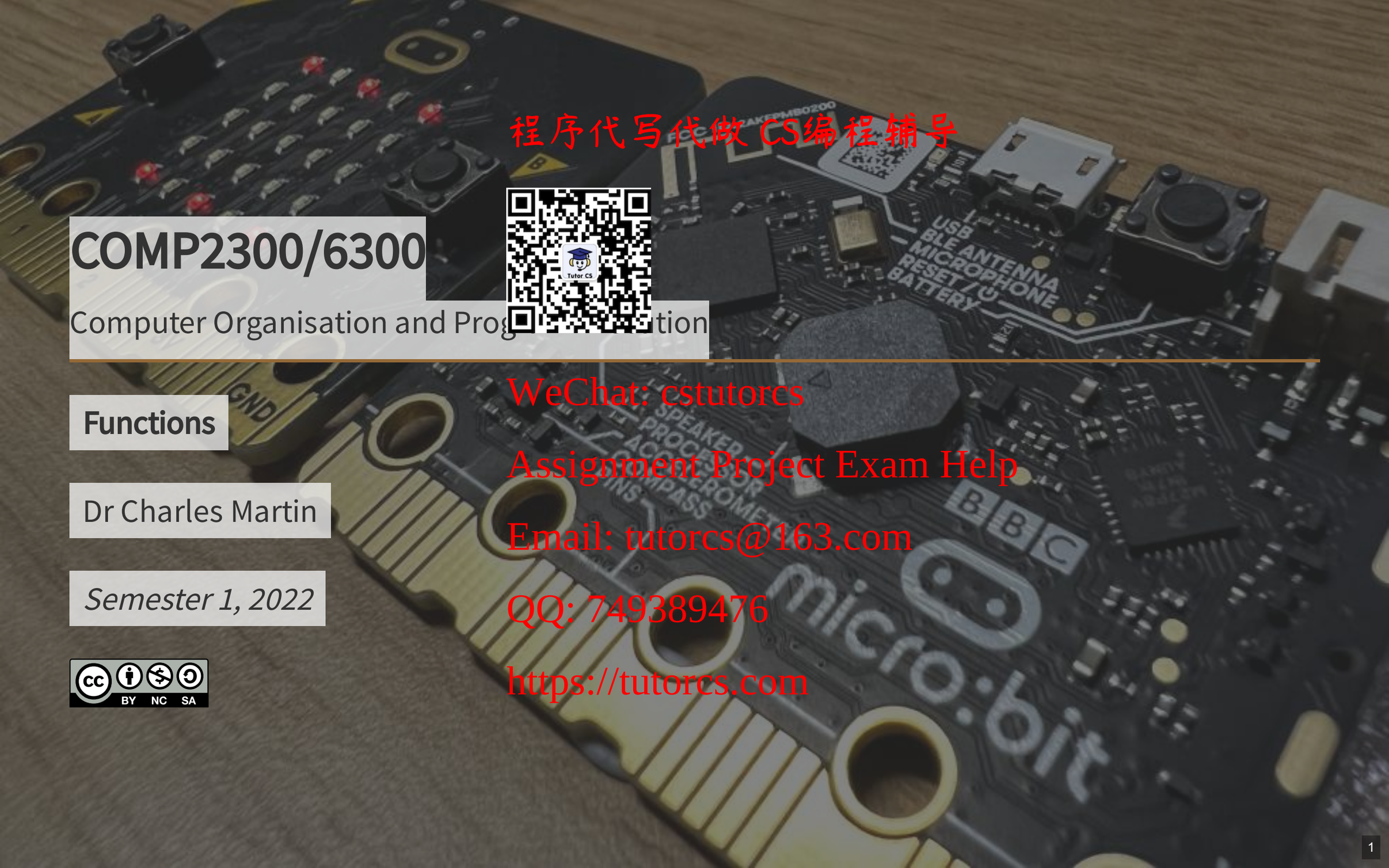


COMP2300/6300

Computer Organisation and Prog[illegible]tion

Functions

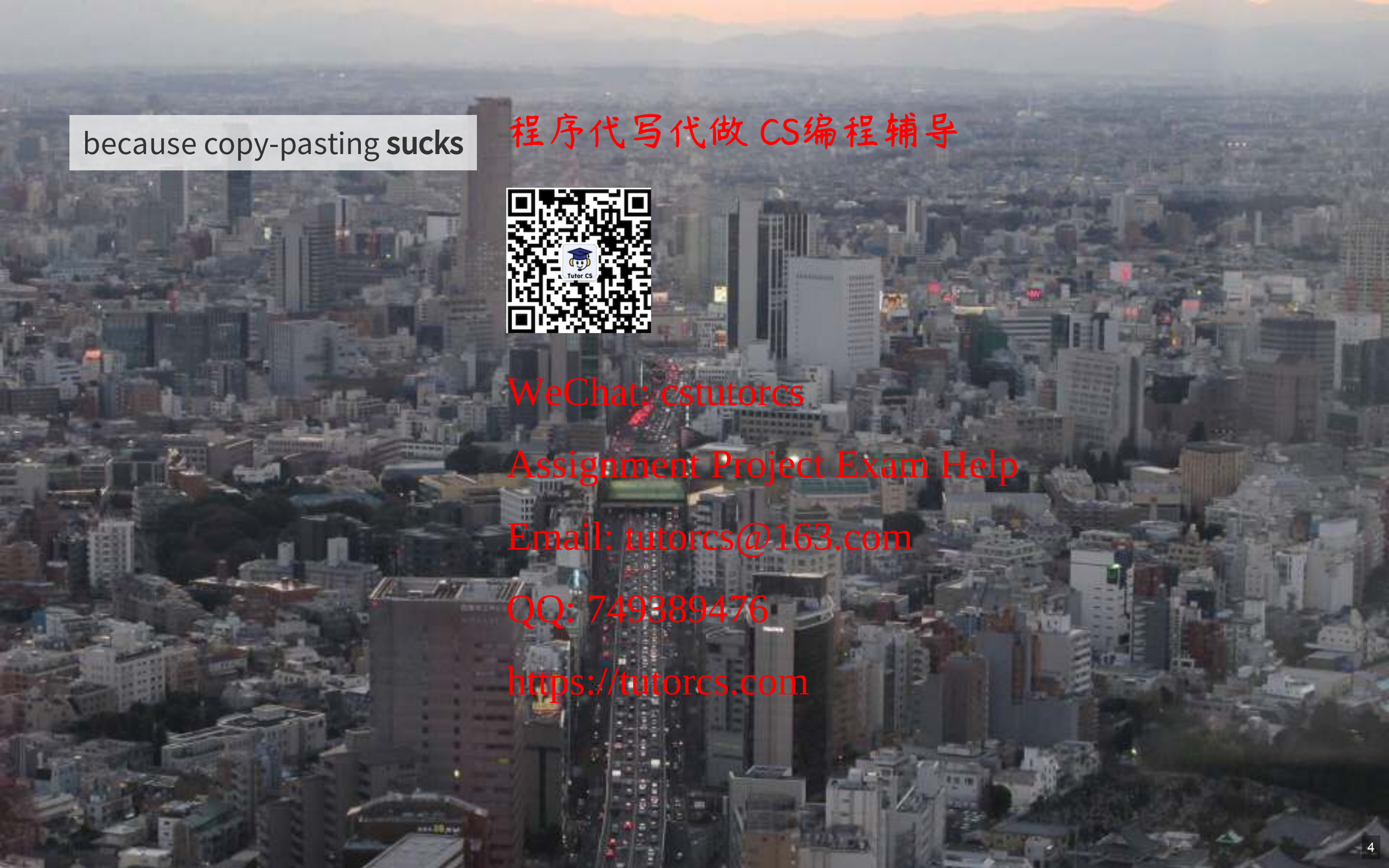
Dr Charles Martin

Semester 1, 2022

Week 5: Functions

Outline

- why functions?
- calling conventions
- the stack


because copy-pasting **sucks**
程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

Function gallery

```
def plus_1(x):
  return x + 1
```

```
public String plusOne(int x) {
  return x + 1;
}
```

```
(define plus-1
  (lambda (x)
    (+ x 1)))
```

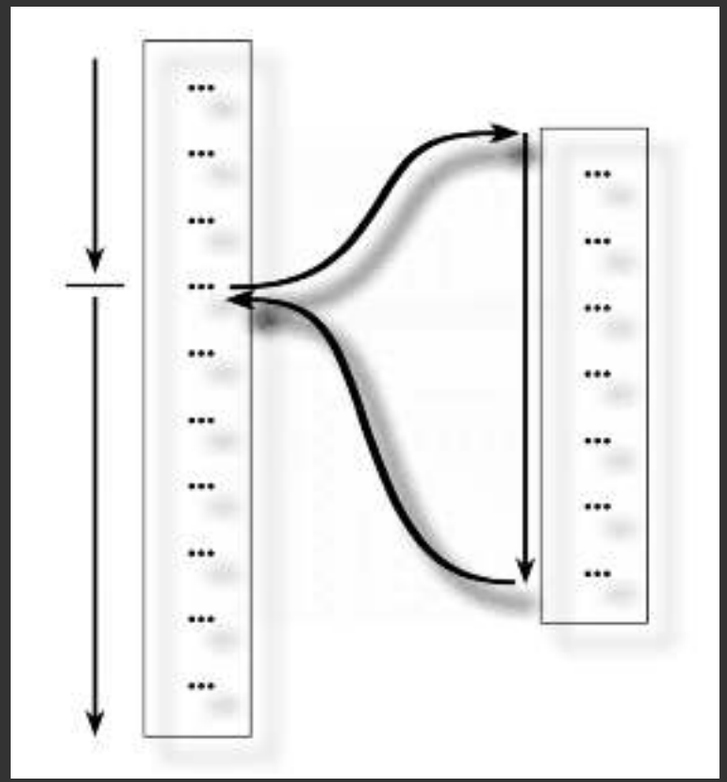
first, some analogies

Good: pipe (input & output)

or “black box”

Better: there, and back again

A function call



talk

Can we do this with branch (`b`)?

Open questions

- how does the program know w me back to?
- how do we pass information (i eters) in?
- how do we get information (i.e., return values) back?
- can we have some “scribble paper”?

note: parameters/arguments - different words for the same thing

Remember Hansel and Gretel?

They *try* and leave a trail of brea behind them so they can find their way back.

`bl` and `bx`

`bl`: branch with link

When the branch **with link** instru is executed, the address of the next instruction (i.e., the one *after* the `bl` instruc ced in a specific register

`lr`: the "link register"

Just like `r15` (`pc`), `r14` also h[illegible]al meaning—it's the *link register*

`bx`: branch and exchange

The `lr` might contain the addre 

struction we want to go back to, but how do we actually **return** there?

The branch and exchange (`bx`) instruction branches not to a static label, but to an address in a register

Don't worry too much about the "exchange" part

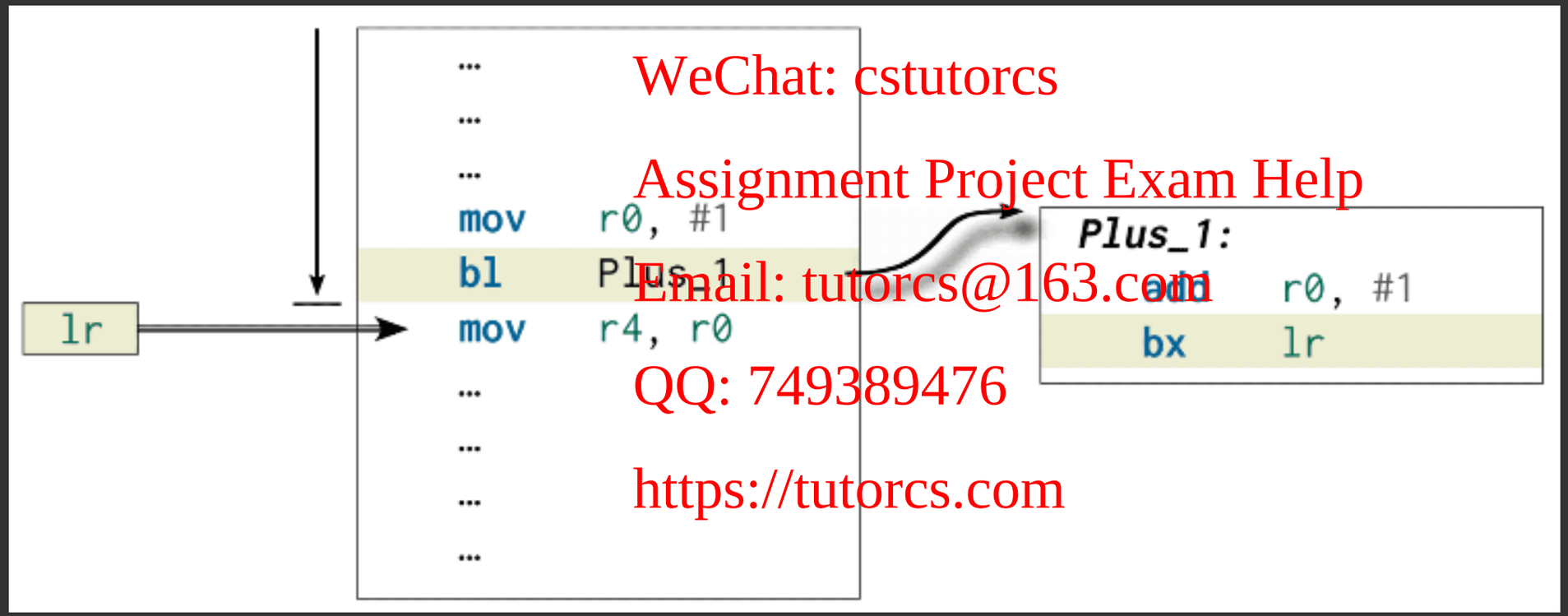
The **"exchange"** part means that we switch the CPU between "ARM" and "Thumb" execution modes.

We **only ever** use Thumb mode.

The way this work is tricky. `bx rN` says "branch to the address located in `rN`".

Code address are aligned to half-words, so the lowest bit of the memory address is always zero. This lowest bit is used by `bx` to change execution mode.

Putting it all together



What about conditional branches?

Both of these new branch instructions (`b`) and (`bx`) can't be used conditionally (e.g. with an `eq` suffix) in the ARMv7-M ISA. 32bit uses

You can get around this with **IT blocks** if you want, or you can use regular conditional branch (e.g., `bgt`)

```
cmp r0, #8
IT eq
bleq add_one
```

Function template

```
@ use the type directi          ell the assembler
@ that fn_name is a fu          (optional)
  .type fn_name, %func


fn_name: @ just a normal label
  @
  @ the body of the function
  @
  bx lr  @ to go back
```

Functions are simple

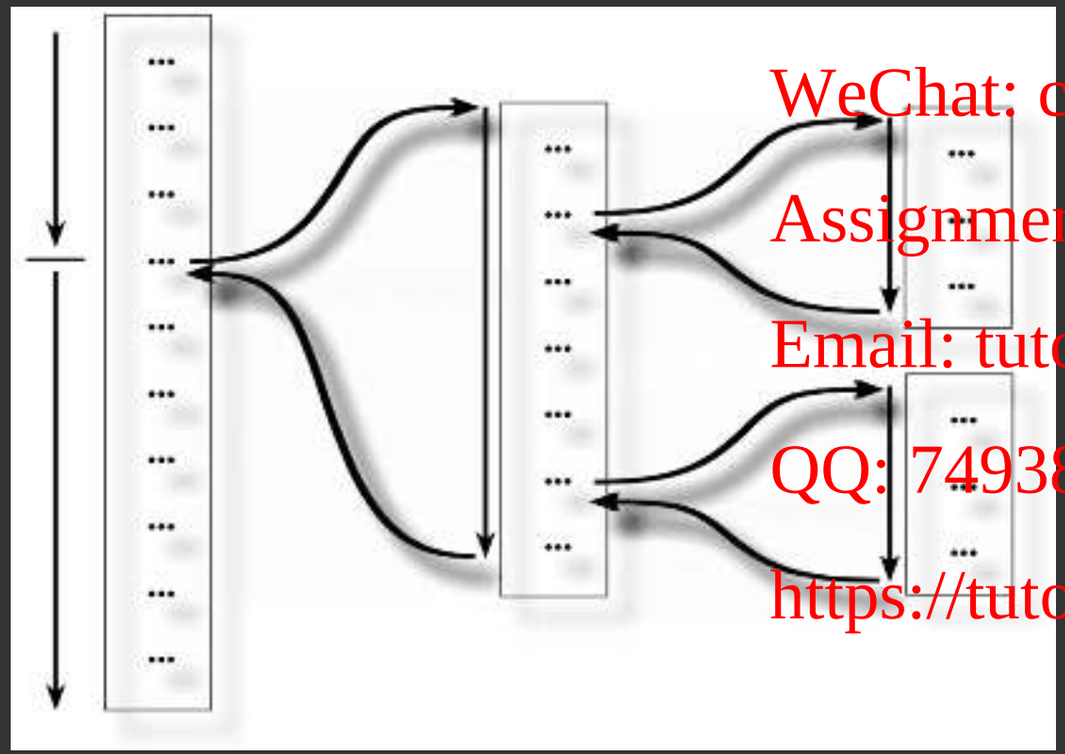
use a `bl <label>` to branch

use a `bx lr` instruction to come back

Analogy Time: RPG quests

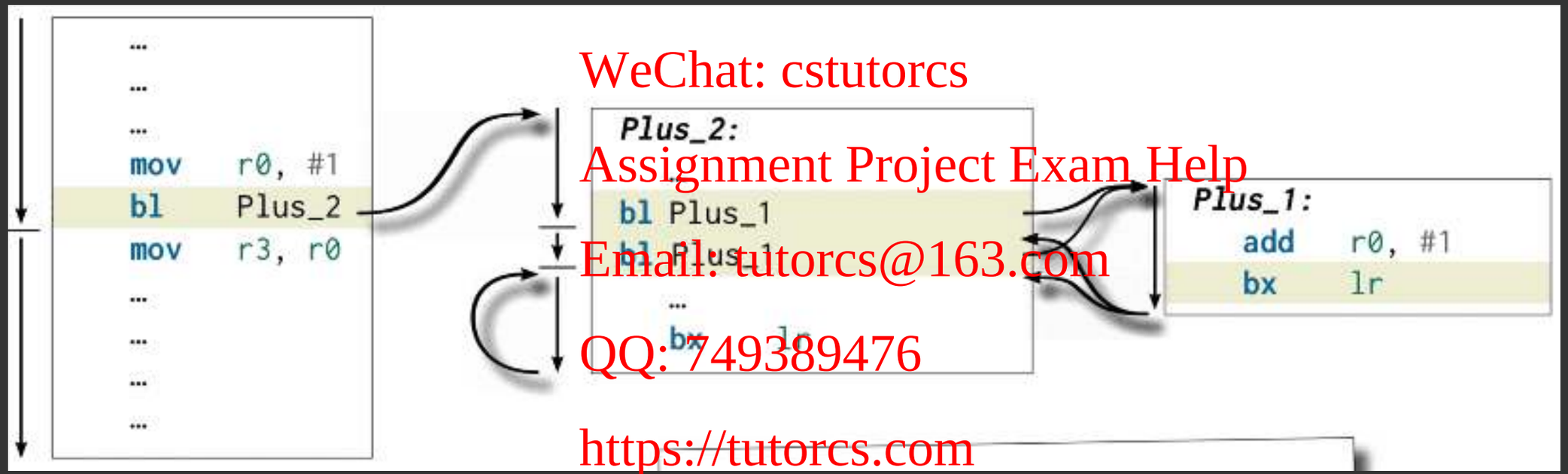
Nested Functions

Nested functions



did the breadcrumbs thing **work** for Hansel & Gretel?

Nested `Plus_1` (broken!)

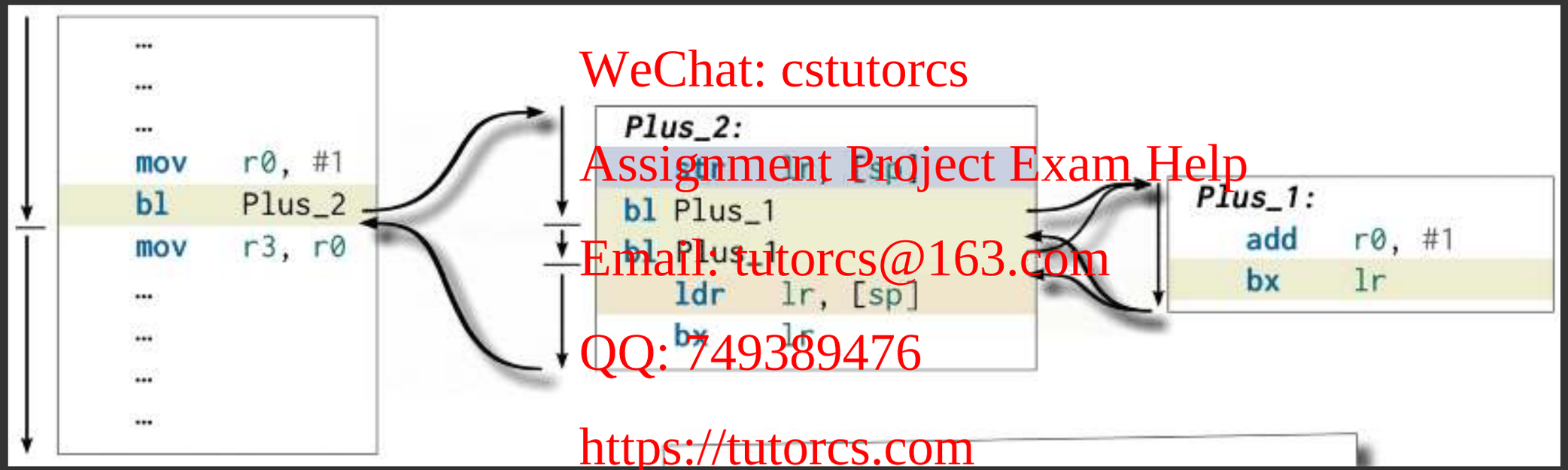


talk

How can we stop the “first” retur getting clobbered?

Sure, store it to memory, but *wh*

Nested `Plus_1` (fixed!)



this will work in this case, but there's still a slight problem with the use of `sp` here—can you spot it?

The stack (sneak peek)

One final new register: the stack `sp` (`sp`, but it's actually `r13`)

By convention: the value of the `sp` is an address in the SRAM region of the address space (like with the `.data section`)

basically, it's memory you can use **to get things done**

We'll return to the stack **later**…

Calling conventions

Open questions

- ~~how does the program know v[illegible]me back to?~~
- how do we pass information (i[illegible]eters) in?

- how do we get information (i.e., return values) back?
- can we have some “scribble paper”?

We need a convention

an agreed-upon plan for where input(s) and where to leave the result

Calling convention definition

This is called a **calling conventio**

It's a contract between the caller (the code which makes the function call with `bl <label>`) and the callee (the code between `<label>` and the `bx lr` instruction)

What does the CC specify?

- where to look for the paramet … he *inputs*)
- where to leave the *outputs*

- which registers to touch, which to leave alone

talk

Which calling convention does th use?

```
int do_all_the_things(int how_many_things){
  // lies! does *none* of the things
  return 0;
}
```


trick question!

There are many possible CCs

It doesn't matter which calling c you use (as we'll see), as long as the caller and the callee use the same conven

CC example

Do these two two `Plus_1` func give the right answer (i.e., `x+1`)? What's the difference?

```
Plus_1:
  add r0, r0, 1
  bx lr
```

```
Plus_1:
  add r5, r2, 1
  bx lr
```

AAPCS

The **ARMv7 Architecture Procedu[illegible]ndard** is the convention we'll (try to) adhere to in programming our microbits.

The full standard is quite detailed, but the general summary is:

- `r0`-`r3` are the parameter and scratch registers
- `r0`-`r1` are also the result registers
- `r4`-`r11` are callee-save registers
- `r12`-`r15` are special registers (`ip`, `sp`, `lr`, `pc`)

What are scratch registers?

`r0`-`r3` are “scratch” registers, 

ns that the caller can freely use them (and not worry about messing anything u

These are also called “caller-save” registers, because if the caller wants to preserve the values in them they need to save them somewhere

Parameters and Return Values

Different ways to get data in/out

Do *these* two two `Plus_1` func  give the right answer (i.e., `x+1`)? What's the difference?

```
@ pass by value
Plus_1:
  add r0, 1
bx lr
```

```
@ pass by reference
Plus_1:
  ldr r3, [sp]
  add r3, 1
  str r3, [sp]
bx lr
```

Pass-by-value vs pass-by-reference

Two different approaches to pas eters and return values in and out of a function.

- **pass by value** makes a “copy” with it without affecting the caller)
- **pass by reference** gives the callee access to the same bits as the caller

pros and cons to both, depends on the nature of the things being passed in and out

in general, data needs to **live** in memory (registers are not for long-term storage)

The stack

Open questions

- ~~how does the program know w[illegible]me back to?~~
- ~~how do we pass information (i[illegible]eters) in?~~
- ~~how do we get information (i.e., return values) back?~~
- can we have some “scribble paper”?

What about local variables?

```
function doStuff(a, b)
  let c = a+b;
  let d = a-b;

  let e = a*b;

  // function body here

}
```

maybe put `c`, `d` and `e` in more registers?

What about local variables?


```
function doArrayStuff(
  let person = {
                  name: "Esmerelda",
                  age: 94,
                  pets: ["rex", "daisy"]
                };
  let junk = new Array(1000);

  // function body here

}
```


there aren't enough registers this time

The stack pointer (revisited)

The stack pointer (`sp`) contains  *address*, and this can be used by functions for various purposes:

- "saving" values in registers which would otherwise be overwritten (e.g. `lr`)
- passing parameters/returning values
- temporary variables, e.g. "scribble paper"

It's called the stack because (in general) it's used like a first-in-last-out (FILO) **stack "data structure"** with two main *operations*: *push* a value on to the stack, and *pop* a value off the stack

but only if you follow the rules

Setting up the stack

Look at the first instruction executed in the startup file:

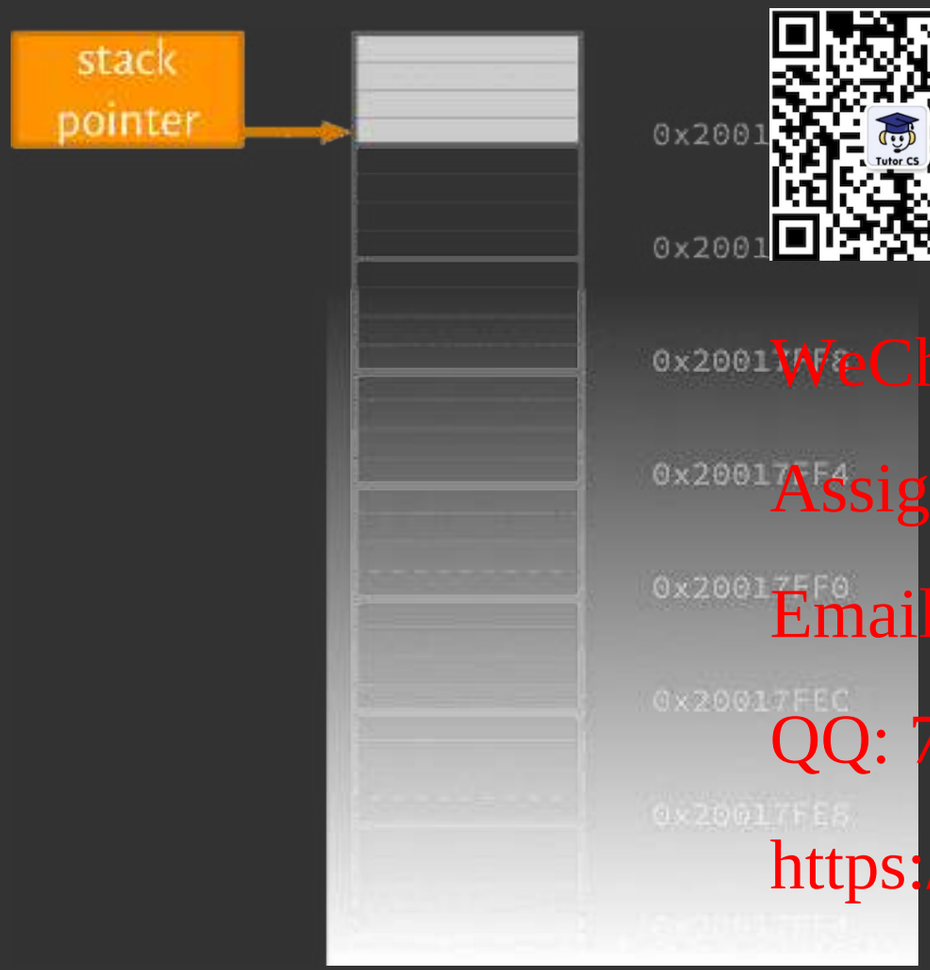
```
ldr    sp, =_estack
```

Loads a value (`_estack`) into `sp` using the **ldr pseudo-instruction**

The exact value of `_estack` comes from the **linker file** (line 34):


```
/* Highest address of the user mode stack */
_estack = 0x20018000;     /* end of RAM */
```

Stack pointer in memory



More about the stack pointer

- the value (remember, it's a memory address) in `sp` changes as your program runs

- `sp` can either point to the last "used" address used (full stack) or the first "unused" one (empty stack)
- you (usually) don't care about the absolute address, because you use it primarily for offset (or relative) addressing
- stack can "grow" up (ascending stack) or down (descending stack)
- in ARM Cortex-M (e.g., your microbit) the convention is to use a **full descending** stack starting at the highest address in the **address space** which points to actual RAM

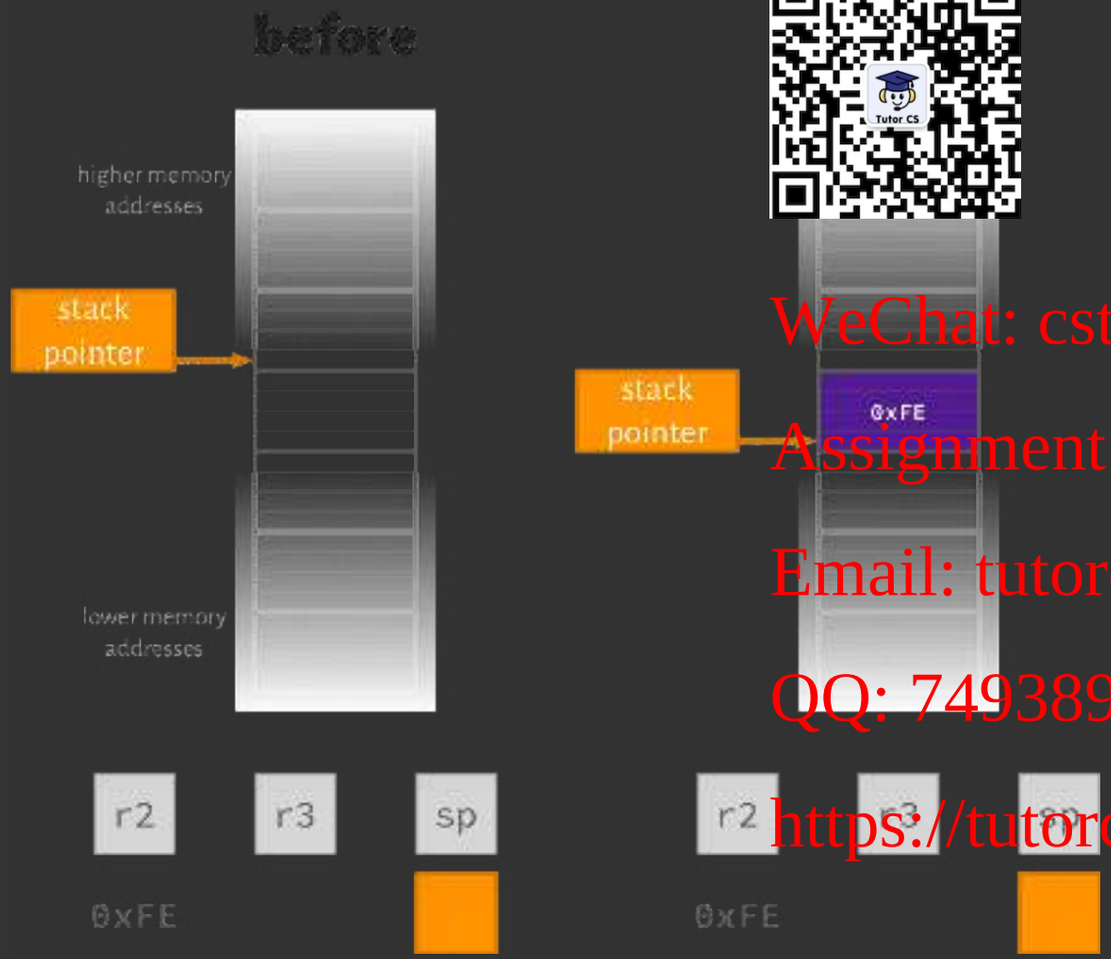
Stack Instructions

Using the stack

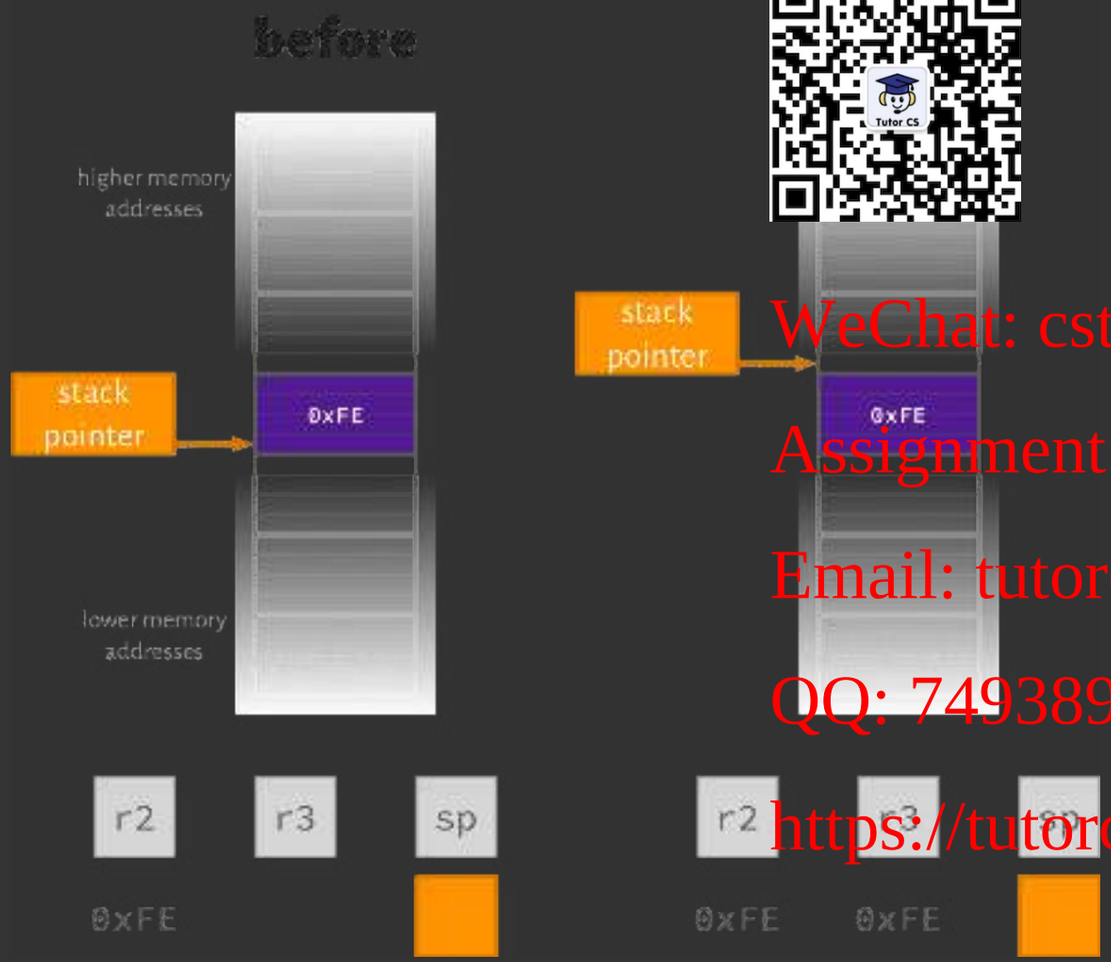
Just use `sp` like any other register containing a memory address:

```
mov r2, 0xfe

@ push the value in r2 onto the stack
str r2, [sp, -4]
sub sp, sp, 4

@ do some stuff here

@ pop the value from the top of the stack into r3
ldr r3, [sp]
add sp, sp, 4
```

Push, illustrated



Pop, illustrated



the “missing” values in the diagrams aren’t empty, just unknown

Offset load and store with write-back

`ldr`/`str` with offset can write the address (base + offset) back to the address register (in this case `r1`) in two different ways:

- pre-offset: update the index register *before* doing the store (or load)

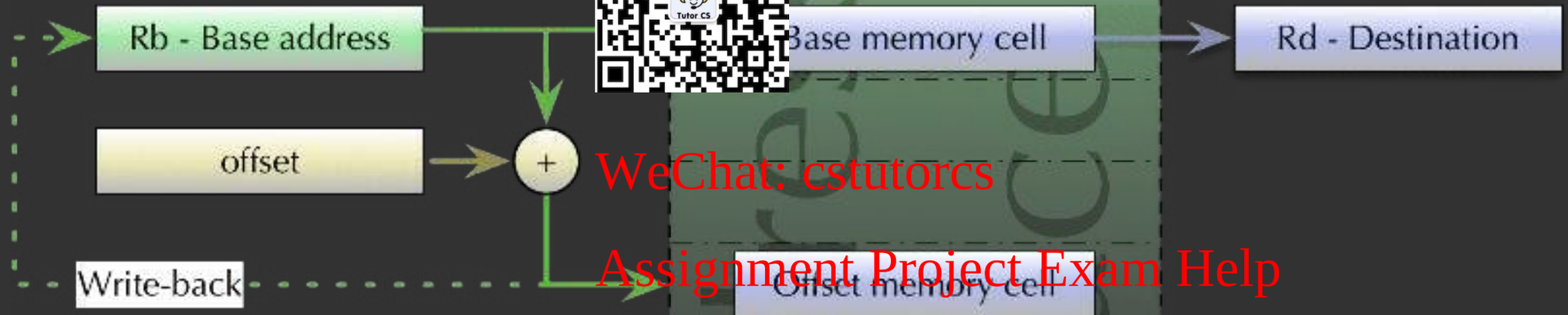
```
@ r1 := r1 + 4
str r0, [r1, 4]! @ note the "!"
```

- post-offset: update the index register *after* doing the load (or store)

```
@ r1 := r1 - 8
ldr r0, [r1], -8 @ no "!" for post-offset
```

Pre-offset addressing

Post-offset addressing



Stack pointer example (again)

Pre/post offset addressing mean[illegible]tructions

```
mov r2, 0xbc

@ push
str r2, [sp, -4]!

@ do stuff...

@pop
ldr r3, [sp], 4
```

`push` and `pop` instructions

Doing this with the stack pointer as the base address is so common that the ISA even has specific `push` and `pop` instructions


```
mov r2, 0xfe

@ gives same result as `str r2, [sp, -4]!`
push {r2}

@ do stuff...

@ gives same result as `ldr r3, [sp], 4`
pop {r3}
```


note that the `sp` base address is implicit

Register list syntax

There was one other difference i[illegible] and `pop` syntax: the brace (`{` `}`) syntax around the register name

Certain instructions take **register lists**—they can apply to multiple registers at once, e.g.

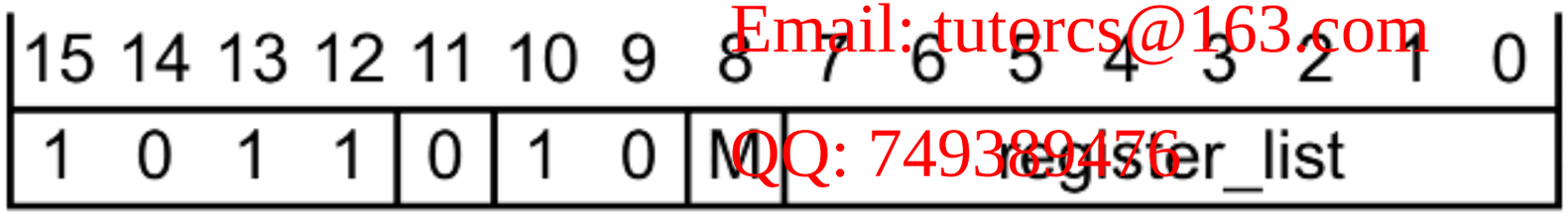

```
@ push r0, r1, r2, r9 to stack, decrement sp by 4*4=16
push {r0-r2,r9}

@ pop 4 words from the stack into r0, r1, r2, r9
pop {r0-r2,r9}
```

`push` instruction encoding

from A7.7.99 of the reference ma

Encoding T1 All versions of the Thumb instruction set.

PUSH<c> <registers>



Load/store multiple

There are also instructions for lo ing multiple words using *any* register as the base register

- `ldmdb` **l**oad **m**ultiple, **d**ecrement **b**efore
- `ldmia` **l**oad **m**ultiple, **i**ncrement **a**fter
- `stmdb` **s**tore **m**ultiple, **d**ecrement **b**efore
- `stmia` **s**tore **m**ultiple, **i**ncrement **a**fter

But if `sp` is the base address, then `push` and `pop` are probably easier to read

be careful about the order!

Further reading

http://www.davespace.co.uk/arm-introduction-to-arm/stack.html

Ben Eater - What is a stack and how does it work? (YouTube)

Plantz - Passing Data in Registers

Plantz - The Stack

Functions and Stack Frames

Function prologue & epilogue

The beginning (or **prologue**) of a function should:

- store (to the stack) `lr` and any other values (e.g. parameters) in registers which will clobbered during the execution of the function (remember the AAPCS)
- make room for any temporary variables by decreasing the stack pointer

Function prologue & epilogue

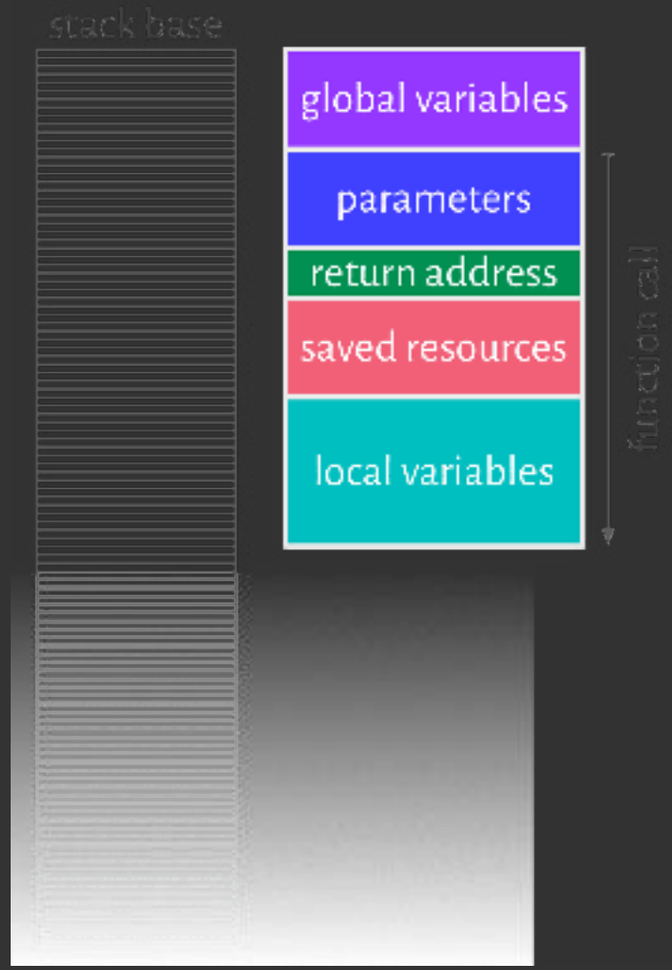
The end (or **epilogue**) of a functi

- re-increment the stack pointe the room for temporary variables
- restore all the stored values back to the registers (e.g. `lr`)
- make sure the return value is left in the right place
- restore the stack state (e.g. put the `sp` back where it was)

Function prologue & epilogue example

```
  .type my_func, %function

@ assume three parameters in

my_func:
  @ prologue
  push {r0-r2} @ sp decreases by 12
  push {lr}    @ sp decreases by 4

  @ body: do stuff, leave "return value" in r3

  @ epilogue
  mov r0, r3 @ leave return value in the right place
  pop {lr} @ sp increases by 4
  add sp, sp, 12  @ balance out the initial "push"
  bx lr
```

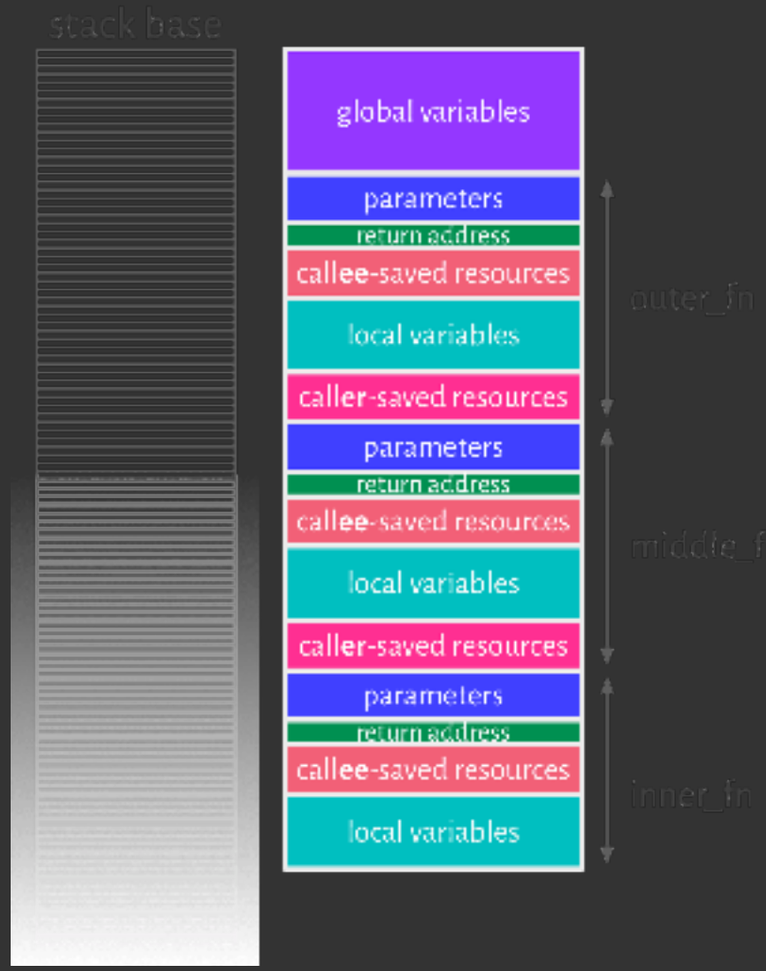
Function stack frame



Nested function calls

```
outer_fn:
  push {r0,lr}
  bl middle_fn

  pop {r0,lr}
  bx lr

middle_fn:
  push {r0,lr}
  bl inner_fn
  pop {r0,lr}
  bx lr

inner_fn:
  @ do inner function stuff
  bx lr
```

Nested stack frames



the `sp` “zippers” up and down as the program executes

There’s lots more to say…

- there’s more you *can* put in yo[illegible]ame (e.g. frame pointer `fp`)
- ARMv7/AAPCS is pretty registe[illegible]her ISA/CCs use the stack more, e.g. for parameter passing and return[illegible])
- an optimizing compiler will almost certainly not generate the code you expect
- recursion is an interesting case (wait till lab 7)

These are all *conventions*

It's the programmer's job to adhere to them: the operating systems programmer, the compiler programmer, the library programmer, the application programmer, …

For bare-metal assembly programming, you're **all** of those

Questions?