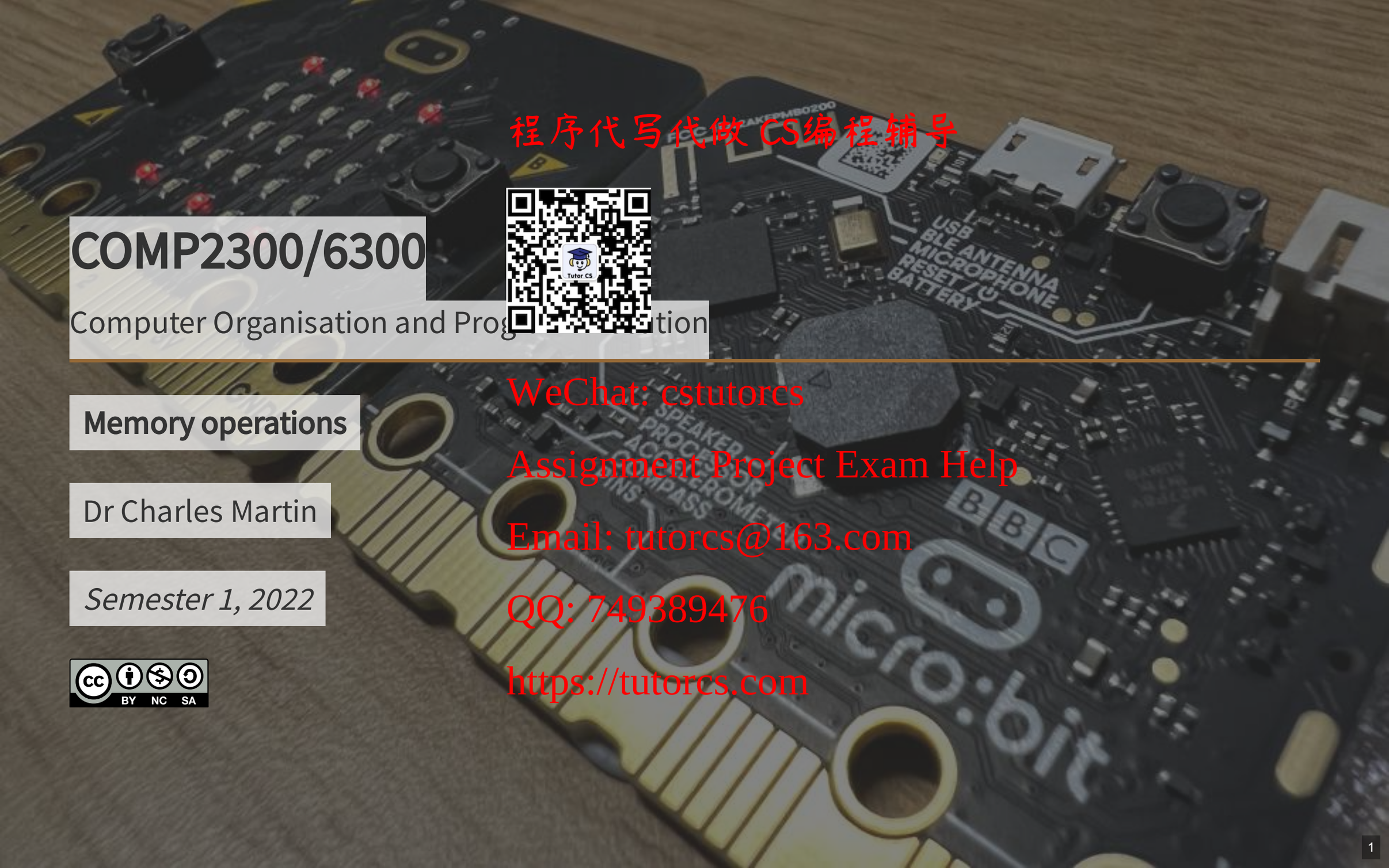


COMP2300/6300

Computer Organisation and Prog[illegible]tion

Memory operations

Dr Charles Martin

Semester 1, 2022

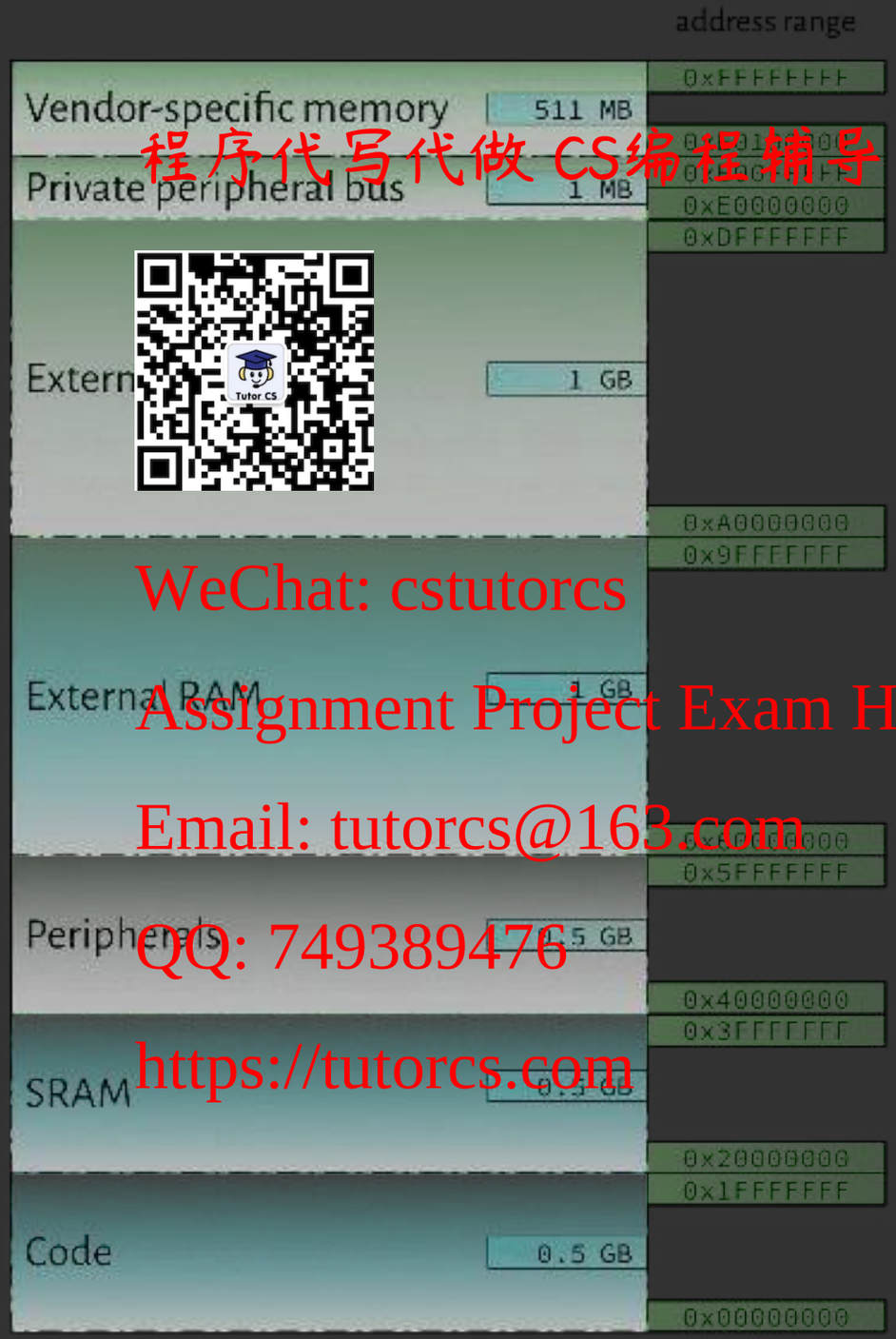
Week 3: Memory Operations

Outline

- addresses
- load/store instructions

- address space
- labels & branching


Memory is how your CPU interacts with the outside world
程序代写代做 CS编程辅导
Tutor CS
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com



but first, a few more instructions

Bitwise instructions

Not all instructions treat the bit p the registers as “numbers”

Some treat them like bit vectors `r`, etc.)

There are even some instructions (e.g. `cmp`, `tst`) which don’t calculate a “result” but they **do** set the flags

Look at the **bit operations** section of your cheat sheet

Example: bitwise clear


```
mov r1, 0xFF
mov r2, 0b10101010
bic r3, r1, r2
```

r1

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

r2

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0

r3

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0

Bit-shifts and rotations

Other instructions will shift (or ro its in the register, and there are lots of different ways to do this!

See the **Shift/Rotate** section of the cheat sheet

Be careful of the difference between **logical** shift and **arithmetic** shift

ARM barrel shifter

Your microbit's CPU actually has hardware (called a "barrel shifter") to perform these shifts as part of another instruction (e.g. an add); that's what `{, <shift>}` means on e.g. the cheat sheet

There are dedicated bit shift instructions (e.g. `lsl`) and other instructions which can take an extra shift argument, e.g.

```
@ some examples
adds r0, r2, r1, lsl 4

mov r3, 4
mov r3, r3, lsr 2
mov r3, r3, lsr 3 @ offset end!
```

Is that everything?

We haven't looked at everything in the cheat sheet

(not even close!)

The cheat sheet doesn't have everything in the reference manual

(not even close!)

But you can do a lot with just the basics, and you can refer to the cheat sheet whenever you need it

talk

- how do you keep track of all th you're using in your program?
- what if you "run out"?

Memory addresses

…but registers can store data

Yes, they can! That's what we've  with the instructions so far (e.g. `mov`, `add`, etc.) manipulating values in regis

Registers are super-convenient for the CPU, because they're inside the CPU itself.

And we can give them all special names—`r0`, `r9`, `lr`, `pc`, etc.

A pet duckling for COMP2300

Random Access Memory

RAM (Random Access Memory) is [illegible] lots data

Perhaps

- character data for a MMORPG
- rgb pixel data from a high-resolution photo
- or the machine code instructions which make up a large program

Current price: ~$100 for 16GB

Types of memory

Three technologies to keep in mi

- **s**tatic RAM (SRAM): uses flip flops (faster, but more expensive and physically larger)—it's used in registers & caches
- **d**ynamic RAM (DRAM): is slow(er), more power efficient, cheaper and physically denser—it's used where you need more capacity (bytes)
- **Flash**: flash, or "non-volatile" memory is used for storage with power turned off (e.g., SSD), it's slower again and more complicated to read/write.

Most computers have all of these types, but the "RAM" in your computer usually refers to DRAM

On your microbit:

- 16 general purpose registers (
- 128 kilobytes of RAM
- 512 kilobytes of flash

(a bit small compared to your laptop/desktop!)

Further reading on memory

Shift and Rotate Instructions

Ben Eater - 8bit memory intro

Essentials of Computer Organization and Architecture, Ch. 6 “Memory”

now we have an addressing problem...

Memory addresses

The solution: refer to each differe 

of memory with a (numerical) address

Each of these addressable units *cell*

Think of it like a giant array in your favourite programming language:

```
byte[] memory = { 80, 65, 54, /* etc. */ };
```

Analogy: street addresses

Byte addressing (addresses in blue)

The byte: the smallest addressable unit

One interesting question: what s[illegible] smallest addressable unit be? In other words, how many bits are in each bucke[illegible]32, 167?

The ARMv7-M ISA uses **8-bit byte** addressing (so do *most* of the systems you'll come across these days)

8 bits == 1 **byte**

Usually, we use a lowercase b to mean bits, and an uppercase B to mean bytes, e.g. 1M**b**ps == 1 million **bits** per second, 3.9 G**B** means 3.9 billion **bytes**

Why 8 bits to a byte?

Again, there's no fundamental r d to be that way

But there's a trade-off between the number of bits you can store and the address granularity (why?)

8 bits provides 256 (2^8) different values, which is enough to store an **ASCII** character

A memory address is just a number

A note about “drawing” memory

It’s a one-dimensional array (i.e. [illegible] a single numerical address for each memory cell)

When “drawing a picture” of memory (like in the earlier slides) sometimes we draw left-to-right (with line wrapping!), sometimes top-to-bottom, sometimes bottom-to-top

It doesn’t matter! The **address is all that matters**

talk

Can you get data in and out of m the instructions we've covered already in the course?

nope.

Load/store instructions

Load instructions

We need a new instruction (well, f them actually)

`ldr` is the the `l` oa `d` `r` egister m

It's on the cheat sheet under **Load & Store**

Loading from memory *into* a register

Any load instruction loads (reads) data from memory and puts them in a register of your choosing

The data in memory is unaffected (it doesn't take the bits "out" of memory, they're still there after the instruction)

```
@ load some data into r0
ldr r0, [r1]
```

What's with the `[r1]`?

Here's some new syntax for your [illegible] using a register name inside square brackets (e.g. `[r1]`)

This means interpret the value in `r1` as a **memory address**, and read the 32-bit word at that memory address into `r0`

remember, memory addresses are just a number

Addresses in immediate values?

Can we specify the memory address in an immediate value?

Yes, but the number of addresses would be limited to what could fit in the instruction encoding (remember, that's what immediates are!)

But more often you'll read the address from a register (so you get the full 2^{32} possible addresses, but you have to get the address into a register before the `ldr` instruction)

`ldr` example

```
mov r1, 0x20000000 @ p[illegible]address in r1
ldr r0, [r1]       @ l[illegible] data into r0
```

What value will be in `r0`?


Let's find out
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

The converter slide

Decimal	0
Hex	0x 0000 0000
Binary	0000 0000 0000 0000 0000 0000 0000 0000

Are these valid memory addresses?

`0x55`

`0x5444666`

`-9`

`0x467ab787e`

Answers: yes, yes, yes, no (too big!)

ARM immediate value encoding

ARM instructions have at most 12 bits for immediate values (depending on encoding), but it can't represent values 0 to 4096 (2^{12})

Instead, it uses an 8-bit immediate with a 4-bit rotation—Alistair McDiarmid has a **really nice blog post** which explains how it works

Storing to memory

Ok, so we *probably* want to put s[illegible] in memory first

The ARMv7-M has a paired `st`ore `r`egister instruction for `ldr`, which takes a value in a register and stores (writes) it to a memory location


```
str r0, [r1]
```


Again, the `[r1]` syntax means “use the value in `r1` as the memory address”—this time the address to store the data to

`str` example

```
mov r0, 42
mov r1, 0x20000000
str r0, [r1]
```

What will the memory at `0x20000000` look like after this?

Endianness

Memory is **byte** addressable, but can fit 4 bytes

So we can load up to 4 bytes into –which order do we “combine” them in?

Why do I need to care?

Because the memory at those a[illegible]ight have been:

- the result of a `str` operation from your microbit
- read from a file created on some other machine
- received over the network

Little-endian is now more common, but it's important to know that other options exist

Further reading on endianness

https://betterexplained.com

https://www.embedded.com

Computerphile: Endianness

Load/store halfwords & bytes

Sometimes, you just want to rea  a halfword (2 bytes), even though you've got a 4 byte register

The instruction set provides additional load/store instructions for this:


```
ldrb @ load byte from register
ldrh @ load halfword from register
strb @ store byte to register
strh @ store halfword to register
```


They work just the same, but they read fewer bytes from memory (and pad the value in the register with zeroes)

talk

Are these byte/halfword versions of the load instructions necessary? Or could you live without them?

Is it an address or a number?

the *address* & the *value at that* are different (but they're both just numbers)

Questions

Memory address space

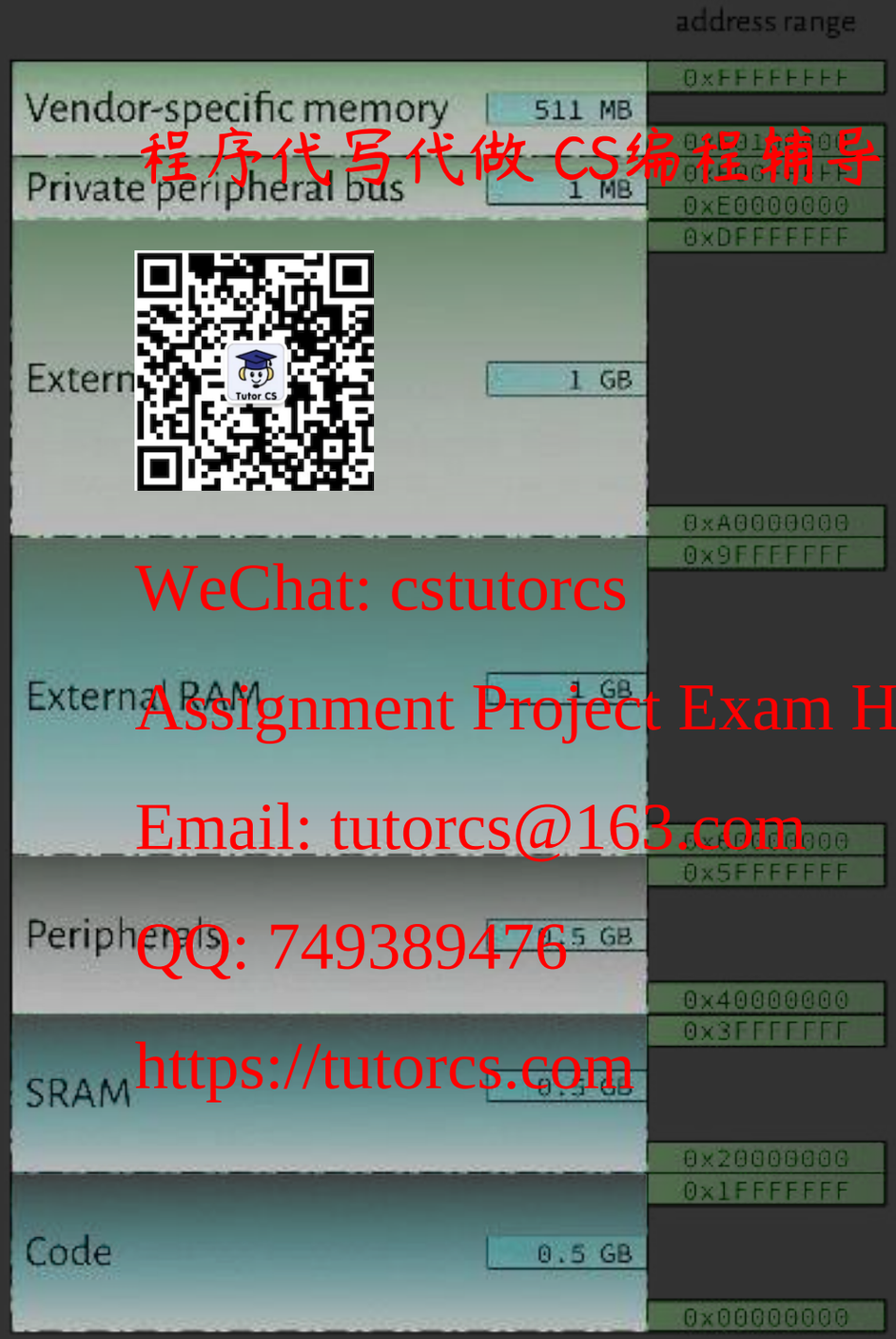
Address space?

Address space

The address space is the set of addresses

So on a machine with 32-bit addresses (like your microbit) that's $2^{32} = 4294967296$ different addresses

So you can address about 4GB of memory (is that a lot?)



A memory address is just a number

Not all memory is the same

You can see from the diagram on the previous slide: the address space is divided into "chunks"

Some parts look like "memory" as we've been talking about so far (e.g., SRAM, External RAM) but some parts don't (e.g. Peripherals)

The load/store architecture

What if everything the CPU did in dealing with the outside world was treated like a load or a store to a memory address?

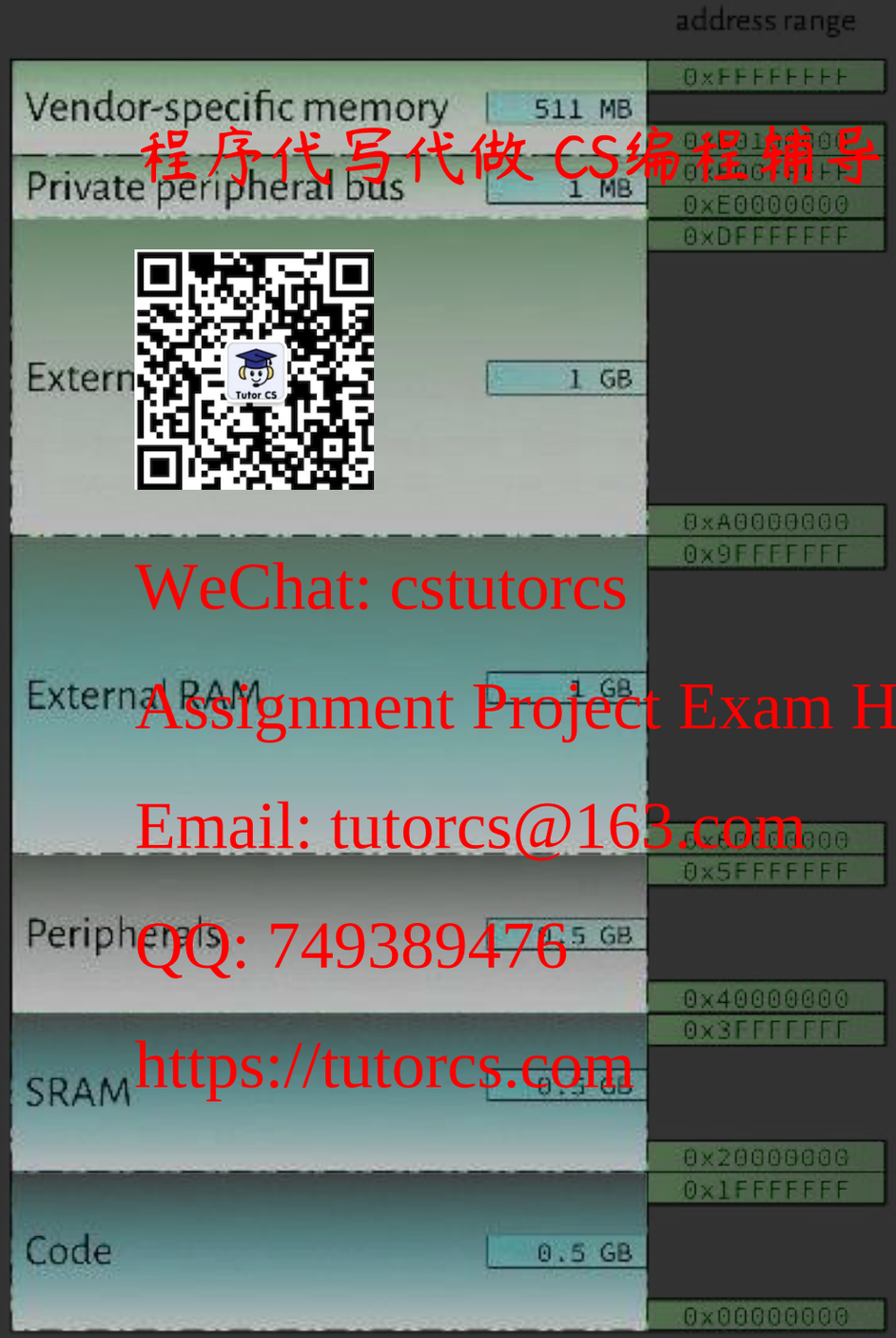
- loading & storing data to RAM
- configuring the various peripherals on the board
- blinking the LEDs
- beeping the speaker

This is the idea behind the load/store architecture, and it's the model your microbit CPU uses

Recap: reading memory diagrams

You'll see "Memory diagrams" (p[illegible]esentations of data in memory, or at least in the Cortex address space)

Look for the addresses—which direction are they ascending/descending?



Not all memory is “data”

Some of it is:

- **r**eadable
- **w**ritable
- e**x**ecutable
- connected to external peripherals (which could still be **r**, **w**, **x** or some combination)

This is a consequence of the load/store model: we *treat* everything like memory, because it makes the CPU simpler

Nordic nRF52833 memory map

The microbit conforms to that Co[illegible]mory map (since it's a Cortex M CPU)

But even within those memory ranges the addresses of specific peripherals (e.g. timers, GPIO, LCD, audio codec) are **unique** to this particular model of microbit

To find out more, you need the **nRF52833 Product Specification**

Code in memory

You probably noticed the **Code** s[illegible]e bottom (i.e. the lower memory addresses) of the address space/memory map

That’s where the encoded **instructions** are: this is sometimes called the *instruction stream*

Each instruction has a memory address

That’s where the **fetch-decode-execute** cycle *fetches* from (based on the **address** in the `pc` register)

Labels and branching

Labels: addresses for humans

All these 32-bit numbers are fine for the computer, but not so good for humans

Labels provide a way to (temporarily) assign a name to a memory address

You've seen labels already—`main` is one! Any word followed by a colon (`:`) in your assembly code is a label

Label gotchas

- only certain characters are allowed in label names
- a label is not an instruction, it doesn't get encoded, it's not in memory

- by default, labels aren't "visible" outside the source file
- the label points to the address of the *next* instruction (whether it's on the same line or a newline)


```
@ these two are the same
label1: mov r0, 5

label1:
  mov r0, 5
```

Branch: select the next instruction to execute

How do we get back to a previou[illegible]ur program?

The answer: change the value in the program counter (`pc`) to "jump back" to an earlier instruction

To do this, use a `b` (branch) instruction, e.g.

```
b 0x80001c8
```

Branch? Why not jump?

but where to branch to?

Labels in the instruction stream

You don't want to have to figure out the address of the instruction "by hand" and move it into the `pc`

So we use labels in the assembly code to keep track of the addresses of specific instructions

And there's a `b` (branch) instruction to tell your microbit to make the jump to that instruction

Branch & labels example


```
main:
   mov r0, 0

@ infinite loop - r0 will overflow eventually
loop:
   add r0, 1
   b loop
```

Branches & labels are best friends :)

Conditional branch


```
b<c> <label>
```

The `<c>` suffix tells us that the branch instruction knows about the **condition flags**, i.e. NZCV

This is **huge**.

Conditional branch examples


```
beq <label> @ branch i
bne <label> @ branch i
bcs <label> @ branch i

bcc <label> @ branch if C = 0
bmi <label> @ branch if N = 1
bpl <label> @ branch if N = 0
bvs <label> @ branch if V = 1
bvc <label> @ branch if V = 0
```


See the back of the **cheat sheet** for the full list.

talk

Does your microbit need to know labels? Where is that information stored?

Labels: just for humans

When you build your program, th ogram:

1. figures out what exact memor s the labels refer to, and

2. swaps all the label names for the 32-bit address values the microbit understands


```
arm-none-eabi-ld -nostdlib -T lib/link.ld --print-memory-usage src/main.o lib/st
Memory region         Used Size  Region Size  %age Used

           FLASH:         800 B       512 KB      0.15%
             RAM:         184 B       124 KB      0.14%

        CODERAM:           0 GB         4 KB      0.00%
```

Your CPU never knows about the labels

The linker replaces them all wit s before you create the binary file (e.g. `program.elf`) which is uplo ur microbit

Memory segmentation

The other thing that the linker do[illegible]ake sure that the various parts of your program get put in the right part of the ad[illegible]e

- make sure secret data isn't readable
- make sure code/instructions isn't writable
- make sure "storage" memory isn't executable

This is a good thing™

It's all controlled by the linker file.

Questions?