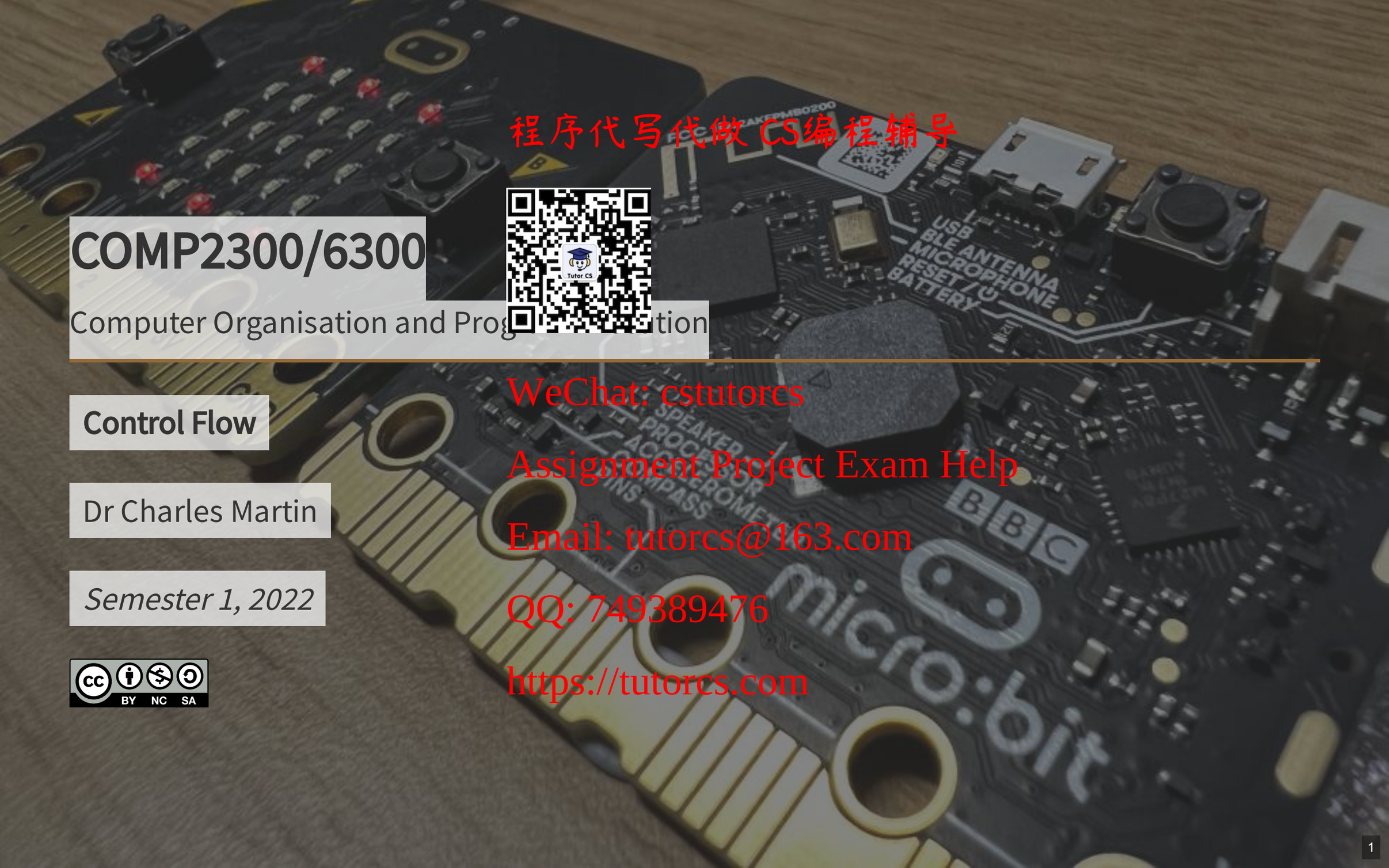


COMP2300/6300

Computer Organisation and Prog[illegible]tion

Control Flow

Dr Charles Martin

Semester 1, 2022

Week 4: Control Flow

Outline

- conditionals
- loops

- some more conditionals?

Conditional Execution

How do we organise our programs?

What are elements of **Structure** ming?

How does that stuff translate into assembly code?

control flow is about conditional execution

condition expressions

1. `x < 13`
2. `x == 4`
3. `x != -3 && y > x`
4. `length(list) < 128`

These all evaluate to a **boolean** **True** or **False** (depending on the value of the variables)

Quiz

How might you express:

- `>` (greater than)
- `==` (equals)
- `!=` (not equals)
- `<=` (less than *or* equal to)

xPSR table

<c>	meaning	flags
eq	equal	Z=1
ne	not equal	Z=0
cs	carry set	C=1
cc	carry clear	C=0
mi	minus/negative	N=1
pl	plus/positive	N=0
vs	overflow set	V=1
vc	overflow clear	V=0
hi	unsigned higher	C=1 ∧ Z=0
ls	unsigned lower or same	C=0 ∨ Z=1
ge	signed greater or equal	N=V
lt	signed less	N≠V
gt	signed greater	Z=0 ∧ N=V
le	signed less or equal	Z=1 ∨ N≠V

Example: `if (x == -24)`


```
@ assume x is in r0
adds r1, r0, 24
beq then
```

In words:

- **if** x + 24 is zero (i.e. if it sets the **Z flag**)
- **then** branch to the `then` label

Example: `if (x > 10)`


```
@ assume x is in r0
subs r1, r0, 10
bgt then
```

In words:

- **if** x - 10 is (signed) greater than 0
- **then** branch to `then`

Alternatives?

assume *x* is in `r0`

```
cmp r0, 10
bgt then
```

```
mov r1, 10
cmp r1, r0
bmi then
```

```
mov r1, 11
cmp r0, r1 @ note the opposite order of r0, r1
bge then
```

are there others?

which is the best?

Conditional expressions in assembly

You need to get to know the diffe tion codes:

- what flags they pay attention t
- what they mean
- how to translate “variable” expressions into the right assembly instruction(s)

It’s hard at first, but you get the hang of it. Practice, practice, practice!

if-else statement gallery (see Wikipedia)


```
if (register1 == regis
    register3 = 1;
} else {
    register3 = 0;
}
```

```
register3 := if register1 = register2 then 1 else 0;
```

```
if register1 == register2:
   register3 = 1
else:
   register3 = 0
```

```
register3 = register1 == register2 ? 1 : 0
```

if-else statement components

Same structure, different syntax.

All of these have:

1. an expression (`if`)
2. a boolean condition (`if`)
3. code for True (`then`)
4. code for False (`else`)

How do these look in assembly?

In assembly

1. check the condition (i.e., set s
2. a **conditional branch** to the “if n(s)
3. the “else” instruction(s), which get executed if the conditional branch *isn’t* taken

if-else with labels, but no code (yet)


```
if:
   @ set flags here
   b<c> then


then:
   @ instruction(s) here

else:
   @ instruction(s) here

rest_of_program:
   @ continue on...
```

talk

What are the problems with this? (a few!)


```
if:
   @ set flags here
   b<c> then

then:
   @ instruction(s) here

else:
   @ instruction(s) here

rest_of_program:
   @ continue on...
```

A better if statement


```
if:
   @ set flags here
   b<c> then

   b else @ this wasn't here before

then:
   @ instruction(s) here
   b rest_of_program

else:
   @ instruction(s) here

rest_of_program:
   @ continue on...
```

The *best* if statement

```
if:
   @ set flags here
   b<c> then


@ else label isn't necessary
else:
   @ instruction(s) here
   b rest_of_program

then:
   @ instruction(s) here

rest_of_program:
   @ continue on...
```

Example: absolute value function


```
if:
  @ x is in r0
  cmp r0, 0
  blt then

else:
  @ don't need to do anything!
  b rest_of_program

then:
  mov r1, -1
  mul r0, r0, r1

rest_of_program:
  @ "result" is in r0
  @ continue on...
```

Label name gotchas

Labels must be unique, so you c[illegible]more than one `then` label in your file

So if you want more than one if s[illegible]n your program, you need

- `if_1`
- `then_1`
- `else_1`
- etc…

Loops

while loop gallery

```
while register1 < 100
    register1 := regis          2;
end loop;

while (register1 < 100) {
    register1 = register1 * register1;
}

while register1 < 100:
    register1 = register1 ** 2
```

while loop components

1. an expression (`if`)
2. a boolean condition (`if`)
3. code inside the loop

Remember that the while loop *checks* the condition and *then runs* (not run then check).

In assembly

1. check the condition (i.e. set so
2. a **conditional branch** to test w ot to “break out” of the loop
3. if branch not taken, execute “loop body” code
4. branch back to step 1

while loop with labels, but no code (yet)


```
begin_while:
  @ set flags here
  b<c> while_loop

  b rest_of_program

while_loop:
  @ loop body
  b begin_while

rest_of_program:
  @ continue on...
```

Example: `while (x != 5)`


```
while(x != 5){
   x = x / 2;
}
```

```
begin_while:
   cmp r0, 5
   bne while_loop
   b rest_of_program

while_loop:
   asr r0, r0, 1
   b begin_while

rest_of_program:
   @ continue on...
```

A better while statement?


```
begin_while:
   cmp r0, 5

   @ "invert" the conditional check
   beq rest_of_program

   asr r0, r0, 1
   b begin_while

rest_of_program:
   @ continue on...
```

Things to note

- we needed to “reverse” the co[illegible]e while loop had a **not** equal (`!=`) test, but the assembly used a branch if equ[illegible]nstruction
- we (again) use a `cmp` instruction to set flags without changing the values in registers
- loop body may contain several assembly instructions
- if *x* is not a multiple of 5, what will happen?

for loop gallery

```
for register1 in 1..10
  register3 := registe          gister1;
end loop


for (register1 = 1; register1 <= 100; register1++) {
  register3 += register1;
}

for register1 in range(1, 101):
  register3 += register1

for register1 in 1..100 do
  register3 += register1;
```


What are the components?

for loop components

1. an **index**
2. a **start** value
3. an **end** value
4. **code** inside the loop

How do these look in assembly?

In assembly

1. check some condition on the “ iable (i.e. set some flags)
2. a **conditional branch** to test w ot to “break out” of the loop
3. if branch not taken, execute “loop body” code (which can use the index variable)
4. increment (or decrement, or whatever) the index variable
5. branch back to step 1

for loop with labels, but no code (yet)


```
begin_for:
  @ init "index" regis                g. i)
loop:

  @ set flags here
  b<c> rest_of_program

  @ loop body

  @ update "index" register (e.g. i++)
  b loop

rest_of_program:
  @ continue on...
```

Example: oddsum

```
// sum all the odd num[illegible]10
int oddsum = 0;
for (int i = 0; i < 10[illegible]{

  if(i % 2 == 1){
    oddsum = oddsum + i;
  }
}
```

Oddsum in asm (worked example)

```
  mov r0, 0 @ oddsum
  mov r1, 0 @ i (index)

for:
  cmp r1, #10 @ expression
  bge exit_for @ boolean test: if i >= 10, exit loop

  @ loop body, need to test if i is odd
  tst r1, #1 @ tests if bit 0 is set i.e., i is odd
  beq not_odd @ test if NOT odd, then exit if
  @ then: is odd
  add r0, r0, r1

not_odd: @ else: not odd
  add r1, #1 @ increment index: i = i + 1
  b for @ go back to top of for loop

exit_for:
```

There are other “looping” structures

- `do while` instead of just `wh`
- iterate over collections (e.g. **C-**

- loops with “early exit” (e.g. `break`, `continue`)
- Wikipedia has a **list**

But in assembly language they all share the basic features we’ve looked at here

You **need** to be confident at writing control structures in assembly! This is core knowledge.

Demo: Looping through an array

Goal: write a program to SHOUT

1. ASCII-encode the string (see ta

2. store it in memory
3. loop over the characters:
   - if it's lowercase, overwrite that memory address with the uppercase version
   - if it's uppercase, leave it alone
4. stop when it reaches the end of the string

IT blocks

Have you noticed that there are lots of instructions on the cheat sheet?

What happens if you try `addeq #1`?

```
Error: thumb conditional instruction
should be in IT block -- `addeq r1,r1,#1'
```


Remember that the Thumb-2 ISA is a compromise between 16bit Thumb and 32bit ARM ISAs. Some things (e.g., conditions on every instruction) just don't fit in 16 bits!

IT blocks

`IT` blocks *cleverly* use 8 bits in the instruction to store a plan for an if-then-else statement that can have up to **four** instructions.

You have to say what the *condition* is (here EQ), and which instructions are going to be "thens" or "elses".

The first instruction following the `IT` instruction is **always** a "then".

```
cmp r0, 42
IT EQ
addeq r1, r1, #1
```

IT blocks

You can add up to three `T`s (thens/elses) after the `IT`, e.g., here's an if-then-else.

```
cmp r0, 42
ITE EQ
addeq r1, r1, #1
subne r1, r1, #1
```


Saves some space if you're only doing a few instructions!

Have a look at A7.3 in the ARMv7-M Architecture Ref Manual or here for more information

Questions?

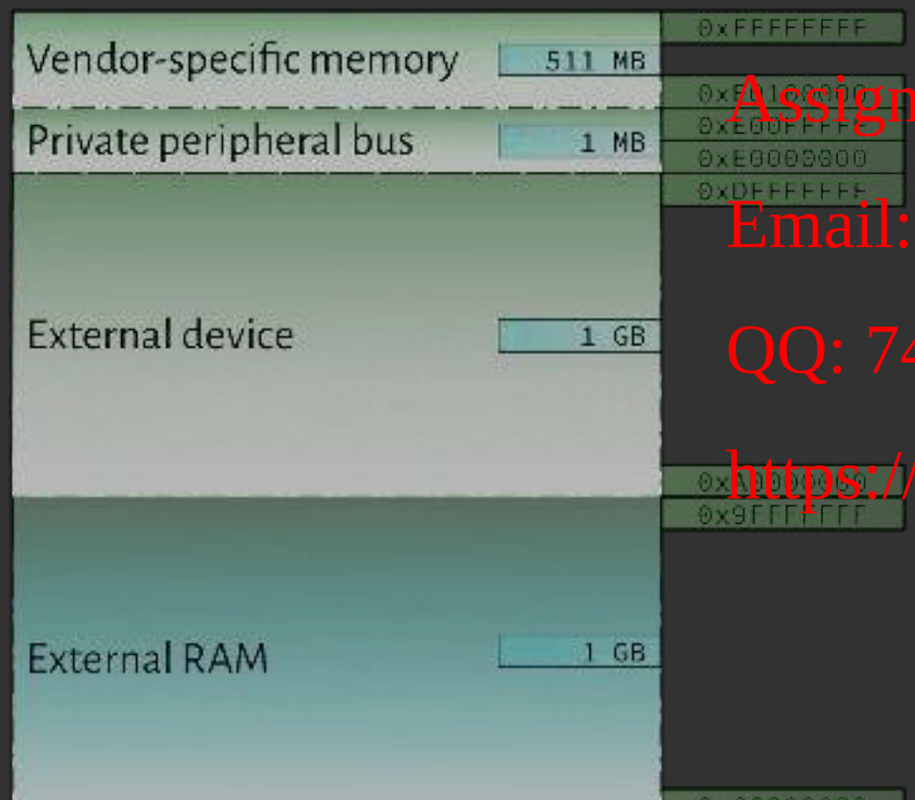
Memory, Value Directives, and Sections

“But *where* in memory does it go?”

Recap: Cortex M4 memory map

As we saw **last week** the lowest (memory addresses) part of the address space is for instructions/code

The SRAM is the next lowest—how do we put stuff in there?



“Value” directives in assembler code

As well as instructions (e.g. `mov`) there are certain assembler **directives** where the assembler doesn’t do any “encoding” but just plonks the value in to the instruction stream as-is

- `.byte` inserts a byte
- `.hword` inserts a halfword (2 bytes/16-bits)
- `.word` inserts a word (4 bytes/32-bits)
- `.ascii` inserts an ASCII encoded sequence of bytes
- `.asciz` inserts an ASCII encoded sequence of bytes followed by a `0`

Multiple value syntax

Each of these directives allows you to put multiple values, one-after-the-other:

```
.byte 1, 5, 0xf2, 0b110100 @ 4 bytes total
.hword 0, 0, 0x1234        @ 3x2=6 bytes
.word 0xdeadbeef, 0x5      @ 2x4=8 bytes
```

Load and store with offset

Recall that `ldr`/`str` require a address to load/store to

```
ldr r0, [r1] @ r1 hold emory address
```

There are also “offset” versions of these instructions:


```
@ address in r1, load value at address+4
ldr r0, [r1, 4]

@ address in r1, store value to address-4
str r0, [r1, -4]
```


it’s all on the **cheat sheet**

talk

When might these “load/store w  versions of the `ldr` / `str` instructions be useful?
Think of as many scenarios as yo

Putting values in the instruction stream

What will this program do? Hint:  ress does the `pc` register "point to"?

```
main:
   ldr r0, [pc, 4]
   b main

   .align 2
beefword:
   .word 0xdeadbeef
```

The `ldr=` pseudo-instruction

Storing little bits of data in the instruction stream is such a useful trick that the assembler provides a special syntax for it (notice the `=` sign before the value):

```
ldr r2, =0xdeadbeef
```


It's called a ***pseudo*-instruction** because the assembler might actually produce a different instruction (e.g. a `mov` instead of an `ldr`)


What instruction is actually used?
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

Why `0xDEADBEEF`?

There are a bunch of numeric li s which are often used in systems programming, e.g. `0xDEADBEEF`, `0x8BADF` on iOS)

Wikipedia has **a list of them** if you're interested

But there's nothing special about them (from the microbit's perspective)

Loading a label address into a register

This is used all the time to load t a **label** (which is just a memory address) into a register (so you can load or store dress)

This instruction loads it's *own address* into `r0` (how meta!)


```
loop:
  ldr r0, =loop
```

What's code and what's data?

We need to be careful about these (code and data), because there's no difference between them from the microbit's point of view

- you can **put instructions in your program using** `.hword`
- you can put data in your program with an assembly instruction (how?)

When you look at *any* assembly code, think:

what will it get encoded to (0s a

where in memory (i.e. at which *addresses*) will those 0s and 1s live when the program is running?

The `.data` section

All of this stuff still only affects w[illegible]n the code section—how do we put stuff in SRAM?

We use the `.data` assembler directive (and a **label** for keeping track of the memory address)


```
    ldr r0, =stuff @ load address of stuff into r0
    ldr r1, [r0]
    @ more code here...

    .data @ from here on, everything goes in the data section
stuff:
    .word 0xdeadbeef
```

talk

What will be in `r0` after the secc  he program has been executed?

```
  ldr r0, =stuff @ load address of stuff into r0
  ldr r1, [r0]
  @ more code here...

  .data
stuff:
  .word 0xdeadbeef
```

What did we just do?

1. put some data in SRAM (near `0x20000000`) using a `.data` section
2. read, modified and wrote back the value

the extra stuff in the startup file (e.g. `LoopCopyDataInit`) *is* important here (try deleting it and re-running the program)

This is necessary because the microbit doesn't let you **write** to any addresses in the code section

Sections in an assembly code file

You can organise the sections in the `.S` file however you like, e.g.,

```
.text
@ anything here is code
@ ...
.data
@ anything here will go in SRAM
@ ...
.text
@ back to code
@ ...
```


`.text` means “code” (it’s also the default section)

the **linker file** makes sure everything gets put into the right place in the memory space

Further reading

Patterson & Hennessy

Chapter 2: “Instructions: Language of the Computer”

Questions?