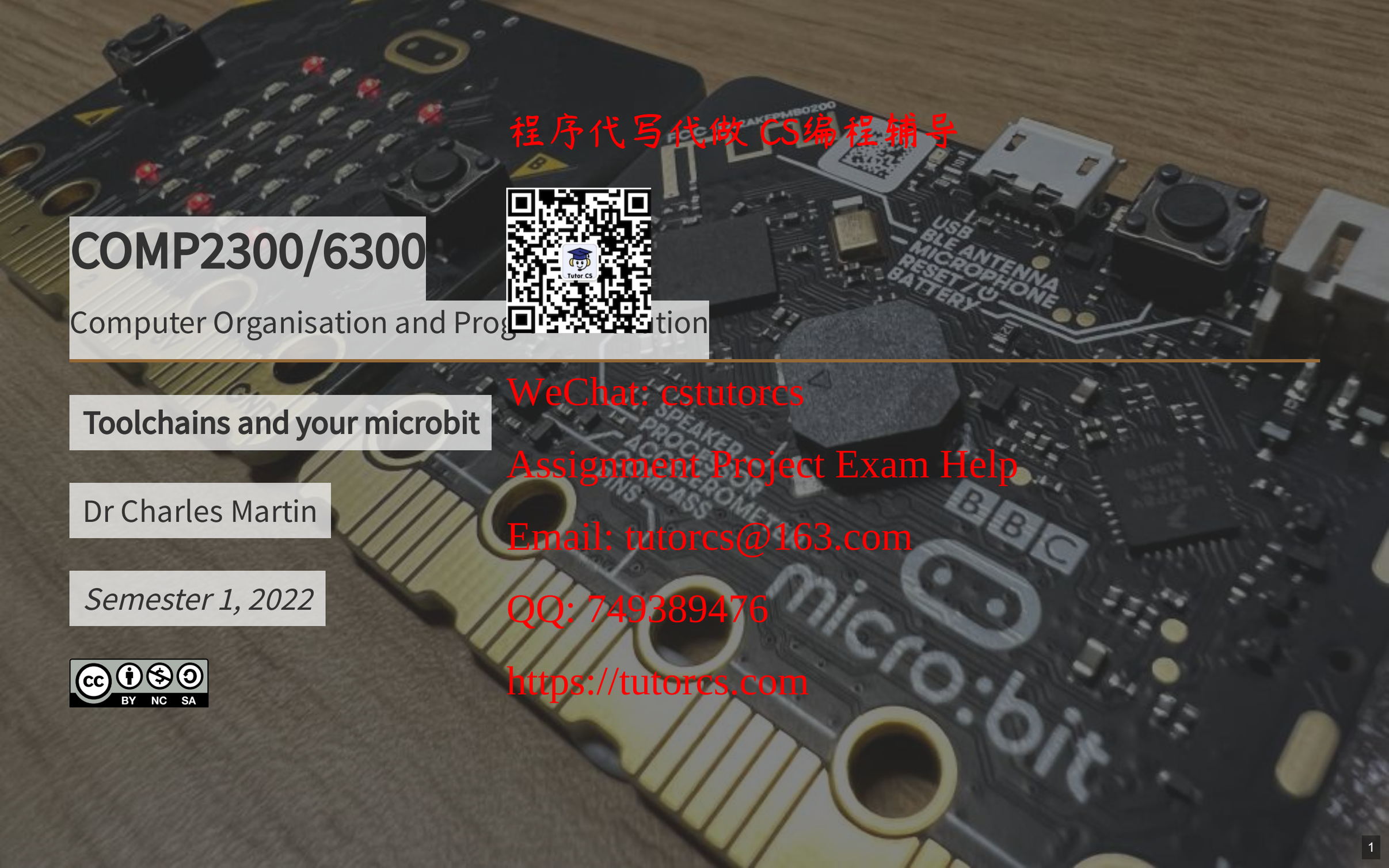

COMP2300/6300

Computer Organisation and Prog[illegible]tion

Toolchains and your microbit

Dr Charles Martin

Semester 1, 2022

Week 6: Assemblers, Compilers, and Toolchains

Admin Time

Course Survey

Assessment results so far

Assignment 1

Course Survey 1

What do you like about the course?

- Tutors/Charles (🥰)
- lectures and labs: engaging, c[illegible]o understand,
- micro:bit: seeing the LED turn on, how computers work at a lower level
- learning to code in assembly
- labs useful for the assignment / lectures and labs support each other
- quiz: helps learning without being stressful

> “ This course is keeping me at uni, loving it so much.

What have you found hard so far?

- memory, overflow flag, carry f…cks, stack, memory
- reference manual (needs a ref…nual)

- **constraints** of assembly
- **distance** between labs and lectures (takes time to absorb lectures)
- change from **“standard programming”** learned before.
- **creativity** needed for the assignment
- assignment seems hard given **deadline** (want more time)
- “limited information online”

everything 😵😵😵

What would you change?

- some time discussing how the
- more engagement with classm p work)

- more details (lectures/labs/assignments)
- more quizzes
- lab solutions
- easier labs / more extensions
- posting demo code
- more live coding
- drop in sessions
- in-person lectures

Topics: Where are we at?

- ✅ Digital Logic
- ✅ ALU Operations

- ✅ Memory Operations
- ✅ Control Flow
- ✅ Functions
- Data Structures
- Interrupts and Asynchronism
- Networks
- Operating Systems
- Architectures

Assessments: Where are we at?

Assessment Item	ghting	Date
✅ Lab Tasks		Each Week
✅ Quiz 1		Week 4
✅ Assignment 1 Pre-submission	1%	Week 5
Assignment 1	20%	Week 6 (~~2022-04-01~~ 2022-04-04 9am)
Mid-Semester Exam	20%	22-04-2022
Assignment 2 Pre-submission	1%	Week 9
Quiz 2	1%	Week 10
Assignment 2 (released in week 8)	20%	Week 11 (2022-05-20)
Final Exam	30%	Exam Period

Assignment 1 Extended: **2022-04-04 09:00 Canberra time**

Assignment 1: What do you need to know?

“Your program must use the LE[illegible] a light show that changes over time”

Need: how to blink any LED on the microbit (lab 4)

Need: how to scan across columns/rows quickly to display an image on the LEDs (lab 5)

Want: excellent knowledge about memory, control flow, and functions (all labs and lectures)

Scanning on the LED screen

- The LED is a matrix, so you **ca** a static image.
- You **can** activate any combina s in an individual row or column

- To display an image, divide it into rows or columns and display each one individually (and quickly) so that it appears to be a single image.

No sample code for this—you have to work on it for yourself. We do have an example though!

Scanning an Arrow, column-by-column

Row and Column Cheatsheet


```
@ ROW 1: P0.21
@ ROW 2: P0.22
@ ROW 3: P0.15

@ ROW 4: P0.24
@ ROW 5: P0.19


@ COL 1: P0.28
@ COL 2: P0.11
@ COL 3: P0.31
@ COL 4: P1.05
@ COL 5: P0.30
```

To light an LED, ROW should be high and COL should be low.

A bit of revision: control flow and functions

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

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...
```

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...
```

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...
```

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
  push {lr} @ maybe other things too
  @
  @ the body of the function
  @
  pop {lr} @ maybe other things
  bx lr  @ to go back
.size fn_name, .-fn_name
```

Stack Frames and the Frame Pointer

One extra detail:

- the frame pointer (`fp` or `r11`) points to the start of the *current stack frame*.
- this means you can access local variables by loading/storing relative to `fp`
- doesn't work to do this relative to `sp` (why?)
- `fp` needs to be pushed at the start of a function and restored at the end.

At the start of a function, you push `fp`, then make space on stack for all local variables.

At the end of a function, you "delete" local vars from stack, and restore `fp`.

Toolchains and Compilers

What is a toolchain?

- a collection of programming t[illegible]ams) that are used one-after-another to create a program
- the **GNU compiler toolchain** is [illegible]ed
- When you "build" your assembly code in VSCode there's actually four GNU tools being used:
  - `make`, `gcc`, `ld`, `objcopy`
- and one extra tool: `openocd`

Where is it on my computer?

When you installed the COMP230[illegible] extension, it automatically installed the GCC ARMv7 toolchain and `openocd`

You can find it in `~/.comp2300/`

(BTW, after the course, you can safely delete `~/.comp2300` from your computer)

If you have a MacOS or Linux computer, `make` was already installed. On Windows you installed `make` in the first lab in your “git bash” folder.

How does it work?

`make`: build automation—runs [illegible] of the build process in sequence

`gcc`: GNU compiler collection—turns assembly commands into CPU instructions (why not use `as`? because `gcc` is smarter)

`ld`: linker—creates a binary data from the CPU instructions and allocates memory locations for variables (knows about the memory layout of the microbit

`objcopy`: “object copy”—copies binary data out of the ELF file produced by `ld` in the exact layout that the microbit can use (removing debug and extra info)

`openocd`: “open on-chip debugger”—helps upload `.bin` file to microbit and allows step-by-step debugging

So how does a compiler work?

How a compiler works (in detail) is out of scope for this course.

But think about the “best if statement”, the “best function template”, the “best while loop”

High level programming constructs can be represented as “templates” in assembly. Compilers can build assembly by using good templates for each high level construct.

(and then they do a lot more stuff as well… but this is good enough for today)

To be a good assembly programmer: need a **very clear understanding** of high-level programming constructs. Then you can optimise from there.

Compiling/Assembling starts with copy-pasting templates

Let’s do some experiments to se works…

Godbolt compiler explorer

https://godbolt.org/: a super-cool interactive resource for exploring stack frames (and code generation in general)

A few tips:

- in the compiler select dropdown, select one of the `ARM gcc` options
- in the *Compiler options…* box, try `-O0` (unoptimised) vs `-O3` (optimised)
- try modifying the C code on the left; see how the asm output on the right changes
- remember the **stack frames**!

Good Godbolt experiments

Try this C++ funnction:

```
int boringfunction() {
    int i = 0;
    int a = 1;
    while (i < 10) {
        a = a + i;
        i = i + 1;
    }
    return a;
}
```


What is the difference between the `-O0` (unoptimised) and `-O3` (optimised) versions?

Demo: Table Branch

How to use the `tbb` and `tbh` s - tricky but powerful and efficient way to do a `case`-style construct.

From the reference:

> “
>
> These instructions cause a PC-relative forward branch using a table of single byte offsets (TBB) or halfword offsets (TBH). Rn provides a pointer to the table, and Rm supplies an index into the table. The branch length is twice the value of the byte (TBB) or the halfword (TBH) returned from the table. The target of the branch table must be in the same execution state.

Table Branch (byte)

```
.type case_example, %function
case_example:
  tbb [pc, r0] @ tbb gets an byte sto                away from pc and branches to pc + 2*byte
branchtable: @ this will be `pc`
  .byte (case0 - branchtable)/2 @ off                to case0 (stored in halfwords)

  .byte (case1 - branchtable)/2 @ offset from pc to case1 (stored in halfwords)
  .byte (case2 - branchtable)/2 @ offset from pc to case2 (stored in halfwords)
  .byte (case3 - branchtable)/2 @ offset from pc to case3 (stored in halfwords)
  .align
case0:
  mov r0, 0x0A
  b exit_case_example
case1:
  mov r0, 0x0B
  b exit_case_example
case2:
  mov r0, 0x0C
  b exit_case_example
case3:
  mov r0, 0x0D
  b exit_case_example
exit_case_example:
  bx lr
.size case_example, .-case_example
```

Demo: Function with Stack Parameters

We saw parameters stored on th *godbolt*, but how can we do that ourselves?

Stack parameter and return value function

```
.type stack_function, %function
stack_function:
  @ take one argument (place on the stack)

  @ argument is at sp + 0
  ldr r0, [sp] @ load in our argument
  push {r4-r11, lr} @ just pushing stuff to the stack for no rea
  lsl r0, r0, #1 @ r0 = r0 * 2
  @ str r0, [sp, 36]
  @ do some other stuff.
  pop {r4-r11, lr} @ return registers and stack state
  str r0, [sp]  @ return one value (same place on the stack)
  bx lr
.size stack_function, .-stack_function
```

Demo: Recursive Fibonacci

Why is this tricky?

Need to calculate `fib(n-1) + fib(n-2)`

That's two recursive function calls, need to look after local registers in between (use the stack, make sure to balance pushes and pops).

(BTW – this is definitely not the most **efficient** implementation…)

Recursive Fibonacci

```
.type fibonacci, %function
fibonacci:
  @ 1 1 2 3 5 8 13....
  @ argument from r0
  @ if r0 is 0 or 1, return 1

  @ else return fibonacci(r0-1) + fibonacci(r0-2)
  push {lr}
  cmp r0, 0 @ check for base case
  beq fib_base_case
  cmp r0, 1 @ check for base case
  beq fib_base_case

fib_recursive_case: @ recursive case: return fibonacci(r0-1) + fibonacci(r0-2)
  push {r0} @ save r0
  sub r0, r0, 1 @ r0 = r0 - 1
  bl fibonacci
  mov r1, r0 @ store output in r1
  pop {r0} @ restore r0
  push {r1} @ store r1 safely
  sub r0, r0, 2 @ r0 = r0 - 2
  bl fibonacci
  pop {r1} @ restore r1
  add r0, r0, r1 @ fibonacci(r0-1) + fibonacci(r0-2)
  b exit_fib

fib_base_case:
  mov r0, 1
```

Further reading

From Zero to `main()`: Demyst ware linker scripts

`.macro` as directive docs

Useful assembler directives and macros for the GNU assembler m.community.arm.com

ARM Stack Frames

Azeria Labs: Functions and the stack

VSCode Function Snippet (thx Benjamin Gray)

Want to make functions quickly? pet!

> “ Put this in the file opened by Command Palette > Preferences: Configure User Snippets > arm (ARM). You can replace the whole file contents with the above. After doing this, in your assembly files you can start typing function, and a popup will appear letting you expand it to the function template.


```
{
    "function": {
        "prefix": "function",
        "body": [
            ".type ${1:name}, %function",
            "${1:name}:",
            "\t${0:nop}",
            "\tbx lr",
            ".size ${1:name}, . - ${1:name}"
        ],
        "description": "Function template"
    },
    "global_function": {
        "prefix": "gfunction",
        "body": [
```

Questions?