The Australian National University
Final Examination – June 2021

Comp2300 & Comp6300
Computer Organisation & Program Execution

Study period:	15 minutes
Time allowed:	3 hours (after study period)
Total marks:	100
Permitted materials:	None

Questions are **not** equally weighted – sizes of answer boxes do **not** necessarily relate to the number of marks given for this question.

All your answers must be written in the boxes provided in this exam form. You can use scrap paper or other documents for working, but only those answers written into the answer boxes of this form will be marked. Do not upload your exam anywhere but the prescribed exam submission system. There is additional space at the end of the booklet in case the answer boxes provided are insufficient. Label any answer you write at the end of the exam form with the number of the question it refers to and note at the question itself, that you provided addition material at the end.

Greater marks will be awarded for answers that are simple, short and concrete than for answers of a sketchy and rambling nature. Marks will be lost for giving information that is irrelevant to a question. Unstructured or not indented code will not be marked.

Student number:

The following are for use by the examiners

Q1 mark	*Q2 mark*	*Q3 mark*	*Q4 mark*	*Q5 mark*	*Q6 mark*	*Total mark*

1. [8 marks] Logic & Instructions

(a) [4 marks] Your ARM CPU produces four main flag outputs for specific instructions:

N - Negative
Z - Zero
C - Carry
V - Overflow

Provide the smal[illegible]M instruction which will set each of these flags individually while clearing all other flags. So you should produce the flag sequence:

N - Negative	Z - Zero	C - Carry	V - Overflow
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

To make your life easier, assume that the following instructions were executed right before your instructions. Hint: the overflow flag is the tricky one, which requires a minimum of two ARM instructions.


```
movs    r0, #0x00000001
movs    r1, #0xFFFFFFFF
movs    r2, #0x7FFFFFFF
movs    r3, #0x80000000
```


(b) [4 marks] Explain why your ARM CPU does not provide an arithmetic-shift-left instruction.

2. [24 marks] Functions

(a) [24 marks] Write a function in ARM assembly code, which doubles the individual values in a given array, which holds (word-sized) natural numbers. Your function will need to be able t[illegible] sized arrays. Four different versions of this function are required. The [illegible]uage fragments below are for illustration only, as you can answer t[illegible]ut being able to read any of those.

(i) [6 marks] Wri[illegible]ion of this function which doubles the values inside the original array[illegible]e function is side-effecting, but does not have any direct return value (hence it is called a `procedure` or `void function`). Your code will resemble the output of a compiler for any of the following higher languages fragments:

Algol-style:

```
type Naturals is array (Positive range <>) of Natural;
procedure Double_In_Place (Numbers : access Naturals) is
begin
   for E of Numbers.all loop
      E := 2 * E;
   end loop;
end Double_In_Place;
```


C-style:

```
void double_in_place (unsigned int array [], unsigned int length) {
   unsigned int i;
   for (i = 0; i + 1 <= length; i++) {
      array [i] = 2 * array [i];
   }
}
```

(ii) [6 marks] Write an iterative version of this function which has no side-effects, thus returns a new array where each element is double the value of the corresponding element in the original array. Your code will resemble the output of a compiler for the following higher language fragment (note that neither this, nor any of the following versions can [illegible] style languages, unless heap memory allocation is available):

Algol-style:

```
type Naturals [illegible] ve range <>) of Natural;

function Double_Iterative (Numbers : Naturals) return Naturals is

   Doubled_Numbers : Naturals (Numbers'Range);

begin
   for I in Numbers'Range loop
      Doubled_Numbers (I) := 2 * Numbers (I);
   end loop;
   return Doubled_Numbers;
end Double_Iterative;
```

(iii) [6 marks] Write a recursive version of this function, which also has no side effects. Your code will resemble the output of a compiler for any of the following higher languages fragments. Warning: this is a hard question.

Algol-style:

```
type Naturals [illegible]ve range <>) of Natural;

function Doub[illegible]bers : Naturals) return Naturals is

  (if Numbers [illegible]
   then Numbe[illegible]
   else 2 * Numbers (Numbers'First)
        & Double_Recursive (Numbers (Numbers'First + 1 .. Numbers'Last)));
```

Functional-style:


```
double_recursive :: (Num a) => [a] -> [a]

double_recursive list = case list of
  []     -> []
  x: xs -> 2 * x : double_recursive xs
```

(iv) [6 marks] Write a more flexible version of this function, which can be used to apply any operation on the individual array elements. This version is also to be without side-effects, but you can chose an iterative or recursive form. Your code will resemble the output of a compiler for any of the following higher languages fragments:

Algol-style:

```
type Naturals [illegible]e range <>) of Natural;

function Map_I[illegible] : Naturals;
  Operation : a[illegible]X : Natural) return Natural) return Naturals is

    Doubled_Numbers : Naturals (Numbers'Range);

 begin
    for I in Numbers'Range loop
       Doubled_Numbers (I) := Operation (Numbers (I));
    end loop;
    return Doubled_Numbers;
 end Map_Iterative;
```

Functional-style:

```
map_recursive :: (a -> b) -> [a] -> [b]

map_recursive op list = case list of
   []    -> []
   x: xs -> op x: map_recursive op xs
```

3. [15 marks] Decompile & Optimize

(a) [5 marks] Write a program in any programming language of your choice which might have compiled down to the below ARM assembly code.

```
func:
    stmdb   sp!,
    add     fp,
    sub     sp,
    str     r0,
    cbz     r0,
    sub     r0, #1
    bl      func
    ldr     r1, [fp, #-8]
    add     r0, r1
end_func:
    add     sp, #4
    ldmia   sp!, {fp, lr}
    bx      lr
```


(b) [10 marks] Optimize the above ARM code as much as you can for performance, while keeping the exact same response to "`bl func`" (everything else can change).

4. [17 marks] Writing programs

(a) [5 marks] Your embedded device has been equipped with a key device, which provides secret key data via a single address in memory (ADR_KEY). You can load this address in the usual way int[illegible]

```
ldr     r0, =A[illegible]
```

On reading data [illegible] you will be provided with unique key data. The next time you wil[illegible] this address, new key data will be provided. Similarly your device [illegible]ipped with a laser sensor which will provide fresh readings every time you read data from memory address `ADR_LASER`.

Your mission is top secret and the raw data from this laser cannot fall into the wrong hands. Consequently you will need to encrypt the raw data, before you can store it into local memory. Do this by a bit-wise ex-or operation between the key data and the raw laser data. The key data will disappear from your device upon usage (the only other, known copy of the key data rests inside the ruler-of-the-universe's toaster). The key data as well as the raw laser data is sensitive and should only be held inside your device for processing for the shortest time possible and should never be stored in memory. Store the encrypted laser data at the address which is stored at the address `ADR_SECRET_ADR`. You will need to increment this address, after each usage, so that all data can be recorded (don't worry about memory overflows).

Write an ARM assembly function `Laser_Handler` which can be used as an interrupt handler responding to the laser indicating that new data is available. Do not worry about how to connect the interrupt handler to the specific interrupt: this has already been done for you.

(b) [12 marks] Write an ARM assembly function, which returns the maximum value of an unknown-sized array of integer numbers, which it takes as a parameter. If the array is empty your function should return -2^{31}.

(i) [6 marks] Wri[illegible]ion. Which parameter modes did you chose?

(ii) [6 marks] Write an iterative version. Which parameter modes did you chose?

5. [20 marks] Asynchronous Programming

(a) [10 marks] Implement the following concurrent code in ARM assembly, such that the final value of Counter will be 0, after both tasks completed. For performance reasons, keep the duratio[illegible] code sections to a minimum.

The following Al[illegible]ode (as used in lectures) shows: one task increments a global C[illegible] 100, while another task decrements the same global Counter variable[illegible]

```
Counter : Integer := 0;

task Up is

begin
   Counter := Counter + 100;
end Up;

task Down is

begin
   for I in 1 .. 10 loop
      Counter := Counter - 10;
   end loop;
end Down;
```


Assume the following global memory declaration:


```
Counter:
   .word      0x00000000
```

(b) [5 marks] Can you use a semaphore-wait operation inside of an interrupt handler? What would be the possible uses or dangers? Will your answer be different for a single-core and a multi-core CPU? Explain as precisely as you can.

(c) [5 marks] In a press release on an unnamed manufacturer, it was revealed that a malfunction of a fast moving device was caused by a stack-overflow by interrupts. Could this be true? What would you recommend for this manufacturer to introduce in their design processes? Explain as precisely as you can.

6. [16 marks] Operating Systems, Networks & Architectures

(a) [3 marks] What are the minimal hardware requirements which enable you to write a multi-tasking operating system with a round robin scheduler? Give precise reasons for your answers.

(b) [3 marks] In a computer network, how can you send user data to devices which are not directly connected to your sending device? Which administrative data (in terms of OSI layers) will need to be looked at along the way, before your user data reaches the target device. Give precise reasons for your answers.

(c) [4 marks] The following function is an attempt of one of your fellow students to write a fast sum-function adding up all values inside of an array. You are however more educated than your fellow student and know about pipeline hazards. Optimize the program below such that pipeline hazards are avoided and the performance will increase (obviously [illegible] e logical correctness). Re-write the function inside the answer box a [illegible] f your changes which pipeline hazard you address.

```
@ r0: Address of [illegible]
@ r1: Size of ar[illegible]
@ return the sum [illegible]

Fast_Sum:
   mov    r2, r0
   add    r3, r2, r1, lsl #2
   mov    r0, #0
Fast_Sum_Loop:
   ldr    r1, [r2], #4
   add    r0, r1[illegible]
   cmp    r2, r3
   blt    Fast_Sum_Loop
   bx     lr
```

(d) [6 marks] If you consider the following code fragment, which hardware support would you suggest to run this program correctly and at high performance. Relate each suggested hardware architecture to a concrete aspect of the code.

The code defines [illegible]y with 10,000 floating point values (all initialized to 1.0), as well as 10[illegible] double all elements in this array.

```
Data : array [illegible]Float := (1 .. 10_000 => 1.0);

task type Wor[illegible]
task body Wor[illegible]

begin
   for I in Data'Range loop
     Data (I) := 2.0 * Data (I);
   end loop;
end Worker;

Workers : array (1 .. 100) of Worker;
```

continuation of answer to question part

continuation of answer to question part

continuation of answer to question ☐ *part* ☐

continuation of answer to question ☐ *part* ☐