

# UNIVERSITY OF VICTORIA

## FINAL EXAM DECEMBER 2011

<b>Course Name &amp; No.:</b>	<b>C SC 230 - Introduction to Computer Architecture and Assembly Language</b>
<b>Section(s):</b>	A01 and A02
<b>CRN:</b>	15729 and 15730
<b>Instructor:</b>	M. Serra
<b>Duration:</b>	3 hours
<b>STUDENT NUMBER:</b>	

Question No.	Value	Mark	Question No.	Value	Mark
1	9		10	12	
2	6		11	8	
3	10		12	15	
4	5		13	15	
5	14		14	2	
6	8		15	3	
7	8		16	10	
8	15		17	16	
9	8		<b>TOTAL</b>	<b>164</b>	

### INSTRUCTIONS:

1. Students must count the number of pages and report any discrepancy immediately to the Invigilator.
2. This examination paper has a total of 18 pages including this cover page, and 1 page of handout.
3. No aids are permitted. However, a handout describing the ARM instruction set is provided.
4. The marks assigned to each question are shown within square brackets. Partial marks are available for all questions.
5. This exam is to be answered on the paper.
6. It is strongly recommended that you read the entire exam through from beginning to end before beginning to answer the questions.

**Question 1. [9]** Fill in the table below with the appropriate information about the instructions. The column "Is the Bus used?" refers ONLY to the execution phase, not the fetch phase.

Instruction	Addressing Modes of All Operands	Is the Bus used?	What it does (be precise)
LDR r1, =0xFF8A			
STR r1, [r2], #4			
MOV r1, r2, LSL #4			

**Question 2. [6]** Suppose you have the following:

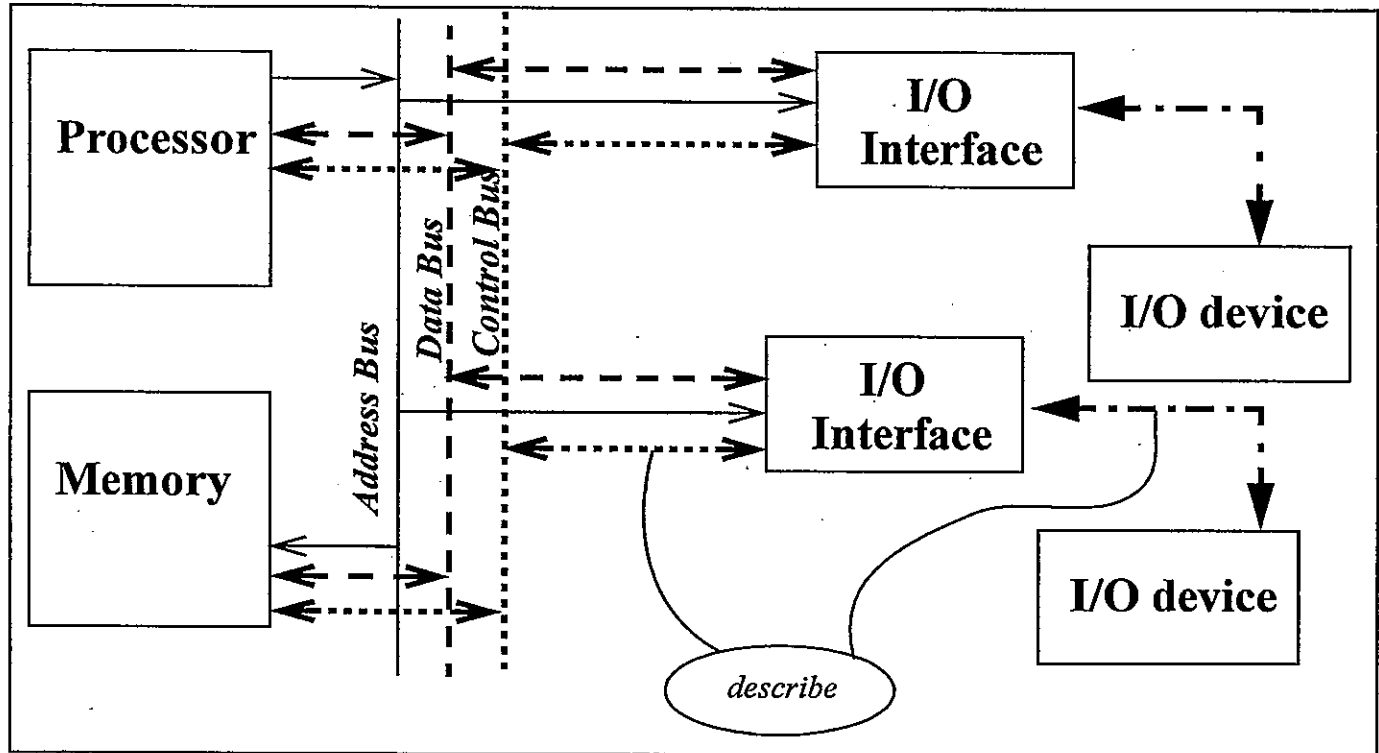
- [i] a compiler front-end which understands the programming language C
- [ii] a compiler front-end which understands the programming language Ada
- [iii] a compiler back-end which understands the architecture of the Intel x86 processor
- [iv] a compiler back-end which understands the architecture of the PowerPC processor

The intermediate format produced by the front-ends is the input used by the back-ends. Indicate which of the following language translations you can do using the above components:

Translation	YES/NO
C code → Ada code	
C code → PowerPC code	
Ada code → ARM code	
PowerPC code → Intel x86 code	
FORTTRAN code → Intel x86 code	
Ada code → PowerPC code	

**Question 3. [10] (a) [2]** Describe the main function of a DMA.

- (b) [4] You have seen this diagram before, showing some possible interconnections between peripheral devices and a processor using interfaces. List, in point form, the possible protocols for communications between (i) the processor and interfaces and (ii) the interfaces and the devices. You do not need to describe the details.



(i)

(ii)

- (c) [4] A handshake type protocol is most often used to communicate between an interface and a peripheral. Summarize the main steps.

**Question 4. [5]**

- (a) [3] Can you state at least 3 main characteristics which differentiate between static and dynamic RAM? Explain how they apply to each.

- (b) [2] Where is static RAM used mostly and where is dynamic RAM used mostly in a system?

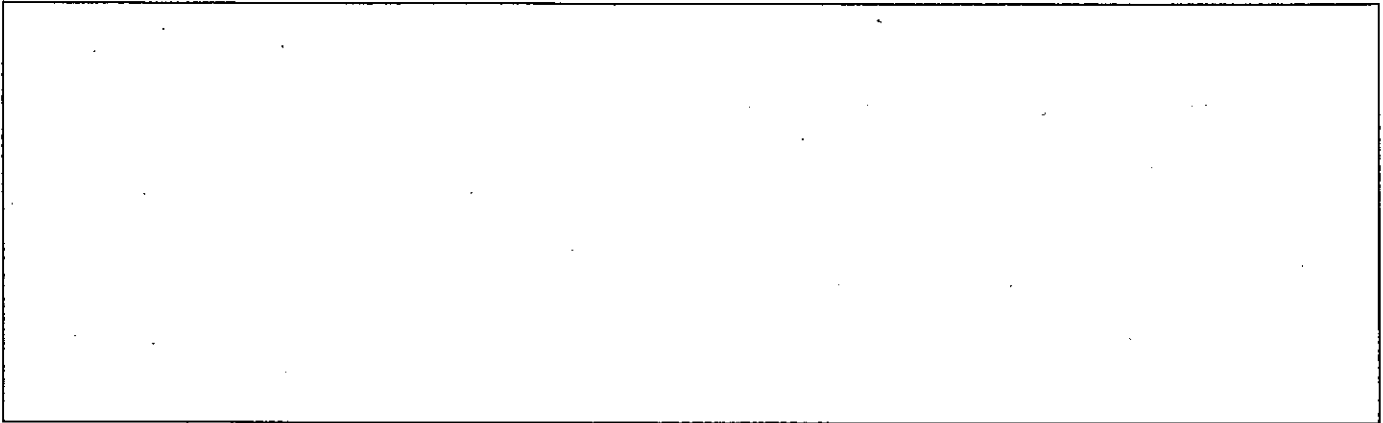
**Question 5. [14]** Program execution time,  $T$ , can be defined as:  $T = \frac{N \times S}{R}$  where

- $T$  is the total elapsed time,
- $N$  is the number of machine language instructions used during the execution (not necessarily the number of machine instructions in the object code),
- $S$  is the number of basic steps needed to execute one machine instruction (where each basic step is assumed to take 1 clock cycle),
- $R$  is the clock rate.

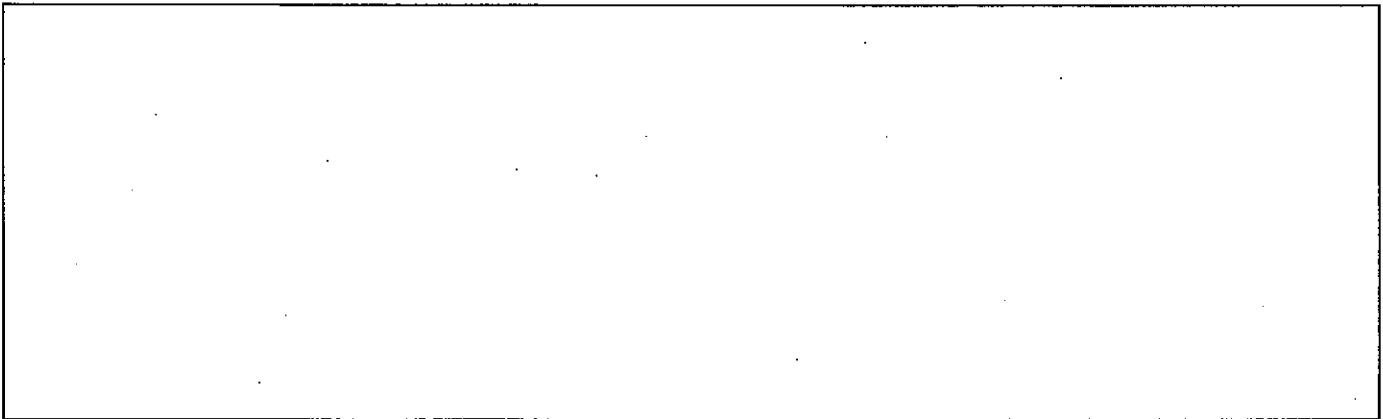
- (a) [4] How would you calculate the speedup which could be obtained when using parallel processing? If you introduce a new formula explain its parameters (as done above).

- (b) [4] Draw the diagram for the Flynn taxonomy, label each portion clearly, but do not describe.

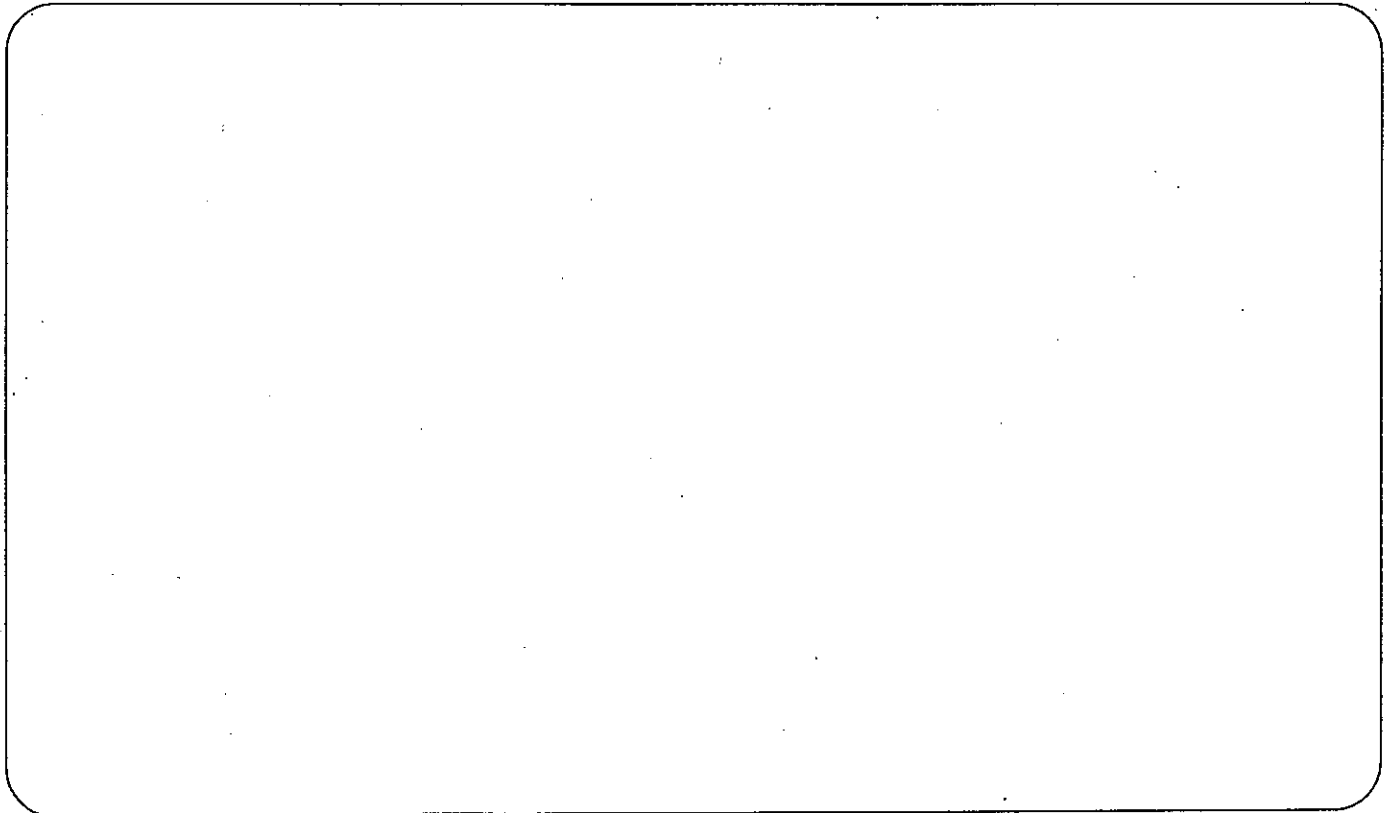
(c) [2] Choose one portion of the Flynn taxonomy above and describe it *briefly*.



(d) [4] Draw a diagram showing the difference organization of what is called a "Multi-core" design versus a "Multi-processor" design.



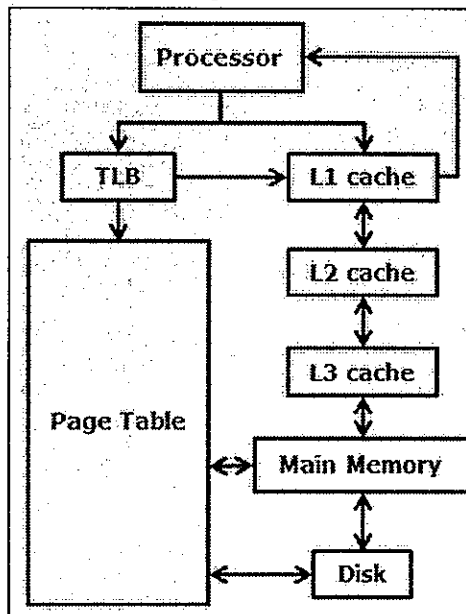
**Question 6.** [8] Explain briefly what is dynamic linking. Be precise and concise.



**Question 7. [8]** Answer the questions below as True or False.

Feature	TRUE	FALSE
Between big-endian and little-endian, the numbering of words is reversed (that is left-to-right or right-to-left), the numbering of bytes is reversed and the numbering of bits within a byte is reversed.		
RISC processors allow at most one operand to be in memory in arithmetic instructions.		
In a Virtual Memory organization, the TLB is an index to the Page Table.		
Imperative languages like C, Fortran and ARM use a compiler to translate to machine code native to the underlying processor, while object-oriented languages like C++, C# and Java translate only to an intermediate code (e.g. bytecode) which later executes on a virtual machine.		
In the INTEL architectures, the L1 cache is found inside the processor chip itself, while the L2 and L3 caches are external to the processor (albeit close by).		
Each level of cache, whether L1, L2 or L3, is further subdivided into I-Cache (for instructions) and D-Cache (for data).		
In the traffic light simulator of assignment 3, polling was implemented to capture the pedestrian crossing signals after states S1 and S7, while interrupt processing was used for state S2.		
After an interrupt, processing may not return to the previous execution, but it always returns to the OS.		

**Question 8. [15]** The diagram below is a reproduction from your textbook, discussed in class (we talked also about other possibilities for arrows). Consider such a system with Virtual Memory, using an MMU, a Page Table and a TLB, plus 3 levels of cache, L1, L2 and L3.



(a) [1] Where is the TLB located?

(b) [1] Where is the Page Table located?

(c) [1] Where is the MMU located?

(d) [12] Suppose the CPU needs access to some data. There are 4 cases:

1. The data is in L1
2. The data is in L2 or in L3
3. The data is in memory

**4. The data is on disk**

State algorithmically the steps to be followed in the search and retrieval of this data for each of the 4 cases, clearly naming which component is involved and how.

Case 1: data is in L1 cache

Case 2: data is in L2 or L3 cache

Case 3: data is in memory

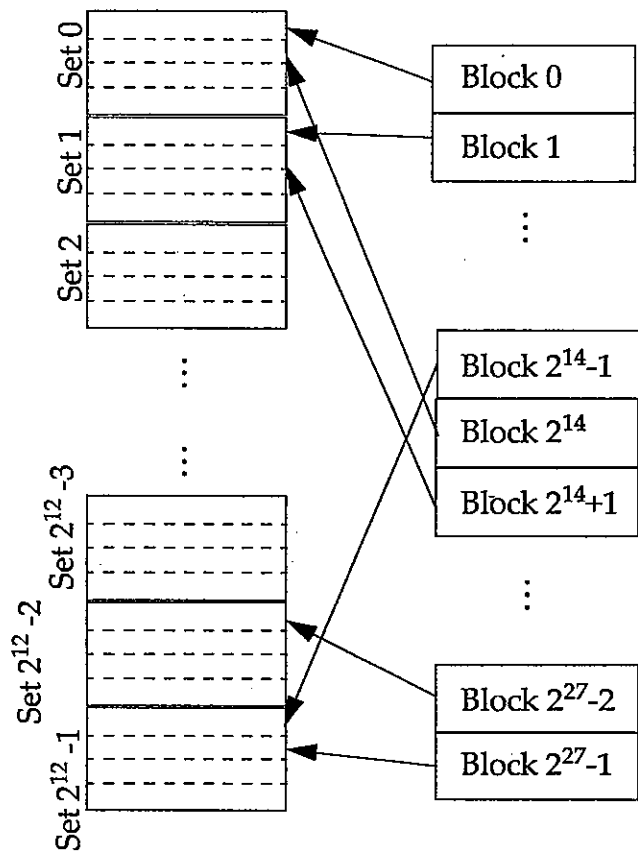
Case 4: data is on disk

**Question 9. [8]** In interrupt processing "Interrupt Vectors" are used. Explain concisely and clearly what they are and how they are integrated in the protocol for handling interrupts.

**Question 10. [12]** The diagram below shows a memory and a 4-way set associative mapped cache organization. It also shows the mapping of a few memory blocks to their set/cache lines (memory is comprised of  $2^{27}$  blocks). Redraw the diagram to show what would change if the cache used a direct mapping strategy and state which cache blocks would be used by the memory blocks in the order presented in the table below. Relabel the number of blocks precisely.



Redraw the cache here



Memory Block Number	State where it is mapped in a Direct Mapped Cache
0	
1	
$2^{14}-1$	
$2^{14}$	
$2^{14}+1$	
$2^{27}-2$	
$2^{27}-1$	

**Question 11. [8]** It is always good to learn something new, even in a test situation, to show that one has really grasped the material by being able to expand on it. On the topic of parameter passing to a function, you have learned to have them placed in registers only. Another technique is to place them

on the system stack. You have also seen how a mixture of these two protocols is used as a standard convention by the C compiler (in your assignment 4 and quiz 4). This requires careful management in order to avoid mixing the parameters with the storage of locally used registers and with the *allocation of local variables* (new to you). The protocol is described in the following specifications and you are to show your understanding of it by practising on an example.

- The caller routine places the parameters on the stack using either STR or STMFD instructions. In this example a mixture is used.

These three ARM instructions are used to call the function Foo with three input parameters passed on the stack. The current values of the parameters are in R2, R3 and R4.

CallFoo:

```
STR      R2, [SP, #-4]!
STMFD    SP!, {R3-R4}
BL       Foo
```

Next: . . . .

- (a) [2] Show the contents of the stack and the stack pointer after the execution of these instructions and immediately after the execution of "BL Foo".

- At the entry point in the function "Foo", the usual STMFD instruction to save registers used locally is issued. This is followed by the setting a value for the "Frame Pointer", labelled "FP", which is register R12 in ARM. The Frame Pointer provides a local pointer within the stack for the *frame* (also called *scope*) of the function currently being executed. This is needed, especially during recursion, since the global stack pointer may be moved by subsequent calls, while FP can be saved and restored locally. In fact FP is treated in the same way as the Link Register, "LR", and a copy of it is saved together on the stack together with LR and the other registers. If local variables need to be allocated (as is often the case), their space is found on the stack (*not* in the .DATA). Why? Because local variables need to be de-allocated after the exit from a function - they disappear. For this example "Foo", the code at the entry point is shown below, given that R5, R6 and R7 are used locally and that space for 2 local variables needs to be allocated on the stack.

```
Foo: STMFD SP!, {R5-R7, FP, LR}  @save registers
      ADD  FP, SP, #12           @ set FP
      ADD  SP, SP, #-8           @allocate space on
                                @stack for 2 local
                                @variables
```

- (b) [3] Show the updated contents of the stack, with SP and FP clearly marked.

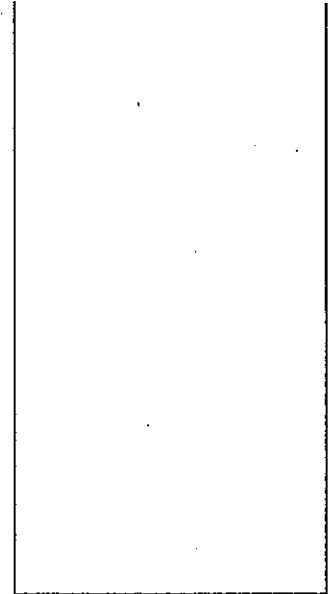
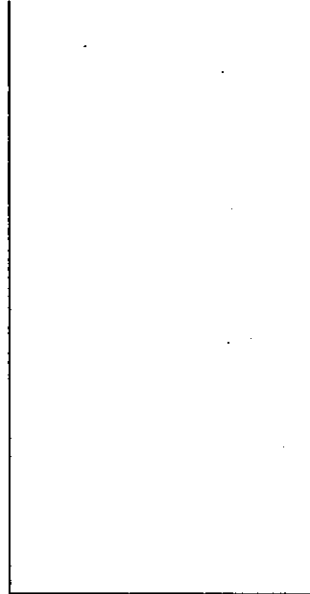
- When exiting a function, the stack must be restored and similarly the various pointers SP, FP and LR. First of all the space allocated for local variables must be released and then the stack cleared. This is done with two instructions shown below for this example Foo.

```
exitFoo:
```

```
    ADD SP, SP, #8 @free space
                        @for 2 locals
    LDMFD    SP!, {R5-R7, FP, PC}
```

Control next returns to the calling routine, at "Next" above.

- (c) [2] Show the updated contents of the stack and of the *SP pointer only*, after execution of each of these two instructions using the two diagrams (one diagram after each instruction).



- (d) [1] Finally, back to the calling routine at label Next. It was the calling routine which pushed the parameters on the stack to start with and thus it is its responsibility to clear the stack. Give the one instruction needed at this point to restore the stack to the condition it was before any preparation for the call to Foo was done, that is, before execution at label "CallFoo".

**Question 12. [15]** The *Log2* program computes an estimate for the base 2 logarithm of an integer. More precisely, given a positive number  $N$ , it finds the smallest value of  $k$  such that  $2^k$  is greater than or equal to  $N$ . You are asked to write programs in C and ARM to implement the *Log2* program. In ARM you can assume that you can load  $N$  into a register from a data storage variable named  $N$ , and place the result  $k$  into a variable named  $K$ . Use a simple loop to find the correct value for  $k$ . For example, if  $N = 17$ , then  $k = 5$ ; if  $N = 8$ , then  $k = 3$ .

Write a single program, i.e. one "main" for each, with no functions being called.

You will be evaluated on the following items:

- precise and neat pseudo-code;
- a flow-chart as used to design your ARM code and reflecting the structure of your code;
- the correctness of your code;
- the style of your code;
- the well written comments.

(a) [5] C code. Make sure that it is clear what algorithm/logic you are using (else you'll lose marks). Note: you may not need all this space!

(b) [2] Flow chart for your ARM code, followed by [8] the ARM code itself. Note: you may not need all this space!

**Question 13. [15]** The pseudocode shown below implements an array of counters. The length of the array is *Len*, and an array of words named *Count* contains the counters. Given a position *Pos*, the pseudocode increments the counter in position *Pos*.

```
@ given Pos (where  $0 \leq \text{Len}$ )
IF Pos < Len THEN
    Count[Pos] = Count[Pos] + 1
NEXT: code continues here
```

Write the ARM Assembly code segment to implement the pseudocode. State any assumptions. Do not write any loop structure, only the lines for the IF statement and the increment to Count [Pos] . Note: you may not need all this space!

```
.EQU      Len, #7
.text
.global   _start

_start:
    @ on entry, assume Pos is in R1, R2 = &Count and Len has been
    @ declared by an EQU
    @ use R3-R7 for temporary computation as needed
```

```
.data
Count:   .word    13, 5, 4, 23, 6, 8, 12
        .end
```

**Question 14. [2]** In your program for the traffic light controller (assignment 3) you used the function Wait which was given to you. The function contained the following code segment which you were expected to understand fully. Show now your understanding by stating what the value in each register is after the execution of each line. An example value for R0 is given to you to help you show exactly what goes on.

```
.equ EmbestTimerMask, 0x7fff @15 bit mask for Embest timer
...
ldr r7, =EmbestTimerMask @R7 = _ _ _ _ _
swi SWI_GetTicks @get time T1 @R0 = 3 5 F D F A 1 1
and r1, r0, r7 @T1 in 15 bits @R1 = _ _ _ _ _
```

**Question 15. [3]** Given the 4-bit hexadecimal number below, state the *decimal* equivalent according to the representation shown in each heading:

Hexadecimal	Unsigned Integer	2's Complement	Signed Magnitude
<b>D</b> <sub>16</sub>			

**Question 16. [10]** Consider the ARM instruction LDR r1, [r2, r4]!. List precisely all the steps required in the fetch-decode-execute cycle in the ARM architecture, stating clearly which registers and which buses may be involved in the process (similarly to what you did in the assignments).

**Question 17. [16]** A full example of how the cache works was given in some extra notes to you, showing all the steps from a textbook problem. Here it is reproduced for you.

A computer uses a small direct-mapped cache between the main memory and the processor. The cache has four 16-bit words, and each word has an associated 13-bit tag, as shown in the figure on the right, where, however, the tag has been ignored for this task. When a miss occurs during a read operation, the requested word is read from the main memory and sent to the processor. At the same time, it is copied into the cache, and its block number is stored in the associated tag.

	16 bit content
address 0	
address 2	
address 4	
address 6	

Consider the following loop in a program where all instructions and operands are 16 bits long and the code starts at address 0x02EC:

```

LOOP:ADD  (R1)+, R0      02EC
      DECR R2            02EE
      BNE  LOOP          02F0
  
```

Assume that, before this loop is entered, registers R0, R1 and R2 contain: R0 = 0, R1 = 0x054E, R2 = 3. Also assume that main memory contains the data as shown on the right

memory address	16 bit content
054E	A03C
0550	05D9
0552	10D7

The example went through the execution of the code segment showing how the cache is used in every step. First of all, it makes it easy to decide beforehand the location in cache to which each word is mapped. For the 3 instructions, the calculation was as follows:

instruction	address of instruction	
LOOP:ADD (R1)+, R0	02EC = 0000 0010 1110 1100	mapped to cache 4
DECR R2	02EE = 0000 0010 1110 1110	mapped to cache 6
BNE LOOP	02F0 = 0000 0010 1111 0000	mapped to cache 0

For the 3 memory locations, the calculation was as follows:

content	address in memory	
A03C	054E = 0000 0101 0100 1110	mapped to cache 6
05D9	0550 = 0000 0101 0101 0000	mapped to cache 0
10D7	0552 = 0000 0101 0101 0010	mapped to cache 2

Looking at the execution, here is the first pass through the loop:

- The "ADD" instruction is fetched (1 Memory access) and placed in cache slot 4.
- The data at address "054E" (as pointed to by R1) is retrieved from memory and placed in cache slot 6 (the data is "A03C"), with 1 Memory access.
- Now the "ADD" is executed, followed by the increment to R1.
- The "DECR" instructions is fetched (1 Memory access) and placed in cache slot 6, deleting the previous entry "A03C" from the data.
- Now the "DECR" is executed.
- The "BNE" instructions is fetched (1 Memory access) and placed in cache slot 0.
- Now the "BNE" is executed and control returns to the first instruction.

After 1st iteration

cache line	16 bit content
0	BNE
2	
4	ADD
6	A03C DECR

After this first iteration the cache looks as shown above and there have been 4 memory accesses.



The second pass is similar, but some memory accesses are avoided, as follows:

<ul style="list-style-type: none"> <li>• The "ADD" instruction is found in cache slot 4 (1 cache access).</li> <li>• The data at address "0550" (as pointed to by R1) is retrieved from memory and placed in cache slot 0 (the data is "05D9"), with 1 Memory access, and overwriting the previous content.</li> <li>• Now the "ADD" is executed, followed by the increment to R1.</li> <li>• The "DECR" instructions found in cache slot 6 (1 cache access).</li> <li>• Now the "DECR" is executed.</li> <li>• The "BNE" instructions is fetched again from memory (1 Memory access) and placed in cache slot 0, overwriting the previous content.</li> <li>• Now the "BNE" is executed and control returns to the first instruction.</li> </ul>	After 2nd iteration	
	cache line	16 bit content
	0	05D9 BNE
	2	
	4	ADD
	6	DECR

Thus the second iteration uses 2 memory accesses and 2 cache accesses. The final 3rd iteration proceeds as follows:

<ul style="list-style-type: none"> <li>• The "ADD" instruction is found in cache slot 4 (1 cache access).</li> <li>• The data at address "0552" (as pointed to by R1) is retrieved from memory and placed in cache slot 2 (the data is "10D7"), with 1 Memory access.</li> <li>• Now the "ADD" is executed, followed by the increment to R1.</li> <li>• The "DECR" instructions found in cache slot 6 (1 cache access).</li> <li>• Now the "DECR" is executed.</li> <li>• The "BNE" instructions is found in cache slot 0 (1 cache access). and placed in cache slot 0, overwriting the previous content.</li> <li>• Now the "BNE" is not executed and control continues to the next instruction after the loop structure.</li> </ul>	After 3rd iteration	
	cache line	16 bit content
	0	BNE
	2	10D7
	4	ADD
	6	DECR

In summary the table below shows the totals for memory and cache accesses. The example also assumes that the access time to main memory is  $10t$  and that of cache is  $1t$ . The execution time for each pass, ignoring the time taken by the processor between memory cycles, is calculated in the right column.

1st iteration	4 memory accesses	$(10t \times 4) = 40t$	Total execution time: $75t$
2nd iteration	2 memory accesses, 2 cache accesses	$(2t \times 2) + (2t \times 2) = 22t$	
3rd iteration	1 memory access, 3 cache accesses	$(10t \times 1) + (1t \times 3) = 13t$	

Your task now is to repeat the process using separate instruction and data caches, each of the same size as the previous one. You do not need to show every step with explanation as above, only:

- A diagram of the cache after each iteration is completed.
- A summary of the memory and cache accesses after each iteration is completed.
- A final table showing the total speed of execution.
- A final comment on whether separate caches appear to speed up execution.

Use the tables below to fill in your answers.

After 1st iteration

INSTR. CACHE		DATA CACHE	
cache line	16 bit content	cache line	16 bit content
0		0	
2		2	
4		4	
6		6	
Memory accesses =			
Cache accesses =			

After 2nd iteration

INSTR. CACHE		DATA CACHE	
cache line	16 bit content	cache line	16 bit content
0		0	
2		2	
4		4	
6		6	
Memory accesses =			
Cache accesses =			

After 3rd iteration

INSTR. CACHE		DATA CACHE	
cache line	16 bit content	cache line	16 bit content
0		0	
2		2	
4		4	
6		6	
Memory accesses =			
Cache accesses =			

Iterations	Total accesses	Total time per iteration	Total execution time:
1st iteration			
2nd iteration			
3rd iteration			

----- THE END -----

Operation	Assembler	Action
Move	MOV {S} Rd, <Oprnd2>	Rd := Oprnd2 {CPSR}
	MVN {S} Rd, <Oprnd2>	Rd := NOT Oprnd2 {CPSR}
Arithmetic	ADD {S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 {CPSR}
	ADC {S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	SUB {S} Rd, Rn, <Oprnd2>	Rd := Rn - Oprnd2 {CPSR}
	SBC {S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	RSB {S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn {CPSR}
	RSC {S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn - NOTCarry {CPSR}
	MUL {S} Rd, Rm, Rs	Rd := Rm * Rs {CPSR}
Logical	MLA {S} Rd, Rm, Rs, Rn	Rd := Rm * Rs + Rn {CPSR}
	CLZ Rd, Rm	Rd := # leading zero in Rm
	AND {S} Rd, Rn, <Oprnd2>	Rd := Rn AND Oprnd2 {CPSR}
	EOR {S} Rd, Rn, <Oprnd2>	Rd := Rn EXOR Oprnd2 {CPSR}
	ORR {S} Rd, Rn, <Oprnd2>	Rd := Rn OR Oprnd2 {CPSR}
	TST Rn, <Oprnd2>	Update CPSR on Rn AND Oprnd2
	TEQ Rn, <Oprnd2>	Update CPSR on Rn EOR Oprnd2
	BIC {S} Rd, Rn, <Oprnd2>	Rd := Rn AND NOT Oprnd2 {CPSR}
	NOP	R0 := R0
	CMP Rd, <Oprnd2>	Update CPSR on Rn - Oprnd2
Compare	B {cond} label	R15 := label
Branch	BL {cond} label	R14 := R15-4; R15 := label
Swap	SWP Rd, Rm	temp := Rn; Rn := Rm; Rd := temp
Load	LDR Rd, <a_mode2>	Rd := address
	LDM <a_mode4L> Rd {}, <reglist>	Load list of registers from [Rd]
Store	STR Rd, <a_mode2>	[address] := Rd
	STM <a_mode4S> Rd {}, <reglist>	Store list of registers to [Rd]
SWI	SWI <immed_24>	Software Interrupt

## Addressing Mode 2 - Data Transfer

Pre-indexed	Addressing Mode 2 - Data Transfer	
	Immediate offset	[Rn, #+/-<immed_12>]{!}
	Zero offset	[Rn]
	Register offset	[Rn, +/-Rm]{!}
	Scaled register offset	[Rn, +/-Rm, LSL #<immed_5>]{!}
		[Rn, +/-Rm, LSR #<immed_5>]{!}
		[Rn, +/-Rm, ASR #<immed_5>]{!}
Post-indexed		[Rn, +/-Rm, ROR #<immed_5>]{!}
	Immediate offset	[Rn, #+/-<immed_12>
	Register offset	[Rn], +/-Rm
	Zero offset	[Rn]
	Scaled register offset	[Rn], +/-Rm, LSL #<immed_5>
		[Rn], +/-Rm, LSR #<immed_5>
		[Rn], +/-Rm, ASR #<immed_5>
		[Rn], +/-Rm, ROR #<immed_5>
		[Rn], +/-Rm, ROR

## Key to tables

{cond}	See Condition Field
<Oprnd2>	See Operand 2
{S}	Updates CPSR if present
<immed>	Constant
<a_mode2>	See Addressing Mode 2
<a_mode4>	See Addressing Mode 4
<reglist>	List of registers with commas
{!}	Updates base register if present

2 of 2

Operand 2	
Immediate value	#<immed_8>
Logical shift left immediate	Rm, LSL #<immed_5>
Logical shift right immediate	Rm, LSR #<immed_5>
Arithmetic shift right immediate	Rm, ASR #<immed_5>
Rotate right immediate	Rm, ROR #<immed_5>
Register	Rm
Rotate right extended	Rm, RRX
Logical shift left register	Rm, LSL Rs
Logical shift right register	Rm, LSR Rs
Arithmetic shift right register	Rm, ASR Rs
Rotate right register	Rm, ROR Rs

Addressing Mode 4 - Multiple Data Transfer			
Block load		Stack pop	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending
Block store		Stack push	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

Condition Field	
EQ	Equal
NE	Not equal
CS	Carry Set
CC	Carry clear
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always

Dec	Bin	Hex
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F

D	BIN	H	D	BIN	H	D	BIN	H
0	00000000	00	4	00000100	04	8	00001000	08
1	00000001	01	5	00000101	05	9	00001001	09
2	00000010	02	6	00000110	06	10	00001010	0A
3	00000011	03	7	00000111	07	11	00001011	0B
						12	00001100	0C
						13	00001101	0D
						14	00001110	0E
						15	00001111	0F