Monash University
Faculty of Information Technology
Semester Two 2017 – Mid-Semester Test

EXAM CODES:	**FIT2085**
TITLE OF PAPER:	**INTRODUCTION TO COMPUTER SCIENCE FOR ENGINEERS**

THIS PAPER IS FOR STUDENTS STUDYING AT: (office use only - tick where applicable)

Berwick ☐	Clayton ✓	Peninsula ☐	Distance Education ☐	Open Learning ☐
Caulfield ☐	Gippsland ☐	Malaysia ☐	Enhancement Studies ☐	Other (specify) ☐

Candidates must complete this section

STUDENT ID __ __ __ __ __ __ __ __

SURNAME ....SIGNATURE...

OTHER NAMES (in full) ....

Candidates are reminded that they should have no material on their desks unless their use has been specifically permitted by the following instructions.

<u>**AUTHORISED MATERIALS**</u>

CALCULATORS	**YES** ☐	**NO** ✓
OPEN BOOK	**YES** ☐	**NO** ✓
SPECIFICALLY PERMITTED ITEMS	**YES** ☐	**NO** ✓

if yes, items permitted are:

<u>**INSTRUCTIONS TO CANDIDATES**</u>

1. Print your name and ID number in the section above.
2. Answer all questions in the space provided.
3. The duration of the test is **50 minutes.**
4. Total marks for this test are 40.
5. Individual marks are indicated for each question.
6. Calculators are **not** permitted.
7. **Candidates must NOT remove this paper from the examination room.**

Do not open this paper until you are instructed to do so.

Official use only

Question		Marks
1		8
2		7
3		8
4		10
5		7
Total:		40

Question 1 (8 marks)

This question is about Big-O time complexity. For each of the following fragments of code, give the **best** and **worst** time complexity using Big-O notation, assuming `the_list` has length `n` and its elements are strings of size `m`. Provide an explanation (no explanation, no marks). Note that function `range(n)` generates a sequence of integers from `0` to `n-1`, while function `range(k,n)` generates a sequence of integers from `k` to `n-1`.

a)
```
def code1(the_list):
    for i in range(len(the_list)):
        for j in range(i+1, len(the_list)):
            if the_list[i] == the_list[j]:
                print(i,j)
```

b)
```
def code2(the_list, string):
    i = 0
    while i < len(the_list) and string < the_list[i]:
        i = i + 1
    return i
```

c)
```
def code3(the_list):
    n = len(the_list)
    return (n > 0 and n%2==0 and the_list[0] == the_list[n-1])
```

8

Question 2 (7 marks)

This question is about *exceptions* and *assertions*. The program below can be improved to check its precondition and take care of input errors (by asking the user to re-input the number until it is correct). We ask that you do so by using appropriate assertion(s) and exception(s) handling. Note however that you do not need to check the type of the parameter of **`inverse()`**. Recall that the function **`int()`** throws an exception of the type **`ValueError`** if an improper parameter is passed.

```
#Takes a non-zero number as parameter and returns the inverse value
def inverse(a):
    return 1/a

#Reads an integer number n>0 and prints the result of inverse(n)
def main():
    n = int(input("Input a non-negative integer number: "))
    print(inverse(n))
```

7

Question 3 (8 marks)

Consider the following memory diagram of the layout of the stack and the heap at a point in the life of function **`f`**, which is about to call function **`g(a, b)`**, where **`a`** and **`b`** are local variables of **`f`**. Assume function **`g(a, b)`** receives in **`a`** an array of integers, in **`b`** and integer, and has a local integer variable **`temp`** which is initialized to **`a`**.

Use the empty spaces for the heap, stack and registers appearing on the right hand side to draw the memory diagram at the point once **`g(a, b)`** has been called and the local variable **`temp`** has been allocated and initialized. Make sure you name the registers and memory cells appropriately and provide adequate values for its contents. Assume also that the return address to **`f`** is **`0x00401234.`**

	Name	Contents	Address
Heap	a.length	1	0x10014F20
	a[0]	4	0x10014F24
			0x10014F28
Stack			0x7FFEFFE4
			0x7FFEFFE8
			0x7FFEFFEC
			0x7FFEFFF0
			0x7FFEFFF4
	b	5	0x7FFEFFF8
	a	0x10014F20	0x7FFEFFFC
			0x7FFF0000
Regs	$fp	0x7FFF0000	
	$sp	0x7FFEFFF8	

Contents	Address
	0x10014F20
	0x10014F24
	0x10014F28
	0x7FFEFFE4
	0x7FFEFFE8
	0x7FFEFFEC
	0x7FFEFFF0
	0x7FFEFFF4
	0x7FFEFFF8
	0x7FFEFFFC
	0x7FFF0000

8

Question 4 (10 marks)

This question is about *MIPS programming*. Assume **`i`** and **`a`** are local variables located at **`-4($fp)`** and **`-8($fp)`**, respectively, with **`a`** being an array of integers. In the space next to the (fragment of) Python code, complete the MIPS program so that it constitutes a ***faithful*** translation of the original Python program (no need to write a full MIPS program). **Remember to comment the code and use only the instructions provided in the MIPS reference sheet given at the end of the test.**

Python code	MIPS code	MIPS comments
`if  i <= 0:`		
`    print(a[i])`		
`else:`		
`    print(i)`		

10

Question 5 (7 marks)

This question is about *MIPS programming.* In the space next to the (uncommented) MIPS code, provide Python code that has the same functionality as the MIPS code. As we have done in the lectures, assume you have a **`read(i)`** function in Python that reads an integer and assigns it to variable **`i`**. I have left space for MIPS comments; they are not needed to get full marks, but might help you to add them.

MIPS code	MIPS comments	Python code

`.data`
`g: .word 0`
`.text`
`main:`
`addi $v0, $0, 5`
`syscall`
`sw $v0, g`
 	
`loop:`
`lw $t0, g`
`slt $t1, $0, $t0`
`beq $t1, $0, end`
`addi $v0, $0, 1`
`add $a0, $t0, $0`
`syscall`
`addi $t0, $t0, -1`
`sw $t0, g`
`j loop`
 	
`end:`
`addi $v0, $0, 10`
`syscall`		

END OF TEST

7

MIPS reference sheet for FIT2085
Semester 1, 2017

Table 1: System calls

Call code (`$v0`)	Service	Arguments	Returns	Notes
1	Print integer	`$a0` = value to print	-	value is signed
4	Print string	`$a0` = address of string to print	-	string must be terminated with `'\0'`
5	Input integer	-	`$v0` = entered integer	value is signed
8	Input string	`$a0` = address at which the string will be stored `$a1` = maximum number of characters in the string	–	returns if `$a1-1` characters or Enter typed, the string is terminated with `'\0'`
9	Allocate memory	`$a0` = number of bytes	`$v0` = address of first byte	-
10	Exit	-	-	ends simulation

Table 2: General-purpose registers

Number	Name	Purpose
R00	`$zero`	provides constant zero
R01	`$at`	reserved for assembler
R02, R03	`$v0, $v1`	system call code, return value
R04–R07	`$a0--$a3`	system call and function arguments
R08–R15	`$t0--$t7`	temporary storage (caller-saved)
R16–R23	`$s0--$s7`	temporary storage (callee-saved)
R24, R25	`$t8, $t9`	temporary storage (caller-saved)
R28	`$gp`	pointer to global area
R29	`$sp`	stack pointer
R30	`$fp`	frame pointer
R31	`$ra`	return address

Table 3: Assembler directives

.data	assemble into data segment
.text	assemble into text (code) segment
.word w1[, w2, ...]	allocate word(s) with initial value(s)
.space n	allocate n bytes of uninitialized, unaligned space
.ascii "string"	allocate ASCII string, do not terminate
.asciiz "string"	allocate ASCII string, terminate with `'\0'`

Table 4: Function calling convention

On function call:	**Caller:** saves temporary registers on stack passes arguments on stack calls function using `jal fn_label`	**Callee:** saves value of `$ra` on stack saves value of `$fp` on stack copies `$sp` to `$fp` allocates local variables on stack

On function return:	**Callee:** sets `$v0` to return value clears local variables off stack restores saved `$fp` off stack restores saved `$ra` off stack returns to caller with `jr $ra`	**Caller:** clears arguments off stack restores temporary registers off stack uses return value in `$v0`

Table 5: MIPS instruction set

Instruction format	Meaning	Operation	Immediate	Unsigned
add Rdest, Rsrc1, Src2	Add	Rdest = Rsrc1 + Src2	addi	addu (no overflow trap)
sub Rdest, Rsrc1, Src2	Subtract	Rdest = Rsrc1 - Src2	-	subu (no overflow trap)
mult Rsrc1, Src2	Multiply	Hi:Lo = Rsrc1 * Src2	-	mulu
div Rsrc1, Src2	Divide	Lo = Rsrc1/Src2; Hi = Rsrc1 % Src2	-	divu
and Rdest, Rsrc1, Src2	Bitwise AND	Rdest = Rsrc1 & Src2	andi	-
or Rdest, Rsrc1, Src2	Bitwise OR	Rdest = Rsrc1 $\vert$ Src2	ori	-
xor Rdest, Rsrc1, Src2	Bitwise XOR	Rdest = Rsrc1 $\wedge$ Src2	xori	-
nor Rdest, Rsrc1, Src2	Bitwise NOR	Rdest = (Rsrc1 $\vert$ Src2)	-	-
sllv Rdest, Rsrc1, Src2	Shift Left Logical	Rdest = Rsrc1 << Src2	sll	-
srlv Rdest, Rsrc1, Src2	Shift Right Logical	Rdest = Rsrc1 >> Src2 (MSB=0)	srl	-
srav Rdest, Rsrc1, Src2	Shift Right Arithmetic	Rdest = Rsrc1 >> Src2 (MSB preserved)	sra	-
mfhi Rdest	Move from Hi	Rdest = Hi	-	-
mflo Rdest	Move from Lo	Rdest = Lo	-	-
lw Rdest, Addr	Load word	Rdest = mem32[Addr]	-	-
sw Rsrc, Addr	Store word	mem32[Addr] = Rsrc	-	-
beq Rsrc1, Rsrc2, label	Branch if equal	if (Rsrc1 == Rsrc2) PC = label	-	-
bne Rsrc1, Rsrc2, label	Branch if not equal	if (Rsrc1 != Rsrc2) PC = label	-	-
slt Rdest, Rsrc1, Src2	Set if less than	if (Rsrc1 < Src2) Rdest = 1 else Rdest = 0	slti	sltu
j label	Jump	PC = label	-	-
jal label	Jump and link	$ra = PC + 4; PC = label	-	-
jr Rsrc	Jump register	PC = Rsrc	-	-
jalr Rsrc	Jump and link register	$ra = PC + 4; PC = Rsrc	-	-