Monash University
Faculty of Information Technology
Semester Two 2018 – 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 _ _ _ _ _ _ _ _

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

AUTHORISED MATERIALS

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

if yes, items permitted are:

INSTRUCTIONS TO CANDIDATES
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 80.
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

Page	Score	Points
2		6
3		14
4		20
6		24
8		8
9		8
Total:		80

Question 1: [20 marks]

Consider the naive implementation of bubble sort given below

```
def swap(the_list,a,b):
        tmp = the_list[a]
        the_list[a] = the_list[b]
        the_list[b] = tmp

def bubbleSort(aList):
        for i in range(len(aList)):
                for j in range(len(aList)-1):
                        if aList[j] > aList[j+1]:
                                swap(aList,j,j+1)
        return aList
```

(a) (6 marks) We can modify this implementation to replace `range(len(aList)-1)` with `range(len(aList)-i-1)` in the inner (`j` dependent) loop. Will this have any impact on its:

(i) ***time complexity***?

(ii) ***actual runtime***?

Justify your answers.

6

(b) (6 marks) Explain in words how the implementation of bubble sort given can be updated to have a best case complexity of $O(N)$. Justify your answer.

(c) (8 marks) Consider the sorting (non-)algorithm known as bogoSort. One implementation of this is defined as follows:

```
terribleApproachBogoSort...
... get the_list
... while not inOrder(the_list)...
... ... set pos1 to a random index of the_list
... ... set pos2 to a random index of the_list
... ... swap elements at pos1 and pos2 in the_list
```

(i) What is sorting stability?

(ii) Assuming bogoSort terminated in finite time, would it be stable? Justify your answer with an example.

14

Question 2: [20 marks]

This question is about MIPS programming. Translate the following MIPS code into Python. Note that in this piece of code all variables are global variables. Use the space to the right of the MIPS code for your answer.

```
.data
a:  .word 20
b:  .word 41
c:  .word 0

.text
        lw $t0, a
        lw $t1, b

        add $t2, $t1, $t0
        sw $t2, c
        addi $t3, $0, 2
        lw $t2, c
        div $t2, $t3
        mfhi $t4

        beq $t4, $0, ed

jUp:    lw $t6, c
        addi $t6, $t6, 1
        sw $t6, c

ed:     lw $t5, c
        addi $t3, $0, 2
        div $t5,$t3
        mflo $t5
        sw $t5, c
        addi $v0, $0, 1
        lw $a0, c
        syscall
```


20

This page intentionally left blank, use if needed but it will not be marked *unless* you explicitly ask us to do so

0

Question 3: [24 marks]

Consider the following python code, which you wish to translate into MIPS. Complete the memory diagram when execution reaches the line with the comment #HERE. Include names and contents. PC for the `jal getMax` instruction would be `0x04000018` and $fp was `0x7FF00000` before jumping. You will need to add to the stack and heap anything *getMax* will need access to.

```python
def getMax(the_list):
        index = 0
        theMax = the_list[index]
        index+=1
        #HERE
        while index < len(the_list):
                if the_list[index]>theMax:
                        theMax = the_list[index]
                index+=1
        return theMax
getMax([4,1,9])
```

	Name	content	Address
Heap			0x10014F20
			0x10014F24
			0x10014F28
			0x10014F2C
			0x10014F30
			0x10014F34
Stack			0x7FFFEFC8
			0x7FFFEFCC
			0x7FFFEFD0
			0x7FFFEFD4
			0x7FFFEFD8
			0x7FFFEFDC
			0x7FFFEFE0
			0x7FFFEFE4
			0x7FFFEFE8
			0x7FFFEFEC
			0x7FFFEFF0
			0x7FFFEFF4
			0x7FFFEFF8
			0x7FFFEFFC
			0x7FFFF000
Registers	$fp		
	$ra		

24

This page intentionally left blank, use if needed but it will not be marked *unless* you explicitly ask us to do so

Question 4: [16 marks]

(a) (8 marks) Consider the 32bit instruction format for MIPS. Going to 64bit instructions can improve the performance or abilities of the system. Provide one reason (there are several) for this and justify your answer.

You should use your knowledge of the instruction format to justify this.
Note: you do not need to know the precise number of bits used for each part of the instruction but you should have a general idea of what's involved.

8

(b) (8 marks) Consider the MIPS code shown below.

```
.data
            value:  .word 25
.text
1.          lw $t0, value
2.          addi $t1,$0,4

3.          div $t0, $t1
4.          mflo $t0
5.          sw $t0, value

6.          lw $t0, value
7.          div $t0, $t1
8.          mflo $t0
9.          mfhi $t1

10.         lw $a0, value
11.         addi $v0, $0, 1
12.         syscall
```

Using the table provided, show the effect of each MIPS instruction on the ***HI, LO, t*0 *and* t*1 registers and on the contents of the label ***value*** in main memory.
Note: You may use '?' to represent an undefined value

Line number	HI	LO	$t0	$t1	value:
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					

line run...

End of Mid Semester Test

8

This page intentionally left blank, use if needed but it will not be marked *unless* you explicitly ask us to do so

0

MIPS reference sheet

Table 1: **SPIM 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	`$a0` = address to store string at `$a1` = maximum number of chars	–	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, $t9`	temporary storage (caller-saved)
R26, R27	`$k0, $k1`	reserved for kernel code
R28	`$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
.byte b1[, b2, ...]	allocate byte(s), with initial value(s)
.half h1[, h2, ...]	allocate halfword(s), with initial value(s)
.word w1[, w2, ...]	allocate word(s) with initial value(s)
.space n	allocate n bytes of uninitialized, unaligned space
.align n	align the next item to a 2^n-byte boundary
.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:	**Callee:**
saves temporary registers on stack	saves `$ra` and `$fp` on stack
passes arguments on stack	copies `$sp` to `$fp`
calls function using `jal fn_label`	allocates local variables on stack

On function return:

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

Table 5: **Instruction Set**

A partial instruction set is on the next page. The following conventions apply.

Instruction Format

Rsrc, Rsrc1, Rsrc2: source operand(s), - must be a register value(s)

Src2; source operand - may be an immediate value or a register value

Rdest: destination, must be a register

Imm: Immediate value, may be 32 or 16 bits

Imm16: Immediate 16-bit value

Addr: Address in the form: offset(Rsrc) ie. absolute address = Rsrc + offset

label: label of an instruction

⋆: pseudoinstruction

Immediate Form -: no immediate form, or this is the immediate form

⋆: immediate form synthesized as pseduoinstruction

Unsigned form (append 'u' to instruction name):

- : no unsigned form, or this is the unsigned form

Table 6: **MIPS instruction set**

Instruction format	Meaning	Operation	Immediate form	Unsigned form(u)
add Rdest, Rsrc1, Rsrc2	Add	Rdest = Rsrc1 + Rsrc2	addi	no overflow trap
sub Rdest, Rsrc1, Rsrc2	Subtract	Rdest = Rsrc1 - Rsrc2	$\star$	no overflow trap
mul Rdest, Rsrc1, Rsrc2 $\star$	Multiply	Rdest = Rsrc1 * Rsrc2	$\star$	unsigned operands
mulo Rdest, Rsrc1, Rsrc2 $\star$	Multiply (with 32-bit overflow)	Rdest = Rsrc1 * Rsrc2	$\star$	unsigned operands
mult Rsrc1, Rsrc2	Multiply (machine instruction)	Hi:Lo = Rsrc1 * Rsrc2	-	unsigned operands
div Rdest, Rsrc1, Rsrc2 $\star$	Divide	Rdest=Rsrc1/Rsrc2	$\star$	unsigned operands
div Rsrc1, Rsrc2	Divide (machine instruction)	Lo = Rsrc1/Rsrc2; Hi = Rsrc1 % Rsrc2	-	unsigned operands
rem Rdest, Rsrc1, Rsrc2 $\star$	Remainder	Rdest = Rsrc1 % Rsrc2	$\star$	unsigned operands
neg Rdest, Rsrc $\star$	Negate	Rdest = -Rsrc1	-	no overflow trap
and Rdest, Rsrc1, Rsrc2	Bitwise AND	Rdest = Rsrc1 & Rsrc2	andi	-
or Rdest, Rsrc1, Rsrc2	Bitwise OR	Rdest = Rsrc1 | Rsrc2	ori	-
xor Rdest, Rsrc1, Rsrc2	Bitwise XOR	Rdest = Rsrc1 $\wedge$ Rsrc2	xori	-
nor Rdest, Rsrc1, Rsrc2	Bitwise NOR	Rdest = ∼(Rsrc1 | Rsrc2)	$\star$	-
not Rdest, Rsrc $\star$	Bitwise NOT	Rdest = ∼(Rsrc)	−	-
sll Rdest, Rsrc1, Rsrc2	Shift Left Logical	Rdest = Rsrc1 << Rsrc2	-	-
srl Rdest, Rsrc1, Rsrc2	Shift Right Logical	Rdest = Rsrc1 >> Rsrc2 (MSB=0)	-	-
sra Rdest, Rsrc1, Rsrc2	Shift Right Arithmetic	Rdest = Rsrc1 >> Rsrc2 (MSB preserved)	-	-
move Rdest, Rsrc $\star$	Move	Rdest=Rsrc	-	-
mfhi Rdest	Move from Hi	Rdest = Hi	-	-
mflo Rdest	Move from Lo	Rdest = Lo	-	-
li Rdest, Imm $\star$	Load immediate	Rdest=Imm	-	-
lui Rdest, Imm16	Load upper immediate	Rdest=Imm16 << Imm	-	-
la Rdest, Addr(or label) $\star$	Load Address	Rdest=Addr (or Rdest=label)	-	-
lb Rdest, Addr (or label $\star$)	Load byte	Rdest = mem8[Addr]	-	zero-extends data
lh Rdest, Addr (or label $\star$)	Load halfword	Rdest = mem16[Addr]	-	zero-extends data
lw Rdest, Addr (or label $\star$)	Load word	Rdest = mem32[Addr]	-	-
sb Rsrc2, Addr (or label $\star$)	Store byte	mem8[Addr] = Rsrc2	-	-
sh Rsrc2, Addr (or label $\star$)	Store halfword	mem16[Addr] = Rsrc2	-	-
sw Rsrc2, Addr (or label $\star$)	Store word	mem32[Addr] = Rsrc2	-	-
beq Rsrc1, Rsrc2, label	Branch if equal	if (Rsrc1 == Rsrc2) PC = label	$\star$	-
bne Rsrc1, Rsrc2, label	Branch if not equal	if (Rsrc1 != Rsrc2) PC = label	$\star$	-
blt Rsrc1, Rsrc2, label $\star$	Branch if less than	if (Rsrc1 < Rsrc2) PC = label	$\star$	unsigned operands
ble Rsrc1, Rsrc2, label $\star$	Branch if less than or equal	if (Rsrc1 <= Rsrc2) PC = label	$\star$	unsigned operands
bgt Rsrc1, Rsrc2, label $\star$	Branch if greater than	if (Rsrc1 > Rsrc2) PC = label	$\star$	unsigned operands
bge Rsrc1, Rsrc2, label $\star$	Branch if greater than or equal	if (Rsrc1 >= Rsrc2) PC = label	$\star$	unsigned operands
slt Rdest, Rsrc1, Rsrc2	Set if less than	if (Rsrc1 < Rsrc2) Rdest=1 else Rdest=0	slti	unsigned operands
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	$ra = PC + 4; PC = Rsrc	-	-
syscall	System call	depends on call code in `$v0`	-	-