One and a half hours

Closed Book Examination
(A copy of an "ARM Instruction Set Summary" is attached)

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Fundamentals of Computer Architecture

Thursday 22\textsuperscript{nd} January 2015?

Time: 14:00 – 15:30?

Please answer Question One
and
ONE other Question from the remaining TWO questions provided

Use a SEPARATE answer book for each QUESTION

For full marks your answers should be CONCISE as well as ACCURATE. Marks will be awarded for reasoning and method as well as being correct. Excessively verbose answers will be penalised.

The use of electronic calculators is NOT permitted.
1. [Javier Navaridas-Palma]

a) Define briefly: (5 marks)

i) 3-address instruction style
ii) Von Neumann computational model
iii) Register
iv) Pseudo instruction
v) Assembler Directive

Bookwork - 1 mark each.

**3-address instruction style**: An instruction style in which all operands (typically 2) and results (typically 1) are read/written directly from/to memory. It is opposed to the load/store instruction style in which all operands and results need to be in registers.

**Von Neumann model**: The most common computational model. It divides computers in 3 main components.
Memory: A big storage area in which programs and data reside. Note that variables and instructions are indistinguishable for the processor. It will execute variables and use instructions as data if instructed to do so.
CPU: The working force: it manipulates the data as instructed by the programs.
Bus: A bidirectional communication path between memory and CPU.

**Register**: A very fast, very small piece of memory located within the processor used to reduce the transit between memory and cpu and to simplify instructions.

**Pseudo-instructions**: pieces of assembly code that are understood by the compiler and translated into one or several instructions the processor is able to understand. e.g. ADR, ADRL, some LDR and STR, …

**Directives**: pieces of assembly code which are actually commands for the compiler, and hence, are not translated into any instruction. They are typically used to define state; e.g. DEFW, ORIGIN, …
b) Describe the difference between the following pairs of ARM instructions:
(3 marks)

i) LDR and LDRLT
ii) STR and STRB
iii) SUBNE and SUBNES

Bookwork - 1 mark each.

LDR and LDRLT: Both of them load a word from memory. The first one will always be executed, whereas the second one is only executed if condition LT (Lower Than) is true (from last CMP or …S instruction).

STR and STRB: Both of them store a datum in memory. The first stores a word, the second a byte.

SUBNE and SUBNES: Both of them will execute a substraction only if the NE condition (Not Equal) applies. The second one will modify the status register so that the following conditional instructions depend on its result.

c) Discuss the difference between the Java Boolean operators & and &&. Explain conceptually, i.e. without giving any code, how each of them should be translated into assembly language: (2 marks)

Applied knowledge – 1 mark for understanding lazy operations + 1 mark for explaining the implementations

They both implement a Boolean AND operation, the difference is that & will evaluate the two subexpressions whereas && will only evaluate the second subexpressions if it is needed to know the final result (i.e. if the first subexpressions evaluates to true)

& is implemented using the bitwise AND operation between the results of the subexpressions. Note: sometimes it can be optimized by using conditional MOV instructions.

&& is implemented as an ‘if’ structure. If the first subexpression is true then evaluate the second subexpression else don’t.
d) What is the ‘heap’ in a Java Virtual Machine? Your explanation should include an explanation of when it is used and what is stored there. (2 mark)

<table>
<thead>
<tr>
<th>Bookwork (2 marks):</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following points should be covered to some degree in the answer:</td>
</tr>
<tr>
<td>The heap is an area of permanent memory which is allocated dynamically when needed. Its usual use in the implementation of Java is to store information associated with an instance of an object. The memory is allocated when a ‘new’ is executed. The major content of the memory allocated is the instance variables of the object. However, there is also extra information associated with the class, size and structure of the object.</td>
</tr>
<tr>
<td>ALSO:</td>
</tr>
<tr>
<td>The JVM has a memory manager.</td>
</tr>
<tr>
<td>This contains a subroutine which is called whenever a ‘new’ is executed.</td>
</tr>
<tr>
<td>The JVM has an area of store, the heap, where the memory manager allocates memory for newly created objects. This is clearly dynamic, i.e. the heap grows as the program runs.</td>
</tr>
</tbody>
</table>

2 marks for an answer that depicts all the salient facts in a sensible way; all facts are correctly delineated and briefly described;
1 ½ marks for a right-lines approach;
1 mark for some basic understanding (or attempt).

Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s.): Lecture 18: Java Memory Usage.

TOTAL marks (2 marks) [2]
e) Explain the term ‘Garbage Collection’ and explain why it is important to the implementation of Java.  

**Application (Critique) (4 marks):**

The following points should be covered to some degree in the answer:

i) Programs which use dynamic memory allocation may make use of the memory for a period but then, after a time, use it no longer. In these circumstances, a Garbage Collector tries to identify dynamically allocated memory which is no longer used and returns it to the memory allocation system.

Also, as we are continually creating new objects, we can run out of store. We would like to reclaim store; if it is no longer useful. An important part of the memory manager is a garbage collector. Called when the heap is getting full. Works out what is ‘garbage;’ e.g. un-referenced objects; no references [to it] exist …

ii) In Java, objects are continually allocated and discarded, so GC is very important. If this was not done by the GC the heap would fill up with un-referenced [unused] objects; this is important as the heap [nominally] has a set size (or limit on how much memory it can use) so the GC will stop the heap becoming full and remove unwanted [un-referenced] objects.

4 marks for an answer that depicts all the salient facts in a sensible way; good briefly described and expression is evaluated totally correct;  
3 marks for correct answer but not detailed [enough];  
2 mark for a right-lines approach;  
1 mark for some basic understanding (or attempt).  
Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s): Lecture 18: Java Memory Usage.  
TOTAL marks (4 marks) [6]
f) An Operating System consists of basically two main components: the kernel and its libraries. The kernel has ‘managers’ for the resources. Name the three managers and briefly describe each. 

Bookwork & explanation (4 marks):

The following points should be covered explicitly in any exam; to gain full marks for your answer:

The three-managers and descriptions are:

1. **Process Manager** (CPU): creates the illusion that several things can be going on at once.

2. **Memory Manager** (main memory): each program has all the memory it needs, but can’t access anything else.

3. **Peripheral Managers**: separate manager (device driver) for each (kind of) peripheral.

4 **marks** for an answer that depicts all the salient facts in a sensible way; good briefly described and expression is evaluated totally correct;

3 **marks** for correct answer but not detailed [enough];

2 **mark** for a right-lines approach;

1 **mark** for some basic understanding (or attempt).

Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s): Lecture 15: System Software.

TOTAL marks (4 marks) [10]
Section B

2.

[Javier Navaridas-Palma]

a) Explain briefly the following addressing modes and their limitations and benefits. (8 marks)

i) Direct addressing
ii) Indirect addressing
iii) Base+Offset addressing
iv) Pre-index addressing
v) Post-index addressing

Bookwork – 1 mark per correct definition (5), + 0.5 marks per correct Pros/Cons (up to +3).

**Direct addressing**: the target memory address is contained within the instruction
- **Pros**: don’t use extra resources (e.g. registers)
- **Cons**: not very flexible, needs to know every address in advance. Instruction needs to be larger than 1 word.

**Indirect addressing**: the target memory address is held elsewhere (typically a register)
- **Pros**: more flexible than Direct addressing, allows to modify the target address.
- **Cons**: modifying the target address needs extra instructions and loses the original address

**Base+Offset addressing**: the target address is calculated from a base address held elsewhere (typically a register) and a displacement (either in a literal or a register).
- **Pros**: flexibility, allows to move around the target address without changing it.
- **Cons**: extra instructions may be needed to calculate displacement

**Pre-index addressing**: Similar to base+offset addressing but changes the value of the base register before accessing memory.
- **Pros**: flexibility, updating the register within the memory access instruction, speeds up the access to data structures (e.g. in loops)
- **Cons**: As it modifies the target address the original base address is lost

**Post-index addressing**: Similar to Indirect addressing but changes the value of the register after accessing memory.
- **Pros**: flexibility, changing the register saves an instruction, speeds up the access to data structures (e.g. in loops)
- **Cons**: As it modifies the target address the original base address is lost

Other Pros/Cons may get some marks if justified appropriately
b) Consider the following Java switch statement:

```java
switch (day) {
    case 1:  dayString = "Monday";
        break;
    case 2:  dayString = "Tuesday";
        break;
    case 3:  dayString = "Wednesday";
        break;
    case 4:  dayString = "Thursday";
        break;
    case 5:  dayString = "Friday";
        break;
    case 6:  dayString = "Saturday";
        break;
    case 7:  dayString = "Sunday";
        break;
    default: dayString = "Invalid";
        break;
}
```

Describe two different ways of implementing it in ARM, discussing their efficiency for this specific example: (4 marks)

Application (Critique) – 2 marks implementations + 2 marks for a proper discussion of efficiency.

The 2 possible implementations covered in the lectures are:
- Nested IFs
- An address table

Discussions on exotic implementations not covered in the lectures (e.g. binary search of cases) may get some marks if justified adequately.

Efficiency:
For a large switch as the one in the example, using nested IFs is not very efficient because the last cases would need to check all the previous conditions. As all the cases are together discriminate against them will be pretty easy when using a table. For this reason, all the cases will take exactly the same time to be selected, which will be lower than the average case in the IF implementation. Also the table implementation would require a smaller memory footprint (3 instructions for default trapping + 2 instructions for accessing the table + 7 elements in the table vs 2 instruction x 7 cases).
The following function computes the factorial of a non-negative number recursively. Transform it into ARM assembly language, simplifying and optimising it as much as possible. Explain your optimisations and how your code passes parameters and results. Show the values in all the relevant registers and memory locations just before the fourth recursive call to function factorial. Assume the function is called originally with n=10 and that all registers and memory locations are initialized elsewhere. (8 marks)

```c
int factorial (int n){
    if (n==0)
        return 1;
    else
        return n * fact(n-1);
}
```

Problem solving – 2 marks for understanding the stack NEEDS to be used + 3 marks for correct use of stack (using BL and SP, not overwriting any registers and not leaving spurious elements in the stack) + 1 mark for using PUSH/POP or LDRM/STRM + 2 marks for correctly showing the stack and registers.

Passing parameter ‘n’ in R1 and returning the result in R0:

```asm
fact CMP R1, #0        ; If n==0 branch to retn
BEQ retn            
else PUSH (R1, LR)    ; Save LR and R1 in the stack so to avoid overwriting.
SUB R1, R1, #1       ; Put n-1 for the next call into R1
BL fact              ; and call fact to get fact(n-1).
POP (R1)             ; Retake the original parameter (R1) from the stack.
MUL R0, R1, R0       ; Compute fact(n). n*fact(n-1). R1:n, R0:fact(n-1).
POP (PC)             ; Return and remove LR from the stack.
retn MOV R0, #1      ; Put 1 in R0
MOV PC, LR           ; and return.
```

<table>
<thead>
<tr>
<th>Register Bank</th>
<th>Stack (in memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 = (Undetermined)</td>
<td></td>
</tr>
<tr>
<td>R1 = 7</td>
<td></td>
</tr>
<tr>
<td>Others = (unused)</td>
<td></td>
</tr>
<tr>
<td>SP = (see figure aside)</td>
<td></td>
</tr>
<tr>
<td>LR = @POP(R1)</td>
<td></td>
</tr>
<tr>
<td>PC = @BL fact</td>
<td></td>
</tr>
<tr>
<td>SP = R1 = 8</td>
<td>R1 = 9</td>
</tr>
<tr>
<td>LR = @POP(R1)</td>
<td>R1 = 9</td>
</tr>
<tr>
<td>R1 = 10</td>
<td>LR = @return</td>
</tr>
</tbody>
</table>

Page 9 of 16
Section C

3.

a) In your lectures and exercise classes you were introduced to code, in the form of a loop, that can check a table of peripherals to see which caused an interrupt; which would be something like that presented in figure 3.a; except that the code in the figure is out of order! Reorder the code into the correct order in your answer so it enables an interrupt handler to deal with several peripherals. Note: the data structure the code uses is given above the out-of-order code in figure 3.a. (5 marks)

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Labels</th>
<th>Instruction</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>0x00000088 Address_0_StatusRegister</td>
<td>DEFW</td>
<td>0x40000004</td>
</tr>
<tr>
<td></td>
<td>0x0000008C peripheral_0_TestPattern</td>
<td>DEFW</td>
<td>0x00000080</td>
</tr>
<tr>
<td></td>
<td>0x00000090 Address_1_StatusRegister</td>
<td>DEFW</td>
<td>0x40000008</td>
</tr>
<tr>
<td></td>
<td>0x00000094 peripheral_1_TestPattern</td>
<td>DEFW</td>
<td>0x00000100</td>
</tr>
</tbody>
</table>

(Data structure)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td>R1, R2</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>R2, [R0], #4</td>
<td></td>
</tr>
<tr>
<td>ADR</td>
<td>R0, table</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>R1, [R1]</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>R1, [R0], #4</td>
<td></td>
</tr>
</tbody>
</table>

(Code)

Question figure 3.a. Data structure and code; showing out-of-order code.
3.a. Application (example re-coding) (4 marks):
The following points should be covered to some degree in the answer:

The typical ordered code is:
1: ADR R0, table ;
2: loop
3: LDR R1, [R0], #4 ;
4: LDR R1, [R1] ;
5: LDR R2, [R0], #4 ;
6: TST R1, R2 ;
7: BEQ loop ;

From the previously out-of-order code:
7: BEQ loop ;
6: TST R1, R2 ;
5: LDR R2, [R0], #4 ;
1: ADR R0, table ;
4: LDR R1, [R1] ;
3: LDR R1, [R0], #4 ;
2: loop

4 marks for majority of above; e.g. for an answer that orders all lines correctly,
3 marks for correct answer but not all correct order [enough],
2 mark for some in order & for a right-lines approach,
1 marks for some basic understanding (or attempt).
Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s).: Lecture 14: Input/Output (2).

TOTAL marks (5 marks) [5]
b) Figure 3.b shows a code snippet to poll a simple device. Copy the code into your answer book (with a comments column) and then describe in full what happens when the ARM program is executed. In the table clearly describe the movement of information (both values and instructions) between memory, registers and the CPU in the comments column, and how the values in the registers R0, R1, and memory change, at each step. (5 mark)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
<th>Detailed comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>ADR</td>
<td>R1, Status_Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TST</td>
<td>R0, #0x80</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEQ</td>
<td>loop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADR</td>
<td>R1, Data_Reg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SVC</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Status_Reg</td>
<td>DEFW</td>
<td>0x80</td>
<td></td>
</tr>
<tr>
<td>Data_Reg</td>
<td>DEFW</td>
<td>0x72</td>
<td></td>
</tr>
</tbody>
</table>

Question figure 3.b. A table; requiring adding comprehensive comments.
3.b.
Application (example details) (5 marks):
The following points should be covered to some degree in the answer:

The typical fully commented ordered code is:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
<th>Detailed comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>ADR</td>
<td>R1, Status_Reg</td>
<td>; R1 points to status reg.; load address of ‘Status_Reg’ in to R1</td>
</tr>
<tr>
<td></td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td>; read status reg. [of device]; using R1 to point at address of ‘Status_Reg’</td>
</tr>
<tr>
<td></td>
<td>TST</td>
<td>R0, #0x80</td>
<td>; test ready bit (bit 7); test [or AND] R0 with immediate value 0x80</td>
</tr>
<tr>
<td></td>
<td>BEQ</td>
<td>loop</td>
<td>; if not ready, try again; branch if bit 7 = 0 (zero); or if equal zero, zero bit set in status reg.</td>
</tr>
<tr>
<td></td>
<td>ADR</td>
<td>R1, Data_Reg</td>
<td>; R1 points to data reg. [of device].; load address of ‘Data_Reg’ in to R1</td>
</tr>
<tr>
<td></td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td>; read data [from data register]; using R1 to point at address of ‘Data_Reg’</td>
</tr>
<tr>
<td></td>
<td>SVC</td>
<td></td>
<td>; end [exit or stop] program</td>
</tr>
<tr>
<td>Status_Reg</td>
<td>DEFW</td>
<td>0x80</td>
<td>; data definition, define word</td>
</tr>
<tr>
<td>Data_Reg</td>
<td>DEFW</td>
<td>0x72</td>
<td>; data definition, define word</td>
</tr>
</tbody>
</table>

5 marks for majority of above; e.g. for an answer that orders all lines correctly (2 marks for a detailed and concise description.),
3 marks for correct answer but not concise [enough],
2 mark for some information & for a right-lines approach,
1 marks for some basic understanding (or attempt).
Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s): Lecture 14: Input/Output (2).
TOTAL marks (5 marks) [10]
c) Given, in figure 3.c, that the program polls a simple device. In the context of
data exchange between CPU and peripherals:
Describe in detail exactly what happens when the following ARM program is
obeyed.
Clearly describe the movement of information (both numbers and instructions)
between the peripheral and the CPU, and how the values in the registers R0,
and R1 change at each step. Assume that the program starts at memory
location 0000 and the Status_Reg contains 0x80 while the Data_Reg contains
0x84 or ascii character ‘H.’ R0 and R1 should be the values after each [row]
instruction has been executed.

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
<th>R0</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>loop</td>
<td>ADR</td>
<td>R1, Status_Reg</td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>TST</td>
<td>R0, #0x80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000C</td>
<td>BEQ</td>
<td>loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>ADR</td>
<td>R1, Data_Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018</td>
<td>SVC</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001C</td>
<td>Status_Reg</td>
<td>DEFW</td>
<td>0x80</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>Data_Reg</td>
<td>DEFW</td>
<td>0x72</td>
<td></td>
</tr>
</tbody>
</table>

Question figure 3.c. A table; of program polls a simple device.
3.c. Application (example re-coding) (4 marks):
The following points should be covered to some degree in the answer:

Registers R0 and R1 are sequentially loaded with the following data:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
<th>R0</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>loop</td>
<td>ADR</td>
<td>R1, Status_Reg</td>
<td>- 0x001C</td>
</tr>
<tr>
<td>0x0004</td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td>0x80</td>
<td>0x001C</td>
</tr>
<tr>
<td>0x0008</td>
<td>TST</td>
<td>R0, #0x80</td>
<td>0x80</td>
<td>0x001C</td>
</tr>
<tr>
<td>0x000C</td>
<td>BEQ</td>
<td>loop</td>
<td>0x80</td>
<td>0x001C</td>
</tr>
<tr>
<td>0x0010</td>
<td>ADR</td>
<td>R1, Data_Reg</td>
<td>0x80</td>
<td>0x0020</td>
</tr>
<tr>
<td>0x0014</td>
<td>LDRB</td>
<td>R0, [R1]</td>
<td>0x72</td>
<td>0x0020</td>
</tr>
<tr>
<td>0x0018</td>
<td>SVC</td>
<td>2</td>
<td>0x72</td>
<td>0x0020</td>
</tr>
<tr>
<td>0x001C</td>
<td>Status_Reg</td>
<td>DEFW</td>
<td>0x80</td>
<td>-</td>
</tr>
<tr>
<td>0x0020</td>
<td>Data_Reg</td>
<td>DEFW</td>
<td>0x72</td>
<td>-</td>
</tr>
</tbody>
</table>

4 marks for majority of above; e.g. for an answer that orders all lines correctly (2 marks for a detailed and concise description),
3 marks for correct answer but not concise [enough],
2 mark for some information & for a right-lines approach,
1 marks for some basic understanding (or attempt).
Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s): Lecture 14: Input/Output (2).
TOTAL marks (4 marks) [14]

d) Given the code in 3.b. is now defined as ‘Status_Reg DEF 0x00’:

i) What would you expect to happen; when the program sequences through lines 0x0000 to 0x000C?
In your answer also state:
ii) what decision has the loop enabled the processor to make with respect to polling a simple device? And finally:
iii) Is the branch (BEQ) activated; and if so what caused the branch?

In your answers to i) to iii) please be explicit and comprehensive when you write the answers to each subsection. (6 marks)
3.d. Application (example) (Critique) (6 marks):

a) Once the Status_Reg [data] has been loaded (line 0x0004); the TST will set the zero status bit (flag); hence the BEQ (conditional) branch will be activated and the PC (R15) will be loaded with address 0x0000; i.e. the program will branch back to the loop labelled instruction [line].

b) The loop has enabled the processor to decide (or check) that the data_reg of the peripheral does not contain data as bit 7 in the status reg. is set to zero (=0). Hence, the basic decision is that the peripheral does not contain data in its data reg.

c) The initial cause of the sequence of events: Was the loading of R0 with the value (0x00) at address 0x001C; then when the TST is undertaken (which performs an “0x00 AND 0x80” operation, whose resultant would be 0x000) is sets the status register zero bit Z=1; hence when the BEQ is evaluated (as the Z=1; hence EQ is true; e.g. the result of the TST was 0x00 which set the Z bit (in the status register) to1) and the conditional branch was activated; and it branched the program back to the loop label (address 0x0000).

(4 mark)

6 marks for an answer that depicts all the salient facts in a sensible way; all keywords correctly delineated [in context] and aligned to appropriate step.
3 marks for correct answer but not all keywords aligned correctly,
2 mark for a right-lines approach; but more alignment errors,
1 mark for some basic understanding (or attempt).
Reference Learning Resources, Background Reading, and Lecture itself for detailed information; Lecture(s) No(s): Lecture 14: Input/Output (2).
TOTAL marks (6 marks) [20]