SECTION 1 : SOFTWARE (50 Marks Total)

Question 1. [20]

Describe briefly, in point form, the main differences and similarities in conventions for parameter passing and call/return for a function in C and Pascal. A reminder: in Pascal input parameters are pushed on the stack left to right, the output parameter is also on the stack, and the stack is cleaned by the called routine. In order to be clear, use along each step of your explanation the example function call in C:
int TraceMat (row_size,col_size,elements)
and its counterpart in Pascal: *TraceMat (row_size,col_size,elements) : int,*
where *row_size* and *col_size* are integers passed by value, *elements* is the address of a two-dimensional array containing the elements of a matrix. For each question, show also the corresponding Assembly language instructions.

(i) [5] describe how parameters are passed in C and give the equivalent Asm code.

(ii) [5] describe how parameters are passed in Pascal and give the equivalent Asm code.

(iii) [5] describe how the stack is cleaned in C and give the equivalent Asm code.

(iv) [5] describe how the stack is cleaned in Pascal and give the equivalent Asm code.

Question 2. [30]

(a) [3] Briefly define the function of an Assembler.

(b) [3] Briefly define the function of a Linker/Loader

(c) [3] Briefly define dynamically relocatable code.

(d) [12] Give the machine code, together with the extension word(s), if any, corresponding to the MC68000 Assembly language instuction below. Use the Appendix pages as needed.

MOVE.L 24(A4),D7

(e) [9] Give syntactically correct examples of instructions using ADD or MOVE as the instructions and including the following addressing modes :

(i) address register indirect + index + offset;

(ii) absolute;

(iii) immediate.

SECTION 2 : PROGRAMMING (60 Marks Total)

Question 3. [40]

DO NOT PANIC at the length of this question! You will be asked to read a problem description, then provide some code for parameter passing and accessing of the stack, and finally you will be asked to translate a small portion of pseudo code to Assembly language.

Consider the following declarations in C for a circular queue structure:

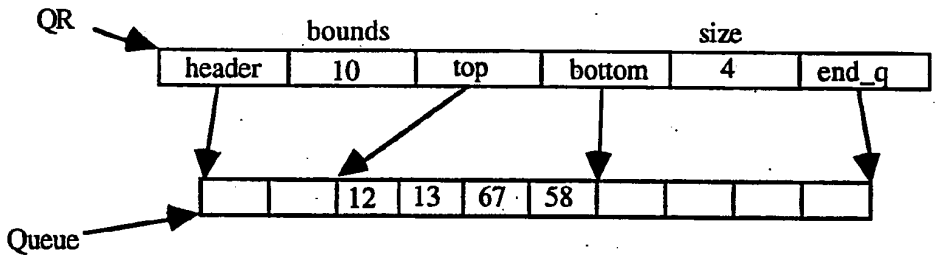
```
#define MAXBOUNDS 10
struct   Queue_Record {
         long      *header;        /*pointer to Queue itself*/
         long      bounds;         /*stores maximum size of Queue=MAXBOUNDS*/
         long      *top;           /*pointer to current 1st element in Queue*/
         long      *bottom;        /*pointer to current last element in Queue*/
         long      size;           /*current number of elements in Queue*/
         long      *end_q          /*pointer to end of Queue*/
};
struct   Queue_Record   QR;       /*a queue header*/
long     Queue [MAXBOUNDS];       /*the queue itself*/
long     item, okay;              /*element to be inserted in queue; flag*/
```

If you do not understand perfectly the C syntax, the following explanation and pictures should help. We want to store a Queue in an array (declared by "long Queue [MAXBOUNDS];"). We want to manipulate the Queue using a record with 6 fields (allocated by "struct Queue_Record QR;" and defined in the structure declaration above it), and the address of this record is in QR.
Four of these fields are pointers (addresses):

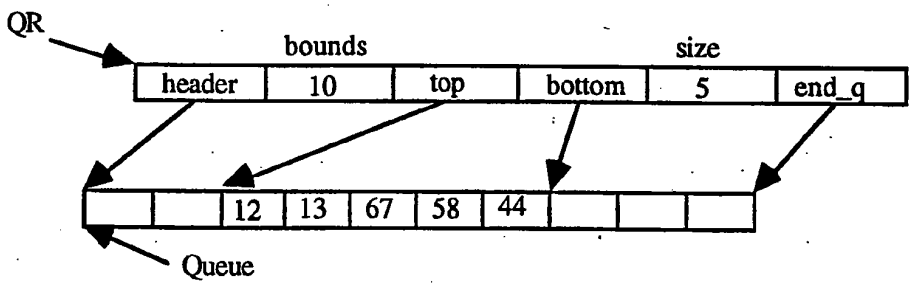
(i) "header" has the address of the *beginning* of the array for the Queue;
(ii) "top" has the address of the *beginning* of the first element currently in the Queue;
(iii) "bottom" has the address of the *end* of the last element currently in the Queue (that is, "bottom" points to the start of the memory location immediately after the last element currently in the queue);
(iv) "end_q" has the address of the *end* of the last element of the array for the Queue (that is, "end_q" points to the start of the memory location immediately after the last array element for the queue).

Thus "header" and "end_q" stay fixed, while "top" and "bottom" change after every insertion or deletion.
Moreover, the record contains two other fields, "bounds" and "size", which state the maximum number of elements which the Queue array can hold (does not change) and the current number of elements in the Queue (changes after insertion and deletion), respectively.

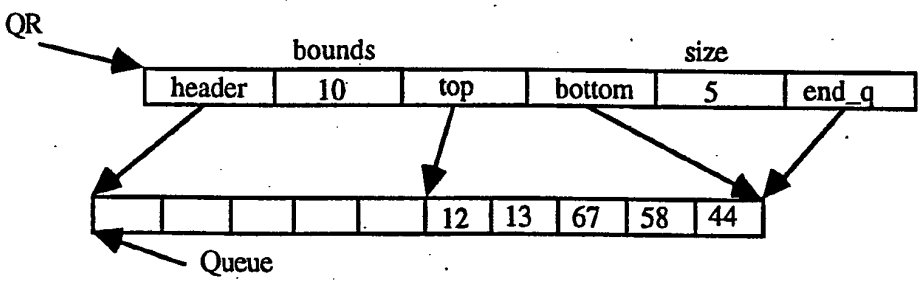
An instance of such a data structure might look like this:



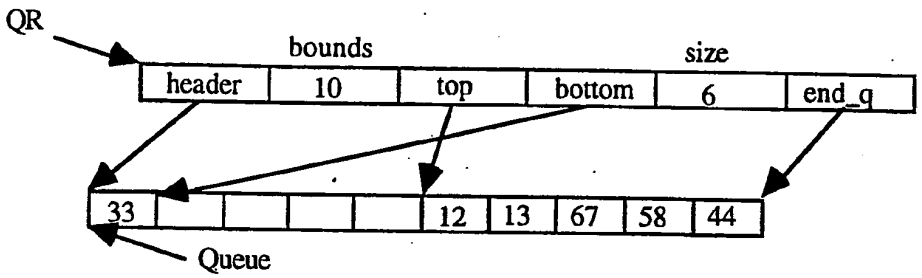
You are going to use this record declaration to implement a function to insert new elements in this FIFO queue. This is to be a circular queue, that is, a new element gets inserted at the end of the queue, if there is room left. However, if this would run off the end of the array, then the queue wraps around to the beginning of the array. For example, starting from the previous snapshop of a queue, a new insertion would cause the following changes:



However, if the initial snapshopt were as follows:



A new insertion would cause the following changes:



ENQUEUE is a function to insert elements in a Queue and it returns a flag indicating whether or not the insertion was successful. Given the call to *ENQUEUE* as in:

okay = ENQUEUE (QR,item);

(a) [5] show a picture of the stack at entry to *ENQUEUE* before any executable statement;

(b) [2] give the correct statements for LINK and MOVEM, assuming you are saving registers D1 to D7 and A0 to A4 locally, and you need no local storage;

(c) [5] show a picture of the stack after the execution of the statements in (b);

(d) [4] write the statements necessary to copy the parameters into local registers;

(e) [6] assign correctly to local registers copies of ALL the fields from Queue_Record and provide comments; [e.g. MOVE.L 4(A0),D2 would copy the content of field "bounds" into D2, assuming that A0 has been assigned the address QR]

(f) [18] write the rest of the code for *ENQUEUE* following the pseudo-code given (if you think some cases may be missing from the pseudo-code, ignore them).

```
PSEUDO_CODE for okay = ENQUEUE (QR,item);
temp_return = 0;
IF size = bounds, THEN temp_return = -1;      /*array full*/
ELSE
      IF bottom ≠ end_q  THEN                 /*enter at end of queue*/
            enter new element at bottom;
            update bottom;   update size;     /*update in record as well*/
      ELSE
            bottom = header;
            enter new element at bottom;
            update bottom;   update size;     /*update in record as well*/
return (temp_return);
```

5/8

Question 4. [20]

Assume you have two lists of integers, called L1 and L2. Each list contains five 32-bit signed integers, which you may assume have already been initialized. A third list, L3, is declared to contain five 32-bit signed integers, but has not been initialized. Subtract elements 1 to 5 of L1 from elements 5 to 1 of L2, respectively, and place the result in elements 1 to 5 of L3.
For example, if L1 contains -2,0,6,9,10 and L2 contains 5,6,11,33,64, then L3 should contain 66, 33, 5, -3, -5. You are to write the segment of code to accomplish the above task, using DBcc and predecrement and postincrement addressing as appropriate. Load the addresses of L1, L2 and L3 in A1, A2 and A3 respectively.

SECTION 3 : SOFTWARE/HARDWARE INTERACTION (40 Marks Total)

Question 5. [20]

Consider the following segment of code (machine code and Assembly is given), currently loaded at address 00002222_{16}:

(i)	D5AA	0002	ADD.L	D2,2(A2)
(ii)	C242		AND.W	D2,D1

(a) [2] Fetch the opcode of (i) into the IR. State the content of the IR and of the PC after the fetch.

IR =

PC =

(b) [2] State the number of extension words in instruction (i). State the new content of the PC if you have to fetch any extension word(s).

Number of ext. word(s) =

PC =

(c) [3] State in english the effect of instruction (i). Be precise.

(d) [3] For the instruction in (i), state:

Number of Read cycles during fetch and decode =

Number of Read cycles during execution =

Number of Write cycles during execution =

(e) [2] Fetch the opcode of (ii) into the IR. State the content of the IR and of the PC after the fetch.

IR =

PC =

(f) [2] State the number of extension words in instruction (ii). State the new content of the PC if you have to fetch any extension word(s).

Number of ext. word(s) =

PC =

(g) [3] State in english the effect of instruction (ii). Be precise.

(h) [3] For the instruction in (ii), state:

Number of Read cycles during fetch and decode =

Number of Read cycles during execution =

Number of Write cycles during execution =

Question 6. [20]

In the following questions you will step through the interrupt handling process for externally-generated interrupts. Base your answers on the assumption that you are using a 68000-style architecture.

(a) [4] If an external device wishes to indicate to the CPU that an interrupt needs servicing, how is the existence and priority of the interrupt communicated to the CPU?

(b) [2] What steps are taken by the CPU to determine if the interrupt should be handled *based on its priority*?

(c) [2] What signal values need to be set by the CPU as part of the interrupt acknowledgement (IACK) process?

(d) [6] What are the three possible categories of response to the IACK, and what signal lines are they returned on?

(e) [4] If, from the response to the IACK, the CPU calculates an exception number of 2 (i.e. a Bus error) what is the exception vector address, and how is the address of the exception handling routine determined?

(f) [2] Name two possible causes of a Bus Error.

SECTION 4 : ARCHITECTURE (50 Marks Total)

Question 7. [16]

(a) [4] Describe two distinguishing features of a RISC processor system.

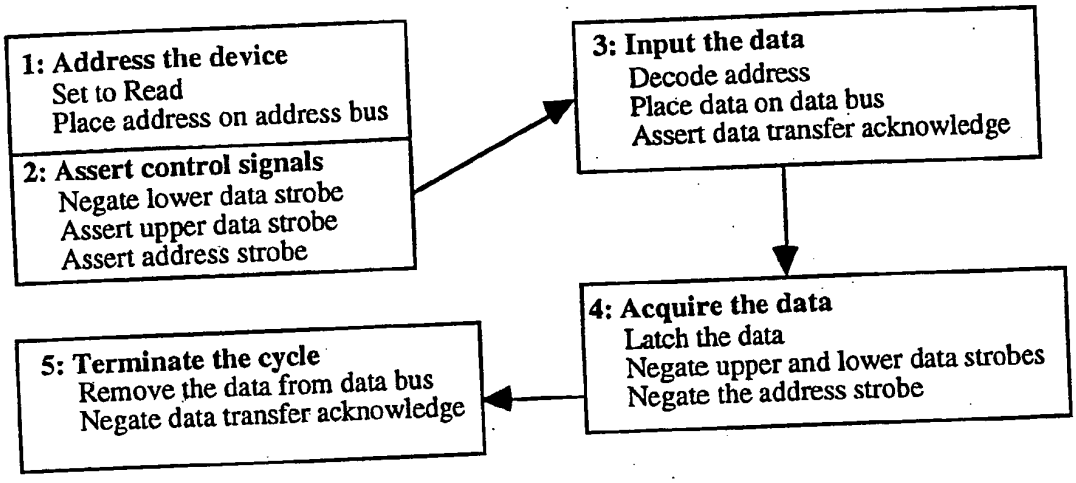
(b) [4] Describe two significant problems in the design or operation of systems with multiple processors.

(c) [4] Describe two significant differences between a typical microcomputer architecture and a typical mainframe architecture.

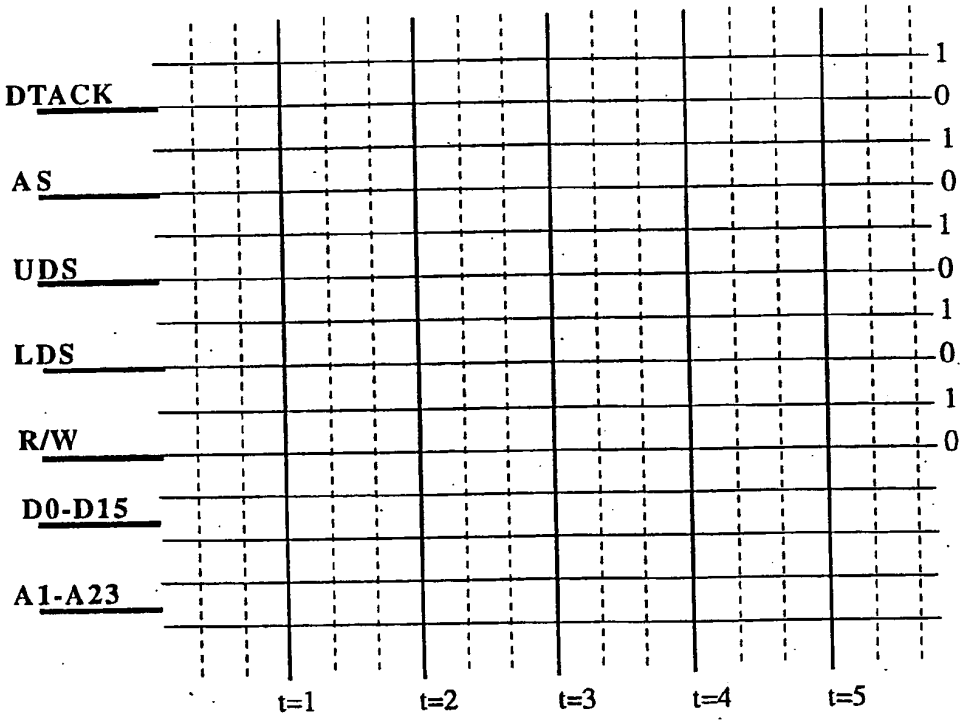
(d) [4] Describe the major differences between the use of memory mapped I/O and isolated I/O for the use of I/O peripheral devices.

Question 8. [14]

The chart below describes a 5-step byte read protocol for a 68000-style architecture. Fill in the timing diagram below the chart to show the relative values (high or low) on the indicated signal time lines.



Timing Diagram



Question 9. [10]

In order to reduce hardware costs, some architectures have the address bus and data bus share a single physical bus. At any given time the physical bus can operate either as the address bus or as the data bus (but of course not as both at the same time).

List briefly four significant points with such an architectural arrangement (points might reflect advantages, disadvantages, design complications, etc.).

Question 10. [10]

Suppose we have a computer system with the following characteristics:
1) A 16-bit (64kbyte) logical address space.
2) A 23-bit (8 mbyte) physical address space.
3) Both logical and physical memory is regarded as being divided into pages (blocks) of 8kbytes each

You are to show how a memory mapping unit might be used to utilize the large physical memory.

(a) [6] Provide a block diagram showing the general placement and connection of the CPU, memory, MMU, control bus, address bus, and data bus. Show the size of the address bus and the size of the data bus.

(b) [4] Give a small example showing how the logical-to-physical address translation map (or table) might work.

Appendix

MOVE — Move Data from Source to Destination

Operation : Source → Destination

Assembler Syntax : MOVE <ea>, <ea>

Attributes : Size = (Byte, Word, Long)

Description : Moves the content of the source to the destination location. The data is examined as it is moved, and the condition codes set accordingly. The size of the operation may be specified to be byte, word, or long.

Condition Codes :

X	N	Z	V	C
-	*	*	0	0

N Set if the result is negative. Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.
X Not affected.

Instruction Format :

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
0	0	Size	Destination Register	Destination Mode	Source Mode	Source Register

Instruction Fields :

Size field - Specifies the size of the operand to be moved:

01 - byte operation
11 - word operation
10 - long operation

Destination Effective Address field - Specifies the destination location. Only data alterable addressing modes are allowed as shown :

Addr. Mode	Mode	Register
Dn	000	reg. number:An
An	-	-
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addr. Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	-	-
(d_{16},PC)	-	-
(d_8,PC,Xn)	-	-

Source Effective Address field - Specifies the source operand. All addressing modes are allowed as shown:

Addr. Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addr. Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d16,PC)	111	010
(d8,PC,Xn)	111	011

* For byte size operation, address register direct is not allowed.

Notes : 1. MOVEA is used when the destination is an address register. Most assemblers automatically make this distinction.

2. MOVEQ can also be used for certain operations on data registers.