Question 1: [27 marks]

[5] (a) Draw the block diagram of the main components of a computer system using memory mapped I/O. Make sure that you label all blocks and interconnecting paths.
[5] (b) Draw the block diagram of the main components of a computer system using isolated I/O. Make sure that you label all blocks and interconnecting paths.
[5] (c) State the main differences between (a) and (b) in functionality and programming.
[8] (d) What is an MMU? (i.e. what does it do). State the advantages of having one. Is it generally a fixed hardware unit, or is it programmable and changeable? If it is programmable, who can program it and how? Is it possible to have an MMU in systems whether they use memory-mapped I/O or isolated I/O? If so, (for both or either setup), state where the MMU is placed.
[4] (d) What is a DMA channel? Is it possible to have a DMA in systems whether they use memory-mapped I/O or isolated I/O? If so, (for both or either setup), state where the DMA is placed.

Question 2: [12 marks]

The smallest unit the MSERRA-230 CPU is capable of addressing is a byte, and it can deal directly with integers (words) up to 24 bits. Furthermore, it has a 20-bit address bus.
[3] (a) Classify the MSERRA-230 by its type of addressing.
[3] (b) What is the data bus width? Classify the MSERRA-230 according to its word length.
[3] (c) Specify the memory address range and the maximum addressable memory size for the MSERRA-230 (use kilobytes or megabytes for the latter).
[3] (d) The next version of MSERRA-230 needs to be able to address twice as much memory. What are the modifications which must be made?

Question 3: [23 marks]

[3] (a) A compiler translates high-level language code to the native code of the machine. This translation may be directly to machine code or to assembly language. The latter serves as intermediate code requiring an additional translation to machine code. Give some reasons why a translation to assembly language may be done.
[6] (b) Why do most assemblers make two passes? Explain in point form the major steps for each pass.
[6] (c) Define the symbol table and describe its function. Give an example of its use (possibly by showing a segment of code).
[6] (d) Define static and dynamic position independent code.
[2] (e) Define the differences between absolute and relocatable quantities.

Question 4: [17 marks]

A small segment of code has been loaded at memory address $000500 and it appears as in the table below. It is an example of "self-modifying" code. As you notice, for two of the lines of code you have already been given the corresponding Assembly language statement.

[1] (a) Explain what self-modifying code is.
[2] (b) State in English the precise effect of the instruction in line 1. Moreover, show the effect on memory and registers after execution by making a copy of memory [address,content] and showing the "before and after" for each of your answers.
[4] (c) Give the Assembly language equivalent to the machine code in line 2 (use the Appendix).
[2] (d) State the effect on memory and registers after execution of line 2.

[2] (e) State the effect on memory and registers after execution of line 3.
[4] (f) Give the Assembly language equivalent to the machine code in line 4 (use the Appendix).
[2] (g) State the effect on memory and registers after execution of line 4.

Line	*Address*	*Content*	*Asm*
1	000500	3039 0000 0770	move.w $000770,D0
2	000506	D079 0000 0772	*to be decoded*
3	00050C	33C0 0000 0512	move.w D0,$000512
4	000512	0000 1007 0000 0774	*to be decoded*
	.....		
	000770	3300	
	000772	00FC	
	000774	0007	

Question 5 : [30 marks]

(a)[3] What is an interrupt?
(b [8] State the major steps in general interrupt processing.
(c) [2] State the difference between an internal and an external exception.
(d) [2] Explain briefly the necessity for "levels" of interrupts.
(e) [9] A subset of an exception vector table is shown below. Assume that an external interrupt signal has reached the CPU, giving an interrupt priority of 4 and an interrupt vector number of 3. Describe precisely, in point form, what happens next, and what happens at the end of the exception processing.
(f) [6] After an exception has been processed, it may be the case that (i) normal processing can resume within the application, or (ii) that the application is aborted and there is a return to the operating system or (iii) even that the whole system is aborted and it must be rebooted from scratch. Give examples of exceptions which fall in each of the three cases.

Exception Vector Table	
V. address	Type
000	address of supervisor stack pointer (for RESET)
004	address of PC (for RESET)
008	bus error
00C	address error
010	un-implemented instruction
014	division by zero
018	CHK

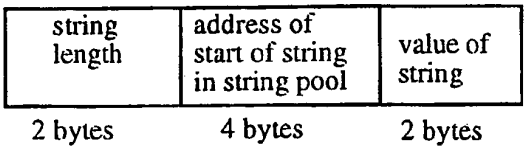
Question 6: [26 marks]
[6] (a) Compose three instructions, using only MOVE and/or ADD, each using at least one of the following addressing modes: "address register indirect plus offset and index", a "address register indirect with pre-decrement", "immediate". Each instruction must be complete and syntactically valid in MC68000 assembly language.
[6] (b) For each instruction produced in (a), state the addressing mode of each operand.
[6] (c) For each instruction produced in (a), state its precise effect in English.
[8] (d) For the instruction below, give the precise sequence of events in the "fetch-decode-execute" cycle. That is, state what happens at every step, how many "reads" and "writes" to/from memory during fetch and during execute, the ultimate effects on registers and/or memory, etc. Be precise about what goes on the bus and when.

```
ADDI.W      #26,-4(A2)
```

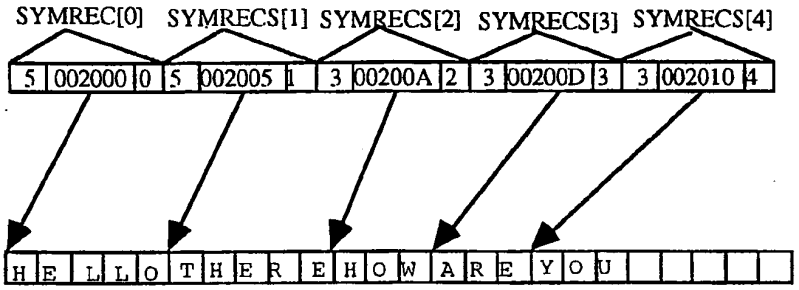
Question 7: [35 marks]

3

An application programs deals with the storage of variable length character strings. The strings are stored in a "pool", which is simply an array of bytes. In order to know where each character string is placed and its length, an array of records, called SYMRECS, is maintained for the "string pool access". Each record in the array has the following 3 fields:



Thus a typical snapshot of the system after a few insertions might look like this:



(a) Write an insert routine in assembly language which implements the following pseudo-code:

```
READ (N);
Value := 0;
FOR I := 1 TO N DO
        Read (char-string) and place into string pool;
        Place length into SYMRECS[I];
        Place address into SYMRECS[I];
        Place value into SYMRECS[I];
        Value := Value +1;
END FOR
```

Use the appropriate library routines in MAS BasePak for the input and string manipulation (DECIN, GETSTR, STRLENGTH, STRCOPY - see also the Appendix). Document the code well. Assume that the address of SYMRECS has been placed already as a parameter in A1 and the address of the string pool has been placed in A2. Restore all registers on exit. Moreover, assume you can declare : BUFFER DS.B 80 to be used by GETSTR to read a string. Additional requirements: you must incorporate the use of DBcc for loops and "address register indirect plus offset plus index" for the array of records.

Question 8: [30 marks]

4

You must set up the templates in Assembly Language for calling and returning from functions in both a THINK Pascal program and in a THINK C program. The conventions are similar yet different between the two languages. Some details are the same as :

- All parameters are to be passed through the stack.
- All input parameters are to be passed by reference (i.e. their address).
- Integers should be implemented as 16-bit two's complement numbers.
- Storage is also created on the stack for whatever local variables are declared inside the function, using the LINK instruction.

The differences are stated below and you must provide a template for <u>both</u> calling conventions. Each template must include :

- for the calling program:
  - (a)[2] the instructions for pushing all parameters on the stack
  - (b)[1] the calling instruction
  - (c)[1] a picture of the stack after execution of (a) and (b)
  - (d) [1] a picture of the stack before (e) and after return from the called function
  - (e)[1] the instructions, after the function returns, to retrieve the result
- in the called function :
  - (f)[1] the instructions to allocate and deallocate local storage on the stack
  - (g) [1] a picture of the stack after execution of (f)
  - (h) [1] the instructions to save and restore local registers
  - (i) [2] the instructions inside the function to copy the parameters from the stack into local registers
- (j) [2] the instructions to clean up the stack wherever appropriate (in caller or function - state precisely)
- (k) [2] the instructions to return from the called function.

Notes : (j), (k), and (e) may interact depending on the template. Make sure that you comment your work well so that I can evaluate your logic and understanding.

TEMPLATE 1 (THINK Pascal Calling Conventions).
The caller pushes the arguments in left-to-right order, then calls the function. Upon return, the result may be found on the stack. Space for the result is made on the stack <u>before</u> pushing the other arguments. The function stores its return result (if any) on the stack in the location reserved by the caller. It is the function's responsibility to clean the stack.

TEMPLATE 2 (THINK C Calling Conventions).
The caller pushes the arguments in right-to left order, then calls the function. When the function returns, it is the caller's responsibility to remove the arguments from the stack. After pushing the arguments, but before issuing the actual call, the caller pushes the address of the location where the return value is to be placed. The function must copy this address and store the result at the location it points to. The address is considered a hidden argument to the function and it is the caller's responsibility to remove it from the stack.

Use the pseudo-code below for a general function for the two templates. (Note: do not mix the "formal" and "actual" parameters).

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			Mode			Source Mode			Register		

```
FUNCTION TEST (Var R,S,T : INTEGER) : INTEGER;
VAR A,B : INTEGER; {locals to function TEST}
BEGIN
      .....
      code
TEST:= return value;  {Pascal syntax for return value}
return (whatever) ;   {C syntax for return value}
END;
```

In the high level language, the caller has:

```
Q := TEST ( A,B,C)
```

HAVE A GOOD SUMMER !

B6

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.

ADD

Add Data from Source to Destination

Operation : Source + Destination → Destination

Assembler Syntax :
ADD <ea>, Dn
ADD Dn, <ea>

Attributes : Size = (Byte, Word, Long)

Description : Adds the source operand to the destination operand using binary addition, and stores the result in the destination location. The size of the operation may be specified as byte, word, or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes :

X	N	Z	V	C
*	*	*	*	*

- **N** Set if result is negative. Cleared otherwise.
- **Z** Set if result is zero. Cleared otherwise.
- **V** Set if an overflow is generated. Cleared otherwise.
- **C** Set if a carry is generated. Cleared otherwise.
- **X** Set the same as the carry bit..

Instruction Format :

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	Register	Opmode	Effective Address Mode	Register

Instruction Fields :

Register field - Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	<ea> + <Dn> → <Dn>
100	101	110	<Dn> + <ea> → <Dn>

Effective Address field - Determines addressing mode:

a. If the location specified is a 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
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

b. If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addr. Mode	Mode	Register
Dn	-	-
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>	-	-
(d16,PC)	-	-
(d8,PC,Xn)	-	-

String Input : GETSTR (S)

Parameters passed
The address of the input string S is in A0
Parameters returned
None
Function
Reads characters from the keyborad into the buffer pointed to by A0 and then appends a NULL character. The read terminates on carriage return, which is then deleted from the input string.

String length : STRLENGTH (S)

Parameters passed
Address of string S on the stack
Parameters returned
The length of the string in D0.W
Function
Returns the length of a string S.

String Copy: STRCOPY (S,T)

Parameters passed
The address of string S on the stack
The address of string T on the stack
Assume the two string do not overlap in memory
Parameters returned
None
Function
Copies string T to string S.

Decimal Input : DECIN

Parameters passed
None
Parameters returned
D0.W contains the 16-bit number read
Function
A 16-bit signed decimal number is read from the keyborad and stored in D0.W