Write your answers to the special response sheet you received (with your name and photograph). If you are using more than single sheet of paper for your answers, then mark each sheet with its number / total number of sheets you will hand over.

Task 1

Assume you have a computer with the original 8-bit ISA bus (that supports 20-bit memory address space, 16-bit I/O address space). It is a parallel bus without support for burst transfers, and its clock frequency is 8 MHz.

We are designing a part of a 10BASE-T network interface card controller that will be responsible only for receiving network packets (completely independent packet sending part will be designed later). The controller has a 12144 B long built-in SRAM buffer designed as a cyclic queue for temporary storage of Ethernet frames (assume max frame length of 1518 bytes). Upon receipt of every complete frame the controller should issue an interrupt request. The controller should also support DMA bus mastering without scatter/gather I/O support as the only way to allow transfers of saved frame data from the controller's buffer into computer's main memory.

Your task is to design and describe an HCI for such a controller, so that its device driver can command it to transfer data at least in units of whole packets. If the device driver is not able to "download" the packet data fast enough, the controller is allowed to overwrite the oldest packet data with ones just received over the network. You can use only I/O addresses in range `$1000` to `$2000` and IRQs 10 to 12 for your HCI design.

Task 2

Assume variable x has all bits cleared to 0, with the exception of bits 0 and 4 that are set to 1. Write down value of the x variable in hexadecimal format after execution of the following Pascal code:

```
x := x + (
($AABBCCDDEEFF0011 or
($0005500000000000 shl 8)) and (not(1) xor
65536));
```

Task 3

One of the goals for a higher programming language is to shield programmers from specifics of concrete target processor architecture. Having this in mind, is it ever necessary to know the *endianness* of a target processor when developing code in a higher programming language (e.g. in Pascal)? If no, explain why. If yes, explain on an example why.

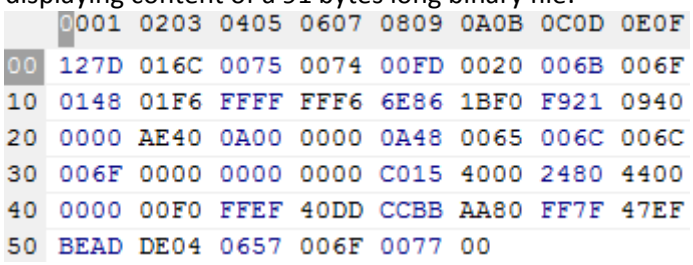
Task 4

The Unicode encoding defines the following assignment of codes to given characters: code 10Ch for char 'Č', code 61h for char 'a', code 73h for char 's'.

We would like to store text "Čas" (without the quotes) as a null-terminated string in little endian UTF-16 encoding beginning at address 0. Using a hexadecimal format write values of all bytes of memory (starting with byte at address 0) representing the string given above in the given encoding.

Task 5

The following figure shows a screenshot of a hex editor displaying content of a 91 bytes long binary file:



	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F
00	127D	016C	0075	0074	00FD	0020	006B	006F
10	0148	01F6	FFFF	FFF6	6E86	1BF0	F921	0940
20	0000	AE40	0A00	0000	0A48	0065	006C	006C
30	006F	0000	0000	0000	C015	4000	2480	4400
40	0000	00F0	FFEF	40DD	CCBB	AA80	FF7F	47EF
50	BEAD	DE04	0657	006F	0077	00		

All the data stored in the file are written in little endian ordering. Beginning byte number 51 (counting from 0) a 64-bit floating point real number in the IEEE 754 double format is stored in the file (IEEE 754 doubles have a normalized mantissa [significand] with hidden 1 occupying lowest 52 bits of the value, followed by 11 bit exponent in bias +1023 format, and a single sign bit). In the decimal system write down value of the number defined above.

Task 6

Assume that into an OS we are implementing a mechanism for sharing physical memory among multiple processes all using a specific DLL library – we are targeting 32-bit CPUs with paging support. The sharing mechanism must be transparent for all the processes involved, and cannot break the virtual address space separation between processes. Thus for all the pages containing the DLL code, we need to enforce no such process is able to write into them (we explicitly do not support self-modifying code, any such attempt should result in target thread termination). Furthermore, for all the pages containing DLL global data so called *copy-on-write* mechanism has to be implemented in the OS (your task): as long as all processes are only reading from pages containing DLL's global data, all such processes should share a single physical frame for these data. However, whenever a process tries to write into such a share page, a private copy of the page needs to be allocated into a new frame (that is not shared with any other processes). The process's write attempt must succeed, but it has to be directed into the newly allocated private frame. Design such a sharing mechanism and in Pascal (pseudocode) implement algorithm for the page fault handler (for simplicity assume no "sharing unrelated" page faults can occur). Assume the target CPU page table entries contain all the typical information describing the mapping, and that every page table entry has 4 bytes of spare space that you can use for any purpose.

Task 7

Describe and explain how programs written Java or C# are commonly compiled and executed. Include explanation of the *intermediate language* term, and explain its purpose and advantages in this context.

Task 8

Assume a computer with a simplified variant of 32-bit ***big endian*** CPU Motorola 68000. The processor has the following ***registers:***

8 general purpose data registers D0 to D7 – they can be used only as a direct source value or target for instructions;
8 general purpose address registers A0 to A7 – they can be written only via special instructions or operand variants, in regular instructions the address registers can be used only as an *address* operand. Register A7 is commonly used as a *stack pointer* (assume a common organization of the call stack).
The CPU contains also a 32-bit register PC and a 16-bit status register CCR (with all common flags).

Instruction set: Most instructions have 32-bit (`.l` suffix in assembler), 16-bit (`.w` suffix), and 8-bit (`.b` suffix) variants. The processor supports these basic instructions (*<op>* = any operand variant, see below, A*n* = any of the A0 to A7 registers, D*n* = any of the D0 to D7 registers, the rightmost operand is the target):

Instr.	Operands	Operand sizes	Description
MOVE	*<op>*,*<op>*	8, 16, 32	Value copy src→dest
MOVEA	*<op>*,A*n*	32	Copy to addr. reg.
ADD	D*n*,*<op>* *<op>*,D*n*	8, 16, 32	Adding of/into data register
ADDA	*<op>*,A*n*	32	Adding into addr. reg.
SUB	D*n*,*<op>* *<op>*,D*n*	8, 16, 32	Subtraction
SUBA	*<op>*,A*n*	32	Subtraction of addr. r.
JSR	*<op>*	32	Call subroutine
RTS	*None*	*None*	Return from subrout.

Operands: The *<op>* symbol represents any of the following variants for operands (written in typical Motorola 68000 assembler syntax):

- `#`*imm* immediate value
- D*n* operation with D*n* register
- (A*n*) mem operation with address given by A*n*
- *imm*(A*n*) mem operation, target address is sum of A*n* register value and the *imm* value.
- -(A*n*) predecrement: A*n* register value is decremented by size of the operation by bytes, and the new value of A*n* is used as the target address of the op.
- (A*n*)+ postincrement: current value of A*n* is used as the target address of the operation, followed by autoincrementing the A*n* register value by size of the operation

Program example: Assume register A0 contains `0x00100000`, register D1 contains `0xFFFFFFFF`, and memory from address `0x00100000` contains bytes: `00 00 00 05 00 00 00 00 00 00 00 03`, then after:

```
move.l #7,(a0)+    { copy 32-bit value 7 to 32-bit value at
                     address given by content of A0, and increment A0 by 4 }
add.w 6(a0),d1     { increment lowest 16-bits of D1
                     register by 16-bit value stored at address A0 + 6 }
adda.l #4,a0       { increment address in A0 by 4 }
```

register A0 will contain `0x00100008`, D1 will contain `0xFFFF0002`, and memory at `0x00100000` will contain bytes: `00 00 00 07 00 00 00 00 00 00 00 03`

Actual task: Rewrite the following Pascal program body into an equivalent Motorola 68000 assembler program using a common Motorola calling convention (arguments passed on stack, stored from right to left, return value stored in the D0 register) and only using the instructions described above (assume size of `Integer` type is 16 bits, and variable `Result` begins at address `0x00010000`):

```
var Result : Integer;
function Sum(a, b, c : Integer) : Integer;
{ main program body }
begin
    Result := 10;
    Result := Result + Sum(1, 2, 3);
end.
```

Task 9

Rewrite the following `Sort` procedure, so that on a two processor system it could finish in a nontrivially shorter time than on a single processor system:

```
type PLongint = ^Longint;
procedure Quicksort( { not using glob.vars }
    array : PLongint; n : Longint);
procedure Merge(   { not using global vars }
    source1 : PLongint; source2 : PLongint;
    n1, n2 : Longint; target : PLongint);
{ n >= 1 }
procedure Sort(source : PLongint;
    target : PLongint; n : Longint);
var
    half : Longint;
begin
    half := n div 2;
    Quicksort(source, half);
    Quicksort(source + half, n - half);
    Merge(source, source + half,
          half, n - half, target);
end;
```

Write declarations of all multithreading related procedures and functions you require from the target OS. Include a short description of their expected behavior for each.

Task 10

Assume MS-DOS OS running on Intel 8088 CPU (16-bit CPU, gen. purp. registers AX, BX, CX, DX, 16 & 16-bit [`seg:ofs`] logical addresses, 20-bit physical address = `seg*16+ofs`, a PC represented as two 16-bit registers `CS:IP`). Using a command line debugger we have created the following machine code of correct and tested function (its only argument and return value are stored in the BX register):

```
-u 5000:0
5000:0000 B80000          MOV     AX,0000
5000:0003 39C3            CMP     BX,AX
5000:0005 7403            JZ      000A
5000:0007 BB0500          MOV     BX,0005
5000:000A 83C302          ADD     BX,+02
5000:000D C3              RET
```

We have stored this machine code into a 14 bytes long file as a direct binary image of memory from address `5000:0000`. After restarting the computer we will reload the image at address `5000:01FF` Will the function still work and behave in the same way as before? Explain why!