Odpovědi pište na zvláštní odpovědní list s vaším jménem a fotografií. Pokud budete odevzdávat více než jeden list s řešením, tak se na 2. a další listy nezapomeňte podepsat. Do zápatí všech listů vždy napište i/N (kde i je číslo listu, N je celkový počet odevzdaných listů).

Společná část pro otázky označené X

Předpokládejte, že máme původní 8-bitovou sběrnicí ISA (tj. podpora pro 20-bitový paměťový adresový prostor, a 16-bitový I/O adresový prostor). Sběrnice je paralelní, nemá podporu pro burst přenosy a je taktována na 8 MHz.

Otázka č. 1 (X)

Předpokládejte, že chceme navrhnout část řadiče sítové karty pro připojení k 10BASE-T síti. Navrhovaná část bude zodpovědná pouze za příjem paketů (zcela nezávislou část pro odesílání paketů budeme řešit později). Řadič bude mít v sobě zabudován 6072 B buffer implementovaný pamětí SRAM, který bude sloužit jako cyklická fronta pro příjem Ethernetových rámců (maximální délka rámce 1518 bytů). Při příjmu každého kompletního paketu má řadič vyvolat žádost o přerušení. Dále má řadič podporovat mechanizmus DMA bus mastering ***s podporou*** scatter/gather I/O jako jediný způsob přenosu uložených paketů z interního bufferu řadiče do hlavní paměti počítače.

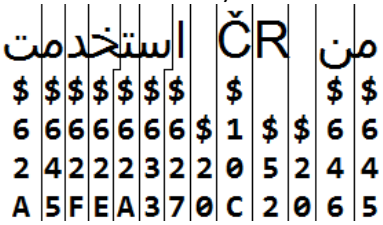
Navrhněte a detailně popište HCI pro tuto síťovou kartu tak, aby si její ovladač mohl vždy vyžádat přenos minimálně jednoho celého paketu. Počítejte s tím, že pokud ovladač nestíhá „stahovat“ data paketů ze síťové karty, tak korektní chování řadiče je, že začne nejstarší pakety v interním bufferu přepisovat pakety nejnověji přijatými. Pro HCI máte k dispozici pouze I/O adresy z následujícího rozsahu $4000 až $4500, a pouze IRQ 11.

Otázka č. 2 (X)

Pro uvedenou sběrnici navrhněte novou transakci pro zápis 1 slova do paměti pomocí DMA bus masteringu (bez potřeby dedikovaného DMA řadiče). Pro tuto transakci nakreslete její časový diagram (skupiny adresových a datových vodičů kreslete jako jeden „seskupený“ signál). Pro každý takt takové transakce napište jeho stručný popis (co v něm kdo dělá).

Otázka č. 3

Následující obrázek zobrazuje v horní části část věty v arabštině (čte se zprava doleva, prostřední slovo je v latince a čte se zleva doprava), navíc je pod každým písmenem/znakem zapsán jeho kód v Unicode v šestnáctkové soustavě (číslo čtěte shora dolů):



Tento text uložíme do textového souboru v kódování UTF-16 big endian. Zapište obsah jednotlivých bytů tohoto souboru v šestnáctkové soustavě (v pořadí od 0. do posledního bytu).

Otázka č. 4

Odpozorovali jsme, že když zapneme počítač Atari 800XE, tak se po několika sekundách „bručení“ dostane do stavu, kdy jsme schopni psát příkazy v jazyce BASIC, které počítač provádí. Pokud ale před zapnutím počítače zasuneme do konektoru v jeho zadní části (do kterého je vyvedena systémová sběrnice) tzv. cartridge „se hrou“ (malá krabička, do které nevidíme), a pak teprve počítač zapneme, tak na počítači začne běžet daná hra (a nikoliv prostředí BASICu). Detailně popište a vysvětlete, co se v těchto dvou situacích v počítači děje, kdo a kdy provádí jaký kód, a zároveň vysvětlete, jak je možné, že při zasunuté cartridge „magicky“ nedojde k běhu prostředí BASICu, ale běží hra z cartridge. Do vysvětlení zahrňte i to, z jakých součástek se asi cartridge uvnitř skládá.

Otázka č. 5

Vlastníme dvě zařízení s OS Android: tablet ASUS MeMO Pad 7 (ME176C) s procesorem Intel Atom Z3745 quad-core, 1,86 GHz (architektura x86) s nainstalovaným Androidem 4.4, a americkou verzi telefonu Samsung Galaxy S III Mini s procesorem Qualcomm Snapdragon 400 dual-core 1,2 GHz (architektura ARM) s nainstalovaným Androidem 4.2. Na obou těchto zařízeních jsme byli schopni z obchodu Google Play stáhnout stejný binární spustitelný soubor (`.apk`) nějaké aplikace, a na obou zařízeních ho přímo z shellu Androidu spustit, přestože by to na první pohled nemělo být možné. Popište a vysvětlete, v čem je tento potenciální problém, a dále vysvětlete, proč nakonec v popsané situaci nenastává.

Otázka č. 6

Do jádra operačního systému s podporou pro preemptivní přepínání vláken a pouze jednoprocesorové systémy chceme naimplementovat podporu pro takzvané APC volání (Asynchronous Procedure Call), tj. možnost donutit nějaké vlákno v libovolném stavu vykonávat nějakou uživatelsky danou proceduru. Pro takový systém v Pascalu naprogramujte následující API funkci `CallAPC`:

```
type
    PProc = procedure (param : Longword);

procedure CallAPC(
    tid : Longword; procPtr : PProc;
    param : Longword
);
```

kde parametr `tid` identifikuje cílové vlákno, ve kterém má být zavolána procedura, na kterou ukazuje parametr `procPtr` (proceduře má být v cílovém vlákně předán parametr `param`). Procedura má být v cílovém vlákně zavolána co nejdříve. Pokud cílové vlákno spí, tak má být procedura provedena až po jeho vzbuzení po splnění původní podmínky probuzení.

Otázka č. 7

Předpokládejte, že máme počítač se zjednodušenou variantou 32-bitového ***big-endian*** procesoru Motorola 68000. Tento procesor má následující ***registry:***
8 obecných registrů D0 až D7 – lze je použít pouze jako přímou zdrojovou nebo cílovou hodnotu nějaké operace;
8 obecných tzv. adresových registrů A0 až A7 – pro jejich zápis je potřeba použít speciální instrukce, v běžných instrukcích je lze použít pouze jako operand typu *adresa*. Registr A7 se běžně používá jako *stack pointer* (předpokládejte typickou organizaci volacího zásobníku).
Dále má procesor 32-bitový registr PC a 16-bitový registr stavu CCR (obsahující všechny běžné příznaky).
Instrukční sada: Většina instrukcí má 32-bit (přípona `.l` v assembleru), 16-bit (přípona `.w`), i 8-bit (přípona `.b`) variantu dané operace. Procesor má mimo jiné následující instrukce (*<op>* = libovolná varianta operandu, viz dále, A*n* = libovolný z registrů A0 až A7, D*n* = libovolný z registrů D0 až D7, cílový je vždy nejpravější operand):

Instr.	Operandy	Velikost operandů	Popis
MOVE	*<op>*,*<op>*	8, 16, 32	kopie hodnoty zdr→cíl
MOVEA	*<op>*,A*n*	32	kopie do adr. registru
ADD	D*n*,*<op>* *<op>*,D*n*	8, 16, 32	přičtení datového nebo k datovému reg.
ADDA	*<op>*,A*n*	32	přičtení k adr. registru
SUB	D*n*,*<op>* *<op>*,D*n*	8, 16, 32	odečtení hodnoty
SUBA	*<op>*,A*n*	32	odečtení od adr. reg.
JSR	*<op>*	32	volání podprogramu
RTS	*žádné*	*žádné*	návrat z podprogramu
BEQ BNE	*<op>* *<op>*	32 32	podmíněný skok: Branch if EQual Branch if Not Equal

Operandy: Za *<op>* je možno dosadit libovolnou z následujících variant operandů (zapisováno v syntaxi běžného Motorola 68000 assembleru):

- #*imm* hodnota immediate
- D*n* operace s obsahem registru D*n*
- (A*n*) operace s pamětí daná obsahem registru A*n*
- *imm*(A*n*) operace s pamětí, cílová adresa je daná součtem obsahu registru A*n* a hodnoty *imm*
- -(A*n*) tzv. predekrementace: jako součást instrukce je hodnota v registru A*n* zmenšena o velikost prováděné operace v bytech, a nová hodnota v registru A*n* slouží jako cílová adresa
- (A*n*)+ tzv. postinkrementace: aktuální hodnota registru A*n* slouží jako cílová adresa, po provedení instrukce je ale hodnota v registru A*n* automaticky zvětšena o velikost provedené operace v bytech

Příklad programu: Pokud je v registru A0 hodnota `0x00100000`, a v registru D1 hodnota `0xFFFFFFFF`, a od adresy `0x00100000` jsou v paměti následující byty:
`00 00 00 05 00 00 00 00 00 00 00 03`, potom po:

```
move.l #7,(a0)+   { nastavení 32-bit hodnoty 7 do 32-bit hodnoty na adrese dané reg. A0, a zvětšení obsahu A0 o 4 }
add.w 6(a0),d1    { zvětšení spodních 16-bitů registru D1 o 16-bitovou hodnotu na adrese A0 + 6 }
adda.l #4,a0      { zvětšení adresy v A0 o 4 }
```

bude v registru A0 hodnota `0x00100008`, v D1 `0xFFFF0002`, a v paměti od adresy `0x00100000` bude:
`00 00 00 07 00 00 00 00 00 00 00 03`
Úloha:
Víme, že od adresy `01AA19E0h` je v paměti uložen kód funkce `DoSomeMagic`, která má dva 32-bitové unsigned parametry, a vrací 32-bitovou unsigned hodnotu. Funkce používá variantu Cčkové volací konvence (parametry se předávají na zásobníku zprava doleva, parametry odstraňuje volající, návratová hodnota se ukládá do registru D0). Po spuštění disassembleru na adresu `01AA19E0h` jsme zjistili, že kód funkce `DoSomeMagic` je následující:

```
01AA19E0 move.l  $8(a7), d0
01AA19E6 sub.l   $4(a7), d0
01AA19EC beq     #$01AA1A08
01AA1A02 sub.l   #1, d0
01AA1A08 move.l  d0, -(a7)
01AA1A0C move.l  $C(a7), d0
01AA1A12 add.l   $8(a7), d0
01AA1A18 sub.l   (a7)+, d0
01AA1A1C rts
```

Zapište v Pascalu její kód i její kompletní deklaraci bez použití inline assembleru. Jména proměnných a parametrů, která nejsou z disassemblovaného kódu zřejmá, si vhodně zvolte. Předpokládejte, že typ `Longword` je 32-bitové celé číslo bez znaménka.

Otázka č. 8

Víme, že v 64-bit proměnné `x` jsou všechny bity rovné `0`, pouze bit `0` a bit `4` mají hodnotu `1`. Zapište v šestnáctkové soustavě hodnotu proměnné `x` pro provedení níže uvedeného kódu v Pascalu, když víme, že se kód bude překládat pro little endian platformu:

```
x := x xor (($ABCDEF0123456789 and (not ($0003300000000000 shr 8))) xor 65536);
```

Otázka č. 9

Předpokládejte, že bychom chtěli pro OS zajišťující oddělení adresových prostorů procesů pomocí stránkování naimplementovat aplikaci sloužící jako debugger. Debugger a laděný program budeme spouštět v separátních procesech. Navrhněte a vysvětlete, jak takový debugger naimplementovat – zvlášť, jak debugger provede umístnění „breakpointu“ do cílového procesu, a v čem bude tento „breakpoint“ spočívat, a co se stane po jeho dosažení. Popište všechny funkce OS, které budete pro implementaci tohoto konceptu potřebovat.

Otázka č. 10

Víme, že jedna z hlavních funkcí vyššího programovacího jazyka je odstínění programátora od specifik architektury konkrétního cílového procesoru. Je tedy někdy potřeba při programování ve vyšším programovacím jazyce (např. v Pascalu) znát tzv. *endianitu* cílového procesoru, kde náš program poběží? Pokud ne, tak vysvětlete proč. Pokud ano, tak na příkladu vysvětlete proč.