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).

Otázka č. 1

Vysvětlete, co to je tzv. *znaménkové rozšíření* (sign extension) a jak se liší od tzv. *bezznaménkového rozšíření* (zero extension).

Společná část pro otázky označené X

Předpokládejte níže popsaný CPU vycházející z architektury procesorů Intel 80386 – je to **32-bitový little-endian** CPU s obecnou registrovou architekturou a s 32-bitovým adresovým prostorem. Procesor má obecné registry EAX, EBX, ECX, EDX, DSI, EDI, EBP, 32-bitový příznakový registr EFLAGS s běžnými příznaky, registr ESP (stack pointer, ukazuje na poslední využitý byte, roste dolů), a registr EIP (instruction pointer). V instrukční sadě jsou **instrukce pro standardní bitové operace** a dále mimo jiné **i následující instrukce** (příznakový registr modifikují pouze aritmetické operace, ale instrukce přenosu dat nikoliv):

- `MOV reg,DWORD PTR imm32/[addr]` (load register)
- `MOVZX reg,WORD PTR imm16/[addr]`
  (load 16-bit value with **z**ero e**x**tension into register)
- `MOV DWORD PTR [addr],reg` (store register)
- `MOV reg0,reg1` (transfer from `reg1` to `reg0`)
- `ADD reg,imm32/[addr]/reg` (add without carry)
- `SUB reg,imm32/[addr]/reg` (subtract without carry)
- `PUSH imm16/imm32/[addr]/reg`, `POP [addr]/reg`
- `JMP addr` (direct jump), `JNE addr` (jump if not equal)
- `JB addr` (jump if below, tj. když `carry = 1`)
- `JAE addr` (jump if above or equal, tj. když `carry = 0`)
- `CMP DWORD PTR [addr],imm32` (32-bit compare)
- `CMP WORD PTR [addr],imm16` (16-bit compare)
- `CALL addr` (direct call), `CALL [addr]` (indirect call)
- `RET` (return from subroutine)

Všechny výše uvedené instrukce i bitové operace se dvěma operandy mají vždy vlevo cílový a vpravo zdrojový operand. Instrukce mohu mít jednu z následujících variant operandů (povolené varianty viz definice konkrétní instrukce):

- 32-bitový immediate `imm32`
- 16-bitový immediate `imm16`
- absolutní adresa `[addr]`, kde `[addr]` může být jedno z:
  - `[imm]` adresa daná konstantou
  - `[reg +/- imm]` adresa daná součtem/rozdílem obsahu registru `reg` a konstanty `imm`
- libovolný registr `reg`

Dále pro uvedený CPU známe následující mapování vybraných instrukcí do jejich strojového kódu:

- `PUSH xxxxxxxxh` (32-bit immediate):
  `68` `xx xx xx xx`
- `PUSH xxxxh` (16-bit immediate):
  `66 68` `xx xx`
- `PUSH [xxxxxxxxh]` (32-value from absolute address):
  `FF 35` `xx xx xx xx`
- `PUSH [xxxxxxxxh]` (16-value from absolute address):
  `66 FF 35` `xx xx xx xx`
- `CALL xxxxxxxxh` (direct **relative** call):
  `E8` `xx xx xx xx`
- `CALL [xxxxxxxxh]` (indirect call):
  `FF 15` `xx xx xx xx`
- `RET` (return): **`C3`**

Otázka č. 2 (X)

Víme, že na logické adrese `$001F8000` je v počítači uložený následující strojový kód **relativního volání** nějakého podprogramu:

```
E8 EB FF FF FF
```

Pokud dále víme, a že adresa cíle skoku je relativní k prvnímu bytu následující instrukce, tak napište a zdůvodněte, jaká bude logická bázová adresa výše volaného podprogramu.

Otázka č. 3 (X)

Předpokládejte, že nějakým běžným překladačem Pascalu, který používá variantu Pascalové volací konvence (argumenty se předávají na volacím zásobníku zprava doleva a **odstraňuje je volaný**), přeložíme níže uvedený program do strojového kódu. Napište v šestnáctkové soustavě hodnoty všech bytů strojového kódu procedury `P2` (víme, že generovaný kód má prázdný prolog a součástí epilogu je pouze instrukce `RET`). Víme, že proměnné `x`, `y`, `ptr` budou uložené na adresách `$0040C000`, `$0040C004`, `$0040C008`, procedury `P1`, `P2` budou začínat na adresách `$001F8A04`, `$001F8B40`.

```
type PProc
     = procedure(a : word; var b : word; c : word);
var x, y : word; ptr : PProc;
procedure P1(a : word; var b : word; c : word);
  begin ... end;
procedure P2; begin
  ptr(x, y, 4111);
end;
begin ptr := @P1; P2; end.
```

Otázka č. 4 (X)

Napište v Pascalu bez použití inline assembleru kód procedury (i s deklarací), která by mohla být běžným překladačem přeložena do níže uvedeného kódu v assembleru 80386 (předpokládejte, že procedura používá běžnou Cčkovou volací konvenci, tj. argumenty se předávají na volacím zásobníku zprava doleva, a **odebírá je volající**):

```
  PUSH  EBP
  MOV   EBP,ESP
  SUB   ESP, 4
  MOV   EAX,DWORD PTR [EBP+0Ch]
  MOV   DWORD PTR [EBP-04h],EAX
label1:
  CMP   WORD PTR [EBP+08h],000Ah
  JNE   label2
  MOVZX EAX,WORD PTR [EBP+08h]
  SHL   EAX,6
  ADD   EAX,DWORD PTR [EBP-04h]
  MOV   DWORD PTR [EBP-04h],EAX
  JMP   label3
label2:
  MOVZX EAX,WORD PTR [EBP+08h]
  ADD   EAX,DWORD PTR [EBP-04h]
  MOV   DWORD PTR [EBP-04h],EAX
label3:
  CMP   DWORD PTR [EBP-04h],00001234h
  JB    label1
  MOV   EAX,DWORD PTR [EBP-04h]
  MOV   DWORD PTR [0040c000h],EAX
  MOV   ESP, EBP
  POP   EBP
  RET
```

Otázka č. 5

Máme-li nějaký moderní CPU s podporou stránkování (a pro jednoduchost s jednoúrovňovou stránkovací tabulkou, jak ji známe z přednášek) a nějaký moderní operační systém. Detailně na nějakém vhodném příkladu vysvětlete, co to je koncept tzv. *demand (lazy) loadingu* spustitelného souboru. K vašemu příkladu nakreslete část obsahu stránkovací tabulky (s označením nastavených flagů u každé význačné stránky) a popište změny stránkovací tabulky při demand loadingu alespoň 3 různých stránek.

Otázka č. 6

Předpokládejte, že přeložíme běžným překladačem Pascalu níže uvedený zdrojový soubor `prg6.pas` do spustitelného souboru `prg6.exe`:

```
begin WriteLn('Hello world!'); end.
```

Detailně vysvětlete, kdo, kdy (v jaké fázi překladu/běhu programu), a jakým způsobem určí adresu procedury `WriteLn`, a tedy kdo takovou finální adresu uloží jako argument instrukce `CALL` ve strojovém kódu v `prg6.exe` vygenerovaném z uvedeného hlavního programu.

Otázka č. 7

Naprogramujte v Pascalu funkci `Conv` s níže uvedeným prototypem, která převede ***číslo* ve floating-point formátu *double*** (typ *double* je 64-bit dle IEEE 754, tj. mantisa je normalizovaná se skrytou 1 a zabírá spodních 52 bitů, pak následuje 11-bitový bias +1023 exponent, a poslední bit je znaménkový) **do floating-point formátu *single*** (typ *single* je 32-bit číslo dle IEEE 754, tj. mantisa je normalizována se skrytou 1 a zabírá spodních 23 bitů, pak následuje 8-bit exponent uložený ve formátu bias +127, poslední bit, tedy MSb, je znaménkový). *Nekonečna* jsou v obou formátech reprezentována jako maximální možná hodnota exponentu s mantisou se všemi nulovými bity, hodnoty *Not a Number* jsou reprezentovány jako maximální možná hodnota exponentu a libovolná nenulová mantisa – oba tyto druhy speciálních hodnot má vaše funkce podporovat. Přebytečné bity mantisy s nejnižší vahou při převodu jednoduše oříznete a zahoďte. Pokud je jako argument funkce předána reprezentace hodnoty s větším exponentem než je nejvyšší možný pro typ single, tak má výsledek funkce reprezentovat hodnotu **nekonečno**, pokud má argument funkce příliš malý exponent, tak má být výsledek **nula** (reprezentovaná jako číslo s nejmenším možným exponentem a denormalizovanou mantisou bez skryté 1 – pro jednoduchost předpokládejte, že jiná čísla než nula v takovém denormalizovaném tvaru uložena nejsou). Celý výpočet/zpracování zapište **jen s využitím celočíselné aritmetiky** Pascalu (bez použití Pascal typů pro reálná čísla), a zvažte možnost použít pro výpočet bitové operace podporované v Pascalu. Poznámka: typ `qword` je unsigned 64-bit integer, `longword` je 32-bitový.

```
function Conv(flt64 : qword) : longword;
```

Otázka č. 8

Detailně vysvětlete, k čemu na I^2C sběrnici slouží tzv. *repeated start*, a zda by bylo možné se bez něj obejít. Jakým způsobem se *repeated start* rozpozná od běžného bitu?

Otázka č. 9 (X)

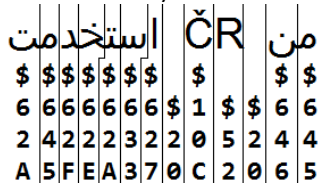
Předpokládejte, že programujeme nějaký jednoduchý operační systém s kooperativním přepínáním vláken pro jednoprocesorovou architekturu x86 ze **společné části X** na předchozí straně (32-bitové CPU s vypnutým stránkováním, tj. logické adresy se přímo rovnají fyzickým adresám), do jehož jádra jsme naimplementovali následující část plánovače s níže uvedenou variantou syscallu `Yield`:

```
const UNUSED = 0; CREATED = 1; READY_TO_RUN = 2;
      RUNNING = 3; TERMINATED = 4;
type PThreadProc = procedure; cdecl;
     TThread = record
       SP : longword; OwnerPID : longword;
       State : longword;
     end;
var currTID, currPID : longword;
    threads : array[0..255] of TThread;
    tempSavedSp : longword; nextTID : longword;

function SchedulerGetNextTID : longword; forward;

procedure Yield; cdecl;
begin
  nextTID := SchedulerGetNextTID;
  asm
    mov [tempSavedSp], esp
  end;
  threads[currTID].SP := tempSavedSp;
  if threads[currTID].State = RUNNING then
    threads[currTID].State := READY_TO_RUN;
  threads[nextTID].State := RUNNING;
  tempSavedSp := threads[nextTID].SP;
  asm
    mov esp, [tempSavedSp]
    mov ebp, esp
  end;
  currTID := nextTID;
end;
```

Dopište do uvedeného jádra OS v Pascalu implementaci funkce `SchedulerGetNextTID` tak, aby implementovala koncept *round robin* plánovače.

Otázka č. 10

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):



Napište v Pascalu proceduru, která správně na libovolném 32-bitovém **little-endian** procesoru uloží od adresy `$057A3000` právě výše uvedený text a to jako **null-terminated řetězec** v kódování **UTF-16 little-endian**. Pozor, předpokládejte, že používáte běžnou variantu jazyka Pascal, kde textové řetězce typu `string` a typ `char` reprezentují znaky v 8-bitovém rozšíření ASCII kódování a ve stejném kódování očekává váš překladač i zdrojový kód vašeho programu. Dále můžete očekávat, že typ `longword` je 32-bitový bezznaménkový a typ `word` je 16-bitový bezznaménkový.