Pytania i zadania przykładowe do ćwiczeń z „Architektury komputerów”
cz. I, listopad 2015

1. W czterech kolejnych bajtach pamięci znajdują się liczby:

adres	*zawartość*
65BH	2AH
65AH	00H
659H	37H
658H	F2H

Przyjmując, że podane bajty stanowią liczbę binarną 32-bitową podać wartość tej liczby w zapisie szesnastkowym przy założeniu, że stosowana jest konwencja *mniejsze wyżej* (ang. big endian).

ODP. F237002AH

2. Przed wykonaniem poniższego fragmentu programu w rejestrze EAX znajdowała się liczba p, w rejestrze EDX – liczba q. Określić zawartość tych rejestrów po wykonaniu poniższych rozkazów.

```
xor   eax, edx
xor   edx, eax
xor   eax, edx
```

Wskazówki: określić zawartość k-tego bitu rejestru EDX, jeśli wiadomo, że przed wykonaniem podanego fragmentu programu na bitach o numerze k znajdowały się wartości a_k (rejestr EAX) i d_k (rejestr EDX). Ponadto operacja XOR:

- jest przemienna $x \oplus y = y \oplus x$
- jest łączna $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- zachodzi związek $x \oplus x = 0$.

1) EAX = p, EDX = q
2) EAX = p xor q, EDX = q
3) EAX = p xor q, EDX = q xor (p xor q)
4) EAX = (p xor q) xor (q xor (p xor q)),
EDX = q xor (p xor q)

EDX = q xor (p xor q) = p xor (q xor q) =
= p xor 0 = p
EAX = (p xor q) xor p = (p xor p) xor q =
= 0 xor q = q

ODP. EAX = q, EDX = p

3. Napisać fragment programu w asemblerze, który wykona działania równoważne działaniu poniższego rozkazu

```
xor   edi, esi
```

W napisanym fragmencie nie można używać rozkazu `xor`.

EDI = p, ESI = q
p xor q = (p and (not q)) or ((not p) and q)

```
mov eax, edi    ; eax = edi = p
mov ebx, esi    ; ebx = esi = q
```

```
not ebx          ; not q
and edi, ebx     ; p and (not q)
not eax          ; not p
and esi, eax     ; (not p) and q
or edi, esi
```

```
//Da sie krócej
// Nie potrzebujemy ebx
mov eax, esi
not eax
and eax,edi
not edi,
and edi,esi
or edi,eax
```

4. Wyjaśnić dlaczego wykonanie poniższego fragmentu programu spowoduje wygenerowanie wyjątku procesora?

```
mov   ax, 0
mov   dx, 1
div   dx
```

ODP. Ponieważ iloraz nie zmieści się w rejestrze AX
Wytłumaczenie:
AX=0000h
DX=0001h
Rozkaz div działa w tym przypadku tak, że będziemy dzielić to, co w DX:AX przez to co w DX i wynik zapiszemy w AX (ewentualną resztę w DX). W tym przypadku w DX:AX mamy 10000h, więc po podzieleniu przez DX (0001h) wynik to 10000h (cokolwiek nie podzielisz przez 1 daje… cokolwiek), co nie mieści się w rejestrze przeznaczonym na wynik, czyli AX.

5. Na czym polega różnica w sposobie wykonania poniższych rozkazów:

```
push  dword PTR esi
push  dword PTR [esi]
```

ODP. (1) wrzuca na wierzchołek stosu zawartość rejestru ESI
(2) wrzuca na wierzchołek stosu wartość wskazywaną przez adres zawarty w ESI

6. Podać zawartości rejestrów EBX i CX po wykonaniu niżej podanego fragmentu programu

```
.data
stale       DW 2,1
napis       DW 10 dup (3),2
tekst       DB 7
            DQ 1
.code
_main:
      MOV    CX, napis -1
      SUB    tekst, CH
      MOV    EDI,1
      MOV    tekst[4*EDI],CH
      MOV  EBX, DWORD PTR tekst+1
```

ODP. EBX = 03000001H, CX = 0300H

Komentarz: warto sobie rozpisać co i jak w pamięci (wzięte z edycji 2013 bo ładnie zrobili):

```
Adres  Wartość
2232E  00h    ; 03h
2232D  00h
2232C  00h
2232B  01h
2232A  07h    ; tekst, 7-3=4
22329  00h
22338  02h
...    ...
22325  00h
22324  03h    ; napis
22323  00h
22322  01h
22321  00h
22320  02h    ; stale
```

```
.data
stale       DW 2,1
napis       DW 10 dup (3),2
tekst       DB 7
            DQ 1
.code
 main:
      MOV   CX, napis -1        ; CX = 0300h
      SUB   tekst, CH           ; tekst = 07h - 03h = 04h
MOV   EDI,1                     ; EDI = 1
      MOV   tekst[4*EDI],CH     ; tekst[4] = 03h
MOV   EBX, DWORD PTR tekst+1    ; EBX = 03000001h
```

7. Poniższy fragment programu może służyć do rezerwacji obszaru pamięci na dane o nieokreślonych wartościach początkowych. Podać równoważną deklarację tego obszaru używając dyrektywy dd.

```
obroty      LABEL       dword
            ORG         $ + 28
```

ODP. obroty dd 7 dup (?)
potwierdzone u AJ /Paulina

ROZWAŻANIA
obroty LABEL dword wpisuje adres nowo zarezerwowanego obszaru czterobajtowego do zmiennej ‘obroty’ dodatkowo zwiększamy licznik lokacji o 28 bajtów (co jest równoznaczne z tym, że pomiędzy ostatnim bajtem z tego czterobajtowego podwojnego slowa a jakimiś kolejnymi bajtami za ORG $ + 28 zostanie 28 bajtów co w praktyce oznacza, że zarezerwowaliśmy obszar 32 bajtów, można to zobaczyć podgladajac listing z asemblacji), dd definiuje 4 bajty, czyli potrzebujemy 8 takich definicji - obroty dd 8 dup (?).

Sprawdzałem pod asemblerem i wg mnie poprawny wynik to: obroty dd 7 dup (?)

W opracowaniach z wyższych lat jest również dd 7 dup(?). I jest tak dlatego, że licznik lokacji $ ma w sobie adres aktualnie tłumaczonego rozkazu albo danej. I faktycznie jak się sprawdzi w visualu to wychodzi dd 7 dup (?)

do czerwonego: //nie mieszajmy w to polityki...
obroty LABEL dword owszem, wpisuje adres tego obszaru do zmiennej ‘obroty’ ale samo w sobie nie rezerwuje podwójnego slowa (czterech bajtów) - potwierdzone, czerwone źle :)

8. Określić zawartości znaczników OF, ZF i CF po wykonaniu podanego niżej fragmentu programu.

```
xor   eax, eax
sub   eax, 0FFFFFFFFH
```

ODP. OF = 0 (liczbę w eax można interpretować też jako liczbę w U2 wtedy eax = -1 i wynik odejmowania 0 -(-1) nie powoduje nadmiaru)
ZF = 0 (wynik różny od zera)
CF = 1 (wystąpiła prośba o pożyczkę)
XOR nie ustawia ZF na 1? Ustawia ale potem sub ją zeruje

9. W wyniku wykonania podanego niżej fragmentu programu w języku C, zmiennej p została przypisana wartość 0x12. Określić czy w komputerze stosowana jest konwencja mniejsze niżej (little endian) czy mniejsze wyżej (big endian).

```
unsigned char p ;
unsigned short int proba = 0x1234 ;
unsigned char * wsk =
    (unsigned char *) &proba ;
p = *wsk ;
```

ODP. Big endian
(w 3 linii do wsk został zapisany adres bajtu o niższym adresie. Jeżeli p = 0x12, to w komputerze stosowana jest konwencja big endian (mniejsze wartości na wyższym adresie)

10. W rejestrze EBX znajduje się liczba całkowita w kodzie U2. Zakładamy, że liczba zawarta jest w przedziale $< -(2^{31} - 1), 2^{31} - 1>$. Napisać fragment, który przekoduje tę liczbę na kod *znak-moduł*.

ODP.
U2->ZM algorytm:

1. sprawdź czy liczba dodatnia
2. jeśli dodatnia, nic nie rób
3. jeśli ujemna, zaneguj wszystkie bity oprócz bitu znaku i dodaj do liczby 1

a zatem:

```
        rol ebx, 1
        mov al,bl
        and eax, 1
        ror ebx,1
        cmp al, 1
        jne dodatnia
        rcl ebx,1
        not ebx
        rcr ebx,1
        inc ebx
dodatnia:
        ; nie interesuje nas co sie tu stanie
```

DZIAŁA.

o wiele prościej można: (tak samo napisałem hah pozdro :D)

```
cmp ebx, 0
jge    koniec
neg ebx; zamienienie ujemnej liczby na dodatnia liczbe o tej samej wartosci
or ebx, 80000000H ; ustawienie najstarszego bitu na 1, mlodsze bity bez zmian->nie powinno
tu sie jeszcze dodać jedynki na koniec zgodnie z alg. zamiany z u2 na zm ?
czy skoro idziemy w odwrotną stronę to nie powinno być -1?
-aa ok dzieki, faktycznie z notem pomylilem
; rozkaz neg neguje wszystkie bity i dodaje jedynkę na końcu
;rozkaz not neguje wszystkie bity
koniec:

--nie jestem pewna, jakieś sugestie? :
xor ebx, 01111111h
bt ebx, 7
jnc dalej
add ebx, 1
w sumie na jakims przykladowej liczbie to zrobilem to działa wiec pewnie tak moze byc, tylko
tak sie zastanawiam, bo te xor'owanie w pierwszej linijce zalatwia pozniejsza negacje, czemu i
tak to dobrze dziala?
;rozkaz neg neguje wszystkie bity i dodaje jedynkę na końcu
;rozkaz not neguje wszystkie bity


        rcl ebx, 1  ; przesuwam bit znaku do flagi CF
        jnc dodatnia ; sprawdzam czy jest 0 czy 1
niedodatnia:
        neg ebx ; neguję pozostałe bity (bit znaku jest w CF)
        rcr ebx, 1 ; przesuwam bit znaku na swoje miejsce
        ;inc ebx bo neg dodaje jedynkę na końcu :D
```

dalej: ...

11. Podać zawartość rejestru DH w postaci liczby dziesiętnej po wykonaniu poniższych rozkazów:

```
mov    dh, 15
xor    dh, 12
```

dh = 00001111 B

dh = 00001111 B XOR 00001100 B =00000011 B

ODP. dh = 3

12. Uzupełnić zdanie: *W wyniku wykonania poniższego rozkazu zawartość rejestru ESI zostanie . .*

. . . . . .

```
lea    esi, [esi + esi*8]
```

ODP. zostanie pomnożona przez 9

13. Na czym polega błąd w poniższym fragmencie programu:

```
sub   esp, 4
mov   [esp], 'A'
```

ODP. nie jest określony rozmiar operandu! nie wiadomo na ilu bajtach zapisać ‘A’ - powinno być np. mov dword PTR [esp], ‘A’

14. Rozkazy

```
push  ebx
push  ecx
```

można zastąpić równoważna sekwencją:

```
sub    esp, 8
mov    [...], ebx
mov    [...], ecx
```

Uzupełnić pola adresowe podanych wyżej rozkazów mov.

ODP. sub esp, 8
mov [esp+4], ebx
mov [esp], ecx

15. Jakie wartości zostaną wpisane do rejestrów EDX i EAX po wykonaniu niżej podanego fragmentu programu?

```
mov   eax, 0FFFFFFFFH
mov   ebx, 0FFFFFFFFH
imul  ebx
```

ODP. (-1) * (-1) = 1
EAX = 1, EDX = 0

tu mam pytanie, te liczby zawsze trzeba traktowac jako liczby w U2? nie mozna tego potraktowac jako liczba w nkb? wiem, ze jesli pomnozymy najwieksza mozliwa liczbe zapisana na 32 bitach i pomnozymy ją razy ta sama liczbe to wyjdzie jakas wielka liczba, ktora nie zmiesci sie w EAX, ale to w koncu tez jakas liczba :D? gdyby np. bylo 1101 razy cos tam , to skad mamy wiedziec czy 1101 to jest 13 w nkb czy -3 w U2 ?
Komputer widzi tylko zera i jedynki. Dla niego nie ma różnicy czy liczba jest w U2 czy w NKB bo to tylko postrzeganie programisty. W tym przykładzie wykonujemy rozkaz imul, który ma z góry powiedziane, że ma traktować te liczby jako zapisane w kodzie U2. Gdybyśmy mieli rozkaz mul, on potraktowałby te liczby jako liczby w kodzie NKB. Tak samo byłoby przy rozkadzie div, ale już przy add i sub nie - dlatego, że dla komputera nie ma znaczenia czy dodajemy lub odejmujemy liczby w kodzie U2 czy NKB - na zerojedynkach wychodzi dokładnie to samo. Tylko dla mnożenia i dzielenia są osobne rozkazy imul i idiv
-aa ok dzieki

16. Na czym polega błąd w poniższym fragmencie programu?

```
v2    dw    ?
- - - - - - - - - - -
      mov   v2, 11111H
```

ODP. Wartość 11111H nie zmieści się na dwóch bajtach. Należy użyć dd zamiast dw.
11111h zajmuje 5*4 bitów, czyli 2 bajty i 4 bity, a define word rezerwuje 2 bajty
Można użyć

```
mov v2, word PTR 11111h ,
```

ale wtedy te nadmiarowe 4 bity bedza juz w kolejnej komorce pamieci.

Proszę, żeby mnie poprawić,gdy moja odpowiedź nie spełniłaby oczekiwań sprawdzającego kolokwium

17. Określić zawartości znaczników OF, ZF i CF po wykonaniu podanego niżej fragmentu programu.

```
mov   ax, 1
add   ax, 0FFFFH
```

ODP. OF = 0, ZF = 1, CF = 1

18. Podać liczbę, która zostanie wyświetlona na ekranie w wyniku wykonania poniższego fragmentu programu. Podprogram wyswietl32 wyświetla na ekranie w postaci dziesiętnej liczbę binarną zawartą w rejestrze EAX.

```
qxy   dw    254, 255, 256
-    -    -    -    -    -    -    -
      mov   eax, dword PTR qxy + 1
      call  wyswietl32
```

ODP. 65280 (FF00H)

komentarz:
254 = 00000000 11111110 = 00FE
255 = 00000000 11111111 = 00FF
256 = 00000001 00000000 = 0100

przykładowe rozmieszczenie w pamięci:
22325 01h
22324 00h
22323 00h
22322 FFh
22321 00h ; qxy + 1
22320 FEh ; qxy

mov eax, dword PTR qxy + 1 ; do eax wpisujemy 4 bajty (bo taki ma rozmiar eax) (nie, ze eax ma taki rozmiar, tylko przesylam dword PTR czyli podwojne slowo czyli 4 bajty, ale tak to cale zadanie chyba dobrze)począwszy od qxy +1 czyli od adresu 22321 (taki sobie wymysliliśmy). little endian więc eax = 0000FF00h (tak dla pokazania dokladniej :P)

19. W rejestrach EDX:EBX:EAX znajduje się 96-bitowy ciąg bitów. Napisać fragment programu, w którym ciąg ten zostanie przesunięty cyklicznie w lewo o 1 pozycję.

EDX	EBX	EAX

Wskazówka: wykorzystać rozkazy przesunięcia w lewo SHL (bity wychodzące z rejestru wpisywane są do CF) i RCL (zawartość CF wpisywana jest na najmłodszy bit rejestru, a bity wychodzące z rejestru wpisywane są do CF). Wykorzystać także rozkaz BT.

ODP. bt edx, 31
rcl eax, 1
rcl ebx, 1
rcl edx, 1
tak nawiasem mówiac, wie ktoś po co kazali tego shl uzywać :P ?

ja mam tak:
shl eax, 1
adc ebx, 0
adc edx, 0
adc eax, 0

chyba też pyknie

mam pytanie, tutaj rozkazem bt kopiujemy najstarszy bit EDX do CF, potem przesuwamy cyklicznie eax w lewo, czyli na najmłodszy bit EAX wchodzi ten z CFa,a reszte przesuwamy w lewo,a potem kolejne dwa przesuniecia w lewo (ebx i edx), nie wyjdzie to na koniec, ze przesunelismy o 3 w lewo ? wiem, ze nie uzywamy juz rozkazu BT przed rcl ebx i rcl edx, no ale i tak przechodzą bity przez CF (bo tak działa rcl). Odpowiedź do pytania: Nie przesuniemy o 3 lewo bo każdy rejestr przesuwamy tylko o jeden bit. Pokażę jak to wygląda jeżeli by to były krótsze rejestry:
(D0 D1 D2 D3) - edx, (B0 B1 B2 B3) - ebx, (A0 A1 A2 A3)- eax ?-CF
bt edx, 31; (Kolejne rozkazy dają takie efekty
D0 D1 D2 D3) - edx, (B0 B1 B2 B3) - ebx, (A0 A1 A2 A3)- eax D0-CF
rcl eax, 1; (D0 D1 D2 D3) - edx, (B0 B1 B2 B3) - ebx, (A1 A2 A3 D0)- eax A0-CF
rcl ebx, 1; (D0 D1 D2 D3) - edx, (B1 B2 B3 A0) - ebx, (A1 A2 A3 D0)- eax B0-CF
rcl edx, 1; (D1 D2 D3 B0) - edx, (B1 B2 B3 A0) - ebx, (A1 A2 A3 D0)- eax D0-CF
-dzieki, po tej wizualizacji od razu skumalem :D

20. Napisać fragment programu w asemblerze, który zamieni młodszą (bity 15 – 0) i starszą (bity 31 – 16) część rejestru EDX.

ODP. rol edx, 16 (sprawdziłem dla ffff1111h i wyszło 11117fff więc chyba jest błąd)
nie ma błędu. prawdopodobnie użyłeś rcl zamiast rol. rol daje dobry wynik. - potwierdzone info

21. W czterech kolejnych bajtach pamięci począwszy od adresu podanego w rejestrze w EBX znajduje się 32-bitowa liczba całkowita bez znaku zakodowana w formacie *mniejsze wyżej* (big endian). Nie używając rozkazu BSWAP załadować tę liczbę do rejestru EAX w formacie *mniejsze niżej* (little endian).

ODP. mov ecx, 4 ;muszą byc cztery obiegi petli bo zapisujemy liczbe 32bitowa (4bajtowa)

ptl:
mov al, byte ptr [edx+ecx-1] ;do najmloszej czesci (8bitowej) eax wpisujemy po jednym bajcie, to jest mega ważne żeby zrozumieć. zaczynamy od końca czyli ecx = 4, wiec do al wpisuejmy dokładnie [edx+3] - ostatni bajt naszego obszaru. w kolejnym obiegu petli zmniejszy sie ecx wiec bedziemy pisać przedostatni bajt itd.
ror eax, 8 ; przewijamy eax zeby pisać na al kolejny bajt

```
        loop ptl
```

jak ktoś chce sobie sprawdzić to tutaj wersja całościowa:

```
.686
.model flat
extern _ExitProcess@4 : PROC

public _main

.data
pamiec dd 12345678h

.code
        _main:
                mov edx, offset pamiec
                mov ebx, pamiec     ;zeby sprawdzic co tam jest ładnie
                mov ecx, 4
                ptl:
                        mov al, byte ptr [edx+ecx-1]
                        ror eax, 8
                        loop ptl

                push 0
                call _ExitProcess@4 ; zakonczenie programu
END
```

/\ działa git, sprawdzone.

// A ty nie masz błedu w tym u góry ? nie przekreca sie eax przypadkiem o raz za duzo :) ?
moje rozwiazanie

```
_littel_to_big PROC
        push ebp
        mov ebp,esp
        push ebx;
        mov eax,0

        mov ebx,[ebp+8]
        mov ecx ,3
        ptl:
                mov dl,bl
                mov al,dl
                rol eax,8
                ror ebx,8

                loop ptl

        mov dl,bl
        mov al,dl


        pop ebx
        pop ebp
        ret
_littel_to_big ENDP
```

//wg mnie ten pierwszy jest dobry, ale brakuje mu jednej linijki:
Od początku:

załóżmy że mamy w pamieci
0x100 4A z tresci zadania to jest big endian, czyli w ludzkim zapisie to liczba 4A 4B 4C 4D
0x101 4B
0x102 4C
0x103 4D

zatem jesli chcemy little endian, to w eax powinno być wpisane 4D 4C 4B 4A (CZYLI OD KOŃCA)
wiec aby uzyskac taki efekt, to musimy dodać do Twojego programu na koncu

Ten ror w kodzie poniżej jest po to,a by cofnac zmiany po nadmiarowym rol.
bo zauwazcie, ze ostatni bajt ktory wchodzi do al juz nie musi byc przesuwany w lewo, a jedak petla go do tego zmusza
//czy skoro numeruje bity w rejestrach od prawej,[31-0], to najmłodsza część czyli 4D nie
//powinna być najbardziej po prawej w rejestrze, na najmniejszych bitach?
Yyy? Ja tu widzę dwa razy ror, i w petli i poza nia

```
.686
.model flat
extern _ExitProcess@4 : PROC

public _main

.data
pamiec dd 12345678h

.code
        _main:
                mov edx, offset pamiec
                mov ebx, pamiec     ;zeby sprawdzic co tam jest ładnie
                mov ecx, 4
                ptl:
                        mov al, byte ptr [edx+ecx-1]
                        ror eax, 8
                        loop ptl
```

ror eax, 8 - z tym rorem o jedno przejscie za duzo, sprawdzone, czyli bez tego rora i jest dobrze

```
                push 0
                call _ExitProcess@4 ; zakonczenie programu
END
```

```
mov al, byte ptr[ebx]
rol eax,8
mov al, byte ptr[ebx+1]
rol eax,8
...itd
```

po co petla, tak mniej pisania

22. Napisać fragment programu w asemblerze, który obliczy liczbę bitów o wartości 1 zawartych w rejestrze EAX. Wynik obliczenia wpisać do rejestru CL.

ODP.

```
mov ebx,32
xor cl,cl
ptl:
        rcl eax,1 ;przesuń bitowo eax o 1 w lewo, jeśli na ostatnim bicie eax jest jedynka to co carry flag wpadnie 1 jeśli 0 to 0
        jnc dalej ;jump if CF = 0 (http://www.keil.com/support/man/docs/is51/is51_jnc.htm)
        inc cl    ;zwiększ jeśli carry flag było 1
        dalej:
dec ebx
jnz ptl
```

Moja werjsa działająca:

```
mov eax,1100h ;nasz dana
mov ecx,32 ;do petli
mov dl,0 ;nasza zapasowa zmienna
ptl:
rcl eax, 1
jnc dalej
inc dl
dalej:
loop ptl
mov cl, dl
```

23. Napisać fragment programu, w którym liczba 32-bitowa bez znaku znajdująca się w rejestrze EAX zostanie pomnożona przez 10 (dziesięć). Wynik mnożenia w postaci liczby 32-bitowej powinien zostać wpisany do rejestru EAX. Zakładamy, że mnożenie nie doprowadzi do powstania nadmiaru. W omawianym fragmencie nie mogą być używane rozkazy MUL lub IMUL. *Wskazówka*: wykorzystać rozkazy przesunięć i zależność $a \cdot 10 = a \cdot 8 + a \cdot 2$.

ODP. lea eax, [eax + 4*eax]
shl eax, 1

wersja druga:

```
mov ebx,eax
shl ebx,1          ;ebx = eax*2^1
shl eax,3          ; ebx = eax*2^3
add eax,ebx        ; eax = 2*eax + 8*eax
```

24. Ile bajtów zarezerwuje asembler na zmienne opisane przez poniższe wiersze?

```
v1      dq      ?, ?
v2      dw      4 dup (?), 20
v3      db      10 dup (?)
```

ODP. v1 - 16 bajtów
v2 - 10 bajtów
v3 - 10 bajtów

25. Na czym polega błąd w poniższym fragmencie programu?

```
const2      db    ?
- - - - - - - - - - -
      mov   const2, 256
```

ODP.
do const2 wejdzie maxymalnie liczba 255, więc wyraźnie mamy tutaj operandy o róznych rozmiarach co nie jest dopuszczalne przy rozkazie mov

26. Wyjaśnić działanie poniższego fragmentu programu

```
start:      mov   ecx, 3
            sub   ax, 10
            loop  start
```

ODP.nieskończona pętla, w nieskończoność od ax będzie odejmowana liczba 10, nie będzie wyjątku procesora bo do ax będa zapisywały się jedynie bity które do niego wejdą, na początku pętli ustawiany jest ecx na 3, loop zmniejsza ecx o 1 i sprawdza czy ecx jest równy zero i jak nie jest to skacze do start, więc nigdy się to nie skończy.
głównie chodzi o to, ze na poczatku zawsze jest ustawiany ecx=3, wiec nawet jak loop zmniejszy ecx po pierwszym obiegu na 2, leci na poczatku petli a tam znow ecx jest ustawiany na 3 tak? gdyby mov ecx,3 było przed etykieta start, to by wykonalo sie trzy razy i koniec tak?
-tak

27. Podać zawartość rejestru EIP po wykonaniu poniższej sekwencji rozkazów

```
      mov   edx, 347
      xchg  [esp], edx
      ret
```

ODP. EIP = 347
// Warto zaznaczyć ze xchg dziala jak swap. Jakby ktoś nie wiedział

```
mov edx, 347          edx =  0x15B
                      ;tutaj esp wskazuje na komorki pamieci, gdzie jest zapisany jakas
                      wartosc po EIP
xchg [esp], edx       ;patrzymy do komorki w pamieci o adresie przechowywanym w esp,
                     bierzemy wartosc ktora sie znajduje w owej komorce i zamieniamy z 347
                      (0x15B ). Wiec [esp] = 0x15B , edx = adres z ESP
ret                   ; on bierze ze stosu adres powrotu i wpisuje do EIP, ale na stosie
                      wpisane zostalo przeciez 0x15b, czyli
      EIP = 0x15b, czyli 347
```

oczywiscie pozniej program nam sie wywali

oczywiscie prosze wprowadzic poprawki w moim komentarzu, jesli cos jest niezgodne z prawdą!!!!!!
[[[[O.O]]]]

28. Dla podanych niżej dwóch rozkazów podać równoważny ciąg rozkazów, w którym nie wystąpi rozkaz loop.

```
            loop  oblicz
   oblicz:  add   dh, 7
```

ODP.

dec ecx
jnz oblicz
oblicz add dh,7

myślę, że może być też takie coś:
dec ecx
add dh,7

czy warunek (ZF=0) będzie spełniony czy nie to i tak przejdzie do następnej komendy bo jak ZF będzie równe 0 to przejdzie do pierwszego rozkazu od razu po oblicz czyli do add dh,7 a jesli nie bedzie to też tam przejdzie.

pewnie bezpieczniej będzie wykorzystać pierwszą wersję - nie znamy reszty programu

29. Na czym polega błąd w podanym niżej zapisie rozkazu

```
mov byte PTR [eax], byte PTR [edx]
```

ODP.
Podczas użycia instrukcji mov, mamy kilka rodzajów operandów: z adresowaniem, rejestr, bądź operand bezpośredni (np. 20h). Nie może wystąpić sytuacja, że każdy z nich jest operandem z adresowaniem (czyli przesłanie typu "pamięć-pamięć"

30. W pewnym programie została zdefiniowana zmienna

```
wskaznik        dd      ?
```

Napisać fragment programu w asemblerze, który wpisze do tej zmiennej adres komórki pamięci, w której znajduje się ta zmienna.

ODP.
mov wskaznik,OFFSET wskaznik

31. Jaka wartość zostanie wprowadzona do rejestru EDX po wykonaniu podanego niżej fragmentu programu

```
linie dd    421, 422, 443,
      dd    442, 444, 427, 432
_ _ _ _ _ _ _ _ _ _
      mov   esi, (OFFSET  linie)+4
      mov   ebx, 4
      mov   edx, [ebx] [esi]
```

ODP. EDX = 443, adresowanie bazowo-indeksowe, suma ebx + esi i to jest adres
objasni ktos moze jak dziala adresowanie bazowo-indeksowe?

32. Napisać fragment programu w asemblerze, który obliczy sumę cyfr dziesiętnych liczby zawartej w rejestrze EAX. Wynik obliczenia wpisać do rejestru CL. Przykład: jeśli w rejestrze EAX znajduje się liczba 1111101 (dziesiętnie 125), to po wykonaniu fragmentu rejestr CL powinien zawierać 00001000.

ODP.

xor cl,cl
mov ebx, 10

```
petla:
    cmp eax, 0
    je koniec
    xor edx, edx

    div ebx
    add cl, dl
    jmp petla

koniec:
    push 0
    call _ExitProcess@4
```

33. Określić postać komunikatu wyświetlanego przez funkcję `MessageBox` po wykonaniu poniższego fragmentu programu.

```
napis db 'informatyka', 0, 4 dup (?)
_    _    _    _    _    _    _    _
            mov   ecx, 12
przepisz:   mov   al, napis[ecx-1]
            mov   napis[ecx+3], al
            loop  przepisz

            push  0
            push  OFFSET napis
            lea   eax, napis[3]
            push  eax
            push  0
            call  _MessageBoxA@16
```

ODP.
przed obiegami pętli:
napis : informatyka,0,?,?,?,?
po 1. obiegu:
informatyka,0,?,?,?,0
po 2. obiegu:
informatyka,0,?,?,a,0
3.
informatyka,0,?,k,a,0
4.
informatyka,0,y,k,a,0
5.
informatyka,t,y,k,a,0
6.
informatyk,a,t,y,k,a,0
7.
informat,m,a,t,y,k,a,0
8.
informat,r,m,a,t,y,k,a,0
9.
informa,o,r,m,a,t,y,k,a,0
10.
inform,f,o,r,m,a,t,y,k,a,0
11.
infor,n,f,o,r,m,a,t,y,k,a,0
12.

info,i,n,f,o,r,m,a,t,y,k,a,0

nagłówek okienka message box będzie:infoinformatyka
tekst w środku będzie : oinformatyka

34. W tablicy znaki znajduje się pewien tekst zakodowany w formacie UTF–8. Tekst zakończony jest bajtem o wartości 0. Napisać fragment programu w asemblerze, który wyznaczy liczbę bajtów, które zajmować będzie ww. tekst po zamianie na 16-bitowy format UTF-16. Obliczoną liczbę bajtów wpisać do rejestru ECX. Przyjąć, że tekst w zawiera znaki zakodowane na jednym, dwóch lub trzech bajtach. Reguły kodowania opisuje poniższa tabela:

Zakresy od ...do..		Kodowanie UTF–8
00H	7FH	0xxxxxxx
80H	7FFH	110xxxxx 10xxxxxx
800H	FFFFH	1110xxxx 10xxxxxx 10xxxxxx

ODP.

```
xor ecx,ecx
mov ebx,OFFSET znaki
ptl:
        mov al,[ebx]
        cmp al,0
        je koniec
        rcl al,1
        jc wielobajtowy
        add ebx,1
        add ecx,2
        jmp ptl
wielobajtowy:
        rcl al,2
        jc trzybajtowy
        add ebx,2
        add ecx,2
        jmp ptl
trzybajtowy:
        add ebx,3
        add ecx,2
        jmp ptl

koniec:
add ecx,2 ;dodatkowe dwa bajty na zero na końcu
```

inny sposób: (może komuś bardziej podejdzie)

```
.data

znaki db 50h,6fh,0c5h,82h,50h,0e2h,91h,0a4h,0
.code

        _main:
        mov ecx, 0
        mov ebx, 0 ;licznik
```

```
ptl:
        mov dl, znaki[ecx]
        cmp dl, 0
        je koniec

        bt edx, 7
        jc kod_wielobajtowy
        add ebx,2
        inc ecx
        jmp ptl

kod_wielobajtowy:
        bt edx, 5
        jc trzy_bajty
        add ebx,2
        add ecx,2
        jmp ptl

trzy_bajty:
        add ebx,2
        add ecx,3
        jmp ptl

koniec:
        add ebx, 2 ;dodatkowe dwa bajty puste

        push 0
        call _ExitProcess@4
```

Wrzucę jeszcze swoją wersję, może komuś podpasuje:
http://pastebin.com/7rEGepWd

35. Podany poniżej podprogram `dodaj` sumuje dwie liczby 32-bitowe umieszczone bezpośrednio za rozkazem `call`, który wywołuje ten podprogram. Obliczona suma pozostawiana jest w rejestrze EAX.

```
dodaj PROC
      mov   esi, [esp]
      mov   eax, [esi]
      add   eax, [esi+4]
      ret
dodaj ENDP
```

Przykładowe wywołanie podprogramu może mieć postać:

```
      call  dodaj
      dd    5
      dd    7
      jmp   ciag_dalszy
```

Wyjaśnić dlaczego wywołanie podanego podprogramu może spowodować bliżej nieokreślone działania procesora, prowadzące do błędu wykonania programu? Następnie, do podanego kodu podprogramu wprowadzić dodatkowe rozkazy, które wyeliminują ww. błędne działania.

ODP. Po wyjściu z podprogramu natrafiamy na ciąg bajtów będący liczbą 5, która jest podwójnym słowem (4-bajty) i w tym miejscu zostanie potraktowana jako rozkaz, co w tym wypadku da nieoczekiwane rezultaty. Przed wyjściem z podprogramu można dać:

add dword PTR [esp],8 - jak patrze to pod debugerem to chyba nie tak działa, bo zawsze dodaje +4 , i esp tylko o 4 sie zwieksza , dodaje 4 bajty niewazne czy dword ptr czy byte ptr czy word ptr dam,
Apropos pomarańczowego komentarza wyżej:
ret zdejmuje z wierzcholka stosu adres nastepnego rozkazu do wykonania. Wedlug mnie zeby zwiekszyc ten adres wystarczy uzyc pop do jakiegos rejestru, zwiekszyc wartosc o 8 i wrzucic na stos zeby ret sobie go ladnie wzial. (zdebugowane, dziala)

mozesz napisac te instrukcje ? bo juz nie mysle..

ret; pierwotnie na wierzchołku stosu był adres pierwszego bajtu 4 bajtowej liczby 5, gdybyśmy do tej wartosci na wierzchołku stosu dodali 4 to otrzymalibyśmy adres pierwszego bajtu liczby 7, potem gdybyśmy dodali do tego jeszcze 4 to adres pierwszego bajtu rozkazu jmp ciag_dalszy :

36. W programach obsługi kalendarza MS Visual Studio dni świąteczne w miesiącu koduje się w postaci jedynek umieszczonych na odpowiednich bitach słowa 32-bitowego. Tablica zawierająca 12 takich elementów pozwala zakodować informacje obejmujące rok.
Napisać podprogram w asemblerze wyświetlający na ekranie daty dni świątecznych w podanym miesiącu. Parametry wywołania podprogramu znajdują się w rejestrach:
CL – numer miesiąca (1 – 12)
EBX – adres tablicy zawierającej zakodowane daty dni świątecznych w poszczególnych miesiącach.

ODP.

```
        movzx ecx, cl
        dec ecx ;z 1-12 robimy 0-11
        mov eax, [ebx+ecx*4] ;eax przechowuje nasz wybrany miesiac
        mov ecx, 32
        petla:
                dec ecx
                jz koniec

                bt eax, ecx
                jnc petla

                call wyswietlECX ;procedure trzeba sobie napisac, zeby wyswietlic wartosc
                                 ;ecx dziesietnie
                jmp petla
        koniec:
                call __getch

                push 0
                call _ExitProcess@4
```

```
;to ja mam nieco dłużej, ale na ‘chlopski rozum’ i po kolei xd :

.686


.model flat
extern _ExitProcess@4 : PROC
extern _MessageBoxA@16 : PROC
extern __write : PROC
public _main

.data                                   ;przykladowe daty, nie chciało mi sie dokładnych :v
kalendarz dd 10001010000000101000101000000010b          ;styczeń
          dd 10000000001100001000000000110000b          ;luty
          dd 10001010000000101000101000000010b          ;marzec
          dd 10000000001100001000000000110000b          ;kwiecień
          dd 10001010000000101000101000000010b          ;maj
          dd 10000000001100001000000000110000b          ;czerwiec
          dd 10001010000000101000101000000010b          ;lipiec
          dd 10000000001100001000000000110000b          ;siepr
          dd 10001010000000101000101000000010b          ;wrze
          dd 10000000001100001000000000110000b          ;paź
          dd 10001010000000101000101000000010b          ;listpd
          dd 10000000001100001000000000110000b          ;gru

daty db 32 dup(?), '0'


.code
_main:
                                    ;dzielnik do bl
   div bl                                   ;dzielę =>  wynik w al, reszta w ah, np:  12/10 = 1 r 2
   mov [edi],al                                     ; do tablicy wpisujemy cyfre dziesiątek (1)
   add [edi],byte PTRmov ebx,OFFSET kalendarz       ; Błąd
   mov edi,OFFSET daty
   xor ecx,ecx                                      ;zeruje ecx
   mov ecx,5                                        ;numer miesiąca
   sub ecx,1                                        ;liczymy od 0, więc styczeń miesiącem 0
   shl ecx,2                                        ;miesiąc jest podwójnym słowem (4
bajty), więc mnożymy x4
   add ebx, ecx                                     ;zwiekszam adres o odpowiednią wartość
   mov ecx,1                    ;w ecx dzień miesiąca - licznik iteracji po poszczególnych bitach
   mov eax,[ebx]                                    ;wrzucam do eax nasze 32 bity danego
miesiąca
program:
   cmp ecx,32                                       ;jeśli sprawdziło 32 bity, to koniec
   je koniec
   clc                                              ;czyszczę CF
   rcl eax,1                                        ;wrzucam skrajny lewy do CF
   jc swieto                                        ;jeśli to 1, to jest święto
   inc ecx                                      ;else ide na nastepny bit
```

```
    jmp program                                 ;powrót do petli


swieto:
    cmp ecx,10                                  ;w ascii każda cyfra jest kodowana osobno
    jb liczba_jednocyfrowa

;else liczba dwucyfrowa

    push eax                                ;zapamietujemy eax -> wrzucamy na stos

    xor eax,eax                                         ;zeruje eax
    mov ax,cx                                           ;wrzucam dzielną do ax
    mov bl,10               30h             ;dodajemu 30h (zamiana cyfry na ascii),  (31h to 1)
    inc edi                                 ;zwiekszam edi (adres w tab docelowej)
    mov [edi],ah                                ;wpisuje cyfre jednosci
    add [edi],byte PTR 30h                      ;zamieniam na ascii
    inc edi
    mov [edi], byte PTR 020h            ;wpisuje spacje
    inc edi
    inc ecx

    pop eax                                     ; przywracamy wartość eax

    jmp program

liczba_jednocyfrowa:                    ;podobnie jak dwucyfrowa, ale prościej, bo bez dzielenia
    mov [edi],ecx
    add [edi],byte PTR 30h
    inc edi
    mov [edi], byte PTR 020h
    inc edi
    inc ecx
    jmp program
koniec:

    push 0
    push OFFSET daty
    push OFFSET daty
    push 0
    call _MessageBoxA@16

push 0
call _ExitProcess@4
END
```

37. W obszarze pamięci, którego adres początkowy zawarty jest w rejestrze ESI znajduje się ciąg cyfr (w kodzie ASCII) zakończony bajtem o wartości 0. Podany ciąg cyfr należy przepisać do obszaru pamięci o adresie podanym w rejestrze EDI, dodatkowo wprowadzając znak kropki przed dwie ostatnie cyfry. Jeśli w obszarze źródłowym znajdują się tylko dwie cyfry, to przed kropką należy wprowadzić dodatkową cyfrę 0 (przykład: 0.53). Jeśli zaś w obszarze źródłowym znajduje się tylko jedna cyfra, to dodatkowe zera trzeba wprowadzić przed i po kropce (przykład: 0.07).

ODP.

```
.data
cyfry db '8','9','2',0
cyfry2 db '1',0
cyfry3 db '1','2',0
;możecie sobie popróbować, czy dla każdego przypadku działa xd

przepis db 8 dup (?),0  ;przykładowo tutaj 8 bajtów, byle >=cyfry

code:
_main:
   mov esi, OFFSET cyfry
   mov ebx,esi                    ; kopia w ebx
   mov edi, OFFSET przepis
   mov ecx,0

;petla, która liczy ile jest cyfr
petla1:
   cmp [esi], byte ptr 0
   je klasyfikuj
   inc ecx
   inc esi
   jmp petla1

;tutaj określamy, co trzeba wpisać
klasyfikuj:
   cmp ecx,1
   je zero_kropka_zero
   cmp ecx,2
   je zero_kropka

;tutaj, jeżeli są więcej niż 2 cyfry, to skaczemy do 'wpisz kropke' przed 2 ostatnimi
petla2:
   cmp ecx,2
   je wpisz_kropke ; ja wpisz_kropke
   inc ebx
   loop petla2

wpisz_kropke:
   mov dx,[ebx]                   ; 2 ostatnie bajty (2 cyfry) zapamietujemy w dx
   mov [ebx],byte ptr 02Eh        ;wpisujemy kropkę pod adres drugiej od konca
```

```
   mov [ebx+1],dx                ;przepisujemy zapamietane 2 cyfry
   jmp koniec

zero_kropka_zero:
   mov dx,[ebx]
   mov [ebx],byte ptr 030h       ;0
   inc ebx
   mov [ebx],byte ptr 02Eh       ;kropka
   inc ebx
   mov [ebx],byte ptr 030h       ;0
   inc ebx
   mov [ebx],dx
   jmp koniec

zero_kropka:
   mov dx,[ebx]
   mov [ebx],byte ptr 030h       ;0
   inc ebx
   mov [ebx],byte ptr 02Eh       ;kropka
   inc ebx
   mov [ebx],dx

koniec:
        push 0
        call _ExitProcess@4
```

38. W rejestrze EBX znajduje skompresowany ciąg 16 cyfr dziesiętnych. Kolejne pary bitów tego rejestru (tj. bity 30-31, 28-29, 26-27, itd.) zawierają kody przyporządkowane cyfrom kodu ASCII wg schematu: 00 → '3', 01 → '4', 10 → '7', 11 → '9'. Napisać fragment programu w asemblerze, który wykona dekompresję zawartości rejestru EBX, przy czym uzyskane cyfry (w kodzie ASCII) należy wpisać do 16-bajtowej tablicy iskry w sekcji danych.

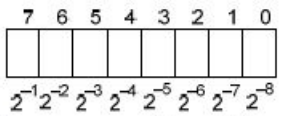
ODP.

```
.data
   iskry db 16 dup (?),0
.code
_main:
   mov edi, offset iskry
   mov ebx,00011001001100100001100100110010b ;przykładowy zakodowany ciąg
   mov ecx,32;będziemy przesuwać o 1 32 razy, bo cały rejestr 32 bitowy, bit po bicie
petla:

   rcl ebx,1                     ;wrzucam pierwszy, skrajny, lewy bit do CF
   jc pierwsza_jedynka           ;skok, jeśli to jedynka
   jmp pierwsze_zero             ;else skok, bo jeśli to nie jest 1, jest to 0

;pierwszy bit może być 1 lub 0
pierwsza_jedynka:
   dec ecx                       ;zmniejszenie ecx (jeden bit przesunęliśmy)
   rcl ebx,1                     ;przesuwamy kolejny bit przez CF
```

```
    jc jeden_jeden              ;jeśli to 1, to mamy przypadek 11
    jmp jeden_zero              ;else, 10

pierwsze_zero:
    dec ecx                     ;zmniejszenie ecx (jeden bit przesunęliśmy)
    rcl ebx,1                   ;przesuwamy kolejny bit przez CF
    jc zero_jeden               ;jeśli to 1, to mamy przypadek 01
    jmp zero_zero               ;else 00

;wszystkie 4 przypadki:
jeden_jeden:
    mov [edi],byte PTR '9'      ;wpisujemy pod adres iskr kod ASCII 9
    inc edi                     ;zwiekszamy adres
    loop petla                  ;zmniejszamy ecx i wracamy do petli
    jmp koniec                  ;jeśli ecx=0 => wykonalismy 32 przesuniecia, wiec nie wróci
jeden_zero:
    mov [edi], byte PTR '7'
    inc edi
    loop petla
    jmp koniec
zero_jeden:
    mov [edi], byte PTR '4'
    inc edi
    loop petla
    jmp koniec
zero_zero:
    mov [edi], byte PTR '3'
    inc edi
    loop petla
    jmp koniec

koniec:
        push 0
        call _ExitProcess@4
END


trochę krótsza wersja:
.686
.model flat
public _main
extern _ExitProcess@4 : proc
extern __write : proc
.data
kody db '3', '4', '7', '9'
iskry db 16 dup(0)
.code

_main :
mov ebx, 0110110001101100B; // 3333 3333 4793 4793
xor edx, edx
```

```
mov esi, 16

petla:
   mov edi, ebx
   and edi, 3H
   mov al, kody[edi]
   mov iskry[esi], al
   dec esi
   shr ebx, 2
   cmp esi, 0
   jne petla


   push 0
   call _ExitProcess@4
END

; tutaj mały “trick” - można było zauważyć, że ten schemat można zgrabnie zapisać za pomocą tablicy:
kod[00] = kod[0] =  ‘3’
kod[01] = kod[1] = ‘4’
kod[10] = kod[2] = ‘7’
kod[11] = kod[3] = ‘9’

co wg mnie upraszczało dalszy algorytm
```

39. W pewnym programie przyjęto reprezentację ułamków właściwych w postaci liczb 8-bitowych binarnych.



Napisać podprogram w asemblerze, który wyświetli na ekranie w postaci dziesiętnej zawartość rejestru AL z dokładnością 3 cyfr po kropce.. Liczba w rejestrze AL zakodowana jest w podanym wyżej formacie.
Przykład: jeśli w rejestrze AL znajduje się liczba binarna 11000000, to na ekranie powinna pojawić liczba 0.750
Konwersję na postać dziesiętną można przeprowadzić w poniższy sposób:
a. Liczbę w rejestrze AL mnożymy przez 10 używając rozkazu mnożenia dwóch liczb 8-bitowych – wynik mnożenia zostaje wpisany do rejestru AX.
b. Liczba wpisana do rejestru AH stanowi wartość kolejnej cyfry liczby dziesiętnej, zaczynając od najstarszej.
c. Powtarzamy opisane operacje wymaganą ilość razy.
d. Zamieniamy uzyskane wartości na kody ASCII i wyświetlamy na ekranie za pomocą funkcji *write*.

```
ODP.
        movzx eax, al
        mov ebx, 0      ;liczba po przecinku do wyswietlenia
        bt eax, 7
        jnc dalejA
        add ebx, 50000000
```

```
dalejA:
        bt eax, 6
        jnc dalejB
        add ebx, 25000000
dalejB:
        bt eax, 5
        jnc dalejC
        add ebx, 12500000
dalejC:
        bt eax, 4
        jnc dalejD
        add ebx, 6250000
dalejD:
        bt eax, 3
        jnc dalejE
        add ebx, 3125000
dalejE:
        bt eax, 2
        jnc dalejF
        add ebx, 1562500
dalejF:
        bt eax, 1
        jnc dalejG
        add ebx, 781250
dalejG:
        bt eax, 0
        jnc dalejH
        add ebx, 390625

dalejH:
        mov ecx, 9      ;wykonuj dzielenie 8-razy, chcemy miec leading (preceding?) zeros z
                        ;przodu w razie w :D dajemy 9 bo na poczatku i tak odejmujemy jeden
                        ;od licznika petli
        mov eax, ebx

petla_dzielenia:
        dec ecx
        jz koniec

        mov edx, 0
        mov ebx, 10
        div ebx

        dec esp
        add dl, '0'
        mov [esp], dl
        jmp petla_dzielenia

koniec:

        dec esp         ;jeszcze "0."
```

```
        mov [esp], byte ptr '.'
        dec esp
        mov [esp], byte ptr '0'
        mov esi, esp

        push 5          ;5 znakow do wypisania "0.xxx"
        push esi        ;adres do liczby
        push 1

        call __write
        add esp, 12  ;zdejmujemy argumenty dla __write ze stosu

        add esp, 10  ;zdejmujemy liczbe po przecinku ze stosu
        push 0
        call _ExitProcess@4
```

trochę krótsza wersja:

```
.686
.model flat
public _main
extern _ExitProcess@4 : proc
extern __write : proc
.data
wynik db '0', '.', 3 dup(0)
.code

_main :

mov esi, 2
mov al, 11100000B
mov bl, 10

petla:
   mul bl
   add ah, 30H
   mov wynik[esi], ah
   mov ah, 0
   inc esi
   cmp esi, 5
   je koniec
   jmp petla

koniec :
   push 5
   push offset wynik
   push 1
   call __write

   push 0
   call _ExitProcess@4
```

END

; to jest dokładnie odwzorowany algorytm z treści zadania, chyba nie wymaga komentarza
+30H to konwersja na kod ASCII

40. Napisać podprogram w asemblerze, który wyświetli na ekranie zawartość rejestru AL w postaci liczby dziesiętnej, wykorzystując niżej opisany algorytm. Zakładamy, że w rejestrze AL znajduje się 8-bitowa liczba binarna bez znaku. Konwersja (używana zazwyczaj do kodu BCD) przebiega następująco:

a. Wyzerować pozostałe bity rejestry EAX (z wyjątkiem AL).
b. Zbadać czy liczba umieszczona na bitach 3 – 0 rejestru AH jest większa od 4, jeśli tak, to do rejestru EAX dodać liczbę 300H.
c. Zbadać czy liczba umieszczona na bitach 7 – 4 rejestru AH jest większa od 4, jeśli tak, to do rejestru EAX dodać liczbę 3000H.
d. Przesunąć rejestr EAX o 1 pozycję w lewo.
e. Powtórzyć 8 razy czynności opisane w pkt. b., c., d.
f. Po przeprowadzeniu ww. operacji w rejestrze EAX znajdować się będą wartości pozycji: setek na bitach 19 – 16, dziesiątek na bitach 15 – 12, jedności na bitach 11 – 8. Wartości te zamienić na kod ASCII i wyświetlić na ekranie za pomocą funkcji *write*.

ODP. Poniżej kod kompletnego programu:

```
.686
.model flat
extern _ExitProcess@4 :PROC
extern __write :PROC
public _main

.data
tekst    db    3 dup (?)

.code

wyswietl_AL    PROC

   mov          edx, 0
   and          eax, 000000FFH    ; zerowanie pozostalej czesci EAX oprocz AL

   mov          ecx, 8
ptl:
   mov          dl, ah                  ; przeniesienie do dl, aby nie naruszyc EAX
   and          dl, 0FH                 ; zerowanie starszej polowy
   cmp          dl, 4
   jna                  niewieksze30    ; skok, jesli liczba na bitach 3-0 w EAX niewieksza
od 4
```

```
   add         eax, 300H
niewieksze30:
   mov         dl, ah
   shr         dl, 4                ; przesuniecie bitow 7-4 na pozycje 3-0
   and         dl, 0FH              ; zerowanie starszej polowy
   cmp         dl, 4
   jna         niewieksze74    ; skok, jesli liczba na bitach 7-4 w EAX niewieksza od 4
   add         eax, 3000H
niewieksze74:
   shl         eax, 1
   loop    ptl

   mov         edx, eax
   shr         edx, 8                     ; przesuniecie znaczacych bitow na poczatek
rejestru
   ror         edx, 8                           ; cyfra setek
   add         dl, 30H
   mov         [tekst], byte PTR dl
   and         dl, 00H
   rol         edx, 4                           ; cyfra dziesiatek
   add         dl, 30H
   mov         [tekst+1], byte PTR dl
   and         dl, 00H
   rol         edx, 4                           ; cyfra jednosci
   add         dl, 30H
   mov         [tekst+2], byte PTR dl

   push    3
   push    OFFSET tekst
   push    1
   call    __write
   add         esp, 12
   ret

wyswietl_AL    ENDP


_main:

   mov         al, 249              ; liczba testowa

   call    wyswietl_AL

   push    0     ; kod powrotu programu
   call    _ExitProcess@4
```

END

41. Dwie liczby całkowite dziesiętne bez znaku, zakodowane w postaci ciągu cyfr w kodzie ASCII, zostały umieszczone w pamięci głównej (operacyjnej). Każdy ciąg cyfr zakończony jest bajtem o wartości 0, a położenie obu ciągów w pamięci określone jest przez zawartości rejestrów ESI i EDI. Napisać fragment programu w asemblerze, który porówna obie liczby i ustawi znaczniki ZF i CF w niżej podany sposób.

{ESI} > {EDI}	⇒	CF = 0 i ZF = 0
{ESI} = {EDI}	⇒	CF = 0 i ZF = 1
{ESI} < {EDI}	⇒	CF = 1 i ZF = 0

Uwagi:

a. Zapisy {ESI} i {EDI} oznaczają, odpowiednio, wartości liczb wskazywanych przez rejestry ESI i EDI.
b. Liczby mogą mieć niejednakową liczbę cyfr.
c. Operację porównania przeprowadzić bez konwersji obu liczb na postać binarną.

ODP.

```
.data
liczbaa db '1','2','3','4',0
liczbab db '1','2','3','4',0

.code

_main:

mov esi, offset liczbaa
mov edi, offset liczbab

mov ecx,0               ;licznik dla pierwszej liczby
mov ebx,0               ;licznik dla drugiej liczby
; wtrącę swoje 3 grosze :D
; ilość cyfr w danej danej (:D) można otrzymać czymś takim: rozmiar_liczbaa = $ - liczbaa
; zgrabniej to wygląda i nie trzeba się bawić w pętle

;liczymy ilosc cyfr pierwszej liczby (jak będzie więcej cyfr, to logiczne, że będzie wieksza)
petla1:
   cmp [esi+ecx],byte ptr 0
   je petla2                    ;jesli trafilismy na bajt zerowy, to przestajemy liczyc
   inc ecx
   jmp petla1
;liczymi ilosc cyfr drugiej liczby
petla2:
   cmp [edi+ebx],byte ptr 0
   je sprawdz_ilosc_cyfr        ;jesli trafilismy na bajt zerowy, to przestajemy liczyc
   inc ebx
   jmp petla2
```

```
sprawdz_ilosc_cyfr:
   cmp ecx,ebx                ;porownujemy ilosc cyfr naszych liczb
   ja pierwsza_wieksza        ;jak pierwsza ma wiecej, to jest wieksza
   je tyle_samo_cyfr          ;jak tyle samo cyfr, to musimy porownywac bajt po bajcie
                              ;else, pierwsza jest mniejsza
druga_wieksza:
   stc                        ;ustawiamy CF na 1
   mov eax,2
   sub eax,1                  ;wykonujemy odejmowanie 2-1 =1; nie zero, więc ZF będzie 0
   jmp koniec

pierwsza_wieksza:
   clc                        ;ustawiamy CF na 0
   mov eax,2
   sub eax,1                  ;2-1 =1 =/=0, więc ZF==0
   jmp koniec
tyle_samo_cyfr:               ;porównujemy w petli kolejne cyfry
   mov eax, [esi]
   mov ecx, [edi]
   cmp eax,byte ptr 0         ;jesli dojdziemy do bajtu zerowego, to liczby są równe
   je rowne
   cmp eax,ecx
   ja pierwsza_wieksza
   jb druga_wieksza
   inc esi                    ;zwiekszamy adresy
   inc edi
   je tyle_samo_cyfr          ;wracamy, aby porownac kolejną cyfrę
rowne:
   clc                        ;CF=0
   mov eax,1
   sub eax,1                  ;1-1=0, więc ustawiamy ZF na 1
koniec:
        push 0
        call _ExitProcess@4
```

trochę krótsza wersja:

```
.686
.model flat
public _main
extern _ExitProcess@4 : proc

.data
liczba1 db '1115'
liczba2 db '1111'
.code
_main :

mov esi, OFFSET liczba1
```

```
mov edi, OFFSET liczba2

mov ecx, 4
xor eax, eax
xor ebx, ebx

compare :
mov al, [esi]
mov bl, [edi]
cmp al, bl
jne koniec
inc esi
inc edi

loop compare

koniec :
push 0
call _ExitProcess@4
END
```

; krótkie objaśnienie jak to działa, nie wiem, czy pamiętacie UC i porównywanie liczb binarnych, ale tutaj podobnie - lecimy od lewej do pierwszej różnicy. Nie ma znaczenia, że tutaj są to kody ASCII: '9' - '0' jest nadal równe 9 itd. znaczników też nie trzeba ręcznie ustawiać, bo jne ich nie zeruje - cmp sam je nam ustawi tak jak jest w opisie do zadania :)
Ale to działa tylko jeśli są takiej samej długości, tak?
To działa dla wszystkich przypadków : http://pastebin.com/kFqJ8ixV

zad. 42

```
;rozwiązanie raczej przydługawe
.686
.model flat

extern _ExitProcess@4 : PROC
extern __write : PROC
public _main

.data
tekst    db        3 dup (?)

.code
tekst1 db 'a', 'b', 0
tekst2 db 'c', 'b', 'd', 0
_main:
   mov esi, offset tekst1
   mov edi, offset tekst2
   mov ecx, 0;licznik
   mov eax, 0;"flaga", bedzie zwiekszala sie o 1, gdy ktorys wyraz sie skonczy, jesli bedzie
rowna 2, znaczy, ze dwa wyrazy rowno sie skonczyly

   ptl:
```

```
        mov eax, 0; zerowanie "flagi"
        mov dl, [esi + ecx];bajt z tekst1 w dl

        cmp dl, 0;wyraz z tekst1 sie skonczyl
        jne dalej
        inc eax; zwiekszamy flage o 1

    dalej:
        ;sprawdzamy czy drugi jest rowny 0
        cmp [edi + ecx],byte ptr 0;wyraz z tekst2 sie skonczyl
        jne dalej2
        inc eax; zwiekszanie "flagi"

    dalej2:
        cmp eax, 2;czy dwa wyrazy skonczyly sie rownoczesnie
        je rowne
        cmp eax, 1;jesli jeden sie skonczyl, sprawdzamy ktory, ten bedzie wyzej w slowniku
        jne nie_koniec;zaden sie nie skonczyl, idziemy dalej
        cmp dl, 0;wyraz z tekst1 sie skonczyl
        je pierwszy_wyzej
        ;sprawdzamy czy drugi jest rowny 0
        cmp [edi + ecx],byte ptr 0;wyraz z tekst2 sie skonczyl, wiec jest wyzej w slowniku
        je drugi_wyzej

    nie_koniec:
        cmp dl, [edi + ecx]
        jb pierwszy_wyzej
        cmp dl, [edi + ecx]
        ja drugi_wyzej
        inc ecx

    jmp ptl;nierozstrzygniecie, kolejny obieg

    rowne:
        xor eax, eax;ustawia zf na 1
        clc; cf = 0
        jmp koniec

    pierwszy_wyzej:
        add eax, 1;zf = 0
        clc; cf = 0
        jmp koniec

    drugi_wyzej:
        xor eax, eax;ustawia zf na 1
        stc; cf = 1
        koniec:

        push 0 ; kod powrotu programu
        call _ExitProcess@4
END
```

zad. 44

Czy ktoś ma i poratuje?
nie jestem pewna, ale dla mniejszych działa

```
mov eax, 10
mov ebx, 2
mov ecx, 32
mov edi, 0
mov edx, 0
ptl:
shl eax, 1
rcl edx, 1
cmp edx, ebx
jb mniejszy
;gdy wiekszy, rowny
shl edi, 1
add edi, 1
sub edx, ebx
jmp dalej
mniejszy:
shl edi, 1
dalej:
loop ptl
mov eax, edi;i tu powinno być do samego ax, ale chyba tak tez moze byc
```