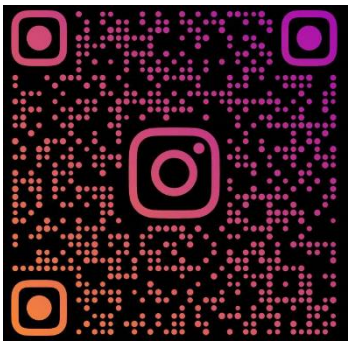
Poradnik do 2 zadania z archi

Autor: puckmoment na discordzie jak coś

Treść zadania:

Napisać program typu .COM, który wczytywać będzie dwie liczby całkowite z przedziału [-32768..32767] i dodawać je do siebie, a otrzymany wynik wyświetlać na ekranie.

Ważne informacje do zadania



To co tutaj opisze można sobie dokładniej odsłuchać w nagraniu wykładu nr 4. Stokfisz dosyć dobrze to tłumaczy jak to działa.

Jakby ktoś w jakimś momencie pisania kodu stracił motywacje niech zeskanuje sobie ten kod QR:

Reprezentacja liczb kodem U2

W zadaniu pierwszym wszystkie dokonywane operacje były na liczbach dodatnich dlatego można było wszystkie liczby zapisać w systemie binarnym, więc jeśli przekazywaliśmy wartość FFFF w systemie szesnastkowym do rejesetru, np. AX komputer zapisywał to jako 16 jedynek co dawało:

$$1111\ 1111\ 1111\ 1111_{(2)} = 65\ 535$$

Można więc wywnioskować, że zapis binarny w 16-bitowym rejestrze pozwala nam na ustawienie w nim wartości od 0 do 65 535. Jak można więc zauważyć zwykły zapis binarny nie pozwala nam na zapisanie liczby ujemnej i właśnie po to jest nam potrzebny kod U2 czyli kod uzupełnień do 2. Liczba taka jak wyżej w systemie U2 wynosi:

$$1111\ 1111\ 1111\ 1111_{(U2)} = -1$$

Bierze się to stąd, że jak w systemie binarnym liczby oblicza się tak:

$$\mathbf{1}111\ 1111\ 1111\ 11\mathbf{1}1_{(2)}$$

$$1*2^{15} + 1*2^{14} + \ldots + 1*2^1 + 1*2^0 = 65\ 535$$

Za to zapisanie tego samego w kodzie U2 wygląda następująco:

$$\mathbf{1}111\ 1111\ 1111\ 11\mathbf{1}1_{(U2)}$$

$$\mathbf{-1*2^{15}} + 1*2^{14} + \ldots + 1*2^1 + 1*2^0 = -1$$

Więc wniosek jest giga prosty, po prostu pierwszy bit od lewej reprezentuje wartość jaką trzeba odjąć od reszty co pozwoli nam zapisać wartości zarówno dodatnie jak i ujemne.

Dwie skrajne wartości można zapisać w taki o to sposób:

$$0111\ 1111\ 1111\ 1111_{(U2)} = 32\ 767$$

$$1000\ 0000\ 0000\ 0000_{(U2)} = -32\ 768$$

Niestety komputer nie zapamięta jakiego typu zapis to jest, więc musi o tym pamiętać programista. Na szczęście stosowanie tego nie jest tak przejebane jak mogłoby się wydawać.

Zakresy oraz jego przekroczenie

W liczbach zapisanych kodem naturalnym może dojść do takiej sytuacji, że ,np. dodając 2 liczby dodatnie wyjdziemy poza zakres który jest możliwy do zapisania w 16-bitowym rejestrze:

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111 \\ +\ \underline{0000\ 0000\ 0000\ 0001} \\ 1\ 0000\ 0000\ 0000\ 0000 \end{array}$$

Pamiętając o, że rejestry są 16-bitowe można łatwo się domyślić, że wynik tego zapisu w rejestrze jest błędny i wartość w rejestrze wyjdzie zero, a bit na czerwono będzie pominięty. Nie jest to jednak do końca prawda bo informacja o tym, że suma tych wartości wyszła poza zakres będzie zapisana w fladze **CF** (*carry flag*) więc można uznać, że to tam trafi ta jedynka.

CF = 1 – jeśli wyszło poza zakres

CF = 0 – jeśli nie wyszło poza zakreś

Jest to flaga do której można się odwołać, np. instrukcją JC, która wykonuje przeskok w zależności od wartości tej flagi.

Flaga CF jest flagą, która działa tylko dla **kodu naturalnego**, więc jeśli korzystamy z kodu U2 będziemy korzystać z flagi **OF** (overflow flag). Działa ona analogicznie do flagi CF.

OF = 1 – jeśli wyszło poza zakres

OF = 0 – jeśli nie wyszło poza zakres

Co trzeba zrobić w zadaniu:

1. Przyjąć od użytkownika 2 liczby z zakresu od –32768 do 32767 i sobie je zapisać do zmiennych. Liczby te mają być przekazane i zapisane jako tablica znaków ASCII.
2. Przekonwertować te 2 liczby do 16 bitowej postaci obliczeniowej w kodzie U2.
3. Przekonwertować te 2 liczby do postaci 32-bitowej i je dodać poprzez wykorzystanie 32-bitowych rejestrów (np. EAX, EBX itd.).
4. Po obliczeniu wyniku należy przekonwertować go do postaci ciągu znaków ASCII i wyświetlić go na ekranie. Do wyświetlenia wyniku można użyć funkcji: 02h (wysłanie znaku na konsolę) lub 09h (wysłanie na konsolę łańcucha znaków).

Mały disclaimer: ten tutorial powstaje wraz z pisaniem kodu. W momencie pisania tej notki nie napisałem nawet linijki kodu i nie wiem jak zrobić większość rzeczy, którą są potrzebne do skończenia zadania. Wszystkiego będę się uczył na bieżąco wraz z tworzeniem kodu dlatego może być tak, że kończąc program i opisując 4-ty punkt wpadnę na to jak lepiej napisać coś co odnosi się do punktu pierwszego. Program na pewno nie zostanie napisany na one-shot'a więc, żeby nikt się nie zdziwił w razie czego. Edit 1: Postał prawie na one-shota więc chuj w tą notke

Punkt 0. Podstawa programu

Po pierwszym zadaniu każdy już powinien wiedzieć jak powinien wyglądać mniej więcej program, ale jako, że będziemy korzystać w rejestrów 32-bitowych trzeba dodać kilka modyfikacji. Podstawa programu powinna wyglądać mniej więcej tak:

```
                .386p
                .MODEL  TINY

; SEGMENT KODU
Kod             SEGMENT USE16

; przesuniecie o 256 bajtow od poczatku segmentu (256 to 100h)
                ORG         100h
;zaloz, ze w cs, ds i ss znajduje sie segment Kod
                ASSUME      CS:Kod, DS:Kod, SS:Kod

Start:
                ; Jakieś instrukcje

koniec:
                mov     ax, 4C00h ; zakonczenie programu
                int     21h

; TUTAJ JEST MIEJSCE NA ZMIENNE

Kod             ENDS

                END     Start
```

.386p i USE16 jest po to bo będziemy używać 32-bitowych rejestrów. Co to dokładnie robi nie ma większego znaczenia raczej.

Punkt 1. Wczytywanie zmiennych

Wyświetlanie komunikatów

Skoro chcemy, żeby użytkownik przekazał nam 2 liczby z jakiegoś przedziału to warto w programie go poinformować co chcemy od niego otrzymać. Wyświetlanie komunikatu najlepiej zacząć od ustalenia treści komunikatu (niesamowite). Treść komunikatu jaki będziemy chcieli wyświetlić na ekranie należy zapisać w zmiennej w postaci tablicy charów. Przykładowe komunikaty powinny w kodzie wyglądać tak:

```
msg     DB "To jest jakas wiadomosc: ", "$"
msg2    DB "To jest jakas druga wiadomosc: $"
```

Znak dolara ($) w assemblerze oznacza koniec stringa, zarówno pierwszy komunikat z przecinkiem jak i ten drugi będą w zapisane w segmencie danych tak samo i będą one tak samo odczytywane (różnica jest tylko wizualna). Oczywiście te wiadomości nie będą w pamięci jako litery tylko jako ich binarna reprezentacja w ASCII, a w samym debuggerze będzie można je odnaleźć jako liczba szesnastkowa. Jak ktoś chce bardzo zmniejszyć czytelność kodu to ten napis może zapisać jako tablica liczb w postaci szesnastkowej.

```
msg     DB      "To jest jakas wiadomosc: ", "$"
msg2    DB      54h, 6Fh, 20h, 6ah, 65h, 73h, 74h
        DB      20h, 6ah, 61h, 6bh, 61h, 73h, 20h
        DB      77h, 69h, 61h, 64h, 6fh, 6dh, 6fh
        DB      73h, 63h, 3ah, 20h, 24h
```

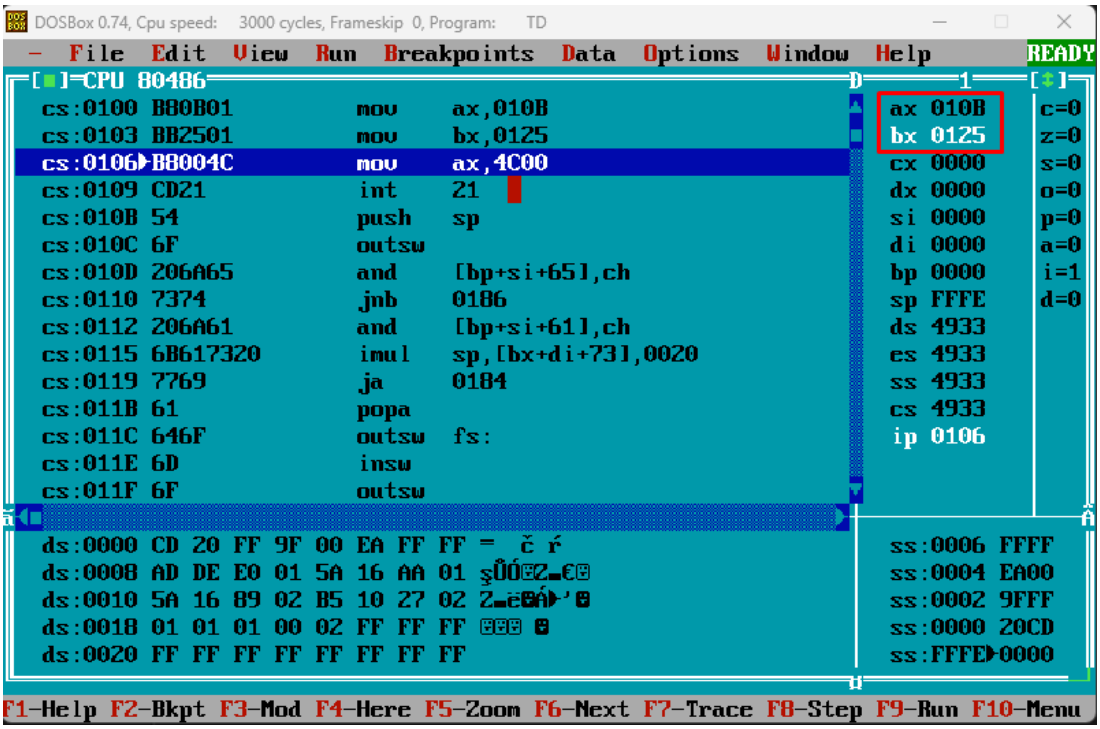
Te 2 zmienne przechowują w sobie dokładnie to samo. Jeśli masz wątpliwości czy twoja wiadomość znajduje się w segmencie danych wystarczy znaleźć ją w odpowiednim miejscu w pamięci, przeliczyć jej wartości z systemu szesnastkowego na dziesiętny, a później po kolei odczytasz czy pokrywają się one z odpowiednimi wartościami z tablicy ASCII. Teraz najpewniej zapytasz: „Ale Marcin, czy da się nauczyć jakiegoś szybkiego sposobu, żeby przeliczyć wartości w systemie szesnastkowy na ASCII?” A skąd ja mam to kurwa wiedzieć? Jestem Dawid i pokaże ci prosty sposób jak ogarnąć co się odpierdala w twoim kodzie.

Po pierwsze dla ułatwienia sobie zadania możesz napisać na początku programu 2 następujące instrukcje:

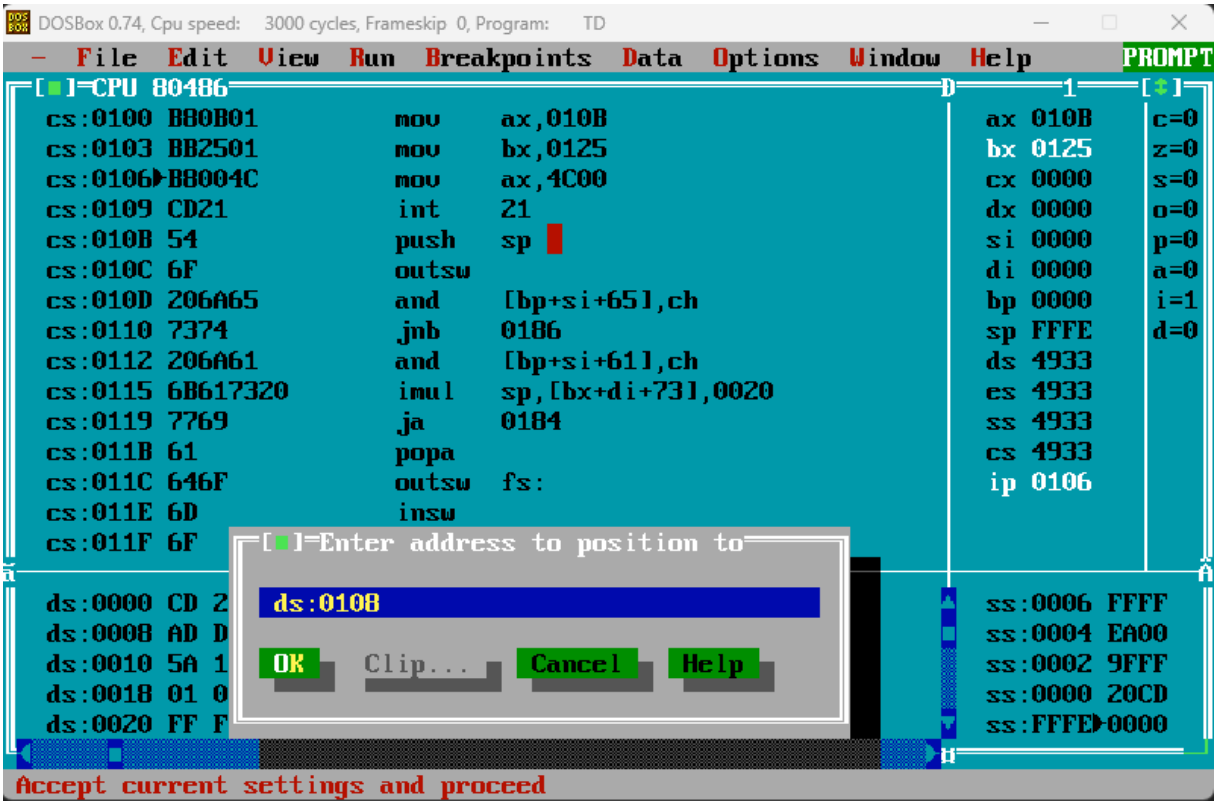
```
Start:
                mov ax, OFFSET msg
                mov bx, OFFSET msg2
```

te 2 linijki kodu spowodują, że w rejestrach AX oraz BX będą adresy początkowego elementu ze zmiennych msg i msg2. Teraz tylko te informacje trzeba wykorzystać:

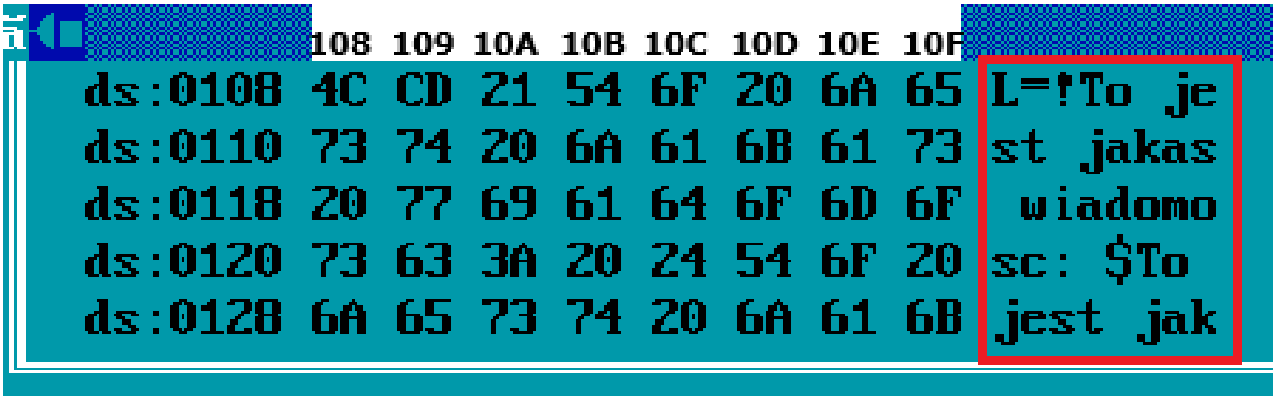
Po skompilowaniu kodu i odpaleniu go w potężnym turbodebuggerze, należy kliknąć tyle razy przycisk F8, żeby wykonały się te nasze 2 wymienione wyżej instrukcje. W rejestrach AX i BX powinniśmy otrzymać coś takiego:



Są to liczby zapisane w formacie szesnastkowym, ale nie ma to większego znaczenia. Zmienne te są zlokalizowane tak trochę po adresie 0100, ze względu na instrukcję ORG 100h, która nam robiła miejsce na PSP. Z wiedzą, że nasze zmienne znajdują się trochę za 256 adresem (100 w szesnastkowym), możemy łatwo je znaleźć nawet nie znając ich dokładnego adresu. Ale jak już go mamy to aż grzech nie skorzystać. Aby przejść szybko do konkretnego adresu wystarczy kliknąć w okienko na dole i użyć skrótu klawiszowego CTRL+G, co wyświetli nam okienko gdzie będzie trzeba wpisać otrzymany adres:



Adres pierwszego elementu to 010B i znajduje się w segmencie danych (**D**ata **S**egment), dlatego wpisujemy adres **ds:0108**. Czemu 0108 a nie 010B? To dlatego, że w każdej linijce mieści się po 8 zmiennych, dlatego jak da się adres który nie jest wielokrotnością 8 to może się zdarzyć, że nie przejdzie do tego adresu, dlatego jak szukacie adresu przy pomocy CTRL+G to warto żeby na końcu było 0 lub 8.



Jako, że przeszliśmy do adresu 108 to trzeba obliczyć gdzie konkretnie znajduje się adres 10B. Nie jest to wybitnie trudne zadanie ale jak ktoś nie radzi sobie zbyt dobrze z liczeniem bądź nie wie jak działa system szesnastkowy to po prawej stronie ma wszystkie wartości z pamięci przekonwertowane na ASCII. Na górze dodatkowo ponumerowałem adresy więc jak ktoś ma wątpliwości może sobie to sprawdzić. Dodatkowo na screenie widać, że tablica przekazana jako liczby w postaci 16 też jest wyświetlana po prawej stronie. Jak nikt nie ma wątpliwości, że zmienne zostały wczytane to teraz najwyższy czas je wyświetlić. Można zrobić to na 2 sposoby:

```
WyswietlMsg:
                mov     dx, offset msg
                mov     ah, 09h
                int     21h
WyswietlMsg2:
                lea     dx, msg2
                mov     ax, 0900h
                int     21h
```

mov dx, offset zmienna
lea dx, zmienna

Te 2 linijki robią dokładnie to samo, wrzucają do rejestru dx adres pierwszego elementu zmiennej. Z tego co wyczytałem to mov jest szybsze i takie rozwiązanie polecają.

mov ah, 09h
int 21h

To przerwanie wyświetla w terminalu po kolei następne znaki aż do momentu znalezienia znaku „$”.

Wynik wykonania tego programu to coś takiego:

```
C:\ZAD2>arch2-3c.com
Podaj pierwsza wartosc: Podaj druga wartosc:
```

Jest spoko ale czegoś tutaj brakuje, np. znaku nowej linii. Aby go zrobić trzeba dodać takie coś:

```
msg     DB      "Podaj pierwsza wartosc:", 10, 13, "$"
msg2    DB      "Podaj druga wartosc:", 10, 13, "$"
```

Wynik to takie coś:

```
C:\ZAD2>arch2-3c.com
Podaj pierwsza wartosc:
Podaj druga wartosc:
```

Przyjmowanie wartości:

Tutaj zaczyna się kontrowersyjny etap bo nie bardzo wiem w jakiej postaci mam przyjmować te liczby, więc jest opcja że to się zmieni.

Trzeba sobie przyjąć 2 zmienne, które przyjmą nasze liczby w postaci tablicy charów.

```
numA1    DB      7, ?, 7 DUP(?)
numA2    DB      7, ?, 7 DUP(?)
```

Pierwsza liczba (7) to ilość znaków ile maksymalnie będzie mogła przyjąć. Druga liczba (?) to ilość znaków ile przyjęło. Trzecia liczba (7) to miejsce na dane. Przyjmiemy z klawiatury maksymalnie 7 znaków bo jeden znak to minus a reszta to 5 liczb i enter. Mając tak przygotowane zmienne można zrobić coś takiego:

```
                mov     dx, offset liczbaA1
                mov     ah, 0Ah
                int     21h
```

Mając tak zapisany numer można już na tym jako tako pracować i zmieniać sobie to na ten kod U2, ale ja proponuje sobie to wyczyścić. Ja to wyczyściłem takim fragmentem kodu:

```
                xor si,si
CleanNumber:
                mov ah, numA[si + 2]
                ;sprawdzanie czy znak jest enterem (nutka w ASCII)
                cmp ah, 0dh
                je  ContinueProg
                mov cleanA[si], ah
                inc si
                jmp CleanNumber
ContinueProg:
                mov cleanA[si], "$"
```

si – rejestr od indeksów (xor si,si resetuje ten rejestr)

numA – to liczba z niepotrzebnymi rzeczami (sprawdzamy od si + 2 bo na 2 pierwszych miejscach są dane opisane wyżej w poradniku)

cleanA – to zmienna przechowująca sam numer (na końcu dodajemy $ bo to znak końca stringa)

Punkt 2. Zmiana z ASCII do U2

Żeby zamienić ciąg znaków na kod zapisany w U2 na początek należy liczbę po prostu zapisać jako postać binarną nie patrząc na to czy liczba jest na minusie czy na plusie. Generalnie wymyśliłem taki kod, który jest giga syfiasty ale działa.

```
                xor     si, si
                xor     dx,dx
                mov     dl, numASCII[1]
                mov     digits, dx
                cmp     cleanASCII[si], "-"
                jne     ConvertLoop
                inc     si
                mov     isNegative, 1
ConvertLoop:
                mov     ax, 1
                mov     cx, digits
                inc     si
                sub     cx, si
PowerLoop:
                cmp     cx, 0
                je      AddLoop
                mov     dl, 10
                mul     dx
                mov     dl, 10
                loop    PowerLoop
AddLoop:
                mov     bl, cleanASCII[si - 1]
                cmp     bl, "$"
                je      ContinueProg
;odejmujemy 30h bo w kodzie ASCII liczby są od 30h do 39h
                sub     bl, 30h
                mov     bh, 0
                mul     bx
                add     num1, ax
                cmp     si, digits
                jne     ConvertLoop
```

Tu może kiedyś trafi dokładniejsze wytłumaczenie co się odpierdala, ale raczej to jest do ogarnięcia. UWAGA: jeśli będziecie sobie sprawdzać czy program działa to jeśli jakaś zmienna zajmuje 2 bajty pamięci to są one zapisane tak:

Dla przykładu liczbę: 4068

W rejestrach wyświetli to tak: AX: 0FE4

W pamięci wyświetli to tak: E4 0F

35 468

Punkt 3. Przekonwertowanie na 32-bity i dodanie tych wartości

Po poprzednim punkcie posiadamy 2 wartości. Są one zapisane kodem binarnym więc obydwie są dodatnie nawet jeśli użytkownik podał wartość ujemną. Zrobiłem tak dlatego bo jeśli chodzi o kod binarny nie ma zbytnio znaczenia ilu bitowy jest rejestr tylko jeśli wartość się w nim mieści tak w kodzie U2 rozmiar rejestru jest kluczowy ponieważ ostatni bit określa nam czy wartość jest plusowa czy minusowa.

$$1000_{(2)} = 8$$

$$0000\ 1000_{(2)} = 8$$

$$1000_{(U2)} = -8$$

$$0000\ 1000_{(U2)} = 8$$

Jeśli nie czaisz odsyłam do początku prezentacji lub wykładu czwartego Stokfisza.

Wiedząc to co napisałem wyżej oraz to, że aktualnie mamy zapisane 2 liczby dodatnie trzeba przekonwertować z ujemne wartości na U2. Trochę nad tym myślałem jak to zrobić, aż w końcu doszedłem do wniosku, że rozwiązanie jest banalne. To czego należy użyć zaczyna się na N, ma w sobie 2 litery G, jedną literę R oraz wykonuje ciężką pracę za zwykłego białego człowieka. Rozwiązaniem naszego problemu jest oczywiście intrukcja:
NEG REG

Gdzie NEG (negate) – tworzy z przyjętej wartości jej wartość na minusie (**w kodzie U2**)

REG (register) – jakiś rejestr którego wartość chcemy sobie zanegować

Kod który odwraca nam wartość gdy tego trzeba wygląda mniej więcej tak:

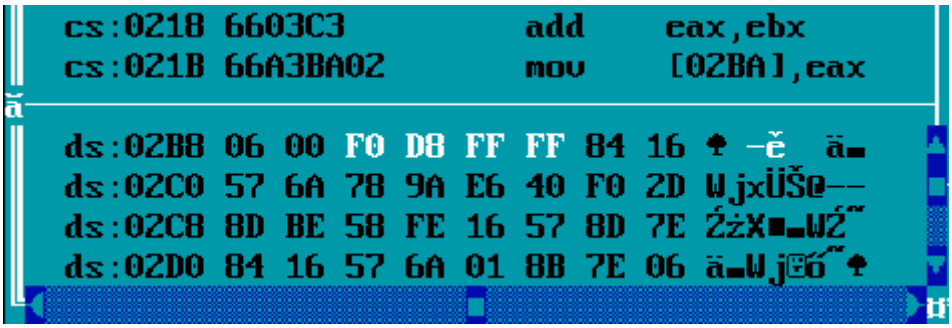
```
NegNumber:
                mov     bx, num
                cmp     isNegative, 1
                jne     AddNumbers
                neg     ebx

ContinueProg:
                Jakieś cosie
```

Jak można zauważyć używamy czegoś takiego jak rejestr EBX. E jest od słowa extended. Wszystkie rejestry dzielone na XH i XL można rozszerzyć z 18 bitów do 32 bitów.

Jeśli to co już powinno być zapisane jako liczba U2 jest zapisane jako U2 wystarczy dodać te 2 wartości:

```
AddNumbers:
                add     eax, ebx
                mov     wynik, eax
```

Przy dodawaniu nie ma znaczenia czy liczba jest po prostu zapisana w systemie dwójkowym czy U2 i tak dodaje się te 2 wartości tak samo i wynik wyjdzie dobry jak go dobrze przeliczymy. Dodając do siebie 5000 i -15000 otrzymałem taką wartość:



Jak wkleicie sobie to w kalkulator programistyczny na Windowsie to wyjdzie syf bo wynik jest w U2 a tam takich dobroci nie ma, ale spokojnie. Tu jest fajny kalkulator który znalazłem:

https://manderc.com/apps/umrechner/index_eng.php

Input:	
Dec	-10000
Hex	
Bin	
Oct	

	Two's Complement			
	Dec	Hex	Bin	Oct
8	-16	f0	11110000	360
16	-10 000	d8 f0	11011000 11110000	154360
32	-10 000	ff ff d8 f0	11111111 11111111 11011000 11110000	377 77754360
64	-10 000	ff ff ff ff ff ff d8 f0	11111111 11111111 11111111 11111111 11111111 11111111 11011000 11110000	177777 77777777 77754360
n	-10 000	d8 f0	11011000 11110000	154360

Pamiętajcie, że oktety (mądre informatyczne słowo) są czytane od końca dlatego w segmencie danych zaczyna się od F0 a nie od FF. Skoro mamy już wynik wystarczy teraz wyświetlić go na ekranie.

Punkt 4. Wyświetlanie wyniku na ekranie

Na początku warto sobie przygotować parę dodatkowych zmiennych.

```
msg3        DB      10, 13, "Suma podanych wartosci to: ", "$"
result      DD      ?
resultNeg   DB      0
resultASCII DB      6 DUP(?), "$"
```

msg3 – wiadomość, która wyświetli się przed wynikiem

result – wynik zapisany na 4 bajtach (dlatego typ zmiennej DD)

resultNeg – określenie czy wynik jest na minusie czy na plusie

resultASCII – tablica na znaki (jest miejsce na znak minusa i maksymalnie 5 innych znaków) zakończona jak każdy szanujący się string „$”

Kod zamieniający liczbę na ASCII zaczniemy od zamiany z kodu U2 na naturalny binarny, zaraz po dodaniu tych 2 liczb.

```
AddNumbers:
                add     eax, ebx
                jns     ConvertResult
                mov     resultNeg, 1
                neg     eax
```

Instrukcja JNS wykorzystuje flage SF, która określa czy wartość jest dodatnia (SF = 0) czy ujemna (SF = 1). Można się łatwo domyślić, że flaga SF jest taka sama jak pierwszy od lewej bit wartości.

Reszta kodu na następnej stronie żeby się wszystko zmieściło!

```
ConvertResult:
                mov     result, eax ; przekazujemy wynik do zmiennej
;idziemy na początek adresu zmiennej i skaczemy o 6 bo jest 6 znakow
                mov     bx, offset resultASCII
                add     bx, 6

                ; przez tyle będziemy dzielić bo system dziesiętny
                mov     ecx, 10
DivideLoop:
                ; zerujemy rejestr gdzie jest reszta z dzielenia
                xor     edx, edx
                ; dzielimy eax przez ecx (10)
                div     ecx
                ; dodajemy 30h do wyniku co odpowiada znaku 0 w ASCII
                add     dl, "0"
                ; cofamy się by dodać kolejne liczby od końca
                dec     bx
                mov     [bx], dl

                ; sprawdzamy czy mamy co jeszcze dzielic
                test    eax, eax
                jnz     DivideLoop
                ; sprawdzamy czy liczba byla ujemna
                cmp     resultNeg, 1
                jne     EndProg
                ; jesli byla dodajemy minus przed pierwszym znakiem
                dec     bx
                mov     dl, "-"
                mov     [bx], dl

EndProg:
                ; wyswietlanie wiadomosci
                mov     dx, offset msg3
                mov     ah, 09h
                int     21h
                ; wyswietlanie wyniku
                mov     dx, offset resultASCII
                mov     ah, 09h
                int     21h
                mov     ax, 4C00h ; zakonczenie programu
                int     21h
```

Punkt 5, który powinien być raczej częścią punktu 3 ale jebać to. Obsługa błędów:

Z tego co wiem to prowadzący laby zwracali uwagę na to czy program przyjmuje wartości, które wychodzą poza zakres czy nie oraz czy przyjmują -0.

Należy to zrobić kiedy nasze liczby już są zapisane w kodzie U2 w 32bitowych rejestrach, a kod który spradza czy dane liczby mieszczą się w zakresie wygląda następująco:

```
[...]; jakiś kod który powoduje że w rejestrach są liczby w U2

CheckNumbers:
                cmp     eax, -32768
                jl      ShowErrorMsg
                cmp     ebx, -32768
                jl      ShowErrorMsg
                cmp     eax, 32767
                jg      ShowErrorMsg
                cmp     ebx, 32767
                jg      ShowErrorMsg
                cmp     eax, 0
                je      NegativeZero1
                cmp     ebx, 0
                je      NegativeZero2
                jmp     AddNumbers

NegativeZero1:
                cmp     isNegative1, 1
                je      ShowErrorMsg
                cmp     ebx, 0
                je      NegativeZero2
                jmp     AddNumbers

NegativeZero2:
                cmp     isNegative2, 1
                je      ShowErrorMsg


AddNumbers:
[...]; kod, który dodaje do siebie 2 liczby i coś tam dalej robi
```

W tym fragmencie kodu użyłem przeskoków **JL** i **JG**. Są to przeskoki, które działają dla wartości zapisanych w kodzie U2. JL (**J**ump if **L**ower) to odpowiednik JB (**J**ump if **B**elow), a JG (**J**ump if **G**reater) to odpowiednik JA (**J**ump if **A**bove). Lista takich odpowedników skoków jest w instrukcji ale po co szukać w instrukcji jak mogę je wkleić poniżej:

Rozkaz	Opis
Liczby w kodzie naturalnym	
JB / JNAE	Skok jeśli mniejsze / jeśli nie większe i nie równe (CF=1)
JBE / JNA	Skok jeśli mniejsze lub równe / jeśli nie większe (CF=1 lub ZF=1)
JAE / JNB	Skok jeśli większe lub równe / jeśli nie mniejsze (CF=0)
JA / JNBE	Skok jeśli większe / jeśli nie mniejsze i nie równe (CF=0 lub ZF=0)
Liczby w kodzie U2	
JL / JNGE	Skok jeśli mniejsze / jeśli nie większe i nie równe (OF<>SF)
JLE / JNG	Skok jeśli mniejsze lub równe / jeśli nie większe (ZF=1 lub OF<>SF)
JGE / JNL	Skok jeśli większe lub równe / jeśli nie mniejsze (OF=SF)
JG / JNLE	Skok jeśli większe / jeśli nie mniejsze i nie równe (ZF=0 i OF=SF)
Liczby w kodzie naturalnym i w kodzie U2	
JE / JZ	Skok jeśli równe / jeśli zero (ZF=1)
JNE / JNZ	Skok jeśli nie równe / jeśli nie zero (ZF=0)

Stan konkretnego znacznika można zbadać również bezpośrednio i na tej podstawie wykonać bądź nie skok warunkowy. Rozkazy tego typu skoków warunkowych znajdują się w poniższej tabeli:

Rozkaz	Opis
JS	Skok jeśli ujemne (SF=1)
JNS	Skok jeśli nieujemne (SF=0)
JC	Skok jeśli wystąpiło przeniesienie (CF=1)
JNC	Skok jeśli nie było przeniesienia (CF=0)
JO	Skok jeśli wystąpił nadmiar (OF=1)
JNO	Skok jeśli nie było nadmiaru (OF=0)
JP / JPE	Skok jeśli parzysta liczba jedynek (PF=1)
JNP / JPO	Skok jeśli nieparzysta liczba jedynek (PF=0)

Jeśli wiemy, że zostały przekazane błędne wartości i program wykonał się nieprawidłowo to wypadało by zmienić zwracany kod wyjścia. Kod programu kończący go wygląda teraz mniej więcej tak:

```
ShowResult:
                ; wyswietlanie wiadomosci
                mov     dx, offset msg3
                mov     ah, 09h
                int     21h
                ; wyswietlanie wyniku
                mov     dx, offset resultA
                mov     ah, 09h
                int     21h

                mov     al, 00h
                jmp EndProg

ShowErrorMsg:
                ; Wyświetlanie tekstu
                mov     dx, offset msg4
                mov     ah, 09h
                int     21h
                mov     al, 01h ; ustawianie wartości kodu wyjscia
```

```
EndProg:
                mov     ah, 4Ch ; zakonczenie programu
                int     21h


msg3        DB      10, 13, "Suma podanych wartosci to: ", "$"
msg4        DB      10, 13, "Podales bledna wartosc. Program sie
konczy.", 10, 13,"$"
```

Koniec

Chciałbym tutaj oficjalnie podziękować Kazowi Bałagane. Gdyby nie on nie było by mnie w miejscu w którym jestem.

Jak ktoś ma jakieś sugestie to chętnie przyjmę, wielkie pozdro.