ROZDZIAL PIERWSZY: **REPREZENTACJA DANYCH**

Prawdopodobnie największą przeszkodą jaką większość początkujących napotyka kiedy próbuje nauczyć się asemblera jest powszechne używanie binarnego i heksadecymalnego systemu liczbowego. Wielu programistów sądzi że liczby heksadecymalne stanowią absolutny dowód na to, że Bóg nigdy nie planował aby ktokolwiek pracował w asemblerze. Chociaż to prawda, że liczby heksadecymalne trochę się różnią od tego czego używamy na co dzień, ich zalety znacznie przewyższają ich wady. Pomimo to, opanowanie tych systemów liczbowych jest ważne ponieważ ich zrozumienie upraszcza inne ważne tematy wliczając w to algebrę boolowską i projekt logiczny, reprezentację numeryczną znaków, kody znaków i dane upakowane.

1.0 WSTĘP

Ten rozdział omawia kilka ważnych pojęć, w tym binarny i heksadecymalny system liczbowy, organizację danych binarnych (bity ,nibbles ,bajty ,słowa i podwójne słowa),system liczb ze znakiem i bez znaku, operacje arytmetyczne, logiczne, przesunięcia i obroty na wartościach binarnych, pola bitów i pakowanie danych ,zbiór znaków ASCII. Jest to podstawowy materiał a dalsze czytanie tego tekstu zależy od zrozumienia tych pojęć. Jeśli już jesteś zapoznany z tymi terminami z innych kursów lub studiów powinieneś przynajmniej przejrzeć ten materiał przed przystąpieniem do następnego rozdziału. Jeśli nie jesteś zapoznany lub tylko ogólnikowo ,powinieneś przestudiować go starannie. CAŁY MATERIAŁ W TYM ROZDZIALE JEST WAŻNY! Nie opuszczaj nadmiernie materiału.

1.1 SYSTEMY LICZBOWE

Większość nowoczesnych systemów komputerowych nie przedstawia wartości liczbowych używając systemu dziesiętnego. Zamiast tego, używają systemu binarnego lub systemu "dopełnienia do dwóch" .Żeby zrozumieć ograniczenia arytmetyki komputerów musimy zrozumieć w jaki sposób komputery przedstawiają liczby.

1.1.1 PRZEGLĄD SYSTEMU DZIESIĘTNEGO

Używasz systemu dziesiętnego (opartego o 10) od tak dawna, że uważasz go za pewnik. Kiedy widzisz liczbę taką jak *123*,nie myślisz o wartości 123;raczej stwarzasz umysłowy obraz tego jaką pozycję ta wartość przedstawia. W rzeczywistości, liczba 123 przedstawia

$$1*10^2+2*10^1+3*10^0$$

lub

100+20+3

Każda cyfra pojawiająca się po lewej stronie przecinka przyjmuje wartość między zero a dziewięć mnożone przez dodatnie potęgi liczby dziesięć. Cyfra pojawiająca się po prawej stronie przecinka przyjmuje wartości między zero a dziewięć mnożone przez rosnące, ujemne potęgi liczby dziesięć. Na przykład wartość 123,456 to:

$$1*10^{2}+2*10^{1}+3*10^{0}+4*10^{-1}+5*10^{-2}+6*10^{-3}$$

lub

100+20+3+0.4+0.05+0.006

1.1.2 BINARNY SYSTEM LICZBOWY

Większość nowoczesnych systemów komputerowych (w tym IBM PC) działa używając logiki binarnej. Komputer przedstawia wartości używając dwóch poziomów napięcia (zwykle 0V i 5V).Poprzez dwa takie poziomy możemy przedstawić dokładnie dwie rożne wartości. To mogłyby być dowolne dwie rożne wartości ale konwencjonalnie używamy wartości zero i jeden. Te dwie wartości, zbiegiem okoliczności, odpowiadają dwom cyfrom używanym przez binarny system liczbowy. Ponieważ jest zgodność między logicznymi poziomami używanymi przez 80x86 i dwoma cyframi używanymi w binarnym systemie liczbowym, nie było niespodzianki, ze IBM PC zastosował binarny system liczbowy. Binarny system liczbowy pracuje tak jak system dziesiętny z dwoma wyjątkami: system binarny uznaje tylko cyfry 0 i 1 (zamiast 0-9) i system binarny używa potęgi dwa zamiast potęgi dziesięć. Dlatego tez jest bardzo łatwo przekształcić liczbę binarna na dziesiętną. Dla każdego "1" lub ”0” w łańcuchu binarnym dodajemy 2^n gdzie n jest pozycja cyfry binarnej liczonej od prawej strony (zera)Na przykład binarna wartosc$11001010_{(2)}$ przedstawia się tak:

$$1*2^7+1*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0$$

lub

$$8+64+8+2$$
$$=$$
$$202_{(10)}$$

Przekształcenie liczby dziesiętnej na binarna jest odrobinę bardziej skomplikowane Musisz znaleźć te potęgi dwójki, które dodane razem stworzą rezultat dziesiętny. Najłatwiejsza metoda to zacząć od dużych potęg dwójki aż do 2^0.Rozważmy dziesiętną wartość 1359:

* 2^{10}=1024. 2_{11}=2048. 1024 jest największą potęga dwójki mniejszą niż 1359. Odejmujemy 1024 od 1359 i zaczynamy wartość binarna od lewej cyfra "1".Binarnie = "1".Resultat dziesiętny to 1359-1024=335.
* Kolejna ,niższa potęga dwójki (2^9=512) jest większa niż powyższy rezultat, więc dodajemy "0" na koniec binarnego łańcucha. Binarnie="10",dziesiętnie jest wciąż 335
* Kolejną, niższą potęgą dwójki jest 256 (2^8).Odejmujemy ją od 335 i dodajemy cyfrę "1" na koniec binarnej liczby.Binarnie="101",rezultat dziesiętny to 79.
* 128 (2^7) jest większe niż 79 więc dołączamy "0" do końca łańcucha binarnego. Binarnie ="1010",rezultat dziesiętny pozostaje 79.
* Następna niższa potęga dwójki (2^6=64) jest mniejsza niż 79,wiec odejmujemy 64 i dołączamy "1" do końca binarnego łańcucha. Binarnie ="10101".Rezultat dziesiętny to 15
* 15 jest mniejsze niż następna potęga dwójki (2^5=32) więc po prostu dodajemy "0" do końca łańcucha binarnego.Binarnie="101010".Dziesietny rezultat to wciąż 15.
* 16 (2^4) jest większa niż dotychczasowa reszta więc dołączamy "0" do końca łańcucha binarnego.Binarnie="1010100",rezultat dziesiętny to 15.
* 2^3 (osiem) jest mniejsze niż 15 więc dokładamy następna cyfrę "1" na koniec binarnego lancucha.Binarnie="10101001",dziesiętnie rezultat to 7.
* 2^2 jest mniejsze niż 7 więc odejmujemy cztery od siedmiu i dołączamy następną jedynkę do binarnego lancucha.Binarnie="101010011",dziesietnie jest 3.
* 2^1 jest mniejsze niż 3 więc dodajemy jedynkę do łańcucha binarnego i odejmujemy dwa od wartości dziesiętnej.Binarnie="1010100111",dziesiętny rezultat to teraz 1.
* Ostatecznie rezultat dziesiętny wynosi jeden (2^0) wiec dodajemy końcową "1" na koniec łańcucha binarnego. Końcowy rezultat binarny to: 10101001111

Liczby binarne, mimo iż maja małe znaczenie w językach programowania wysokiego poziomu, pojawiają się wszędzie w programach pisanych w asemblerze.

1.1.3 FORMAT DWÓJKOWY

Każda binarna liczba zawiera bardzo duża liczbę cyfr (lub bitów - skrót od BInary digiTs).Na przykład ,przedstawiamy liczbę pięć poprzez:

101 00000101 0000000000101 000000000000101

Dowolna liczba zera może poprzedzać liczę binarną bez zmiany jej wartości. Przyjmiemy konwencję ignorowania poprzedzających zer. Na przyklad,$101_{(2)}$ przedstawia liczbę pięć .Ponieważ 80x86 pracuje z grupami ośmiobitowymi, przyjdzie nam dużo łatwiej rozszerzyć o zera wszystkie liczby binarne jako wielokrotności czterech lub ośmiu bitów. Dlatego tez podążając za konwencja, przedstawimy liczbę pięć jako $0101_{(2)}$ lub $00000101_{(2)}$.

W USA większość ludzi oddziela każde trzy cyfry przecinkiem, co czyni duże liczby łatwiejszymi do odczytu, na przykład 1,023,435,208 jest dużo łatwiejsze do przeczytania i pojęcia niż 1023435208.Przyjmiemy podobną koncepcje w tym tekście dla liczb binarnych. Oddzielimy każdą grupę czterech bitów spacją. Na przykład binarną wartość 1010111110110010 zapiszemy jako 1010 1111 1011 0010.
Często pakujemy kilka wartości razem do tej samej liczby binarnej. Jedna z form instrukcji MOV 80x86 używa binarnego kodowania 1011 0rrr dddd dddd dla spakowania trzech pozycji do 16 bitów; pięć bitów kodu operacji (10110),trzy bity pola rejestrów (rrr) i osiem bitów wartości bezpośredniej (dddd dddd).Dla wygody, przydzielimy wartości liczbowe każdej pozycji bitu .Ponumerujemy każdy bit jak następuje:
1) Bit najbardziej na prawo jest bitem z pozycji zero.
2) Każdy bit na lewo ma kolejny ,większy numer
Ośmiobitowa wartość binarna używa bitów od zero do siedem:

$$x_7,x_6,x_5,x_4,x_3,x_2,x_1,x_0$$

Szesnastobitowa wartość binarna używa bitów od zera do piętnastu:

$$x_{15},x_{14},x_{13},x_{12},x_{11},x_{10},x_9,x_8,x_7,x_6,x_5,x_4,x3,x_2,x_1,x_0$$

Do bitu zerowego zazwyczaj odnosimy się jako najmniej znaczącego bitu (L.O). Bit najbardziej z lewej strony jest zwykle nazywany bitem najbardziej znaczącym (H.O.). Będziemy się odnosili do bitów pośrednich poprzez ich numery.

1.2 ORGANIZACJA DANYCH

W czystej matematyce wartości mogą zawierać przypadkowe liczby bitów. Komputery ,generalnie rzecz biorąc pracują z określoną liczbą bitów. Powszechnie przyjęte to pojedyncze bity, grupy czterech bitów (zwane nibbles),grupy ośmiu bitów (zwane bajtami),grupy szesnastu bitów (zwane słowem) i więcej. Ich rozmiary nie są przypadkowe. Ta sekcja opisze grupy bitów powszechnie używanych w chipach Intela 80x86

1.2.1 BITY

Najmniejsza "jednostka" danych w komputerze jest pojedynczy bit. Ponieważ pojedynczy bit jest zdolny do przedstawiania tylko dwóch rożnych wartości (typowo zero i jeden),możemy odnieść wrażenie, ze jest bardzo mało wartości jakie może przedstawić pojedynczy bit. Nie prawda! Jest ogromna liczba pozycji jakie możemy przedstawić za pomocą pojedynczego bitu.
Za pomocą pojedynczego bitu możemy przedstawić dwie odrębne pozycje. Przykładem mogą być: zero lub jeden, prawda lub fałsz, włączony lub wyłączony, mężczyzna lub kobieta, Jednakże nie jesteśmy ograniczeni do przedstawiania typów danych binarnych (to znaczy tych obiektów które maja dwie rożne wartości). Możesz używać pojedynczego bitu do przedstawiania liczb 723 i 1,245 lub 6.254 i 5. Możesz również użyć pojedynczego bitu do przedstawiania kolorów czerwonego i niebieskiego. Możesz nawet przedstawić dwa nie powiązane ze sobą obiekty za pomocą pojedynczego bitu. Na przykład, możesz przedstawić kolor czerwony i liczbę 3.256 za pomocą pojedynczego bitu. Możesz przedstawić jakieś dwie rożne wartości za pomocą pojedynczego bitu. Jednakże możesz przedstawić tylko dwie rożne wartości za pomocą pojedynczego bitu.
To mylące rzeczy, nawet bardzo, rożne bity mogą przedstawiać rożne rzeczy. Na przykład jeden bit może być używany do przedstawiania wartości zero i jeden, podczas gdy sąsiedni bit może być używany do przedstawiania wartości prawda i fałsz .Jak można to odróżnić patrząc na te bity? Odpowiedź, nie można. Ale to ilustruje cała ideę komputerowej struktury danych: dana jest to to co ty zdefiniujesz. Jeśli użyjesz bitu do przedstawienia boolowskiej (prawda/ fałsz) wartości wówczas ten bit (zdefiniowany przez ciebie) reprezentuje prawdę lub fałsz. Dla bitów mających prawdziwe znaczenie musisz być konsekwentny. To znaczy ,jeśli używasz bitu do przedstawiania prawdy lub fałszu w jednym punkcie swojego programu nie powinieneś używać wartości prawda/fałsz przechowywanej w tym bicie do późniejszego przedstawiania koloru czerwonego lub niebieskiego.
Ponieważ większość danych których będziesz używał, wymaga więcej niż dwóch rożnych wartości, pojedyncze wartości bitów nie są najbardziej powszechnymi typami danych, które będziesz stosował. Ponieważ wszystko inne składa się z grup bitów, bity odgrywają ważną role w twoich programach. Oczywiście, jest kilka typów danych które wymagają dwóch odrębnych wartości, wiec wydaje się, ze bity są ważne same w sobie. Jednak wkrótce zobaczysz, że pojedyncze bity są trudne do manipulowania, wiec często będziemy używać innych typów danych do przedstawiania wartości boolowskich.

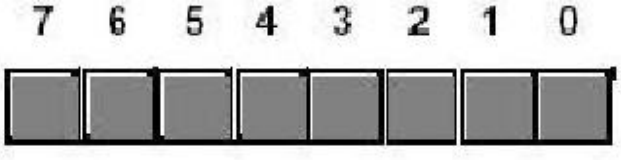
1.2.2 NIBBLESY

Nibble jest zbiorem czterech bitów. To nie jest szczególnie interesująca struktura danych za wyjątkiem dwóch przypadków: liczb BCD (Binary Coded Decimal) i liczb heksadecymalnych. Cztery bity przedstawiają pojedynczą cyfrę BCD lub heksadecymalną. Przy pomocy nibble'a możemy przedstawić do 16 odrębnych wartości. W przypadku liczb heksadecymalnych, wartości 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E i F są przedstawiane przez cztery bity (zobacz "Heksadecymalny System Liczbowy").BCD używa 10 rożnych cyfr (0,1,2,3,4,5,6,7,8,9) i wymaga czterech bitów. Faktycznie, każda z szesnastu odrębnych wartości może być przedstawiana przez nibble'a, ale cyfry heksadecymalne i BCD są podstawowymi pozycjami jakie możemy przedstawić za pomocą pojedynczego nibble'a.

1.2.3 BAJTY

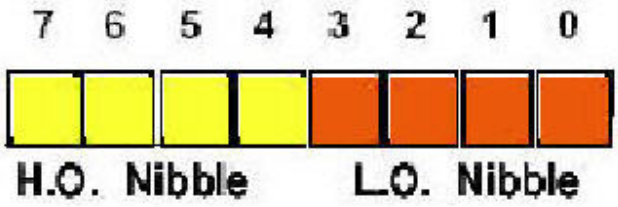
Bez wątpliwości, najważniejszą strukturą danych używaną przez mikroprocesor 80x86 jest bajt .Bajt składa się z ośmiu bitów i jest najmniejszym adresowalnym elementem danych w mikroprocesorze 80x86.Adresy pamięci głównej jak i I/O w 80x86 wszystkie są adresami bajtowymi. To znaczy, ze najmniejsza pozycja która może być dostępna przez program w 80x86 jest wartością ośmiobitową. Dostęp do czegokolwiek mniejszego wymaga abyś odczytał bajt zawierający dane i zamaskował niechciane bity. Bity w bajcie są zazwyczaj numerowane od zera do siedem, przy użyciu konwencji przedstawionej na rysunku 1.1.

Bit 0 jest najmniej znaczącym bitem, bit 7 jest najbardziej znaczącym bitem. Do pozostałych bitów będziemy się odwoływać poprzez ich numery.



Rysunek 1.1 Numerowanie bitów w bajcie

Zauważ, że bajt także zawiera dwa nibble (zobacz rysunek 1.2)



Rysunek 1.2 Dwa nibble w bajcie

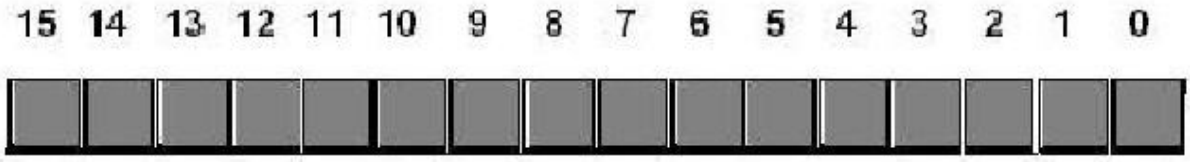
Bity 0..3 stanowią mniej znaczącego nibble'a, bity 4..7 bardziej znaczącego nibble'a. Ponieważ bajt zawiera dokładnie dwa nibble, wartość bajtu wymaga dwóch cyfr heksadecymalnych. Ponieważ bajt zawiera osiem bitów, można przedstawić 2^8 ,lub 256, rożnych wartości. Generalnie będziemy używać bajtu do przedstawiania wartości liczbowych w zakresie 0...255,liczb ze znakiem w zakresie -128..+127 (zobacz "Liczby ze znakiem i bez znaku"),kodów znaków ASCII/IBM, i innych specjalnych typów danych wymagających nie więcej niż 256 rożnych wartości. Wiele typów danych ma mniej niż 256 pozycji, więc osiem bitów jest zazwyczaj wystarczających.

Ponieważ 80x86 jest maszyną adresowalna bajtem (zobacz "Układ Pamięci i Dostęp") okazuje się bardziej efektywne manipulowanie całym bajtem niż pojedynczym bitem czy nibble'm. Z tego powodu większość programistów używa całego bajtu do przedstawienia typu danych który wymaga nie więcej niż 256 pozycji, nawet jeśli mniej niż osiem bitów wystarczyłoby. Na przykład, często przedstawiamy wartości boolowskie prawdę i fałsz (odpowiednio) jako $00000001_{(2)}$ i $00000000_{(2)}$.

Prawdopodobnie najważniejszym zastosowaniem bajtu jest udział w kodzie znaku. Znaki pisane z klawiatury, wyświetlane na ekranie, i drukowane na drukarce, wszystkie mają wartości liczbowe. Pozwala to porozumiewać się z resztą świata, IBM PC używa wariantu ze znakami ASCII (zobacz "Zbiór znaków ASCII"). Jest 128 zdefiniowanych kodów w zbiorze znaków ASCII.IBM używa pozostałych 128 możliwych wartości dla rozszerzonego zbioru kodów znaków wliczając w to znaki europejskie ,symbole graficzne, litery greckie i symbole matematyczne.

1.2.4 SŁOWA

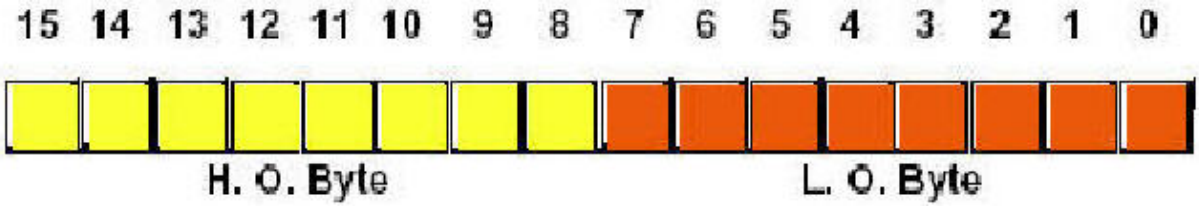
Słowo jest grupą 16 bitów. Ponumerujemy te bity zaczynając od zera w gore aż do piętnastu. Numeracja bitów pokazana jest na rysunku 1.3



Rysunek 1.3 Numeracja bitów w słowie

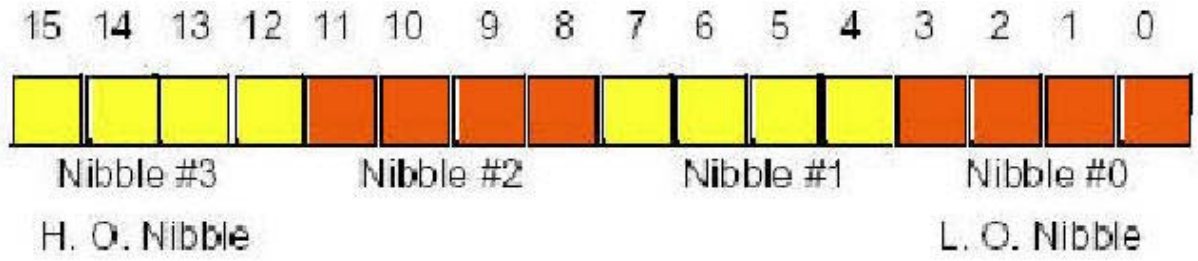
Tak jak przy bajcie, bit 0 jest najmłodszym bitem a bit 15 najstarszym bitem. Kiedy odnosimy się do innych bitów w słowie używamy ich numerów pozycji.

Zauważ, ze słowo zawiera dokładnie dwa bajty. Bity 0 do 7 tworzą mniej znaczący bajt, bity od 8 do 15 tworzą bardziej znaczący bajt.(Zobacz rysunek 1.4)



Rysunek 1.4 Dwa bajty w słowie

Naturalnie, słowo może być dalej rozbite na cztery nibble jak pokazuje rysunek 1.5



Rysunek 1.5 Nibble w słowie

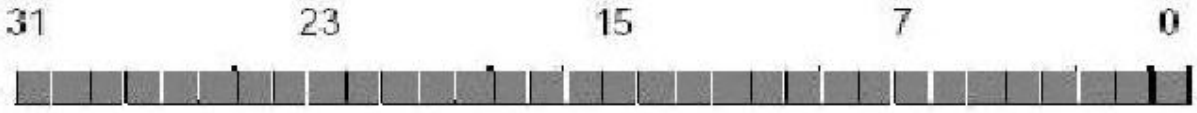
Nibble zero jest najmniej znaczącym nibblem w słowie a nibble trzy najbardziej znaczącym nibblem słowa. Pozostałe dwa nibble to "nibble jeden" i nibble dwa".

Mając 16 bitów możemy przedstawić 2^{16} (65,536) rożnych wartości. To mogą być wartości w zakresie od 0 do 65 536 (lub zazwyczaj -32,768..+32,767) lub każdy inny typ danych o wartościach nie większych niż 65,536.Trzy ważne zastosowania dla słowa to wartości liczb całkowitych, offsetów i wartości segmentów (zobacz "Układ Pamięci i Dostęp" przy opisie segmentów i offsetów).

Słowa mogą reprezentować wartości całkowite w zakresie 0..65 536 lub -32,768..+32,767.Wartości liczb bez znaku są reprezentowane przez wartości binarne zgodne z bitami w słowie. Wartości liczb ze znakiem używają formy „dopełnienia do dwóch" dla wartości liczbowych (zobacz "Liczby ze znakiem i bez znaku").Wartości segmentu, które są zawsze długie na 16 bitów stanowią adres paragrafu kodu, danych, danych specjalnych lub segmentu stosu w pamięci.

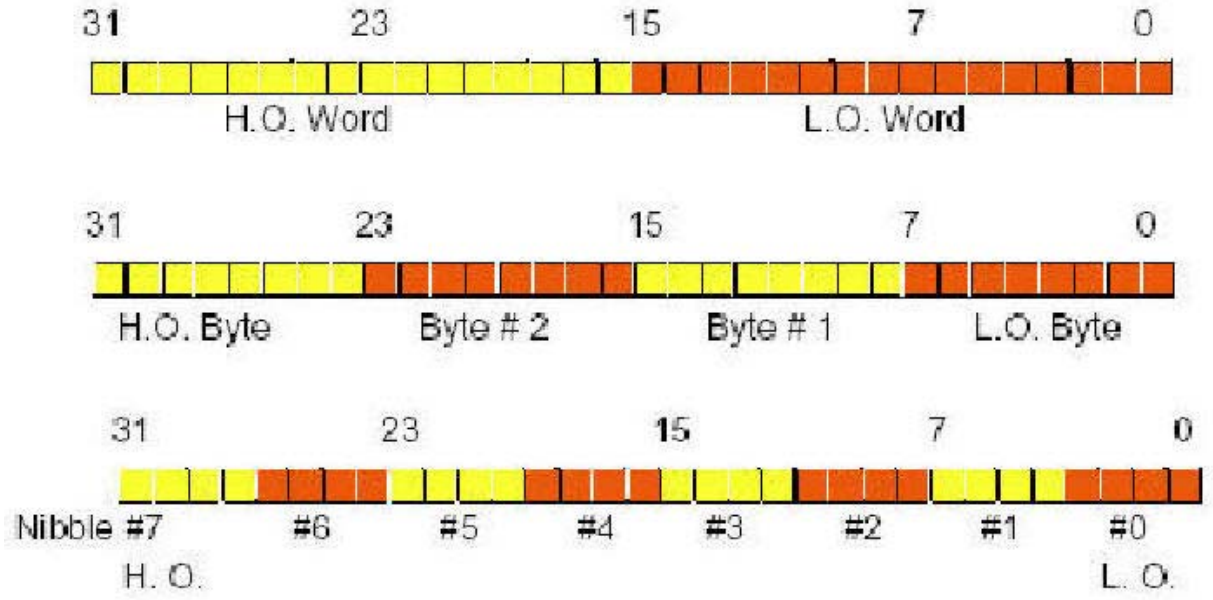
1.2.5 PODWÓJNE SLOWO

Podwójne słowo jest dokładnie tym, na co wskazuje jego nazwa, parą słów .Dlatego tez długość podwójnego słowa wynosi 32 bity ,jak pokazuje rysunek 1.6.



Rysunek 1.6 Liczba bitów w podwójnym słowie

Naturalnie, to podwójne słowo może być dzielone na słowo wyższego rzędu i słowo niższego rzędu, lub cztery rożne bajty, lub osiem rożnych nibbli (zobacz rysunek 1.7).Podwójne słowa mogą reprezentować wiele rodzajów rożnych rzeczy .Przede wszystkim na liście jest adresowanie segmentu. Inna powszechną pozycją reprezentowaną przez podwójne słowo jest 32 bitowa



Rysunek 1.7 Nibble, bajty i słowa w podwójnym słowie

wartość całkowita (która określa liczby bez znaku w zakresie 0..4,294,967,295 lub liczby ze znakiem w zakresie -2,147,483,648..2,147,483,647).32-bitowa wartość zmiennoprzecinkowa także mieści się w podwójnym słowie. Większość czasu będziemy używać podwójnych słów dla adresowania segmentów.

1.3 HEKSADECYMALNY SYSTEM LICZBOWY

Sprawę z binarnym systemem już omówiliśmy. Przedstawienie wartości $202_{(10)}$ wymaga ośmiu cyfr binarnych. Wersja dziesiętna wymaga tylko trzech cyfr dziesiętnych, tak wiec przedstawianie liczb jest dużo łatwiejsze niż w przypadku systemu binarnego. Ten fakt nie umknął uwadze inżynierów którzy projektowali komputerowy system binarny. Kiedy pracowali z dużymi wartościami, liczby binarne szybko stały się zbyt nieporęczne .Niestety, komputer myśli binarnie, większość czasu ,w dogodnym w użyciu binarnym systemie liczbowym. Mimo, że umiemy konwertować między systemem dziesiętnym a systemem dwójkowym, przeliczanie nie jest błahym zadaniem. Heksadecymalny system liczbowy (oparty o 16) rozwiązuje te problemy. .Liczby heksadecymalne oferują dwie cechy ,których szukamy: są niewielkich rozmiarów i prosto zamienia się je na liczby binarne i vice versa .Z powodu tego, większość binarnych systemów komputerowych dzisiaj używa heksadecymalnego systemu liczbowego. Ponieważ podstawą liczby heksadecymalnej jest 16,każda heksadecymalna cyfra przedstawia jakąś wartość którą mnożymy przez kolejną potęgę liczby 16.Na przykład

liczba $1234_{(16)}$ równa się:

$$1*16^3*16^23*16^1*16^0$$

lub

$$4096+512+48+4 = 4660_{(10)}$$

Każda cyfra heksadecymalna może reprezentować jedną z szesnastu wartości między 0 a $15_{(10)}$.Ponieważ jest tylko 10 cyfr dziesiętnych, musimy wymyślić sześć dodatkowych cyfr dla przedstawienia wartości w zakresie $10_{(10)}$ do

$15_{(10)}$ ..Zamiast tworzyć nowe symbole dla tych cyfr, użyjemy liter od A do F. Wszystkie niżej przedstawione wyrażenia są przykładami prawidłowych heksadecymalnych liczb:

$1234_{(16)}$ $DEAD_{(16)}$ $BEEF_{(16)}$ $0AFB_{(16)}$ $FEED_{(16)}$ $DEAF_{(16)}$

Ponieważ będziemy musieli często wprowadzać liczby heksadecymalne do systemu komputerowego, będziemy potrzebować rożnych mechanizmów dla przedstawiania liczb heksadecymalnych. W końcu, w większości systemów komputerowych nie można wprowadzać indeksów dolnych oznaczających rodzaj wprowadzanej liczby. Przyjmiemy następującą konwencję:

* Wszystkie wartości liczbowe (bez względu na ich indeks) zaczynamy cyfra dziesiętną
* Wszystkie wartości heksadecymalne kończymy literą "h" np. 123A4h
* Liczby dziesiętne mogą mieć przyrostek "t" lub "d"

Przykłady prawidłowych liczb heksadecymalnych:

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

Więc jak widzisz, liczby heksadecymalne są niewielkich rozmiarów i łatwe do odczytu. W dodatku możesz łatwo przekształcać między liczbami heksadecymalnymi a binarnymi. Rozważ następująca tablice:

Binarnie	Heksadecymalnie
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Tablica 1 Liczby binarne i heksadecymalne

Ta tablica dostarcza wszystkich informacji których będziesz potrzebował przy konwersji liczby heksadecymalnej na binarną i vice versa. Konwersja liczby heksadecymalnej na binarną polega na zamianie odpowiednich czterech bitów każdej heksadecymalnej cyfry na liczbę. Na przykład, konwersja 0ABCDh na wartość binarną, polega na konwersji każdej heksadecymalnej cyfry według powyższej tabeli:

0 A B C D heksadecymalnie

0000 1010 1011 1100 1101 binarnie

Konwersja liczby binarnej na format heksadecymalny jest prawie tak samo łatwa. Pierwszy krok to dodanie zer do liczby binarnej aby upewnić się, ze występuje wielokrotność czterech bitów w liczbie. Na przykład, mamy daną liczbę binarna 1011001010,pierwszym krokiem będzie dodanie dwóch bitów z lewej strony liczby, tak aby zawierała 12 bitów. Konwertowana wartość binarna to 001011001010.Następny krok to rozdzielenie wartości binarnej w grupy czterobitowe np. 0010 1100 1010.Na koniec sprawdzamy te wartości binarne w powyższej tablicy i zastępujemy je właściwymi cyframi heksadecymalnymi np. 2CA.Porównaj to z trudnościami przy konwersji między liczbami dziesiętnymi a binarnymi, czy dziesiętnymi i heksadecymalnymi. Ponieważ konwertowanie między liczbami heksadecymalnymi a binarnymi jest operacją, którą będziesz wykonywał wielokrotnie, wiec poświęć kilka chwil i naucz się powyższej tabelki na pamięć. Nawet jeśli masz kalkulator, który zrobi te konwersje dla ciebie, odkryjesz, że konwersja ręczna jest dużo szybsza i bardziej dogodna kiedy konwertujesz pomiędzy liczbami binarnymi i heksadecymalnymi.

1.4 OPERACJE ARYTMETYCZNE NA LICZBACH BINARNYCH I HEKSADECYMALNYCH

Jest kilka operacji, które możemy wykonać na liczbach binarnych i heksadecymalnych. Na przykład, możemy dodawać, odejmować, mnożyć, dzielić i wykonać inne operacje arytmetyczne. Mimo, że nie musisz zostać ekspertem w tej dziedzinie, powinieneś, umieć wykonać te operacje ręcznie używając kawałka papieru i ołówka. Mając powiedziane, ze powinieneś wykonać te operacje ręcznie, właściwym sposobem wykonania takiej operacji jest posiadanie kalkulatora który wykona to za ciebie. Jest kilka takich kalkulatorów na rynku; lista poniżej przedstawia kilku producentów którzy produkują takie urządzenia:
Producenci kalkulatorów heksadecymalnych:
* Casio
* Hewlett-Packard
* Sharp
* Texas Instruments
Ta lista nie jest wyczerpująca. Kalkulatory innych producentów są prawdopodobnie równie dobre.
Urządzenia Hewlett-Packarda są być może najlepsze w swojej grupie ,jednakże są one droższe niż inne .Sharp i Casio wytwarzają kalkulatory, które sprzedają poniżej 50$.Jeśli spodziewasz się napisać program w języku asemblera, posiadanie jednego z tych kalkulatorów jest niezbędne. Alternatywa dla zakupu kalkulatora heksadecymalnego jest posiadanie programu TSR firmy SideKick, który zawiera wbudowany kalkulator.
Jednakże, o ile nie masz już jednego z tych programów, lub potrzebujesz kilku innych cech, które oferują ,takie programy nie mają szczególnej wartości, ponieważ kosztują więcej niż kalkulator i nie są tak dogodne w użyciu.
Aby zrozumieć dlaczego wydasz pieniądze na kalkulator rozważ następujący problem arytmetyczny:

```
  9h
+ 1h
----
```

kusi cię, żeby napisać odpowiedź "10h" jako rozwiązanie tego problemu. To nie jest prawidłowo! Prawidłowa odpowiedź to dziesięć, ale zapisana jako "0Ah","10h" nie jest prawidłowym zapisem heksadecymalnym.
Podobny problem występuje w tym arytmetycznym problemie:

```
  10h
- 1h
-----
```

prawdopodobnie kusi cię odpowiedz "9h" pomimo, że prawdziwa odpowiedz to "0Fh".Pamiętaj, ten problem to pytanie "jaka jest różnica między szesnaście a jeden?" Odpowiedź: oczywiście, to piętnaście zapisane "0Fh".
Nawet jeśli te dwa przykłady nie zaniepokoiły cię, w sytuacji stresowej twój mózg wróci do trybu dziesiętnego podczas pracy nad czymś istotnym i twoja praca przyniesie błędne rezultaty. Morał z tej historii - jeśli musisz robić arytmetyczne wyliczenia używaj liczb heksadecymalnych ręcznie, poświęć swój czas i bądź przy tym ostrożny.
Nigdy nie będziesz wykonywał obliczeń w arytmetyce binarnej. Ponieważ liczby binarne zwykle zwierają długie łańcuchy bitów, wiec jest duża możliwość, że popełnisz błąd. Zawsze przekształcaj liczby binarne na heksadecymalne, wykonaj operacje na heksadecymalnych (najlepiej kalkulatorem heksadecymalnym) i przekształć wynik z powrotem na liczbę binarna ,jeśli to konieczne.

1.5 OPERACJE LOGICZNE NA BITACH

Są cztery główne operacje logiczne, które możemy wykonać na liczbach heksadecymalnych i binarnych: AND,OR,XOR (exlusive-or) i NOT.W odróżnieniu od operacji arytmetycznych kalkulator heksadecymalny nie jest konieczny do wykonania tych operacji. Jest to często łatwiejsze do zrobienia ręcznie niż przy użyciu do obliczeń urządzeń elektronicznych. Operacja logiczna AND jest operacją na liczbach binarnych (operuje na dwóch operandach) Są to operacje na pojedynczych bitach. Operacja AND:

```
0 AND 0=0
0 AND 1=0
1 AND 0=0
1 AND 1=1
```

Prostym sposobem dla przedstawienia logicznej operacji AND jest tabela prawdy. Tabela prawdy AND wygląda następująco :

AND	0	1
0	0	0
1	0	1

Tablica 2: Tabela prawdy AND

Wygląda to jak tabliczka mnożenia z którą zetknęliście się w szkole podstawowej. Kolumna z lewej strony i wiersz na górze reprezentują dane wejściowe operacji AND. Wartości umieszczone na przecięciu się wiersza i kolumny (dla poszczególnych par wartości wejściowych) są wynikiem logicznego ANDowania tych dwóch wartości razem. Po angielsku operacja AND : "Jeśli pierwszy operand równa się jeden i drugi operand równa się jeden ,wtedy wynik równa się jeden; w przeciwnym wypadku wynik równa się zero"
Jednym ważnym faktem godnym odnotowania przy logicznej operacji AND, jest to, ze możesz użyć jej do wymuszenia wyniku zero. Jeśli jeden z operandów równa się zero, wynik zawsze jest zero bez względu na drugi operand. W tablicy prawdy powyżej, np. wiersz z zerową wartością wejściową zawiera tylko zera, a kolumna zawierającą zero zawiera wynik zerowy. Odwrotnie, jeśli jeden operand zawiera jeden wynik jest dokładnie wartością drugiego operandu. Ta cecha operacji AND jest bardzo ważna, szczególnie kiedy pracujesz z łańcuchem bitów i chcesz ustawić pojedynczy bit w łańcuchu na zero. Zbadamy to zastosowanie logicznej operacji AND w następnej sekcji.
Logiczna operacja OR jest także operacja na bitach. Jej definicja:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

Tablica prawdy dla operacji OR przybiera następującą formę:

OR	0	1
0	0	1
1	1	1

Tablica 3: Tablica prawdy OR

Potocznie, operacja logiczna OR: "jeśli pierwszy operand lub drugi operand (lub oba) mają wartość jeden, wynik wynosi jeden, w przeciwnym razie wynik równa się zero." Jest to również znane jako operacja inclusive-OR.
Jeśli jeden z operandów operacji logicznej OR równa się jeden, wynik zawsze wynosi jeden bez względu na wartość drugiego operandu. Tak jak w logicznej operacji AND, jest to ważna cecha logicznej operacji OR, która udowadnia całkowitą przydatność podczas pracy z łańcuchami bitów.(zobacz następną sekcję).
Odnotujmy, że jest różnica pomiędzy tą forma operacji logicznej OR a standardowym znaczeniem angielskim. Rozważmy takie zdanie: "Idę do sklepu LUB Idę do parku". Taka wypowiedz wskazuje, że mówca idzie do sklepu lub do parku, ale nie do obu miejsc naraz. Zatem, angielska wersja logicznego OR jest odrobinę rożna niż operacja inclusive-OR, rzeczywiście bliżej jej do operacji exclusive-OR.
Logiczna operacja XOR (exclusive-OR) jest również operacja na bitach. Jej definicja znajduje się poniżej:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

Tablica prawdy dla operacji XOR przybiera następującą formę:

XOR	0	1
0	0	1
1	1	0

Tablica 4: Tablica prawdy XOR

Operacja logiczna XOR: "jeśli pierwszy operand lub drugi operand, ale nie obydwa równocześnie, równają się jeden wynik równa się jeden; w przeciwnym razie wynik równa się zero" Zauważ, ze operacja exclusive-OR jest bliższa angielskiemu znaczeniu słowa "or" niż operacji logicznej OR. Jeśli jeden operand operacji logicznej exclusive-OR równa się jeden, wynik jest zawsze odwrotnością drugiego operandu; to znaczy ,jeśli jeden operand równa się jeden, wynik równa się zero jeśli drugi operand równa się jeden, a wynik równa się jeden jeśli drugi operand równa się zero. Jeśli pierwszy operand zawiera zero, wtedy wynik jest dokładnie wartością drugiego operandu. Ta cecha pozwala na selektywne odwracanie bitów w łańcuchu bitów.
Operacja logiczna NOT jest operacja jedno argumentową (w znaczeniu, że przyjmuje tylko jeden argument):

NOT 0=1
NOT 1=0

Tablica prawdy operacji NOT przyjmuje następującą formę:

NOT	0	1
	1	0

Tablica 5: Tablica Prawdy NOT

1.6 OPERACJE LOGICZNE NA LICZBACH BINARNYCH I ŁAŃCUCHACH BITÓW

Jak zostało opisane w poprzedniej sekcji ,funkcje logiczne pracują tylko z pojedynczymi bitami operandów. Ponieważ 80x86 używa grup 8,16 lub 32 bitów, musimy rozszerzyć definicje tych funkcji dotyczące więcej niż dwóch bitów. Funkcje logiczne w 80x86 operują na zasadzie "bit przez bit" .Jeśli podano dwie wartości, funkcje te operują na bicie zero ustalając w wyniku bit zero. Operując na bicie jeden wartości wejściowych ,ustawiają w wyniku bit jeden itd. itp. Na przykład, jeśli chcesz obliczyć logiczne AND z następujących dwóch ośmiobitowych liczb, musisz wykonać operację logicznego AND na każdej kolumnie niezależnie od pozostałych:

```
1011 0101
1110 1110
----------
1010 0100
```

Ta forma "bit przez bit" może być śmiało zastosowana również dla innych logicznych operacji. Ponieważ zdefiniowaliśmy operacje logiczne pod względem wartości binarnych, przyjdzie ci dużo łatwiej wykonać operacje logiczne na wartościach binarnych niż na wartościach opartych o inne systemy liczbowe. Zatem jeśli chcesz wykonywać operacje logiczne na dwóch liczbach heksadecymalnych, przekonwertuj je najpierw na liczby binarne .Ma to zastosowanie do większości podstawowych operacji logicznych na liczbach binarnych (np. AND, OR ,XOR, itd) Umiejętność ustawiania bitów na zero lub jeden, przy użyciu operacji logicznych AND/OR i umiejętność odwracania bitów przy użyciu operacji logicznej XOR jest bardzo ważna kiedy pracujemy z łańcuchami bitów (np. liczbami binarnymi) .Te operacje pozwalają selektywnie manipulować pewnymi bitami wewnątrz jakiejś wartości podczas gdy pozostawiamy inne bity w spokoju. Na przykład, jeśli masz ośmiobitowa wartość "X" i chcesz mieć pewność, że bity od czwartego do siódmego będą zawierać zera, możesz logicznie "dodać"(AND) wartość "X" z binarna wartością 00001111.Ta operacja logiczna AND, "bit przez bit" ustawi bardziej znaczące cztery bity na zero i pozostawi mniej znaczące cztery bity z "X" bez zmian. Podobnie możesz ustawić mniej znaczące bity z "X" na jeden i odwrócić bit numer dwa z "X" ,odpowiednio, przez logiczne ORowanie "X" przez 0000 0001 i logiczne exclusive-ORowanie przez 0000 0100.Użycie operacji logicznych AND,OR i XOR do manipulowania łańcuchami bitów w ten sposób ,jest znany jako maskowanie łańcucha bitów. Używamy terminu maskowanie, ponieważ możemy używać pewnych wartości (jeden dla AND, zero dla OR/XOR) do "zamaskowania" pewnych bitów podczas operacji zmiany bitów zero/jeden lub ich odwracania.

1.7 LICZBY ZE ZNAKIEM I BEZ ZNAKU

Jak dotąd traktowaliśmy liczby binarne jako wartości bez znaku. Liczba binarna ...0000 reprezentowała zero,...0001 przedstawiała jeden ,0010 przedstawiała dwa, tak aż do nieskończoności. A co z liczbami ujemnymi? Wartości ze znakiem zostały wspomniane w pierwszej sekcji, gdzie wspomnieliśmy system „uzupełniania do dwóch", ale nie omówiliśmy jak przedstawić liczby ujemne używając binarnego systemu liczbowego. To jest właśnie to o czym będzie ta sekcja!

Przy przedstawianiu liczb bez znaku przy użyciu binarnego systemu liczbowego mamy ograniczone miejsce na nasze liczby :musza mieć skończoną i niezmienną liczbę bitów. Tak długo jak 80x86 pracuje ,nie jest to zbyt duże ograniczenie, w końcu 80x86 może adresować skończoną liczbę bitów. Z naszego punktu widzenia, mamy poważne ograniczenie liczby bitów do ośmiu,16,32 lub jakiejś innej, małej liczby bitów.

Z niezmienną liczbą bitów możemy przedstawić tylko pewna liczbę obiektów .Na przykład, przy pomocy ośmiu bitów możemy przedstawić tylko 256 rożnych obiektów. Wartości ujemne są pełnoprawnymi obiektami, tak jak liczby dodatnie. Dlatego tez użyjemy kilku, z 256 rożnych wartości do przedstawienia liczb ujemnych. Innymi słowy, musimy użyć kilku liczb dodatnich do przedstawienia liczb ujemnych. Aby zrobić to sprawiedliwie przydzielimy połowę z możliwych kombinacji dla wartości ujemnych i połowę dla wartości dodatnich. Możemy więc przedstawić wartości ujemne jako -128..-1 i wartości dodatnie jako 0..127 za pomocą pojedynczego bajtu.Za pomocą 16 bitowego słowa możemy przedstawiać liczby w zakresie -32,768..+32.767.Przy pomocy 32 bitowego

podwójnego słowa przedstawimy wartości z zakresu -2,147,483,648..+2,147,483,647.Generalnie,za pomocą n bitów możemy przedstawić wartość ze znakiem z zakresu -2(n-1)..+2(n-1)-1.
Okay, więc możemy przedstawiać wartości ujemne. Ale ,jak to robimy? Cóż, jest wiele sposobów, ale mikroprocesor 80x86 używa notacji „uzupełnienie do dwóch" .W systemie „uzupełnienia do dwóch" najbardziej znaczący bit jest bitem znaku. Jeśli jego wartość wynosi zero, wówczas liczba jest dodatnia, jeśli jego wartość wynosi jeden, wówczas liczba jest ujemna. Przykłady:
Dla liczby 16-bitowej:
8000h jest liczbą ujemną, ponieważ bit znaku równa się jeden
100h jest liczbą dodatnią ponieważ bit znaku równa się zero
7FFFh jest dodatnia
0FFFFh jest ujemna.
0FFFh jest dodatnia
Jeśli najbardziej znaczący bit wynosi zero, wtedy liczba jest dodatnia i jest przechowywana jako standardowa wartość binarna. Jeśli najbardziej znaczący bit równa się jeden, wtedy liczba jest ujemna i jest przechowywana w formie „uzupełnienia do dwóch". Przy konwertowaniu liczby dodatniej na ujemną, używasz następującego algorytmu:
1) odwracasz wszystkie bity w liczbie np. używając logicznej funkcji NOT
2) dodajesz jeden do odwróconego wyniku

Na przykład, obliczymy ośmiobitowy odpowiednik liczby -5:

0000 0101 Pięć (binarnie)
1111 1010 Odwracamy wszystkie bity
1111 1011 Dodajemy jeden do otrzymanego wyniku

Jeśli weźmiemy minus pięć i wykonamy na niej operacje „uzupełnienia do dwóch", otrzymamy naszą oryginalną wartość 000101,tak jak się spodziewaliśmy:

1111 1011 -5 po „uzupełnieni do dwóch"
0000 0100 Odwracamy wszystkie bity
0000 0101 Dodajemy jeden do otrzymanego wyniku (+5)

Następujący przykład pokazuje kilka dodatnich i ujemnych 16-bitowych wartości ze znakiem:

7FFFh +32767, największą 16-bitowa liczba dodatnia
8000h -32768, najmniejsza 16-bitowa liczba ujemna
4000h +16384

Konwersje powyższych liczb do ich ujemnych odpowiedników (tj. ich negacji) robimy następująco:

7FFFh: 0111 1111 1111 1111 +32,767t
1000 0000 0000 0000 Odwrócenie wszystkich bitów (8000h)
1000 0000 0000 0001 Dodanie jedynki (8001h lub -32767t)
8000h: 1000 0000 0000 0000 -32,768t
0111 1111 1111 1111 Odwracamy wszystkie bity (7FFFh)
1000 0000 0000 0000 Dodajemy jedynkę (8000h lub -32768t)
4000h: 0100 0000 0000 0000 16,384
1011 1111 1111 1111 Odwracamy wszystkie bity (BFFFh)
1100 0000 0000 0000 Dodajemy jedynkę (0C000h lub -16,384t)

Odwrócone 8000h zmieniło się w 7FFFh.Po dodaniu jedynki uzyskujemy 8000h!Czekaj!Co się tutaj dzieje? -(-32,768) równa się -32,768? Oczywiście nie! .Ale wartość + 32,768 nie może być przedstawiana jako 16 bitowa liczba ze znakiem, wiec nie możemy zanegować najmniejszej ujemnej wartości. Jeśli spróbujesz takiej operacji, mikroprocesor 80x86 zgłosi błąd. Dlaczego mamy kłopotać się takim nieprzyjemnym systemem liczbowym? Dlaczego nie używać najbardziej znaczącego bitu jako znaku flagi, przechowując dodatni odpowiednik liczby w pozostałych bitach? Odpowiedź leży w hardware. Jak się okazuje negowanie ujemnych wartości jest nużącą pracą. Z systemem „uzupełnienia do dwóch" ,większość innych operacji jest tak łatwa jak system dwójkowy. Na przykład, przypuśćmy, że chcielibyśmy wykonać dodawanie 5+(-5).Wynik równa się zero. Rozważmy co się wydarzy, kiedy dodamy te dwie wartości w systemie uzupełnienia do dwóch:

```
000000101
111111011
----------
1 000000000
```

Skończyliśmy z przeniesieniem do 9 bitu i wyzerowanymi pozostałymi bitami .Okazuje się ,że jeśli zignorujemy przeniesienie najbardziej znaczącego bitu, dodanie dwóch wartości ze znakiem zawsze przyniesie prawidłowe rozwiązanie kiedy użyjemy systemu liczbowego uzupełnienia do dwóch. To oznacza ,że możemy używać tego samego hardware dla dodawania i odejmowania liczb ze znakiem i bez znaku. To nie mogłoby wystąpić w przypadku innych systemów liczbowych. Oprócz odpowiedzi na pytania na końcu tego rozdziału, nie musisz wykonywać „uzupełnienia do dwóch" ręcznie. Mikroprocesor 80x86 dostarcza instrukcje NEG (neguj),która wykona te operacje za ciebie. Co więcej ,wszystkie heksadecymalne kalkulatory wykonają ta operacje poprzez naciśniecie klawisza (+/- lub CHS).Niemniej jednak wykonanie „uzupełnienia do dwóch" jest łatwe a ty już wiesz jak to zrobić. Zapamiętaj raz jeszcze, że interpretacja danych reprezentowanych przez zbiór bitów binarnych zależy wyłącznie od kontekstu. Ośmiobitowa wartość binarna 11000000b może przedstawiać znak IBM/ASCII, może przedstawiać dziesiętną wartość bez znaku 192,lub dziesiętną wartość ze znakiem -64,itd.Jako programista, jesteś odpowiedzialny za właściwe używanie tych danych.

1.8 „ROZSZERZANIE ZNAKIEM" I „ROZSZERZENIE ZERAMI"

Ponieważ format „uzupełnienia do dwóch" liczb całkowitych ma stała formę ,pojawia się mały problem. Co się stanie jeśli musisz skonwertować ośmiobitową wartość „uzupełnieniem do dwóch" do 16 bitów? Ten problem i jego odwrotność (konwertowanie 16 bitowej wartości do ośmiu bitów) może być realizowane przez „rozszerzenie znakiem" i operacje „ściągania".80x86 pracuje ze stała długością wartości ,nawet kiedy pracuje na binarnej liczbie bez znaku. "Rozszerzenie o zero" pozwala ci skonwertować mała wartość bez znaku o dużej wartości bez znaku. Rozważmy wartość "-64".Osiem bitów, po „uzupełnieniu do dwóch „ dla tej wartości to "0C0h".16 bitowy odpowiednik tej liczby to 0FFC0h.Teraz,rozwazmy wartość "+64". Ośmio- i szesnastobitowa wersja tej wartości to odpowiednio 40h i 0040h. Różnica między ośmio i szesnastobitową liczbą może być opisana przez zasadę:" Jeśli liczba jest ujemna, najbardziej znaczący bajt z liczby 16 bitowej zawiera 0FFh;jeśli liczba jest dodatnia, najbardziej znaczący bajt z 16 bitowej liczby wynosi zero". „Rozszerzenie znakiem" wartości jakiejś liczby bitów do większej liczby bitów jest łatwe, skopiuj znak bitu do wszystkich dodatkowych bitów w nowym formacie. Na przykład, rozszerzając znak liczby ośmiobitowej do 16 bitowej, po prostu skopiuj siódmy bit z ośmiobitowej liczby do bitów 8..15 z liczby 16 bitowej. „Rozszerzając znak" 16 bitowej liczby do podwójnego słowa, po porostu skopiuj bit 15 do bitów 16..31 z podwójnego słowa. „Rozszerzenie znakiem" jest wymagane kiedy manipulujemy wartościami ze znakiem o rożnych długościach. Często musisz powiększyć bajt do wielkości słowa. Musisz „rozszerzyć znakiem" bajt do wielkości słowa nim ta operacja będzie miała miejsce. Inne operacje (w szczególności ,mnożenie i dzielenie) mogą wymagać poszerzenia do 32 bitów. Nie wolno ci rozszerzać wartości bez znaku. Przykłady „rozszerzenia znakiem":

Osiem bitów	Szesnaście bitów	Trzydzieści dwa bity
80h	FF80h	FFFFFF80h
28h	0028h	00000028h
9Ah	FF9Ah	FFFFFF9Ah
7Fh	007Fh	0000007Fh
----	1020h	00001020h
----	8088h	FFFF8088h

Do rozszerzenia bajtu bez znaku musisz zastosować „rozszerzenia zerem". Rozszerzanie zerem" jest bardzo łatwe – postaw zero przed najbardziej znaczący bajt(y) . Na przykład, „rozszerzanie zerem" wartości 82h do 16-bitowej,po prostu dodajesz zera do bardziej znaczącego bajtu co daje 0082h.

Osiem bitów	Szesnaście bitów	Trzydzieści dwa bity
80h	0080h	00000080h
28h	0028h	00000028h
9Ah	009Ah	0000009Ah
7Fh	007Fh	0000007Fh
----	1020h	00001020h
----	8088h	00008088h

„Ściąganie znaku", przekształcające wartość jakiejś liczby bitów do identycznej wartości z mniejszą liczbą bitów, jest troszkę bardziej dokuczliwe. „Rozszerzone znaki" nigdy nie zawodzą. Mając m-bitową wartość ze znakiem możesz zawsze przekształcić ją do n-bitowej liczby (gdzie n >m) używając „rozszerzenia znakiem" .Niestety, mając n-bitową liczbę nie zawsze możesz przekształcić ją do m-bitowej liczby jeśli m< n. Na przykład, rozważmy wartość -448. Jako 16-bitowa heksadecymalna liczba która ją reprezentuje mamy 0FE40h. Niestety, rozmiar tej liczby jest zbyt duży aby dopasować ja do wartości ośmiobitowej, wiec nie możesz ściągnąć znaku do ośmiu bitów. To jest przykład przepełnienia stanu ,który zdarza się przy konwersji. Przy stosowaniu „ściągania znaku" jednej wartości do

innej musisz przypatrzyć się najbardziej znaczącym bajt(om),które chcesz „wyrzucić”. Najbardziej znaczące bajty które pragniesz usunąć ,muszą zawierać albo zero albo 0FFh.Jesli napotkasz jakąś inna wartość, nie możesz jej skrócić bez przepełnienia. Ostatecznie, najbardziej znaczący bit z twojej wartości końcowej musi odpowiadać każdemu bitowi który usunąłeś z liczby. Przykłady (16 bitów do 8 bitów):

FF80h znak nie może być skrócony do 80h
0040h znak nie może być skrócony do 40h
FE40h znak nie może być skrócony do 8 bitów
0100h znak nie może być skrócony do 8 bitów

1.9 PRZESUNIĘCIA I OBROTY

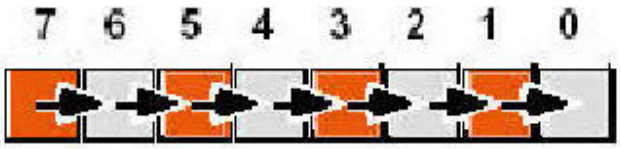
Innym zbiorem logicznych operacji które mają zastosowanie do łańcucha bitów są operacje przesunięcia i obrotów. Te dwie kategorie mogą być dalej rozbite na "przesunięcie w lewo", "obrót w lewo", "przesunięcie w prawo" i "obrót w prawo". Te operacje okazują się być niezwykle użyteczne dla programisty asemblerowego. Operacja przesunięcia w lewo, przesuwa każdy bit w łańcuchu bitów jedna pozycje w lewo (zobacz rysunek 1.8)

Rysunek 1.8: Operacja przesunięcia w lewo

Bit zero przesuwa się na pozycje bitu jeden, poprzednia wartość bitu jeden przesuwa na pozycje bitu dwa itd. Są oczywiście dwa pytania, które pojawiają się w sposób naturalny: "Gdzie zmierza bit zero?" i „Gdzie znika bit siódmy?". Do najmniej znaczącego bitu wpisywana jest wartość zero, a wartość siódmego bitu jest przenoszona podczas tej operacji do znacznika flagi. Zapamiętaj, ze przesunięcie wartości w lewo jest tym samym co pomnożenie jej przez jej podstawę potęgi. na przykład, przesunięcie wartości dziesiętnej jedną pozycję w lewo (dodanie zera z prawej strony) faktycznie mnoży ją przez dziesięć (podstawa potęgi):

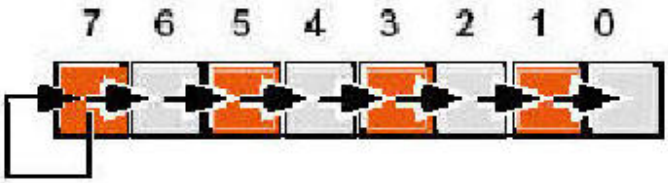
1234 SHL 1 = 12340 (SHL 1 = przesunięcie w lewo o jedna pozycje)

Ponieważ podstawą potęgi liczby binarnej jest dwa, przesunięcie w lewo mnoży ją przez dwa. Jeśli przesuniemy wartość binarną w lewo dwa razy, mnożymy ją przez dwa razy (tj. mnożymy ją przez cztery).Jeśli przesuwamy wartość binarną w lewo trzy razy, mnożymy ją przez osiem (2*2*2).Generalnie, jeśli przesuwamy wartość w lewo n razy, mnożymy tą wartość przez 2^n.Operacja przesunięcia w prawo pracuje na tej samej zasadzie, z wyjątkiem tego, że przesuwamy dane w przeciwnym kierunku. Bit siedem przesuwamy do bitu sześć, bit sześć przesuwamy do bitu piątego itd. Podczas przesunięcia w prawo, do bitu siódmego wpisujemy 0,a bit zero zostaje przeniesiony .(zobacz rysunek 1.9)



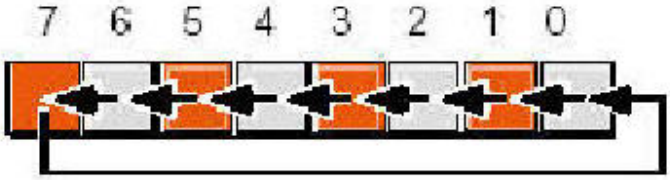
Rysunek 1.9: Operacja przesunięcia w prawo

Jest jeden problem z przesunięciem w prawo w związku z dzieleniem: jak opisano powyżej, przesuniecie w prawo jest tylko odpowiednikiem bezznakowego dzielenia przez dwa. Na przykład, jeśli chcesz przesunąć wartość bez znaku 254 (0EFh) jedno miejsce w prawo, otrzymasz 127 (07Fh),dokładnie to co chciałeś osiągnąć. Jednak, jeśli przesuniesz binarny odpowiednik -2 (0FEh) w prawo o jedną pozycję, otrzymasz 127 (07Fh) która nie jest prawidłowa. Problem występuje ponieważ wstawiamy zero do bitu siódmego. Jeśli bit siódmy poprzednio zawierał jeden, zmienimy ja z liczby ujemnej na dodatnią. Aby stosować przesunięcie w prawo jako operator dzielenia, musimy zdefiniować trzecią operację przesunięcia: arytmetyczne przesunięcie w prawo. Pracuje ona dokładnie tak jak zwykła operacja przesunięcia w prawo, z jednym wyjątkiem: zamiast zmieniać bit siódmy na zero, pozostawia bit siódmy w spokoju, tzn. podczas operacji przesunięcia nie modyfikuje wartości siódmego bitu, jak pokazuje rysunek 1.10

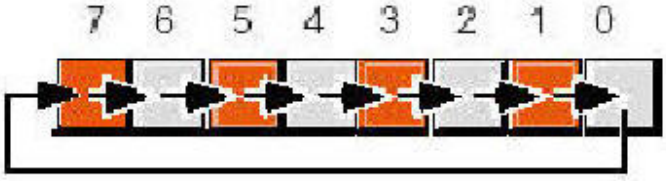


Rysunek 1.10 Operacja arytmetycznego przesunięcia w prawo

Generalnie przynosi to oczekiwane wyniki. Na przykład, jeśli wykonujesz arytmetyczne przesunięcie w prawo na -2 (0FEh) dostajesz -1(0FFh).Zapamiętaj jednak jedną rzecz .Ta operacja zawsze zaokrągla liczby do najbliższej wartości całkowitej która jest mniejsza lub równa rzeczywistemu wynikowi. Opierając się na doświadczeniach z programowania w językach wysokiego poziomu, i standardowych zasadach zaokrąglania wartości całkowitych, większość ludzi przyjmuje założenie, że dzieląc zawsze zaokrągla w kierunku zera. Ale nie jest tak prosto. Na przykład, jeśli zastosujesz operacje arytmetycznego przesunięcia w prawo dla –1 (0FFh),wynik jest -1 ,nie zero.-1 jest mniejsze niż zero wiec operacja arytmetycznego przesunięcia w prawo zaokrągli do -1.To nie jest błąd tej operacji. Jest to sposób dzielenia wartości całkowitych, typowo zdefiniowanych. Instrukcje dzielenia całkowitego 80x86 również dadzą taki wynik. Inną parą użytecznych operacji jest obrót w lewo i obrót w prawo. Te operacje zachowują się tak jak operacje przesunięcia w lewo i w prawo ze znaczącą różnicą: bit który jest przesuwany z jednego końca, zostaje "wsuwany" z końca drugiego .



Rysunek 1.11:Operacja obrotu w lewo



Rysunek 1.12:Operacja obrotu w prawo

1.10 POLA BITÓW I DANE UPAKOWANE

Chociaż 80x86 operuje bardziej wydajnie na bajtach, słowach i podwójnych słowach, czasami musimy pracować na typach danych które używają innej liczby bitów niż 8,16 czy 32.Na przykład, mamy daną w postaci "4/2/88".Te trzy wartości liczbowe przedstawiają taką daną: miesiąc, dzień i rok. Miesiąc, oczywiście, przyjmuje wartości od 1 do 12.To wymagałoby co najmniej czterech bitów (maksimum szesnaście rożnych wartości) dla przedstawienia miesiąca. Zakres dni to 1..31.Wymaga to pięciu bitów (maksimum 32 rożne wartości) dla przedstawienia zapisu dnia. Wartość roku, biorąc pod uwagę, że pracujemy z wartościami z zakresu 0..99,wymaga siedmiu bitów (które mogą być użyte do przedstawienia do 128 rożnych wartości).Cztery plus pięć plus siedem to szesnaście bitów lub dwa bajty. Innymi słowy, możemy spakować nasza datę do dwóch bajtów zamiast do trzech, które byłyby wymagane gdybyśmy używali oddzielnych bajtów dla każdej wartości miesiąca, dnia i roku. Ta oszczędność jednego bajtu w pamięci dla każdej przechowywanej danej, może być pokaźną oszczędnością jeśli musimy przechowywać dużo danych. Te bity mogą być ułożone tak jak pokazano poniżej:

Rysunek 1.13 Format upakowania danych

MMMM przedstawia cztery bity stanowiące wartość miesiąca, DDDDD przedstawia pięć bitów stanowiących wartość dnia a YYYYYYY to siedem bitów składających się na rok. Każdy zbiór bitów przedstawiający daną wartość nazywany jest „polem bitow”.2 kwietnia 1988 będzie przedstawiony jako 4158h:

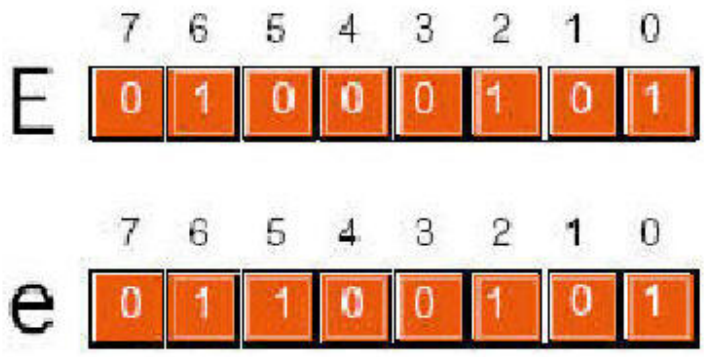
0100 00010 1011000 = 0100 0001 0101 1000b lub 4158h
4 2 88

Chociaż wartości spakowane są wydajne (to znaczy, bardzo wydajne pod względem zużycia pamięci),są niewydajne przy obliczeniach (powolne!) Powód? Mamy dodatkowa instrukcje do wypakowania spakowanych danych ,do kilku pól bitów. Ta dodatkowa instrukcja wymaga dodatkowego czasu na wykonanie (i dodatkowych bajtów dla wykonywanej instrukcji);w związku z tym musisz ostrożnie rozważać czy spakowane pola danych zaoszczędza ci cokolwiek. Przykłady możliwych do zastosowania typów danych spakowanych można długo wyliczać .Możesz spakować osiem wartości boolowskich do pojedynczego bajtu, możesz spakować dwie cyfry BCD do bajtu itp.

1.11 ZBIÓR ZNAKÓW ASCII

Zbiór znaków ASCI I(wyłączając znaki rozszerzone, zdefiniowane przez IBM) są podzielone na cztery grupy po 32 znaki. Pierwsze 32 znaki o kodach ASCII od 0 do 1Fh (31) są specjalnym zbiorem znaków niedrukowalnych zwanych znakami sterującymi Nazywamy je znakami sterującymi ponieważ wykonują one operacje sterujące na urządzeniach typu drukarka/ekran zamiast wyświetlać symbole. Przykładem może być „powrót karetki" ,który ustawia kursor po lewej stronie bieżącej linii znaków, „przesunięcie o jedną linię" (który przesuwa kursor w dół o jedną linię czy „cofnięcie" (który przesuwa kursor jedna pozycje w lewo). Niestety, rożne znaki sterujące wykonują rożne operacje na rożnych urządzeniach wyjściowych. Jest niewielka standaryzacja między urządzeniami wyjściowymi. Chcąc dowiedzieć się jak znaki sterujące wpływają na poszczególne urządzenia musisz zapoznać się z instrukcją.

Druga grupa 32 kodów znaków ASCII stanowi kilka znaków interpunkcyjnych, znaki specjalne i cyfry Najbardziej godne uwagi w tej grupie to znak spacji (Kod ASCII 20h) i cyfry (kody ASCII od 30h do 39h).Zauważ, że cyfry różnią się od swoich wartości liczbowych tylko w najbardziej znaczącym nibble'u. Przez odjęcie 30h z kodu ASCII, od dowolnej cyfry otrzymasz liczbowy odpowiednik tej cyfry. Trzecia grupa 32 znaków ASCII jest zarezerwowana dla dużych znaków alfabetu. Kody ASCII dla znaków "A".."Z" leżą w zakresie 41h..5Ah(65..90).Ponieważ jest tylko 26 rożnych znaków alfabetu, pozostałe sześć kodów otrzymało kilka specjalnych symboli. Czwarta, i końcowa, grupa 32 kodów znaków ASCII jest zarezerwowane dla małych liter alfabetu, pięć dodatkowych symboli specjalnych, i inne znaki sterujące (np. delete). Zauważ, ze znaki małych liter używają kodów ASCII 61h..7Ah.Jeśli przetworzysz kody dużych i małych liter na liczby binarne, zauważysz, że symbole dużych liter różnią się od swoich małych odpowiedników dokładnie w jednym bitem Na przykład, rozważmy kody znaków "E" i "e" (rysunek 1.14)



Rysunek 1.14:Kody ASCII dla "E' i "e"

Te dwa kody różnią się między sobą tylko w jednym miejscu, w bicie piątym. Znak dużej litery zawsze zawiera zero w bicie piątym; znak malej litery zawsze zawiera jeden w tym bicie. Możesz wykorzystać ten fakt do szybkiego przekształcania między duża a małą literą. Jeśli masz dużą literę, możesz ustawić małą literę przez ustawienie bitu piątego na jeden. Jeśli masz małą literę i życzysz sobie zamienić na dużą, możesz zrobić to przez ustawienie bitu piątego na zero .Możesz przełączać znaki alfabetu pomiędzy dużymi i małymi literami poprzez prostą inwersję bitu piątego. Istotnie, bity piąty i szósty określają do której z czterech grup się odnosimy:

Bit 5	Bit 6	Grupa
O	0	Znaki sterujące
0	1	Cyfry i znaki interpunkcyjne
1	0	Duże litery i znaki specjalne

1	1	Małe litery i znaki specjalne

Więc możemy, na przykład, skonwertować każda dużą lub małą literę (lub odpowiednie znaki specjalne) do ich odpowiedników- znaków sterujących przez ustawienie bitów piątego i szóstego na zero. Rozważmy, na chwile, kody ASCI kilku cyfr:

Znak	Dziesiętnie	Heksadecymalnie
„0”	48	30h
„1”	40	31h
„2”	50	32h
„3”	51	33h
„4”	52	34h
„5”	53	35h
„6”	54	36h
„7”	55	37h
„8”	56	38h
„9”	57	39h

Dziesiętne przedstawienie tych kodów ASCII nie jest zbyt pouczające. Jednakże ,przedstawienie heksadecymalne, tych kodów ASCII odsłania coś bardzo ważnego - najmniej znaczący nibble z kodu ASCII jest binarnym odpowiednikiem przedstawianej liczby. Przez pozbawienie (tj. ustawienie na zero) najbardziej znaczącego nibble'a z numeru znaku, możesz przekształcić ten kod znaku na odpowiednią binarną wartość. Odwrotnie, możesz skonwertować wartość binarną w zakresie od 0 do 9 do ich znaków ASCII przez ustawienie bardziej znaczącego nibble'a na trzy. Zauważ, że możesz użyć tylko logicznej operacji AND do ustawienia najbardziej znaczących bitów na zero; podobnie możesz użyć logicznej operacji OR do ustawienia najbardziej znaczących bitów na 0011 (trzy).Nie możesz jednak skonwertować łańcucha znaków liczbowych do ich odpowiedników binarnych poprzez proste pozbawienie najbardziej znaczącego nibble'a z każdej cyfry w łańcuchu. Konwertowanie 123 (31h 32h 33h) w ten sposób przyniesie trzy bajty 010203h,a nie jest to prawidłowa wartość, którą jest 7Bh.Konwertowanie łańcucha cyfr całkowitych wymaga większej złożoności ; powyższa konwersja działa tylko dla cyfr pojedynczych.

Siódmy bit w standardowym ASCII wynosi zawsze zero. To znaczy, ze zbiór znaków ASCII zużywa tylko połowę możliwych kodów znaków w ośmiobitowym bajcie. IBM używa pozostałych 128 kodów znaków dla rożnych znaków specjalnych zawierających znaki międzynarodowe, symbole matematyczne i inne .Zauważ, że te znaki specjalne są niestandardowym rozszerzeniem zbioru znaków ASCII. Oczywiście, nazwa IBM ma dużą siłę przebicia, więc prawie wszystkie nowoczesne komputery osobiste oparte o 80x86 opierają się o rozszerzony zbiór znaków IBM/ASCII. Większość drukarek opiera się również o IBMowski zbiór znaków. Jeśli będziesz musiał wymieniać dane z innymi komputerami, które nie są kompatybilne z PC, masz tylko dwie alternatywy: pozostać przy standardowym ASCII lub upewnić się, że komputer docelowy opiera się o rozszerzony zbiór znaków IBM-PC. Niektóre maszyny, takie jak Apple Macintosh nie dostarczają gotowego wsparcia dla rozszerzonego zbioru znaków, jednakże możesz uzyskać fonty PC, które pozwalają na wyświetlenie rozszerzonego zbioru znaków. Inne komputer(np. Amiga i Atari ST :-)) maja podobne możliwości. .Jednak,128 znaków standardowym zbiorze znaków ASCII są jedynymi ,które mogą liczyć na przeniesienie z systemu do systemu. Pomimo faktu, ze jest to "standard", proste kodowanie twoich danych nie gwarantuje kompatybilności z innymi systemami. To prawda, ze "A" na jednej maszynie jest podobne do "A" na innej, jest niewielkie ujednolicenie w maszynach pod względem używania znaków sterujących. Istotnie, z 32 kodów sterujących plus delete, tylko cztery kody sterujące są powszechnie stosowane – backspace (BS), tab, carriage return (CF) i line feed (LF).Rożne maszyny często używają tych kodów sterujących na rożny sposób. End of line (koniec linii) jest szczególnie pouczającym przykładem. MS-DOS,CP/M i inne systemy oznaczają koniec linii dwuznakowa sekwencja CR/LF. Apple Macintosh, Apple II i wiele innych systemów oznacza koniec linii pojedynczym znakiem CR. System UNIX oznacza koniec linii pojedynczym znakiem LF. Rzecz jasna, próby wymiany prostego pliku tekstowego między takimi systemami mogą być przeżyciem frustrującym Nawet jeśli są używane standardowe znaki ASCII we wszystkich twoich plikach na tych systemach, będziesz musiał jeszcze skonwertować te dane, kiedy wymienisz pliki między nimi. Na szczęście, takie konwersje są dosyć proste. Pomimo kilku ważnych niedostatków, dane ASCII są standardem dla wymiany danych pomiędzy systemami komputerowymi i programami. Większość programów akceptuje dane ASCII; podobnie większość programów może tworzyć dane

ASCII. Ponieważ, będziesz miał do czynienia ze znakami ASCII w języku asemblera, będziesz musiał dokładnie przestudiować rozkład zbioru znaków i nauczyć na pamięć kilku kodów klawiszy ASCII(np. "0","A","a",itp).

1.12 PODSUMOWANIE

Większość nowoczesnych systemów komputerowych używa binarnego systemu liczbowego do przedstawiania wartości. Ponieważ wartości binarne są w pewnym stopniu nieporęczne, często będziemy używać heksadecymalnej reprezentacji dla tych wartości .Jest tak, ponieważ jest bardzo łatwo konwertować pomiędzy heksadecymalną a binarną liczbą, w odróżnieniu od konwersji pomiędzy bardzo dobrze znanym systemem dziesiętnym a binarnym. Pojedyncza cyfra heksadecymalna używa czterech cyfr binarnych (bitów) a grupę czterech bitów nazywamy nibble.

Zobacz:

"Binarny System Liczbowy"
"Format Binarny"
"Heksadecymalny System Liczbowy"

80x86 pracuje najlepiej z grupą bitów o długości 8,16 lub 32 bitów. Obiekty o tych rozmiarach nazywamy, odpowiednio, bajtem, słowem i podwójnym slowem.Za pomocą bajtu, możemy przedstawić jedną z 256 unikalnych wartości. Za pomocą słowa możemy przedstawić jedna z 65,536 rożnych wartosci.Za pomocą podwójnego słowa możemy przedstawić ponad miliard rożnych wartości. Często przedstawiamy wartości całkowite (ze znakiem lub bez znaku) za pomocą bajtu, słowa lub podwójnego słowa.; jednakże często będziemy przedstawiać również inne wartości.

Zobacz:

"Organizacja Danych"
"Bajty"
"Słowa"
"Podwójne Słowa"

Żeby mówić o określonych bitach wewnątrz nibble, bajtu, słowa , podwójnego słowa lub innej struktury ,numerujemy bity zaczynając od zera (od najmniej znaczącego bitu) w gorę do n-1 (gdzie n to numer bitu w obiekcie. Również numerujemy nibble ,bajty i słowa w dużych strukturach w podobny sposób. Zobacz:

"Format Binarny"

Jest dużo operacji które możemy wykonywać na wartościach binarnych wliczając w to normalna arytmetykę (+,-,*, i /) i operacje logiczne (AND, OR, XOR, NOT, Przesuniecie w Lewo, Przesuniecie w Prawo, Obrót w Lewo, Obrót w Prawo).Logiczne AND,OR,XOR i NOT są zwykle określone dla operacji na pojedynczych bitach. .Przesunięcia i obroty są zawsze zdefiniowane dla stałych długości łańcuchów bitów.

Zobacz:

"Operacje Arytmetyczne Na Liczbach Binarnych I Heksadecymalnych"
"Operacje Logiczne Na Bitach"
"Operacje Logiczne NA Liczbach Binarnych I Łańcuchach Bitów"
"Przesunięcia I Obroty"

Są dwa typy wartości całkowitych, które możemy przedstawić za pomocą łańcuchów binarnych w 80x86: wartości całkowite bez znaku i wartości całkowite ze znakiem.80x86 przedstawia wartości całkowite bez znaku za pomocą standardowego formatu binarnego. Przedstawia wartości całkowite ze znakiem używając formatu „uzupełnienia do dwóch.” Ponieważ wartości całkowite bez znaku mogą mieć dowolną długość, można mówić o stałej długości binarnych wartości ze znakiem .Zobacz:

"Liczby Ze Znakiem I Bez Znaku"
"Rozszerzenie Znakiem i Rozszerzenie Zerem"

Często nie można w sposób możliwy do zastosowania w praktyce przechowywać danych w grupach ośmio- ,szesnasto- i trzydziestodwubitowych. Dla zaoszczędzenia miejsca, możemy chcieć upakować kilka rożnych kawałków danych do tego samego bajtu, słowa lub podwójnego słowa. To zmniejszy pamięć potrzebną do wykonywania kosztownych operacji specjalnych do pakowania i rozpakowywania danych. Zobacz:

"Pola Bitów I Dane Spakowane"

Znak jest prawdopodobnie najpopularniejszym typem danych z którym się spotykamy, poza wartościami całkowitymi. IBM PC i kompatybilne używają wariantu ze zbiorem znaków ASCII – rozszerzonym zbiorem znaków

IBM/ASCII. Pierwsze ze 128 znaków jest standardowymi znakami ASCII, dalsze128 to znaki specjalne stworzone przez IBM dla języków międzynarodowych, matematyki i innych .Ponieważ użycie zbioru znaków ASCII jest bardzo popularne w nowoczesnych programach, zaznajomienie z tym zbiorem znaków jest niezbędne. Zobacz:

"Zbiór Znaków ASCII"

1.14 PYTANIA

01) Przekształć następujące liczby dziesiętne na binarne:
a)128 b)4096 c)256 d)65536 e)254 f)9 g)1042 h)15 i)334 j)998 k)255 l)512 m)1023 n)2048 o)4095 p)8192 q)16,384 r)32,768 s)6,334 t)12,334 u)23,465 v)5,643 w)463 x)67 y)888
02) Przekształć następujące wartości binarne na dziesiętne:
a)1001 1001 b)1001 1101 c)1100 0011 d)0000 1001 e)1111 1111 f)0000 1111 g)0111 1111 h)1010 0101 i)0100 0101 j)0101 1010 k)1111 0000 l)1011 1101 m)1100 0010 n)0111 1110 o)1110 1111 p)0001 1000 q)1001 1111 r)0100 0010 s)1101 1100 t)1111 0001 u)0110 1001 v)0101 1011 w)1011 1001 x)1110 0110 y)1001 0111
03) Przekształć liczby binarne z punktu (2) na liczby heksadecymalne
04) Przekształć poniższe liczby heksadecymalne na binarne:
a)0ABCD b)1024 c)0DEAD d)0ADD e)0BEEF f)8 g)05AAF h)0FFFF i)0ACDB j)0CDBA k)0FEBA l)35 m)0BA n)0ABA o)0CDBA p)0DAB q)4321 r)334 s)45 t)0E65 u)0BEAD v)0ABE w)0DEAF x)0DAD y)9876
Wykonaj następujące obliczenia heksadecymalne (wyniki podaj w liczbach heksadecymalnych):
05) 1234 + 9876
06) 0FFF + 0F34
07) 100 - 1
08) 0FFE - 1
09) Jakie znaczenie ma nibble?
10) Ile cyfr heksadecymalnych jest w:
a)bajcie b)słowie c) podwójnym słowie
11) Ile bitów jest w:
a) nibble'u b) bajcie c) słowie d) podwójnym słowie
12) Który bit (numer bitu) jest najbardziej znaczącym bitem w:
a) nibble'u b)bajcie c) słowie d) podwójnym słowie
13) Jakiego znaku używamy jako przyrostka dla oznaczenia liczby heksadecymalnej, binarnej i dziesiętnej?
14) Zakładając 16 bitowy format "dopełnienia do dwóch", ustal która z wartości w pytaniu czwartym jest dodatnia a która ujemna.
15) „Rozszerz znak” wszystkich wartości w pytaniu drugim do szesnastu bitów. Podaj swoją odpowiedź w liczbach heksadecymalnych.
16) Dokonaj "bitowania" operacją AND na następujących parach wartości heksadecymalnych .Przedstaw swoją odpowiedź w liczbach heksadecymalnych.
(Podpowiedź: skonwertuj wartości heksadecymalne na binarne, wykonaj operacje, potem przekonwertuj z powrotem na heksadecymalne):
a)0FF00,0FF0 b)0F00F,1234 c)4321,1234 d)2341,3214 e)0FFF,0EDBC f)1111,5789 g)0FABA,4322 h)5523,0F572 i)2355,7466 j)4765,6543 k)0ABCD,0EEFDC l)0DDDD,1234 m)0CCCC,0ABCD n)0BBBB,1234 o)0AAAA,1234 p)0EEEE,1248 q)8888,1248 r)8086,124F s)8086,0CFA7 t)8765,3456 u)7089,0FEDC v)2435,0BCDE w)6355,0EFDC x)0CBA,6884 y)0AC7,365
17) Wykonaj operacje logiczną OR na powyższych parach liczb
18) Wykonaj logiczną operacje XOR na powyższych parach liczb
19) Wykonaj operacje logiczną NOT na wszystkich wartościach w pytaniu czwartym. Załóż, ze wszystkie wartości są 16 bitowe
20) Wykonaj operacje "dopełnienia do dwóch" na wszystkich wartościach w pytaniu czwartym. Załóż 16 bitowe wartości.
21) Rozszerz znak następujących heksadecymalnych wartości z ośmiu do szesnastu bitów. Odpowiedź przedstaw w liczbach heksadecymalnych.:
a)FF b)82 c)12 d)56 e)98 f)BF g)0F h)78 i)7F j)F7 k)0E l)AE m)45 n)93 o)C0 p)8F q)DA r)1D s)0D t)DE u)54 v)45 w)F0 x)AD y)DD
22) Skróć znak następujących wartości z szesnastu do ośmiu bitów. Jeśli nie możesz wykonać tej operacji, wyjaśnij dlaczego:

a)FF00 b)FF12 c)FFF0 d)12 e)80 f)FFFF g)FF88 h)FF7F i)7F j)2 k)8080 l)80FF m)FF80 n)FF o)8 p)F q)1 r)834 s)34 t)23 u)67 v)89 w)98 x)FF98 y)F98
23) Rozszerz znak 16 bitowych wartości z pytania 22 do 32 bitów
24) Zakładając, że wartości w pytaniu 22 są 16 bitowe, wykonaj na nich operacje przesunięcia w lewo.
25) Zakładając, że wartości w pytaniu 22 są 16 bitowe, wykonaj na nich operacje przesunięcia w prawo.
26) Zakładając, że wartości w pytaniu 22 są 16 bitowe, wykonaj na nich operacje obrotu w lewo.
27) Zakładając, że wartości w pytaniu 22 są 16 bitowe, wykonaj na nich operacje obrotu w lewo.
28) Skonwertuj następujące daty na spakowany format opisany w tym rozdziale (zobacz: "Pola Bitów i Spakowane Dane")Przedstaw swoje wyniki jako 16 bitowe liczby heksadecymalne:
a)1/192 b)2/4/56 c)6/19/60 d)6/16/86 e)1/1/99
29) Opisz jak za pomocą przesunięć i operacji logicznych wydobyć pole dnia ze spakowanej danej z rekordu w pytaniu 28.
30) Przypuśćmy, ze masz wartość z zakresu 0..9.Wyjasnij jak mógłbyś zamienić na znak ASCI używając podstawowych operacji logicznych.