

CSAPP

Lukasz Kopyto

February 29, 2024

2 Rozdział 2. Reprezentacja i Manipulacja Danymi.

[start = 2]

2.1 Przechowywanie informacji

Komputery używają 8-bitowych bloków nazywanych bajtami, jako najmniejsze adresowane jednostki pamięci. Program na poziomie maszynowym widzi pamięć jako bardzo dużą tablicę bajtów, zwana pamięcią wirtualną(ang. virtual memory). Każdy bajt pamięci ma swój unikatowy numer, nazywany adresem. Zbiór wszystkich możliwych adresów jest nazywany jako przestrzenią adresów wirtualnych(virtual address space).

2.1.1 Notacja szesnastkowa/hexadecymalna

Tabela 2.2. Trzeba ją mieć w głowie.

Dziesiętnie	Binarnie	Szesnastkowo
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Zwykle zapisujemy liczbę szesnastkową z przedrostkiem 0x.

Na przykład weźmy liczbę zapisaną w systemie szesnastkowym: 0x173A4C. Chcemy ją zamienić na system binarny. Bierzemy każdą cyfrę z liczby w zapisie szesnastkowym i zamieniamy ją na odpowiadający kwartet binarny.

Dla liczby 0x173A4C wygląda to tak: 1 → 0001

7 → 0111

3 → 0011

A → 1010

4 → 0100

C → 1100

Po ustawieniu kolejno kwartetów, otrzymujemy liczbę w zapisie binarnym:

000101110011101001001100₂

Dla odwrotnej konwersji działa to analogicznie. Dzielimy liczbę w zapisie binarnym na kwartety, od najmniej znaczącego bitu. Po czym zamieniamy kwartety na odpowiadające im cyfry w zapisie szesnastkowym.

Cwiczenie 2.1

Dokonaj ponizszych konwersji

- 0x25B9D2 to binary:

Kazda cyfre szesnastkowo zamieniamy na 'kwartet binarny':

$$0x25B9D2 \rightarrow 001001011011100111010010$$

- binary 1010111001001001 to hexadecimal:

Z liczby binarnej wydzielamy kwartety, pocawszy od LSB(least significant bit-najmniej znaczący bit). Jeśli liczba bitów nie jest podzielna przez 4 to dopelniamy zerami:

$$1010111001001001 \rightarrow 1010111001001001 \rightarrow 0xAE49$$

- 0xA8B3D to binary:

$$0xA8B3D \rightarrow 10101000101100111101$$

- binary 1100100010110110010110 to hexadecimal

$$1100100010110110010110 \rightarrow 001100100010110110010110 \rightarrow 0x322D95$$

Kiedy x jest potega dwójki tzn $x = 2^n$ mozna latwo to przepisać na wartosc szesnastkowa pamietajac, ze binarnie x to bedzie 1 i n zer. Wiec dla n zapisanego w formie $i + 4j$ gdzie $0 \leq i \leq 3$, mozemy zapisac x w formie szesnastkowej biorac za pierwsza cyfre szesnastkowa 1 gdy $i = 0$, 2 gdy $i = 1$, 4 gdy $i = 2$ oraz 8 gdy $i = 3$ dodajac pozniej j zer(szesnastkowych). Jako przyklad spojrzmy na $x = 2048 = 2^{11}$. Mamy tu $n = 3 + 2 * 4$, zatem x szesnastkowo to 0x800.

Cwiczenie 2.2

Uzupelnij ponizsza tabelke:

n	$2^n(decimal)$	$2^n(hexadecimal)$
5	32	$2^{1+1*4} = 0x20$
23	2^{23} duzo	$2^{23} = 2^{3+5*4} = 0x800000$
15	32768	$2^{15} = 2^{3+3*4} = 0x8000$
12	4096	$2^{12} = 2^{0+3*4} = 0x1000$
6	64	$2^6 = 2^{2+1*4} = 0x40$
8	256	$2^8 = 2^{0+2*4} = 0x100$

2.2 Rozmiary danych

Kazdy komputer posiada jakis rozmiar slowa maszynowego, ktory okresla nominalny rozmiar wskaznika na dane. Poniewaz adres wirtualny jest zakodowany poprzez to slowo, najwazniejszym parametrem systemu zdeterminowanym poprzez rozmiar slowa maszynowego jest maksymalny rozmiar przestrzeni adresow wirtualnych. Zatem dla maszyny z w -bitowym rozmiarem slowa maszynowego, adresy wirtualne sa z zakresu od 0 do $2^w - 1$, co daje programowi dostep do co najwyzej 2^w bajtow.

Wiekszosc 64-bitowych maszyn moze korzystac z programow skompilowanych dla 32-bitowych maszyn. Jest to forma wstecznej kompatybilnosci.

Jezyk C wspiera roznorakie formaty danych dla calkowitych jak i zmiennoprzecinkowych danych. Tabelka 2.3 pokazuje typowa liczbe bajtow zarezerwowanych dla roznnych typow danych w C. W pozniejszym rozdziale zostanie przerobiona to co jest zagwarantowane przez standard C a co jest typowe. Dokladna liczba bajtow dla poszczegolnych typow danych zalezy od tego jak program zostal skompilowany.

Tabela 2.3: Typowe rozmiary w bajtach standardowych typów danych w C

Liczba bajtów zarezerwowanych zależy od tego jak program został skompilowany. Poniższa tabela pokazuje wartości typowe dla 32 i 64 bitowych programów.

C Signed	C Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char*		4	8
float		4	4
double		8	8

Aby uniknąć polegania na "typowych" rozmiarach danych i różnych ustawieniach kompilatora, standard ISO C99 wprowadził nową klasę typów danych, gdzie rozmiar danych jest ustalony bez względu na ustawienia kompilatora oraz maszyny. Wśród nich są między innymi int32_t oraz int64_t, które mają dokładnie 4 oraz 8 bajtów kolejno.

Wielkość typów danych jest domyślnie ze znakiem, chyba że poprzedzona słowem zastrzeżonym unsigned lub specjalna forma deklaracji dla ustawionych typów (uint32_t etc.). Wyjątkiem jest jednak typ char. Pomimo iż większość kompilatorów i maszyn traktuje char jako wartość ze znakiem, standard C nie gwarantuje tego. Zamiast tego, aby zagwarantować to aby char był 1-bajtowym typem ze znakiem, programiści powinni używać deklaracji [signed] tak jak zostało to pokazane w tabelce 2.3. W wielu przypadkach program to olega czy char jest signed czy unsigned, ale trzeba mieć to na uwadze.

Język C pozwala na różne sposoby deklarować ten sam typ oraz wprowadzać lub nie opcjonalne słowa kluczowe. Jako przykład poniższe deklaracje mają to samo znaczenie:

- unsigned long
- unsigned long int
- long unsigned
- long unsigned int

Standardy języka C ustalają dolne granice wartości różnych typów danych (później to będzie dokładniej przerobione). Trzeba mieć na uwadze, że nie ustalają one górnych granic (z wyjątkiem typów z ustawionym rozmiarem).

Miedzy 1980 a 2010 32-bitowe maszyny z 32-bitowymi programami były dominujące. Wiele programów zostało napisanych z założeniem, że słowo maszynowe ma 32 bity. Zmiana na 64-bity, pojawiło się wiele problemów. Na przykład, wiele programistów zakładało, że obiekt typu int, może być używany do przechowywania wskaźnika. To działa dobrze dla większości 32-bitowych programów, ale będzie to prowadziło do problemów w 64-bitowych programach.

2.3 Adresowanie oraz uporządkowanie bajtów (Addressing and Byte Ordering)

Wielobajtowy obiekt jest przechowywany jako ciąg bajtów z adresem będącym najniższym adresem z użytych bajtów. Na przykład założymy że zmienna x jest typu int i ma adres 0x100; tzn. wartością wyrażenia adresu &x jest 0x100. Wtedy, zakładając że int ma 32-bitową reprezentację, 4 bajty na których jest zapisany x, będą przechowywane pod adresami 0x100, 0x101, 0x102, 0x103.

Żeby uporządkować bajty reprezentujące jakiś obiekt, są dwie znane konwencje. Rozważmy w-bitową liczbę całkowitą, która ma reprezentację w bitach $[x_{w-1}, x_{w-2}, \dots, x_1, x_0]$, gdzie x_{w-1} jest najbardziej znaczącym bitem (MSB most significant bit) oraz x_0 będącym najmniej znaczącym bitem (LSB least significant bit). Zakładając że w jest wielokrotnością 8, bity zmiennej x mogą być pogrupowane jako bajty z najbardziej znaczącym bajtem mającym bity $[x_{w-1}, x_{w-2}, \dots, x_{w-8}]$ oraz najmniej znaczącym bajtem mającym bity $[x_7, x_6, \dots, x_0]$ oraz pozostałymi bajtami mającymi kolejne bity ze środka. Konwencja ta nazywa się **big endian**. Konwencja odwrotna, w której najmniej znaczące bity są z przodu nazywa się **little endian**. Zobaczmy to na przykładzie.

Założymy że zmienna x typu int o adresie 0x100 ma wartość szesnastkową 0x01234567. Uporządkowanie bajtów o adresach od 0x100 do 0x103 zależy od typu maszyny:

Big endian:

...	0x100	0x101	0x102	0x103	...
...	01	23	45	67	...

oraz:

Little endian:

...	0x100	0x101	0x102	0x103	...
...	67	45	23	01	...

Zauważmy że w słowie 0x01234567 high-order bit ma wartość szesnastkową 0x01, a low-order bit ma wartość 0x67

Uporządkowanie bitów czasami jest problemem.

Jednym z przykładów takich problemów jest wtedy kiedy dane w formie binarnej są przysyłane w sieci pomiędzy różnymi maszynami. Problemem jest odbiór danych w zapisanych w formacie big endian przez maszyny, która operuje w formacie little endian i vice versa. Żeby uniknąć tych problemów kod napisany dla takich aplikacji musi przestrzegać pewnych konwencji. Przyjrzyjmy się temu w rozdziale 11.

Drugim ważnym miejscem gdzie porządek bitów jest kluczowy, jest wtedy gdy patrzymy na ciągi bajtów reprezentujących liczby całkowite (integer data). To się zdarza często gdy patrzymy na kod maszynowy programów. Jako przykład spojrzymy na wiersz pochodzący z pliku który zwraca w formie tekstu kod maszynowy dla procesora Intel x86-64:

```
4004d3:    01 05 43 0b 20 00    add    %eax,0x200b43(%rip)
```

Wiersz ten został wygenerowany przez disassembler, narzędzie służące do określenia ciągu instrukcji reprezentowanych przez plik wykonywalny (executable file). Wiersz ten mówi nam, że szesnastkowa wartość ciągu bitów: 01 05 43 0b 20 00 bitowa reprezentacja instrukcji która dodaje jedno słowo danych do wartości przechowywanej pod adresem obliczonym przez dodanie 0x200b43 do aktualnej wartości 'program counter'- licznik programu?, adresu następnej instrukcji do wykonania.

Jeśli weźmiemy ostatnie 4 bajty z ciągu 43 0b 20 00 i zapiszemy je odwrotnie, dostaniemy 00 20 0b 43. Ignorując początkowe zera otrzymamy wartość 0x200b43, czyli wartość liczbowa po prawej stronie instrukcji. Występowanie bajtów w odwrotnej kolejności jest częstym zjawiskiem przy czytaniu kodu maszynowego wygenerowanego dla little-endian maszyny, takiej jak ta powyżej. Naturalny sposób na zapisanie bajtów jest taki, że najniższy numerowany bit jest po lewej stronie a najwyższy numerowany jest po prawej. Jest to sprzeczne z normalnym zapisem liczb, gdzie najbardziej znacząca cyfra jest po lewej a najmniej znacząca po prawej.