Moduł 5

Techniki programowania asemblerowego

Jawna specyfikacja rozmiaru argumentu w x86

- W większości instrukcji rozmiar argumentu jest określony przez nazwę rejestru, np..

```
mov    eax, [ebx]    ; ładowanie słowa 32-bitowego
add    dl, [esi]     ; dodawanie bajtu
```

- Jeżeli instrukcja nie ma argumentu rejestrowego lub ma argumenty o różnej długości, rozmiar argumentu pamięciowego może być niemożliwy do określenia, np.

```
inc    [ebx]  ; długość nieokreślona – może być dowolna
movzx  eax, [esi]    ; może być bajt lub słowo 16-bitowe
```

- W takich sytuacjach jest wymagana jawna specyfikacja rozmiaru
  - W NASM przed specyfikacją argumentu umieszcza się słowo byte, word, dword, qword

```
inc    word [ebx]    ; słowo 16-bitowe
movzx  eax, byte [esi]      ; bajt
```

Plan wykładu

- Proste optymalizacje
- Procedury
- Pętle
- Korzystanie ze znaczników
- Usuwanie niektórych skoków
- Optymalizacja wywołań procedur
- Błędy i złe nawyki w programowaniu asemblerowym

Zerowanie i testowanie wartości rejestru w x86

- zerowanie: xor eax, eax
  - kod binarny krótszy niż mov eax, 0
- testowanie (sprawdzenie czy wartość zerowa lub ujemna):
  test eax, eax
  js ujemne
  jz zero
  - najnowsze procesory traktują instrukcję test specjalnie (dawniej używano w tym samym celu również instrukcji or i and)

Mnożenie i dzielenie przez potęgi liczby 2

- Mnożenie przez 2^n – przesunięcie bitowe w lewo o n
- Dzielenie przez 2^n – przesunięcie bitowe w prawo o n
  - Dla liczb ze znakiem – przesunięcie arytmetyczne
- Reszta z dzielenia przez 2^n – iloczyn logiczny z 2^n-1

Zastosowanie instrukcji LEA - x86

- Trójargumentowe mnożenie przez 2, 4, 8
- Dwu- i trójargumentowe mnożenie przez 3, 5, 9
- Trój- i czteroargumentowe dodawanie
- Trójargumentowe przesunięcie w lewo o 1, 2, 3
- Przesunięcie w lewo z dodawaniem lub sumą logiczną
  - Jeżeli bity o wartości 1 w argumentach operacji nie nakładają się, suma arytmetyczna jest równoważna sumie logicznej

Procedury

- Procedura to fragment programu przeznaczony do wielokrotnego wykonania
- Wywołanie procedury jest kosztowne czasowo
  - skok ze śladem
  - powrót według śladu
- Procedur nie używamy, gdy:
  - wywołanie następuje tylko w jednym miejscu programu
    - ciało procedury należy w tym przypadku wstawić w miejsce jej wywołania, a instrukcję wywołania usunąć
  - ciało procedury wykonuje się w czasie porównywalnym z czasem wykonania skoku ze śladem i powrotu
    - można w tym przypadku zastosować mechanizm makrogeneracji dostępny w asemblerze

Procedury

- Argumenty do procedur–liści najwygodniej jest przekazywać przez rejestry
  - można użyć tego samego rejestru jako rejestru argumentu i wartości zwracanej

Przekazywanie argumentów w rejestrach

```
hex_word:       ; wypisz słowo z ax w postaci szesnastkowej
        push    ax
        mov     al, ah
        call    hex_byte
        pop     ax
        call    hex_byte
        ret

hex_byte:       ; wypisz bajt z al w postaci szesnastkowej
        aam     16
        xchg    al, ah
        call    hex_digit
        mov     al, ah
        call    hex_digit
        ret

hex_digit:      ; wypisz dolną tetradę z al
        add     al, '0'  ; zamiana na kod ASCII dla cyfr dziesiętnych
        cmp     al, '9'
        jbe     hd8
        add     al, 'A'-'9'-1    ; A..F - popraw kod ASCII
hd8:    call    putchar
        ret
```

Organizacja pętli

- W miarę możliwości pętla powinna być zakończona skokiem warunkowym zamykającym pętlę
  - cel: minimalizacja liczby instrukcji sterujących pętlą
- Pętle, których liczba iteracji jest znana przed rozpoczęciem, wygodnie jest kończyć sekwencją „zdekrementuj licznik, skocz jeśli nie zero”
  - w wielu procesorach istnieje pojedyncza instrukcja, która pełni taką rolę (w x86 – instrukcja LOOP)
- W celu zmniejszenia liczby operacji inkrementacji i dekrementacji wewnątrz pętli można zastosować inne techniki:
  - wyznaczanie zakończenia na podstawie wartości wskaźnika na dane
  - adresowanie danych przy użyciu licznika pętli jako indeksu
- W x86 można użyć instrukcji iteracyjnych

Organizacja pętli - przykład

```
; wypisz blok bajtów w postaci szesnastkowej
        ; edx - adres bloku
        ; ecx - rozmiar bloku w bajtach

hex_block1:     ; wersja 1. - jawny licznik pętli/indeks z inkrementacją
        mov     ebx, 0
hblop2:
        mov     al, [edx + ebx]  ; adresowanie indeksowe
        call    hex_byte
        inc     ebx
        cmp     ebx, ecx
        jne     hblop2
        ret

hex_block2:     ; wersja 2. - jawny licznik pętli z dekrementacją
hblop2:
        mov     al, [edx]
        inc     edx
        call    hex_byte
        dec     ecx             ; tę i następną instrukcję można zastąpić
        jnz     hblop2          ; pojedynczą instrukcją „loop hblop1”
        ret
```

Organizacja pętli - przykład

```
        ; wypisz blok bajtów w postaci szesnastkowej
        ; edx - adres bloku
        ; ecx - rozmiar bloku w bajtach

hex_block3:     ; wersja 3. - porównanie wskaźników
        add     ecx, edx            ; ecx - wskaźnik na koniec bufora (+1)
hblop3:
        mov     al, [edx]           ; adresowanie indeksowe
        inc     edx
        call    hex_byte
        cmp     edx, ecx
        jne     hblop3
        ret

hex_block4:     ; wersja 4. - z użyciem instrukcji iteracyjnych
        mov     esi, edx
hblop4:
        lodsb
        call    hex_byte
        loop    hblop4
        ret
```

Korzystanie ze znaczników

- Znaczniki są ustawiane przez wszystkie podstawowe instrukcje arytmetyczne i logiczne
  - nie tylko przez instrukcje porównań
- W x86 instrukcje jednoragumentowe ustawiają znaczniki nietypowo
  - INC, DEC – nie zmieniają stanu CF
    - w celu sprawdzenia, czy nie nastąpiła dekrementacja poniżej zera można użyć instrukcji JS lub JNS, ale nie JB czy JAE
    - Dzięki temu przeniesienie może propagować do następnego obiegu pętli
  - niekiedy można z tego skorzystać używając dwóch instrukcji ustawiających znaczniki – dwuargumentowej, dekrementacji, a potem skoku warunkowego
    - w ten sposób można szybko uzyskać iloczyn lub sumę dwóch warunków jako warunek skoku

Operacje wielokrotnej precyzji

- Znacznik przeniesienia może być użyty do realizacji operacji na danych, których rozmiar przekracza pojemność rejestrów
- Możliwe operacje:
  - dodawanie
  - odejmowanie
  - przesunięcia i rotacje o jeden bit

Dodawanie i odejmowanie wielokrotnej precyzji

```
Add64:   ; dodawanie danych 64-bitowych
         ; edx - adres źródła/przeznaczenia
         ; ebx – adres źródła
         mov      eax, [ebx]
         add      [edx], eax        ; dla odejmowania - zastąpić „sub”
         mov      eax, [ebx + 4]
         adc      [edx + 4], eax    ; dla odejmowania - zastąpić „sbb”
         ret

add_nx32:         ; dodawanie danych n*32-bitowych
         ; edx, ebx – adresy przeznaczenia, źródła
         ; ecx – licznik słów (n)
         clc      ; zerowanie przeniesienia przed pierwszym dodawaniem
alop1:
         mov      eax, [ebx]
         adc      [edx], eax        ; dla odejmowania - zastąpić „sbb
         lea      ebx, [ebx + 4]    ; nie modyfikuje znacznika przeniesienia
         lea      edx, [edx + 4]    ; nie modyfikuje znacznika przeniesienia
         loop     alop1
```

Przesunięcia wielokrotnej precyzji

```
; przesunięcie o jeden bit w prawo danej 128-bitowej
shr128:
        ; edx - adres źródła/przeznaczenia
        shr     dword [edx + 12], 1       ; CF <- lsb
        rcr     dword [edx + 8], 1        ; msb <- CF, CF <- lsb
        rcr     dword [edx + 4], 1        ; msb <- CF, CF <- lsb
        rcr     dword [edx], 1            ; msb <- CF
        ret

; przesunięcie o jeden bit w lewo danej n*32-bitowej
        ; edx – adres źródła/przeznaczenia
        ; ecx – licznik słów (n)
        clc     ; zerowanie przeniesienia przed pierwszym przesunięci
slop1:
        rcl     dword [edx], 1
        lea     edx, [edx + 4]   ; nie modyfikuje przeniesienia
        loop    slop1
        ret

; przesunięcie eax:edx o n bitów w lewo przy użyciu instrukcji SHLD
        shld    eax, edx, cl     ; liczba pozycji musi być zapisana w
        shl     edx, cl
```

Usuwanie skoków

- Skoki są we współczesnych procesorach bardzo kosztowne!
  - np. Pentium 4 – nawet do ok. 100 instrukcji
  - inne procesory x86 – koszt źle przewidzianego skoku to kilkadziesiąt instrukcji
- Skok jest nieprzewidywalny lub trudno przewidywalny jeśli:
  - zależy od wartości danych
  - ma zmienny adres docelowy
  - występuje w pobliżu innych skoków (np. więcej niż dwa skoki w obrębie kilku kolejnych instrukcji)
- Często usunięcie jednego skoku umożliwia lepsze przewidywanie innego skoku
- Wniosek: należy minimalizować liczbę skoków w programie
  - skoki można zastąpić innymi instrukcjami warunkowymi lub instrukcjami arytmetycznymi

Usuwanie skoków - przykład

```
; wersja z jednym skokiem

hex_digit:      ; wypisz dolną tetradę z al
        add     al, '0'  ; zamiana na kod ASCII dla cyfr dziesiętnych
        cmp     al, '9'
        jbe     hd8
        add     al, 'A'-'9'-1    ; A..F - popraw kod ASCII
hd8:    call    putchar
        ret

; wersja bez skoku, z użyciem instrukcji korekcji dziesiętnej

hex_digit:      ; wypisz dolną tetradę z al
        add     al, 90h  ; 90..9F
        daa              ; 90..99 lub 00..05 i ustawiony CF
        adc     al, 40h  ; D0..D9 lub 41..46
        daa              ; 30..39 lub 41..46 - kod cyfry
        call    putchar
        ret
```

Usuwanie skoków - przykład

```
; wersja bez skoku, z użyciem przesłania warunkowego

hex_digit:          ; wypisz dolną tetradę z al
        add     al, '0'  ; zamiana na kod ASCII dla cyfr dziesiętnych
        lea     edx, [eax + 'A'-'9'-1]    ; dla cyfr 'A'..'F'
        cmp     al, '9'
        cmova   eax, edx
        call    putchar
        ret
```

Optymalizacja wywołań procedur

- Wywołanie procedury jest operacją kosztowną
  - skok ze śladem i powrót według śladu – modyfikują PC
  - dostępy do stosu w pamięci przy składowaniu/odtwarzaniu śladu
- Przy programowaniu asemblerowym często wywołanie procedury występuje jako ostatnia akcja innej procedury
  - sekwencja instrukcji call proc1/ret
- Przed wywołaniem “call proc1” na wierzchołku stosu znajduje się ślad powrotu z procedury wywołującej
- Przy powrocie kolejno wykonają się dwie instrukcje ret
  - powrót z procedury proc1 do instrukcji ret
  - powrót z procedury wywołującej
- Jeśli taka sekwencja zostanie zastąpiona skokiem do proc1, instrukcja powrotu z proc1 zdejmie ze stosu ślad procedury wywołującej – oszczędzamy wykonanie call i ret

Optymalizacja wywołań - przykład

```
hex_word:       ; wypisz słowo z ax w postaci szesnastkowej
        push    ax
        mov     al, ah
        call    hex_byte
        pop     ax
        ;jmp    hex_byte - instrukcja zbędna - usunięta

hex_byte:       ; wypisz bajt z al w postaci szesnastkowej
        aam     16
        xchg    al, ah
        call    hex_digit
        mov     al, ah
        ;jmp    hex_digit - instrukcja zbędna - usunięta

hex_digit:      ; wypisz dolną tetradę z al
        add     al, 90h  ; 90..9F
        daa              ; 90..99 lub 00..05 i ustawiony CF
        adc     al, 40h  ; D0..D9 lub 41..46
        daa              ; 30..39 lub 41..46 - kod cyfry
        jmp     putchar
        ; procedura putchar wykona powrót do procedury wywołującej
```

Minimalistyczna funkcja main() w asemblerze x86-64, Linux

```
; asemblacja:           nasm -f elf64 mmain64.s
; konsolidacja:         cc -o cmain64 mmain64.o

        section .data
msg:    db '64-bit assembly minimalistic main',10, 0

        section .text
        global main     ;
        extern puts
main:
        mov     rdi, msg
        jmp     puts    ; will return to crt0
        ; …with random exit code
```

strrev – odwracanie łańcucha znaków

```
        ; esi - wskaznik na poczatek
reverse: mov     edi, esi ; kopia wskaznika na pierwszy bajt
flop:    lodsb
         test    al, al
         jnz     flop
         sub     esi, 2   ; wskaźnik na ostatni bajt
revlop:
         mov     al, [edi]        ; odczyt z przodu
         xchg    al, [esi]        ; zamiana z tyłem
         stosb                    ; zapis do przodu i przesunięcie wsk. przodu
         dec     esi              ; przesunięcie wskaźnika tyłu
         cmp     esi, edi
         ja      revlop
         ret
;=============================================================
; zastosowanie instrukcji SCAS do poszukiwania konca
reverse: mov     esi, edi
         mov     ecx, 0xffffffff
         xor     al, al
         repne scasb
         sub     edi, 2
         xchg    esi, edi
         jmp     revlop
```

Odwracanie kolejności bitów w słowie

- Częsty problem – grafika, transmisja danych
- Nowe procesory mają do tego celu specjalną instrukcję (np. bitrev w ARMv7)
- Rozwiązanie trywialne - „przelewanie” bitów przy użyciu przesunięć lub rotacji w dwie strony
  - wolne – pętla o liczbie obiegów równej liczbie bitów w słowie
- Rozwiązanie szybkie – zamiana kolejności grup 2^k bitów w obrębie 2^{k+1} bitów
  - liczba kroków równa log2 liczby bitów w słowie
  - krok: x = ((x & m) << k) | (x & ~m) >> k
  - maska m zawiera naprzemiennie grupy k zer i k jedynek
  - poszczególne kroki można optymalizować korzystając ze specyficznych instrukcji procesora (rotacja, zamiana bajtów, mnożenie z akumulacją, LEA)

Odwracanie kolejności bitów w bajcie

```
; przelewanie – 5 instrukcji w programie, 26 wykonywanych
        mov     ecx, 8
        mov     ah, al
brlop:  rcr     ah, 1   ; bit do CF, można zastąpić przez shr
        rcl     al, 1   ; albo adc al, al
        loop    brlop

; zamiany grup – 11 instrukcji (9 z użyciem aad)
        mov     ah, al
        and     ax, 0b0101010110101010   ; bity parzyste, nieparzyste
        shr     al, 1
        shl     ah, 1   ; tę i następną instr. można zastąpić przez „aad 2”
        or      al, ah  ; zamienione bity w parach
        mov     ah, al
        and     ax, 0b0011001111001100   ; pary parzyste, nieparzyste
        shr     al, 2
        shl     ah, 2   ; tę i następną instr. można zastąpić przez „aad 4”
        or      al, ah  ; zamienione pary w tetradach
        rol     al, 4   ; zamiana tetrad przez rotację
```

Odwracanie kolejności bitów w słowie

```
; przelewanie – 5 instrukcji w programie, 98 wykonywanych
        mov     ecx, 32
        mov     edx, eax
brlop:  rcr     edx, 1   ; wysuwany bit do CF, można zastąpić przez shr
        rcl     eax, 1   ; albo adc eax, eax
        loop    brlop

; zamiany grup – 15 instrukcji
        mov     edx, eax
        and     eax, 0b10101010101010101010101010101010    ; bity nieparzyste
        and     edx, 0b01010101010101010101010101010101    ; bity parzyste
        shr     eax, 1
        lea     eax, [eax + edx*2]          ; z przesunieciem o 1 w lewo
        mov     edx, eax
        and     eax, 0b11001100110011001100110011001100    ; pary nieparzyste
        and     edx, 0b00110011001100110011001100110011    ; pary parzyste
        shr     eax, 2
        lea     eax, [eax + edx*4]          ; z przesunieciem o 2 w lewo
        lea     edx, [eax*4]      ; przesuniete o 2 w lewo
        and     eax, 0b11110000111100001111000011110000    ; tetrady niep.
        and     edx, 0b00111100001111000011110000111100    ; tetrady parzyste
        shr     eax, 4
        lea     eax, [eax + edx*4]          ; z przesunieciem o 2 w lewo
        bswap   eax       ; zamiana kolejnosci bajtow w s;owie 32-bitowym
```

Zliczanie jedynek w słowie

- Rozwiązanie trywialne – pętla z przesuwaniem argumentu, testowaniem jednego bitu i zliczaniem – n obiegów, ok. 5 instrukcji na obieg
- Podejście szybkie:
  - Traktujemy słowo jako wektor 1-bitowych liczników jedynek na poszczególnych pozycjach – pozostaje zsumować te liczniki
  - Sumowanie: bitów w parach, par w tetradach, tetrad w oktetach itd..
  - Liczba faz – $\log_2 n$, ok. 5 instrukcji na fazę
  - Możliwa optymalizacja sumowania bitów w parach oraz akumulacji oktetów
- W niektórych procesorach jest dostępna instrukcja POPCNT (x86 od 2010 r.)

Błędy i złe nawyki

- Używanie instrukcji porównania do powtórnego ustawienia znaczników już ustawionych przez wcześniejszą instrukcję arytmetyczną lub logiczną

```
dec     ecx     ; ustawia znaczniki, w tym ZF
cmp     ecx, 0 ; zbędna, powtórnie ustawi ZF
je      koniec
```

- Zbędny skok bezwarunkowy wynikający z niepotrzebnego odwrócenia warunku skoku, np.

```
        jne     dalej
        jmp     bylo_zero
dalej:
```

- Użycie instrukcji mnożenie i dzielenia do mnożenie i dzielenia przez potęgi liczby 2
  - Należy do tego celu używać przesunięć bitowych i LEA

Błędy i złe nawyki

- Używanie zmiennych statycznych do tymczasowego przechowywania danych lub zawartości rejestrów
  - do tego służy stos i instrukcje PUSH, POP
    - patrz przykład – procedura hex_word
- Używanie wartości liczbowych kodów ASCII przepisanych z tabeli
  - znakomicie zaciemnia program
  - asemblery rozpoznają stałe znakowe, podobnie jak kompilatory C
    0x48, 0x65, 0x6C, 0x6C, 0x6F → 'H','e','l','l','o'
- Nieznaczące nazwy etykiet w istotnych miejscach programu
  - np. „L12”, „petla”