Laborator 05: Rolul registrelor, adresare directă și bazată

În acest laborator vom aprofunda lucrul cu registre și modul în care se utilizează memoria atunci când programăm assembly pe un sistem x86 de 32 biți.

Registre

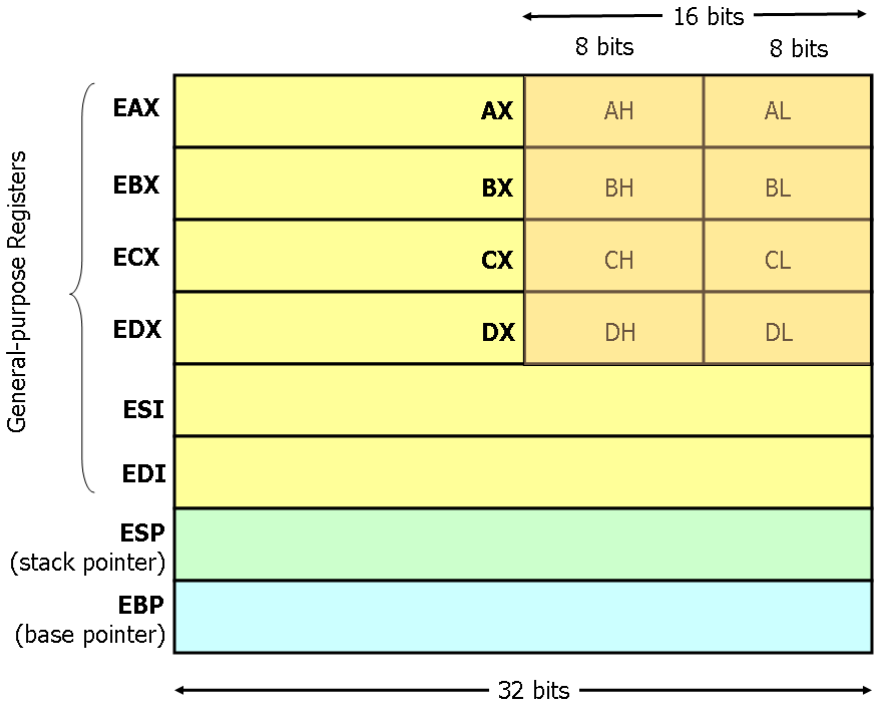
Registrele sunt principalele “unelte” cu care se scriu programele în limbaj de asamblare. Acestea sunt precum variabile construite în procesor. Utilizarea registrelor în locul adresării directe a memoriei face ca dezvoltarea și citirea programelor scrise în assembly să fie mai rapidă și mai ușoara. Singurul dezavantaj al programării în limbaj de asamblare x86 este acela că sunt puține registre.

Procesoarele x86 moderne dispun de 8 registre cu scop general a căror dimensiune este de 32 de biți. Numele registrelor sunt de natură istorică (spre exemplu: EAX era numit registru acumulator din cauza faptului că este folosit de o serie de instrucțiuni aritmetice, cum ar fi idiv). In timp ce majoritatea registrelor și-au pierdut scopul special, devenind “general purpose” în ISA-ul modern, prin convenție, 2 și-au pastrat scopul inițial: esp (stack pointer) și ebp (base pointer).

Subsecțiuni ale registrelor

În anumite cazuri dorim să modificăm valori ce sunt reprezentate pe mai puțin de 4 octeți (spre exemplu, lucrul cu șiruri de caractere). Pentru aceste situații, procesoarele x86 ne oferă posibilitatea de a lucra cu subsectiuni de 1, respectiv 2 octeți ale registrelor EAX, EBX, ECX, EDX.

În imaginea de mai jos sunt reprezentate registrele, subregistrele și dimensiunile lor.



Subregistrele fac parte din registre, ceea ce înseamnă că dacă modificăm un registru, în mod implicit modificăm și valoarea subregistrului.

Subregistrele se folosesc în mod identic cu registrele, doar că dimensiunea valorii reținute este diferită.

Declarări statice de regiuni de memorie

Declarările statice de memorie (analoage declarării variabilelor globale), în lumea x86, se fac prin intermediul unor directive de asamblare speciale. Aceste declarări se fac în secțiunea de date (regiunea .DATA). Porțiunilor de memorie declarate le pot fi atașate un nume prin intermediul unui label pentru a putea fi referite ușor mai târziu în program.

Urmăriți exemplul de mai jos:

```
.DATA
    var         DB 64    ; Declare a byte containing the value 64. Label the
                         ; memory location “var”.
    var2        DB ?     ; Declare an uninitialized byte labeled “var2”.
                DB 10    ; Declare an unlabeled byte initialized to 10. This
                         ; byte will reside at the memory address var2+1.
    X           DW ?     ; Declare an uninitialized two-byte word labeled
“X”.
    Y           DD 3000  ; Declare 32 bits of memory starting at address “Y”
                         ; initialized to contain 3000.
    Z           DD 1,2,3 ; Declare three 4-byte words of memory starting at
                         ; address “Z”, and initialized to 1, 2, and 3,
                         ; respectively. E.g. 3 will be stored at address Z+8
```

DB,DW,DD sunt directive folosite pentru a specifica dimensiunea porțiunii : 1,2, respectiv 4 bytes.

Ultima declarare din exemplul de mai sus reprezintă declararea unui vector. Spre deosebire de limbajele de nivel mai înalt, unde vectorii pot avea multiple dimensiuni, iar elementele lor sunt accesate prin indici, în limbajul de asamblare vectorii sunt reprezentați ca un număr de celule ce se află intr-o zonă contiguă de memorie.

Adresarea Memoriei

Procesoarele x86 moderne pot adresa pana la 2^32 bytes de memorie, ceea ce înseamnă că adresele de memorie sunt reprezentate pe 32 de biți. Pentru a adresa memoria, procesorul folosește adrese (implicit, fiecare label este translatat într-o adresa de memorie corespunzătoare). Pe lânga label-uri mai există și alte forme de a adresa memoria:

```
mov eax, [0xcafebab3]         ; direct (displacement)
mov eax, [esi]                ; register indirect (base)
mov eax, [ebp-8]              ; based (base + displacement)
mov eax, [ebx*4 + 0xdeadbeef] ; indexed (index*scale + displacement)
mov eax, [edx + ebx + 12]     ; based-indexed w/o scale (base + index +
displacement)
mov eax, [edx + ebx*4 + 42]   ; based-indexed w/ scale (base + index*scale +
displacement)
```

Următoarele adresări sunt invalide:

```
mov eax, [ebx-ecx]     ; Can only add register values
mov [eax+esi+edi], ebx ; At most 2 registers in address computation
```

Directive de dimensiune

În general, dimensiunea pe care este reprezentată o valoare ce este adusă din memorie poate fi inferată (dedusă) din codul instrucțiunii folosite. Spre exemplu, în cazul adresărilor de mai sus, dimensiunea valorilor putea fi inferată din dimensiunea registrului destinație, însă în anumite cazuri acest lucru nu este atât de evident. Să urmarim urmatoarea instrucțiune:

```
mov [ebx], 2
```

Dupa cum se observa, se dorește stocarea valorii 2 la adresa conținută de registrul ebx. Dimensiunea registrului este de 4 bytes. Valoarea 2 poate fi reprezentată atat pe 1 cât și pe 4 bytes. În acest caz, din moment ce ambele interpretări sunt valide, procesorul are nevoie de informații suplimentare despre cum să trateze această valoare. Acest lucru se poate face prin directivele de dimensiune:

```
mov BYTE PTR [ebx], 2  ; Move 2 into the single byte at memory location EBX
```

```
mov WORD PTR [ebx], 2  ; Move the 16-bit integer representation of 2 into
the 2 bytes starting at
                       ; address EBX
mov DWORD PTR [ebx], 2 ; Move the 32-bit
```

Tutoriale și exerciții

În cadrul exercițiilor vom folosi arhiva de laborator.

Descărcați arhiva, decomprimați-o și accesați directorul aferent.

Tutorial: Înmulțire două numere reprezentate pe un octet

Parcurgeți rulați și testați codul din fișierul `multiply.asm`. În cadrul programului înmulțim două numere definite ca octeți. Pentru a le aputea accesa folosimo construcție de tipul `byte [register]`.

Atunci când facem înmulțire procesul este următorul, așa cum este descris și aici:

1. Plasăm înmulțitorul în registrul AL (pentru operații pe un byte), registrul AX (pentru operații pentru cuvânt - 16 biți, 2 octeți) și registrul EAX' (pentru operații pe dublu cuvânt - 32 de biți, 4 octeți).
2. Deînmulțitul este transmis ca argument mnemonicii `mul`.
3. Rezultatul este plasat în două registre (partea *high* și partea *low*).

Testați programul. Încercați alte valori pentru `num1` și `num2`.

Înmulțire două numere

Actualizați zona marcată cu `TODO` în fișierul `multiply.asm` pentru a permite înmulțirea și a numelor de tip `word` și `dword`, adică `num1_dw` cu `num2_dw`, respectiv `num1_dd` și `num2_dd`.

Pentru înmulțirea numerelor de tip `word` (pe 16 biți), componentele sunt dispuse astfel:

- În registrul AX se plasează înmulțitorul.
- Argumentul instrucțiunii `mov` (posibil un alt registru) este pe 16 biți (fie valoare fie un registru precum BX, CX, DX).
- Rezultatul înmulțirii este dispus în perechea `DX:AX`, adică partea “high” a rezultatului în registrul DX, iar partea “low” a rezultatului în registrul AX.

Ridicare număr la puterea a treia

Realizați un program în limbajul de asamblare care ridică un număr la puterea a treia (adică `num * num * num`).

Definiți numărul în formatul `dword` adică de forma

```
    num dd 10
```

Nu definiți un număr foarte mare, pentru a putea fi vizualizat rezultatul înmulțirii în registrul `eax`.

Tutorial: Suma elementelor dintr-un vector reprezentate pe un octet

În programul `sum_array.asm` din arhiva laboratorului este calculată suma elementelor unui vector (*array*) de octeți (*bytes*, reprezentare pe 8 biți).

Urmăriți codul, observați construcțiile și registrele specifice pentru lucru cu bytes. Rulați codul.

Treceți la următorul pas doar după ce ați înțeles foarte bine ce face codul. Vă va fi greu să faceți exercițiile următoare dacă aveți dificultăți în înțelegerea exercițiului curent.

Suma elementelor dintr-un vector

În zona marcată cu `TODO` din fișierul `sum_array.asm` completați codul pentru a realiza suma vectorilor cu elemente de tip word (16 biți) și de tip dword (32 de biți); este vorba de vectorii `word_array` și `dword_array`.

Când veți calcula adresa unui element din array, veți folosi construcție de forma:

```
base + size * index
```

În construcția de mai sus:

- `base` este adresa vectorului (adică `word_array` sau `dword_array`)
- `size` este lungimea elementului vectorului (adică 2 pentru vector de word (16 biți, 2 octeți) și 4 pentru vector de dword (32 de biți, 4 octeți)
- `index` este indexul curent în cadrul vectorului

Suma pătratelor elementelor dintr-un vector

TODO

Numărul de numere negative și pozitive dintr-un vector

TODO

Numărul de numere pare și impare dintr-un vector

TODO