
# Dokumentace KIV/OS

Simulace operačního systému DOS

**prosinec 2016**

Ondřej Dvořák
Tomáš Novák
Radek Vavřík

## Zadání

Zvolili jsme si jako tým třetí zadání a to simulaci operačního systému DOS. Zadání jsme si zvolili z důvodu nejlepšího nacvičení a pochopení standartního běhu, jak pracuje operační systém. Chtěli jsme se seznámit jednak s architekturou, pokus se o jednoduchou virtualizaci OS a vyzkoušet si něco jiného než „standartní zadání“.

Součástí zadání byl zdrojový kód a již přeložený soubor COM v 16bit režimu. Pro spuštění bylo potřeba nainstalovat program např. DOSBox. Cílem bylo implementovat minimální emulaci MS-DOSu, která vykoná právě tento program.

## Analýza

Nejdříve jsme se museli začít orientovat v instrukční sadě procesoru Intel x86 a jeho Opcodech – ukázku demonstruje obr:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD Eb Gb *00* | ADD Ev Gv *01* | ADD Gb Eb *02* | ADD Gv Ev *03* | ADD AL Ib *04* | ADD eAX Iv *05* | PUSH ES *06* | POP ES *07* | OR Eb Gb *08* | OR Ev Gv *09* | OR Gb Eb *0A* | OR Gv Ev *0B* | OR AL Ib *0C* | OR eAX Iv *0D* | PUSH CS *0E* | TWOBYTE *0F* |
| ADC Eb Gb *10* | ADC Ev Gv *11* | ADC Gb Eb *12* | ADC Gv Ev *13* | ADC AL Ib *14* | ADC eAX Iv *15* | PUSH SS *16* | POP SS *17* | SBB Eb Gb *18* | SBB Ev Gv *19* | SBB Gb Eb *1A* | SBB Gv Ev *1B* | SBB AL Ib *1C* | SBB eAX Iv *1D* | PUSH DS *1E* | POP DS *1F* |
| AND Eb Gb *20* | AND Ev Gv *21* | AND Gb Eb *22* | AND Gv Ev *23* | AND AL Ib *24* | AND eAX Iv *25* | ES: *26* | DAA *27* | SUB Eb Gb *28* | SUB Ev Gv *29* | SUB Gb Eb *2A* | SUB Gv Ev *2B* | SUB AL Ib *2C* | SUB eAX Iv *2D* | CS: *2E* | DAS *2F* |
| XOR Eb Gb *30* | XOR Ev Gv *31* | XOR Gb Eb *32* | XOR Gv Ev *33* | XOR AL Ib *34* | XOR eAX Iv *35* | SS: *36* | AAA *37* | CMP Eb Gb *38* | CMP Ev Gv *39* | CMP Gb Eb *3A* | CMP Gv Ev *3B* | CMP AL Ib *3C* | CMP eAX Iv *3D* | DS: *3E* | AAS *3F* |
| INC eAX *40* | INC eCX *41* | INC eDX *42* | INC eBX *43* | INC eSP *44* | INC eBP *45* | INC eSI *46* | INC eDI *47* | DEC eAX *48* | DEC eCX *49* | DEC eDX *4A* | DEC eBX *4B* | DEC eSP *4C* | DEC eBP *4D* | DEC eSI *4E* | DEC eDI *4F* |
| PUSH eAX *50* | PUSH eCX *51* | PUSH eDX *52* | PUSH eBX *53* | PUSH eSP *54* | PUSH eBP *55* | PUSH eSI *56* | PUSH eDI *57* | POP eAX *58* | POP eCX *59* | POP eDX *5A* | POP eBX *5B* | POP eSP *5C* | POP eBP *5D* | POP eSI *5E* | POP eDI *5F* |
| PUSHA *60* | POPA *61* | BOUND Gv Ma *62* | ARPL Ew Gw *63* | FS: *64* | GS: *65* | OPSIZE: *66* | ADSIZE: *67* | PUSH Iv *68* | IMUL Gv Ev Iv *69* | PUSH Ib *6A* | IMUL Gv Ev Ib *6B* | INSB Yb DX *6C* | INSW Yz DX *6D* | OUTSB DX Xb *6E* | OUTSW DX Xv *6F* |
| JO Jb *70* | JNO Jb *71* | JB Jb *72* | JNB Jb *73* | JZ Jb *74* | JNZ Jb *75* | JBE Jb *76* | JA Jb *77* | JS Jb *78* | JNS Jb *79* | JP Jb *7A* | JNP Jb *7B* | JL Jb *7C* | JNL Jb *7D* | JLE Jb *7E* | JNLE Jb *7F* |
| ADD Eb Ib *80* | ADD Ev Iv *81* | SUB Eb Ib *82* | SUB Ev Ib *83* | TEST Eb Gb *84* | TEST Ev Gv *85* | XCHG Eb Gb *86* | XCHG Ev Gv *87* | MOV Eb Gb *88* | MOV Ev Gv *89* | MOV Gb Eb *8A* | MOV Gv Ev *8B* | MOV Ew Sw *8C* | LEA Gv M *8D* | MOV Sw Ew *8E* | POP Ev *8F* |
| NOP *90* | XCHG eAX eCX *91* | XCHG eAX eDX *92* | XCHG eAX eBX *93* | XCHG eAX eSP *94* | XCHG eAX eBP *95* | XCHG eAX eSI *96* | XCHG eAX eDI *97* | CBW *98* | CWD *99* | CALL Ap *9A* | WAIT *9B* | PUSHF Fv *9C* | POPF Fv *9D* | SAHF *9E* | LAHF *9F* |
| MOV AL Ob *A0* | MOV eAX Ov *A1* | MOV Ob AL *A2* | MOV Ov eAX *A3* | MOVSB Xb Yb *A4* | MOVSW Xv Yv *A5* | CMPSB Xb Yb *A6* | CMPSW Xv Yv *A7* | TEST AL Ib *A8* | TEST eAX Iv *A9* | STOSB Yb AL *AA* | STOSW Yv eAX *AB* | LODSB AL Xb *AC* | LODSW eAX Xv *AD* | SCASB AL Yb *AE* | SCASW eAX Yv *AF* |
| MOV AL Ib *B0* | MOV CL Ib *B1* | MOV DL Ib *B2* | MOV BL Ib *B3* | MOV AH Ib *B4* | MOV CH Ib *B5* | MOV DH Ib *B6* | MOV BH Ib *B7* | MOV eAX Iv *B8* | MOV eCX Iv *B9* | MOV eDX Iv *BA* | MOV eBX Iv *BB* | MOV eSP Iv *BC* | MOV eBP Iv *BD* | MOV eSI Iv *BE* | MOV eDI Iv *BF* |
| #2 Eb Ib *C0* | #2 Ev Ib *C1* | RETN Iw *C2* | RETN *C3* | LES Gv Mp *C4* | LDS Gv Mp *C5* | MOV Eb Ib *C6* | MOV Ev Iv *C7* | ENTER Iw Ib *C8* | LEAVE *C9* | RETF Iw *CA* | RETF *CB* | INT3 *CC* | INT Ib *CD* | INTO *CE* | IRET *CF* |
| #2 Eb 1 *D0* | #2 Ev 1 *D1* | #2 Eb CL *D2* | #2 Ev CL *D3* | AAM Ib *D4* | AAD Ib *D5* | SALC *D6* | XLAT *D7* | ESC 0 *D8* | ESC 1 *D9* | ESC 2 *DA* | ESC 3 *DB* | ESC 4 *DC* | ESC 5 *DD* | ESC 6 *DE* | ESC 7 *DF* |
| LOOPNZ Jb *E0* | LOOPZ Jb *E1* | LOOP Jb *E2* | JCXZ Jb *E3* | IN AL Ib *E4* | IN eAX Ib *E5* | OUT Ib AL *E6* | OUT Ib eAX *E7* | CALL Jz *E8* | JMP Jz *E9* | JMP Ap *EA* | JMP Jb *EB* | IN AL DX *EC* | IN eAX DX *ED* | OUT DX AL *EE* | OUT DX eAX *EF* |
| LOCK: *F0* | INT1 *F1* | REPNE: *F2* | REP: *F3* | HLT *F4* | CMC *F5* | #3 Eb *F6* | #3 Ev *F7* | CLC *F8* | STC *F9* | CLI *FA* | STI *FB* | CLD *FC* | STD *FD* | #4 INC/DEC *FE* | #5 INC/DEC *FF* |

Na rozdíl od klasického asembleru existuje pro jednu instrukci celá sada opcodů, které se liší parametry. Některé instrukce zapisují přímo do konkrétních registrů, jiné mají zdroj a cíl operace zadány jako parametr. Instrukce se také liší tím, jak kolik bytů načítají jako parametr, s jak velkou částí registru pracují. (Např. zda s celým EAX, nebo jen s AX), zda zapisují na konkrétní paměťové místo přímo a nebo pro výpočet používají další registry. Výpočet takové adresy může být tedy i dán například jako součet DX + ES + hodnota zadaná parametrem. Další kapitolou jsou instrukční prefixy, což jsou prakticky samostatné instrukce, které mění chování následující instrukce. I těchto prefixů může být více. V modelovém příkladu jsme narazilui maximálně na dva zároveň (66 – modifikoval délku registru – tedy že se má pracovat s celým EAX a 26, který říkal, že se nemá použít DS, ale ES pro určení segmentu)

Další rozhodnutí bylo, jakým programovacím jazykem naši semestrální práci budeme vytvářet. Rozhodovali jsme se mezi C# a C. Z důvodu rychlosti, přímého přístupu do paměti, jednodušší správě jsme se rozhodli pro jazyk C.

## Zpracování

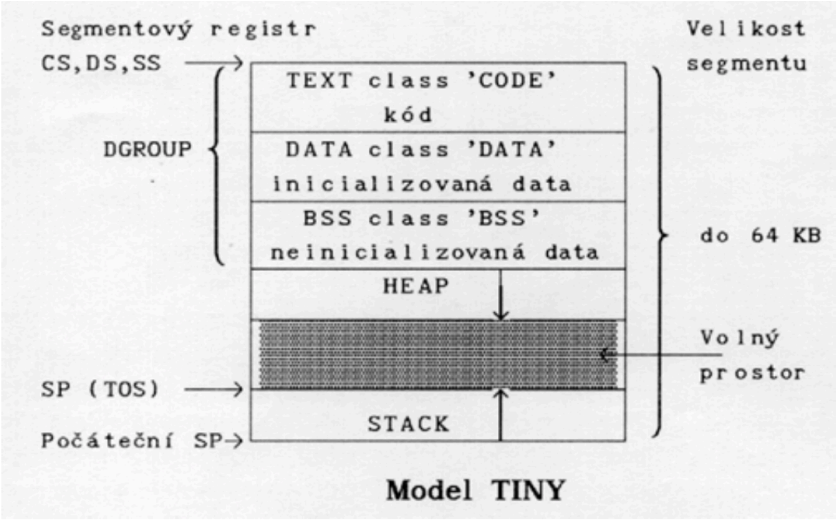
Vývoj programu byl primárně na OS MacOS, debagován v TurboDebageru, testován na Windows 7 64bit, testován ve DOSBoxu.

Program prošel několika významnými úpravami. Tyto úpravy pramenily z našich úvah, průběžné prezentace či emailové konzultace se zadavatelem.

Abychom se vyhnuli častým programátorským chybám v alokování paměti, tak jsme nejdříve používali statické pole, nepoužívali pointery. Ve finální verzi jsme potřebný segment alokovali standardně na hromadě a ne v zásobníku.

Jelikož byl modelový program v paměťovém režimu TINY, bylo jisté, že si vystačíme s alokací jednoho celého segmentu pro kód i data.

Na počátku jsme si vynulovali registry a nastavili zásobník na konec segmentu. Ten jsme ale nakonec nepoužili.



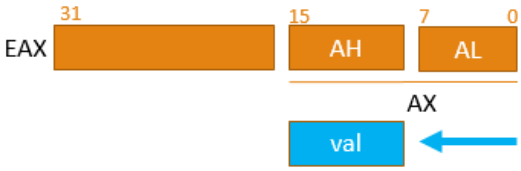
Model TINY

Původní logika zápisu do registru byla zvolena takto:

```
reg32 registers[REG_COUNT];

registers[AX] = 0x1h;

#define EAX 0
#define EBX 1
#define ECX 2
#define EDX 3

#define ESI 4
#define EDI 5

#define ESP 6
#define EBP 7
```

Zápis do registru AH

```
val = memory[registers[IP]];          // načtu 8 bitů
val <<= 8;                            // posunu doleva do tvaru xxxxxxxx00000000
registers[EAX] &= 0xFFFF00FF;         // vynuluji AH cast
registers[EAX] |= val;                // zapíšu bity do AH
```




Zápis 8 bitů do registru

```
reg1 = getRMregister(1,RMcode);
reg2 = getRMregister(2, RMcode);
registers[reg1] = registers[reg2] % 0xFF;   // pracuje pouze s bytem
```

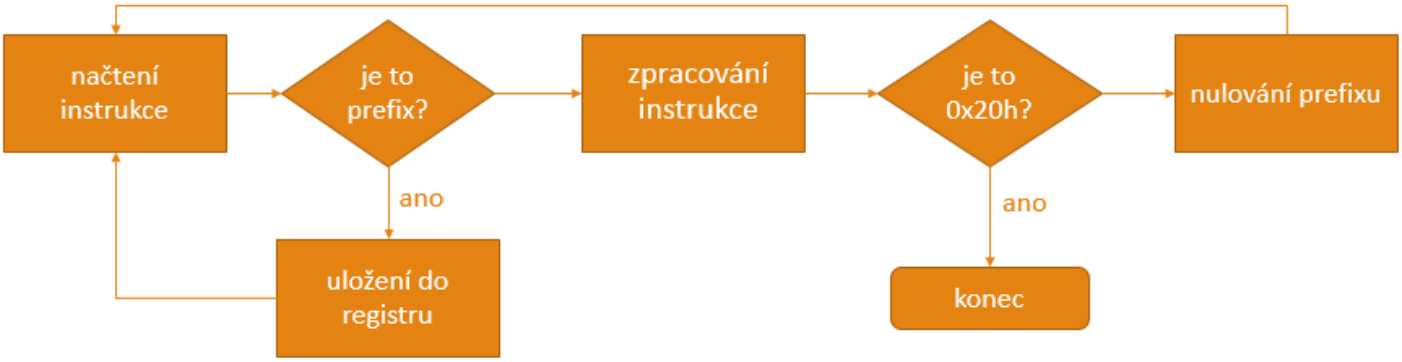
Bylo zvoleno jedno velké 32bitové pole pro všechny registry, do kterého jsme se odkazovali přes indexy definované jako konstanty. Ve finální verzi jsme ale použili union, který je schopen pro jedno paměťové místo použít různé datové typy. Tedy jak 8, 16 tak i 32 bitové číslo. Jediný zádrhel byl, pokud jsme potřebovali do unionu vměstnat dvě proměnné zároveň. Registry AH a AL, protože jsme potřebovali zapisovat i do AH (viz obrázek). Tento problém však bylo možné vyřešit vložením struktury do unionu. Výsledná datová striktura pro jeden registr je tedy následující:

```
typedef unsigned long  reg32;
typedef unsigned int   reg16;
typedef unsigned char byte;

typedef union{
  byte al, ah;
  reg16 ax;
  reg32 eax;
} regAX;

al a ah se vzájemně přepisují

reax.al = 2;
printf("ax %d \n", reax.ax);
reax.ah = 3;
printf("ax %d \n", reax.ax);

Vypíše:
ax 2
ax 3
```

```
//vyreseny problem
typedef union
  {
    reg16 ax;
    reg32 eax;
    struct
    {
      byte al;
      byte ah;
    };
  } regAX;

Nyní stejný kód korektně vypíše:
ax 2
ax 770
```

Všechny registry pak byly poskládány do struktury, jejíž pointer se předával mezi jednotlivými funkcemi zpracovávající instrukce.

Celkové zpracování programu funguje podle tohoto diagramu:



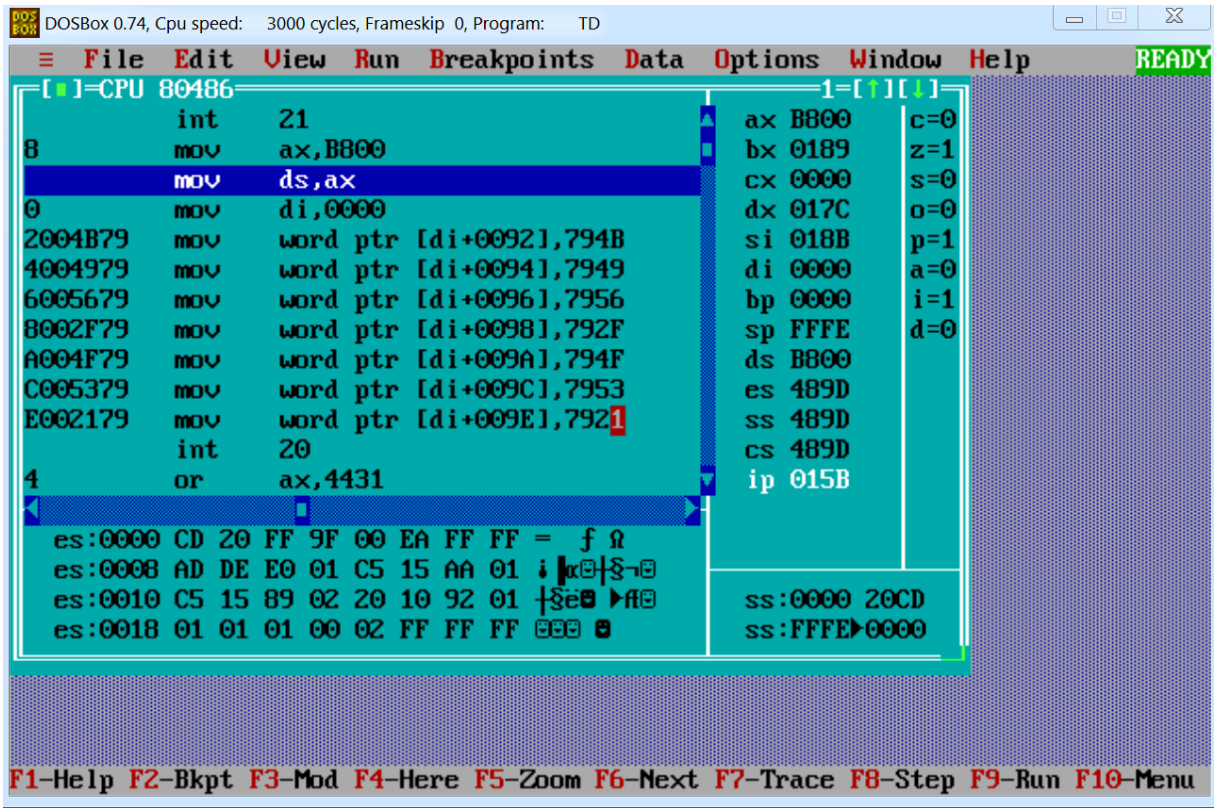

Zpracování jednotlivých instrukcí jsme vyřešili přes velký switch, kde je vidět že jakmile má přijít ke zpracování nějaké instrukce, bude vždy zavolána konkrétní funkce. Parametry instrukce jsou načítány až v konkrétních funkcích, což celý program velmi zpřehledňuje.

```
switch (instr) // pro každou istrukci se volá extra funkce. Načtení parametrů a posun IP se dělá až v té funkci
{
    case MOVEAX: ret = moveEax(&regs, memory); break;
    case MOVEBX: ret = moveEbx(&regs, memory); break;
    case MOVEDX: ret = moveEdx(&regs, memory); break;
    case MOVESI: ret = moveEsi(&regs, memory); break;
    case MOVE8C: ret = move8C(&regs, memory); break;
    case MOVE8E: ret = move8E(&regs, memory); break;
    case XOR: ret = xor_instr(&regs, memory); break;
    case JMP: ret = jmp(&regs, memory); break;
    case INC: ret = inc(&regs, memory); break;
    case ADD: ret = add(&regs, memory); break;
    case MOVE8A: ret = move8A(&regs, memory); break;
    case INCEBX: ret = incebx(&regs, memory); break;
    case INCEDX: ret = incedx(&regs, memory); break;
    case DECECX: ret = dececx(&regs, memory); break;
    case MOVEAH: ret = moveah(&regs, memory); break;
    case ADDADDR: ret = addaddr(&regs, memory); break;
    case CMP: ret = cmp(&regs, memory); break;
    case JNE: ret = jne(&regs, memory); break;
    case MOVEDI: ret = movedi(&regs, memory); break;
    case MOVEC7: ret = movec7(&regs, memory); break;
```

Jednotlivé instrukce jsou uloženy v souboru instrukce.c, jsou nazvány dle svého chování a instrukční sady. Pro názornost přikládáme náhled instrukce ADD:

```
// součet
// opět rm mod pro obecne registry
int add(REG *regs, byte *memory)
{
  byte RMcode=0;
  reg16 *reg1, *reg2;

  RMcode = memory[regs->ip+1];

  reg1 = getRMregister(1, RMcode, regs);
  reg2 = getRMregister(2, RMcode, regs);
  *reg2 = *reg2 + *reg1;

  regs->ip+=2;
  if (DEBUG_MODE == 1)
    printf ("Instrukce ADD\n");
  return(0);
}
```

Při vývoji bylo obtížné, že jsme měli poměrně omezenou možnost vidět průběžné výsledky naší práce. Snažili jsme se program debutovat přes zmíněný DOSBox a TurboDebuger viz obrázek níže. V programu také máme tkz. Debug mód, který nám postupně umožnoval „vidět“ co se děje s danými registry a jejich hodnotami.



Program též obsahoval tabulku přerušení, kterou jsme realizovali jako pole ukazatelů na funkce. V modelovém příkladu jsme se setkali s třemi přerušeními.

První z nich (10h) inicializovalo grafický režim v módu VGA 80x25 při 16 barvách, kde na každý znak připadají dva byty. Jeden pro atributy a druhý pro samotný ASCII kód. Po zavolání tohoto přerušení jsme si tedy vynulovali potřebnou paměť začínající na pozici 0x800h.

Další přerušení (20h) volalo službu DOSu pro zápis řetězce ukončeného znakem $ na obrazovku. Jenže náš simulátor nemůže jen tak vypsat něco na obrazovku, protože vzorový řetězec končil odsazením řádku a program v dalších krocích zapisoval na pozice v pravém horním rohu obrazovky. Bylo tedy nutné vytvořit buffer do kterého se zapisovalo. A který se vypsal na obrazovku v okamžiku ukončení programu (21h). Protože jiný podnět mezi posledním zápisem do paměti určené pro obrazovku a koncem programu nebyl.

## Závěr

Všichni jsme si v této semestrální práci zopakovali jazyk C, pochopili jsme a naučili se několik instrukcí pro Intel x86, pochopili základy virtualizace, přerušení, práce s registry a principy fungování operačního systému. Práce byla navíc zajímavá svým nestandartním zadáním, neznámým výstupem/vstupem. Nejvíce se nám líbil skok o -1 krok na parametr předchozí instrukce, který se zpracoval jako instrukce další.

Žel se nám nepodařilo dosáhnout interpretace znakových atributů. Nenalezli jsme jednoduchý způsob jak zapisovat barvy do konzole OSX. (žel conio.h pro mac k dispozici není.) Takže náš výstup zůstává neblikavý a nudně černobílý. Ve všech ostatních parametrech však zcela totožný se zadáním.