OPERAČNÍ SYSTÉMY 1

Pohádka na dobrou noc 1

HISTORIE

Úplně na začátku byly počítače jen stroje provádějící stále stejnou operaci. Počítač se skládal z relé, elektronek a program byl „zadrátovaný do počítače“.

V 50. letech 19. století se do počítačů dostaly tranzistory a začaly se používat děrné štítky. Počítač dávkově zpracovával data z děrných štítků. V roce 1957 vznikl programovací jazyk Fortran. Fortran navrhla firma IBM pro vědecké výpočty a numerické aplikace, vznikl jako efektivnější alternativa k jazyku symbolických adres.

Roku 1964 je datován patent na integrované obvody. Od následujícího roku (1965) byly a dodnes jsou nedílnou součástí počítače integrované obvody. Objevuje se první podoba multitaskingu, zavádí se pojem proces, který označuje prováděný program a zahrnuje také dynamicky měnící se data). Tento znatelný pokrok umožňuje zavedení interaktivních systémů (počítač v reálném čase reaguje na požadavky uživatele). Mimo velkých sálových počítačů (mainframe) se objevují také první minipočítače a mikropočítače. Nejznámější počítač této doby je IBM Systém 360. Tyto počítače měly shodný soubor instrukcí, tedy mohly používat shodný software, uměly pracovat s pevnou, ale také pohyblivou délkou operandů (dat). Tyto počítače se vyráběly v tisícových sériích, šlo o první komerční využití počítače.

Od roku 1980 se v počítačích objevují integrované obvody s vysokou integrací, tedy mikroprocesory (Mikroprocesor obsahuje v jednom pouzdře celý procesor, dřívější procesory se skládaly z více obvodů). Upustilo se od mainframů a začaly se objevovat osobní počítače. Přichází éra DOSu a vznikají uživatelská grafická rozhraní. Cena je dostupnější, vzniká internet a distribuované systémy. (Intel 8080, x86, CP/M, DOS, Windows 95/NT, Unix, GNU/Linux)

Rozdělení operačních systémů

- Podle určení můžeme OS rozdělovat na:
- Mainframy – OS/400, zOS
- Serverové / Multiprocesorové – BSD, AIX, GNU/Linux, HP-UX, Solaris, Windows NT
- Desktopové - *BSD, GNU/Linux, MacOSx, Windows NT, …
- Realtime – VxWorks, QNX
- Distribuované
- Mobilní telefony – Android, Blackberry OS, iPhone OS, Symbian, Windows Phone, …
- Experimentální / Výukové – Minix, Plan 9

CO JE TO OPERAČNÍ SYSTÉM

Často se za operační systém (OS) označuje pouze jádro OS. Jádro OS se od zbytku architektury liší v tom, že jako jediné má právo provádět některé specifické operace. Například přístup k zařízení, přístup do paměti, kontrolovat stabilní přerušení atd …

Nicméně, jádro samo o sobě je často k ničemu, proto je potřeba k němu dodat věci, jako je standartní knihovna, nějaké nástroje, standartní aplikace, takže často se za OS považuje i část systému, která běží v uživatelském neprivilegovaném prostoru. Vzniká tedy terminologický chaoz.

Z tohoto důvodu je dobré rozlišovat termín operační systém a jádro operačního systému.

Vrstvy HW/SW

- Hardware
- Operační systém (OS)
- Standartní knihovna (libc, CRT)
- Systémové nástroje
- Aplikace

Hranice mezi jednotlivými vrstvami (hranice mezi jádrem OS a zbytkem) nemusí být jednoznačné, je problém rozlišit co je aplikace a co je systémový nástroj.

Aby situace nebyla až tak úplně jednoduchá, tak nám tady můžou vznikat ještě další vrstvy, které vznikají zejména použitím různých prostředků virtualizace. Takže například můžeme mít OS, který běží uvnitř jiného OS, takže vzniká mezivrstva několika dalších OS. Další vrstva, která tuto architekturu komplikuje, jsou různá dějová prostřední. Java Virtual Machine nebo .NET framework. Tato prostředí sice běží, jako běžná aplikace, ale samy produkují jiné aplikace. Tím pádem nám vzniká celá hierarchie vrstev. Toto schéma je tedy pouze ilustrační.

Tedy první úloha OS je abstrakce nad hardwarem, další úlohou je zajištění správy zdrojů.

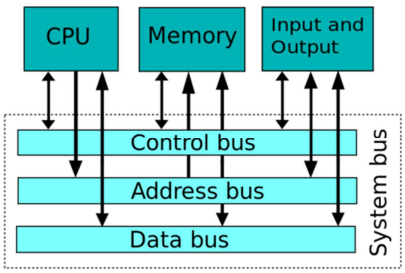
Počítač má v instrukci obrovské množství zdrojů (CPU čas, místo v paměti, místo na disku a další…) a úlohou OS je, aby efektivně rozděloval tyto prostředky.

CPU

Začneme od Adama – tedy od Johna von Neumanna

Von Neuman navrhl architekturu počítače, která se skládá ze tří, respektive čtyř základních kamenů

- CPU – centrální procesorová jednotka
- Paměť
- Vstupy a výstupy



Tyto zařízení spolu komunikují pomocí sběrnice (nebo sběrnic), pomocí kterých se přenáší data (případně adresy na paměť). Mezi sběrnicemi je jedna řídící sběrnice, která všechno řídí.

Základním rysem této VN architektury je, že paměť pro program i data je stejná – to umožňuje jednodušeji projektovat celou architekturu za cenu vyšších nákladů, na rozdíl od Harvardské architektury, která fyzicky odděluje paměť (paměť programu a dat) a spojovací obvody (spojovací obvody programu a dat).

Přestože je **CPU** uveden jako jeden čtvereček, tak se CPU skládá z více částí

- **ALU** Aritmetickologická jednotka, která se stará o výpočty
- **Řadič** (řídící jednotka) který se stará o koordinaci toho celého systému – řízení chodu systému

REGISTRY CPU

CPU má také sadu registrů, které slouží k uchovávání právě zpracovávaných dat, se kterými procesor pracuje. Dotazy se kterými procesor pracuje, jsou obvykle uchovávány v registrech, protože je **přístup k registrům značně rychlejší**, než přístup do paměti. Takže procesor má typicky sadu registrů, kde jsou uložena data, plus má sadu speciálních registrů, které slouží k řízení chodu samotného procesoru.

- **IP** (instuction pointer) – ukazuje na instrukci, která se má právě provádět
- **Program status word** (PSW, FLAGS - pokud došlo k přetečení, nastaví se příznak do registru informace o tom jak dopadla předchozí operace, kde se procesor nachází)
- **IR** (instruction register) - instrukce, která je právě prováděna
- **SP** (stack pointer) - ukazatel na zásobník, kde jsou uloženy výpočty, argumenty, atd…

Každý procesor se vyznačuje takzvanou instrukční sadou označovanou ISA. Je to sada instrukcí, které ovládají tok procesu. Tato sada instrukcí je typická pro daný procesor, případně pro celou rodinu procesorů. Tyto operace, které ovládají procesor jsou instrukce a operandy, které jsou zakódovány jako čísla. Těmto číslům se říká strojový kód. **Každá instrukce má 0 až 3 operandů – registr, konstanta nebo místo v paměti**. Nicméně pro naše porozumění se tyto instrukce zapisují v **jazyce symbolických adres**, někdy též vulgárně označovaných jako **Assembler**.

Výpočet faktoriálu: Intel x86

```
 0:   8b 4c 24 04             mov   ecx, DWORD PTR [esp+0x4]
 4:   b8 01 00 00 00          mov   eax, 0x1
 9:   83 f9 00                cmp   ecx, 0x0
 c:   0f 8e 0a 00 00 00       jle   1c <main+0x1c>
12:   f7 e9                   imul  ecx
14:   83 e9 01                sub   ecx, 0x1
17:   e9 ed ff ff ff          jmp   9 <main+0x9>
1c:   c3                      ret
```

Instrukce procesoru jsou zapsané jako sekvence bajtů. Každá instrukce má svůj význam, ten význam se dá zapsat pomocí assembleru. Na prvním místě je operace, kterou chceme provést, na dalších místech jsou operandy dané operace.

Například `mov ecx DWORD PTR [esp+0x4]` – načte do registru `ecx` hodnotu z paměti.

Když napíšete program v jazyce C, tak data projdou preprocesorem – překladač vygeneruje assembler a následně se generuje další kód, který je posléze proveditelný přímo procesorem.

JAK PROBÍHÁ VÝPOČET

Instrukce jsou zpracovávány v řadě za sebou, což vyplývá z definice Von Neumannovy architektury. Nicméně jedna instrukce není zpracována v jednom kroku. Z důvodu větší efektivity jsou instrukce zpracovávány v několika krocích. Ty kroky se liší v závislosti na implementaci konkrétního procesoru, V zásadě se dá identifikovat 4 až 5 základních kroků.

- **Fetch** načtení instrukce z paměti do procesoru
- **Decode** určení o kterou instrukci se jedná
- **Výpočet adres** operandů
- **Načtení dat** do procesoru
- **Execute** provedení instrukce
- **Write-back** zapsání výsledku na kýžené místo

Toto rozdělení do několika kroků umožňuje zvýšit efektivitu procesoru, díky tak zvanému **pipelingu**, kdy v jeden okamžik je rozpracováno několik instrukcí. Například procesory Pentium4 měli pipeline až 30 různých kroků, což už ale bylo příliš. Toto není jediný způsob, jak dnes procesory pracují.

Tím, že činnost procesoru máme rozdělenou do několika kroků, můžeme jednotky, které se starají o třeba načtení, dekódovaní dat, nebo výpočty, můžeme znásobit. V takovém případě nám vzniká tak zvaný **Superskalární procesor**. To však přináší některé problematické situace. Je potřeba zajistit správné pořadí operací - synchronizovat třeba přístupy do paměti. Navíc pipeling funguje perfektně, pokud se provádí jedna instrukce za druhou, v momentě, kdy je potřeba provést nějaké podmíněné skoky, musíme počkat, než víme, kterou další instrukci máme načíst. Současné procesory to řeší tak, že mimo jednu hromadu tranzistorů, které se starají o samotné výpočty, mají ještě druhou hromadu tranzistorů, které se snaží předvídat běh programu – jakým způsobem bude kód probíhat.

REGISTRY INTEL X86

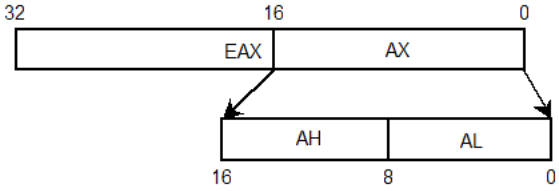
Bude nás zajímat 32b varianta insrukční sady Intel x86. 64b variantu ponecháme, jde pouze o nadmnožinu 32b varianty. Sada má několik set instrukcí – pro tento předmět nám ale postačí několik desítek.

Máme 32b registry, které se dělí do několika kategorií

Obecně použitelné registry

- **EAX** Akumulátor - střadač, který typicky slouží k ukládání aktuálně zpracovávaného čísla má nějaké speciální určení, třeba při násobení a dělení, nebo při vstupních a výstupních operacích
- **EBX** Base - přístup do paměti pomocí nepřímé adresace
- **ECX** Counter – využití při cyklech, posuvech a rotacích
- **EDX** Data - ukládání dat

Tyto registry můžeme využít k uchování jakékoli 32b hodnoty. Každý tento 32b registr se skládá z několika podčástí.



Lze přistupovat k celku EAX, nebo pak AX, AL a AH

Obdobně pro EBX, ECX i EDX

Následující registry se také mohou použít, ačkoli **mají své specifické určení**

- **EDI** adresa cíle - kam se data přenáší
- **ESI** adresa startu - odkud se data přednáší
- **EBP** adresace parametrů funkcí a lokálních proměnných

Další registry **se kterými se nedá jednoduše manipulovat** (pro manipulaci s nimi existují speciální instrukce)

- **ESP** ukazuje na vrchol zásobníku
- **EIP** ukazatel na aktuální místo programu - registr ukazující na instrukci za aktuálně prováděnou instrukci

A zásadní registr

- **EF** obsahuje příznaky nastavené právě proběhlou instrukcí – jsou zde ukládány informace o tom, jak proběhla předchozí instrukce

Tyto zmíněné registry mají také své 16b mladší bratříčky (AX). Ale již nemají 8b (AL a AH). Lze je pomocí těchto 16b podregistrů měnit, ale musí být opodstatněné a hlídané, protože může dojít k fatálním následkům.

Instrukce pro práci s registry jsou ve tvaru – název (zkratka typicky ze tří písmen), po názvu následují operandy, které mohou být – registr, místo v paměti, nebo konstanta (někdy též označována jako přímá hodnota). Pro seznámení se s jednotlivými instrukcemi choďte na cvičení a štětě skripta Assembler od Aleše Keprta.

Pozn.: Rotace nemá v jazyce C ekvivalent, na rozdíl od většiny jiných operací, protože na počítačích kde se Céčko vyvíjelo, nebyla tato Assemblerovská operace, tedy není ani její ekvivalent v jazyce C.

DATOVÉ TYPY JAZYKA C

Co se týče celočíselných typů, máme k dispozici pět typů. Jejich velikost se může lišit na základě použití počítače, překladače a jiné.

- `char` 8b = 1B
- `short` ≥ `char` 16b = 2B
- `int` ≥ `short` 32b = 4B (velikost slova)
- `long` ≥ `int` 32b pro 32b CPU / pro 64b CPU je velikost podle platformy
- `longlong` ≥ `long`

Int je nejpoužívanější datový typ, tedy odpovídá velikost jednoho slova, aby s ním procesor mohl pracovat co nejefektivněji. Tak i v době 64b procesorů má int 32b.
Long má na 64b Windows velikost 32b a na 64b Linuxu má 64b.

Každý z těchto vyjmenovaných datových typů má signed (znaménkový) a unsigned (neznaménkový) hodnoty.

- `signed char` -128 … 127
- `unsigned char` 0 … 255
- `unsigned short` 0 … 665235
- `signed short` -32768 … 32767

Navíc používáme ještě jeden typ signed_t, tento typ typicky odpovídá adresovatelné paměti. Tedy máme-li 32b CPU, je tento typ 32b, při použití 64b CPU, je tento typ 64b.

Pokud chceme mít jistotu, že daný typ má danou velikost, můžeme použít knihovnu stdin.h, která deklaruje typy v daných velikostech.

BITOVÉ OPERACE

Při nízko úrovňovém programování je mnohdy potřeba pracovat na úrovni jednotlivých bitů a proto máme operátory: bitový posun doleva, bitový posun doprava, logický součin, logický součet, negace a Exkluzivní součin – XOR. Pozor ačkoli jde o podobné názvy, nejde o stejné operace známé v jazyce C – porovnává se bit po bitu!

Cv: *Přijděte na přesné rozdíly mezi bitovými operacemi a operacemi dobře známých z jazyka C.*
Dále je dobré znát použití pointerů, referenci, dereferenci, a vše potřebné …
Zopakujte si také struktury.

PAMĚŤ A ORGANIZACE PAMĚTI POČÍTAČE

Představme si paměť jako lineární datovou strukturu. Ačkoli je paměť ve skutečnosti realizována poněkud složitěji, pro tentokrát nám bude stačit tato představa.

V podstatě rozlišujeme dva způsoby přístupu do paměti. Obecně můžeme přistupovat do paměti náhodným způsobem. Adresy můžeme vybírat dvěma způsoby.

Buď pevně odkazujeme na dané místo v paměti, ale také můžeme adresovat paměť nepřímo, tedy se místo v paměti vypočítá z hodnot, které jsou uloženy v registrech. Tento výpočet se provádí podle následujícího vzorce skládajícího se ze čtyř částí.

Adresa = posunutí + báze + index x faktor

Posunutí představuje nějakou konstantu
Báze a *index* jsou registry
Faktor je číslo 1, 2, 4 nebo 8 (počet Bajtů)

Když chceme nepřímo přistupovat do paměti, vždy adresujeme pomocí tohoto vzorce, kde za určitých okolností můžeme některé části vypustit.

V Assembleru se pak čtení, nebo zápis do paměti zapisuje ve tvaru

```
velikost PTR […]
```

Kde velikost může být buď `BAJT`, `WORD`, nebo `DWORD` – tedy přičtení 1B, 2B, nebo 4B. V instrukcích to pak vypadá:

```
mov DWORD PTR [eax], ebx
add ax, WORD PTR [ebx + esi * 2 + 10]
```

Část *velikost PTR* můžeme v některých případech vypustit. A to tehdy, když je velikost odvoditelná z velikosti registrů. Tedy místo `mov DWORD PTR [eax], ebx` můžeme psát `mov eax, ebx`.

Jde o kratší zápis, ale vystavujeme se tak riziku, že uděláme nějaký přehmat. Například když do registru `esi` chceme uložit hodnotu 42. Tedy nejsme schopni odvodit, o jaké velikosti chceme danou hodnotu zapsat, jestli 1B, 2B, 4B. V praxi se tedy doporučuje tuto konvenci používat. Je to praktické, protože je toto častá příčina chyb, které jsou špatně odhalitelné.

Když použijeme `mov eax, a`, kde a je proměnná jazyka C, překladač toto ve skutečnosti přeloží na `mov eax, DWORD PTR [ebp - n].`

Procesor i jazyk C umožňuje práci s 32b adresami. Ve skutečnosti, ale procesory i386 používají 48b adresy. Funguje to tak, že jednotlivé instrukce mají rozdělený program na jednotlivé segmenty. Jeden segment je vyčleněn pro zásobník, jeden pro kód, další pro data. Tyto segmenty se volají pomocí 16 bitů a jsou víceméně zvolené implicitně.

Pozn.: *Instrukce* `movsx` *slouží k nahrání menší hodnoty do větší – nevyužité místo v novém registru vyplní nulami, obdobně instrukce* `movzx` *pro neznaménkové hodnoty*

VZTAH ADRESACE PAMĚTI K JEDNOTLIVÝM KONSTRUKTŮM JAZYKA C

Přístup do paměti odpovídá dereferenci v jazyce C

```
mov eax, DWORD PTR [ebx]      // eax := *ebx
```

Vztah mezi pointerovou aritmetikou a polem – vycházíme z toho, že i když máme pole v jazyce C, tak ho můžeme reprezentovat jako pointer na první prvek tohoto pole.

```
short a[] = malloc(sizeof(short) * 10);
_asm{
1)      mov ebx, a
2)      mov ax, [ebx + esi * 2] // ax := a[esi]
}
```

1) Do `ebx` si uložíme pole, tedy odkaz na nějaké místo v paměti.
2) Zde můžeme vidět použití vzorce pro adresaci paměti, kde `esi` je index a 2 je faktor. Najdeme si adresu pole a posuneme se na adresu, kam nám ukazuje index krát velikost daných prvků.

Jak používat struktury

```
struct foo { int x;
             int y;
             int z[10]; };
struct foo * a = malloc(sizeof(struct foo));
_asm{
      mov ebx, a
      mov [ebx], ecx               // a->x := ecx
      mov [ebx + 4], ecx           // a->y := ecx
      mov [ebx + esi * 4 + 8], ecx // a->z[esi] := ecx
}
```

Poslední řádek kódu Asembleru: Vypočítáme si bázi, uděláme posun o 8b, protože tuto část paměti zabírají hodnoty x a y ze struktury a poté uděláme posun na daný index `esi`.

Nyní získáváme lepší představu o tom, proč je daný vzorec na adresaci tak složitý. Navíc oproti pohybu v poli se ve struktuře můžeme pohybovat vždy o jinou velikost, protože struktura může obsahovat char, short, int i long či dokonce longlong zároveň.

ZAROVNÁNÍ HODNOT V PAMĚTI

Pokud máme nějakou strukturu, pak je tato struktura alokovaný úsek paměti. V případě předchozí struktury `foo` jde o úsek 12 x 4B = x jako int (4B), y jako int (4B) a pak pole deseti int-ů (10 x 4B). Ve skutečnosti to ale ne vždy musí být takto přímočaré. Protože kvůli zvýšení efektivity se používá zarovnání hodnot v paměti. Říkáme pak, že adresa paměti je zarovnaná na n bajtů, pokud je paměť násobkem čísla n.

Zarovnání vzniklo, protože procesor čte z paměti celé slovo, tedy 32b, a tedy je výhodné, aby četl data, která jsou zarovnána na 32b.

Tedy struktura

```
struct foo { char a;
             int b;
             char c;
             short d; };
```

Se zarovná takto:

a	0	0	0	b	b	b	b	c	0	d	d

Překladač, jak jde vidět, tedy vkládá mezery, aby bylo zarovnání na 32b = 4B. Při tomto nákresu můžeme vidět, že lze ušetřit paměť vhodným seskládáním hodnot.

V případě struktur a čísel s plovoucí desetinnou čárkou jsou zarovnávány na násobky 8B. Celé toto zarovnání má v důsledku to, že short je zarovnáván na násobky 2B, int na násobky 4B atd, tedy vždy se můžeme pohybovat po velikosti daného datového typu.

Toto chování překladače lze sice potlačit, ale může to zpomalit čtení dat, nebo skončit systémovou chybou.

ENDIANITA - UKLÁDÁNÍ VÍCEBAJTOVÝCH HODNOT

Endianita je specifická pro každý procesor. Za normálních okolností, pokud pracujeme na jednom procesoru, tak nemáme problém. Jakmile chceme navrhovat datové formáty a protokoly přenositelné mezi různými počítači je potřeba brát ukládání více bajtových hodnot brát v potaz.

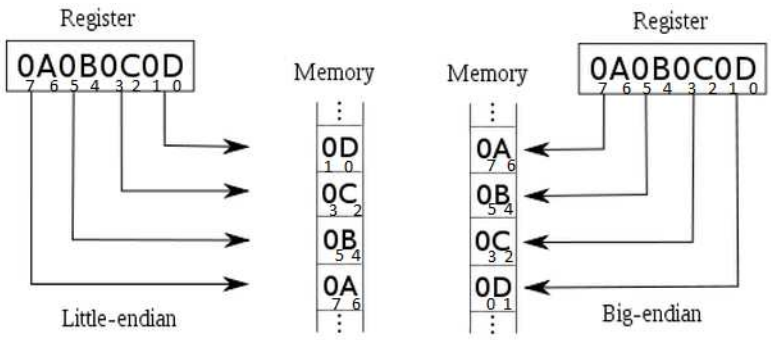
Existují dva, postupy

Little endian – hodnoty jsou zapisovány od nejméně významného bitu

Big endian – hodnoty jsou zapisovány od nevýznamnějšího bitu

Bi-endian – lze přepínat

Dodnes se vedou spory o to, který způsob zápisu je lepší.



REPREZENTACE HODNOT

Znaménkové a neznaménkové hodnoty reprezentujeme pomocí **doplňkového kódu**, kde zápornou hodnotu dostaneme tak, že provedeme inverzi bitů a přičteme 1. Výhodou doplňkového kódu je snadná manipulace.

Ve chvíli, kdy se hodnota nevejde do rozsahu daného typu, dojde k přetečení, nebo podtečení.

```
char a = 127 + 1              // = -128   // přetečení
unsigned char c = 255 + 1     // = 0      // přetečení
char b = -10 – 120            // = 126    // podtečení
```

Jinou možností reprezentace je **BCD kód** (Binary Coded Decimal), kde se ukládají čísla v desítkové soustavě a máme 4b na cifru. I přes to, že je tato možnost výhodná u přístupu k hardwaru, je tato možnost také nepraktická kvůli málo využitým bitům.

Úkol: Nastudovat BCD ze skript Operační systémy

PŘÍZNAKY

Využívají se třeba ve chvíli, kdy bychom například u sčítání prováděli přenos do vyššího řádu, ale již nemáme kam vyšší řád uložit, kdy se ten daný jeden bit, který bychom do vyššího řádu zapsali, už nevejde do reprezentace.

Příznaky jsou uloženy v registru `EF` (EFlags). Tyto „vlajky“ nastavují jednotlivé operace.

Příznaky pro řízení výpočtu jsou:
- **SF** Sign Flag (podle znaménka – odpovídá nejvyššímu bitu výsledku)
- **ZF** Zero Flag (výsledek byla nula)
- **CF** Carry Flag (přenos)
- **OF** Overflow Flag (přetečení znaménkové hodnoty mimo daný rozsah)

Další příznaky:
- **AF** Auxiliary carry Flag (přenos ze čtvrtého do pátého bitu – využití u BCD)
- **PF** Parity Flag (1 = sudá parita)

Řídící příznaky:
- **TF** Trap Flag (slouží ke krokování)
- **DF** Direction Flag (ovlivňuje chování instrukcí blokového přesunu)
- **IOPL** I/O Privilege Level (úroveň oprávnění – může být nastaven pouze jádrem OS)
- **IF** Interrupt enable Flag (možnost jak zablokovat některá přerušení – pouze jádrem OS)

BĚH PROGRAMU A SKOKY NA INTEL X86

Program za normálních okolností provádí jednu instrukci za druhou. Můžeme však říct programu, ať začne vykonávat instrukci na nějaké jiné adrese. Těmto příkazům říkáme skoky.

NEPODMÍNĚNÉ SKOKY

Nepodmíněný skok `jmp`, kde argumentem může být registr, místo v paměti, nebo nějaká konstantní hodnota. S těmito nepodmíněnými skoky můžeme implementovat tak akorát nekonečné smyčky. Možná bychom mohli čekat, že bude Assembler obsahovat ekvivalent pro operaci `if`, ale není tomu tak. Obsahuje místo toho podmíněné skoky, což jsou instrukce, které provedou skok na místo v programu, pokud jsou nastaveny příslušné příznaky.

PODMÍNĚNÉ SKOKY

Například `jz` provede skok, pokud výsledek předchozí operace byl nula. (Obdobně `jnz`, `js`, `jns`, …)

K porovnání čísel se používá jednoduchá úvaha. Pokud od sebe odečteme dvě hodnoty, tak podle výsledku jsme schopni určit, jestli je první prvek menší, větší, nebo roven druhému.

SROVNÁVÁNÍ

K porovnání je zavedena operace `cmp`, která se chová podobně jako operace `sub` (odečítání čísel) s tím navíc, že **nastaví příznaky** CF, ZF, OF, SF a neprovádí následné přiřazení výsledku odečtení.

Nyní nám již může být předtím uvedený kód na výpočet faktoriálu o něco jasnější.

```
 0:   8b 4c 24 04            mov   ecx, DWORD PTR [esp+0x4]
 4:   b8 01 00 00 00         mov   eax, 0x1
 9:   83 f9 00               cmp   ecx, 0x0
 c:   0f 8e 0a 00 00 00      jle   1c <main+0x1c>
12:   f7 e9                 imul  ecx
14:   83 e9 01              sub   ecx, 0x1
17:   e9 ed ff ff ff        jmp   9 <main+0x9>
1c:   c3                    ret
```

V praxi pak používáme skoky ve tvaru: *`instrukce návěstí`*, kde instrukce je `jmp`, `jle`, … a návěstí je místo v programu označené námi zvoleným slovem třeba „navesti“ ve tvaru: `navesti:` a za ním je sekvence kódu, který se má po skoku provést.

CYKLY

Pro snadnější implementaci cyklů byly zavedeny také **smyčky**.

Operace `jecxz` a `jcxz` automaticky testuje registr ECX, jestli je ECX nulový, operace provede skok. Obdobně operace `loop`, která navíc odečítá při každém cyklu jedničku od registru ECX a dokud ECX není nulový, tak skáče.

Bohužel podmíněné skoky zpomalují běh programu. Je to z důvodu pipelingu. Mezitím, co běží několik vláken pipeline se provede jeden skok a následně se čeká, až všechny vlákna pipeline doběhnou.

Procesory implementují různé metody pro odhad, jestli daný skok bude, nebo nebude proveden.
Statický přístup (například u skoku zpět předpokládá, že bude proveden)
Dynamický přístup (rozhoduje na základě historie skoků)
Nápověda poskytnutá programátorem (nastavený příznak v kódu)

ODHAD SKOKŮ

Nejjednodušší varianta předpovědi jestli se daný skok provede, nebo ne, je jednoduché 4 stavové počítadlo se saturací, kde máme 4 stavy, na které se můžeme dívat jako na binární hodnoty představující moýnosti

- **11** veliká pravděpodobnost, že skok bude provedený
- **10** je šance, že skok bude provedený
- **01** je šance, že skok nebude provedený
- **00** veliká pravděpodobnost, že skok nebude provedený

Procesor až na stav 00 předpokládá, že v ostatních stavech bude skok provedený. Na základě toho, jestli skok byl, nebo nebyl reálně proveden, se hodnota počitadla posouvá. Posune se dolů, když skok provedený nebyl, nebo směrem nahoru, když skok byl proveden.

Počet takovýchto počitadel v procesoru se liší procesor od procesoru. Každý výrobce se snaží přijít na způsob, jak nejlépe předvídat.

V případě, že se skok opakuje, tedy jednou se provede a jednou ne, zacyklíme se mezi stavem 10 a 01. Tím pádem dochází ke špatnému odhadu. Proto řada procesorů implementuje dvojúrovňovou adaptivní metodu předpovědi, kdy procesor si ukládá historii skoků na dané adrese a pro každou posloupnost těchto skoků má vlastní prediktor založený na principu toho daného skoku. Na základě tohoto způsobu odhadu lze předvídat také sofistikovanější vzory chování, kde byl třeba skok jednou provedený, pak dvakrát ne, pak zas jednou ano …

Jednoduchý prediktor má přesnost kolem 90%.
Prediktor dvojúrovňový má přesnost 97%.

Další problém související s odhadem je, jestli je dobré zaznamenávat si historii skoku pro jednu instrukci, nebo ne. Ukazuje se, že pokud máme lokální tabulku skoku, tak je přesnost lepší, na rozdíl od globální tabulky, kam zaznamenáváme každý skok, avšak výsledky nejsou o moc horší.

Pokud si ale ukládáme historii pro každou instrukci, tak je přesnost 97%
Pokud jednu historii bez ohledu na to o jakou instrukci se jedná, tak je přesnost kolem 95% s tím, že navíc výrazně zjednodušuje nároky na procesor.

JAK VOLAT FUNKCI / PODPORGRAM

Předtím, než se dostaneme k samotnému volání funkcí, musíme si uvést zásadní část procesoru, kterou je **zásobník**.

ZÁSOBNÍK

Většina procesorů má vyčleněný úsek pro zásobník. Má strukturu typu LIFO. Ukládají se zde mezivýpočty, návratové hodnoty, lokální proměnné, a jiné … Dnešní programovací jazyky neumožňují přímou manipulaci s tímto zásobníkem, přesto má zásadní úlohu pro běh programu. Procesory obecně mají řadu zásobníků. Na procesorech i386 je jeden společný zásobník, který je umístěn někde v paměti. Kde, nás nemusí zajímat. Tento zásobník roste od nejvyšších adres směrem dolů k nižším.

Pro manipulaci s tímto zásobníkem slouží registr **ESP**, který vždy odkazuje na vrchol zásobníku. Tedy, kdybychom měli instrukci `mov eax, esp`, tak do `eax` nahrajeme první 4B hodnotu na zásobníku. Se zásobníkem se pracuje pomocí dvou operací

- **push** umístí hodnotu na zásobník (odečte od ESP danou velikost a nahraje hodnotu)
- **pop** odebere ze zásobníku hodnotu (načte hodnotu z vrcholu zásobníku a přičte k ESP danou velikost)

Je potřeba si uvědomit směr posouvání vrcholu zásobníku.
Z implementačních důvodů musí mít registr ESP hodnotu násobku 4B, tedy kdybychom nahráli 1B hodnotu, musíme se stejně posunout o 4.

K **volání podprogramu** využíváme operaci `call`. Tato operace na zásobník uloží aktuální místo, kde se v programu nacházíme (uloží hodnotu registru EIP), a provede skok do těla funkce. Toto uložení místa slouží k tomu, abychom po dokončení volané funkce věděli, kam se máme vrátit. K návratu se používá operace `ret` (umístěna na konci volané funkce), která odebere hodnotu ze zásobníku a provede skok na adresu danou touto odebranou hodnotou.

S použitím zásobníku můžeme implementovat rekurzivní funkce.

Pozn.: Samotná operace `push eip` *neexistuje.*

KONVENCE VOLÁNÍ FUNKCÍ

Chceme-li mít funkce ekvivalentní s voláním v jazyce C nebo Pascal, musíme zařídit předávání parametrů, musíme mít vyřešené vytváření lokálních proměnných, musíme se postarat o samotné provedení funkce a také potřebuje vyřešit návrat z funkce a předání výsledku.

K tomuto problému existuje několik přístupů. Na platformě x86 tyto kroky nejsou zadrátovány do architektury procesoru. Tedy závisí na libovůli programátora, jak bude tyto problémy řešit. Vesměs jde o domluvené konvence, které se snažíme obecně dodržovat, když ale programátor trvá na svém, případně má opodstatněný důvod, použije své vlastní konvence.

Tyto konvence jsou často specifické pro překladač, nebo jsou součástí specifikace ABI (Application Binary Interface) operačního systému. Programy zkompilované pomocí různých překladačů budou dodržovat stejné binární konvence pro předávání dat a tedy spolu budou moci spolupracovat.

X86 OBECNĚ

První způsob je, že argumenty budeme předávat pomocí domluvených registrů a případné zbylé hodnoty budeme předávat pomocí zásobníku, nebo rovnou všechny argumenty budeme ukládat přímo na zásobník. Po zavolání funkce si funkce přečte hodnoty ze zásobníku, případně z registrů.

Tedy

1) uložíme hodnoty jednotlivých argumentů funkce na zásobník (případně do registrů)
2) zavoláme funkci
3) funkce si přečte hodnoty ze zásobníku (případně z registrů)

Vzniká zde problém, že když si uložíme hodnoty na zásobník, tak po návratu máme na zásobníku hodnoty. Nastává otázka kdo se má starat o **vyčištění zásobníku**. Jestli se má o to starat volaná funkce sama, nebo jestli se má o to start ten, kdo funkci zavolal. Neexistuje vesměs správná odpověď. Historicky se vytříbily tři způsoby předávání argumentů a odstraňování hodnot po návratu z funkce.

KONVENCE JAZYKA C – CDECL

Argumenty funkcí jsou předávány pouze přes zásobník zprava doleva. Argumenty ze zásobníku odstraňuje volající. Výhodou této konvence je, že umožňuje implementovat funkce s proměnlivým počtem parametrů.

KONVENCE JAZYKA PASCAL – PASCAL

Argumenty se předávají opět čistě přes zásobník s tím ale, že se předávají zleva doprava. Je to intuitivnější z pohledu programátora, ale neumožňuje funkce s proměnlivým počtem parametrů. Navíc hodnoty ze zásobníku odstraňuje volaný, tedy daná funkce.

KONVENCE FASTCALL – FASTCALL / MSFASTCALL

Tato konvence je založena na tom, že první dva parametry jsou předávány pomocí registrů ECX a EDX, zbylé argumenty jsou naskládány na zásobník zprava doleva jako u konvence cdecl. Hodnoty ze zásobníku odstraňuje volaná funkce. Tato konvence má tu výhodu, že část hodnot se předává přes registry a tak se zbytečně nemusí manipulovat se zásobníkem. Pokud tedy máme funkce s malým množstvím parametrů, tak by volání funkcí pomocí konvence fastcall mělo být rychlejší.

Pozn.: Uvědomte si, že operace push, ačkoli se tváří jako jedna instrukce je složena z více instrukcí, a to z posunutí vrcholu zásobníku a nahrání hodnoty na vrchol.

PŘEDÁVÁNÍ NÁVRATOVÝCH HODNOT

Teoreticky by bylo možné předávat návratové hodnoty přes zásobník, což je ale velmi pomalé. Proto většina konvencí dodržuje, že návratová hodnota se předává v registru EAX, případně ve dvojregistru EDX:EAX.

V InlineAssembleru si tedy nemusíme návratovou hodnotu složitě předávat přes lokální proměnou, ale můžeme spoléhat na to, že se uloží do registru EAX.

Když se hodnota, kterou chceme předat nevejde do registru, nebo do jednoho pole do zásobníku, tak se hodnota proměnné uloží jinam a předáváme pouze odkaz na hodnotu.

RÁMEC FUNKCE (STACK FRAME)

Tento rámec se vytváří na zásobníku při volání funkce. Představuje úsek na zásobníku, který obsahuje předané argumenty dané funkci, adresu návratu případně lokální proměnné – respektive prostor pro lokální proměnné pro lokální funkce. Pro přístup k tomuto rámci se většinou používá registr **EBP**.

PŘÍKLAD RÁMCE V KONVENCI CDECL

Volání funkce:

1) uložíme na zásobník parametry funkce zprava doleva
```
push 3
push 2
push 1
```
2) zavoláme funkci
```
call <adresa>      // adresa kde se funkce nachází
```
tato instrukce způsobí to, že na zásobníku máme uloženou návratovou adresu (ne hodnotu)
3) nyní funkce uloží obsah registru EBP na zásobník, tedy uloží na zásobník odkaz na předchozí rámec se kterým volající funkce pracovala
4) funkce uloží do registru EBP obsah ESP (začátek nového rámce) – pro krok 2) při návratu
5) funkce si na zásobníku vytvoří místo pro lokální proměnné (posunutí ESP o nějaký násobek 4)
6) následně může následovat několik operací push, pomocí kterých uložíme hodnoty registrů, se kterými budeme pracovat, na zásobník

Pozn.: 4) Ačkoli v průběhu volané funkce pracujeme se zásobníkem, díky EBP můžeme adresovat jednotlivé argumenty funkce s tím, že si odpočítáme, kde by se daný argument měl nacházet.

Nyní provedeme samotnou funkci a započneme **návrat z funkce:**

1) obnovíme hodnoty v registrech ze zásobníku
2) odstraníme lokální proměnné tím, že do ESP nahrajeme EBP (vrátíme zásobník, jak byl)
3) obnovíme hodnotu EBP, tedy provedeme pop EBP
4) provedeme návrat pomocí instrukce `ret` (na zásobníku je adresa návratu díky instrukci `call`)
5) nyní jsme již v původní volající funkci a pouze odstraníme argumenty, které jsme ukládali před voláním na zásobník – posunutím vrcholu zásobníku (přičtení k ESP)

Když si uvědomíme strukturu zásobníku, tak víme, že na adrese, která je uložena v registru EBP se nachází původní hodnota registru EBP, což je celkem nezajímavá věc.
Na EBP + 4 se nachází návratová adresa, což v tomto případě nás taky nemusí zajímat.
Ale **první argument se nachází na adrese EBP + 8**, druhý argument na EBP + 12 a tak dále …
Analogicky lze od EBP odvíjet lokální proměnné: na EBP – 4 se nachází první lokální proměnná a tak dále …

To by v případě konvence pascal, kde se hodnoty ukládají na zásobník opačně, způsobilo paseku, protože bychom vlastně nevěděli, kolik argumentů jsme dané funkci poslali. Takže když máme konvenci cdecl, tak z toho, že známe nějaké první argumenty dané funkce, můžeme odvodit počet argumentů celkem.

Jestliže před zavoláním funkce máme v registrech něco užitečného, je potřeba ošetřit uchování hodnoty v registru. V tomto ohledu jsou dva typy registrů:

Callee-saved, kde se o uchování hodnot stará volaný – EBX, ESI, EDI
Caller-saved, kde se o uchování hodnot stará volající – EAX, ECX, EDX

Je potřeba myslet na to, že jiná funkce může registry EAX, ECX a EDX naprosto přepsat a mohou tak obsahovat naprosto cokoliv.

Pozn.: Povšimněte si, že registry EAX, ECX a EDX jsou ty registry, které využívají instrukce pro dělení a jiné.

PŘERUŠENÍ

Mechanismus přerušení je systém, který umožňuje počítači asynchronně reagovat na události způsobené vnějším zařízením. Představte si situaci, že byste chtěli mít v programu načítání z klávesnice, mít síťové rozhraní. Jedna z možností jak tuto situaci řešit je hlídat si konkrétní port daného procesoru, jestli se tam neobjeví signál, že někdo stiskl danou klávesu, že z hard disku byly načteny data, nebo že přišel síťový packet.

Z praktického pohledu by byl takovýto přístup velmi neefektivní. Procesor by totiž strávil většinu času čekáním na nějakou externí událost a neřešil by svůj vlastní výpočet.

RUTINA - OBSLUHA PŘERUŠENÍ

Proto byl zaveden mechanismus, že vnější zařízení může způsobit přerušení činnost procesoru a tím si vynutit jeho pozornost. V praxi to funguje tak, že máme nějaké HW zařízení, které když potřebuje pracovat s CPU, tak mu pošle signál IRQ (Interrupt Request) a po tomto signálu se spustí zvláštní přednastavená procedura, které se říká obsluha přerušení. Může jít o stisk klávesy, příchod síťového packetu, informace od pevného disku, že jsou nachystána data atd. …

Tedy přijde přerušení od daného HW zařízení, procesor přeruší právě prováděnou instrukci a spustí se obsluha přerušení.

Po ukončení obsluhy přerušení se procesor vrátí zpět do stavu, kde byl přerušen.

Obsluhy přerušení jsou podobné běžným funkcím. Rozdíl je v tom, že procesor musí vědět, kde je která daná rutina uložena. Za tímto účelem má procesor i386 vyčleněný úsek paměti, který obsahuje 256 různých adres obslužných rutin a na základě čísla, které přijde z hardwaru, se vybere daná obsluha přerušení, která se následně provede.

Protože obsluha přerušení je opět prováděný kód, tak v průběhu řešení jednoho přerušení může procesor přerušit provádění jedné rutiny a zažít jinou. Některé systémy umožňují rutinu přerušit. Jiné systémy toto přerušení rutiny nedovolují a v takovém případě se jednotlivá přerušení řadí za sebe do doby, než obsluha prvního přerušení skončí. K tomu slouží řadič přerušení.

V systémech Windows mají jednotlivá přerušení nastaveny priority. A tedy přerušení s nižší prioritou nemůže přerušit rutinu s vyšší prioritou a naopak, vyšší priorita přeruší nižší. Toto řešení je praktické a odpovídá intuitivnímu chápání, že některé přerušení je důležitější, než jiné. Například práce se zvukem a práce s pevným diskem. Zvuk si žádá souvislou stopu a přerušení zvuku byť jen v milisekundách může vadit, tedy můžeme předpokládat, že práce se zvukem by měla mít vyšší prioritu, než práce s pevným diskem.

Obecně rozlišujeme přerušení blokovatelná a neblokovatelná. Podle toho o jaké přerušení jde, se vybere, které se uloží do řadiče přerušení.

OBSLUHA PŘERUŠENÍ NA X86

Právě na procesorech x86, kde nalezneme 256 adres rutin z čehož prvních 32 přerušení je vyčleněno na přerušení generované procesorem a jiné speciální případy. Adresy rutin jsou uloženy v paměti na místě, kterému říkáme IDT (Interupt Description Table), která je uložena v registru IDTR.

Když dojde k přerušení, tak procesor vyhledá danou rutinu, uloží na zásobník adresu, kde se program aktuálně nachází (kombinace registru CS + EIP, kde CS je kódový segment, kde je uložený kód a EIP je adresa aktuálně prováděné operace) + se na zásobník uloží EFLAGS. A začne se rutina provádět. Navíc si každá rutina ukládá na zásobník registry, se kterými pracuje tak, aby program, který byl přerušený nepoznal, že byl přerušen.

Po skončení obsluhy přerušení se procesor vrátí do původního stavu pomocí instrukce IRET, která provede obnovu registrů.

APLIKACE PŘERUŠENÍ

Mechanismus přerušení má celou řadu aplikací. Jedna z nich je **ošetření** dělení nulou, pokus o provedení neplatné operace, přístup do neplatné paměti, kde program nemá oprávnění a jiné.

Zároveň ho lze použít k **debuggování**, nebo dokonce k implementaci **multitaskingu**.

Nebo explicitní vyvolání přerušení INT – **systémové volání**

Pozn.: IRQ0 je Timer, který je schopný vyvolat pravidelné přerušení - multitasking

I/O ZAŘÍZENÍ

Procesor podporuje IN a OUT instrukce, které dokážou na daný port procesoru zapsat, nebo přečíst hodnotu. Problém je, že se dá zapisovat pouze po slovech velikosti 32b. Není to ono, jde to, ale není to ideální. Kdybychom chtěli číst větší objemy dat, nebylo by to efektivní. Proto procesory zavádí DMA (Direct Memory Access) – přímý přístup do paměti. Tento přímý přístup do paměti funguje tak, že procesor předá požadavek řadiči přístupu do paměti a řekne mu: „Potřebuju načíst tato a tato data a chci je uložit na daném místě v paměti“. Procesor takto předá požadavek řadiči, který se dál stará o přenos dat a procesor se tak uvolní na výpočty, které provádí dál.

Data pak přímo do paměti nezapisuje procesor ani řadič, ale disk je zapisuje přímo do paměti. V momentě, kdy jsou data přečtena a zapsána v paměti, tak řadič disku oznámí, že data jsou na místě. Oznámí to tedy řadiči přímého přístupu do paměti, který vyvolá přerušení a oznámí procesoru, že data, která chtěl z disku přečíst na konkrétní místo v paměti, že už tam jsou a může tedy s nimi pracovat.

Bez tohoto systému přístupu by bylo čtení dat zoufale pomalé.

Současná implementace je ale výrazně složitější, protože různá zařízení můžou mít své vlastní řadiče přímého přístupu do paměti a pak je potřeba, aby se mezi sebou domluvili, které zařízení pracuje aktuálně jako řadič a tak dál.

REŽIMY PRÁCE PROCESORU

Od operačního systému očekáváme, že bude schopen nějakým způsobem spravovat prostředky, které má počítač k dispozici, tedy očekáváme nějakou správu sdílení procesorového času tak, možnost spouštět více programů současně, správu paměti tak, aby procesy, respektive běžící programy byly v paměti odděleny a nemohly se navzájem ovlivňovat. Na druhou stranu očekáváme, že OS mezi procesy zajistí možnost komunikace, spolupráce běžících programů. Zároveň očekáváme správu zařízení a nějakou organizaci dat. Typicky ukládání dat na nějaký souborový systém na disku a implementaci síťových rozhraní.

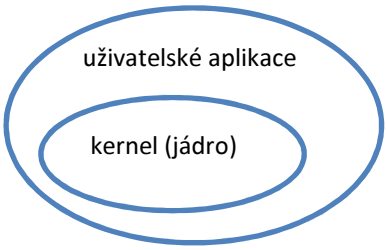
JÁDRO OS

Není žádoucí, aby si každý program tuto funkcionalitu implementoval každý program po svém. S tím souvisí také fakt, že nechceme, aby každý proces, nebo běžící program měl přístup ke všem možnostem HW. Nebylo by totiž dobré, kdyby si jeden program definoval vlastní obsluhu přerušení pro pevný disk a jiný by mu to změnil, vedlo by to k solidnímu guláši.

Proto se zavádí jádro OS, aby všechna tato funkcionalita byla sdílena na jednom místě a tím byla zajištěna konzistence systému. Aby si třeba jeden program ukládal na disk jedním způsobem a jiný program způsobem jiným. Proto existují dva režimy práce.

PRIVILEGOVANÝ REŽIM (KERNEL MODE)

V tomto režimu běží právě jádro OS a dovoluje provádět naprosto vše.



NEPRIVILEGOVANÝ REŽIM (USER MODE)

V něm běží jednotlivé procesy a aplikace. Funkce procesoru jsou v tomto režimu omezené. Když chce proces provsést některou z činností (přístup do paměti, přístup k organizaci dat, …), je potřeba přepnou procesor z jednoho režimu do druhého. To se děje pomocí tzv. systémových volání. Přepnutí procesoru z uživatelského do jaderného prostoru lze dosáhnout několika způsoby.

> Dojde k **výjimce** například dělení nulou, provedení neplatné operace, …
> Při **přerušení** ať už jde o přerušení vyvolané HW, ale lze jej vyvolat také pomocí SW
> Různá **systémová volání**

Důsledkem systémových volání je jednoznačně oddělit komunikaci těch běžících procesů s jádrem OS. Je zde definováno rozhraní, jak tato systémová volání fungují. Toto rozhraní je podobné konvencím s voláním funkcí. Rozdíl je v tom, že nepoužíváme operaci call, ale operace jiné.

Jeden z nejčastějších způsobů jak implementovat systémové volání tzn. Předat nějaké hodnoty jádru, přepnout procesor do jaderného režimu můžeme jednak pomocí SW přerušení, kdy OS má nadefinované nějaké číslo přerušení, které obsluhuje systémová volání. (V Linux č. 80, ve Win 0x2e).

Systémové volání probíhá tak, že je zvolen nějaký registr, typicky registr EAX (na i384), který udává číslo požadavku (chceme otevřít soubor, chceme zapisovat do souboru, nebo jiné…).

Do dalších registrů se uloží předávané argumenty (cesta do souboru, oprávnění, režim otevření souboru, …)

Následně je vyvoláno příslušné SW přerušení. V tento moment dojde k přepnutí do jaderného režimu. OS si tedy může dělat, co chce, takže může vyřídit daný požadavek.

Po vyřízení požadavku se vrací zpátky pomocí `iret` do původního stavu.

Protože chceme, aby přepínání z jednoho režimu do druhého bylo co nejefektivnější, tak řada procesorů zavádí specializované operace pro přepnutí režimu. Na architektuře x86 jde o speciální instrukce SYSENTER / SYSCALL a SYSEXIT / SYSRET.

VOLACÍ BRÁNY (CALL GATES)

Jde o variantu, jak přepnout procesor z jednoho režimu do druhého. Jde o specifikum procesorů x86.

Zavolá se specifická funkce (adresa), která se postará o přechod z jednoho režimu do druhého. Tento mechanismus je celkem složitý, je spojen se segmentací a využívaly je dříve Windows NT.

Tento mechanismus přepínání má jednu výhodu. Nemusíme se omezovat pouze na dva režimy (Privilegovaný a Neprivilegovaný), ale můžeme mít režimů více.

Na x86 jsou to režimy 4, které se označují jako ring 0 až ring 3.
ring 0 – privilegovaný mód, ve kterém běží jádro
ring 3 – neprivilegovaný mód, ve kterém běží uživatelské procesy.
Většina OS si vystačí s touto kombinací, ale může to být i jinak.

Do režimů **ring 1** a **ring 2** můžeme přiřadit procesy, u kterých chceme část omezení, ale také některé z privilegií. Proto třeba v ring 1 běží ovladače.

Další způsob, kde se tohoto dělení používá, jsou systémy založené na virtualizaci. V tom případě v režimu ring 0 běží nějaký revizor, v ring 1 běží jednotlivá jádra OS a v ring 3 běží uživatelské programy. Ale většina soudobých systémů si vystačí s ring 0 a ring 3. S virtualizací je to přeci jen o něco složitější.

Procesory z rodiny x86 mají z historických důvodů schopnost běžet v několika různých režimech.
Chráněný mód (protective mode) obsahuje právě zmíněnou funkcionalitu.
Reálný mód je starší řešení, kde nelze oddělit jádro a aplikace.

MS-DOS

Poskytoval své služby po přerušení 0x21, tedy mohly aplikace provádět, co se jim chtělo.

BIOS

Systémy na platformě IBM PC mají BIOS, kde je implementována základní funkcionalita počítače. Jsou zde obslužné rutiny pro práci s diskem, obrazovkou, klávesnicí a jiné. Vše je implementováno pomocí přerušení. Tak jako tak jde ale o závan minulosti.

Dodnes toto existuje, ale jakmile se zavede OS, tak se tyto zavedené rutiny ignorují a počítač tak řídí přímo OS.

REPREZENTACE ČÍSEL S PLOVOUCÍ ŘÁDOVOU ČÁRKOU

Samotná reprezentace je popsána ve standartu **IEEE 754**.

Čísla jsou dána předpisem

$$\text{hodnota} = (-1)^{znaménko} \times mantisa \times 2^{exponent}$$

U čísel s plovoucí řádovou čárkou je to s přesností na štíru, protože je exponent o základu 2 a ne 10, tak se tato čísla nehodí používat při bankovních operacích, kde jde o každý haléř.

Mimo to zde existuje záporná nula. Je to dáno tím, že si díky mantise a exponentu můžeme nastavit nulu a vlivem znaménka ovlivnit její kladnost, či zápornost.

Zároveň můžeme pomocí těchto čísel zakódovat nekonečno jako maximální exponent a nulovou mantisu.

Hodnota NaN (Not a Number) nejde o číslo, jde o „nějaký nesmysl“, můžeme použít třeba jako zarážku, vytvoříme ji maximálním exponentem a nenulovou mantisou.

Rozlišujeme 3 základní typy

ČÍSLA S JEDNODUCHOU PŘESNOSTÍ (FLOAT)

Velikost 32 bitů

1 bit na znaménko, **8** bitů na exponent (v doplňkovém kódu), **23** bitů na mantisu

ČÍSLA S DVOJITOU PŘESNOSTÍ (DOUBLE)

Velikost 64 bitů

1 bit na znaménko, **11** bitů na exponent (v doplňkovém kódu), **52** bitů na mantisu

ČÍSLA S ROZŠÍŘENOU PŘESNOSTÍ (LONG DOUBLE)

Velikost 80 bitů

1 bit na znaménko, **15** bitů na exponent (v doplňkovém kódu), **64** bitů na mantisu

NÁVRH CPU

REGISTROVÉ PROCESORY

Jedna z koncepcí CPU se označuje jako registrové procesory, kde hodnoty se kterými pracujeme, máme uloženy v registrech. Registry následně přeléváme a podobně.

ZÁSOBNÍKOVÉ PROCESORY

Vlastností zásobníkových procesorů je, že jsou operandy uloženy na zásobníku. Na zásobník se hodnoty přidávají pomocí `push` / `load` a odstraňují pomocí `pop` / `store`. Operace se poté neprovádí s hodnotami v registrech, ale operace pracují s vrcholem zásobníku. Tedy máme jednoduché operace `add`, `sub`, `dup`, `swap`, které pracují přímo se zásobníkem.

Příklad $\mathbf{a^2 - 1}$ bychom v nějaké pseudoinstrukční sadě mohli napsat takto:

```
load 1      // na vrchol zásobníku načti hodnotu 1
load a      // načti hodnotu a
dup         // duplikuj hodnotu na vrcholu zásobníku
mul         // vynásob první dvě hodnoty na zásobníku a ulož
sub         // odečti
```

Všimněte si, že instrukce téměř nemají žádné operandy, tedy se pracuje pouze s vrcholem zásobníku.

Kdybychom na vrchol zásobníku chtěli ukládat také návratové adresy a tomu podobné informace, tak by se vše zbytečně komplikovalo. Proto zásobníkové procesory mají obvykle zásobníků více. Jeden pro operandy, jeden pro ukládání návratových adres a tak dál.

FPU (FLOATING POINT UNIT)

Procesory rodiny x86 řeší výpočty s plovoucí řádovou čárkou pomocí FPU, která vychází z koprocesoru 80x87. Původně šlo o oddělenou jednotku, šlo o podpůrný obvod, který mohl být přítomný na desce, ale taky nemusel.

Tím, že byl tento obvod naprosto mimo, tak práce s ním probíhá jinak, než zbytek procesoru.

Má odlišnou architekturu a platí zde řada omezení. Kvůli oddělení je komunikace s touto jednotkou možná pouze přes paměť.

FPU pracuje s 80b hodnotami (long double) a má zásobník s kapacitou 8 hodnot. Na jednu stranu se tento zásobník chová jako zásobník, a to tehdy, když chceme přidávat nebo odebírat hodnoty z vrcholu zásobníku, ale zároveň se můžeme na tento zásobník podívat jako na registr. Pro tento účel má procesor registry **ST(0) až ST(7)** s tím, že ST(0) představuje vrchol zásobníku.

Pro práci s FPU máme sadu instrukcí.

> **FLD, FST** se starají o načtení hodnot na zásobník, případně ji odeberou. Dále máme několik pomocných operací pro načítání často používaných konstant (FLDZ, FLD0, FLDPI).
> **FADD, FSUB a jiné numerické operace** používají jako jeden argument vrchol zásobníku (ST(0)) a jako druhý argument lze použít kteroukoli jinou hodnotu ze zásobníku (ST(1) až ST(7)), případně hodnotu z paměti. Provede se operace a výsledek se ukládá do ST(0), tedy na vrchol zásobníku.

Větvení kódu je v této části procesoru řešeno jinak, je řešeno pomocí podmíněných přiřazení. Máme speciální operaci FCOM, která provede nám známé srovnání pomocí odečtení, ale mimo to máme také sadu jiných instrukcí FCMOVE, FCMOVB, … které říkají, že se přiřazení provede v případě, že je hodnota menší, větší atd. …

K tomu máme sadu operací pro počítání odmocnin, sinu, cosinu, … (FSQRT, FSIN, FCOS, …).

Pokud voláme funkci, která pracuje s čísly s plovoucí řádovou čárkou, tak jsou hodnoty předávány přes zásobník a návratová hodnota je uložena v registru ST(0).

Toto je sice víceméně historická záležitost, ale pořád to 32b procesory podporují. Nicméně FPU bývá vytlačována pomocí dalších rozšíření, které byly v průběhu let do procesoru x86 začleněny. Velký hon za rozšíření procesoru začal v polovině 90. Let, kdy velký boom zažívala tzv. multimédia. Výrobci procesorů se tehdy začali předhánět, kdo lépe bude zpracovávat právě tato multimédia.

DALŠÍ ROZŠÍŘENÍ

Práce s těmito rozšířeními je intuitivnější práce, než se zásobníkovým procesorem.

První rozšíření, které zaznamenalo velký úspěch, bylo rozšíření MMX.

MMX

V rámci rozšíření MMX byly zavedeny 64b registry mm0 až mm7, které ale bohužel používaly stejné paměťové místo, jako registry ST(0) až ST(7). Tedy si programátor mohl vybrat, zda bude pracovat s FPU, nebo s těmito registry.

Instrukční sada MMX umožnila pracovat s hodnotami jako s vektory 8 Bajtových celých čísel. Tedy jsme mohli rychle provádět vektorové operace, což se uplatnilo při implementaci jednoduché grafiky. Navíc byly zavedeny operace se saturací. (Pomocí tří složek RGB můžeme vyjádřit nějakou barvu. Při sčítání jednotlivých složek různých barev jsme byli schopni ošetřit přetečení, aby v momentě, kdy se hodnota do předpokládaného datového pole nevleze, uložíme maximální možnou hodnotu.)

SSE

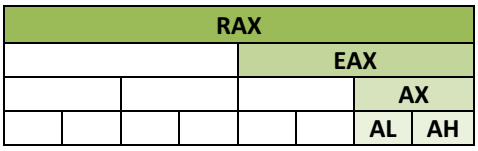
S postupem doby přišlo rozšíření SSE, které zavádí sadu nových 8 registrů XMM0 až XMM7, které jsou 128 bitové. Do těchto registrů se vejdou 4 čísla FP s jednoduchou přesností (float). Toto rozšíření podporuje základní aritmetiku. Tato jednotka umožňuje velice rychle implementovat 3D hry díky tomu, že pomocí čtyřprvkových vektorů se dají dělat nejrůznější transformace s 3D objekty. Na běžné výpočty se tato jednotka tak úplně nehodí právě kvůli přesnosti (jednoduchá přesnost), ale zas u počítačových her, kde na přesnosti tolik nelpíme je dostačující.

SSE2

Tato sada přidává operace pro práci s hodnotami s dvojitou přesností (double). Tato instrukční sada se už dá použít pro CAD systémy a různé výpočty, kde na přesnosti záleží. Navíc naráz se na registry XMM0 až XMM7 můžeme dívat jako na vektory celých čísel, takže buď 16 x 8b hodnota, nebo 8 x 16b hodnota atd. … Pokud potřebujeme pracovat s barevnými složkami, tak jsme schopni toho provádět poměrně dost v rámci jedné instrukce procesoru.

AMD64

Na trhu začínají dominovat procesory, které sice vychází z rodiny x86, ale jsou 64 bitové. Budeme používat označení AMD64, podle výrobce, který první přišel na trh s touto architekturou.



Instrukční sada AMD64 zachovává zpětnou kompatibilitu, tedy lze používat všechny 32b registry a většinu instrukcí. Registry byly rozšířeny na 64 b. Když jsme měli registr EAX, tak šel půlit na AX a následně AX šlo rozdělit na AL a AH. AMD64 přišel s registrem RAX, který nadřadil registru EAX. Obdobně máme RBX, RCX, RSI,… a také RIP.

Vedle toho AMD64 přidává 8 nových 64b registrů **r8 až r15**, kde spodních
32b můžeme adresovat jako **r8d**
16b jako **r8w**
8b jako **r8b**

Zároveň máme nově k dispozici 8 128b registrů pro rozšíření SSE a SSE2.

Navíc nově s procesory Sandy Bridge a Buldozer přišlo rozšíření AVX, které rozšířilo registry XMM0 až XMM15 na 256 bitů a pojmenovali je YMM0 až YMM15. Tedy můžeme provádět vektorové operace s hodně moc čísly v hodně moc registrech.

Všechna rozšíření byla provedena tak, že instrukce lze vybavit prefixem `rex`, tak naráz bude instrukce pracovat s registry RAX a RBX. Nicméně většina překladačů tento prefix vkládá, tedy se o to nemusíme starat.

Zásadní omezení je, že z technických a implementačních důvodů může mít jedna instrukce pouze 15B zakódovaná ve zdrojovém kódu. Což může být problém třeba při práci s velkou konstantou. Pro to bylo zavedeno ještě jedno omezení, že jako konstanty instrukcí můžete používat pouze 32b hodnoty. Pokud do některého registru chcete vložit 64b hodnotu, musíte použít speciální instrukce movabs.

To, že máme více registrů, nám umožňuje rychleji volat funkce, protože hodnoty můžeme předávat pomocí registrů a bez zásobníku. Pak pokud máme 64b registry (což už je docela velké množství dat), tak můžeme do jednotlivých registrů zakódovat jednotlivé hodnoty a opět nemusíme používat zásobník. Příchodem AMD64 se podařilo sjednotit volací konvence napříč platformami. Ne úplně, ale podařilo se rozumnou domluvu.

LINUX A UNIX

V rámci Linuxu a Unix byla vytvořena konvence, kde **prvních 6 argumentů** funkci se předává pomocí registrů **RDI**, **RSI**, **RCX**, **RDX**, **R8**, **R9**. Čísla s plovoucí řádovou čárkou se předávají pomocí XMM0 až XMM3. Kdyby nám počet registrů nestačil, použijeme zásobník. Návratové hodnoty se předají přes RAX, nebo XMM0.

Tato konvence se dodržuje ve většině Unixů, ale Microsoft si to udělal po svém.

WINDOWS

První 4 argumenty se předávají pomocí registrů RCX, RDX, R8 a R9. Čísla s FP se předávají stejně pomocí XMM0 až XMM3, případně zbytek přes zásobník. Návratové hodnoty se předávají pomoci RAX a XMM0.

Z technických důvodů se při předávání argumentů požaduje, aby hodnoty na zásobníku byly zarovnány na 16 B. Při volání funkce, se na zásobníku vytváří místo pro uložení argumentů.

Díky 64b registrům můžeme adresovat rozsáhlý adresní prostor, nicméně zatím nemáme počítače, které by byly schopny obsáhnout takovouto kapacitu paměti. Proto je tedy limit adresace nastaven na 2^{40} B (virtuální 2^{48} B).

AMD 64 pracuje ve více režimech podobně jako x86 (user, kernel mode).

Long mode, kde CPU běží v 64b režimu. Tento mód má dva sub módy – jeden pro běh 64b OS a zároveň aplikace v 64b a druhý mód pro běh OS v 64b režimu a aplikace v 32b.

Legacy mode, zajišťuje zpětnou kompatibilitu

Na AMD64 se pro výpočty s čísly FP používá už výhradně pouze SSE a SSE2.

AT&T

Doteď jsme si představovali syntaxi, kterou označujeme jako Intelovskou, jako INtel. Na řadě jiných platforem, typicky Unixových se z historických důvodů ale používá syntaxe AT&T.

AT&T je nadnárodní telefonní společnost, která před 40 lety stála u vzniku Unixu.

SYNTAX

Operace se zapisují ve tvaru

`<jméno><velikost> zdroj, cíl`

Kde **jméno** je název operace (`mov`, `add`, `cmp`, …)
Velikost je zakódovaná pomocí písmen `b` (8b), `w` (16b), `l` (32b), `q` (64b)
Registry zapisujeme ve tvaru *`%registr`* (`%eax`)
Konstanty začínají $ ($100)
Adresace paměti je ve tvaru *`posunutí, základ, index a faktor`*
(např.: `-10(%ecx, %ebx, 2)` → `[ecx + 2 * ebx – 10]`)

AT&T	Intel
`pushw %ax`	`push ax`
`movl $100, %eax`	`mov eax, 100`
`addl %ebx, %eax`	`add eax, ebx`
`subl (%eax), %ecx`	`sub ecx, [eax]`
`subl (%eax, %ebx), %ecx`	`sub ecx, [eax + ebx]`
`andw $42, -16(%eax)`	`and word ptr [eax – 16], 42`

Tato syntaxe je o něco lepší díky právě označení daných operací. U Intelu můžeme, ale nemusíme používat `word ptr`, který kdybychom nepoužili, nastane problém. U AT&T musíme vždy uvádět velikost operandu.

Se syntaxí AT&T se můžete setkat v praxi, pokud budete chtít louskat kód vygenerovaný překladačem na Unixových systémech. Navíc některé procesory nám známou Intel syntaxi vůbec nepodporují.

SPARC

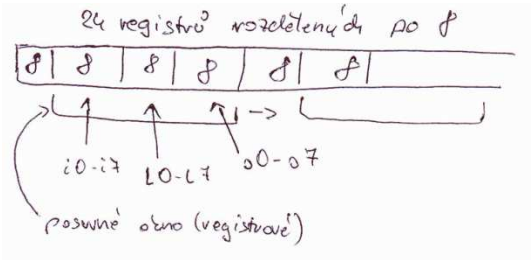
Kompletní dokumentace procesoru je pod licenci GPL, tedy si můžete zdrojové kódy stáhnout a dál použít. Tyto procesory umožňují paralyzovat běh.

REGISTRY

Tento procesor má snahu eliminovat množství operací. Čím méně operací bude mít, tím méně tranzistorů je potřeba a tak můžeme zrychlit jeho běh. V důsledku toho potřebujeme více registrů. V případě procesoru SPARC máme 32 registrů.

Registry lze rozdělit do několika kategorií.

Globální registry G0 až G7, kde G0 je vždy nula a pak funguje něco, co se nazývá **registrové okno**, kde máme řadu registrů a my se po té množině registrů pohybujeme tzv. registrovým oknem. Toto okno má velikost 24 registrů s tím, že spodních 0-7 registrů slouží pro argumenty předávané funkcím, potom uprostřed máme 8 lokálních registrů, které patří funkci, která s těmito registry pracuje. Nakonec máme registry O0 až O7, které slouží jako parametry pro předání dalším funkcím.



Tedy využíváme převážně registry, málo kdy zásobník.

INSTRUKČNÍ SADA

Každá instrukce zabírá v paměti přesně 4B, což výrazně zjednodušuje dekódování programu.

Instrukce pracují se třemi operandy.

ZKOUŠKA ANEB CO SI URČITĚ ZAPAMATOVAT

Deadlock a jeho detekce
Reprezentace čísel v paměti
Fast call/cdecl
Přerušení, přepínání procesu
Semafor a mutex
Rozdíl mezi amd64 a x86
Vlákna
Systémová voláni, ISA
Překlad programu

OTÁZKY Z TESTŮ

SADA I.

1) cdecl a fastcall - popis, co je lepší a proč
2) přerušení - popis, rozdíl mezi voláním podprogramu a případně skokem
3) zapsaní 4 čísel v binárním tvaru. 2x char, 2x unsigned char
4) semafor - k čemu slouží, jak funguje
5) přepínaní procesů
6) detekce deadlocku

SADA II.

1) vlákna v unixech, jak vznikají, vlastnosti a plánováni;
2) 4 čísla zapsat v binárním tvaru

```
unsigned char a = 2;
char b = 2;
unsigned char c = -3;      // Pozor! Překladač jazyka C dovolí tento zápis – dojde k přetečení na 252
char d = -3;
```

3) EBP, co to je, je to nezbytné a proč
4) průběh překladu kódu jazyka C
5) systémová voláni, co to je a jak se to dělá
6) pomocí atomických operací realizujte mutex a napište pseudokód

SADA III.

1) Plánování procesů ve Windows
2) Operace CDQ
3) Registr EF
4) Monitor
5) Detekce deadlocku
6) Rozdíl mezi x86 a AMD64