



Operační systémy

Instrukční sady dalších procesorů a proces překladačů

Petr Krajča



Katedra informatiky
Univerzita Palackého v Olomouci



- rodina procesorů (některé dostupné pod GPL)
- každá instrukce zabírá v paměti 4B
- snaha eliminovat množství instrukcí
- jednoduché instrukce
- potenciálně rychlejší zpracování
- operace běžně se třemi operandy
- velké množství registrů (řádově stovky), běžně dostupných 32 registrů
- globální registry g0 – g7 (g0 je vždy nula)
- registrové okno – 24 registrů
 - i0 – i7 – argumenty předané funkci
 - l0 – l7 – lokální proměnné
 - o0 – o7 – argumenty předávané další funkci
- speciální využití některých registrů
 - fp – frame pointer (i6)
 - sp – stack pointer (o6)
 - návratová adresa – i7/o7



■ příklady operací

```
add    %i0, 1, %l1      # l1 := i0 + 1
subcc  %i1, %i2, %i3    # i3 := i1 - i2
subcc  %i1, %i2, %g0    # g0 := i1 - i2    (cmp)
or     %g0, 123, %l1    # l1 := g0 | 123    (mov)
```

■ malá velikost instrukce

- operace neumožňují adresovat paměť \implies specializované operace ld, st
- \implies load/store architektura
- interně se pracuje s celými registry
- jako konstanty jde běžně používat pouze hodnoty -4096 – 4095
- přiřazení velkých čísel ve dvou krocích

```
sethi  0x226AF3, %l1    # nastavi horni bity
or     %l1, 0x1EF, %l1  # nastavi dolni bity
```

- rodina 32- a 64bitových procesorů typicky využívaná v embedded a přenosných zařízeních
- optimalizace na nízkou spotřebu el. energie a paměti
- není jeden výrobce, licence dalším výrobcům
- základní jádro je licencováno výrobcům k výrobě SoC (Qualcomm Snapdragon, nVidia Tegra, Apple A4-A17, M1-3, Samsung Exynos, ...)
- několik variant instrukční sady—v současné době používané ještě ARMv5, ARMv6, ale především ARMv7 (32 bitů) a ARMv8 (64 bitů)
- děleny ještě podle určení A (aplikační), R (real-time), M (mikrokontrolery)
- architektura *big.LITTLE* – kombinace pomalejších a úspornějších jader (LITTLE) s výkonnými (big), která jsou využívána podle aktuálního zatížení systému

- podpora několika různých typů instrukčních sad (+ rozšíření dle modelu, např. specializované instrukce pro kryptografii)
- přímá podpora až 16 koprocetorů
- load/store architektura

Registry

- 32 obecně použitelných registrů
- z toho jen 16 je v daný okamžik použitelných (R0 – R15)
- R13 (SP) – Stack Pointer, R14 (LR) – Link Register, R15 (PC) – Program Counter
- registry R13 a R14 přepínány podle aktuálního režimu procesoru (jaderný režim, obsluha přerušení, atp.), v případě rychlých přerušení přepnuty R8 až R14
- stavový registr APSR pro případné uložení příznaků proběhlé operace



- všechny instrukce o velikosti 32 bitů
- obvykle 2-3 operandy, příznaky nastavují jen programátorem určené instrukce
- možnost podmíněného vykonávání instrukcí

`CMP R0, 0 ; porovnej R0 s 0`

`RSBLT R0, R0, 0 ; pokud R0 < 0, pak R0 := 0 - R0`

- barrel shifter umístěný před ALU a druhý argument
- umožňuje kombinovat operaci s operací bitových posunů a rotací
- v případě přímých hodnot se používá kombinace 8 bitů pro konstantu a 4 bity pro operaci ROR
- 32bitové konstanty přiřazovány ve dvou operacích (horních/dolních šestnáct bitů)
- operace load/store umožňují měnit registr s indexem
- při volání je návratová adresa uložena do registru R14 (LR)
- argumenty jsou předávány přes registry (první čtyři) a zásobník

Operace typu ADD Rd, Rn, imm

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|----|--------|----|----|----|----|----|----|----|----|----|----|----|----|--------|----|----|----|------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| cond | | | | 00 | | 1 | OpCode | | | | S | Rn | | | | Rd | | | | rotate | | | | imm8 | | | | | | | |

- cond – podmínka (AL, EQ, NE, LT, GT, ...)
- 00 – typ instrukce
- 1 – použije se konstanta
- OpCode – použitá operace ADD, SUB, RSB, MOV, CMP, ...
- Rn, Rd – registry
- imm8 – přímá hodnota
- rotate – aplikuje ROR(imm8, 2 × rotate)

Operace typu ADD Rd, Rn, Rm

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|----|--------|----|----|----|----|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| cond | | | | 00 | | 0 | OpCode | | | | S | Rn | | | | Rd | | | | shift | | | | | | | | Rm | | | |

- shift – dále zakódováno posunutí Rm o pevný počet bitů nebo o hodnotu danou registrem
- podporovány logické i aritmetické posuny, rotace vpravo

00000000 <fact>:

```
0:      e3500000      cmp      r0, #0
4:      0a000005      beq      20 <fact+0x20>
8:      e92d4010      push    {r4, lr}
c:      e1a04000      mov      r4, r0
10:     e2400001      sub      r0, r0, #1
14:     ebfffffe      bl      0 <fact>
18:     e0000094      mul      r0, r4, r0
1c:     e8bd8010      pop      {r4, pc}
20:     e3a00001      mov      r0, #1
24:     e12fff1e      bx      lr
```



Kódování Thumb

- alternativní způsob kódování instrukcí
- zahuštění kódu (použití hlavně v mikrokontrolerech)
- velikost instrukce 16 bitů nebo 32 bitů
- 16 bitová varianta
 - menší počet operandů (podobná ISA x86)
 - možnost přistupovat pouze k části registrů
 - bez možnosti podmíněného provádění instrukcí jednotlivých instrukcí
 - instrukce IT (if-then) supluje předchozí omezení
- 32 bitové instrukce umožňují přístup k dalším vlastnostem ISA
- přepínání mezi kódováním Thumb a ARM přes instrukce (bx, blx)

Další instrukční sady

- Jazelle – spouštění Java bytecode



- 64bitový nástupce architektury ARMv7
- instrukce velikosti 32 bitů
- módy pro zpětnou kompatibilitu
- 31 64bitových registrů (X0 až X30), jde použít i pouze spodní 32bitové poloviny (W0 až W30)
- registr X31/W31 zero registr (podobně jako u SPARC)
- X30 odpovídá LR, samostatné registry SP a PC, PSTATE (stavový registr)
- v 64bitovém režimu některé vlastnosti zrušeny (podmíněné provádění instrukcí) nebo upraveny (bitové posuny u konstant)
- argumenty předávány přes registry (X0 až X7) a další přes zásobník



RISC: Reduced Instruction Set Computer

- zjednodušený návrh a implementace CPU
- rychlejší běh, určitá omezení

CISC: Complete (Complex) Instruction Set Computer

- poskytují operace velice blízké vyšším progr. jazykům
- snadné pro ruční programování
- náročné na implementaci CPU (+ již nepoužívané instrukce v ISA)

Reálně...

- procesory typu CISC provádí rozklad operací na mikrooperace \implies vnitřně RISC
- další úroveň abstrakce
- vnitřně dochází ještě k dalším úpravám kódu, např. přejmenovávání registrů
- out-of-order execution \implies rozdělení (mikro)operací jednotlivým jednotkám \implies paralelismus



- plánování OoO komplikuje návrh CPU

VLIW: very large instruction word

- snaha využít několik funkčních jednotek
- jedna instrukce může obsahovat několik operací
- \implies souběžné zpracování
- spolupráce s překladačem \implies „CPU nemusí hádat, jak poběží program“
- složitější návrh dekodovací jednotky
- Intel Itanium, Digital Signal Processors (DSP)
- v případě AMD64 rozšíření FMA (fused multiply-add)



- 1 preprocessor – expanduje makra, odstraní nepotřebný kód, načte požadované hlavičkové soubory (např. `math.h`) – deklarace struktur, deklarace prototypů, atd.
- 2 překladač – generuje kód v assembleru
- 3 assembler – vygeneruje objektový kód (`foo.c` \implies `foo.obj/foo.o`)
- 4 linker – sloučí několik souborů s objektovým kód + knihovny do spustitelného formátu

Poznámky

- některé kroky mohou být sloučeny nebo vypuštěny
- některé vyšší programovací jazyky jsou překládány do nižšího jazyka (např. C)
- oddělený překlad do objektových souborů a jejich spojení \implies
 - možnost kombinovat různé programovací jazyky (různé jazyky, ale stejné objektové soubory)
 - komplikuje interprocedurální optimalizace (není možné optimalizovat napříč zdrojovými soubory)

- formát specifický pro každý OS
- obecně obsahuje
 - hlavička – informace o souboru
 - objektový kód – strojový kód + data
 - exportované symboly – seznam poskytovaných symbolů (např. funkce nedeklarované jako `static`)
 - importované symboly – seznam symbolů použitých v tomto souboru
 - informace pro přemístění – seznam míst, které je potřeba upravit v případě přesunutí kódu
 - debugovací informace
- rozdělení na sekce
 - kód
 - data jen pro čtení (konstanty, `const`)
 - inicializovaná data (globální proměnné, statické proměnné)
- navíc obsahuje informace o neinicializovaných datech, apod.
- možnost sdílet jednotlivé části mezi instancemi programu
- formát často sdílený i binárními soubory

Linking View

| |
|----------------------|
| ELF header |
| Program Header Table |
| Section 1 |
| Section 2 |
| Section 3 |
| Section <n> |
| Section Header Table |

Execution View

| |
|----------------------|
| ELF header |
| Program Header Table |
| Segment 1 |
| Segment <n> |
| Section Header Table |





- spojí jednotlivé objektové soubory do spustitelného formátu (sloučí jednotlivé sekce)
- postará se o správné umístění kódu a vyřešení odkazů na chybějící funkce a proměnné
- připojení knihoven (hlavičkové soubory většinou neobsahují žádný kód!)

Sticky linkované knihovny

- archiv objektových souborů (+ informace o symbolech)
- výhody: jednoduchá implementace, nulová režie při běhu aplikace, žádné závislosti
- nevýhody: velikost výsledného binárního souboru, aktualizace knihovny \implies nutnost rekompilace

Dynamicky linkované knihovny

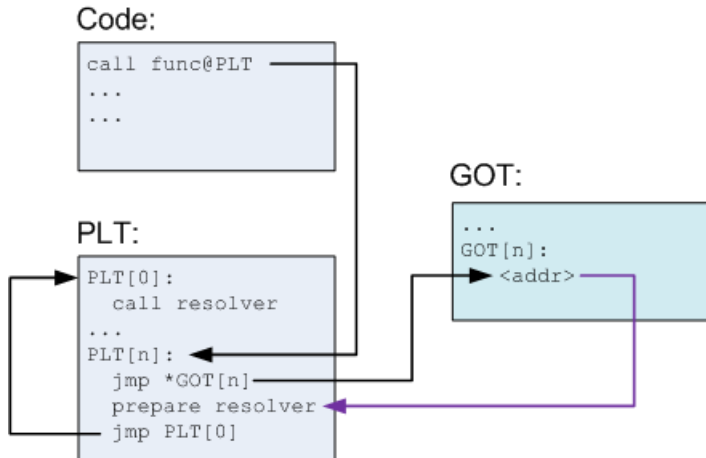
- knihovna je načtena až při spuštění programu
- sdílení kódu mezi programy
- nutnost provázat adresy v kódu s knihovnou
- nutná spolupráce OS



- problém: umístění knihovny v paměti

Řešení v Unixu

- sdílené knihovny (shared objects, `foo.so`)
- \implies position independent code (PIC) – kód, který lze spustit bez ohledu na adresu v paměti
- x86 používá často relativní adresování (i tak PIC pomalejší než běžný kód)
- při spuštění dynamický linker (`ld.so`) provede přenastavení všech odkazů na vnější knihovny
- Global Offset Table (GOT) – tabulka sloužící k výpočtu absolutních adres (nepřímá adresace)
- Procedure Linkage Table (PLT) – tabulka absolutních adres funkcí
 - na začátku PLT obsahuje volání linkeru
 - při volání funkce se provede skok do PLT
 - nastaví se informace o funkci pro linker a ten se zavolá
 - linker najde adresu funkce, nastaví záznam v PLT
 - linker zavolá funkci
 - další volání se provádí bez účasti linkeru \implies adresa v PLT



Code:

```
call func@PLT
...
...
```

PLT:

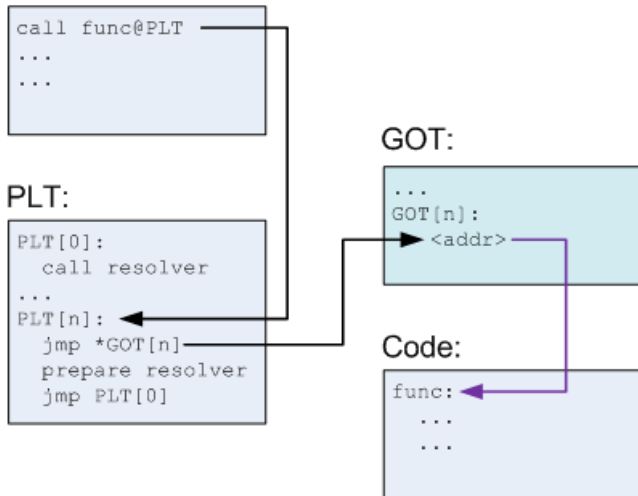
```
PLT[0]:
  call resolver
...
PLT[n]: ←
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

GOT:

```
...
GOT[n]:
  → <addr>
```

Code:

```
func: ←
...
...
```





Řešení ve Windows

- Dynamic-link library (DLL)
- Windows nepoužívá PIC \implies každá knihovna má svou adresu v paměti
- v případě kolize nutnost přesunu + přepočítání absolutních adres
- každý program obsahuje *import address table (IAT)* – tabulka adres volaných funkcí (nepřímá adresace)
- inicializace při spuštění
- volání přes `call [adresa]` nebo *thunk table*

...kód...

```
00401002 CALL 00401D82
```

...thunk table...

```
00401D82 JMP DWORD PTR DS:[40204C]
```

...adresy funkcí...

```
40204C > FC 3D 57 7C ; adresa
```

- vyhledávání funkcí podle čísla nebo jména (binární vyhledávání)



- možnost explicitně nahrát knihovnu za běhu
- implementace pluginu
- mechanismus podobný dynamickému linkování
- Unix: `dlopen`, `dlsym` (vyhledá funkci podle jména)
- Windows: `LoadLibrary`, `GetProcAddress`
- kombinace: zpožděné načítání knihoven

- virtualizace systému vs. virtualizace procesu
- program se nepřekládá do strojového kódu cílového procesoru
- bytecode: instrukční sada virtuálního procesoru (virtuálního stroje, VM)
- bytecode \implies interpretace jednotlivých instrukcí nebo překlad do instrukční sady cílového procesoru běhovým prostředím
- přenositelný kód nezávislý na konkrétním procesoru
- možnost lépe kontrolovat běh kódu (oprávnění, přístupy)
- režie interpretace/překlada
- VM může řešit i komplexnější úlohy než běžný CPU (správa paměti, výjimky, atd.)
- příklady: Java Virtual Machine (& Java Byte Code), Common Language Runtime (& Common Intermediate Language), UCSD Pascal (p-code), LLVM, atd.

- běhové prostředí generuje kód dané architektury za běhu (von Neumannova architektura!)
- možnost optimalizace pro konkrétní typ CPU
- optimalizace podle aktuálně prováděného kódu (profilování)

Zásobníkové virtuální stroje

- jednoduchá instrukční sada \implies snadná implementace
- potřeba více instrukcí, nicméně kratší kód
- JVM, CLR

Registrové virtuální stroje

- efektivní překlad do instrukční sady (pipelined) procesorů
- odolnější proti chybám
- LLVM – optimalizace přes Single Static Assignment; Parrot – Raku (Perl 6); Dalvik – dříve Android, minimální spotřeba paměti



- 1995: SUN prog. jazyk Java 1.0
- překlad Java \implies Java Bytecode (JBC)
- JBC vykonáván pomocí Java Virtual Machine (JVM)
- implementace JVM není definovaná (pouze specifikuje chování), JBC lze
 - interpretovat
 - přeložit do strojového kódu daného stroje (JIT i AOT)
 - provést pomocí konkrétního CPU
- JVM – virtuální zásobníkový procesor
- malý počet instrukcí (< 256)
- zásobník obsahuje rámce (rámec je vytvořen při zavolání funkce)
 - lokální proměnné, mezivýpočty
 - *operand stack* – slouží k provádění výpočtů
- heap s automatickou správou paměti



- jednoduché i velmi komplexní operace (volání funkcí, výjimky)
- základní aritmetika s primitivními datovými typy (hodnoty menší než int převedeny na int)
- speciální operace pro práci s prvými argumenty, lokálními prostředími, jedničkou, nulou
- pouze relativní skoky

Příklad

```
public static void foo(int a, int b) {  
    System.out.println(a + b);  
}
```

Code:

```
0: getstatic      #21; //Field java/lang/System.out:Ljava/io/PrintStream;  
3: iload_0  
4: iload_1  
5: iadd  
6: invokevirtual  #27; //Method java/io/PrintStream.println:(I)V  
9: return
```



- Microsoft .NET implementuje obdobný přístup
- Common Language Runtime (CLR) + Common Intermediate Language (CIL) – běhové prostředí + bytecode
- koncepčně velice podobné JVM a JBC
- od začátku navržen s podporou více jazyků
- při prvním zavolání metody \implies překlad do strojového kódu CPU

Opuštění běhového prostředí

- Java: Java Native Interface – rozhraní pro spolupráci s C++
- .NET: Platform Invocation Services (P/Invoke) – umožňuje spustět kód z DLL