KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Externí assembler, registry a základní aritmetické operace

KMI/OS1 Operační systémy I

Mgr. Markéta Trnečková, Ph.D.
www.marketa-trneckova.cz

Fáze překladu

1. **Předzpracování preprocesorem** – odstranění komentářů, expanze maker, vkládání souborů, ...
   příkaz `cpp`
2. **Překlad do jazyka symbolických adres** – zápis v podobě jednotlivých instrukcí procesoru
3. **Překlad do objektového souboru** – pomocí *assembleru*, obsahuje program přeložený do strojového kódu + další informace (konstanty, názvy funkcí a proměnných, ladící informace, ...)
   příkaz `as`
4. **Sloučení objektových souborů** a spojení s knihovnami, výsledkem je binární soubor
   příkaz `ld`

Rozdělení kódu do více souborů

- lepší přehlednost kódu
- oddělený překlad
- **Výhody odděleného překladu**:
  - není nutné vždy překládat celý zdrojový kód
  - jednotlivé části mohou být psány v jiném jazyce
- pro pohodlný překlad – nástroj `make`
- `make` sestaví program podle pravidel, která zapisujeme do `Makefile`

Makefile

- Pravidla

```
cil: zavislosti
<TAB!>prikaz pro sestaveni cile
<TAB!>prikaz pro sestaveni cile
```

Příklad

```
hello: hello.o
	gcc -o hello hello.o
hello.o: hello.c
	gcc -c hello.c
```

Sestavení většího programu

```
gcc -o hello hello.o myfuncs.o
```

Příklad (Příklad z minula)

Vezměte funkce `int2bits()` a `bits2int()` a umístěte je do samostatného souboru `bits.c` a vytvořte odpovídající hlavičkový soubor `bits.h`. Vytvořte program, který popsané funkce bude používat. Pro překlad programu vytvořte vhodný makefile a program přeložte.

Příklad

bits.h

Příklad (Řešení)

```
// bits.h

#ifndef BITS_H
#define BITS_H

#define VELIKOST_INT (sizeof(int) * 8)

void int2bits(char*, int);
int bits2int(char *);

#endif
```

Příklad

bits.c

Příklad (Řešení)

```c
// bits.c
#include "bits.h"

void int2bits(char* retezec, int cislo){
    for(int i = VELIKOST_INT - 1; i >= 0; i--){
        retezec[VELIKOST_INT - 1 - i] = (cislo & (1 << i)) ? '1' : '0';
    }
    retezec[VELIKOST_INT] = '\0';
}

int bits2int(char *retezec){
    int cislo = 0;
    for(int i = 0; i < VELIKOST_INT; i++){
        cislo=(retezec[i]=='1') ? (cislo|(1<<(VELIKOST_INT-1-i))) : cislo;
    }
    return cislo;
}
```

Příklad

reseni.c

Příklad (Řešení)

```c
#include <stdio.h>
#include "bits.h"

int main(int argc, char* argv[]){
    char retezec[VELIKOST_INT + 1];

    int2bits(retezec,100);
    printf("%s\n", retezec);
    printf("%i\n", bits2int(retezec));
     return 0;
}
```

Příklad
Makefile

Příklad (Řešení)

```
reseni: reseni.o bits.o
	gcc -o reseni reseni.o bits.o
reseni.o: bits.h reseni.c
	gcc -c reseni.c
bits.o: bits.h bits.c
	gcc -c bits.c
```

Externí assembler, registry a základní aritmetické operace

Učební text ke cvičení:
`https://phoenix.inf.upol.cz/~krajcap/courses/2025LS/OS1/tutorial03.pdf`

Assembler

- Assembler – vytváření programů na té nejnižší úrovni
- dříve kvůli rychlosti
- nyní
  - v systémovém programování (operační systémy, překladače, . . .)
  - systémy s omezenými prostředky (řídící systémy, spotřební elektronika, . . .)
  - v programech využívající specializované funkce procesoru (zpracování obrazu, kryptografie)
- My budeme pracovat s procesorem Intel x86 (resp. AMD64)

Assembler – tvorba kódu

- všechen kód napsat v assembleru a přeložit jej pomocí assembleru do strojového kódu, který lze spustit jako samostatný program
- je možné také kombinovat kód ve vyšším programovacím jazyce (např. C) s kódem v assembleru pomocí tzv. inline assembleru

Příklad (inline assembler)

```
int inc(int n){
    _asm{
        mov eax, n // do registru eax nacteme hodnotu argumentu
        add eax, 1 // k hodnote pricteme jednicku
        mov n, eax // hodnotu vratime do promenne n
    } return n;
}
```

Překladač MSVC nepodporuje v inline assembleru jinou architektu než i386 a inline assembler v překladači GCC není úplně intuitivní. Budeme pracovat v externím assembleru.

Programování v assembleru

Překlad programu

- zdrojový kód `demo.asm`
- překlad pomocí `nasm` (přívětivější než `as`)
  `nasm -f elf64 demo.asm`
- vznikne soubor `demo.o`, ve formátu `elf64` (implicitně využívaný v `gcc`)

Programování v assembleru
Příklad

Příklad

```
; soubor demo.asm
global foo
section .text
foo:
    mov eax, 42
    ret
```

- `;` komentář
- `global` jaké proměnné a funkce daný zdrojový kód poskytuje
- `section .text` – vyznačení sekce `.text` (obsahuje kód programu v jazyce symbolických adres)
- `foo:` návěští uvozující kód funkce (`foo` je identifikátor funkce)

Programování v assembleru
Příklad

Příklad

```
; soubor demo.asm
global foo
section .text
foo:
    mov eax, 42
    ret
```

- `mov eax 42`
  uloží do registru `eax` hodnotu 42
  `eax` (resp. `rax`) slouží k předání celočíselné návratové hodnoty
- `ret`
  instrukce návratu z funkce

Programování v assembleru
Příklad

Příklad

Přeložte

```
; soubor demo.asm
global foo
section .text
foo:
    mov eax, 42
    ret
```

pomocí příkazu

```
nasm -f elf64 demo.asm
```

Programování v assembleru

Spojení s jazykem C

Příklad

```
// hello.c
#include <stdio.h>

/* Prototyp funkce napsane v asembleru */
int foo();

int main(int argc, char* argv[]){
    printf("Answer to life, etc.: %i\n",foo());
    return 0;
}
```

Samotné provázání pomocí linkeru.

Příklad

Jak by vypadal Makefile?

Příklad (Řešení)

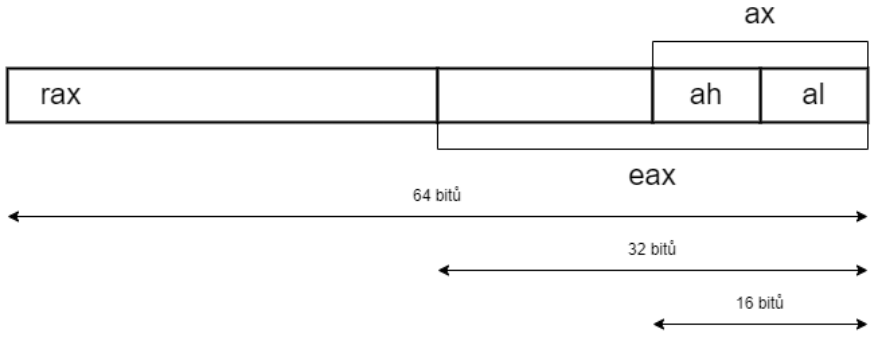
```
hello: hello.o demo.o
	gcc -o hello hello.o demo.o
hello.o: hello.c
	gcc -c hello.c
demo.o: demo.asm
	nasm -f elf64 demo.asm
```

Instrukční sada x86/AMD64

Registry

- **64bitové**: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8` až `r15`,
- **32bitové**: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `r8d` až `r15d`,
- **16bitové**: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `r8w` až `r15w`,
- **8bitové**: `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh`, `dl`, `sil`, `dil`, `r8b` až `r15b`.

Vzájemně si odpovídající registry sdílí stejnou paměť



Instrukční sada x86/AMD64

Registry

- **Další registry**:
  - `rsp`, `rbp` – specifická funkce, nelze je libovolně měnit
  - `rip`, `rf` – lze měnit jen určitými instrukcemi

Instrukční sada x86/AMD64

Aritmetické operace

```
nazev_instrukce operand [, dalsi_operand, ...]
```

- **Operandy**:
  - `r` – registr
  - `m` – adresa místa v paměti
  - `i` – přímá hodnota (konstanta)
- paměť lze v instrukci adresovat pouze jednou

Instrukční sada x86/AMD64

Aritmetické operace

- **Přiřazení hodnoty** `op1 = op2`:
  - `mov r/m, r/m/i`
- **Součet** `op1 = op1 + op2`:
  - `add r/m, r/m/i`
- **Rozdíl** `op1 = op1 - op2`:
  - `sub r/m, r/m/i`
- **Změna znaménka** `op1 = -op1`:
  - `neg r/m`
- **Inkrementace** `op1 = op1 + 1`:
  - `inc r/m`
- **Dekrementace** `op1 = op1 - 1`:
  - `dec r/m`

Instrukční sada x86/AMD64

Aritmetické operace

- **Neznamenkové násobení** `edx:eax = eax * op1`:
- **Neznamenkové násobení** `rdx:rax = rax * op1`:
  - `mul r/m`
- **Znaménkové násobení** `op1 = op1 * op2`:
  - `imul r, r/m`
- **Znaménkové násobení** `op1 = op2 * op3`:
  - `imul r, r/m, i`

Instrukce určené registry, se kterými pracují, a ty jsou ještě ovlivněny velikostí operandu dané instrukce. To znamená, že pokud máme operand instrukce `mul` 32bitový, bude výsledek násobení uložen do dvojice registrů `edx` (horních 32 bitů výsledku) a `eax` (spodních 32 bitů výsledku).

Instrukční sada x86/AMD64

Aritmetické operace

- **Neznaménkové dělení** `eax = edx:eax / op1; edx = edx:eax % op1`:
- **Neznaménkové dělení** `rax = rdx:rax / op1; rdx = rdx:rax % op1`:
  - `div r/m`
- **Znaménkové dělení** `eax = edx:eax / op1; edx = edx:eax % op1`:
- **Znaménkové dělení** `rax = rdx:rax / op1; rdx = rdx:rax % op1`:
  - `idiv r/m`

Instrukce určené registry, se kterými pracují, a ty jsou ještě ovlivněny velikostí operandu dané instrukce. To znamená, že pokud máme operand instrukce `div` 32bitový, budeme pracovat s registry `edx` a `eax`, je nutné tedy správně nastavit obojí. Pracujeme-li s kladnými čísli, stačí nastavit `edx` na 0.

Praktická práce s assemblerem

- Pomocí direktivy `global` určíme, jaké funkce jsou implementovány v assembleru
- V sekci `.text` implementujeme jednotlivé funkce, kgteré vyznačíme návěštím.
- Funkce vrací výsledek v `eax` (`rax`) a je ukončena instrukcí `ret`.
- V jazyce C definujeme prototypy daných funkcí a voláme je standardním způsobem.
- Při sestavování programu sloučíme kód v C a v assembleru.

- Uvažujeme-li unixový operační systém, pak můžeme funkcím předávat celočíselné argumenty po řade v registrech:
  `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
- **Obsah registrů:** `rbx`**,** `rsp`**,** `rbp`**,** `r12`**,** `r13`**,** `r14`**,** `r15`**, musí být na konci funkce stejný, jako na jejím začátku!!!**

Příklad

Příklad

```
; soubor tutorial03.asm
global inc1
global rectangle_circumference

section .text
; funkce jednoho argumentu, zvetsi argument o 1
inc1:
    mov eax, edi ; prvni argument presuneme do eax
    add eax, 1 ; zvetsime o 1
    ret
; funkce pro vypocet obvodu obdelnika
rectangle_circumference:
    mov eax, edi ; prvni argument presuneme do eax
    add eax, esi ; druhy argument pricteme k eax
    add eax, eax ; vynasobime 2
    ret
```

Příklad

Příklad

```
// tutorial03-test.c
#include <stdio.h>

/* Prototyp funkce napsane v asembleru */
int inc1(int arg);
int rectangle_circumference(int a, int b);

int main(int argc, char* argv[]){
    printf("5 + 1 = %d\n", inc1(5));
    printf("Obvod obdelnika je %d\n", rectangle_circumference(4, 5));
    return 0;
}
```

Příklad

Příklad (Makefile)

```
tutorial03-test: tutorial03-test.o tutorial03.o
	gcc -o tutorial03-test tutorial03-test.o tutorial03.o
tutorial03-test.o: tutorial03-test.c
	gcc -c tutorial03-test.c
tutorial03.o: tutorial03.asm
	nasm -f elf64 tutorial03.asm
```

Jazyk C
Úkoly k procvičení

1. Napište v assembleru funkci `int obsah_obdelnika(int a, int b)`, která spočítá obsah obdélníka.
2. Napište v assembleru funkci `int obvod_ctverce(int a)`, která spočítá obvod čtverce.
3. Napište v assembleru funkci `int obsah_ctverce(int a)`, která spočítá obsah čtverce.
4. Napište v assembleru funkci `int obvod_trojuhelnika(int a, int b, int c)`, která spočítá obvod trojúhelníka.
5. Napište v assembleru funkci `int obvod_trojuhelnika2(int a)`, která spočítá obvod rovnostranného trojúhelníka.
6. Napište v assembleru funkci `int obsah_trojuhelnika2(int a, int b)`, která spočítá obsah pravoúhlého trojúhelníka.
7. Napište v assembleru funkci `int objem_krychle(int a)`, která spočítá objem krychle.
8. Napište v assembleru funkci `unsigned int avg(unsigned int a, unsigned int b, unsigned int c)` pro výpočet aritmetického průměru tří čísel typu `unsigned int`.