Arhitektura računala 2, vježba 2: arhitekura x86

Predmet ove vježbe je programiranje u strojnom jeziku arhitekture x86, te povezivanje strojnog koda s višim programskim jezikom.

1. Priprema

Upoznati se s osnovnim svojstvima strojnog jezika za instrukcijsku arhitekturu x86, te posebno proučiti dostupne na adresne načine i registre [1], [2]. Upoznati se s načinima (konvencijama) prenošenja parametara u potprograme [3], te posebno proučiti konvenciju `cdecl` za 32-bitne operacijske sustave koja će biti korištena u okviru ove vježbe.

2. Programsko okruženje

Potprograme u strojnom jeziku x86 pozivat ćemo iz programa u C++-u. Izvođenje strojnih instrukcija pratit ćemo iz standardnih programa za praćenje (*debuggera*). Nažalost, sintaksa unošenja strojnog koda u programski jezik C/C++-u nije ujednačena kod popularnih prevodioca. Zato su u nastavku dane upute za pisanje strojnih potprograma pod prevodiocima **MSVC** i **gcc**.

3. Rad s gcc-om

Strojni potprogram je **gcc**-u najlakše zadati u posebnoj datoteci s ekstenzijom **.s**. Datoteka koja definira strojni potprogram `potprogram_asm` ima sljedeću osnovnu strukturu:

```
// ovo je komentar
//
// oznaka sintakse:
.intel_syntax noprefix

// neka simbol potprogram_asm
// bude vidljiv izvana:
.global potprogram_asm

// odredišna oznaka potprograma:
potprogram_asm:

// ... strojni kod
```

Ovako definirani strojni potprogrami pozivaju se na jednak način kao i obični potprogrami u C-u, kao što će biti detaljnije objašnjeno kasnije. Za sada samo pretpostavimo da je glavni program smješten u datoteci `lab1gcc.cpp`, dok je strojni potprogram smješten u `lab1gcc.s`. Tada se prevođenje i povezivanje može obaviti naredbom (pripazite, na 64-bitnim Unixima ćete možda htjeti promijeniti ovaj poziv kao što je objašnjeno u odjeljku 7):

```
$ g++ -g -o lab1gcc lab1gcc.s lab1gcc.cpp
```

Praćenje programa sad se može inicirati naredbom:

```
$ gdb lab1gcc
```

Za ovu vježbu nam treba samo mali podskup svih mogućnosti programa gdb [4], tj. naredbe `break`, `run`, `next` ,`step`, `print`, te `info registers`. Način upotrebe tih naredbi može se proučiti u dokumentaciji [5].

4. Rad s MSVC-om

Strojni potprogram je MSVC-u najlakše zadati u tijelu tzv. *gole funkcije*, kako slijedi:

```
// direktiva __declspec(naked) kaže prevodiocu
// da pozivatelj parametre prebaci konvencijom cdecl,
// te da se ne generira nikakav kod
// prije i poslije tijela funkcije

int __declspec(naked) potprogram_asm(int i){

  __asm{
    // ... strojni kod
  }
}
```

Ovako definirani strojni potprogrami pozivaju se na jednak način kao i obični potprogrami u C-u, kao što će biti detaljnije objašnjeno kasnije. Prevođenje datoteke sa strojnim potprogramom odvija se na standardan način. U konzolni projekt potrebno je dodati datoteku s izvornim kodom, te pokrenuti prevođenje (*Build solution*).

Praćenje programa može se inicirati iz integrirane razvojne okoline (*Start debugging*). Korisne akcije su *Toggle breakpoint*, *Step over*, *Step into*, korisni prozori su Watch i Registers.

5. Struktura strojnog potprograma

Standardni način pristupanja parametrima i lokalnim varijablama koristi registar `ebp`. Da bi se to omogućilo u okviru konvencije `cdecl`, strojni potprogrami često imaju sljedeću strukturu [3]:

```
                    /* cdecl prolog: */
push  ebp           /* spremi ebp */
mov   ebp, esp      /* ubaci esp u ebp */

                    /* zauzmi 4 bajta za lokalne varijable: */
sub   esp, 4        /* lokalne varijable su "ispod" ebp */


                    /* glavna funkcionalnost potprograma */
...
                    /* povratna vrijednost je zadržana u eax*/


                    /* oslobodi lokalne varijable:*/
add   esp, 4

                    /* cdecl epilog: */
pop ebp             /* umjesto 'add esp,4, pop ebp' može biti 'leave'*/
ret                 /* povratak iz potprograma */
```

6. Primjer potprograma

Razmotrimo potprogram (`potprogram_c`):

```
int potprogram_c(int a, int b, int c) {
  return (a + b) * c;
}
```

Tijelo odgovarajućeg strojnog potprograma (`potprogram_asm`) bilo bi:

```
                    /* [ebp] je pohranjena vrijednost ebp  */
                    /* [ebp+4] je povratna adresa!         */
mov   eax, [ebp+12] /* b */
add   eax, [ebp+8]  /* a */
imul  eax, [ebp+16] /* c */
```

Povratna vrijednost se vraća preko registra `eax`. Prije i poslije gornjeg koda, potrebno je navesti standardni prolog i epilog (zbog referenciranja preko registra ebp). Bez prologa i epiloga, potprogram bi izgledao:

```
sub_asm_noebp:
                      /* [esp] je povratna adresa! */
  mov   eax, [esp+8]  /* b */
  add   eax, [esp+4]  /* a */
  imul  eax, [esp+12] /* c */
  ret                 /* povratak iz potprograma */
```

Obratite pažnju da potprogrami moraju očuvati vrijednosti nekih registara. Prilikom povratka iz potprograma ti registri (npr. ebx) moraju imati istu vrijednost kao i u trenutku poziva. Ti registri su u dokumentaciji označeni terminom *callee-saved*.

7. Strojni potprogrami pod g++-om za 64-bitne Linux, FreeBSD i OS X sustave

Upute iz točaka 3, 5 i 6 nisu primjenjive na 64-bitnim UNIX-ima, jer tamo g++ ne podržava konvenciju cdecl. Taj problem može se riješiti na dva načina:

1. prevesti program za 32-bitnu platformu navođenjem zastavice `-m32`
   - obratite pažnju da ćete možda morati instalirati pakete s 32-bitnim bibliotekama;
2. prijenos parametara implementirati prema standardiziranoj konvenciji za 64-bitovnu inačicu instrukcijske arhitekture x86 (System V AMD64 ABI):
   - prvih šest argumenata koji su cijeli brojevi ili pokazivači prenose se preko registara `rdi`, `rsi`, `rdx`, `rcx`, `r8`, i `r9`;
   - povratna vrijednost prenosi se preko registra `rax`.

8. Poziv strojnog potprograma

Strojni potprogram se poziva transparentno, sasvim jednako kao i potprogram u C-u. Dakle, potprogrami `potprogram_asm` i `potprogram_c` pozivaju se na sasvim jednak način. Ukoliko definicija potprograma nije vidljiva na pozivnom mjestu (npr, ako je potprogram u zasebnoj datoteci) potrebno je kreirati odgovarajući prototip.

Strojni potprogram koji se zadaje u datoteci s čistim asemblerom (.s, gcc) proizvodi objektni kod u skladu s platformskim binarnim standardom (ABI) za jezik C. Ukoliko takav potprogram

želimo pozivati iz C++-a, prototip je potrebno prefiksirati s `extern C`, kako bi se spriječilo dekoriranje imena potprograma.

```
extern "C" int potprogram_asm(int,int,int);
```

Ako želite koristiti gcc na Windowsima, potrebno je prevoditelju reći da prilikom prevođenja glavnog programa ime vanjske funkcije *ne prefiksira* podvlakom. To se postiže ključnom rječju *asm()* u deklraciji prototipa vanjske funkcije:

```
extern "C" int potprogram_asm(int,int,int) asm("potprogram_asm");
```

Ako se to ne napravi, linker će se potužiti da ne može razriješiti referencu na simbol `potprogram_asm`. Drugi način da ovo postignemo jest da u asembleru dodamo i eksternu labelu s podvlakom:

```
.global potprogram_asm
.global potprogram_asm_

potprogram_asm:
potprogram_asm_:
...
```

9. Zadatci

a. Razraditi primjer s procedurama `potprogram_asm` i `potprogram_c`:
   - sastaviti, prevesti i isprobati program koji se sastoji od potprograma `potprogram_c` i `potprogram_asm` te (glavnog) ispitnog programa:

```
int main(){
    std::cout <<"ASM: " <<potprogram_asm(3,5,6) <<std::endl;
    std::cout <<"C++: " <<potprogram_c(3,5,6) <<std::endl;
}
```

   - pogledati kakav strojni kod generira prevodioc za `potprogram_c`, i to za različite stupnjeve optimizacije (MSVC: Project properties -> C/C++ -> Output files -> Assembler output; gcc: opcije -S -masm=intel)
   - sastaviti, prevesti i isprobati program koji se sastoji od potprograma `potprogram_c` i `potprogram_asm` te (glavnog) ispitnog programa.

b. Korištenjem dokumentacije za instrukcijsku arhitekturu x86 [6] napisati strojni potprogram koji:
   - smješta broj 42 u registar eax.
   - smješta broj 0x42 u registar ebx.
   - upisuje 0xffff u gornjih 16 bitova registra edx (uputa: iskoristite instrukcije push, mov i pop).
   - zamjenjuje donjih 8 bitova registra edx s brojem 0xdd.
   - Točno izvođenje programa provjerite praćenjem.

c. Napisati dvije verzije (C, asembler) potprograma za zbrajanje svih cijelih brojeva u intervalu [0,n> gdje je n parametar potprograma. Upute:
   - napisati ispitni program koji testira funkcionalnost oba potprograma

- trebaju nam tri lokalne varijable (brojač, zbroj i granica) koje možemo spremiti u registre ecx, eax, edx (ne trebaju nam memorijske lokalne varijable)
- brojač i zbroj inicijalizirati na 0,
- uzmite u obzir da možemo imati 0 prolaza kroz petlju jer n može biti 0: cbog toga će nam trebati dva grananja
- brojač i zbroj uvećavati instrukcijom add
- uvjetni skok izvesti naredbama cmp i jl
- kako se povratna vrijednost vraća preko eax, nakon uvjetnog skoka možemo se vratiti u glavni program
- (jedno) rješenje ima 15 linija asemblera uključujući 3 odredišne oznake i 2 prazna retka

d. Napisati tri verzije potprograma za zbrajanje dvaju polja podataka tipa float, i to koristeći i) standardni C, ii) strojni jezik s instrukcijama iz podskupa x87, iii) strojni jezik s instrukcijama iz podskupa SSE. Upute:
   - Neka potprogram ima prototip `void sum_c(float const* A, float const* B, int count, float *R);`
   - Napisati potprogram u C-u, te ispitni program, testirati potprogram u C-u.
   - Napisati i ispitati ekvivalentan potprogram u strojnom jeziku koji bi koristio strojne instrukcije iz poskupa x87 (`fld`, `fadd`, `fstp`) [8,9]
     - za pristupanje elementima polja može se koristiti indeksno adresiranje, npr: `fld DWORD PTR [eax+ecx*4]`
     - ako koristimo indeksno adresiranje, treba nam 5 lokalnih varijabli; logičan raspored je: A -> `eax`, B -> `ebx`, index -> `ecx`, count -> `edx`, R -> `edi`
     - kako `ebx` i `edi` moramo vratiti netaknute, na početku procedure ih treba smjestiti na stog, a pri izlasku vratiti sa stoga
     - (jedno) rješenje ima 23 linije asemblera uključujući 3 odredišne oznake i 3 prazna retka
   - napisati i ispitati ekvivalentan potprogram u strojnom jeziku koji bi koristio vektorske instrukcije iz poskupa SSE (`movq` ili `movaps`, `addps`) u kombinaciji s registrima iz skupa XMM [7,8,9].
   - usporediti brzine izvođenja triju implementacija
   - **BONUS** Oblikuj dvije dodatne implementacije postupka. U prvoj implementaciji umanji broj prolaza kroz petlju Duffovim pristupom, a u drugoj naprosto pozovi funkciju saxpy iz neke optimirane implementacije biblioteke BLAS za tvoju platformu. Komentiraj relativnu performansu svih isprobanih pristupa.

10. Upute za predaju u sustav Ferko

Molimo vas da u sustav Ferko uploadate zip arhivu sa rješenjem zadatka 9c. Arhiva treba sadržavati samo datoteke s izvornim kodom. Datoteke nazovite main_gcc.cpp i p_asm.s za GCC odnosno main_msvc.cpp za MSVC.

Preporučujemo da napravite cijelu pripremu kako biste bili spremni za kolokviranje vježbe.

Reference

[1] Wikipedia: x86 architecture

[2] Wikipedia: x86 assembly language

[3] Wikipedia: x86 calling conventions

[4] Wikipedia: GNU Debugger

[5] Using GNU's GDB Debugger

[6] x86 Instruction Set Reference

[7] Wikipedia: Streaming SIMD Extensions

[8] x86 Instruction Set Reference

[9] x86 Instruction Set Reference

[10] x86 instruction listings