Asemblersko programiranje za x86_64 arhitekturu

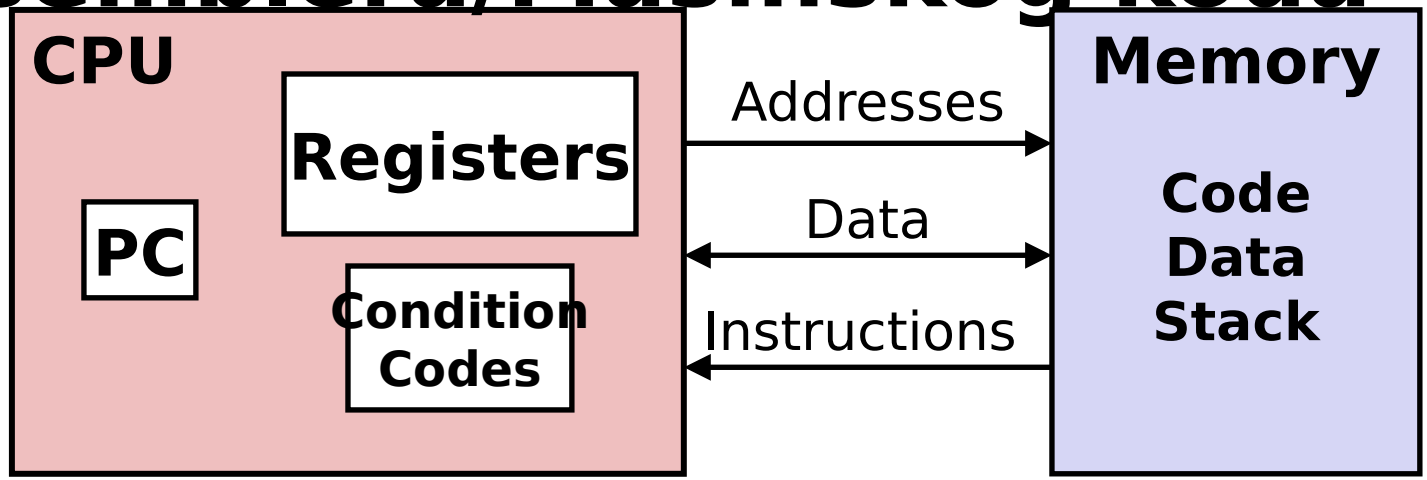
Zašto učiti asembler?

- **Najverovatnije se nećete profesionalno baviti pisanjem asemblerskih programa**
  - Kompajleri to rade na mnogo efikasniji način od ljudi
- **Međutim, razumevanje asemblera je ključ poznavanja modela izvršavanja programa na mašinskom nivou**
  - Ponašanje programa u prisustvu bagova
    - Modeli visokog nivoa ne mogu da opišu ta ponašanja
  - Optimizovanje performansi programa
    - Razumevanja koje optimizacije kompajler radi ili ne radi
    - Razumevanje uzroka neefikasnosti programa
  - Implementacija sistemskog softvera
    - Kompajleri imaju mašinski kod kao izlaz
    - Operativni sistemi moraju upravljati procesima

Definicije

- **Arhitektura procesora: (engl. ISA: instruction set architecture) Delovi procesorskog dizajna koji se moraju poznavati za pisanje korektnog mašinskog/asemblerskog koda**
  - Obuhvata specifikaciju registara, skupa instrukcija, načina adresiranja,...
  - **Mašinski kod**: Niz bajtova koji čini program koji procesor izvršava
  - **Assemblerski kod**: Tekstualna reprezentacija mašinskog koda
- **Mikroarhitektura: Konkretna implementacija arhitekture**
  - Obuhvata na primer veličinu keša i osnovnu frekvenciju procesora

Perspektiva Asemblera/Mašinskog koda



Stanje vidljivo programeru

- **PC: Program counter**
  - Adresa sledeće instrukcije
  - Zove se "RIP" u x86-64 arhitekturi
- **Registarska banka**
  - Često upotrebljavani podaci programa
- **Kodovi uslova (condition codes)**
  - Čuvaju statusne informacije o poslednjoj aritmetičko logičkoj

- **Memorija**
  - Adresiranje na nivou bajta
  - Kod (**.text**) i podaci (inicijalizovani **.data**, neinicijalizovani **.bss**)
  - Stek za podršku potprogramima

Tipovi podataka u asembleru

- **Celobrojni “Integer” podaci od 1, 2, 4, ili 8 bajtova**
  - Vrednosti podataka
  - Adrese (pokazivači opšteg tipa)
- **Podaci u pokretnom zarezu (Floating point) od 4, 8, ili 10 bajtova**
- **(SIMD vektorski tipovi podataka od 8, 16, 32 or 64 bajta)**
- **Kod: Sekvenca bajtova koja predstavlja niz instrukcija**

Predstavljanje prostih C tipova u asembleru

- **Inicijalizovani podaci u Cu**

```
char  c = '0';

short s = 1;

int i = 2;

long l = 3l;

void *p = &l;

float f = 5.0f;

double d = 6.0;

char str[] = “Niz znakova";
```

- **Asemblerski ekvivalenti**

```
 .data #mem. sekcija za podatke
c:    .byte   99
      .align 2 #uravnavanje adr.
s:    .word  1
      .align 4
i:    .long   2
      .align 8
l:    .quad   3
      .align 8
P:    .quad   l
      .align 4
f:    .long   0x40a000000
      .align 8
d:    .quad 0x4018000000000000
      .align 8
str: .string "Niz znakova"
```

Predstavljanje prostih C tipova u asembleru

- **Neinicijalizovani podaci u Cu**

```
char  c;
short s;
int i;
long l;
void *p;
float f;
double d;
char str[10];
```

- **Asemblerski ekvivalenti**

```
.comm   c,1,1
.comm   s,2,2
.comm   i,4,4
.comm   l,8,8
.comm   p,8,8
.comm   f,4,4
.comm   d,8,8
.comm   str,10,8
```

`.comm simbol, broj_bajtova, [uravnavanje]`

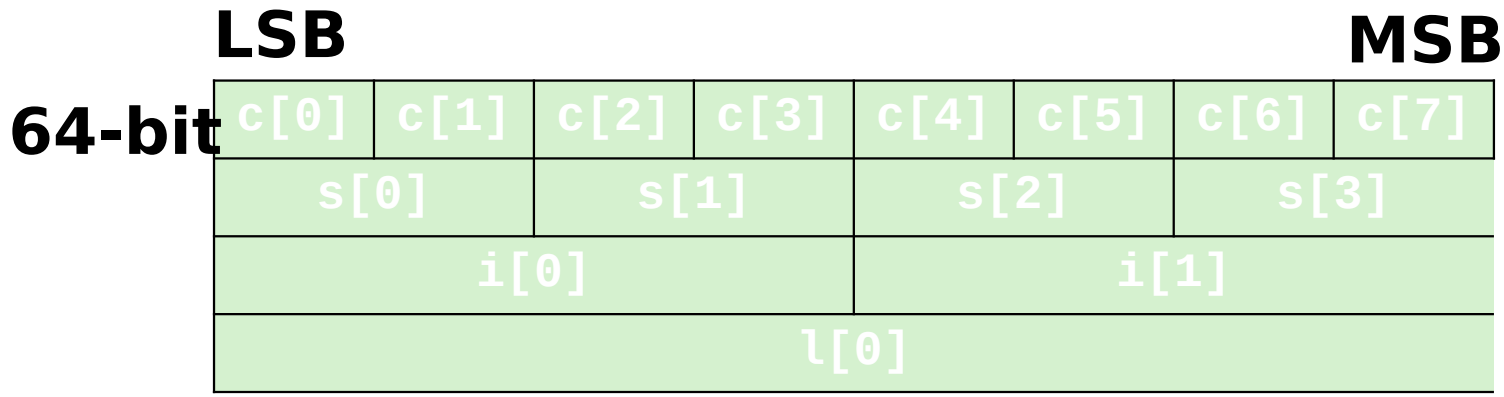
Uvodi simbol bez inicijalne vrednosti. Ako ga ne vidi linker (static u Cu) onda ovi simboli idu redom u .bss sekciju. Inače linker pokušava da upari sa definijom istog

Ređanje bajtova u memoriji

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

X86 arhitektura je little-endian
što znači da na najnižu memor
adresu ide bajt najmanje težin
broja (LSB=least significant by
a na najvišu bajt najviše
Težine (msb=most s.b.)

Smer rasta memorijskih adresa →



Vrste asemblerskih instrukcija

- **Transfer podataka između memorije i registara**
  - Učitavanje podatka iz memorije u registar
  - Čuvanje sadržaja registara u memoriji
- **Obavljajne aritmetičko logičkih operacija nad podacima u registrima ili memoriji**
- **Kontrola toka izvršavanja**
  - Bezuslovni skokovi u/iz potprograma
  - Uslovni skokovi (grananja kontrole)
  - Indirektni skokovi

x86-64 programski model

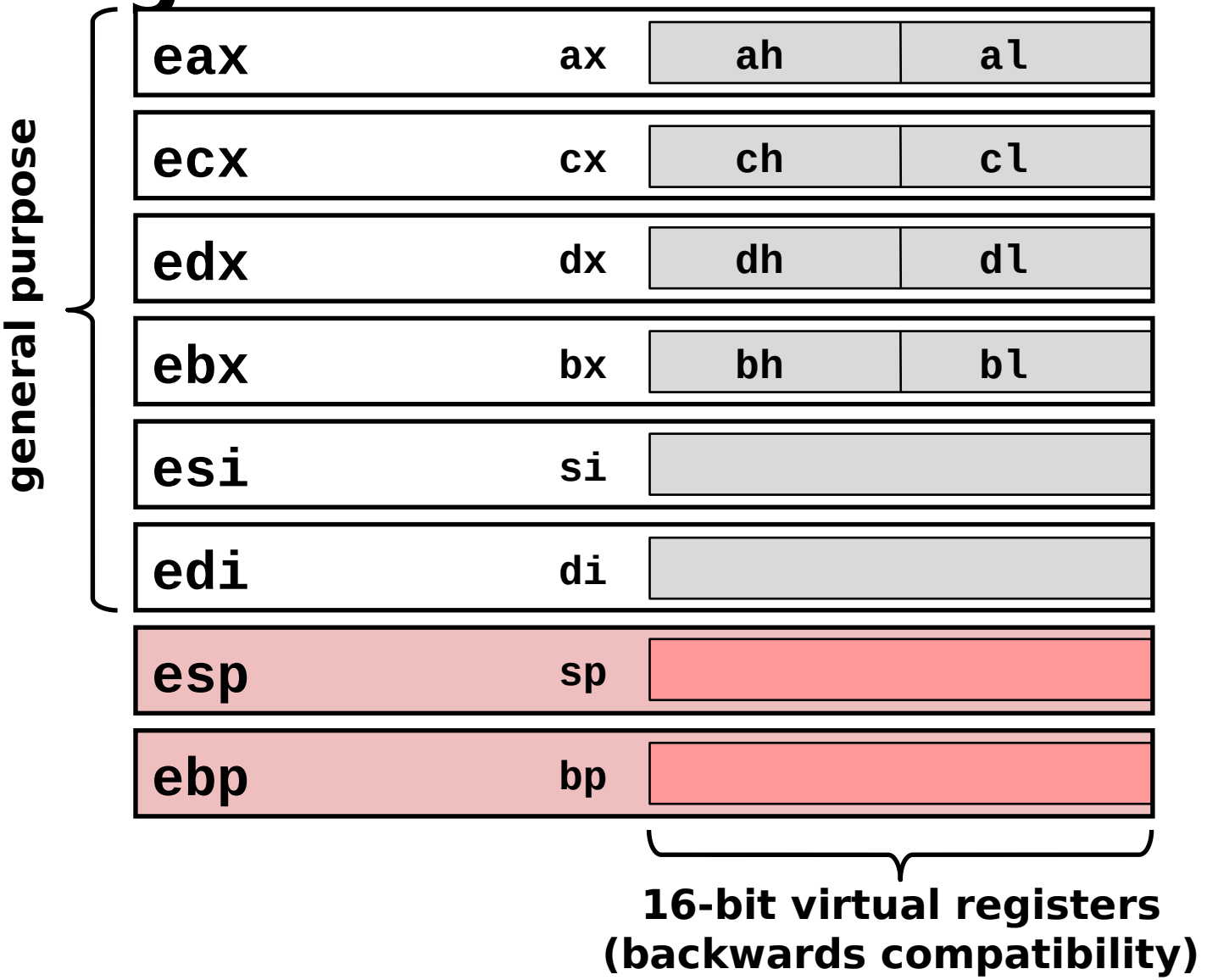
- Može se pristupati nižim delovima dužine 4 bajta (isto nižim delovima od 1 i 2 bajta)

rax	eax	r8	R8d |R8w| R8b
rbx	ebx	r9	r9d |R9w| R9b
rcx	ecx	r10	r10d|R10w| R10b
rdx	edx	r11	r11d|R11w| R11b
rsi	esi	r12	r12d|R12w| R12b
rdi	edi	r13	r13d|R13w| R13b
rsp	esp	r14	r14d|R14w| R14b
rbp	ebp	r15	r15d|R15w| R15b

%rip — Programski brojač

CF ZF SF OF — Kodovi uslova (flegovi)

Istorijski nazivi: IA32 Registri



Transfer podataka

dozvoljene kombinacije mov operanada

	Source	Dest	Dest,Src	C Analog
mov	*Imm*	*Reg*	`mov rax, 0x4`	`temp = 0x4;`
		Mem	`mov QWORD PTR[rax],-147`	`*p = -147;`
	Reg	*Reg*	`mov rdx,rax`	`temp2 = temp1;`
		Mem	`mov QWORD PTR[rdx],rax`	`*p = temp;`
	Mem	*Reg*	`mov rdx,QWORD PTR[rax]`	`temp = *p;`

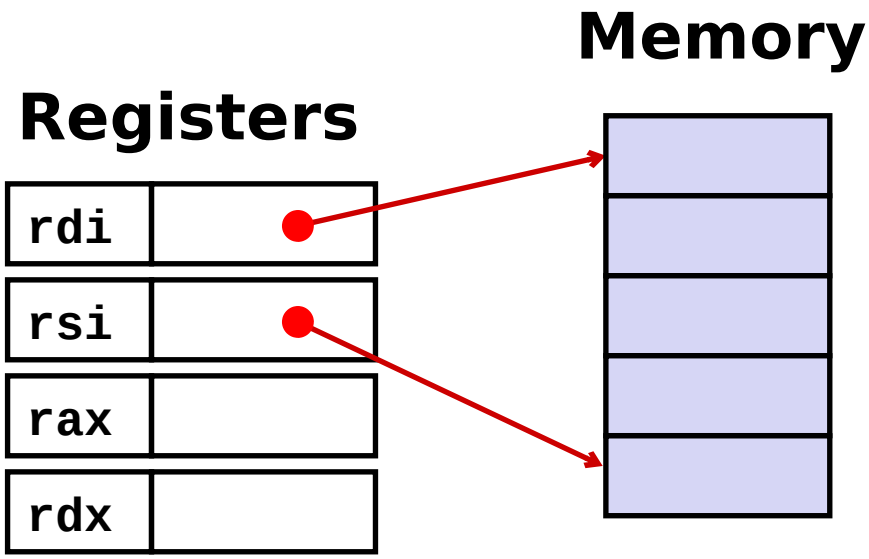
Ne može se raditi transfer iz memorije u memoriju

Načini adresiranja

Type	Form	Operand value	Name
Immediate	**konst , ili OFFSET labela**	*Imm*	Immediate
Register		$R[r_a]$	Register
Memory	**Ra**	$M[Imm]$	Absolute
Memory	**[konstanta] ili Labela**	$M[R[r_a]]$	Indirect
Memory		$M[Imm + R[r_b]]$	Base + displacement
Memory	**[Ra]**	$M[R[r_b] + R[r_i]]$	Indexed
Memory	**Konst[Rb] [Rb,Ri] ili [Rb][Ri]**	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory		$M[R[r_i] \cdot s]$	Scaled indexed
Memory	**Im[Rb+Ri] ili [im+Rb+Ri]**	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory		$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	**[Ri*s] Imm[Ri*S]**	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory RIP rela	**[Rb+Ri*s] Imm[Rb+Ri*s]**	M[*OFFSET labela*]	

Primer za neke načine adresiranja

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
rdi	xp
rsi	yp
rax	t0
rdx	t1

```
swap:
    mov     rax, QWORD PTR [rdi] # t0 = *xp
    mov     rdx, QWORD PTR [rsi] # t1 = *yp
    mov     QWORD PTR [rdi], rdx # *xp = t1
    mov     QWORD PTR [rsi], rax  # *yp = t0
    ret
```

Načini adresiranja

- **Generalni oblik scaled indexed adresiranja**

  D[Rb+Ri*S] Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Konstantni pomeraj “displacement” 1, 2 ili 4 bajta
  - Rb: Bazni registar: bilo koji od 16 celobrojnih registara
  - Ri: Indeksni registar: Bilo koji osim **`rsp`**
  - S: Faktor skaliranja: 1, 2, 4, or 8

- **Specijalni slučajevi**

  [Rb,Ri] Mem[Reg[Rb]+Reg[Ri]]

  D[Rb,Ri] Mem[Reg[Rb]+Reg[Ri]+D]

  [Rb+Ri*S] Mem[Reg[Rb]+S*Reg[Ri]]

Primeri izračunavanja adresa

rdx	0xf000
rcx	0x0100

Expression	Address Computation	Address
0x8(rdx)	0xf000 + 0x8	0xf008
[rdx][rcx]	0xf000 + 0x100	0xf100
[rdx][rcx*4]	0xf000 + 4*0x100	0xf400
0x80[rdx*2]	2*0xf000 + 0x80	0x1e080

Ostale instrukcije transfera

Efekat izvršavanja: DEST ← SignExtend(SRC);

Instrukcija	Opis
MOVSX *r16, r/m8*	*Move byte to word with sign-extension.*
MOVSX *r32, r/m8*	*Move byte to doubleword with sign-extension.*
MOVSX *r64, r/m8*	*Move byte to quadword with sign-extension.*
MOVSX *r32, r/m16*	*Move word to doubleword, with sign-extension.*
MOVSX *r64, r/m16*	*Move word to quadword with sign-extension.*
MOVSXD *r16, r/m16*	*Move word to word with sign-extension.*
MOVSXD *r32, r/m32*	*Move doubleword to doubleword with sign-extension.*
MOVSXD *r64, r/m32*	*Move doubleword to quadword with sign-extension.*

Efekat izvršavanja: DEST ← ZeroExtend(SRC);

Instrukcija	Opis
MOVZX *r16, r/m8*	*Move byte to word with zero-extension.*
MOVZX *r32, r/m8*	*Move byte to doubleword, zero-extension.*
MOVZX *r64, r/m8*	*Move byte to quadword, zero-extension.*
MOVZX *r32, r/m16*	*Move word to doubleword, zero-extension.*
MOVZX *r64, r/m16*	*Move word to quadword, zero-extension.*

Aritmetičke i logičke operacije

Instrukcija za računanje adresa

- **lea Dst, Src**
  - Src je izraz nekog načina adresiranja
  - Postavlja vrednost Dst na adresu određenu izrazom

- **Upotreba**
  - Sračunavanje adresa bez obraćanja memoriji
    - Na primer, za prevođenje C iskaza p = &x[i];
  - Generalno računanje aritmetičkih izraza oblika x + k*y
    - k = 1, 2, 4, or 8
- **C primer:**

```
long m12(long x)
{
  return x*12;
}
```

Kompajlerski prevod na asembler

```
lea rax, [rdi+rdi*2]      # t = x+2*x
sal     rax, 2             # return t<<2
```

Neke aritmetičke i logičke operacije

- **Dvoadresne instrukcije:**

Format	Izračunavanje		
`add`	Dest,Src	Dest = Dest + Src	
`sub`	Dest,Src	Dest = Dest - Src	
`imul`	Dest,Src	Dest = Dest * Src	Označeno množenje
`shl`	Dest,Src	Dest = Dest << Src	Sinonim: sal
`sar`	Dest,Src	Dest = Dest >> Src	Aritmetičko pomeranje
`shr`	Dest,Src	Dest = Dest >> Src	Logičko pomeranje
`xor`	Dest,Src	Dest = Dest ^ Src	
`and`	Dest,Src	Dest = Dest & Src	
`or`	Dest,Src	Dest = Dest | Src	

- **Ne prave razliku između označenih i neoznačenih brojeva**

Neke aritmetičke i logičke operacije (2)

- **Jednoadresne instrukcije**

`inc`	Dest	Dest = Dest + 1
`dec`	Dest	Dest = Dest - 1
`neg`	Dest	Dest = - Dest
`not`	Dest	Dest = ~Dest

- Prešli smo mali podskup x86 instrukcija. Neke ćemo uvesti u narednim lekcijama

- Ukupno ima preko 2000 različitih x86 instrukcija, a broj ide i preko 3000 ako se u obzir uzmu različiti načini adresiranja, veličine operanada, kodovi uslova.

Primeri aritmetičkih izraza

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
    lea  rax, [rdi+rsi]   #t1
    add  rax, rdx         #t2
    lea  rcx, [rsi+rsi*2]
    sal  rcx, 4           #t4
    lea  rdx, 4[rdi+rcx]  #t5
    imul rax, rdx         #rval
    ret
```

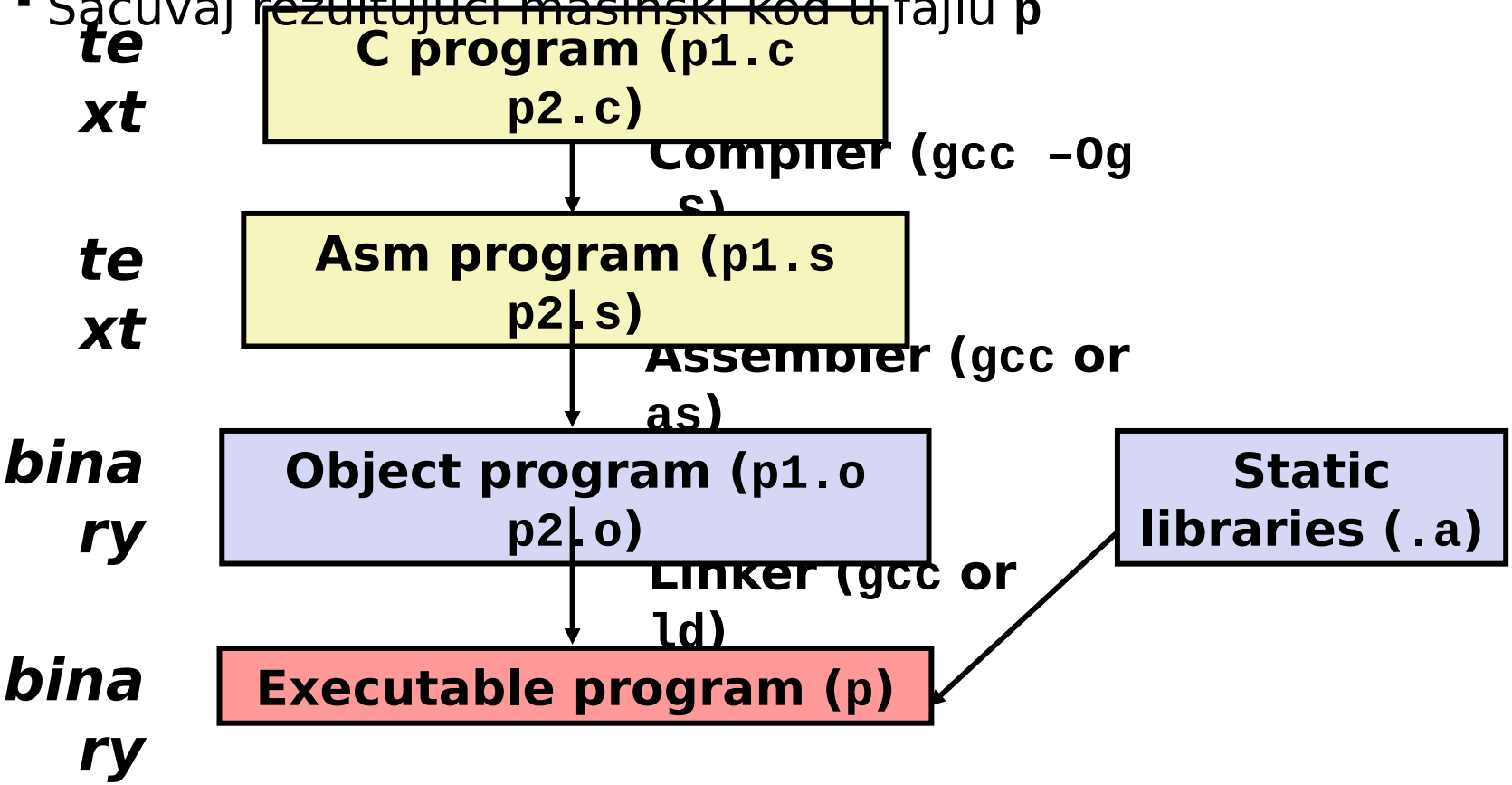
Registar	Upotreba
rdi	Argument x
rsi	Argument y
rdx	Argument z, t5
rax	t1, t2, rval
rcx	t4

Zanimljive instrukcije

- **lea**: adresna aritmetika
- **sal**: pomeranje bitova
- **imul**: množenje

Prevođenje C programa u objektni kod

- Neka je kod u fajlovima **`p1.c p2.c`**
- Pokretanje kompajlera: **`gcc –Og p1.c p2.c -o p`**
  - Koristiti osnovne optimizacije (**`-Og`**) [Noviji gcc-ovi imaju ovu opciju]
  - Sačuvaj rezultujući mašinski kod u fajlu **`p`**



Prevođenje u asembler

C program

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generisani x86-64

```
sumstore:
   push   rbx
   mov    rbx, rdx
   call   plus
   mov  QWORD PTR [rbx],rax
   pop    rbx
   ret
```

Dobija se komandom

`gcc -masm=intel –Og –S –fno-pie sum.c –o sum.s`

Pravi fajl `sum.s`

Rezultati će se razlikovati od prikazanih na drugim Pcevima sa drugim verzijama gcc kompajlera i drugačijim podrazumevanim opcijama prevođenja.

Kako zaista izgleda sum.c

```
 .file   "sum.c"
        .intel_syntax noprefix
        .text
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB0:
        .cfi_startproc
        push    rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        mov     rbx, rdx
        call    plus
        mov     QWORD PTR [rbx], rax
        pop     rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
 .size   sumstore, .-sumstore
        .ident  "GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0"
        .section        .note.GNU-stack,"",@progbits
```

Linije koje počinju sa‘.’ su asemblerske direktive.

```
sumstore:
   push   rbx
   mov    rbx, rdx
   call   plus
   mov  QWORD PTR [rbx],rax
   pop    rbx
   ret
```

Primer mašinske instrukcije

```
*dest = t;
```

- **C kod**
  - Smešta “long” vrednost **t** na lokaciju na koji ukazuje pokazivač **dest**

```
mov  QWORD PTR [rbx], rax
```

- **Asemblerski kod**
  - Kopira 8-bajtnu vrednost u memoriju
    - Quad word u rečniku x86-64
  - Operandi:

    t: Registar **rax**

    dest: Registar **rbx**

    ***dest:** Memorija **M[rbx]**

```
0x40059e:  48 89 03
```

- **Objektni kod**
  - 3-bajtna instrukcija
  - Smeštena na adresu **0x40059e**

Razlike između Intelove i AT&T sintakse

	Intel	AT&T
Instrukcije	Bez sufiksa add	Sufiks veličine operanda: addq
Registri	eax, ebx, …	%eax,%ebx, …
Konstante	0x100	$0x100
Memorijsko indirektno	QWORD PTR [eax]	Veličina operanda sufiks opkoda (%eax)
Skalirano indeksno	[reg + reg * scale + displacement]	displacement(reg, reg, scale)
Redosled operanada	add eax, 4	addl $4, %eax

Literatura

- **Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition**
- **15-213 Introduction to Computer Systems Carnegie Mellon University**