UNIVERZITET METROPOLITAN BEOGRAD

CS120 - ORGANIZACIJA RAČUNARA

Prevođenje programa

Lekcija 12

PRIRUČNIK ZA STUDENTE

CS120 - ORGANIZACIJA RAČUNARA

Lekcija 12

PREVOĐENJE PROGRAMA

UVOD

Digitalni računar je mašina koja ljudima pomaže tako što izvršava zadate instrukcije. Niz instrukcija čijim se izvršavanjem obavlja određeni posao naziva se program.

Iako većina programa može i treba da se piše na jeziku visokog nivoa, postoje situacije u kojima je neophodan asemblerski jezik, bar delimično. Programi za prenosiv računar sa skrornnim resursima kao što su inteligentne kartice, procesori ugrađeni u kućne aparate i bežični prenosivi digitalni pomoćnici, potencijalni su kandidati. Program na asemblerskom jeziku simbolička je predstava programa na mašinskom jeziku.

Iako svaki računara ima drugačiji asemblerski jezik, proces rada asemblera je dovoljno sličan u svima njima da ga je moguće opisati uopšteno.

Upravljačka jedinica pribavlja, dekodira i upravlja izvođenjem instrukcija napisanog asemblerskog programa nakon prevođenja i punjenja memorija. Preko adresne, sabirnice podataka i kontrolne sabirnice komunicira sa svim komponentama mikroprocesora i mikroračunara.

Nivo asemblera sadrži mašinski jezik iz nižih nivoa preveden u skup tekstualnih simbola. Ovaj nivo pruža ljudima mogućnost da pišu programe za nivoe 1, 2 i 3 u obliku koji nije tako neprijatan kao jezici virtuelnih mašina. Programi napisani u asembleru najpre se prevode na jezik nivoa 1, 2 ili 3, a zatim ih interpretira odgovarajuća virtuelna mašina ili stvarni računar. Program koji obavlja prevođenje zove se **asembler** (engl. assembler).

Kao i u svakom drugom programskom jeziku, program u asembleru je sačinjen od niza uzastopno navedenih instrukcija. Stoga bi se moglo pretpostaviti da je svaku instrukciju dovoljno pročitati i prevesti u mašinski jezik u samo jednom prolazu kroz program.

Drugi prolaz prevođenja generiše objektni kod, prikazuje informacije koje su neophodne za proces **povezivanja** (linker) i opciono prikazuje listing prevođenja. **Proces povezivanja** (engl. linking) je usko povezan sa načinom organizacije softvera.

Treba istaći razliku između prevođenja i interpretiranja. Pri prevođenju se originalni program napisan na izvornom jeziku ne izvršava direktno. On se najpre pretvara u ekvivalentan program koga zovemo **objektni** program (engl. object program) ili **izvršni binarni program** (engl. executable binary program), koji se izvršava tek kada je prevođenje završeno.

Programi se pišu na asemblerskom jeziku a ne na mašinskom (u obliku heksa-decimalnih brojeva) zato što je na asemblerskom jeziku mnogo lakše programirati.

Poglavlje 1

DVOPROLAZNI ASEMBLER

PRVI PROLAZ PREVOĐENJA ASEMBLERSKOG KODA

Kao i u svakom drugom programskom jeziku, program u asembleru je sačinjen od niza uzastopno navedenih instrukcija.

Pošto se program na asemblerskom jeziku sastoji od niza jednorednih naredbi, na prvi pogled bi bilo logično da asembler učita jednu naredbu, prevede je na mašinski jezik i uputi je u datoteke za čuvanje ili za štampanje listinga.

Opisani postupak bi se ponavljao, red pored sve dok se ne prevede ceo program. Nažalost, ovakva strategija je pogrešna.

Razmotrićemo slučaj kada je prva naredba skok na naredbu L.

Asembler ne može da prevede ovu naredbu sve dok ne sazna adresu naredbe L. Naredba L se može nalaziti pri kraju programa, pa asembler ne može da sazna njenu adresu dok ne pročita skoro čitav program.

Ovaj problem se naziva **problem zbog referenciranja unapred** (engl. forward reference problem) jer je simbol L upotrebljen pre nego što je definisan, drugim rečima referenciran je simbol čija definicija će tek uslediti.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Problem referenci unapred se može rešiti na više načina. Jedan pristup je prevođenje koda u dva prolaza.

U prvom prolazu se registruju definicije simbola, labela i makroa i njihove vrednosti se smeštaju u odgovarajuće tabele. Na početku drugog prolaza, vrednosti svih simbola su poznate, pa se problem reference unapred ne može pojaviti. U drugom prolazu se sve linije koda čitaju, prevode i upisuju u objektnu datoteku.

Ovakav način prevođenja je konceptualno jednostavan, ali zahteva dodatni prolaz kroz kod.

PRVI PROLAZ PREVOĐENJA ASEMBLERSKOG KODA,

Osnovna funkcija prvog prolaza prevođenja je formiranje tabele simbola.

Na prvom prolazu, definicije simbola, uključujući oznake iskaza, se prikupljaju i čuvaju u tabeli. Do trenutka kada započne drugi prolaz, vrednosti svih simbola su poznate; tako da ne ostaje nikakva referenca unapred i svaka naredba se može pročitati, sastaviti i izvesti.

Iako ovaj pristup zahteva dodatni prelaz preko ulaza, konceptualno je jednostavan.
Iako različite hardverske platforme poseduju različite implementacije asemblera, proces prevođenja asemblerskog koda u mašinski jezik je dovoljno sličan da se može opisati opštim principima.

Program pisan u asembleru je sačinjen od niza uzastopno navedenih instrukcija. Pretpostavlja se da je svaku instrukciju dovoljno pročitati i prevesti u mašinski jezik u samo jednom prolazu kroz program, ali to nije slučaj.
Postojanje simboličkih imena, labela, makro definicija i operacija skokova u sintaksi asemblera zahteva da se pri prevođenju prođe više puta kroz kod programa.
Primer: Program koji počinje instrukcijom skoka na labelu **A_Dest**.

```
__asm
{
jmp A_Dest
...
}
```

Ova vrsta problema je poznata kao **problem reference unapred** (engl. forwardreference problem) i označava situaciju u kojoj se simbol koristi pre nego što je definisan.
Da bi se program mogao prevesti neophodno je poznavati konkretne vrednosti svih promenljivih.

Drugi pristup se sastoji od jednokratnog čitanja asemblerskog programa, pretvaranja u među oblik koji se smešta u tabeli u memoriji. Zatim se vrši drugi prolaz preko tabele umesto preko izvornog programa. Ako ima dovoljno memorije (ili virtuelne memorije), ovim pristupom se štedi vreme potrebno za ulazno-izlazne operacije.

Bez obzira na pristup , u prvom prolasku asemblera snimaju se sve sve makro definicije i svi pozivi makroima se proširuju. Tako se definisanje simbola i proširenje makroa generalno kombinuju u jedan prolasku.

Osnovna funkcija prvog prolaza prevođenja je formiranje tabele simbola. Pod simbolom se podrazumeva vrednost labele ili vrednost kojoj je dodeljeno simboličko ime, upotrebom pseudo instrukcije.

PRVI PROLAZ

Osnovna funkcija prvog prolaza prevođenja je formiranje tabele simbola. Tabela simbola sadrži zaseban unos za svaki simbol.

Na primer, pseudo instrukcijom

BUFSIZE EQU 1234

se vrednosti 1234 dodeljuje simboličko ime BUFSIZE. U prvom prolazu se moraju registrovati i mesta u kodu na kojima su definisane labele.

Primeri definicija nekoliko labela (MARIA, ROBERTA, MARILYN i STEPHANY) su dati na slici 1.

Labele su značajne jer često predstavljaju odredišta skokova.
Za potrebe praćenja adresa instrukcija u toku izvršenja, asembler definiše promenljivu ***ILC*** (engl. Instruction Location Counter) ili ***PC*** (engl. program counter). Na početku programa ova promenljiva je jednaka nuli, a nakon svake instrukcije se inkrementira za dužinu te instrukcije.

Labela	Opkod	Operandi	Komentari	Dužina	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	**skoči na** DONE	5	129

Slika 1.1 Definisanje labela u asemblerskom programu [Izvor: Autor]

TABELA SIMBOLA

U prvom prolazu većina asemblera koriste interne tabele.

U prvom prolazu većina asemblera koristi tri interne tabele:

- **tabelu simbola**
- **tabelu pseudo instrukcija**
- **tabelu operacionih kodova**

Po potrebi se koristi i tabela literala. Literali su konstante za koje asembler automatski rezerviše memoriju.

Tabela simbola sadrži zaseban unos za svaki simbol, kao što je prikazano na slici 2.

Svaki unos sadrži ime simbola, njegovu vrednost i polje za ostale informacije. Ostale informacije mogu biti dužina podatka pridruženog simbolu, bitovi za relokaciju i odrednica da li je simbol vidljiv za van procedure. Pojam relokacije će biti kasnije objašnjen.
Tabela simbola se koristi u fazi drugog prolaza prevođenja. Jedan od bitnih činilaca tabele simbola je efikasnost pretrage. Struktura podataka u tabeli odgovara asocijativnoj memoriji, odn. parovima simbol – vrednost.

Najjednostavnija implementacija ove strukture podataka je formiranje niza parova simbol – vrednost, ali se takva implementacija najsporije pretražuje.

Labela	Opkod	Operandi	Komentari	Dužina	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	**skoči na** DONE	5	129

Slika 1.2 Izgled tabele simbola [Izvor: Autor]

TABELA OPERACIONIH KODOVA

Tabela operacionih kôdova sadrži bar jedan unos za svaku asemblersku instrukciju.

Drugi način je formiranje sortirane tabele i upotreba efikasnog algoritma za pretragu. Takvo rešenje zahteva održavanje sortirane tabele. Konačno tabela se može ostvariti tehnikom **heširanja.**

Tehnika podrazumeva upotrebu heš funkcije koja preslikava ukupan broj simbola (n) u k grupa mogućih heš vrednosti. Ako je vrednost k bliska vrednosti n, svaki simbol će u proseku biti pronađen u jednom koraku pretrage. Smanjivanjem vrednosti k, smanjuje se i veličina tabele po cenu dužeg vremena pretrage.

Tabela operacionih kodova sadrži bar jedan unos za svaku asemblersku instrukciju, kao na slici 3.

U tabeli je opisan broj i vrsta operanada instrukcije, heksadecimalni operand instrukcije u mašinskom jeziku, dužina instrukcije i klasa instrukcije.

Na primer, prva definicija instrukcije ***ADD*** objašnjava da je prvi operand registar ***EAX***, drugi operand 32-bitna konstanta, da je mašinski operand instrukcije 0x05, dužina instrukcije 5 bajtova, a klasa instrukcije 4.

Labela	Opkod	Operandi	Komentari	Dužina	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	**skoči na** DONE	5	129

Slika 1.3 Izgled tabele operacionih kôdova [Izvor: Autor]

FUNKCIJE PRVOG PROLAZA

Na kraju prvog prolaza prevođenja se može obaviti uređivanje i provera dvostrukih unosa u tabeli simbola.

Pseudo kod prvog prolaza prevođenja je dat na slici 4.

Pseudo kod daje potpun uvid u sve funkcije prvog prolaza:

- dodavanje simbola i labela u tabelu
- nalaženje formata i dužine instrukcije u tabeli instrukcija ili pseudo instrukcija
- upisivanje podataka u datoteku koju će koristiti drugi prolaz
- ažuriranje ILC promenljive
- pripremu za drugi prolaz prevođenja

Na kraju prvog prolaza prevođenja se može obaviti uređivanje i provera dvostrukih unosa u tabeli simbola.

```
public static void pass_one( ) {
   // This procedure is an outline of pass one of a simple assembler.
   boolean more_input = true;                  // flag that stops pass one
   String line, symbol, literal, opcode;       // fields of the instruction
   int location_counter, length, value, type;  // misc. variables
   final int END_STATEMENT = –2;               // signals end of input

   location_counter = 0;                       // assemble first instruction at 0
   initialize_tables( );                       // general initialization

   while (more_input) {                        // more_input set to false by END
      line = read_next_line( );                // get a line of input
      length = 0;                              // # bytes in the instruction
      type = 0;                                // which type (format) is the instruction

      if (line_is_not_comment(line)) {
         symbol = check_for_symbol(line);      // is this line labeled?
         if (symbol != null)                   // if it is, record symbol and value
            enter_new_symbol(symbol, location_counter);
         literal = check_for_literal(line);    // does line contain a literal?
         if (literal != null)                  // if it does, enter it in table
            enter_new_literal(literal);

         // Now determine the opcode type.  –1 means illegal opcode.
         opcode = extract_opcode(line);        // locate opcode mnemonic
         type = search_opcode_table(opcode); // find format, e.g. OP REG1,REG2
         if (type < 0)                         // if not an opcode, is it a pseudoinstruction?
            type = search_pseudo_table(opcode);
         switch(type) {                        // determine the length of this instruction
            case 1: length = get_length_of_type1(line);  break;
            case 2: length = get_length_of_type2(line);  break;
            // other cases here
         }
      }
      write_temp_file(type, opcode, length, line);   // useful info for pass two
      location_counter = location_counter + length;      // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         rewind_temp_for_pass_two( );          // like rewinding the temp file
         sort_literal_table( );                // and sorting the literal table
         remove_redundant_literals( );         // and removing duplicates from it
      }
   }
}
```

Slika 1.4 Pseudokôd prvog prolaza prevođenja asemblerskog programa [Izvor: Autor]

DRUGI PROLAZ PREVOĐENJA

Drugi prolaz prevođenja generiše objektni kôd, prikazuje informacije koje su neophodne za proces povezivanja (linker) i opciono prikazuje listing prevođenja.

Drugi prolaz prevođenja generiše objektni kod, prikazuje informacije koje su neophodne za proces povezivanja (linker) i opciono prikazuje listing prevođenja.

Zbog veće preglednosti i konciznosti, funkcije drugog prolaza su prikazane pseudo kodom na slici 5.

Ključnu funkciju u drugom prolazu obavljaju funkcije koje su u pseudo kodu nazvane **eval_type1, eval_type2**, itd. One prevode asemblerske instrukcije u mašinski jezik, a rezultat vraćaju u promenljivu *code* i upisuju je u datoteku sa objektnim kodom.

Svi koraci prevođenja koji su opisani na slikama 4 i 5 podrazumevaju da u asemblerskom kodu nisu pravljene greške. Naravno, ova pretpostavka nije realna.

Realno prevođenje poseduje procedure za uočavanje ili ispravljanje čestih tipova grešaka, kao što su upotreba nedefinisanih simbola, upotreba nedefinisanih instrukcija i formata instrukcija, neregularna upotreba sistemskih resursa, itd.

```
public static void pass_two( ) {
   // This procedure is an outline of pass two of a simple assembler.
   boolean more_input = true;                  // flag that stops pass two
   String line, opcode;                        // fields of the instruction
   int location_counter, length, type;         // misc. variables
   final int END_STATEMENT = –2;               // signals end of input
   final int MAX_CODE = 16;                    // max bytes of code per instruction
   byte code[ ] = new byte[MAX_CODE];          // holds generated code per instruction

   location_counter = 0;                       // assemble first instruction at 0

   while (more_input) {                        // more_input set to false by END
      type = read_type( );                     // get type field of next line
      opcode = read_opcode( );                 // get opcode field of next line
      length = read_length( );                 // get length field of next line
      line = read_line( );                     // get the actual line of input

      if (type != 0) {                         // type 0 is for comment lines
         switch(type) {                        // generate the output code
            case 1: eval_type1(opcode, length, line, code);  break;
            case 2: eval_type2(opcode, length, line, code);  break;
            // other cases here
         }
      }

      write_output(code);                      // write the binary code
      write_listing(code, line);               // print one line on the listing
      location_counter = location_counter + length;      // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         finish_up( );                         // odds and ends
      }
   }
}
```

Slika 1.5 Pseudokôd drugog prolaza prevođenja asemblerskog programa [Izvor: Autor]

Poglavlje 2

POVEZIVANJE I UČITAVANJE

POVEZIVANJE I UČITAVANJE,

Programi uglavnom ne sadrze samo jednu. već više procedura.

Većina programa se sastoji od više od jedne procedure.

Prevodioci i asembleri uglavnom prevode jednu po jednu proceduru i stavljaju prevedeni izlaz na disk.

Pre nego što se program pokrene, sve prevedene procedure moraju biti pronađene i pravilno povezane zajedno. Ako virtuelna memorija nije dostupna, povezani program se takođe mora eksplicitnoučitati u glavnu memoriju.

Programi koji obavljaju ove funkcije nazivaju se različitim imenima, uključujući **linker, učitavač povezivanja i uređivač povezivanja**.

Kompletan prevod izvornog programa zahteva dva koraka:

1. Prevođenje ili asembliranje izvornih datoteka.

2. Povezivanje objektnih modula.

Prevod iz izvorne procedure u objektni modul predstavlja promenu nivoa jer izvorni jezik i ciljni jezik imaju različite instrukcije i sintaksu.

Proces povezivanja, međutim, ne predstavlja promenu nivoa, pošto su i ulaz u ovaj program i njegov rezultat programi za istu virtuelnu mašinu.

Uloga programa za povezivanje je da prikupi procedure koje su nezavisne i da ih međusobno poveže u jedinstvenu celinu zvanu **izvršni binarni program** (engl. executable binary program)

PROCES POVEZIVANJA

Proces povezivanja (engl. linking) je usko povezan sa načinom organizacije softvera. Kao i u višim programskim jezicima i u asembleru postoji potreba da se softver piše modularno.

Proces povezivanja (engl. linking) je usko povezan sa načinom organizacije softvera. Kao i u višim programskim jezicima i u asembleru postoji potreba da se softver piše modularno.

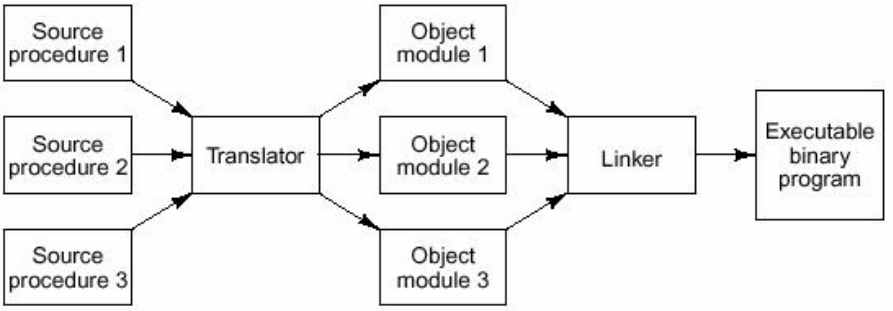
Program se deli u nekoliko zasebnih procedura koje obavljaju relativno nezavisne poslove. Ovakav način organizacije čini kod lakšim za održavanje. Takođe, tako napisani moduli se mogu skoro bez izmena preneti u druge softverske projekte.

Suština procesa povezivanja je prikazana na desnom delu slike 1.

U procesu prevođenja svaka izvorna procedura generiše zaseban objektni modul. Zadatak programa za povezivanje, linkera, je da poveže objektne module u jedinstven izvršni program.

Potrebno je naglasiti da linker ne vrši nikakvo prevođenje, jer je objektni kod već sačinjen od mašinskih instrukcija.
Povezivanje modula asemblerskog programa omogućava da se pri izmeni jednog dela softvera ne prevodi ceo program iz početka.

Da bi se dobio izvršni program potrebno je ponovo prevesti samo proceduru u kojoj je došlo do izmene i izvršiti ponovno povezivanje.



Slika 2.1 Proces prevođenja i povezivanja asemblerskog programa [Izvor: Autor]

Proces povezivanja se gotovo uvek obavlja brže od procesa prevođenja

ZADACI KOJI SE OBAVLJAJU U PROCESU POVEZIVANJA

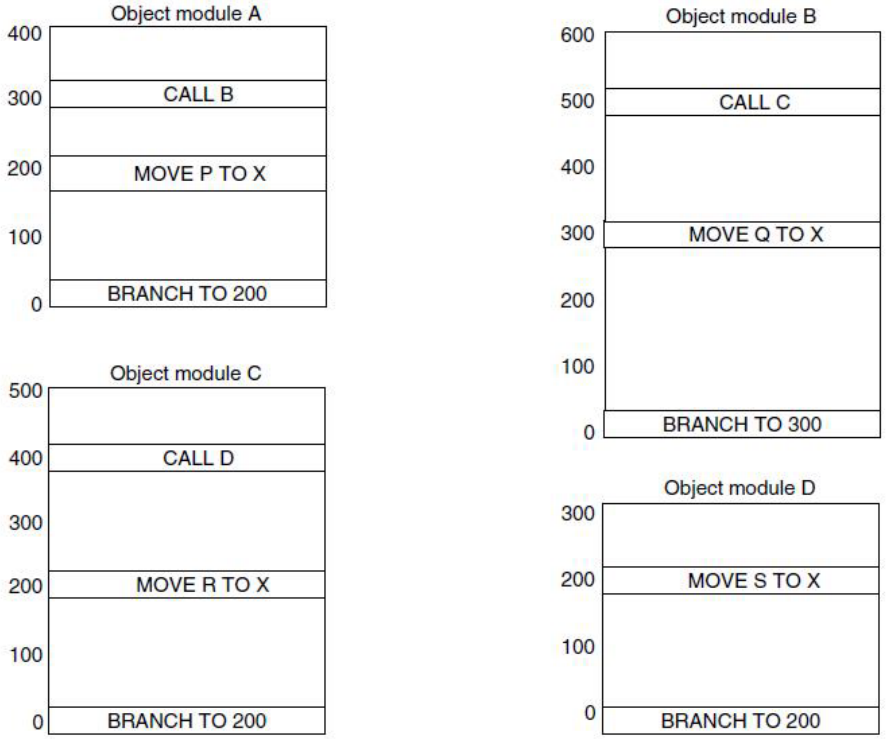
Proces povezivanja se gotovo uvek obavlja brže od procesa prevođenja.

Kao što je prikazano na slikama 4 i 5 u prethodnom objektu učenja, ILC se na početku svake faze prevođenja postavlja na vrednost 0.

To znači da se za svaki modul objektnog prevođenja; da pretpostavlja da će biti postavljan na početnu adresu 0 pre početka izvršenja.

Na slici 2 su prikazana četiri objektna modula nastala kao proizvod procesa prevođenja.

Da bi se program sačinjen od ovakvih modula mogao izvršiti, linker dovodi objektne module u radnu memoriju i formira sliku izvršnog kôda.



Slika 2.2 Objektni moduli nastali kao rezultat procesa prevođenja (svi moduli su na početnoj adresi 0) [Izvor: Autor]

PROBLEM RELOKACIJE

Takođe se može primetiti da su moduli zadržali odredišta instrukcije skoka, iako je adresni prostor promenjen. Ovo predstavlja problem relokacije.

Da bi se program sačinjen od ovakvih modula mogao izvršiti, linker dovodi objektne module u radnu memoriju i formira sliku izvršnog modula kao na slici 3(a).

Kao što se vidi na slici, program često ne počinje od virtuelne adrese 0. Na niskim adresama memorijskog adresnog prostora se često nalaze vektori prekida, komunikacija sa operativnim sistemom i sl.

Takođe se može primetiti da su moduli zadržali odredišta instrukcije skoka, iako je adresni prostor promenjen.

Ovo predstavlja problem relokacije.
Potreba za relokacijom je posledica činjenice da svaki objektni modul poseduje zaseban adresni prostor. Na hardverskim platformama sa segmentiranim adresnim prostorom ovo ne bi trebalo da predstavlja problem. Svaki objektni modul bi mogao da se postavi u zaseban

segment i ne bi bilo potrebe za prevođenjem adresa.

Problem ipak postoji zato što popularni operativni sistemi Windows i UNIX podržavaju samo jedinstven linearni adresni prostor.

Stoga svi objektni moduli moraju biti spojeni u jedinstven adresni prostor.

PROCES POVEZIVANJA KOD RELOKACIJE

Proces povezivanja prevodi program iz stanja pre relokacije u stanje posle relokacije

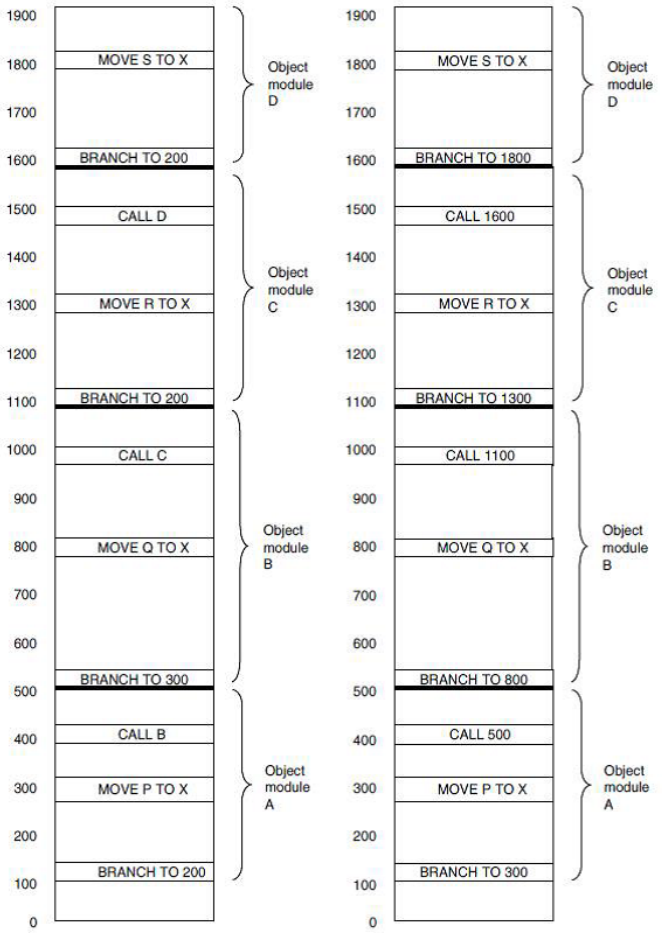
Na slici 3(a) takođe postoji problem eksternih referenci. Problem nastaje kada se unutar jedne procedure pozove neka druga procedura koja je prevedena u zasebnom objektnom modulu.

Adresa skoka na tu proceduru ne može biti poznata dok proces povezivanja ne pronađe i ne poveže sve module u jedinstveni adresni prostor, odn. u jedinstveni izvršni program.

Proces povezivanja se odvija u nekoliko koraka:

1. Kreira se tabela svih objektnih modula i njihovih dužina
2. Na osnovu ove tabele, svakom objektnom modulu se dodeljuje početna adresa
3. Pronalaze se sve instrukcije koje referenciraju memoriju i svakoj se dodaje konstanta relokacije, jednaka novoj početnoj adresi odgovarajućeg modula
4. Pronalaze se sve instrukcije koje referenciraju druge procedure i u njih se upisuju nove adrese na kojima se te procedure nalaze

Nakon ovih koraka, proces povezivanja prevodi program iz stanja na slici 3(a) u stanje prikazano na slici 3(b).



Slika 2.3 (a) Izgled objektnih modula u memoriji pre relokacije,(b) izgled objektnih modula u memoriji [Izvor: Autor]posle relokacije

Poglavlje 3

OBJEKTNI MODUL

OBJEKTNI MODULI

Objektni moduli su rezultati procesa prevođenja. Svaki objektni modul je uobičajeno sadrži šest celina.

Proces povezivanja se često obavlja udvaprolaza,kao iproces prevođenja.

U prvom prolazu se čitaju svi objektni moduli i formiraju se tabela imena i dužina modula i globalna tabela simbola, koja sadrži sve ulazne tačke i eksterne reference.

U drugom prolazu se objektni moduli čitaju, relociraju i jedan po jedan linkuju.

Objektni moduli su rezultati procesa prevođenja. Svaki objektni modul je uobičajeno sadrži šest celina (Slika 1):

1. **identifikaciju**
2. **ulaznu tabelu**
3. **tabelu eksternih referenci**
4. **deo za mašinske instrukcije i konstante**
5. **rečnik relokacije**
6. **polje završetka modula**

Većina ovih celina služi kako potpora radu linkera.

Slika 3.1 Struktura objektnog modula [Izvor: Autor]

Blok za identifikaciju sadrži informacije o imenu modula, elementima potrebnim za rad linkera (npr. dužine navedenih celina unutar modula)i ime prevedenog modula.

STRUKTURA OBJEKTNOG MODULA

U drugom prolazu se objektni moduli čitaju, relociraju i jedan po jedan linkuju.

U ulaznoj tabeli se nalazi lista naziva i vrednosti simbola definisanih u modulu kojima se može pristupati iz drugih modula.

Na primer ako u prevedenom modulu postoji procedura nazvana vidljivost kojoj bi drugi moduli trebalo da pristupaju, ulazna tabela će imati unos koji će se zvati vidljivost i imaće vrednost adrese na kojoj se navedena procedura nalazi.

Programer odlučuje koji će se simboli naći u ulaznoj tabeli, tako što u definiciji simbola navede pseudoinstrukciju ***PUBLIC***.

Funkcija tabele eksternih referenci je suprotna funkciji ulazne tabele.

Tabela eksternih referenci sadrži imena simbola definisanih u drugim objektnim modulima kao i listu koja navodi koje mašinske instrukcije koriste koje simbole.

Ova lista je neophodna za rad linkera, koji ubacuje tačne adrese eksternih simbola u odgovarajuće mašinske instrukcije.

Da bi se koristile procedure iz drugih objektnih modula, mora se navesti pseudoinstrukcija ***EXTERN*** u pozivu date procedure.

VEZIVANJE I DINAMIČKA RELOKACIJA

Pojam vezivanja (engl. binding) se odnosi na rad u više programskim okruženjima i na ažuriranje realne početne adrese programa koji stupa u fazu izvršenja.

Deo za mašinske instrukcije i konstante sadrži preveden kod.

Ovo je jedini blok objektnog modula koji se učitava u memoriju u toku izvršenja, dok ostali služe samo kao podrška procesu povezivanja.
Rečnik relokacija sadrži informacije o tome koje adrese treba relocirati.
Polje završetka modula sadrži adresu od koje počinje izvršenje i često kontrolnu sumu kao indikator da li je modul dobro pročitan.

Pojam vezivanja (engl. binding) se odnosi na rad u višeprogramskim okruženjima i na ažuriranje realne početne adrese programa koji stupa u fazu izvršenja.

U više-programskom okruženju svaki program dobija određen vremenski interval za izvršenje. Nakon toga se taj program povlači iz radne memorije, a na izvršenje dolazi drugi program. Kada ponovo dođe vreme za izvršenje posmatranog programa, on verovatno neće biti učitan na isto mesto u memoriji.

To znači da će reference adresa, koje su formirane u procesu povezivanja, biti pogrešne.

Problem bi se mogao rešiti ponovnim povezivanjem i relokacijom, što bi znatno usporilo rad sistema.

VEZIVANJE I DINAMIČKA RELOKACIJA,

U realnim okruženjima, linker sakuplja adresne prostore različitih modula u jedinstven linearni adresni prostor.

U realnim okruženjima, problem se rešava izborom vremena kada će se obaviti konačno vezivanje simboličnih vrednosti u fizičke adrese.

Završetak ovog procesa se naziva **vreme vezivanja** (engl. bindingtime) i u opštem slučaju se može desiti u jednom od sledećih trenutaka:

- **kada se program piše**
- **kada se program prevodi**
- **kada se program povezuje, ali pre nego što se učita**
- **kada se program učita**
- **kada se učita bazni registar za adresiranje**
- **kada se izvršava instrukcija, koja sadrži adresu**

Proces preslikavanja adresnih prostora se može podeliti u dva dela.

- U prvom koraku se imena simbola povezuju sa virtuelnim adresama
- Drugi korak podrazumeva vezivanje virtuelnih adresa sa fizičkim adresama.

U realnim okruženjima, linker sakuplja adresne prostore različitih modula u jedinstven linearni adresni prostor.
Zbog funkcionisanja više-programskog okruženja, odn. potrebe da se programi izvršavaju naizmenično, linker ustvari kreira virtuelni adresni prostor.

Poglavlje 4

DINAMIČKO POVEZIVANJE

DINAMIČKO POVEZIVANJE

U procesu dinamičkog povezivanja (engl dynamic linking) ovakve celine se povezuju u trenutku kada su potrebne, odn. u trenutku prvog poziva u nekom od programa.

Da bi se ostvarilo **više-programsko okruženje** (engl. multi-programing) neophodno je ostvariti prevođenje virtuelnih adresa u fizičke adrese. Ovaj proces se može realizovati na nekoliko načina.

Prvi način je straničenje (engl. paging) memorije, koje podrazumeva da se pri učitavanju programa glavnu memoriju, menja samo njegova **tabela stranica** (engl. page table), a ne ceo program.

Drugi način prevođenja virtuelnih u fizičke adrese je upotreba relokacionog registra. Platforme koje koriste ovaj pristup poseduju registar koji uvek pokazuje na memorijsku adresu početka tekućeg programa. Stoga se pre učitavanja programa u memoriju obavlja sabiranje virtuelnih memorijskih lokacija sa relokacionim registrom i tako se dobijaju fizičke adrese.

Proces je neprimetan za korisničke programe i obavlja se u hardveru. Naravno, neophodno je postojanje mehanizma ažuriranja relokacionog registra.

U odnosu na straničenje, upotreba relokacionog registra je restriktivnija jer zahteva da se program pomera u celini.

Treći mehanizam je moguć na platformama koje se obraćaju memoriji posredstvom **programskog brojača** (engl. program counter). Kada se program učita u memoriju neophodno je samo prilagoditi vrednost programskog brojača. U ovom slučaju se kaže da je program nezavistan od pozicije.Njegove procedure mogu biti postavljene bilo gde u virtuelnom adresnom prostoru be zpotrebe za relokacijom. Sve fizičke adrese takvog programa su ili zavisne od vrednosti rogramskog brojača ili su apsolutne.

Na primer, registri za ulaz i izlaz uređaja se nalaze na apsolutnim adresama.

Mnogi programi imaju procedure koje se pozivaju samo u sasvim ne uobičajenim okolnostima. Na primer,prevodioci imaju procedure za prevođenje retko korišćenih naredbi kao i procedure za obradu grešaka koje se retko događaju.

Fleksibilniji način povezivanja nezavisno prevedenih procedura bilo bi povezivanje svake procedure u trenutku kada se prvi put pozove. Ovaj postupak je poznat kao dinamicko povezivanje.

Prvi put je uveden u operativnom sistemu **MULTICS** i ta implcmentacija je i danas u izvesnom smislu neprevaziđena.

DINAMIČKO POVEZIVANJE NA RAZLIČITIM PLATFORMAMA

Dinamičko povezivanje omogućuje uštedu memorije i prostora na disku, kao i lakše održavanje softvera.

U **MULTICS** obliku dinamičkog povezivanja, svakom programu je pridružen segment, nazvan segment povezivanja, koji sadrži jedan blok informacija za svaku proceduru koja se može pozvati.

Ovaj blok informacija počinje rečju rezervisanom za virtuelnu adresu procedure, a prati je ime procedure koje se čuva kao niz znakova..

Kada se koristi dinamičko povezivanje, pozivi procedura u izvornom jeziku se prevode u instrukcije koje indirektno adresiraju prvu reč odgovarajućeg bloka povezivanja.
Prevodilac popunjava ovu reč neispravnom adresom ili posebnim nizom bitova koji izazivaju upadanje programa u klopku.

Kada se pozove procedura u drugom segmentu, pokušaj adresiranja nevažeće reči indirektno izaziva klopku za dinamički linker.

Povezivač zatim pronalazi niz znakova u reči koja sledi nevažeću adresu i pretražuje korisnički direktorijum datoteka u potrazi za kompajliranom procedurom sa ovim imenom.

Toj proceduri se zatim dodeljuje virtuelna adresa, obično u sopstvenom privatnom segmentu, i ova virtuelna adresa zamenjuje nevažeću adresu u segmentu veze.

Zatim, instrukcija koja je izazvala grešku ponovo se izvršava, dozvoljavajući programu da nastavi sa mesta gde je bio pre klopke.

Svako sledeće pristupanje proceduri proći će bez greške jer indirektna reč sada sadrži ispravnu virtuelnu adresu. Prema tome program za dinamičko povezivanje pokreće se samo kada se procedura prvi put pozove.

DINAMIČKO POVEZIVANJE NA RAZLIČITIM PLATFORMAMA ,

Dinamičko povezivanje omogućuje uštedu memorije i prostora na disku,

Na platformama koje omogućavaju upotrebu virtuelne memorije, obavljanje potpunogpovezivanja pre početka izvršenja ne pruža sve mogućnosti koje nudi virtuelna memorija.

Mnogi programi sadrže procedure koje se relativno retko pozivaju.

Takođe, potreba za ponovnim korišćenjem koda zahteva formiranje bibliotečkih procedura, koje se mogu koristiti u više programa.

U oba slučaja, takve procedure se mogu prevesti i povezati kao zasebna celina.

U procesu **dinamičkog povezivanja** (engl. dynamic linking) ovakve celine se povezuju u trenutku kada su potrebne, odn. u trenutku prvog poziva u nekom od programa.

Dinamičko povezivanje omogućuje uštedu memorije i prostora na disku, kao i lakše održavanje softvera.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 5

DINAMIČKO POVEZIVANJE NA RAZLIČITIM OS

DINAMIČKO POVEZIVANJE NA WINDOWS OPERATIVNOM SISTEMU

Windows operativni sistemi intenzivno koriste koncept dinamičkog povezivanja.

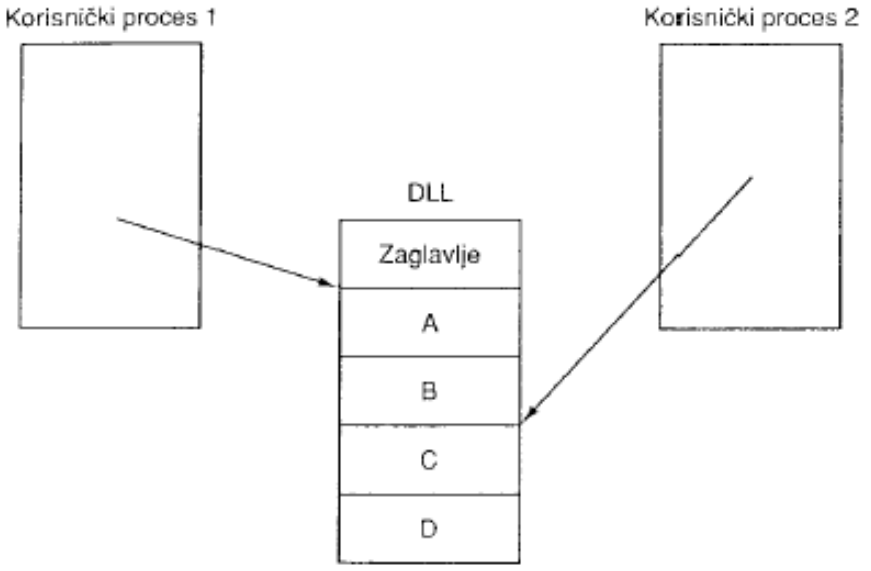
Windows operativni sistemi intenzivno koriste koncept dinamičkog povezivanja. U tu svrhu se koristi poseban format datoteke nazvan datoteka dinamičkog povezivanja ili DLL.

DLL može da sadrži i podatke i procedure. Najčešća forma DLL-a je sačinjena od niza procedura kojima može pristupiti nekoliko procesa (Slika 1).

Formiranje DLL-a je veoma slično procesu prevođenja i povezivanja izvršnog programa.

Ključna razlika u odnosu na izvršni program je to što se DLL ne može nezavisno izvršavati. Pored toga, svaki DLL sadrži nekoliko specifičnih procedura, koje alociraju ili dealociraju memoriju i upravljaju drugim resursima neophodnim za funkcionisanje DLL-a.

Primer specifičnosti DLL-a su procedure, koje se automatski pozivaju kada se novi program veže ili odvoji od DLL-a.



Slika 5.1 Forma DLL-a i mogućnosti istovremenog pristupa DLL-u iz više programa [Izvor: Autor]

DINAMIČKO POVEZIVANJE NA WINDOWS OS

Kod Windows operativnog sistema postoje dva načina vezivanja programa na DLL.

Postoje dva načina vezivanja programa na DLL.

Prvi način je implicitno povezivanje. Ovaj način vezivanja podrazumeva postojanje **biblioteke uvoza** (engl. import library) koju generiše pomoćni program koji uzima neophodne informacije iz DLL-a.

Program se statički povezuje na biblioteku uvoza i tako dobija prozor u DLL. Program može biti povezan na nekoliko DLL-ova.

Kada se program učita u memoriju, operativni sistem po potrebi učitava i neophodne DLL-ove ako se oni već ne nalaze u memoriji.
Bitno je naglasiti da se ne mora učitati ceo DLL, jer unutar njega postoji straničenje. Nakon toga se prilagođavaju strukture u biblioteci uvoza i procedure DLL-a se mapiraju u virtuelni adresni prostor programa.

Drugi način vezivanja programa na DLL je eksplicitno povezivanje, koje ne zahteva biblioteke uvoza, niti postoji potreba za učitavanjem DLL-ova u trenutku kada se učitava program.

U ovom slučaju u vremenu izvršenja program izdaje eksplicitni poziv za vezivanje na DLL. Zatim se izdaju pozivi kojima se saznaju adrese zahtevanih procedura, čime se omogućava poziv tih procedura.

Proces raskidanja veze se DLL-om se takođe vrši eksplicitnim pozivom. Kada se svi programi odvoje od DLL-a, DLL se može ukloniti iz memorije.

DINAMIČKO POVEZIVANJE NA UNIX OS

Koncept deljenih biblioteka (engl. shared library) UNIX operativnih sistema je veoma sličan Windows DLL-ovima.

Koncept **deljenih biblioteka** (engl.shared library) UNIX operativnih sistema je veoma sličan Windows DLL-ovima.

Deljena biblioteka može da sadrži nekoliko procedura i blokova podataka i može se istovremeno vezati na više programa. Većina standardnih biblioteka jezika C i većina koda za umrežavanje je realizovana konceptom deljenih biblioteka.

UNIX podržava samo implicitno povezivanje.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 6

Pokazne vežbe

ZADATAK #1

Zadatak 1: (10 minuta)

Zadatak #1 (10 minuta)

Napisati program u NASM asembleru koji ispisuje Vaše ime i prezime na ekranu.

https://rextester.com/l/nasm_online_compiler

```
section .text
    global _start ;                 must be declared for linker (ld)
    _start: ;                       tells linker entry point
    mov edx,duz ;                   message length
    mov ecx,ime ;                   message to write
    mov ebx,1 ;                     file descriptor (stdout)
    mov eax,4 ;                     system call number (sys_write)
    int 0x80 ;                      call kernel
    mov eax,1 ;                     system call number (sys_exit)
    int 0x80 ;                      call kernel
    section .data
    ime db 'IME i PREZIME', 0xa ;   our dear string
    duz equ $ - ime ;               length of our dear string
```

ZADATAK #2

Zadatak 2 (10 minuta)

Zadatak #2 (10 minuta)

Napisati program u NASM asembleru koji ispisuje 9 zvezdica (*) na ekranu.

https://rextester.com/l/nasm_online_compiler

```
section .text
global _start ;                         must be declared for linker
_start: ;                               tell linker entry point
mov edx,len ;                           message length
mov ecx,msg ;                           message to write
mov ebx,1 ;                             file descriptor (stdout)
mov eax,4 ;                             system call number (sys_write)
```

```
int 0x80 ;                                  call kernel
mov edx,9 ;                                 message length
mov ecx,s2 ;                                message to write
mov ebx,1 ;                                 file descriptor (stdout)
mov eax,4 ;                                 system call number (sys_write)
int 0x80 ;                                  call kernel
mov eax,1 ;                                 system call number (sys_exit)
int 0x80 ;                                  call kernel
section .data
msg db 'Prikazi 9 zvezdica na ekranu',0xa ; message
len equ $ - msg ;                           length of message
s2 times 9 db '*'
```

ZADATAK #3

Zadatak 3: (10 minuta)

Zadaci za samostalni rad:

Zadatak #3,(10 minuta)

- Objasniti svaki red koda. Registri su 32 bitni kao i memorijske reči.

Analizirajte desne operande:

1. koja vrsta adresiranja u instrukciji broj 1
2. koja vrsta adresiranja u instrukciji broj 4
3. koja vrsta adresiranja u drugoj instrukciji **ADD**
4. koja vrsta adresiranja u instrukciji **CMP**

```
1.              MOV R1, #0
2.              MOV R2, #A
3.              MOV R3, #A + 1024
4.  LOOP:       ADD R1, (R2)
5.              ADD R2, #4
6.              CMP R2, R3
7.              BLT LOOP
```

Slika 6.1 Kod za analizu [Izvor: Autor]

REŠENJE:

1. neposredno adresiranje, broj 0 ide u registar R1
2. registarsko indirektno preko registra R2
3. neposredno, R2 se sabira sa brojem 4

4. registarski direktno

```
        MOV R1,#0     ; accumulate the sum in R1, initially 0
        MOV R2,#A     ; R2 = address of the array A
        MOV R3,#A+1024; R3 = address if the first word beyond A
LOOP:               ADD R1,(R2); register indirect through R2 to get operand
        ADD R2,#4     ; increment R2 by one word (4 bytes)
        CMP R2,R3     ; are we done yet?
        BLT LOOP      ; if R2 < R3, we are not done, so continue
```

Slika 6.2 Rešenje zadatka [Izvor: Autor]

ZADATAK #4

Zadatak 4: (10 minuta)

Zadaci za samostalni rad:

Zadatak #4,(10 minuta)

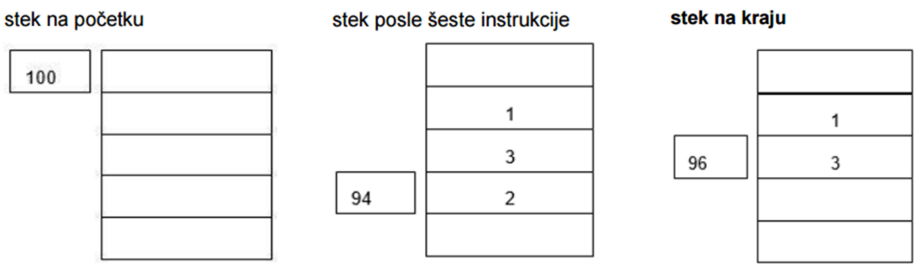
- Vrednost SP registra je100. Stek raste na dole i ukazuje na zadnju popunjenu lokaciju. Registri su 16-bitni. (Unos na / Izlaz iz steka je dužine WORD, 2 bajta)

1. Skicirati stek memoriju na početku, na kraju 6. instrukcije, i na kraju programa.
2. Kolika je vrednost SP posle instrukcije broj 3?
3. Kolika je vrednost SP posle instrukcije broj 6?
4. Kolika je vrednost SP na kraju?
5. Kolika je vrednost dx na kraju?

```
1.      MOV ax, 1
2.      MOV bx, 2
3.      MOV cx, 3
4.      PUSH ax
5.      PUSH cx
6.      PUSH bx
7.      POP dx
```

Slika 6.3 Kod za analizu [Izvor: Autor]

REŠENJE:



Slika 6.4 Rešenje zadatka zod 1. [Izvor: Autor]

2) 100

3) 94

4) 96
5) 2

PROJEKTNI ZADATAKIZ PREDMETA CS120

Predmet CS120 od predispitnih obaveza sadrži i PZ koji se sastoji od tri mini-projekta.

Na predmetu CS120 - Organizacija računara svaki student samostalno radi projektni zadatak (PZ), koji se sastoji od tri mini-projekta.

Mini-projekti brane se u petoj, desetoj i petnaestoj nedelji semestra!

Svaki student dužan je da tok izrade projekta izveštava svake nedelje u obliku kratkih izveštaja (do jednog pasusa) preko LAMS lekcija u okviru dodatnih aktivnosti interaktivnih lekcija.
O formatu mini-projekata svi studenti (tradicionalni i Internet) biće obavešteni mejlom od strane predmetnog profesora/asistenta.

Poglavlje 7

Zadaci sa samostalni rad

ZADACI ZA SAMOSTALNI RAD ZA LEKCIJU #12-ZADATAK 1

Zadatak #1 - Zadatak za samostalni rad za lekciju #12 radi se okvirno 10 min.

Zadaci za samostalni rad:

Zadatak #1,(10 minuta)

- Objasniti svaki red koda. Registri su 32 bitni kao i memorijske reči.

Analizirajte desne operande:

1. koja vrsta adresiranja u instrukciji broj 1
2. koja vrsta adresiranja u instrukciji broj 2
3. koja vrsta adresiranja u instrukciji broj 3
4. koja vrsta adresiranja u instrukciji broj 4
5. koja vrsta adresiranja u instrukciji OR
6. koja vrsta adresiranja u instrukciji ADD
7. koja vrsta adresiranja u instrukciji broj CMP

```
1.              MOV R1, #0
2.              MOV R2, #0
3.              MOV R3, #4096
4.      LOOP:   MOV R4, (R2)
5.              AND R4, (R2)
6.              OR R1, R4
7.              ADD R2, #4
8.              CMP R2, R3
9.              BLT LOOP
```

Slika 7.1 Kod za analizu [Izvor: Autor]

ZADACI ZA SAMOSTALNI RAD ZA LEKCIJU #12 - ZADATAK 2

Zadatak #2 - Zadatak za samostalni rad za lekciju #12 radi se okvirno 10 min.

Zadaci za samostalni rad:

Zadatak #2,(10 minuta)

- Vrednost SP registra =200. Stek raste na dole i ukazuje na zadnju popunjenu lokaciju. Registri su 16-bitni.
  U toku obrade prekida, na stek se smeštaju 2 registra: CS i IP.
  U toku procedure PROC1 dogode se 2 prekidna signala.

Izračunati stek pointer u toku procedure PROC1, dok traje obrada oba prekida.

```
1.      MOV ax, 1
2.      MOV bx, 2
3.      MOV cx, 3
4.      PUSH ax
5.      CALL PROC1
6.      ROR al, 1
7.      PUSH bx
8.      POP dx
```

Slika 7.2 Kod za analizu [Izvor: Autor]

Poglavlje 8

Domaći zadatak

DOMAĆI ZADATAK #12

Domaći zadatak #12 okvirno se radi 20 minuta

Domaći zadatak #12 (20 minuta)

Napisati program u NASM asembleru koji u pet reda ispisuje Vaše ime i prezime, broj indexa i godinu studiranja na ekranu.
Koristiti NASM online ili NASM asembler instaliran na Vašem računaru.
https://rextester.com/l/nasm_online_compiler

Predaja domaćeg zadataka
Tradicionalni studenti:
Domaći zadatak treba dostaviti najkasnije 7 dana nakon vežbi, za 100% poena. Nakon toga poeni se umanjuju za 50%.

Domaći zadatak poslati predmetnim asistentima, sa predmetnim profesorima u CC.

Predati domaći zadatak koristeći .doc/.docx uputstvo dato u prvoj lekciji.
Internet studenti treba poslati domaće zadatke najkasnije do 10 dana pred izlazak na ispit predmetnom asistentu zaduženog za internet studente.

Napomena:

Svaki domaći zadatak treba da bude napisan prema dokumentu za predaju domaćih zadataka koji je dat na kraju interaktivne lekcije.

ZAKLJUČAK

Recime lekcije #11

Kao i u svakom drugom programskom jeziku, program u asembleru je sačinjen od niza uzastopno navedenih instrukcija. Stoga bi se moglo pretpostaviti da je svaku instrukciju dovoljno pročitati i prevesti u mašinski jezik u samo jednom prolazu kroz program. Nažalost stvari nisu tako jednostavne, kao što smo opisali u nastavku ove lekcije.
Da bi se program mogao prevesti neophodno je poznavati konkretne vrednosti svih promenljivih.
Osnovna funkcija prvog prolaza prevođenja je formiranje tabele simbola. Tabela simbola sadrži
zaseban unos za svaki simbol. Tabela operacionih kodova sadrži bar jedan unos za svaku asemblersku instrukciju. U tabeli je opisan broj i vrsta operanada instrukcije, heksa-decimalni kod
instrukcije u mašinskom jeziku, dužina instrukcije i klasa instrukcije.
Drugi prolaz prevođenja generiše objektni kod, prikazuje informacije koje su neophodne za proces povezivanja (linker) i opciono prikazuje listing prevođenja. Proces povezivanja (engl. linking) je usko povezan sa načinom organizacije softvera. Kao i u višim programskim jezicima i u asembleru postoji potreba da se softver piše modularno.

Zasebno asemblirani programi mogu se povezati u izvršni binarni program Taj posao obavlja program za povezivanje. Njegovi osnovni zadaci su relociranje koda i vezivanje imena.

Dinamičko povezivanje je tehnika kojom se procedure ne povezuju sve dok ih glavni program stvarno ne pozove.
U Windows-ovim DLL datotekama i UNIX-ovim deljenim bibliotekama koristi se dinamičko povezivanje.

Literatura:

A. Tanenbaum, Structured Computer Organization,

Chapter 07, pp. 530 – 552,