UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Đorđe Z. Milićević

REALIZACIJA PROGRAMSKOG JEZIKA AKCENT ZA PROGRAMIRANJE ATMEGA328P MIKROKONTROLERA POMOĆU LLVM INFRASTRUKTURE

master rad

Beograd, 2018.

Mentor:

doc. dr Milena Vujošević Janičić
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Filip Marić
Univerzitet u Beogradu, Matematički fakultet

prof. dr Saša Malkov
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: 28. septembar 2018.

Naslov master rada: Realizacija programskog jezika aKcent za programiranje ATmega328P mikrokontrolera pomoću LLVM infrastrukture

Rezime: U radu su opisani dizajn i implementacija imperativnog, statički tipiziranog programskog jezika aKcent. Razvijen je sa ciljem da programiranje mikrokontrolera učini jednostavnim i zabavnim.

Implementaciju programskog prevodioca prate alati Flex i Bison (upotrebljeni za izgradnju leksičkog i sintaksičkog analizatora), kompajlerska infrastruktura LLVM (upotrebljena za generisanje LLVM-ovog međukoda, njegovo unapređivanje i prevođenje na mašinski kôd za odgovarajuću ciljnu arhitekturu) i programski jezik C++ u kom je implementiran najveći deo prevodioca. Specifične funkcionalnosti ATmega328P mikrokontrolera omogućene su posredstvom pomoćne biblioteke napisane u programskom jeziku C, po uzoru na mikrokontrolersku platformu Arduino. U njoj su implementirane funkcije za konfigurisanje odgovarajućih pinova i rad sa perifernim komponentama mikrokontrolera.

Programski jezik aKcent podržava izdvajanje delova koda u zasebne programske celine i njihovu kasniju upotrebu (funkcije i funkcijske pozive), uvođenje korisničkih promenljivih odgovarajućih tipova, naredbe za kontrolu toka izvršavanja programa (ciklične programske strukture i grananje) i proširivanje trenutno podržanog skupa funkcionalnosti ATmega328P mikrokontrolera posredstvom korisnički definisanih funkcija iz pomoćne biblioteke. Omogućeno je nekoliko tipova podataka, pokazivači i jednodimenzioni statički nizovi. Nad odgovarajućim tipovima podataka omogućene su implicitne konverzije (promocije i democije). Semantički analizator sprovodi provere tipova, kontrolu dosega promenljivih i provere funkcijskih deklaracija, definicija i poziva. Sintaksa programskog jezika predstavlja sintezu sintaksičkih konstrukcija postojećih programskih jezika, od kojih je najdominantniji programski jezik C.

Ključne reči: programski jezik, LLVM, kompajler, mikrokontroler, aKcent

Sadržaj

1 Uvod **1**

2 Opis ciljne arhitekture **3**
2.1 Centralna procesorska jedinica 4
2.2 Organizacija memorije 5
2.3 U/I portovi 6
2.4 Sistemski prekidi 7
2.4.1 Tehnički aspekt rada sistemskih prekida 8
2.5 Tajmersko-brojački modul 8
2.6 AD konvertor 9
2.7 Serijska komunikacija: USART 10

3 Opis softverskih alata **12**
3.1 Alat Flex 12
3.1.1 Format ulazne datoteke 13
3.1.2 Tehnički aspekt rada generisanog skenera 14
3.2 Alat Bison 15
3.2.1 Format ulazne datoteke 15
3.2.2 Zapis gramatičkih pravila 17
3.2.3 Opis rada generisanog parsera 18
3.2.4 Konflikti 19
3.2.5 Povezivanje sa skenerom generisanim alatom Flex 19
3.3 Infrastruktura LLVM 19
3.3.1 Središnji deo LLVM-ove arhitekture 20
3.3.2 Organizacija i sintaksa LLVM-ovog međukoda 21
3.3.3 Pregled korišćenih alata 23

4 Opis programskog jezika **25**
4.1 Tipovi podataka 25
4.2 Funkcije 26
4.2.1 Korisnički definisane funkcije 27
4.3 Promenljive i doseg 30
4.3.1 Pokazivačke promenljive 30
4.3.2 Nizovi 30
4.4 Operatori 31
4.5 Naredbe za kontrolu toka izvršavanja programa 32
4.5.1 Naredba if-else 32
4.5.2 Petlja while 33
4.5.3 Petlja forever 34
4.5.4 Petlja for-in 34

5 Implementacija kompilatora **36**
5.1 Leksička analiza 36
5.2 Sintaksička analiza 37
5.2.1 Apstraktno sintaksičko stablo 37
5.3 Semantička analiza 39
5.3.1 Tablica simbola 39
5.3.2 Registrovanje promenljivih i prototipova funkcija 39
5.3.3 Provera tipova 41
5.3.4 Kontrola dosega 42
5.4 Generisanje LLVM-ovog međukoda 43
5.5 Pomoćna biblioteka 44
5.5.1 Funkcije za merenje vremena 45
5.5.2 Funkcije za konfigurisanje pinova, upotrebu AD konvertora i PWM 46
5.5.3 Funkcije za korišćenje UART protokola 47
5.6 Primer programa 48
5.7 Faze prevođenja programa 52

6 Zaključak **56**

Literatura **57**

Glava 1

Uvod

Mikrokontroleri pružaju detaljniji uvid u osnovne koncepte i način funkcionisanja složenih računarskih sistema. Njihova sveprisutnost u najrazličitijim vrstama elektronskih uređaja i ugrađenih sistema (eng. *embedded systems*), ekonomičnost u potrošnji električne energije, visok stepen konfigurabilnosti, minijaturne dimenzije i širok dijapazon izbora, samo su neke od mnogobrojnih karakteristika koje doprinose značaju i težnji za njihovim izučavanjem. Primarni aspekt njihove upotrebe ogleda se u povezivanju virtuelnog i fizičkog sveta.

U radu su predstavljeni dizajn i implementacija imperativnog, statički tipiziranog programskog jezika, koji nosi naziv aKcent. Razvijen je sa ciljem da na jednostavan i zabavan način korisnika uvede u svet mikrokontrolera, zadržavajući određeni nivo apstrakcije koji je prisutan u mnogim drugim programskim jezicima i mikrokontrolerskim platformama. Implementaciju programskog prevodioca prate programski jezik C++, alati za obradu struktuiranog teksta (Flex i Bison) i kompajlerska infrastruktura LLVM, koja nudi pregršt alata za unapređivanje i prevođenje programskog koda na mašinski jezik. Od verzije 4, LLVM uvodi podršku za AVR arhitekturu, kojoj pripada ATmega328P mikrokontroler.

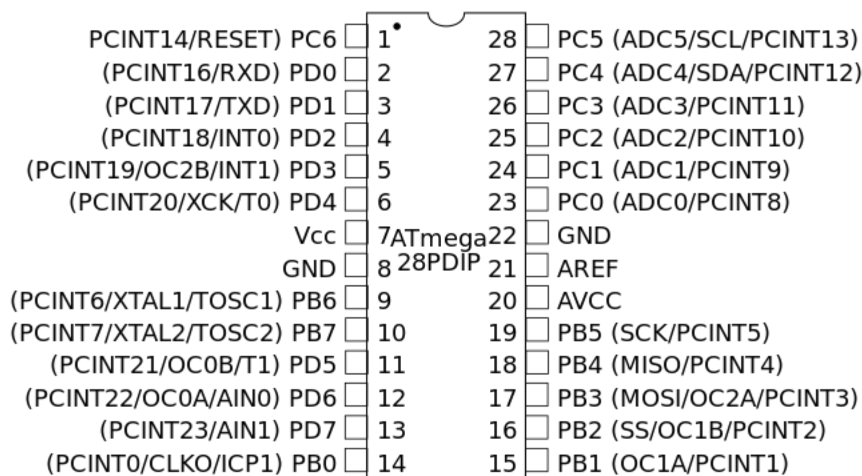
Glava 2 opisuje arhitekturu ATmega328P mikrokontrolera na kojem će se prevedeni program izvršavati i otkriva osnovne koncepte na kojima počiva upotreba mikrokontrolera. U glavi 3 predstavljeni su alati koji su umnogome olakšali implementaciju programskog prevodioca, način njihovog rada i formalizmi na kojima su zasnovani. Glava 4 pruža uvid u sintaksu i trenutne mogućnosti programskog jezika aKcent. Centralni deo rada (glava 5) opisuje implementacione detalje programskog prevodioca (leksičku, sintaksičku, semantičku analizu i fazu generisanja međukoda), pomoćnu biblioteku kojom su apstrahovane specifične funkcionalnosti

mikrokontrolera i sadrži primer koji demonstrira pisanje i prevođenje programa napisanih na programskom jeziku aKcent. Takođe, prikazan je funkcionalno ekvivalentan program, napisan za Arduino platformu, koja predstavlja glavnu inspiraciju za nastanak ovog programskog jezika. U glavi 6 izveden je zaključak i date su smernice za buduća unapređivanja programskog jezika.

Glava 2

Opis ciljne arhitekture

ATmega328P predstavlja osmobitni mikrokontroler, zasnovan na unapređenoj AVR *RISC* (eng. *reduced instruction set computer*) arhitekturi. Radi postizanja maksimalnih performansi, realizovana je kao Harvard (eng. *Harvard*) arhitektura, gde se podrazumeva razdvojenost magistrala i memorija za programe i podatke [9]. Posredstvom navedenog mikrokontrolera, u nastavku će biti predstavljene glavne karakteristike arhitekture, njena organizacija i različite komponente mikrokontrolera. Raspored i oznake pinova prikazani su na slici 2.1.



Slika 2.1: Raspored i oznake pinova[1]

[1]Slika je preuzeta sa naredne adrese: `https://commons.wikimedia.org/wiki/File:ATmega328P_28-PDIP.svg`

2.1 Centralna procesorska jedinica

Glavna uloga centralne procesorske jedinice (eng. *central processing unit*) je da obezbedi korektno izvršavanje programa, te je neophodno da vrši izračunavanja i kontrolu perifernih komponenti, rukuje sistemskim prekidima (eng. *interrupts*) i pristupa memoriji [9].

Najbitnije komponente centralne procesorske jedinice su [9]:

Registrator (eng. *register file*)
Registrator čine trideset dva osmobitna, radna registra, opšte namene. Za potrebe indirektnog adresiranja memorijskog prostora za podatke omogućeno je kombinovanje zadnjih šest registara u tri šesnaestobitna registra. Većina instrukcija poseduje direktan pristup registratoru. Svaki od registara takođe je mapiran u delu memorijskog prostora za podatke, kao što je navedeno u poglavlju 2.2.

Statusni registar (eng. *status register*)
Osmobitni registar, koji sadrži informacije o rezultatu poslednje izvršene aritmetičke ili logičke operacije, naziva se statusni registar. Te informacije mogu biti iskorišćene za određivanje daljeg toka izvršavanja programa. Prilikom izvršavanja rutine za obradu prekida (eng. *interrupt service routine*), očuvanje sadržaja registra ne vrši se implicitno. Operacije čitanja i upisivanja podržane su za svaki statusni bit ovog registra.

Aritmetičko-logička jedinica (eng. *arithmetic-logic unit*)
Aritmetičko-logička jedinica predstavlja mesto izvršavanja aritmetičkih i logičkih operacija. Operandi mogu biti radni registri, ili radni registar i konstanta. Nakon izvršavanja određene operacije u statusni registar se upisuju informacije o rezultatu primenjene operacije.

Pokazivač na vrh steka (eng. *stack pointer*)
Programski stek (eng. *stack*) uglavnom se koristi za smeštanje privremenih podataka, lokalnih promenljivih i adresa povratka. Smešten je u delu memorijskog prostora za podatke, a njegovo popunjavanje vrši se od viših ka nižim adresama. Pokazivač na vrh steka čuva adresu prve slobodne lokacije na steku, a sačinjen je od dva osmobitna registra.

Brojač instrukcija (eng. *program counter*)
Brojač instrukcija predstavlja registar koji sadrži adresu naredne instrukcije koju je potrebno izvršiti. Kako bi bio u mogućnosti da adresira sve lokacije memorijskog postora za programe, čije adrese pripadaju intervalu [0x000, 0x3FFF], brojač instrukcija je četrnaestobitan.

2.2 Organizacija memorije

Arhitektura AVR poseduje dva glavna memorijska prostora [9]:

Memorijski prostor za programe (eng. *program memory space*)
ATmega328P sadrži 32 KiB fleš (eng. *flash*) memorije, koja služi za smeštanje instrukcija programa. Pošto AVR arhitektura poseduje šesnaestobitne i tridesetdvobitne instrukcije, memorija je organizovana u 2^{14} šesnaestobitnih memorijskih lokacija. Memorijski prostor za programe podeljen je na dve sekcije:

Sekcija za aplikativne programe (eng. *application program section*)
Kao što naziv sugeriše, sekcija za aplikativne programe predstavlja mesto za smeštanje aplikativnog koda, tj. instrukcija programa koje mikrokontroler izvršava.

Sekcija za učitavačke programe (eng. *boot loader section*)
Učitavačkim programima omogućeno je da vrše upisivanje i iščitavanje celokupnog memorijskog prostora za programe, uključujući i memorijski prostor u kome se i sami nalaze, posredstvom bilo kog dostupnog interfejsa i odgovarajućeg protokola.

Memorijski prostor za podatke (eng. *data memory space*)
ATmega328P sadrži 2304 osmobitnih memorijskih lokacija SRAM (eng. *static random-access memory*) memorije. Prvih 256 memorijskih lokacija, čije adrese pripadaju intervalu [0x0000, 0x00FF], adresiraju naredne memorijske prostore:

- *registarski prostor* (eng. *register file space*) — sačinjen od trideset dva osmobitna, radna registra, opšte namene, čije adrese pripadaju intervalu [0x0000, 0x001F];

- *U/I prostor* (eng. *I/O space*) — sačinjen od šezdeset četiri registra, čije adrese pripadaju intervalu [0x0020, 0x005F];
- *prošireni U/I prostor* (eng. *extended I/O space*) — sačinjen od sto šezdeset registara, čije adrese pripadaju intervalu [0x0060, 0x00FF].

Narednih 2048 memorijskih lokacija, čije adrese pripadaju intervalu [0x0100, 0x08FF], predstavljaju interni memorijski prostor za podatke.

Mikrokontroleri takođe poseduju tzv. EEPROM (eng. *electrically erasable and programmable read-only memory*) memoriju za smeštanje podataka, čiji sadržaj, kao i sadržaj memorije u kojoj se nalaze programi, ostaje nepromenjen nakon isključivanja mikrokontrolera [12]. ATmega328P sadrži 1 KiB opisane memorije [9].

2.3 U/I portovi

Interakcija sa spoljašnjim svetom pretežno se odvija posredstvom digitalnih U/I pinova[2] (eng. *digital I/O pins*) mikrokontrolera, koji su organizovani u U/I portove (eng. *ports*). ATmega328P poseduje tri porta, označena slovima: *B*, *C* i *D*. Svakom pinu *n*, porta *x*, dodeljena su tri registarska bita, kojima se vrši njegovo podešavanje:

DDRxn bitovi

Logička vrednost bita *DDRxn*, u okviru registra *DDRx*, određuje da li će pin biti definisan kao ulazni (eng. *input*) ili izlazni (eng. *output*). Ulazni pinovi služe za očitavanje logičke vrednosti koja im je prosleđena (npr. posredstvom digitalnih senzora), dok izlazni omogućavaju propagaciju logičke vrednosti upisane u registarski bit *PORTxn*.

PORTxn bitovi

Ukoliko je pin označen kao izlazni, vrednosti bita *PORTxn*, u okviru registra *PORTx*, određuje logičku vrednost koja će biti dodeljena pinu. U slučaju ATmega328P mikrokontrolera, logička jedinica označava strujni napon od 5 V, a logička nula strujni napon od 0 V.

[2]Termin *pin* označava nožicu mikrokontrolera.

PINxn bitovi

Nezavisno od logičke vrednosti bita *DDRxn*, logička vrednost, prisutna na određenom pinu mikrokontrolera, može se pročitati u registarskom bitu *PINxn*. Upisivanjem logičke jedinice u registarski bit *PINxn* vrši se invertovanje bita *PORTxn*.

Adrese prethodno navedenih registara nalaze se u U/I memorijskom prostoru [9].

2.4 Sistemski prekidi

Sistemski prekidi predstavljaju način reagovanja na različite vrste prioritetizovanih događaja. Kad god se steknu uslovi za generisanje određenog prekida, dotadašnji tok izvršavanja programa se prekida, i prelazi se na izvršavanje odgovarajuće rutine za obradu nastalog prekida. Kada se rutina izvrši, program nastavlja sa normalnim tokom izvršavanja od mesta na kom je prekid nastupio [12].

Sa stanovišta mesta generisanja, razlikuju se dve vrste sistemskih prekida [12]:

Interni prekidi (eng. *internally triggered interrupts*)

Interni prekidi odgovaraju događajima koje generišu periferne komponente mikrokontrolera. Primera radi, do generisanja internog prekida može doći usled pristizanja podataka u periferne komponente (opisane u poglavljima 2.7 i 2.6), ili kada brojački registar (komponente opisane u poglavlju 2.5) dostigne određenu vrednost. Umesto iščekivanja novih podataka, u petlji koja blokira dalje izvršavanje programa, elegantnije rešenje je napisati odgovarajuću rutinu koja će automatski biti izvršena kada novi podaci pristignu, tj. kada se prekid dogodi.

Eksterni prekidi (eng. *externally triggered interrupts*)

Eksterni prekidi nastaju usled promene vrednosti napona, tj. logičke vrednosti, na određenim pinovima mikrokontrolera. Česta situacija, u kojoj se navedena vrsta prekida koristi, predstavlja reagovanje na pritisak dugmeta. Umesto stalnog proveravanja logičke vrednosti pina na koji je dugme povezano, elegantnije rešenje je napisati odgovarajuću rutinu koja će automatski biti izvršena kada se pritisak registruje.

2.4.1 Tehnički aspekt rada sistemskih prekida

Svakom sistemskom prekidu dodeljena su po dva bita određenih kontrolnih registara. Jedan od njih služi za omogućavanje konkretnog sistemskog prekida (tzv. *interrupt enable* bit), dok drugi biva postavljen na odgovarajuću logičku vrednost kada se steknu uslovi za generisanje konkretnog sistemskog prekida (tzv. *interrupt flag* bit). Takođe, statusni registar centralne procesorske jedinice poseduje bit za globalno omogućavanje sistemskih prekida (tzv. I bit). Kada je I bit postavljen na logičku jedinicu, mikrokontroler vrši stalne provere svih *interrupt flag* bitova. Ukoliko je neki od tih bitova postavljen na odgovarajuću logičku vrednost, I bit biva postavljen na logičku nulu[3] i sledi izvršavanje rutine za obradu najprioritetnijeg prekida. Kada se rutina za obradu prekida izvrši, I bit biva vraćen na logičku jedinicu. Ukoliko nema dodatnih sistemskih prekida, izvršavanje programa se nastavlja od mesta na kom je prethodni prekid nastupio, dok se u suprotnom opisani postupak ponavlja [12].

Sistemski prekidi su predstavljeni svojim vektorima prekida (eng. *interrupt vectors*), koji su smešteni u najnižim adresama memorijskog prostora za programe. Što je adresa vektora prekida manja, to je njegov prioritet viši. Svaki vektor prekida sadrži instrukciju skoka na odgovarajuću labelu, nakon koje slede instrukcije koje se nalaze u telu rutine za obradu tog prekida. Sa aspekta programskog jezika C, rutina za obradu prekida predstavljena je makro funkcijom, koja ne sadrži oznaku tipa povratne vrednosti, naziv joj je *ISR*, a jedini parametar je predefinisana oznaka sistemskog prekida za koji se ta rutina navodi [9].

2.5 Tajmersko-brojački modul

Tajmersko-brojački (eng. *timer/counter*) moduli predstavljaju hardverske komponente koje se obično koriste za precizno merenje vremena, određivanje broja promena logičkih vrednosti na određenim pinovima mikrokontrolera i generisanje tzv. *PWM* (eng. *pulse with modulation*) signala. Naivan metod merenja vremena predstavlja trošenje (često dragocenog) procesorkog vremena izvršavanjem unapred određenog broja instrukcija. Stoga, upotreba modula donosi sledeće prednosti: kraći i čitljiviji programski kôd, značajno veću preciznost i oslobađanje procesora za rad na drugim zadacima. Uzimajući u obzir da su mikrokontroleri digitalne

[3]Potrebno je onemogućiti izvršavanje ostalih sistemskih prekida, dok se rutina za obradu nastalog prekida ne izvrši.

sprave, *PWM* tehnika omogućava generisanje različitih vrednosti napona. Način na koji se navedena pojava ostvaruje je veoma brzo invertovanje logičkih vrednosti (logičkih nula i jedinica) na određenim pinovima. Generisana vrednost napona, na određenom pinu, proporcionalna je srednjoj vrednosti vremena u kom je izlaz na tom pinu bio logička jedinica [12].

ATmega328P poseduje dva osmobitna (*TC0 i TC2*) i jedan šesnaestobitni (*TC1*) tajmersko-brojački modul [9]. Ulaz modula predstavljaju impulsi, generisani posredstvom sistemskog sata (eng. *system clock*) ili spoljašnjeg izvora (povezanog na određeni pin mikrokontrolera). Ukoliko se kao izvor impulsa koristi sistemski sat, njegova frekvencija se može skalirati na odgovarajuću vrednost podešavanjem deliteljske komponente (eng. *prescaler*). Izbor nekog od navedenih izvora vrši se podešavanjem *CS* (eng. *clock select*) bitova kontrolnog registra. Broj generisanih impulsa čuva se u internom brojačkom registru čiji je kapacitet određen rezolucijom modula [12]. U zavisnosti od izabranog režima rada, vrednost brojačkog registra biva inkrementirana, dekrementirana ili anulirana pri svakom detektovanom impulsu [9]. Modul takođe poseduje nekoliko upoređivačkih registara čije se vrednosti konstantno porede sa vrednošću brojačkog registra. Kada se korisnički definisana vrednost upoređivačkog registra poklopi sa vrednošću brojačkog registra, aktivira se generator signala (eng. *waveform generator*) [12]. Režim rada generatora određuje se podešavanjem *COM* (eng. *compare output mode*) i *WGM* (eng. *waveform generation mode*) bitova kontrolnih registara. Prvi podešavaju ponašanje odgovarajućih pinova mikrokontrolera, dok drugi podešavaju način brojanja, gornju vrednost brojača i tip generatora signala. Takođe, događaj poklapanja vrednosti brojačkog i upoređivačkog registra može generisati odgovarajuće sistemske prekide [9].

2.6 AD konvertor

Uzimajući u obzir digitalnu prirodu mikrokontrolera, interakcija sa spoljašnjim svetom često se odvija posredstvom različitih vrsta analognih senzora, povezanih na odgovarajuće ulazne pinove mikrokontrolera. Njihova uloga je prevođenje intenziteta različitih prirodnih pojava (svetlosti, zvuka, pritiska, ...) u odgovarajuće vrednosti napona. Očitavanje analognih senzora, tj. vrednosti napona koji generišu, vrši se posredstvom kompleksne, periferne, hardverske komponente — AD konvertora (eng. *analog-to-digital converter*) [12]. ATmega328P poseduje jedan de-

setobitni AD konvertor [9]. Nekoliko pinova mikrokontrolera deli taj AD konvertor posredstvom odgovarajućeg multipleksera, što za posledicu ima nemogućnost istovremenog očitavanja više ulaznih pinova. Izbor frekvencije rada ove komponente vrši se podešavanjem deliteljske komponente, koji skalira frekvenciju sistemskog sata na odgovarajuću vrednost. Kako se sva merenja vrše u odnosu na korisnički određenu referentnu vrednost napona, AD konvertor poseduje nekoliko načina za njeno definisanje [12].

Merenje napona vrši se metodom sukcesivne aproksimacije (eng. *successive approximation*), koja je veoma bliska algoritmu binarne pretrage. Unutar AD konvertora nalazi se DA konvertor (eng. *digital-to-analog converter*), koji vrši generisanje određene vrednosti napona. Ta vrednost je inicijalno postavljena na referentnu vrednost. Generisani napon se, posredstvom uporeĐivača (eng. *comparator*), upoređuje sa naponom na (multiplekserom) izabranom ulaznom pinu mikrokontrolera, na koji se priključuju analogni senzori. U zavisnosti od rezultata poređenja, uporeĐivač šalje odgovarajuću logičku vrednost kolu za konverzije (eng. *conversion logic*). Na osnovu prosleđene logičke vrednosti, kolo za konverzije je u mogućnosti da odredi vrednost napona koji će DA konvertor generisati u narednoj iteraciji. Kao i kod binarne pretrage, svaka iteracija sužava prostor pretrage za polovinu prethodnog. Nakon deset iteracija, približna vrednost napona sa ulaznog pina smeštena je u odgovarajuće registre [12].

Podešavanje rada AD konvertora obuhvata izbor odgovarajuće radne frekvencije. Postavljanjem odgovarajućeg bita na logičku jedinicu započinje se proces konverzije, a na kraju konverzije taj bit automatski biva vraćen na logičku nulu. Kako je proces konverzije relativno brz (između 13 i 260 mikrosekundi [9]), obično se u blokirajućoj petlji proverava da li je logička vrednost prethodno opisanog bita vraćena na logičku nulu. Pored predstavljenog naivnog načina, AD konvertor može da generiše prekid kada zavši sa konverzijom i na taj način oslobodi procesor za rad na drugim zadacima [12].

2.7 Serijska komunikacija: USART

Kao i većina stvari u računarstvu, komunikacija između digitalnih uređaja, poput mikrokontrolera i računara, zasnovana je na jednostavnom konceptu. Unapred utvrđeni skup pravila, kojima se podaci kodiraju u logičke vrednosti (i obratno), predstavlja protokol komunikacije. Stoga, razmena podataka između uređaja svo-

di se na razmenu logičkih vrednosti korišćenjem izabranog protokola. Periferna hardverska komponenta, koju mikrokontroleri koriste radi uspostavljanja komunikacije sa drugim digitalnim uređajima, korišćenjem serijskog protokola, naziva se USART (eng. *universal synchronous and asynchronous receiver and transmitter*) [12].

Kako bi se komunikacija realizovala, potrebno je izvršiti određena podešavanja USART komponente. Brzina prenosa podataka, izražena u baudima[4] (eng. *bauds*), mora biti usaglašena između učesnika. Podešavanje brzine prenosa vrši se upisivanjem bitova u određene registre, na osnovu čega se frekvencija sistemskog sata skalira na odgovarajuću vrednost. Nakon toga se vrši podešavanje okvira za prenos podataka (eng. *data frame*), što obuhvata: određivanje broja završnih (eng. *stop*) bitova, omogućavanje kontrole parnosti (eng. *parity check*) i određivanje broja bitova koji predstavljaju podatke. Sledeći korak je omogućavanje slanja i primanja podataka postavljanjem odgovarajućih bitova. Pre svake transakcije potrebno je izvršiti proveru spremnosti komponente da prosledi ili prihvati podatke. Provera spremnosti se može realizovati proverom odgovarajućih bitova, koje komponenta održava, ili putem sistemskih prekida. Razmena podataka odvija se bajt po bajt posredstvom za to predviđenog registra [12].

[4]Broj bauda označava broj razmenjenih bitova u sekundi.

Glava 3

Opis softverskih alata

Flex i Bison predstavljaju alate pomoću kojih je moguće generisati programe za obradu strukturiranog teksta [4]. Iako je zasebna upotreba navedenih alata moguća (eng. *standalone*), obično se koriste kombinovano. Infrastruktura LLVM predstavlja softverski projekat orijentisan ka izgradnji i istraživanju naprednih kompilatorskih tehnologija. Kako su navedeni alati dosta kompleksni, u nastavku će biti predstavljene njihove određene mogućnosti, važne sa aspekta konstrukcije kompilatora za programski jezik opisan u poglavlju 4.

3.1 Alat Flex

Uloga alata Flex ogleda se u generisanju programa koji vrše leksičku analizu, tzv. *skenera* (eng. *scanners*). Prvu verzija alata, čiji je naziv Lex, napisali su Majkl Lesk (eng. *Michael Lesk*) i Erik Šmit (eng. *Eric Schmidt*) 1975. godine. Vern Pekson (eng. *Vern Paxson*) je 1987. godine napisao verziju alata na programskom jeziku C, koja nosi naziv Flex [4]. Najnovija verzija alata je 2.6.4.

Specifikacija skenera obično se zadaje odgovarajućom ulaznom datotekom, na osnovu koje Flex generiše izvorni kôd opisanog skenera. Ugrubo govoreći, ulazna datoteka sadrži listu parova regularnih izraza (eng. *regular expressions*) i odgovarajućih instrukcija. Prilikom iščitavanja ulaza, skener poredi pročitane karaktere sa navedenim regularnim izrazima. Bitna karakteristika skenera ogleda se u tome da se skup navedenih regularnih izraza prevodi u internu formu koja omogućava simultano poređenje ulaza sa svim navedenim regularnim izrazima [4]. Nakon svakog podudaranja izvršavaju se instrukcije dodeljene izrazu na osnovu kog je deo ulaza prepoznat.

3.1.1 Format ulazne datoteke

Specifikacija skenera sastoji se iz tri dela: *deo za definicije* (eng. *definitions section*), *deo za pravila* (eng. *rules section*) i *deo za korisnički kôd* (eng. *user code section*). Delovi su međusobno razdvojeni linijama koje sadrže simbole *%%*.

Deo za definicije

Deo za definicije može da sadrži opcije (eng. *options*), zamene (eng. *substitutions*) i formatirane blokove koda na programskom jeziku C++ [4].

Opcijama se vrši podešavanje rada skenera, a sintaksa njihovog navođenja je sledeća:

```
%option naziv_opcije
```

Neke od često korišćenih opcija su [4]:

- **noyywrap** — izostavlja se poziv funkcije *yywrap* pri nailasku na kraj datoteke koja se tog trenutka obrađuje;
- **yylineno** — deklariše se i održava vrednost globalne promenljive *yylineno* u kojoj se čuva redni broj linije koja se obrađuje. Pošto se podrazumevano ne vrši inicijalizacija te promenljive, potrebno ju je inicijalizovati na vrednost 1.

Zamene se koriste radi uprošćavanja specifikacije skenera. Njihovim korišćenjem moguće je dodeliti određeni naziv nekom regularnom izrazu i posredstvom tog naziva referisati na njega. Deklarisanje zamene vrši se navođenjem naziva zamene i odgovarajućeg regularnog izraza. U nastavku je dat primer zamene kojom se regularni izraz, za prepoznavanje jednocifrenih brojeva, zamenjuje navedenim nazivom:

```
SINGLE_DIGIT_NUMBER [0-9]
```

Prilikom referisanja, naziv zamene se navodi između vitičastih zagrada [4].

Delovi programskog koda, napisani na programskom jeziku C++, koji se navode između simbola *%{* i *%}*, doslovno se prepisuju u izlaznu datoteku, ispred definicije funkcije *yylex*. Obično se u njima uključuju potrebna zaglavlja ili navode deklaracije promenljivih i funkcija koje će biti korišćene u delu za pravila [4].

Deo za pravila

Deo za pravila predstavlja centralni deo specifikacije skenera i sastoji se od skupa pravila. Pravila se navode kao parovi obrazaca i akcija, razdvojeni belinama. Obrasci su predstavljeni regularnim izrazima. Delovi koda, na programskom jeziku C++, koji odgovaraju svakom od navedenih obrazaca, predstavljaju akcije. Svako podudaranje ulaza određenim obrascem rezultuje izvršavanjem akcije dodeljene tom obrascu. Ukoliko se programski kôd neke akcije prostire u više od jedne linije, tada mora biti naveden između vitičastih zagrada.

Deo za korisnički kôd

Kao što naziv sugeriše, u ovom delu se navode definicije pomoćnih funkcija. Kod jednostavnijih programa se ovde može definisati funkcija *main*. Navođenje ovog dela je opciono. Ukoliko je naveden, vrši se doslovno prepisivanje njegovog sadržaja u izlaznu datoteku.

3.1.2 Tehnički aspekt rada generisanog skenera

Izlazna datoteka, koja predstavlja izvorni kôd generisanog skenera, sadrži funkciju za skeniranje *yylex*, tablice simbola i druge pomoćne funkcije, promenljive i makroe [11].

Kada se izvrši poziv funkcije *yylex*, vrši se skeniranje ulaza na koji pokazuje promenljiva *yyin*. Svaki put kada se pronađe podudaranje ulaza sa nekim od navedenih pravila, odgovarajuća akcija, posredstvom funkcije *yylex*, vraća prepoznati token[1], a promenljiva *yytext* sadrži adresu niske prepoznate tim pravilom. Funkcija *yylex* nastavlja sa skeniranjem ulaza sve dok se ne dosegne *EOF* (eng. *end-of-file*), ili dok neka od akcija ne izvrši *return* naredbu (tj. vrati prepoznati token). Svaki skener poseduje interni bafer, određenog kapaciteta, u kom se nalazi deo pročitanog teksta. Kad god je bafer prazan, poziva se makro *YY_INPUT*, koji deo ulaza smešta u bafer, dok u celobrojnu promenljivu *result* upisuje odgovarajuću vrednost. Ukoliko je dosegnut *EOF*, u promenljivu *result* se upisuje vrednost *YY_NULL*, dok se u suprotnom upisuje broj pročitanih karaktera. Kada se dosegne kraj ulaza, skener vrši proveru funkcije *yywrap* kako bi odredio dalji tok izvršavanja. Ako funkcija *yywrap* vrati vrednost 0, pretpostavlja se da je u telu

[1] U kontekstu alata Flex, tokeni predstavljaju male celobrojne vrednosti koje označavaju vrstu prepoznate niske.

funkcije pokazivač *yyin* postavljen na novi ulaz, pa se skeniranje nastavlja, dok se u slučaju vraćanja nenula vrednosti skeniranje završava [11].

3.2 Alat Bison

Alat Bison služi za generisanje programa koji obavljaju sintaksičku analizu, tzv. *parsera* (eng. *parsers*). Nastao je kao unapređenje alata Yacc, napisanog između 1975. i 1979. godine, čiji je autor Stiven Džonson (eng. *Stephen Johnson*) [4]. Autori unapređene verzije su Robert Korbet (eng. *Robert Corbett*) i Ričard Stolmen (eng. *Richard Stallman*) [4]. Najnovija verzija alata je 3.0.4.

Specifikacija parsera obično se zadaje odgovarajućom ulaznom datototekom na osnovu koje Bison generiše izvorni kôd parsera na programskom jeziku C++.

3.2.1 Format ulazne datoteke

Format ulazne datoteke, kojom se zadaje specifikacija parsera, sličan je opisanom formatu ulazne datoteke koji koristi Flex. Datoteka se sastoji iz tri dela (čiji su nazivi dati u poglavlju 3.1.1), razdvojena linijama koje sadrže simbole *%%* [4].

Deo za definicije

Kao i kod programa Flex, deo za definicije može da sadrži delove koda napisane na programskom jeziku C++, koji se doslovno prepisuju u izlaznu datoteku, ispred definicije funkcije *yyparse*. Takvi delovi koda se navode između simbola *%{* i *%}*. Obično se u njima uključuju potrebna zaglavlja ili navode deklaracije promenljivih i funkcija koje će biti korišćene u ostalim delovima ulazne datoteke [4].

Novinu, u odnosu na Flex, predstavljaju naredne direktive:

%token

Simboli koje skener, posredstvom funkcije *yylex*, prosleđuje parseru nazivaju se terminirajućim. Sintaksa navođenja terminirajućih simbola je sledeća:

```
%token simbol_1, ..., simbol_n
```

Svakom terminirajućem simbolu dodelje se (implicitno ili eksplicitno) odgovarajuća celobrojna vrednost koja predstavlja njegov identifikator [4].

%type

Simboli navedeni ovom direktivom smatraju se neterminirajućim. Pored naziva neterminirajućih simbola, potrebno je navesti i njihov tip, koji se uvodi odgovarajućim poljem u telu *%union* direktive [4]. Sintaksa navođenja neterminirajućih simbola je sledeća:

```
%type <tip> simbol_1, ..., simbol_n
```

%left, %right, %nonassoc

Navedene direktive pridružuju asocijativnost terminirajućim simbolima. Redosled navođenja direktiva utiče na prioritet simbola. Svim simbolima, koji su deklarisani u okviru jedne direktive, pridružuje se isti prioritet. Ako su simboli deklarisani u više od jednog nivoa, prvi nivo pridružuje najniži, dok poslednji nivo pridružuje najviši prioritet. U narednom primeru, tokeni *PLUS* i *MINUS* imaju niži prioritet od tokena *MUL*, dok je asocijativnost sva tri tokena identična:

```
%left PLUS MINUS
%left MUL
```

Treba napomenuti da dodeljivanje asocijativnost i prioriteta operatorima i pravilima predstavlja mehanizam za razrešavanje konflikata. Prioritet pravila moguće je promeniti direktivom *%prec*, koja određenom pravilu pridružuje prioritet tokena, navedenog u okviru direktive [4].

%start

Neterminirajući simbol, naveden ovom direktivom, predstavlja aksiomu gramatike. Ako se direktiva izostavi, aksiomom gramatike smatra se prvi neterminirajući simbol naveden na levoj strani prvog pravila [4]. Sintaksa navođenja aksiome gramatike je sledeća:

```
%start simbol
```

%union

Svakom simbolu gramatike (terminirajućem ili neterminirajućem) može biti pridružena odgovarajuća vrednost. Stoga, Bison omogućava lako navođenje

tipova podataka koji će biti dodeljeni određenim simbolima. Svi korišćeni tipovi podataka navode se u telu *%union* direktive. Sintaksa je sledeća:

```
%union {
    tip_1 identifikator_1;
    ...
    tip_n identifikator_n;
}
```

Telo direktive doslovno se prepisuje u uniju programskog jezika C, čiji je naziv *YYSTYPE*. Pridruživanje tipova simbolima vrši se na različite načine, u zavisnosti od vrste simbola. Za neterminirajuće simbole koristi se direktiva *%type*:

```
%type <identifikator_i> simbol_1, ..., simbol_n
```

Za terminirajuće simbole mogu se iskoristiti sledeće direktive: *%token*, *%left*, *%right*, *%nonassoc*. Sintaksa je ista za sve navedene direktive:

```
%token <identifikator_i> simbol_1, ..., simbol_n
```

Prilikom referisanja na simbole posredstvom pseudo-promenljivih (*$i*), Bison automatski koristi odgovarajuće polje unije. Ukoliko se izostavi navođenje *%union* direktive, podrazumevani tip unije je *int*, što znači da su sve vrednosti simbola celobrojne [4].

Deo za pravila

Deo za pravila predstavlja mesto gde se kontekstno slobodnom gramatikom (eng. *context-free grammar*) opisuje ulazni jezik (eng. *input language*). Gramatika ulaznog jezika predstavljena je skupom gramatičkih pravila. Svakom pravilu se pridružuje odgovarajuća akcija koju čine određene instrukcije. Kada se deo prosleđenog ulaza prepozna nekim od navedenih pravila, akcija pridružena tom pravilu biva izvršena.

3.2.2 Zapis gramatičkih pravila

Najčešći formalni sistem za predstavljanje gramatičkih pravila je Bakus—Naurova forma (eng. *Backus—Naur form*), čija se uprošćena varijanta koristi pri-

likom navođenja specifikacije parsera [3]. Svako gramatičko pravilo određeno je svojom levom i desnom stranom, koje su međusobno razdvojene *:* simbolom, dok se završetak pravila označava *;* simbolom. Levu stranu pravila čine neterminirajući simboli, dok se na desnoj strani mogu navoditi neterminirajući, terminirajući simboli, literali i akcije. Simboli prepoznati od strane skenera nazivaju se terminirajućim ili tokenima. Pod literalom se podrazumeva karakter naveden između jednostrukih navodnika. Akcije predstavljaju programski kôd naveden između vitičastih zagrada, na desnoj strani pravila. Ako uzastopna pravila imaju identičnu levu stranu, samo je za prvo pravilo neophodno navesti istu, dok naredna pravila mogu početi / simbolom. Radi poboljšanja čitljivosti, u praksi postoji konvencija po kojoj se neterminirajući označavaju malim, a terminirajući simboli velikim slovima [4]. U nastavku je dat primer gramatike kojom se opisuje prikaz dva vremenska formata:

```
vreme : SAT ':' MINUT
      | SAT 'h' MINUT
      ;
```

Svaki simbol gramatičkog pravila može da sadrži određenu vrednost. Vrednostima simbola, koji se nalaze na levoj strani pravila, pristupa se pomoću pseudo-promenljive *$$*, dok se vrednostima simbola, koji sa nalaze sa desne strane pravila, pristupa pomoću pseudo-promenljivih *$i*, gde *i* označava redni broj simbola[2]. Prosleđivanje vrednosti tokena, između skenera i parsera, vrši se posredstvom promenljive *yylval* [4].

3.2.3 Opis rada generisanog parsera

Parser se sastoji od niza stanja. Pozivom funkcije *yyparse* započinje se parsiranje ulazne struje karaktera (eng. *input stream*). Ukoliko se parsiranje uspešno završi, funkcija vraća vrednost nula, dok se u slučaju greške vraća nenula vrednost. Kada parser, posredstvom funkcije *yylex*[3], naiđe na token koji ne može da kompletira desnu stranu nijednog od navedenih gramatičkih pravila, pročitani token biva postavljen na interni stek i prelazi se u novo stanje. Suprotno, kada se naiđe na token koji kompletira desnu stranu nekog od navedenih gramatičkih pravila,

[2]Numeracija simbola kreće od vrednosti 1, tako da je vrednost prvog simbola sa desne strane *$1*.

[3]Eksplicitno definisanom ili generisanom pomoću alata Flex.

svi simboli koji pripadaju desnoj strani pravila bivaju skinuti sa steka, a na stek se postavlja simbol koji se nalazi sa leve strane prepoznatog pravila i prelazi se u novo stanje. Prva od navedenih akcija naziva se potiskivanjem (eng. *shift*), a druga redukcijom (eng. *reduce*). Svaki put kada se vrši redukcija, deo koda pridružen prepoznatom pravilu biva izvršen. Ako se prilikom parsiranja dogodi sintaksička greška, poziva se funkcija *yyerror* [4].

3.2.4 Konflikti

Pojava konflikata česta je pri konstrukciji parsera. Višeznačne (eng. *ambiguous*) gramatike su one koje ne grade jedinstveno sintaksičko stablo, pa samim tim dovode do konflikata. Štaviše, iako gramatika nije višeznačna, konflikti se mogu ispoljiti usled nedovoljnog broja preduvidnih (eng. *look-ahead*) tokena. Postoje dve vrste konflikata [4]:

- **shift/reduce** konflikti — nastaju u slučaju kada se parser nađe u stanju gde ima mogućnost da izvrši obe od prethodno navedenih akcija;
- **reduce/reduce** konflikti — nastaju u slučaju kada se parser nađe u stanju gde ima mogućnost da izvrši redukciju na osnovu više zadatih pravila.

3.2.5 Povezivanje sa skenerom generisanim alatom Flex

Da bi se izvršilo ispravno povezivanje skenera (generisanog alatom Flex) i parsera (generisanog alatom Bison), svi kodovi terminirajućih simbola moraju biti poznati skeneru. Navođenje opcije *-d*, prilikom generisanja parsera, instruiše alat Bison da generiše datoteku zaglavlja koja sadrži definicije svih terminirajućih simbola prisutnih u gramatici ulaznog jezika. Generisana datoteka zaglavlja se tada može uključiti u datoteku kojom se zadaje specifikacije skenera, a njen naziv se može postaviti *%defines* direktivom [4].

3.3 Infrastruktura LLVM

Projekat LLVM, započet kao istraživački projekat na Univerzitetu Ilinois, predstavlja skup modularnih i ponovno iskoristivih kompajlerskih tehnologija, čiji je cilj podrška statičkoj i dinamičkoj kompilaciji proizvoljnih programskih jezika [7].

Inicijatori projekta su Kris Latner (eng. *Chris Lattner*) i Vikram Adve (eng. *Vikram Adve*) [8].

Najvažniji delovi infrastrukture su [8]:

Prednji deo (eng. *frontend*)
Uloga prednjeg dela je prevođenje i analiza izvornog koda, programa napisanih u višim programskim jezicima[4], u LLVM-ovu međureprezentaciju (eng. *intermediate representation*). Prevođenje obuhvata leksičku, sintaksičku i semantičku analizu, a završava se fazom generisanja LLVM-ovog međukoda.

Središnji deo (eng. *middleend*)
Mesto sprovođenja mnogih mašinski nezavisnih optimizacija predstavlja središnji deo. U nastavku je dat detaljniji opis organizacije, sadržine i razloga za njegovo uvođenje.

Zadnji deo (eng. *backend*)
Generisanje mašinskog koda za navedenu ciljnu arhitekturu[5], na osnovu prosleđenog LLVM-ovog međukoda, vrši se u zadnjem delu infrastrukture. Takođe, u ovom delu se vrše mnoge mašinski zavisne optimizacije.

3.3.1 Središnji deo LLVM-ove arhitekture

Postojanje središnjeg dela omogućava efektivnu obradu više različitih ulaznih jezika i generisanje mašinskog koda za različite tipove arhitektura, posredstvom jedinstvenog kompilatora[6]. LLVM-ova međureprezentacija predstavlja sponu između prednjeg i zadnjeg dela kompilatora, gde se izvršava najveći broj mašinski nezavisnih optimizacija koda. Inicijalna ideja projekta bila je istraživanje doprinosa koji se ostvaruje sprovođenjem optimizacija na nižim nivoima apstrakcije (u odnosu na izvorni jezik) i razvoj optimizacija koje se vrše u fazi povezivanja (eng. *link-time optimizations*). Izbor odgovarajuće međureprezentacije predstavlja veoma važnu odluku koja direktno utiče na količinu informacija dostupnih u fazi optimizacije. Ako je međureprezentacija veoma visokog nivoa, optimizatori će sa lakoćom izdvojiti suštinski bitne informacije iz izvornog koda. U suprotnom, ako je međureprezentacija veoma niskog nivoa, kompilator će biti u prilici da generiše

[4]Neki od trenutno podržanih programskih jezika su: C, C++, Haskell, Fortran, Pascal, Swift.
[5]Neke od trenutno podržanih arhitektura su: MIPS, ARM, x86, SPARC.
[6]Takozvani retargetabilni (eng. *retargetable*) tip kompilatora.

mašinski kôd fino prilagođen ciljnoj arhitekturi. Kako je izgradnja opisanog tipa kompilatora veoma izazovna, prvenstveno zbog odlike da zadnji deo kompilatora mora generisati mašinski kôd za različite tipove arhitektura, došlo se do rešenja koje podrazumeva postojanje opšte međureprezentacije, koja je zajednička za sve zadnje delove kompilatora. Da bi se izbeglo preprilagođavanje određenoj arhitekturi, a sa druge strane sprečilo podizanje nivoa aprstakcije, kompilatori takođe poseduju specifične međureprezentacije, kako bi se optimizacije mogle sprovoditi i na nižim nivoima apstrakcije. Kako bi se napravila jasna razlika između zvanične LLVM-ove i drugih međureprezentacija, u nastavku teksta će prva međureprezentacija biti označavana pojmom *međukod* [8].

LLVM-ov međukod može biti predstavljen pomoću narednih, međusobno ekvivalentnih formi [8]:

- **memorijska** (eng. *in-memory*) **reprezentacija** — implicitna forma međukoda, smeštena u radnoj memoriji računara;
- **bitkod** (eng. *bitcode*) **reprezentacija** — prostorno efikasna reprezentacija, smeštena u datotekama ekstenzije *bc*;
- **pseudo-asemblerska reprezentacija** — lako čitljiva, pseudo-asemblerska (eng. *pseudo-assembly*) reprezentacija, smeštena u datotekama ekstenzije *ll*.

3.3.2 Organizacija i sintaksa LLVM-ovog međukoda

LLVM-ov međukod organizovan je u hijerarhijsku strukturu koja se sastoji od nekoliko važnih entiteta. Sadržaj datoteke u kojoj se nalazi međukod definiše *modul*, koji predstavlja najviši entitet hijerarhije [8]. Modul odgovara jednoj jedinici prevođenja (eng. *translation unit*) i sačinjen je od proizvoljnog broja *funkcija* i drugih entiteta. Funkcije konceptualno odgovaraju funkcijama programskog jezika C [10]. Naredni entitet u hijerarhiji predstavljaju *osnovni blokovi* (eng. *basic blocks*), od kojih je sačinjena svaka funkcija. Osnovni blok predstavlja najduži niz troadresnih instrukcija koje se izvršavaju sekvencijalno. Važnu karakteristiku osnovnog bloka predstavlja činjenica da se u njega može uskočiti jedino sa početka, a iskočiti jedino sa kraja istog [1]. Hijerarhijski najniže entitete predstavljaju same *instrukcije*.

Najzastupljeniju vrstu podataka čine virtuelni registri, koji odgovaraju promenljivama međukoda i čiji naziv počinje % simbolom. Osnovne karakteristike međukoda su [8]:

SSA forma

Predstavljanje međukoda SSA (eng. *static single assignment*) formom uprošćava sprovođenje mnogih optimizacija. Forma podrazumeva da se svakoj promenljivi vrednost dodeljuje na tačno jednom mestu u programu. Postojanje ϕ-funkcija predstavlja još jedno bitno svojstvo SSA forme, a u nastavku sledi jedan od razloga za njihovo uvođenje. Neka se izmena vrednosti promenljive t vrši u svakom od mogućih tokova programa, određenih naredbom grananja višeg programskog jezika. S obzirom na prvo svojstvo SSA forme, u svakom od mogućih tokova biće definisana po jedna promenljiva t_i, koja predstavlja vrednost promenljive t u toku i. Kako bi se znalo koju od promenljivih t_i koristiti nakon izvršavanja naredbe grananja, uvodi se odgovarajuća ϕ-funkcija čiji su argumenti promenljive t_i. Ukoliko je izvršen tok i, njena povratna vrednost biće upravo promenljiva t_i [1].

Troadresne instrukcije

LLVM-ov međukod predstavlja linearni, troadresni kôd (eng. *three-address code*). Svojstvo linearnosti odnosi se na izvršavanje instrukcija redom kojim su navedene u programu. Većina instrukcija troadresnog koda, tj. troadresnih instrukcija, predstavljena je formom $t_3 := t_1\ op\ t_2$, gde se rezultat operacije op, izvršene nad operandima t_1 i t_2, smešta u t_3. Neke instrukcije zahtevaju manji broj operanada. Kompaktnost zapisa, omogućavanje različitih vrsta optimizacija i implementacija troadresnih instrukcija u mnoge moderne procesore, čine troadresni kôd veoma poželjnom formom međureprezentacije [2].

„Beskonačan" broj virtuelnih registara

Virtuelni registri mogu biti označeni bilo kojim nazivima, koji počinju prefiksom %. Broj registara jedino je ograničen memorijom računara.

Naredni fragment međukoda predstavlja definiciju funkcije *oduzmi*, koja prihvata dva celobrojna argumenta širine četiri bajta i vraća celobrojni rezultat iste širine:

```
define i32 @oduzmi(i32 %0, i32 %1) {
```

```
entry:
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    store i32 %1, i32* %3, align 4
    %4 = load i32, i32* %2, align 4
    %5 = load i32, i32* %3, align 4
    %6 = sub nsw i32 %4, %5
    ret i32 %6
}
```

Kao što se u primeru može videti, lokalni identifikatori imaju prefiks %, a globalni prefiks @. LLVM podržava bogat skup tipova, od kojih su najkorišćeniji:

- celobrojni tipovi proizvoljne širine — *i1*, *i8*, *i16*, *i32*, *i64*, *i128*
- tipovi podataka zapisanih u pokretnom zarezu — *float*, *double*
- vektorski tipovi — vektor koji sadrži deset celobrojnih elemenata širina osam bita predstavlja se kao *<10 x i8>*

Funkcija iz primera sadrži jedan osnovni blok, čiji početak je označen labelom *entry*, a kraj instrukcijom *ret*. U primeru se mogu videti naredne instrukcije: *alloca* (rezerviše prostor na stek okviru funkcije), *load* (vrši iščitanje iz memorije), *store* (vrši upis u memoriju), *sub* (vrši operaciju oduzimanja), *ret* (vraća kontrolu toka izvršavanja programa pozivaocu funkcije i (opciono) navedenu vrednost) [8].

Memorijska reprezentacija međukoda vrlo blisko modeluje prikazanu sintaksu. Klase za izgradnju memorijske reprezentacije nalaze se u zaglavlju *include/llvm/IR*, a najvažnije od njih su: *Module*, *Function*, *BasicBlock* i *Instruction*. Nazivi klasa jasno ukazuju na svrhu njihove primene prilikom konstruisanja implicitne forme međukoda [8].

3.3.3 Pregled korišćenih alata

U ovom odeljku je dat pregled dva, u radu korišćena alata LLVM-ove infrastrukture. Predstavljeni alati su korišćeni prilikom optimizacije međukoda i generisanja asemblerskog koda za ciljnu arhitekturu opisanu u poglavlju 2.

opt

Alat *opt* predstavlja LLVM-ov optimizator (i analizator) koji sprovodi određena unapređivanja (i analize) međukoda. Neke od opcija koje mu mogu biti prosleđene su: *-S* (ukoliko je opcija navedena, izlaz će biti generisan na LLVM-ovoj pseudo-asemblerskoj reprezentaciji), *{-passname}* (navodi se naziv odgovarajućeg optimizacionog prolaza koji treba izvršiti) [6].

llc

Alat *llc* predstavlja LLVM-ov statički kompilator koji vrši prevođenje međukoda u asemblerski kôd navedene ciljne arhitekture. Neke od opcija koje mu mogu biti prosleđene su: *-march* (ciljna arhitektura), *-mcpu* (model procesora), *-filetype* (tip izlazne datoteke), *-O* (nivo optimizacije) [6].

Glava 4

Opis programskog jezika

Programski jezik aKcent pripada grupi imperativnih, statički tipiziranih programskih jezika. Razvijen je sa ciljem da pojednostavi i približi početnicima osnovne koncepte programiranja mikrokontrolera. Ideja za njegov nastanak dolazi od Arduino[1] i PICAXE[2] mikrokontrolerskih platformi, od kojih je prva zasnovana na programskom jeziku C++, a druga na programskom jeziku BASIC. Sam jezik predstavlja kompromis između jednostavnosti i ekspresivnosti, kao bitnih osobina koje karakterišu programske jezike.

4.1 Tipovi podataka

Programski jezik podržava nekoliko tipova podataka različitih opsega i preciznosti. Detaljnije informacije prikazane su u tabeli 4.1. Kombinovanje odgovarajućih numeričkih tipova podataka omogućeno je kroz implicitne konverzije, kojima se uži tipovi podataka konvertuju u šire i obrnuto. Prilikom konverzije se ispisuje odgovarajuća poruka o vrsti i poziciji primenjene konverzije. Ukoliko konverzija datih tipova nije moguća, ispisuje se poruka o nastaloj grešci. Tip celobrojnih numeričkih literala je *i16*, a tip numeričkih literala u pokretnom zarezu je *real*. Sintaksičke oznake tipova podataka motivisane su sintaksičkim oznakama tipova podataka koje se koriste u LLVM-ovom međukodu.

[1] Više informacija nalazi se na narednoj adresi: `https://www.arduino.cc/`
[2] Više informacija nalazi se na narednoj adresi: `http://www.picaxe.com/`

Tabela 4.1: Tipovi podataka

Oznaka	*Opis*	*Opseg*
bool	Logički tip.	{true, false}
i8	Celobrojni tip širine jednog bajta.	[-128, 127]
i16	Celobrojni tip širine dva bajta.	[-32768, 32767]
i32	Celobrojni tip širine četiri bajta.	[-2147483648, 2147483647]
real	Tip podataka zapisanih u pokretnom zarezu.	Jednostruka preciznost.

4.2 Funkcije

Funkcije omogućavaju particionisanje programa u odgovarajuće logičke celine, kako bi razvoj i održavanje programa bili efektivniji. Izdvajanje programskog koda u funkcije pospešuje čitljivost i doprinosi modularnosti programa, koja otvara vrata ponovnoj upotrebljivosti jednom napisanog koda.

Definiciju funkcije čine: tip povratne vrednosti, naziv, parametri i telo funkcije. Tip povratne vrednosti može biti neki od tipova iz tabele 4.1, kao i odgovarajući pokazivački tipovi. Specijalni tip *void* označava da funkcija ne treba da vraća rezultat. Naziv funkcije mora biti jedinstven u odnosu na nazive ostalih funkcija. Parametri funkcije razdvojeni su zapetama, a navode se između malih zagrada, nakon imena funkcije. Deklaracija parametra podrazumeva navođenje njegovog identifikatora, nakon čega slede simbol *:* i tip parametra. Tip parametra može biti neki od tipova iz tabele 4.1, kao i odgovarajući pokazivački tip. Funkcija ne mora imati nijedan parametar ili ih može imati više (u drugom slučaju neophodno je da identifikatori parametara budu međusobno različiti). Telo funkcije navodi se između velikih zagrada i čine ga naredbe koje funkcija treba da izvrši. Naredbe su međusobno razdvojene simbolom *;*. Argumenti funkcija prenose se po vrednosti. Redosled navođenja definija funkcija u odnosu na njihove pozive potpuno je proizvoljan. Sintaksa funkcijskih definicija najvećim delom je preuzeta iz programskog jezika C.

Neophodno je da svaki program ima definisanu funkciju *main*, od koje počinje izvršavanje programa, a koja ne sadrži parametre i ne vraća rezultat.

Pored standardnih funkcija programski jezik podržava i *korisnički definisane* funkcije, koje predstavljaju mehanizam kojim se jezik može dodatno proširiti funkcijama napisanim u programskom jeziku C.

4.2.1 Korisnički definisane funkcije

Kako bi se omogućilo jednostavnije programiranje mikrokontrolera, bilo je potrebno apstrahovati njihove često korišćene funkcionalnosti. Najznačajnije su konfigurisanje pinova i podešavanje perifernih hardverskih komponenti mikrokontrolera. Stoga, razvijena je pomoćna biblioteka (napisana na programskom jeziku C) koja nudi veliki broj funkcija koje se mogu jednostavno pozivati u ovom programskom jeziku. Tabela 4.2 sadrži spisak funkcija i njihov kratak opis. Pošto spisak korisnički definisanih funkcija ne pokriva sve funkcionalnosti mikrokontrolera, bilo je neophodno omogućiti mehanizam pomoću kog je moguće dodati nove funkcije postojećoj biblioteci.

Kako nove funkcije nisu implicitno deklarisane u samom jeziku, za razliku od funkcija iz tabele 4.2, potrebno ih je eksplicitno deklarisati. Deklaracija se vrši navođenjem ključnih reči *extern function*, nakon čega slede tip povratne vrednosti, naziv i parametri funkcije koja se deklariše. Deklaracija se završava simbolom *;*. Sledi primer deklarisanja korisnički definisane funkcije:

```
extern function i32 naziv_funkcije(param_1: real, param_2: i16*);
```

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se deklariše korisnički definisana funkcija *available*, čija se definicija nalazi u pomoćnoj biblioteci.

```
extern function i8 available();
```

Kao i u slučaju standardnih funkcija, redosled navođenja deklaracija u odnosu na pozive funkcija potpuno je proizvoljan.

Tabela 4.2: Spisak implicno deklarisanih, korisnički definisanih funkcija

Sintaksički zapis	*Opis*
input *izraz_1, ..., izraz_n*	Funkcija *input* konfiguriše navedene pinove kao ulazne. Ako je pin konfigurisan kao ulazni, sa njega je moguće očitavati logičku vrednost. Vrednost izraza *izraz_i* predstavlja redni broj odgovarajućeg pina.

output *izraz_1, ..., izraz_n*	Funkcija *output* konfiguriše navedene pinove kao izlazne. Ako je pin konfigurisan kao izlazni, u njega je moguće upisivati logičku vrednost. Vrednost izraza *izraz_i* predstavlja redni broj odgovarajućeg pina.
high *izraz_1, ..., izraz_n*	Funkcija *high* upisuje logičku jedinicu u navedene pinove. Vrednost izraza *izraz_i* predstavlja redni broj odgovarajućeg pina.
low *izraz_1, ..., izraz_n*	Funkcija *low* upisuje logičku nulu u navedene pinove. Vrednost izraza *izraz_i* predstavlja redni broj odgovarajućeg pina.
toggle *izraz_1, ..., izraz_n*	Funkcija *toggle* invertuje logičku vrednost dodeljenu navedenim pinovima. Vrednost izraza *izraz_i* predstavlja redni broj odgovarajućeg pina.
read *izraz*	Funkcija *read* očitava logičku vrednost sa navenog pina. Vrednost izraza *izraz* predstavlja redni broj odgovarajućeg pina. Povratna vrednost je tipa *bool*.
init_pwm *izraz*	Funkcija *init_pwm* inicijalizuje PWM funkcionalnost na odgovarajućem pinu. Vrednost izraza *izraz* predstavlja redni broj odgovarajućeg pina.
pwm *izraz vrednost*	Funkcija *pwm* primenjuje PWM vrednost na zadati pin. Vrednost izraza *izraz* predstavlja redni broj odgovarajućeg pina. Vrednost izraza *vrednost* mora biti u intervalu [0, 255].
setInternalAREF	Funkcija *setInternalAREF* podešava referentnu vrednost, koju koristi funkcija *adc* za očitavanje napona na određenim pinovima, na 5 V.

setExternalAREF	Funkcija *setExternalAREF* podešava referentnu vrednost, koju koristi funkcija *adc* za očitavanje napona na određenim pinovima, na vrednost napona dovedenog na AREF pin mikrokontrolera.
adc *izraz*	Funkcija *adc* očitava napon na navedenom pinu. Vrednost izraza *izraz* predstavlja redni broj odgovarajućeg pina. Povratna vrednost je celobrojnog tipa iz intervala [0, 1023].
wait *izraz*	Funkcija *wait* blokira izvršavanje programa na određeni vremenski period. Vrednost izraza *izraz* predstavlja vreme izraženo u milisekundama.
time	Funkcija *time* vraća proteklo vreme od uključivanja mikrokontrolera, izraženo u milisekundama.
utime	Funkcija *utime* vraća proteklo vreme od uključivanja mikrokontrolera, izraženo u mikrosekundama.
begin *izraz*	Funkcija *begin* konfiguriše brzinu serijskog prenosa podataka. Izraz *izraz* predstavlja neku od sledećih vrednosti, izraženu u baudima: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 115200.
receive	Funkcija *receive* prihvata podatak veličine jednog bajta smešten u dolazni bafer serijskog porta. Povratna vrednost je celobrojnog tipa i8.
send *bajt_1, ..., bajt_n*	Funkcija *send* prosleđuje podatak veličine jednog bajta u odlazni bafer serijskog porta.

4.3 Promenljive i doseg

Promenljive predstavljaju objekte koji čuvaju vrednost odgovarajućeg tipa, koja se može iščitavati ili menjati. Da bi se promenljive mogle koristiti neophodno ih je deklarisati.

Deklaraciju promenljive čine identifikator promenljive, nakon kog se navode simbol *:* i tip vrednosti koju promenljiva sadrži. Prilikom deklaracije, promenljivu je moguće inicijalizovati na odgovarajuću vrednost. Svaka deklarisana promenljiva ima doseg nivoa bloka, kojim se podrazumeva da naziv promenljive važi od tačke deklarisanja pa sve do kraja bloka u kom je promenljiva deklarisana.

4.3.1 Pokazivačke promenljive

Pokazivači su specijalni tip podataka čije su vrednosti memorijske adrese veličine šesnaest bita. Za sve tipove podataka iz tabele 4.1 dostupni su odgovarajući pokazivački tipovi. Pokazivačke promenljive mogu da sadrže adresu promenljivih ili elemenata niza. Takođe, omogućen je rad sa višestrukim pokazivačima.

Deklarisanje pokazivačke promenljive vrši se na sledeći način:

```
identifikator: tip*;
```

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se deklariše promenljiva *x* koja predstavlja pokazivač na podatak tipa *i8*.

```
x: i8*;
```

Određivanje adrese odgovarajuće promenljive ili elementa niza vrši se navođejem operatora *&*, ispred identifikatora te promenljive ili elementa niza. Dereferenciranje odgovarajuće pokazivačke promenljive vrši se navođenjem operatora ˆ, ispred identifikatora promenljive.

4.3.2 Nizovi

Nizovi se deklarišu na sledeći način:

```
identifikator tip[dimenzija];
```

Pristup elementu niza vrši se navođenjem identifikatora niza, nakon čega se u srednjim zagradama navodi nenegativni indeks elementa kom se pristupa.

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se deklariše niz *a* koji se sastoji od deset elemenata tipa *i32*.

```
a i32[10];
```

4.4 Operatori

U tabeli 4.3 dat je spisak podržanih operatora, njihov sintaksički zapis, prioritet, asocijativnost i opis. Prioritet operatora opada od vrha ka dnu tabele.

Aritmetički operatori se primenjuju nad numeričkim operandima, a rezultat primene operatora je takođe numeričkog tipa. Relacijski operatori se primenjuju nad numeričkim operandima, a rezultat primene operatora je tipa *bool*. Logički operatori se primenjuju nad operandima tipa *bool*, a rezultat primene operatora je takođe tipa *bool*.

Tabela 4.3: Spisak operatora

Zapis	*Opis*	*Asocijativnost*
- not ^ &	aritmetički unarni operator oduzimanja logički unarni operator negacije operator dereferenciranja operator adresiranja	desno-asocijativni desno-asocijativni desno-asocijativni desno-asocijativni
* / %	aritmetički binarni operator množenja, deljenja i ostatka pri deljenju	levo-asocijativni
+ -	aritmetički binarni operatori sabiranja i oduzimanja	levo-asocijativni
< >	relacijski binarni operator manje i veće	levo-asocijativni
== !=	relacijski binarni operator jednakosti i nejednakosti	levo-asocijativni
and	logički binarni operator konjukcije	levo-asocijativni
or	logički binarni operator disjunkcije	levo-asocijativni
<=>	logički binarni operator ekvivalencije	levo-asocijativni
=	operator dodele	desno-asocijativni

,	zapeta	levo-asocijativni

4.5 Naredbe za kontrolu toka izvršavanja programa

U nastavku su opisane implementirane naredbe za kontrolu toka izvršavanja programa kojima jezik raspolaže. Sintaksa većine naredbi motivisana je sintaksom sličnih naredbi, koje se koriste u postojećim programskim jezicima.

4.5.1 Naredba if-else

Naredba grananja *if-else* ima sledeći oblik:

```
if (izraz)
   naredba
else
   naredba
```

Uslovni izraz *izraz* predstavlja proivoljni izraz koji se izračunava u vrednost tipa *bool*. Deo koji sadrži *else* granu nije neophodno navoditi. Naredba *naredba* može biti blok naredbi ili pojedinačna naredba.

Ukoliko uslovni izraz ima više od dva nivoa odlučivanja, moguće je koristiti narednu konstrukciju:

```
if (izraz_1)
   naredba
else if (izraz_2)
   naredba
else if (izraz_3)
   naredba
else
   naredba
```

Poslednja naredba će biti izvršena ukoliko nijedan od prethodnih uslova nije zadovoljen. Sintaksa je motivisana sintaksom naredbe *if-else*, koja se koristi u programskom jeziku C.

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se promenljivoj q dodeljuje odgovarajuća vrednost, u zavisnosti od vrednosti promenljive p.

```
p: i8 = 0;
q: i16;

if (p > 0)
   q = 1;
else if (p == 0)
   q = 0;
else
   q = -1;
```

4.5.2 Petlja while

Petlja *while* ima sledeći oblik:

```
while (izraz) do
   naredba
```

Kao i kod *if-else* naredbe, uslovni izraz *izraz* predstavlja proivoljni izraz koji se izračunava u vrednost tipa *bool*. Na početku izvršavanja petlje, vrši se izračunavanje istinitosne vrednosti uslovnog izraza. Ako je izračunata vrednost *netačno*, izvršavanje petlje se prekida, a nastalja se od naredbe koja sledi nakon tela petlje. Ako je izračunata vrednost *tačno*, izvršava se naredba *naredba*, nakon čega sledi ponovno izračunavanje uslovnog izraza na osnovu čije vrednosti se odlučuje dalji tok izvršavanja programa. Naredba *naredba* može biti blok naredbi ili pojedinačna naredba. Sintaksa je motivisana sintaksom petlje *while*, koja se koristi u programskom jeziku C.

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se vrši inkrementiranje vrednosti promenljive x, dok god joj je vrednost manja od pet.

```
x: i8 = 0;
```

```
while (x < 5) do
    x = x + 1;
```

4.5.3 Petlja forever

U programiranju mikrokontrolera, beskonačna petlja predstavlja neizostavnu konstrukciju. Iako se može realizovati na različite načine, jezik nudi beskonačnu petlju i u eksplicitnom obliku:

```
forever do
    naredba
```

Naredba *naredba* može biti blok naredbi ili pojedinačna naredba. Sintaksa je motivisana makroom *forever*, koji se koristi u frejmvorku Qt[3].

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se vrši pozivanje funkcije *doSomething*, dok god je mikrokontroler uključen.

```
forever do
    doSomething();
```

4.5.4 Petlja for-in

Petlja *for-in* ima sledeći oblik:

```
for x in y...z do
    naredba
```

Promenljiva x predstavlja implicitno deklarisanu brojačku promenljivu čija vrednost, u toku izvršavanja petlje, pripada intervalu određenim izrazima y i z. Prilikom svake iteracije, vrednost promenljive x inkrementira se u odnosu na prethodnu vrednost te promenljive. Telo petlje će se izvršavati sve dok vrednost brojačke promenljive pripada intervalu $[y, z-1]$. Ako vrednost brojačke promenljive nije u zadatom intervalu, izvršavanje programa se nastavlja od naredbe koja sledi nakon tela petlje. Naredba *naredba* može biti blok naredbi ili pojedinačna naredba.

[3]Više informacija nalazi se na narednoj adresi: `https://www.qt.io/`

Sintaksa je motivisana sintaksom petlje *for*, koja se koristi u programskom jeziku Rust[4].

Segment koda koji ilustruje navedenu konstrukciju nalazi se u nastavku teksta. Datom naredbom se vrši inkrementiranje celobrojne promenljive x, dok god joj vrednost pripada intervalu $[1, 9]$.

```
for x in 1...10 do
    x = x + 1;
```

[4] Više informacija nalazi se na narednoj adresi: https://www.rust-lang.org/en-US/

Glava 5

Implementacija kompilatora

Kompilator za programski jezik opisan u radu implementiran je u programskom jeziku C++, posredstvom alata opisanih u glavi 3. U nastavku teksta dat je opis različitih faza prevođenja i način na koji je implementirana pomoćna biblioteka. Nakon faze generisanja LLVM-ovog međukoda, sa prevođenjem se nastavlja korišćenjem gotovih alata koje pomenuta infrastruktura pruža, što će biti opisano u poglavlju 5.7. Izvorni kôd programskog prevodioca i primeri programa nalaze se na narednoj adresi: `https://bitbucket.org/user0x7351/rad_1091_2016/`.

5.1 Leksička analiza

Leksička analiza predstavlja početnu fazu kompilacije. Iščitavanje ulazne struje karaktera (od kojih je ulazni program sačinjen) i njihovo grupisanje u smislene celine koje nazivamo leksemama (eng. *lexemes*) glavni su zadatak leksičke analize. Za svaku prepoznatu leksemu leksički analizator generiše token (eng. *token*) koji predstavlja ulaz naredne faze kompilacije — sintaksičke analize [1]. Svaki token čine dve komponente: tokenska klasa (ključna reč, identifikator, numerički literal, itd.) i odgovarajuća leksema koja joj je pridružena.

Leksički analizator implementiran je posredstvom alata Flex. Datoteka *lexer.lex* sadrži specifikaciju leksera u formatu opisanom u poglavlju 3.1.1.

Jezik se sastoji od sledećih leksema: **function**, **extern**, **ret**, **forever**, **while**, **for**, **in**, **do**, **if**, **else**, **...**, **i8**, **i16**, **i32**, **void**, **bool**, **real**, **true**, **false**, **and**, **or**, **not**, <=>, +, **-**, *, /, %, &, <, >, ==, **!=**,^, **{**, **}**, (,), [,], **.**, **,**, **;**, **:**, = i naziva korisnički definisanih funkcija, čiji su identifikatori dati u tabeli 4.2.

5.2 Sintaksička analiza

Primarni zadatak sintaksičke analize[1] je da odredi da li se ulazni program, reprezentovan tokenima dobijenim iz prethodne faze prevođenja, uklapa u navedenu gramatiku programskog jezika [2]. Gramatika pruža precizan i razumljiv način specifikacije sintakse programskog jezika [1].

Sintaksički analizator implementiran je posredstvom alata Bison. Datoteka *parser.ypp* sadrži specifikaciju parsera (tj. gramatiku programskog jezika) u formatu opisanom u poglavlju 3.2.1.

5.2.1 Apstraktno sintaksičko stablo

Proizvod sintaksičke analize je apstraktno sintaksičko stablo (eng. *abstract syntax tree*) kojim je ulazni program predstavljen. Razlog za korišćenje apstraktnog sintaksičkog stabla je izostavljanje nepotrebnih sintaksičkih detalja koji nisu relevantni za dalje faze prevođenja. Izgradnja stabla vrši se eksplicitno u toku parsiranja. Kada se određeno gramatičko pravilo prepozna, akcija koja mu je dodeljena sadrži programski kôd koji konstruiše odgovarajući čvor stabla. U tabeli 5.1 predstavljeni su nazivi i opisi klasa kojima se uvode odgovarajući čvorovi. Dijagram nasleđivanja sa slike 5.1 prikazuje odnose među klasama opisanim u prethodnoj tabeli.

Tabela 5.1: Čvorovi apstraktnog sintaksičkog stabla

Naziv klase	*Opis klase*
BaseNodeAST	Roditeljska klasa svih ostalih klasa stabla.
BaseStatementAST	Naredba programskog jezika.
BaseExpressionAST	Izraz programskog jezika.
BoolAST	Literal logičkog tipa.
ByteAST	Osmobitni celobrojni literal.
WordAST	Šesnaestobitni celobrojni literal.
DWordAST	Tridesetdvobitni celobrojni literal.
RealAST	Literal zapisan u pokretnom zarezu.
BinaryOperatorAST	Binarni operator.
ArithmeticBinaryOperatorAST	Aritmetički binarni operator.

[1]Često se za sintaksičku analizu koristi termin *parsiranje*.

ArithmeticUnaryOperatorAST	Aritmetički unarni operator.
RelationBinaryOperatorAST	Relacijski binarni operator.
LogicBinaryOperatorAST	Logiči binarni operator.
UnaryOperatorAST	Unarni operator.
LogicUnaryOperatorAST	Logički unarni operator.
AdditionAST	Operator sabiranja.
SubtractionAST	Operator oduzimanja.
MultiplicationAST	Operator množenja.
DivisionAST	Operator deljenja.
ModAST	Operator celobrojnog deljenja sa ostatkom.
LessThanAST	Relacijski operator manje.
GreaterThanAST	Relacijski operator veće.
EqualToAST	Relacijski operator jednakosti.
NotEqualToAST	Relacijski operator nejednakosti.
AndAST	Logički operator konjukcije.
OrAST	Logički operator disjunkcije.
NotAST	Logički operator negacije.
NegAST	Aritmetički operator negacije.
EquivalentToAST	Logički operator ekvivalencije.
FunctionCallExpressionAST	Izraz funkcijskog poziva.
LocalVariableAccessAST	Pristup vrednosti lokalne promenljive.
LocalPointerAccessAST	Dereferenciranje pokazivača.
LocalVariableAddressAST	Pristup adresi lokalne promenljive.
ExpressionCommandAST	Ugrađene komande izraza.
BlockAST	Blok naredbi.
PrototypeAST	Prototip funkcije.
FunctionDefinitionAST	Funkcijska definicija.
ReturnStatementAST	Naredba povratka.
WhileAST	Petlja *while*.
ForAST	Petlja *for*.
IfThenElseAST	Naredba grananja.
IOCommandAST	Ugrađene komande naredbe.

FunctionCallStatementAST	Naredba funkcijskog poziva.
LocalVariableDeclarationAST	Deklaracija lokalne promenljive.
AssignmentAST	Naredba dodele lokalnim promenljivama.
PointerAssignmentAST	Naredba dodele pokazivačkim promenljivama.

5.3 Semantička analiza

Semantička analiza predstavlja fazu prevođenja u kojoj se vrše određene vrste semantičkih provera apstraktnog sintaksičkog stabla dostupnog iz prethodne faze prevođenja — sintaksičke analize. Pojava sintaksički ispravnih, a semantički neispravnih konstrukcija, nije ograničena samo na veštačke jezike (poput programskih jezika), već se često javlja i u okviru prirodnih jezika[2]. Semantički analizator implementiran je korišćenjem obrasca za projektovanje poznatijeg kao *posetilac* (eng. *visitor*). Klase zadužene za semantičku analizu nazivaju se *SemanticCheckVisitor* i *PrototypeRegisterVisitor*.

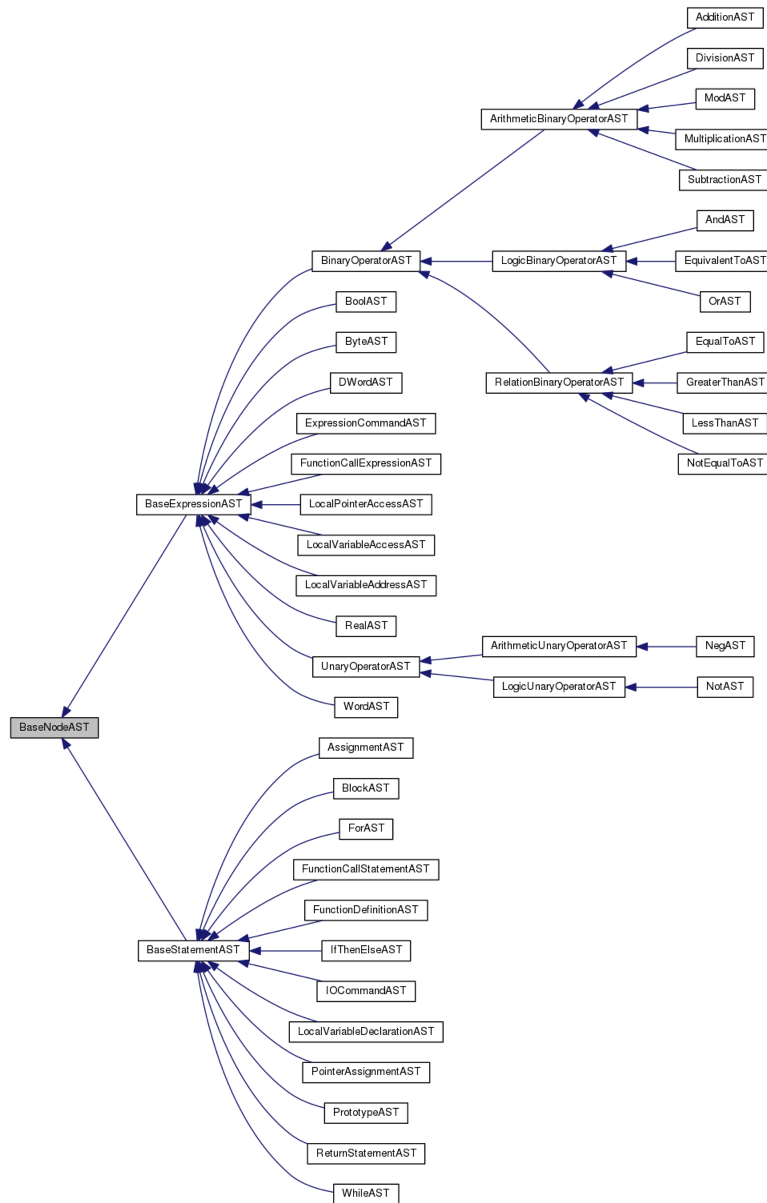
5.3.1 Tablica simbola

Tablica simbola (eng. *symbol table*) predstavlja centralnu strukturu podataka semantičke analize, nad kojom su definisane operacije upisivanja, modifikovanja i pretraživanja podataka koji se u njoj nalaze. Prilikom obilaska sintaksičkog stabla, u njoj se čuvaju određeni podaci o objektima uvedenim u okviru programa koji se obrađuje. Analizator aktivno održava tablice simbola koje sadrže podatke o uvedenim funkcijama i promenljivama. Svaka od navedenih tablica nalazi svoju primenu, kako u proverama opisanim u nastavku teksta, tako i u fazi generisanja međukoda, koja sledi nakon semantičke analize. Klasa kojom je tablica simbola predstavljena naziva se *SymbolTable*.

5.3.2 Registrovanje promenljivih i prototipova funkcija

Tablica simbola za promenljive organizovana je u obliku LIFO (eng. *last in, first out*) strukture. Njene unose predstavljaju vektori promenljivih, koje su deklarisane

[2]Naredna rečenica predstavlja takav primer: *Elokventni lav besno spava.*



Slika 5.1: Dijagram nasleđivanja klasa apstraktnog sintaksičkog stabla

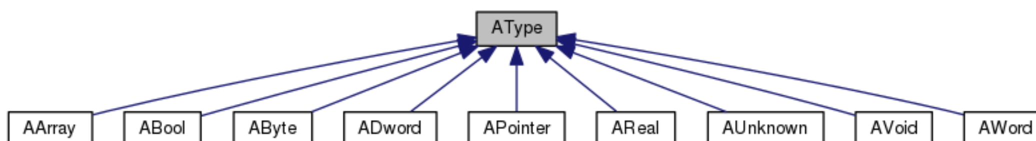
u okviru određenog dela programa. Kada se naiđe na čvor sintaksičkog stabla, koji označava deklaraciju promenljive, u tablicu simbola bivaju upisani njen tip, identifikator i alokator (bitan za fazu generisanja međukoda). Ako se na datu promenljivu kasnije referiše u okviru programa, u tablici simbola se mogu pronaći podaci potrebni za realizaciju odgovarajuće semantičke provere.

Slično se rukuje i sa tablicom simbola za funkcije. Unose predstavljaju prototipovi funkcija na koje se naišlo prilikom obilaska sintaksičkog stabla. Za svaku

funkciju čuvaju se podaci o tipu povratne vrednosti, identifikatoru i tipovima parametara. Faza registrovanja funkcijskih prototipova izvršava se pre svih ostalih semantičkih provera. Time je omogućeno proizvoljno navođenje funkcijskih definicija u odnosu na njihove pozive, jer se podaci o svim uvedenim funkcijama već nalaze u odgovarajućoj tablici simbola i analizator ih može iskoristiti u kasnijim proverama.

5.3.3 Provera tipova

Centralni deo semantičke analize predstavlja provera tipova (eng. *type checking*) u kojoj semantički analizator proverava da li su određenim operatorima dodeljeni odgovarajući operandi [1]. Za svaki od tipova podataka, definisanih u okviru programskog jezika, uvedena su pravila kojima se određuje način na koji objekat datog tipa može interagovati sa ostalim objektima. Na slici 5.2 nalazi se dijagram nasleđivanja klasa koje predstavljaju tipove podataka, a imena i opisi klasa su dati u tabeli 5.2 [3].



Slika 5.2: Dijagram nasleđivanja klasa tipova podataka

Tabela 5.2: Klase tipova podataka

Naziv klase	*Opis klase*
AType	Roditeljska klasa svih ostalih klasa za tipove podataka.
ABool	Logički tip podataka.
AByte	Celobrojni tip podataka širine jednog bajta.
AWord	Celobrojni tip podataka širine dva bajta.
ADword	Celobrojni tip podataka širine četiri bajta.
APointer	Pokazivački tip podataka.
AReal	Tip podataka u pokretnom zarezu, jednostruke preciznosti.
AArray	Tip podataka koji predstavlja jednodimenzione nizove.
AUnknown	Fiktivni tip podataka.

[3]Prefiks *A* u nazivu klasa potiče od naziva programskog jezika — aKcent.

AVoid	Tip podataka koji predstavlja prazan skup vrednosti.

Provera tipova biće objašnjena na narednom fragmentu programskog koda:

```
x = y + z(2, 3);
```

Obilaskom sintaksičkog stabla, analizator nailazi na čvor sintaksičkog stabla koji predstavlja navedenu naredbu dodele. Potrebno je proveriti da li su tipovi leve i desne strane date naredbe kompatibilni. Tip promenljive x može se pročitati iz tablice simbola za promenljive, dok je tip izraza sa desne strane potrebno izračunati (jer inicijalno nije poznat). Kako je navedeni izraz u sintaksičkom stablu predstavljen čvorom operatora sabiranja, analizator vrši proveru tipova njegovih operanada. Tip levog operanda (tj. promenljive y) određuje se iz tablice simbola za promenljive. Pošto je desni operand funkcijski poziv, njegov tip je moguće odrediti iz tablice simbola za funkcije, nalaženjem prototipa date funkcije. Takođe, analizator vrši proveru tipova navedenih argumenata, upoređujući ih sa tipovima parametara funkcije z. Ukoliko su dobijeni tipovi leve i desne strane kompatibilni, analizator može da izvede tip izraza sa desne strane naredbe dodele, koji se zatim upoređuje sa tipom promenljive x, kojoj će izračunata vrednost izraza biti dodeljena.

5.3.4 Kontrola dosega

Kontrola dosega (eng. *scope*) predstavlja mehanizam kojim se određuje deo programa u kome se uvedeni identifikator može koristiti za imenovanje određene promenljive. Kao i u programskom jeziku C, dopušten je konflikt identifikatora, gde je u različitim nivoima dosega moguće koristiti promenljive koje imaju identične nazive. Ukoliko su dosezi tih promenljivih ugnežđeni, promenljiva iz unutrašnjeg dosega prikriva promenljivu iz spoljašnjeg dosega, kada se u okviru unutrašnjeg dosega referiše na nju.

Kontrola dosega realizovana je posredstvom tablice simbola za promenljive. Doseg je predstavljen vektorom identifikatora koji su u okviru njega uvedeni. Kada se unutar jednog dosega uvede novi, na vrh tablice simbola za promenljive postavlja se novi vektor koji će sadržati identifikatore uvedene u okviru tog, novog

dosega. Prilikom napuštanja određenog dosega iz tablice simbola se vrši uklanjanje vektora koji je tom dosegu bio pridružen. Prilikom referisanja na određenu promenljivu pretraga se vrši od najskorijeg ka najstarijem dosegu. Ukoliko ni u jednom dosegu nije pronađen traženi identifikator, ili je u okviru istog pronađen identifikator sa identičnim imenom, kompilacija se obustavlja i generiše se odgovarajuća poruka koja korisnika informiše o nastaloj grešci. Klasa kojom se predstavljaju dosezi naziva se *Scope*.

5.4 Generisanje LLVM-ovog međukoda

Faza generisanja međukoda vrši transformisanje apstraktnog sintaksičkog stabla u LLVM-ov međukod. Generator međukoda implementiran je korišćenjem obrasca za projektovanje poznatijeg kao *posetilac*. Klasa zadužena za generisanje međukoda naziva se *CodeGenVisitor*.

Obilaskom apstraktnog sintaksičkog stabla ulaznog programa, za svaki čvor se poziva odgovarajuća metoda koja generiše međukod koji odgovara konkretnom čvoru. Na taj način se gradi memorijska reprezentacija LLVM-ovog međukoda, koja će u kasnijim fazama biti upotrebljena za generisanje koda za ciljnu arhitekturu. Naredni primer ilustruje generisanje međukoda za *WhileAST* čvor apstraktnog sintaksičkog stabla, kojim je predstavljena *while* petlja sa jednom ulaznom i jednom izlaznom tačkom:

```
llvm::Function *f = _irBuilder.GetInsertBlock()->getParent();
llvm::BasicBlock *whileCondBB =
    llvm::BasicBlock::Create(_llvmContext, "while_cond");
llvm::BasicBlock *whileLoopBB =
    llvm::BasicBlock::Create(_llvmContext, "while_loop");
llvm::BasicBlock *whileContBB =
    llvm::BasicBlock::Create(_llvmContext, "while_cont");

_irBuilder.CreateBr(whileCondBB);
f->getBasicBlockList().push_back(whileCondBB);
_irBuilder.SetInsertPoint(whileCondBB);
llvm::Value *condValue = this->getLLVMValue(node->getCondition());
_irBuilder.CreateCondBr(condValue, whileLoopBB, whileContBB);

f->getBasicBlockList().push_back(whileLoopBB);
```

```
_irBuilder.SetInsertPoint(whileLoopBB);
this->getLLVMValue(node->getBlock());
_irBuilder.CreateBr(whileCondBB);

f->getBasicBlockList().push_back(whileContBB);
_mainContext.getIRBuilder().SetInsertPoint(whileContBB);
```

Klasa *IRBuilder* sadrži mnogobrojne metode koje mogu olakšati generisanje međukoda. Na početku metoda potrebno je odrediti funkciju u kojoj će biti smešteni naredni osnovni blokovi: blok za proveru uslova petlje (*whileCondBB*), telo petlje (*whileLoopBB*) i instrukcije nakon tela petlje (whileContBB). Zatim se generiše instrukcija bezuslovnog skoka na početak osnovnog bloka *whileCondBB*, prosleđenog kao argument metoda *CreateBr*. Prosleđeni osnovni blok se dodaje u listu osnovnih blokova odgovarajuće funkcije, a zatim se metodom *SetInsertPoint* postavlja kao polazište od kog će biti generisane instrukcije koje mu pripadaju. Nakon izračunavanja vrednosti uslovnog izraza, metod *CreateCondBr* generiše instrukciju uslovnog skoka na prosleđene osnovne blokove, u zavisnosti od izračunate vrednosti uslovnog izraza. Zatim se u listu osnovnih blokova odgovarajuće funkcije dodaje osnovni blok za telo petlje (*whileLoopBB*) i poziva metod koji generiše odgovarajuće instrukcije tela pelje. Nakon toga se generiše instrukcija bezuslovnog skoka na početak osnovnog bloka za izračunavanje vrednosti uslovnog izraza. Na kraju se u listu osnovnih blokova dodaje *whileContBB* blok u koji će biti smeštene instrukcije programa koje slede nakon tela petlje.

5.5 Pomoćna biblioteka

Pomoćna biblioteka služi za realizaciju funkcija koje omogućavaju jednostavno korišćenje perifernih komponenti mikrokontrolera. Implementirana je u programskom jeziku C i lako se može proširivati dodatnim funkcijama koje se mogu pozivati u okviru programskog jezika aKcent, na način opisan u poglavlju 4.2.1. Implementaciona rešenja ovih funkcija predstavljaju pojednostavljene verzije sličnih funkcija koje koristi Arduino platforma.

U mnogim funkcijama koriste se promenljive čije vrednosti mogu biti modifikovane u rutinama za obradu sistemskih prekida. Iščitavanje takvih promenljivih u okviru funkcija potrebno je izvršiti atomično. Rešenje je sačuvati sadržaj statusnog registra, zatim pozvati funkciju *cli* koja globalno onemogućava generisanje

sistemskih prekida (postavljanjem *I* bita statusnog registra na logičku nulu), napisati kritični deo koda, a zatim sačuvani sadržaj statusnog registra učitati u statusni registar. Analogno rešenje je upotreba *ATOMIC_BLOCK* makroa, definisanog u datoteci zaglavlja *util/atomic.h*.

Oznake odgovarajućih bitova, registara, njihove uloge i način rada perifernih komponenti mogu se pronaći u [9].

5.5.1 Funkcije za merenje vremena

Merenje vremena uključuje korišćenje periferne hardverske komponente opisane u poglavlju 2.5 — tajmersko-brojačkog modula. Funkcije opisane u nastavku teksta koriste tajmerski modul oznake 0, osmobitne rezolucije.

Postavljanjem bitova *CS00* i *CS11*, registra *TCCR0B*, određuje se vrednost deliteljske komponente. Navedena konfiguracija podešava tajmer da otkucava frekvencijom koja je 64 puta sporija od frekvencije sistemskog sata (čija frekvencija je 16 MHz). Dakle, tajmer će otkucavati frekvencijom od 250 KHz, odnosno, vrednost brojačkog registra *TCNT0* biće inkrementirana na svake 4 µs. Kako je brojački registar osmobitni, njegova maksimalna vrednost je 255. Pokušaj njegovog inkrementiranja, kada mu je trenutna vrednost 255, dovodi do prekoračenja (eng. *overflow*). Nakon toga, njegova trenutna vrednost postaje 0, a sam događaj prekoračenja može da generiše odgovarajući sistemski prekid. Kako bi prekid bio registrovan, neophodno je globalno omogućiti generisanje sistemskih prekida pozivom funkcije *sei* i postaviti bit *TOIE0*, registra *TIMSK0*.

Funkcija *time* određuje vreme izvršavanja programa izraženo u milisekundama. Uzimajući u obzir rezoluciju navedenog tajmera, prekid će biti generisan svakih 1.024 ms. Kada se prekoračenje dogodi, u rutini za obradu tog prekida vrši se inkrementiranje promenljive koja predstavlja brojač milisekundi. Prilikom svakog otkucaja tajmera, tolerancija greške od 0.024 ms nije prihvatljivo rešenje[4]. Umesto toga, grešku je moguće akumulirati sve dok njena vrednost ne dostigne 1 ms. Kada se to desi, brojač milisekundi treba dodatno inkrementirati, a brojač akumulirane greške anulirati. Dakle, nakon $\frac{1000}{24}$ broja prekoračenja, akumulirana greška će dostići vrednost od 1 ms. Navedeni razlomak se može skratiti vrednošću 8 i na taj način dobiti kompaktniji zapis — $\frac{125}{3}$. Imajući ovo u vidu, dovoljno je obezbediti uvećanje brojača akumulirane greške za 3, sve dok se ne dostigne vrednost koja

[4]Nakon samo 60 s izvršavanja programa greška bi iznosila čak 1.44 s.

je veća ili jednaka od 125. U tom slučaju brojač akumulirane greške potrebno je umanjiti za 125, a brojač milisekundi inkrementirati.

Za razliku od funkcije *time*, funkcija *utime* određuje vreme izvršavanja programa izraženo u mikrosekundama. Na osnovu ukupnog broja prekoračenja (koji su se dogodili do trenutka poziva funkcije), i trenutne vrednosti brojačkog registra (sačuvane u registru *TCNT0*), moguće je izračunati traženu vrednost. Takođe, potrebno je proveriti da li se u međuvremenu dogodilo prekoračenje brojačkog registra, koje nije moglo biti registrovano usled nemogućnosti izvršavanja sistemskih prekida. Proverom bita *TOV0*, registra *TIFR0*, moguće je ustanoviti da li je u toku izvršavanja funkcije došlo do prekida, i u zavisnosti od toga inkrementirati promenljivu koja predstavlja brojač prekoračenja. Kako je poznato da svaki otkucaj tajmera traje 4 µs, a da je za jedno prekoračenje potrebno 255 otkucaja, broj mikrosekundi moguće je odrediti narednim izrazom: $((brojPrekoracenja \cdot 255 + trenutnaVrednostBrojackogRegistra) \cdot 4$.

Funkcija *wait* predstavlja blokirajuću funkciju koja odlaže izvršavanje programa za navedeni broj milisekundi. U okviru tela funkcije koristi se prethodno opisana funkcija *utime*. Nakon perioda od 1 ms, vrednost parametra funkcije, tj. zadato vreme čekanja, biva dekrementirana. Funkcija se napušta kada vrednost parametra postane 0.

5.5.2 Funkcije za konfigurisanje pinova, upotrebu AD konvertora i PWM

U poglavlju 2.3 predstavljeni su registri čijim podešavanjem se direktno utiče na rad određenih pinova. Funkcije *input*, *output*, *low*, *high* i *toggle* (opisane u poglavlju 4.2.1) implementirane su korišćenjem jednostavnih bitovskih operacija nad opisanim registrima i njima odgovarajućim bitovima.

Merenje vrednosti napona na odgovarajućim pinovima mikrokontrolera (pinovi sa prefiksom *ADC* na slici 2.1), posredstvom AD konvertora, vrši se funkcijom *adc*. Nakon izbora odgovarajućeg kanala multipleksera na osnovu prosleđene oznake pina (konfigurisanjem registra *ADMUX*), postavljanjem bita *ADSC*, registra *ADCSRA*, započinje se merenje. Kada vrednost bita *ADSC* postane logička nula, vrednost napona je izmerena i smeštena u registru *ADC*. Funkcijama *setInternalAREF* i *setExternalAREF* vrši se izbor referentne vrednosti napona (konfigurisanjem *REFS* bitova, registra *ADMUX*), podešavanje brzine (konfigurisanjem

ADPS bitova, registra *ADCSRA*) i iniciranje rada AD konvertora (postavljanjem *ADEN* bita, registra *ADCSRA*).

Podešavanje PWM funkcionalnosti realizuje se korišćenjem tajmersko-brojačkih modula. Svakom tajmeru dodeljena su po dva pina mikrokontrolera (pinovi sa prefiksom *OC* na slici 2.1). Svaki od tih pinova potrebno je postaviti kao izlazni, odnosno, omogućiti propagaciju odgovarajuće logičke vrednosti na tom pinu. Nakon podešavanja deliteljske komponente (na sličan način kao što je opisano u prethodnom poglavlju), potrebno je podesiti režim rada generatora signala. Funkcija *init_pwm* vrši potrebne inicijalizacije odgovarajućeg tajmera. Svaki od tajmera podešen je za rad u režimu „brzog" PWM-a (eng. *fast PWM*), koji podrazumeva inkrementiranje brojačkog registra od donje (od vrednosti 0) ka gornjoj granici (ka vrednosti 255). Kada se gornja granica prekorači, sa brojanjem se iznova nastavlja počevši od donje granice. Podešavanjem *COM* bitova, registara *TCCR*, određuje se logička vrednost koja će biti propagirana na *OC* pinovima, kada se vrednost registra *OCR* poklopi sa vrednošću brojačkog registra. Funkcija *pwm* upisuje vrednost u intervalu od 0 do 255 u prethodno spomenuti *OCR* registar, dodeljen odgovarajućem pinu. Svaki od tajmera radi na sledeći način: kada se vrednost brojačkog registra i odgovarajućeg *OCR* registra poklope, logička vrednost na odgovarajućem *OC* pinu postaje logička nula sve dok tajmer ne dostigne donju granicu. Tada logička vrednost na odgovarajućem *OC* pinu postaje logička jedinica.

5.5.3 Funkcije za korišćenje UART protokola

Za inicijalizaciju ove periferne komponente zadužena je funkcija *begin* u kojoj se definiše brzina komunikacije (upisivanjem predefinisanih vrednosti u registre *UBRR0H* i *UBRR0L*), konfiguriše okvir za prenos podataka (postavljanjem *UCSZ* bitova, registra *UCSR0C*), omogućavaju sistemski prekidi za događaje poput pristizanja podataka u dolazni registar (postavljanjem *RXCIE0* bita, registra *UCSR0B*) i inicira rad komponenti za prenos i prijem podataka (postavljanjem *TXEN0* i *RXEN0* bitova, registra *UCSR0B*).

Dolazni podaci se čuvaju u kružnom baferu fiksirane širine koji poseduje pokazivače na pozicije za upis i iščitavanje, podrazumevano postavljene na vrednost 0. Količinu pristiglih bajtova, koji se nalaze u dolaznom baferu, izračunava funkcija *available*, koristeći prethodno spomenute pokazivače.

Iščitavanje bajtova smeštenih u dolaznom baferu vrši se posredstvom funkcije *receive*. Prilikom svakog poziva te funkcije proverava se vrednost pokazivača dodeljenih dolaznom baferu. Ukoliko se njihove pozicije razlikuju funkcija vraća bajt sa pozicije na koju pokazuje pokazivač za iščitavanje, a njegovu vrednost inkrementira. Prihvatanje dolaznih podataka vrši se u odgovarajućoj rutini za obradu prekida koja pristigli bajt (smešten u registru *UDR0*) smešta u dolazni bafer, tj. na poziciju na koju pokazuje pokazivač za upis, a zatim inkrementira vrednost tog pokazivača.

Slanje podataka omogućeno je funkcijom *send*. Nakon što je odlazni registar spreman za korišćenje (na šta ukazuje bit *UDRE0*, registra *UCSR0A*), u registar *UDR0* se upisuje bajt koji treba poslati, a zatim se čeka na njegovu isporuku. Bajt je isporučen kada je bit *TXC0*, registra *UCSR0A*, postavljen na logičku jedinicu.

5.6 Primer programa

Pisanje programa u programskom jeziku aKcent ilustrovano je primerom u kom je potrebno instruisati mikrokontroler, da posredstvom UART protokola, prihvata komande prosleđene sa računara, i u zavisnosti od toga podesi odgovarajući intenzitet crvenih, zelenih i plavih svetlosnih dioda (eng. *light-emitting diodes*). Za potrebe testiranja iskorišćena je Arduino Uno razvojna ploča, koja sadrži ATmega328P mikrokontroler. Operativni sistem na kom se prevođenje izvršava je Ubuntu 16.04 x64.

Komande su predstavljene odgovarajućim karakterima:

- **r** — postaviti intenzitet crvenih dioda na maksimalnu vrednost;
- **g** — postaviti intenzitet zelenih dioda na maksimalnu vrednost;
- **r** — postaviti intenzitet plavih dioda na maksimalnu vrednost;
- **l** — omogućiti treperenje dioda frekvencijom koja zavisi od vrednosti napona koja je prisutna na pinu 14 (na razvojnoj ploči Arduino Uno obično se koristi makro *A0*).

Podešavanje intenziteta svetlosti vrši se PWM tehnikom, dok se za očitavanje vrednosti napona koristi AD konvertor.

U nastavku teksta priložen je izvorni kôd programa, dok se jedno njegovo izvršavanje može videti na narednoj adresi: `http://www.alas.math.rs/~mi13086/`

`master/video.html`. U okviru ovog izvršavanja programa, traka sa svetlosnim diodama nalazi se ispod komada pleksiglasa, na kojem je ugraviran logo programskog jezika. Kada se sa računara prosledi odgovarajući karakter, mikrokontroler ga iščitava i preduzima odgovarajuću akciju, odnosno, uključuje odgovarajuće svetlosne diode. Bela boja (vidljiva na početku izvršavanja programa) dobija se istovremenim uključivanjem crvenih, zelenih i plavih dioda, dok se ljubičasta boja (vidljiva u toku treperenja) dobija uključivanjem crvenih i plavih dioda. Veličina generisanog mašinskog koda iznosi 1984 bajtova.

```
extern function i8 available();

void setColor(redValue i16, greenValue i16, blueValue i16)
{
    pwm 9, redValue;
    pwm 10, greenValue;
    pwm 11, blueValue;
}

void setMode(mode i8, blinkIndicator bool*)
{
    if (mode == 'r')
        setColor(255, 0, 0);
    else if (mode == 'g')
        setColor(0, 255, 0);
    else if (mode == 'b')
        setColor(0, 0, 255);
    else if (mode == 'l')
    {
        if (^blinkIndicator <=> true)
            ^blinkIndicator = false;
        else
            ^blinkIndicator = true;
    }
}

void blink(previousMillis i32*, blinkState bool*)
{
    currentMillis i32 = time;
```

```
        if ((currentMillis - ^previousMillis) > (adc 14))
        {
            if (^blinkState <=> true)
                setColor(150, 0, 255);
            else
                setColor(0, 0, 0);

            ^previousMillis = currentMillis;
            ^blinkState = not ^blinkState;
        }
    }

    void main()
    {
        begin 9600;
        init_pwm 9;
        init_pwm 10;
        init_pwm 11;
        setInternalAREF;

        blinkState bool = true;
        blinkIndicator bool = false;
        previousMillis i32 = 0;

        setColor(255, 255, 255);

        forever do
        {
            if (available() > 0)
                setMode(receive, &blinkIndicator);

            if (blinkIndicator <=> true)
                blink(&previousMillis, &blinkState);
        }
    }
```

Izvorni kôd funkcionalno ekvivalentnog programa, koji je prilagođen Arduino platformi (verzija 1.8.6), prikazan je u nastavku teksta. Veličina generisanog

mašinskog koda je u ovom slučaju nešto veća i iznosi 2200 bajtova. Ipak, treba uzeti u obzir implementacione razlike u korišćenim bibliotekama i drugačiji pristup prilikom pisanja programa.

```
void setColor(uint8_t redValue, uint8_t greenValue, uint8_t blueValue)
{
    analogWrite(9, redValue);
    analogWrite(10, greenValue);
    analogWrite(11, blueValue);
}

void setMode(uint8_t mode, bool *blinkIndicator)
{
    if (mode == 'r')
        setColor(255, 0, 0);
    else if (mode == 'g')
        setColor(0, 255, 0);
    else if (mode == 'b')
        setColor(0, 0, 255);
    else if (mode == 'l')
    {
        if (*blinkIndicator == true)
            *blinkIndicator = false;
        else
            *blinkIndicator = true;
    }
}

void blink(uint32_t *previousMillis, bool *blinkState)
{
    uint32_t currentMillis = millis();
    if ((currentMillis - *previousMillis) > analogRead(A0))
    {
        if (*blinkState == true)
            setColor(150, 0, 255);
        else
            setColor(0, 0, 0);
```

```
        *previousMillis = currentMillis;
        *blinkState = !(*blinkState);
    }
}

void setup()
{
    Serial.begin(9600);

    setColor(255, 255, 255);
}

bool blinkState = true;
bool blinkIndicator = false;
uint32_t previousMillis = 0;

void loop()
{
    if (Serial.available() > 0)
        setMode(Serial.read(), &blinkIndicator);

    if (blinkIndicator == true)
        blink(&previousMillis, &blinkState);
}
```

5.7 Faze prevođenja programa

Postupak prevođenja programa od izvornog koda do njegove mašinske reprezentacije, i podizanja u fleš memoriju mikrokontrolera, sastoji se od nekoliko faza. Za unapređivanje međukoda i generisanje mašinskog koda koriste se alati koje pruža zadnji deo LLVM-ove infrastrukture za arhitekturu AVR — AVR-LLVM[5].

Inicijalna faza je prevođenje programa do LLVM-ovog međukoda, posredstvom kompilatora opisanog u radu. Za program, čiji se izvorni kôd nalazi u datoteci *program.ak*, potrebno je izvršiti narednu komandu:

[5] Trenutno se nalazi u eksperimentalnoj fazi razvoja.

```
./akcent program.ak
```

Ukoliko je prevođenje uspešno izvršeno, biće generisana datoteka *program.ll* u kojoj će se nalazi LLVM-ov međukod datog programa. Generisani međukod potrebno je unaprediti korišćenjem optimizatora *opt*, opisanog u poglavlju 3.3.3. To se može uraditi izvršavanjem naredne komande:

```
opt -S -mem2reg program.ll -o program.ll
```

Optimizacioni prolaz *mem2reg* sprovodi algoritam konstruisanja SSA forme, kao i mnogobrojna unapređivanja međukoda [5]. Izvršavanjem naredne komande poziva se LLVM-ov statički prevodilac *llc* (opisan u poglavlju 3.3.3), koji će za ciljnu arhitekturu i navedeni mikrokontroler generisati objektni modul programa i smestiti ga u datoteku *program.o*.

```
llc -O3 -filetype=obj -march=avr -mcpu=atmega328p program.ll -o
    program.o
```

Objektni modul je potrebno povezati sa pomoćnom bibliotekom (opisanom u poglavlju 5.5). Za to se može iskoristiti *avr-gcc* prevodilac:

```
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p
    -I/usr/lib/avr/include -flto -Os -fdata-sections
    -ffunction-sections program.o -o program.elf libAkcent.a
```

Na ovaj način je dobijena datoteka *program.elf*, koju je potrebno prevesti u *hex* format narednom komandom:

```
avr-objcopy -O ihex -R .eeprom program.elf program.hex
```

Generisana datoteka (*program.hex*) sadrži mašinski kôd koji je potrebno smestiti u fleš memoriju mikrokontrolera posredstvom programa *avrdude*:

```
avrdude -c arduino -p atmega328p -P /dev/ttyUSB0 -b 115200 -U
    flash:w:program.hex
```

Skript datoteka koja izvršava opisane faze nalazi se u nastavku teksta. Celokupno prevođenje izvornog koda i podizanje generisanog mašinskog koda u fleš memoriju mikrokontrolera inicira se narednom komandom:

```
make PROGRAM=<naziv_programa>
```

Prilikom prevođenja izvornog koda, programski prevodilac aKcent će automatski generisati datoteku *naziv_programa.ll* i smestiti je u direktorijum u kom se nalazi skript datoteka.

```
AVR_LLVM_DIRECTORY = .../avr_llvm_folder/build/bin

PROGRAM = "program"

LLC = "$(AVR_LLVM_DIRECTORY)/llc"
LLVM_MC = "$(AVR_LLVM_DIRECTORY)/llvm-mc"
OPT = "$(AVR_LLVM_DIRECTORY)/opt"

COMPILER = avr-gcc
OBJCOPY = avr-objcopy
AVRDUDE = avrdude

CPP_FLAGS = -Os -DF_CPU=16000000UL -mmcu=atmega328p
AVRDUDE_FLAGS = -c arduino -p atmega328p -P /dev/ttyACM0 -b 115200 -U
    flash:w:$(PROGRAM).hex

INCLUDES = -I/usr/lib/avr/include
AKCENT_LIB = ../akcent_lib/libAkcent.a

build:
    ./akcent $(PROGRAM).ak
    $(OPT) -S -mem2reg $(PROGRAM).ll -o $(PROGRAM).ll
    $(LLC) -O3 -filetype=obj -march=avr -mcpu=atmega328p $(PROGRAM).ll
        -o $(PROGRAM).o
    $(COMPILER) $(CPP_FLAGS) $(INCLUDES) -flto -Os -fdata-sections
        -ffunction-sections $(PROGRAM).o -o $(PROGRAM).elf
        $(AKCENT_LIB)
    $(OBJCOPY) -O ihex -R .eeprom $(PROGRAM).elf $(PROGRAM).hex
    sudo $(AVRDUDE) $(AVRDUDE_FLAGS)

clean:
    rm $(PROGRAM).o
```

```
rm $(PROGRAM).hex
rm $(PROGRAM).elf
rm $(PROGRAM).ll
```

Glava 6

Zaključak

U radu su predstavljeni dizajn i implementacija imperativnog, statički tipiziranog programskog jezika aKcent. Razvijen je u programskom jeziku C++, sa ciljem da omogući jednostavno programiranje određenih funkcionalnosti ATmega328P mikrokontrolera. Centralno mesto u implementaciji programskog prevodioca zauzima kompajlerska infrastruktura LLVM, iskorišćena za generisanje međukoda i njegovo dalje prevođenje na mašinski kôd za AVR arhitekturu. Pored pomenute infrastrukture, u fazama leksičke i sintaksičke analize iskorišćeni su alati za obradu strukturiranog teksta (Flex i Bison) koji su umnogome olakšali implementaciju programskog prevodioca. Pomoćna biblioteka, implementirana u programskom jeziku C, omogućava korišćenje specifičnih funkcionalnosti mikrokontrolera (poput inicijalizacije i upotrebe perifernih hardverskih komponenti i konfigurisanja pinova) i pruža mehanizam za proširivanje programskog jezika.

Upoređivanje veličina generisanih mašinskih kodova za primer iz poglavlja 5.6, napisanog na programskom jeziku aKcent i uobličenog (eng. *ported*) za mikrokontrolersku platformu Arduino, pruža veoma optimističan razlog za dalje unapređivanje i razvoj programskog jezika. Buduća unapređenja mogla bi da obuhvate implementiranje višedimenzionih nizova i struktura podataka, nalik strukturama programskog jezika C. Poželjno bi bilo razmotriti podršku za još neke često korišćene mikrokontrolere osmobitne AVR arhitekture, kao što su ATiny84 (8 pinova) i ATiny85 (14 pinova). Proširivanje postojeće pomoćne biblioteke dodatnim funkcionalnostima (poput podrške za TWI (eng. *two wire interface*) i SPI (eng. *serial peripheral interface*) protokole) značajno bi doprinelo izražajnosti i upotrebljivosti programskog jezika.

Literatura

[1] Alfred V. Aho i dr. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] Keith Cooper i Linda Torczon. *Engineering: A Compiler (2nd Edition)*. Morgan Kaufmann, 2011.

[3] Free Software Foundation. *GNU Bison — The Yacc-compatible Parser Generator*. on-line at: `https://www.gnu.org/software/bison/manual/bison.html`. 2015.

[4] John R. Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, 2009.

[5] LLVM. *Kaleidoscope: Extending the Language: Mutable Variables*. on-line at: `https://llvm.org/docs/tutorial/LangImpl07.html`. 2018.

[6] LLVM. *LLVM Command Guide*. on-line at: `http://llvm.org/docs/CommandGuide/index.html`. 2018.

[7] LLVM. *LLVM Overview*. on-line at: `www.llvm.org`. 2018.

[8] Bruno Cardoso Lopes i Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[9] Microchip. *ATmega328/P - AVR Microcontroller with picoPower Technology*. 2018.

[10] Adrian Sampson. *LLVM for Grad Students*. on-line at: `https://www.cs.cornell.edu/~asampson/blog/llvm.html`. 2015.

[11] Westes. *Lexical Analysis With Flex, for Flex 2.6.2*. on-line at: `https://westes.github.io/flex/manual/`. 2018.

[12] Elliot Williams. *AVR Programming: Learning to Write Software for Hardware*. Maker Media, 2014.