Simulator PicoBlaze računala u JavaScriptu

Autor: Teo Samaržija

SAŽETAK: *Autor ovog teksta bio je frustriran nedostacima postojećih simulatora malog računala PicoBlaze, te je napravio znatno drukčiji simulator PicoBlazea u programskom jeziku JavaScript. Simulator koji je autor napravio može se pokrenuti u modernim internetskim preglednicima[1]. U tekstu slijede detalji o tome kako je autor napravio taj svoj simulator te koje su prednosti i mane tog simulatora u usporedbi s već postojećim simulatorima. Nisu korišteni nikakvi radni okviri (frameworksi), kod je pisan u VIM-u, za uređivanje slika korišteni su GIMP i Inkscape, za traženje pogrešaka u programu korišteni su alati za programiranje koji se dobiju uz Firefox i internetski servis LGTM.*

Uvod

PicoBlaze (od latinskog *piccus*, rupica na igli, u prenesenom značenju *nešto maleno*, i engleskog *blaze*, plamen) je malo računalo koji proizvodi tvrtka Xilinx. Koristi se u ugrađenim sustavima, te kao primjer jednostavnog računala na kolegiju *Arhitektura računala* na FERIT-u. Njegov je procesor dizajniran tako da se u cijelosti može implementirati programibilnim elektroničkim sklopovima (FPGA-ovima...), te je pisan u programskom jeziku VHDL. PicoBlazeov procesor jako se razlikuje od procesora korištenih u stolnim računalima (koje proizvode tvrtke Intel i AMD) i procesora u mobilnim telefonima i tabletima (uglavnom ARM-ovi procesori), i potreban nam je simulator da bismo pokrenuli programe za njega na računalu na kojem programiramo, radi testiranja ili traženja pogreške u programu. Simulator je program koji omogućuje računalu da se pretvara da je nekakvo drukčije računalo. Riječ *simulator* znači *onaj koji kopira ili imitira*, dolazi iz latinskog i prvi put se spominje u Ovidijevim Metamorfozama. Dolazi od *simulare* (pretvarati se), od *similis* (sličan). PicoBlaze vjerojatno ne može pokrenuti alate za programiranje (ili možda i može, ali bi oni bili jako spori i nepraktični za korištenje – daleko gori od alata za programiranje mobitela koji se pokreću na mobitelima), on bez korištenja pomoćnih uređaja može koristiti svega 0.25 KB memorije za spremanje podataka i 9 KB-a memorije za spremanje programa. Za simuliranje PicoBlazea najčešće se koriste FIDEX, koji proizvodi tvrtka Fautronix, ili Xilinx ISE, koji, naravno, proizvodi tvrtka Xilinx. Postoje legalne besplatne verzije tih programa koji se mogu skinuti s interneta, i te besplatne verzije podržavaju vjerojatno sve što nekome treba, tako da cijena nije problem.

Mane današnjih simulatora PicoBlazea

Mnogi bi smatrali činjenicu da su danas najčešće korišteni simulatori PicoBlazea zatvorenog koda njihovom velikom manom. PicoBlaze mekani procesor (procesor koji se može u cijelosti implementirati FPGA-ovima) jest otvorenog koda. Međutim, on se može compilirati samo Xilinxovim compilerom za VHDL, koji je zatvorenog koda. Najnapredniji compiler za VHDL koji je otvorenog koda danas je, bez sumnja, GHDL, ali njegova kompatibilnost sa Xilinxovim compilerom je slaba. Zato danas najčešće korišteni simulatori PicoBlazea, koji ciljaju na to da ga simuliraju u VHDL-ovske detalje, sadrže elemente zatvorenog koda. Dakle, to su programi za koje je ilegalno provjeravati da nisu zlonamjerni.

Po autoru ovog teksta, jedna od glavnih mana današnjih simulatora PicoBlazea je upravo to što oni ciljaju na simuliranje PicoBlazea (i druge mekane procesore) u VHDL-ovske detalje. Da bi to napravili, simulatori moraju biti programi od po stotine ili čak i tisuće MB-a. Ako pokušamo s interneta skinuti i na disk pohraniti na stotine MB-a ili GB-e podataka, gotovo sigurno ćemo naletjeti na neke neočekivane probleme (pucanje internetske veze zbog kojeg moramo krenuti ispočetka, greške u datotečnom sustavu koje ne bivaju očite dok ne pokušamo spremiti neku ogromnu datoteku...).

Glavna mana današnjih simulatora PicoBlazea je to što ih se na računalo mora instalirati da bi funkcionirali. Nije ih moguće pokrenuti u internetskom progledniku ili skinuti ZIP-arhivu i otpakirati je gdje želimo. To je osobit problem na Linuxu iz dva razloga. Prvo, instaliranje programa obično zauzima mjesta na SSD-u (gdje je spremljen Linux i sistemski direktorij gdje se obično spremaju instalirani programi, `/usr/bin`), a ne na HDD-u (na kojem obično ima daleko više slobodnog mjesta, i koji se ne troši kada na njega pišemo ili brišemo s njega). Jedan od načina da se to riješi je postavljanje virtualne mašine čiji se virtualni tvrdi disk nalazi na HDD-u, no to je dugotrajan i kompliciran posao, a i nezgodno je čekati da se pokrene još jedan Linux kad god nam treba neki program koji nam možda često treba. Drugo, programeri koji nemaju iskustva s radom na

1 `https://flatassembler.github.io/PicoBlaze/PicoBlaze.html`

Linuxu obično pretpostavljaju da postoji samo jedan, nekakav apstraktni, Linux. Simulatori PicoBlazea obično su testirani na Red Hat Linuxu, i programeri vjerojatno pretpostavljaju da, ako tamo radi, radit će na drugim verzijama Linuxa. No to vrijedi samo za najjednostavnije programe pisane u C-u ili Assembleru, čak ne ni za *Hello World* program pisan u C++-u. Istina je da će program koji radi na Red Hat Linuxu vjerojatno raditi bez problema na Oracle Linuxu, CentOS-u i Scientific Linuxu, a možda i na Fedori. No, ako želite pokrenuti program za Red Hat Linux na Ubuntu Linuxu, Debianu ili Mint Linuxu (danas najčešće korištene verzije Linuxa), puno sreće s time. Vrijedi i obratno: programi za Debian rijetko kad se mogu jednostavno pokrenuti na Oracle Linuxu. Iako postoje programi za Linux koji izvrsno rade na mnogim verzijama Linuxa (Firefox, recimo, radi savršeno na Ubuntu Linuxu, a na Oracle Linuxu samo ima problema s prikazom MP4 videa), da bi se to postiglo trebaju programeri koji poznaju Linux u najveće detalje, a takvi su rijetki. Većinom se programi za Linux na mnogim verzijama Linuxa ne daju niti instalirati. I, zapravo, instalacija je nerijetko najveći problem. Pokušaj da se na Oracle Linux instalira Chrome pomoću RPM datoteke skinute s Googlea (namijenjene za Fedoru) dovodi do hrpe poruka o pogrešci, a, ipak, izvršna datoteka Chromiuma s AppSpota funkcionira uz manje probleme. Programi otvorenog koda, kao što su VIM, mogu funkcionirati na mnogim verzijama Linuxa tako što se oslanjaju na compilere i srodne alate prisutne na Linuxu za instalaciju. No, to za napredne PicoBlaze simulatore, kojima je barem dio koda zatvoren, nije opcija.
Simulator PicoBlazea u JavaScriptu otvorenog je koda, to jest, kôd je dostupan javno na GitHubu (i, budući da je web-aplikacija, ne može učiniti ništa loše računalu ukoliko se pokrene u sigurnom internetskom pregledniku), velik je svega 160KB, i ne zahtijeva nikakve instalacije.

Struktura simulatora za PicoBlaze u JavaScriptu

Simulator PicoBlazea u JavaScriptu nema back-end (kôd koji se vrti na serveru), već se u cijelosti vrti u internetskom pregledniku. Njegov kôd podijeljen je u 7 datoteka, ukupno 3'200 redaka:

1. `PicoBlaze.html` – sadrži HTML kôd i CSS kôd te JavaScript vezan za postavljanje izgleda web-aplikacije, sintaksno bojanje asemblerskog koda, postavljanje simulatorske niti, komunikaciju između tokenizera, parsera, pretprocesora i asemblera (u svijetu compilera to se zove *driver*) te za dohvaćanje primjera asemblerskog koda s autorovoga GitHub profila. Ta datoteka ima 1'000 redaka koda. CSS koji se koristi je relativno primitivan (recimo, nema medijskih upita), za pozicioniranje elemenata na ekranu uglavnom se koristi JavaScript. Iskreno, ne da mi se učiti napredni CSS kad izgleda da mogu i bez toga.
2. `TreeNode.js` – sadrži JavaScript klasu pod nazivom `TreeNode`, koja sadrži metode vezane za evaluaciju parsiranih aritmetičkih izraza, metodu za ispis LISP-ovih izraza radi debugiranja parsera, te metode za pretragu struktura koje radi pretprocesor. Ta datoteka ima 100 redaka koda.
3. `assembler.js` – radi semantičku provjeru asemblerskog koda (recimo, je li prvi argument naredbe `load` uistinu registar) pomoću strukture koje radi parser te spaja strukturu koju radi parser i strukture koje radi pretprocesor u strojni kod u heksadekadskom obliku. Ima 1'050 redaka koda. Pretvoriti strojni kod u heksadecimalnom obliku u binarni oblik (kakav razumije PicoBlaze) nije lagano u JavaScriptu, jer najmanja jedinica memorije koja se u JavaScriptu može adresirati jest byte, 8 bitova, a svaka naredba u strojnom kodu PicoBlazea je 18 bitova, što nije cijeli broj byteova. Za razliku od ostalih potprograma, `assembler.js` i `simulator.js` svoje rezultate ne vraćaju kao povratnu vrijednost funkcije, nego ih pišu u globalne varijable.
4. `parser.js` – Parser je dio compilera (u ovom slučaju, compilera za asemblerski jezik) koji drugim dijelovima compilera kaže koja je riječ u programskom jeziku gramatički povezana s kojom drugoj riječi. To radi tako što radi strukturu zvanu AST, *abstract syntax tree*, apstraktno sintaksno stablo. Kao objašnjenje zašto je to potrebno, uzmite u obzir sljedeću rečenicu iz Cezarovog *De Bello Gallico* (koja je bila na županijskom natjecanju iz latinskog jezika 2016. godine, 6. svitak, 24. poglavlje): *Ea, quae fertilissima totius Germaniae sunt, loca Graecis aliquibus nota fama esse loquuntur.* S kojom je riječi povezana riječ *nota* (poznata)? Ako znate samo malo latinskog, vjerojatno biste pomislili da je riječ *nota* gramatički povezana s *fama* (slava, glasina), da je to pleonazam i da riječ *nota* treba zanemariti. No, to je krivo. Riječ *nota* povezana je s riječju *loca* (mjesta, lokacije). Riječ *fama* je u ablativu jednine (ablativ je latinski padež koji odgovara hrvatskom lokativu, instrumentalu te genitivu u značenju *iz koga* ili *iz čega*), a igrom slučaja na latinskom jeziku akuzativ množine u drugoj deklinaciji u srednjem rodu i ablativ jednine prve deklinacije imaju isti nastavak (to jest, isti su u pisanom latinskom ako ne označavamo naglaske, inače je *a* u nastavku u *fama* dugo, a u *nota* kratko), rečenica znači: *Kažu (loquuntur) da su (esse) ona (ea) mjesta, koja (quae) su (sunt) najplodnija (fertilissima) u cijeloj (totius) Germaniji, nekim (aliqubus) Grcima (Graecis) glasinom poznata.* Glagol

biti je jednom u prezentu (*sunt*), a jednom u infinitivu (*esse*), zbog jednog nevažnog detalja iz latinske sintakse koji se zove *akuzativ s infinitivom*. Glagol *loquuntur* je takozvani deponentni glagol, morfološki je pasivan, a semantički aktivan (zato je nastavak *-untur*, a ne *-unt*). Pridjev *totus* (cijeli) ima nepravilni genitiv jednine na *-ius* (*totius*), kao još nekolicina latinskih pridjeva. Za slučajeve kad se takve stvari dogode u programskom jeziku, parser bi spojio riječi *nota* i *loca* u jedan čvor sintaksnog stabla. Ta mi je rečenica ostala u sjećanju jer mi je profesor pričao da je ispravljao test neke učenice koja je tu rečenicu krivo prevela nešto kao *Najplodnija Njemica...*, no to bi se već teško dalo objasniti kao rezultat pogrešnog parsiranja. Ta su se najplodnija mjesta u Germaniji nalazila oko nekakve šume. Kasnije je u tom tekstu bilo *Ea (tamo = u toj šumi) nascuntur (rađaju se) alces (sjeverni jeleni)...*, a ona je to navodno prevela s *Ona rađa sjeverne jelene...*, a to opet nije stvar krivog parsiranja kad po morfologiji vidimo da je *nascuntur* pasiv i vidimo da je u množini. Parser za asemblerski jezik bio je mnogo lakši za napisati nego parser za AEC, moj programski jezik. Parser za moj programski jezik[2] dugačak je 950 redaka, dok datoteka `parser.js` sadrži 125 redaka. Zapravo, jedino što je nužno parsirati u asembleru za PicoBlaze jesu aritmetički izrazi. Algoritam napisan u datoteci `parser.js` ide ovako:

1. Pronađi parove otvorenih i zatvorenih zagrada u nizu koji ti je dao tokenizer. Parovi otvorenih i zatvorenih zagrada nalaze se, naime, u aritmetičkim izrazima te kao oznaka da je ono što se nalazi u registru pokazivač. Kada nađeš neki par zagrada, prebaci ono između zagrada u novi niz, obriši to iz originalnog niza, i pokreni rekurziju s novim nizom kao argumentom. Ako zagrade nisu dobro zatvorene, javi poruku o pogrešci. U parseru za svoj programski jezik zadao sam da se i zagrade obrišu iz sintaksnog stabla. U asemblerskom jeziku za PicoBlaze to ne bi imalo smisla, budući da zagrade imaju značenje da je u registru pokazivač, pa bih si time samo zakomplicirao asembler. Potprogram u `parser.js` zato za svaki par zagrada umeće čvor s tekstom `()`, i njegova su djeca (polje `children` iz klase `TreeNode`) niz koji vrati rekurzija.
2. Prolazi kroz niz koji ti je dao tokenizer i za svaku riječ provjeri nalazi li se na popisu mnemonika (tako se tradicionalno zovu glagoli u asemblerskom jeziku[3]) ili pretprocesorskih direktiva. Ti se popisi nalaze u datoteci `PicoBlaze.html`. Ako se riječ na koju si upravo naišao nalazi jednom od tih popisa, a nije da je riječ jednaka `enable` ili `disable` i da je dužina niza jednaka jedinici (jer `enable` i `disable`, osim što mogu biti glagoli, mogu biti i, recimo to tako, prilozi glagola `returni`), premjesti sve između tog glagola i znaka za novi red (isključivo) u novi niz, pokreni rekurziju i proglasi ono što rekurzija vrati djecom čvora u kojem je taj glagol. To funkcionira zato što svaka rečenica u asemblerskom jeziku počinje s glagolom te, osim u aritmetičkim izrazima, ne postoji lingvistička rekurzija, to jest, u asemblerskom jeziku ne postoje složene rečenice. Kao zanimljivost, neki lingvisti (ustvari, danas možda samo Daniel Everett) tvrde da je pirahanski jezik, slabo dokumentirani jezik iz Brazila, takav.
3. Parsiraj aritmetičke izraze. Prvo se baktaj s unarnim operatorima, njih se detektira kao tokeni + (plus) i − (minus) ispred kojih se ne nalazi ništa (prvi token u nizu), ili se ispred njih nalaze tokeni `,` (zarez), `(` (otvorena zagrada) ili token koji sadrži znak za novi red. Zatim konstruiraj lambda-funkciju `parseBinaryOperators` koja prima niz operatora te prolazi niz u potrazi za njima, i, kada ih nađe, njihove susjedne tokene proglašava njihovom djecom i briše iz niza.

Na kraju bismo trebali dobiti asemblerski kod u obliku LISP-ovog S-izraza, koje JavaScript u datoteci `PicoBlaze.html` za potrebe traženja grešaka u asembleru ispisuje na preglednikovu konzolu JavaScriptinom naredbom `console.log`. Profesor Ivan Aleksi predlagao mi je da ne ugradim podršku za aritmetičke izraze u svoj asembler i da ni ne parsiram asemblerski kod nego da ga pretvorim u dvodimenzionalno polje stringova, gdje svaki redak iz asemblerskog koda predstavlja jedno jednodimenzionalno polje u tom dvodimenzionalnom polju, da prvi string u tom jednodimenzionalnom polju bude potencijalni naziv labela ili prazan string, da drugi string bude glagol, i tako dalje. Ja mislim da je raditi na taj način još kompliciranije.

5. `preprocessor.js` – Prima strukturu koju pravi parser i određuje adrese labelsa. To je znatno lakše napraviti za PicoBlaze nego za Intelove i AMD-ove (x86) procesore, jer su za PicoBlaze sve naredbe u strojnom jeziku jednake dužine (18 bitova), pa možemo svaki puta kada naiđemo na mnemoniku u AST-u povećati trenutnu adresu za jedan. Za x86, taj algoritam ne bi bio točan, jer, recimo,

2 `https://raw.githubusercontent.com/FlatAssembler/AECforWebAssembly/master/parser.cpp`

3 Navodno se tako zovu jer ih je lakše zapamtiti nego nizove nula i jedinica u strojnom jeziku, od starogrčkog *μνημονικος* (*mnemonikos*) što znači *pamćenje*. Morate se naviknuti da je informatika prepuna besmislenih imena.

asemblerska naredba `int 0x3` (u doba DOS-a služila za pozivanje debuggera) ima 8 bitova (u heksadekadskom formatu je `cc`), dok `int 0x20` (u doba prvih verzija DOS-a služila za zatvaranje programa) ima 16 bitova (u heksadekadskom formatu je `20cd`), a `mov rax,[3*rbx+7]` ima 40 bitova (heksadekadski `8b485b440007`). Potprogram `preprocessor.js` također izvršava i sprema rezultate direktiva `constant` i `namereg` (za preimenovanje registara u smislene nazive) u `Map` (klasa čiji objekti sadržavaju parove ključ-vrijednost, dostupna u standardnoj biblioteci JavaScripta od vremena Internet Explorera 11). Primijetite da se pretprocesor u kontekstu asemblera jako razlikuje od pretprocesora u kontekstu compilera. U compilerima, pretprocesor se pokreće još prije tokenizera. U asemblerima, pretprocesor se pokreće nakon parsera, ali prije jezgre asemblera. U mnogim je asemblerima pretprocesor Turing-potpun (Turing-complete), što je rijedak slučaj u višim programskim jezicima (koliko znam, to još vrijedi jedino za PERL). Zapravo, to se u višim programskim jezicima smatra lošim jer programske jezike ne treba moći parsirati samo compiler za taj programski jezik, nego i drugi alati za programiranje. Jedna od čestih kritika PERL-a je upravo *Only PERL can parse PERL.*. Potprogram `preprocessor.js` ima 140 redaka.

6. `simulator.js` – Taj se potprogram pokreće u zasebnoj dretvi kad pritisnemo tipku *play* ili *fast forward*, a u istoj dretvi ako pritisnemo tipku *single step*. On čita strojni kod u heksadekadskom obliku koji je u globalni objekt `machineCode` (deklariran u `PicoBlaze.html`) upisao potprogram `assembler.js`, te simulira PicoBlaze pišući i čitajući iz memorije (globalni objekt `memory` tipa `Uint8Array` deklariran u `PicoBlaze.html`), registara (niz, zvan `registers`, od dva globalna objekta tipa `Uint8Array` deklarirana u `PicoBlaze.html`) i zastavica (gloablnih nizova `flagC`, `flagZ` te globalnih varijabli `flagIE` i `regbank`), pišući u izlaze (globalni objekt `output`), čitajući, koristeći DOM, ulaze iz one tablice s 256 HTML-ovih `input`-a, te upravljajući stogom `callStack`. Potprogram `simulator.js` ima 680 redaka. Razmišljao sam isprva o tome da jezgru simulatora napišem u svom programskom jeziku, a ne u JavaScriptu (preko svog compilera koji cilja WebAssembly, standardizirani JavaScript bytecode), no odlučio sam da to ipak ne radim tako. Naime, compiler za moj programski jezik kompatibilan je samo s najnovijim internetskim preglednicima, ne može ciljati niti Microsoft Edge, a, *rebus sic stantibus*, danas tako nešto nije opcija ako želimo da nam web-aplikacija bude popularna. Uostalom, sigurno bih naletjeo na neke probleme vezane za komunikaciju potprograma pisanih u JavaScriptu i potprograma pisanih u mom programskom jeziku, tako da je upitno bih li se manje namučio pišući taj simulator u svom programskom jeziku.
7. `tokenizer.js` – Tokenizer je dio compilera koji kaže drugim dijelovima compilera gdje završava koja riječ, a gdje počinje druga, u programskom jeziku. Riječi u programskom jeziku zovu se tokeni (engleski *token* izvan svijeta informatike znači *oznaka*). U programskim jezicima riječi ne moraju nužno biti odvojene razmacima, nego većina programskih jezika koristi malo kompleksnije mehanizme odvajanja riječi. Velika većina programskih jezika koristi mehanizam odvajanja riječi koji podsjeća na japansko pismo. Naime, japansko pismo ima tri skupa znakova: *hiragana*, *katakana* i *kanđi*. *Kanđi* su kineski znakovi i koriste se za pisanje korijena riječi. *Katakana* je slogovno pismo koje se koristi za pisanje korijena stranih riječi ili onomatopeja. *Hiragana* je slogovno pismo koji se koristi za pisanje gramatičkih afiksa (prefiksa i sufiksa). Većina gramatičkih afikasa u japanskom su sufiksi, tako da, kada dođemo do nekog hiraganskog znaka, znamo da je to nastavak prethodne riječi, a ne nova riječ. S druge strane, ako naiđemo na kineski znak ili katakanski znak nakon niza hiraganskih znakova, to je najčešće nova riječ. Sličan postupak za odvajanje riječi postoji u većini programskih jezika. Potprogram `tokenizer.js` ima 95 linija. Tokenizer za moj programski jezik[4], za usporedbu, ima 260 linija. Jedna od glavnih razlika između algoritma koji sam koristio za svoj programski jezik i algoritma koji sam koristio za PicoBlaze je ta što u PicoBlazeu nisam pokušavao zapisivati broj stupca (*column number*), jer su svi reci u asemblerskom jeziku kratki, pa podaci o stupcima gdje počinje koji token neće previše pomoći u traženju pogreške. Također, tokenizer za moj programski jezik ne smatra znak za novi red tokenom, dok tokenizer za PicoBlazeov asemblerski jezik smatra. Tokenizer za moj programski jezik mora se brinuti o escape-sequence znakovima u stringovima (`\"`...), dok tokenizer za PicoBlazeov asembler ne mora. U compileru za moj programski jezik tokenizer zamjenjuje tokene poput `'a'` odgovarajućim ASCII vrijednostima, u asembleru za PicoBlaze to radi potprogram `TreeNode.js`.

4 `https://github.com/FlatAssembler/AECforWebAssembly/raw/master/tokenizer.cpp`

Osim tih datoteka, u GitHub repozitoriju nalaze se slike napravljene u GIMP-u (Linuxov Paint) i Inkscapeu (Linuxov Publisher) te primjeri programa za PicoBlaze koje je moguće dohvatiti klikajući na *examplese*. Slike koje predstavljaju *play*, *pause*, *stop*, *fast forward* i *single step* su u SVG formatu i uređivane u Inkscapeu. Ikona za *Assembler Test* u GIF je formatu i uređivana u GIMP-u. Pozadina je fotografija PicoBlaze računala koju je profesor Ivan Aleksi uključio u svoju prezentaciju, posvijetljena u GIMP-u i spremljena kao GIF.

Primjeri programa za PicoBlaze

Web-aplikacija nudi tri primjera programa za PicoBlaze. Prvi se zove *Fibonacci Sequence*. On ispisuje Fibonaccijeve brojeve koji stanu u jedan byte (koliko su veliki PicoBlazeovi registri, svih 32), dakle, manje od 256. One koje se mogu prikazati u BCD (*binary coded decimal*, tako se u 8 bitova mogu prikazati brojevi manji od 100) formatu tako i ispisuje, a, one koji se ne mogu, ispisuje u heksadekadskom formatu. Također koristi bitovne operacije da bi izbrojao koliko ima jedinica u binarnom zapisu svakog od tih brojeva. Kad završi, javlja da je završio naredbom `return`, što ne bi prošlo na pravom PicoBlazeu. Da bi ispisao broj u novi red, on ispisuje na port sa sljedećom adresom, što na pravom PicoBlazeu isto ne bi prošlo. Ima 116 redaka, uključujući brojne komentare. Program *Gray Code* koristi bitovne operacije da bi binarne brojeve pretvarao u i iz Grayevog koda. To je način kodiranja brojeva koji se često koristi u digitalnoj elektronici, ima svojstvo da se susjedni brojevi razlikuju samo u jednoj binarnoj znamenki. Recimo, brojanje od 0 do 10 u binarnom sustavu ide (promijenjene znamenke u susjednim brojevima su boldirane, a lijeve nule, koje nije potrebno pisati, u zagradama su): (000)0, (000)**1**, (00)**10**, (00)1**1**, (0)**100**, (0)10**1**, (0)1**10**, (0)11**1**, **1000**, 100**1**, 10**10**. U Grayevom kodu ti brojevi su: (000)0, (000)**1**, (00)**1**1, (00)1**0**, (0)**1**10, (0)11**1**, (0)1**0**1, (0)10**0**, **1**100, 110**1**, 11**1**1. To svojstvo je korisno jer ne moramo računati na to da su računala koja izmjenjuju poruke potpuno sinkronizirana, da jedno računalo počne čitati broj sa žice tek kad ga je drugo računalo već dovršilo mijenjati. Taj program koristi algoritam opisan na engleskoj Wikipediji, te se vrti u beskonačnoj petlji čitajući s ulaza. Ima 25 redaka. Program *Assembler Test* je besmislen program koji koristi sve naredbe koje podržava assembler za PicoBlaze, da bude lakše testirati assembler. Ako se pokrene u simulatoru, vrti se u beskonačnoj petlji. Ima 83 reda.

Mane simuliranja PicoBlazea u JavaScriptu

Naravno, taj simulator PicoBlazea što sam ga napravio u JavaScriptu ima i svoje nedostatke naspram *fully-featured* simulatora. Prvo, on ne pokušava interpretirati VHDL, tako da ne može simulirati PicoBlazeove s modificiranim VHDL kodom (osim ako ne promijenimo JavaScript, ali opet nam to neće pomoći da nađemo greške u VHDL kodu). Drugo, grafičko sučelje mu je mnogo manje efektivno nego sučelje koje pružaju najčešće korišteni simulatori PicoBlazea. Dobro je poznato da je u web-aplikacijama teško napraviti dobro korisničko sučelje. Profesor Ivan Aleksi mi je predlagao da probam koristiti neki JavaScript radni okvir (framework) za pravljenje korisničkih sučelja, kao što je ReactJS, no za njih treba vremena da se nauče, a i pitanje je koliko uistinu pomažu. Treća je mana što je nemoguće napraviti realistični tajming. Jedna od prednosti PicoBlazea za korištenje u ugrađenim sustavima, gdje trebaju mala računala, jest upravo to što je lagano odrediti koliko će se dugo neki komad koda izvršavati ako znamo na koliko MHz-a radi (PicoBlaze može raditi na frekvenciji do oko 130 MHz), svaka instrukcija traje točno dva takta. U JavaScriptu je to nemoguće simulirati, jer JavaScript, na primjer, ima sakupljanje smeća koje se pokreće (što se JavaScriptskog programa tiče) nedeterministički. Iako moj simulator nije dostatan za neke potrebe, smatram da je dostatan za potrebe studenata i da će ih spasiti *hasslea* instalacije naprednijih simulatora.

Zahvale

Posebno zahvaljujem profesoru Ivanu Aleksiju što me je potakao da ovo napravim i što je sakupio informacije na internetu potrebne za to. Zahvaljujem i programerima koji su napravili internetski servis LGTM, statički analizer za JavaScript koji me je upozorio na neke greške koje sam napravio. JavaScript je relativno loš programski jezik i takvi su alati korisni.