Simulator PicoBlaze računala u JavaScriptu

Autor: Teo Samaržija

SAŽETAK: *Autor teksta bio je frustriran nedostacima postojećih simulatora malog računala PicoBlaze, te je napravio drukčiji simulator PicoBlazea u programskom jeziku JavaScript. Taj se simulator može pokrenuti u modernim internetskim preglednicima[1] (autor misli da je najstariji internetski preglednik u kojem simulator funkcionira kako treba Firefox 52, posljednja verzija Firefoxa koja se može vrtjeti na Windows XP-u). U tekstu slijede detalji o tome kako je autor napravio svoj simulator te koje su prednosti i mane tog simulatora u usporedbi s već postojećim simulatorima. Nisu korišteni nikakvi radni okviri (frameworksi), kôd je pisan uglavnom u VIM-u (za manje izmjene) i Eclipseu (za veće izmjene), 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. Za formatiranje koda korišteni su Prettier (za HTML i CSS) i ClangFormat (za JavaScript).*

Uvod

PicoBlaze (od latinskog *piccus*, rupica na igli, u prenesenom značenju *nešto maleno*, i engleskog *blaze*, plamen. Latinska riječ *piccus* možda je povezana s *picus*, *djetlić*, jer djetlić svojim kljunom buši drvo. Engleska riječ *blaze* dijeli isti korijen kao i latinski *flamma.* Od latinske riječi *flamma* dolazi, preko francuskog, danas daleko poznatija engleska riječ za plamen, *flame.* Hrvatska riječ *plamen* vjerojatno **nije** povezana s latinskim *flamma*.) 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, ASIC-ima...), te je pisan u programskom jeziku VHDL (VHDL je kratica od ***V**ery **H**igh Speed Integrated Circuits **H**ardware **D**escription **L**anguage*, što znači *jezik za opis strojne opreme (hardware) od integriranih krugova jako velike brzine*. Među studentima elektrotehnike česta je šala, s kojom se i ja slažem, da je VHDL zapravo kratica od ***V**ery **H**ard **D**ifficult **L**anguage – vrlo čvrsto težak jezik*). Takvi procesori, koji su namijenjeni da se sintetiziraju na FPGA-ovima ili ASIC-ima, zovu se mekani procesori (*soft processor*). PicoBlazeov procesor razlikuje se po mnogo čemu 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. Dok programiramo za male uređaje, korisno je moći pokrenuti program 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 (11. poglavlje, 634. stih). Dolazi od *simulare* (pretvarati se), od *similis* (sličan). PicoBlaze vjerojatno ne može pokrenuti alate za programiranje (ako i može, oni bi bili jako spori i nepraktični za korištenje – daleko lošiji 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 usporedbu, simulator koji je autor napravio velik je 196 KB, a jedna disketa može sadržavati, ovisi kako je formatirana, do 2'000 KB podataka (obično se formatira u FAT12 format, jer je jedini razlog zašto se diskete danas koriste kompatibilnost s prastarim računalima koja jedino to i podržavaju, a FAT12 format dopušta da se koristi oko 1'400 KB). 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.

Eclipse

Eclipse (nazvan po grčkoj riječi za pomrčinu) je besplatan IDE (*integrated developing environment*, program namijenjen da se u njemu piše kod u programskom jeziku i koji olakšava rukovanje raznim alatima za programiranje) primarno namijenjen za pisanje programa u Javi. Prva verzija izdana je 2001. godine. Međutim, danas Eclipse pruža relativno dobru podršku i za JavaScript, te je velik dio koda ovog simulatora napisan u njemu. Prednost nad VIM-om je to što koristi TypeScript Service (TypeScript je programski jezik

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

koji je dizajnirao Microsoft, on se prevodi u JavaScript i Microsoft tvrdi da olakšava automatsko traženje grešaka u programima) kako bi davao smislene sugestije za popunjavanje koda te što za naredbe iz JavaScriptine standardne biblioteke (i one relativno opskurne, koje često zaboraviš kako se koriste) daje kratku dokumentaciju. Nedostatak je što na mome računalu treba dugo da se Eclipse pokrene i što se ponekad zaglavljuje.

VIM

VIM (od *VI Improved*, poboljšani *VI*, jer je *VI*, *Visual Interface*, bio često korišteni tekstualni editor na računalima 1980-ih) je tekstualni editor koji se dobije uz gotovo sve verzije Linuxa i sličnih operativnih sustava (FreeBSD, MacOS...). Prva verzija je napravljena 1991. On je nešto kao Notepad na Windowsu, osim što pruža neke mogućnosti korisne za programiranje, recimo, sintaksno bojanje koda (za to koristi brze algoritme koji su uglavnom točni, ali nekada nisu, što pomalo živcira), nadopunjavanje djelomično napisane naredbe (drukčiji algoritam nego Eclipse, uglavnom daje lošije rezultate), brzi odlazak na određenu liniju (korisno kada compiler javi da je greška u određenoj liniji), prikaz linija koda, i tako dalje. Prednost nad Eclipseom je što se brzo otvara i što se na današnjim računalima nikad ne zaglavljuje, što je osobito važno kad želim napraviti neku izmjenu na brzinu (da vidim kako će internetski preglednik na nju reagirati). Dodatna prednost, iako to meni nije bilo važno, je što se može koristiti dok na računalu nije pokrenuto nikakvo grafičko sučelje, što je važno ukoliko radimo s udaljenog računala na serveru s kojim smo spojeni preko SSH-a (kada možemo koristiti samo tekstualno sučelje slično DOS-u). Naravno, mnoge su značajke VIM-a (uključujući i alatnu traku i izbornu traku, da ne moraš pamtiti kako razne opcije pokrenuti preko tipkovnice) tada nedostupne. Uz Ubuntu Linux dobije se smanjena verzija VIM-a koja ne podržava mnoge korisne stvari (recimo, sintaksno bojanje koda, da se različite vrste riječi u programskom jeziku oboje različitom bojom), a uz Oracle Linux dobije se zastarjela verzija (koja, recimo, krivo oboji višelinijske stringove u JavaScriptu). No, to se može riješiti tako da se s interneta skine izvorni kod novije verzije i da se compilira.

GIMP

GIMP (GNU Image Manipulation Program) je program za uređivanje rasterske grafike koji se dobije uz većinu verzija Linuxa. Na Oracle Linux mora se zasebno instalirati, no radi uz samo manje probleme (recimo, ruši se ako ga pokušamo postaviti u *Single Window Mode*). GIMP je, tako reći, Linuxov Paint. Ustvari, podržava on i mnoge korisne stvari koje Paint (barem starije verzije, dugo nisam koristio Paint pa ne znam kako novije verzije) ne podržava, kao što su slojevi (layers) i prozirnost (transparency) dijelova slike. Ipak je znatno manje moćan (ali time i lakši za korištenje) od PhotoShopa. Prva verzija GIMP-a izdana je 1996. godine.

Inkscape

Inkscape je Linuxov program za uređivanje vektorske grafike. Prva verzija izdana je 2003. godine. Dok je GIMP primarno namijenjen za uređivanje slika u rasterskim formatima kao što su PNG, BMP i JPG, gdje se za (jako pojednostavljujem) svaki piksel nalazi informacija koje je boje, Inkscape služi za uređivanje slika u vektorskim formatima kao što je SVG, gdje se slika ne opisuje pojedinim pikselima nego oblicima kao što su crte određene debljine i boje, poligoni, pravokutnici, krugovi, elipse, Bezierove krivulje, i tako dalje. Donekle sličnu funkcionalnost ima Microsoft Office Publisher (no on ne podržava SVG format, koji meni treba, a nije ni besplatan) ili Corel Draw (on podržava SVG, ali nije besplatan, a i relativno je težak za korištenje).

Firefox

Firefox je internetski preglednik koji se dobije uz Oracle Linux i jedini je internetski preglednik koji radi prihvatljivo na Oracle Linuxu. Uz Oracle Linux još se dobije i Konqueror 4.14.8 (prastara verzija iz 2008. godine, doba Internet Explorera 7, i nema očitog načina da se instalira nova), koji je praktički beskoristan za današnji internet (rijetko koja se današnja internetska stranica prikazuje ispravno u njemu), i uz njega se ne dobivaju nikakvi alati za programiranje. Danas i najmanje informatički pismeni ljudi znaju što je to Firefox, Firefox i Chrome danas su najpopularniji internetski preglednici. Uz Firefox dobivaju se alati za programiranje: debugger za JavaScript, inspektor CSS-a, alati kojima se može aproksimirati kako će web-stranica izgledati na raznim mobitelima (naravno, kako je većina mobilnih internetskih preglednika bazirana na Chromu, a ne na Firefoxu, rezultati nisu pretjerano točni), inspektori AJAX-a i tako dalje. Alati za

programiranje koji se dobiju uz internetske preglednike su primarno post-hoc, oni ne pomažu pri izradi web-aplikacije ili web-stranice, ali pomažu dijagnosticirati problem kad se dogodi. Postoje razni dodaci za IDE-jeve koji su namijenjeni za to da daju iskustvo pisanja programa u JavaScriptu slično kao pisanju programa u drugim popularnim programskim jezicima tako što povezuju Firefox ili Chrome i IDE, no njih je na Oracle Linuxu teško namjestiti.

Prettier

Prettier je formater za kôd, program koji se brine da kôd na programskom jeziku dobro izgleda što se tiče uvlaka, prelaska u nove redove, i slične stvari koje ne mijenjaju značenje koda. Jedan je od rijetkih takvih programa koji može formatirati JavaScript, HTML i CSS kad su miješani u istoj datoteci, kao što je datoteka `PicoBlaze.html`. Po meni, on ne daje baš lijepe rezultate, no takve tvrdnje već spadaju u subjektivni dio programiranja.

ClangFormat

ClangFormat je formater za kôd koji se dobije uz CLANG compiler za C, C++ i Objective-C. Može ga se koristiti s mnogim programskim jezicima, u stvari, uspio sam ga namjestiti da formatira i kôd pisan na mom programskom jeziku, AEC-u (iako ima nekih problema s mojim programskim jezikom, recimo, nema očitog načina da mu se objasni da je `:=` operator pridruživanja u mom programskom jeziku, a ne oznaka za label plus operator =, no tokenizer za moj programski jezik ignorira razmake koje on stavlja između `:` i =). Po meni, on daje lijepe rezultate za JavaScript. Mana mu je što ne može formatirati HTML i CSS niti JavaScript spremljen u HTML datoteci.

LGTM

LGTM (*Looks Good To Me*) je besplatan internetski servis koji na svojim serverima na zahtijev pokreče razne statičke analizere da analiziraju kôd pisan u raznim programskim jezicima pohranjen na GitHubu i sličnim servisima. Statički analizer je program koji pokušava naći greške u drugim programima, a da ih ne pokrene. To je različito od debuggera, programa koji pomaže dijagnosticirati grešku koja se dogodi kad se program pokrene pod njegovim nadzorom. Statički analizeri osobito su korisni za *popustljive* jezike kao što je JavaScript. U JavaScriptu, recimo, ako krivo napišemo ime varijable, compiler neće javiti grešku, nego će se program izvršavati bez upozorenja sve dok ne dođe do mjesta u programu gdje se nalazi takva pogreška (ako smo u strogom načinu rada, *strict mode*), ili čak i dalje (ako smo izvan strogo načina rada, compiler za JavaScript će na tom mjestu deklarirati novu varijablu s imenom jednakim tome krivo napisanom imenu). Isto tako, ako krivo napišemo ime neke funkcije, program će se vrtjeti sve dok ne dođe do te točke, i tek onda javiti pogrešku, tako da nam tako nešto pri brzinskom testiranju lako može promaknuti. U većini drugih programskih jezika takve vrste grešaka ne mogu promaknuti: compiler će javiti o takvim greškama prije no što se program uopće pokrene. Za JavaScript, ako to želimo, moramo koristiti statički analizer.

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. Upitno je mogu li nam ekonomski faktori i američki pravni sustav garantirati da Xilinxovi programi nisu Trojanski konji. Često se postavlja pitanje jesu li programi zatvorenog koda odraz prava na privatnost softwareskih firmi, ili su odraz nedostatka transparentnosti? Gdje je crta između privatnosti i nedostatka transparentnosti? Što ljudi pišu u privatnim pismima i privatnim elektroničkim pismima ne tiče se nikoga drugog. Ali programi koji se javno objavljuju ipak se tiču svih nas, zar ne? Isto tako, što se događa u klaonicama ili privatnim zatvorima (koji su česti u Australiji, i smatraju se iznimno nehumanima) nije samo stvar ljudi koji tamo rade, nego svih nas, zar ne? Stoga ne bi trebali postojati zakoni koji sprječavaju da se te stvari provjeravaju.
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 nekoliko GB-a podataka, gotovo sigurno ćemo naletjeti na neke neočekivane probleme

(pucanje internetske veze zbog kojeg moramo krenuti ispočetka, greške u datotečnom sustavu nastale naglim gašenjem računala 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 pregledniku 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 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. Nema jednostavnog rješenja za taj problem. Linux je otporniji na računalne viruse od Windowsa dijelom i upravo zato što ne postoji samo jedan Linux, nego mnogo verzija Linuxa koje međusobno nisu posve kompatibilne. Ako bismo napravili da programi za jednu vrstu Linuxa funkcioniraju na svim vrstama Linuxa, olakšali bismo posao nekim programerima, ali još bismo više olakšali posao kriminalcima koji rade računalne viruse. Kompatibilnost među računalima može se koristiti i za dobro i za zlo. Isto vrijedi i za biološke viruse: Ako nema genetske raznolikosti, oni se znatno lakše prenose. Neke su nekoć veoma popularne sorte banana izumrle zbog virusne infekcije, jer su sve jedinke te sorte imale više-manje iste gene za imunitet od virusa. Čim se dogodila mutacija da može jednu zaraziti, mogao je zaraziti sve istim načinom napadanja. Isto tako, šišmiši i ljudi imaju veoma sličan imunološki sustav, i zato su virusi koji mogu napasti i šišmiše i ljude relativno česti (tim više što šišimiši žive noću, kad nema sunca da ubije viruse). Psi i ljudi imaju donekle različit imunosni sustav, i zato su virusi koji mogu napasti i ljude i pse relativno rijetki. Virusi koji mogu napasti i ptice i ljude vrlo su rijetki (pod svjetlosnim mikroskopom vidi se razlika između leukocita sisavaca i leukocita ptica), a virusi koji mogu napasti i ljude i gmazove ne postoje. Nekima bi se možda učinilo da su rješenja snažni antivirusni programi, međutim, bilo koji stvaran algoritam za detektiranje računalnih virusa krivo će detektirati neke dobročudne programe kao viruse, a posljedice mogu biti jednako loše kao i posljedice samih virusa. Kao što ljudski imunosni sustav, postane li presnažan, može uzrokovati auto-imune bolesti i time biti kontraproduktivan, isto se događa s antivirusnim programima. Uostalom, rijetko se koji novi virus može detektirati zastarjelim algoritmima ugrađenima u antivirusne programe, kriminalci koji rade računalne viruse znaju kako ti algoritmi funkcioniraju i kako ih prevariti.
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 196KB, 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 3'550 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. Naknadno je dodana

mogućnost dodavanja točaka prekida (*breakpoints*), mogućnost skidanja heksadekadske datoteke koju proizvodi assembler (koja se uz pomoć alata za programiranje PicoBlazea veoma lako pretvori u binarni format koji treba PicoBlazeu) te zoran prikaz sedam-segmentnih pokaznika (*seven-segment display*) i prekidača (*switches*, ovdje reagiraju na pritisak miša) pomoću SVG-a generiranog u JavaScriptu. Skidanje heksadekadske datoteke radi se preko standardnih JavaScript klasa `HTMLAnchorElement` (svojstvo `download` i metoda za stvaranje eventa iz JavaScripta `click`), `Uint8Array`, `URL` i `Blob`, bilo je relativno komplicirano za napraviti. Točke prekida rade se tako da se brojevi linija koda nalaze u zasebnim HTML elementima `<div>` koje generira JavaScript i ti elementi osim brojeva linija koda sadržavaju i ikone koje predstavljaju točke prekida koje su po defaultu nevidljive. Globalni objekt `machineCode`, osim heksadekadskih stringova koje predstavljaju naredbe u strojnom kodu, čuva i podatke o linijama koda iz kojih dolaze naredbe. Datoteka `PicoBlaze.html` ima 1'320 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. U CSS-u se koriste varijable, no dodan je i *fallback* za internetske preglednike koji ne podržavaju CSS-ove varijable (njih ne podržava, koliko je meni poznato, niti jedan internetski preglednik koji se može vrtjeti na Windows XP, zastarjelom, ali na zastarjelim računalima u obrazovnim ustanovama još uvijek popularnom operativnom sustavu). Velik dio HTML-a (sve tablice) također se generira u JavaScriptu, uglavnom pomoću JavaScriptinih višelinijskih stringova i JavaScriptine naredbe `innerHTML`. Razni internetski preglednici rade probleme ako se tako pokušaju generirati SVG elementi (u JavaScriptu se generiraju SVG elementi koji predstavljaju sedam-segmentne pokaznike, prekidače i LED-ice), pa sam za svaki slučaj to radio JavaScriptinim naredbama `createElementNS`, `setAttribute` i `appendChild`. U datoteci `PicoBlaze.html` još se nalazi i kod za simulaciju UART-a, protokol pomoću kojega PicoBlaze može komunicirati s terminalima i simulatorima terminala, ali on je po defaultu isključen. Terminali su, naime, uređaji s tipkovnicom i ekranom pomoću kojih možemo komunicirati s programima na udaljenim računalima koji imaju CLI sučelje, to jest, sučelje slično tipičnim DOS-ovim programima.

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. Također radi neke sintaksne provjere koje parser ne radi, recimo, nalazi li se između dva argumenta naredbe `load` zarez. 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. Potprogram `assembler.js` piše svoje rezultate u globalni objekt `machineCode`, odakle ih potprogram `simulator.js` čita.
4. `parser.js` – Parser (od latinskog *pars*, *dio*) 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 originalni niz u potrazi za njima, i, kada ih nađe, njihove susjedne tokene proglašava njihovom djecom i briše iz niza. Ta se lambda-funkcija prvo poziva za niz samo s operatorom potenciranja, `^`, jer on ima najveći prioritet, zatim se poziva za operatore množenja i dijeljenja, `*` i `/`, te konačno za zbrajanje i oduzimanje, `+` i `−`. Parser za moj programski jezik mora paziti na razliku između

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. Uzmite u obzir i da je ime *digitalna elektronika* poprilično besmisleno, ono se prevodi kao *prst-jantar*. Nekome bi u antici, tko zna latinski i grčki, to ime vjerojatno bilo smiješno. Imam jednu anegdotu iz svog studentskog života o tome: Došla je knjižničarka na početku odmora u predavaonicu informirati profesora Martinovića da će biti nekakav simpozij gdje bi trebali doći profesori koji se bave humanističkim predmetima, a da misli da se to tiče profesora Martinovića jer on predaje kolegij zvan *Dizajn programske podrške*. A profesor Martinović joj odgovori: *Ah, kolegice, pa nije Vam to nikakav dizajn, to se samo tako zove.* Knjižničarka ga je, naravno, onda začuđeno pogledala.

lijevo-asocijativnih operatora i desno-asocijativnih operatora. No, budući da su svi aritmetički operatori lijevo-asocijativni, to ovdje nije potrebno.

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 (labelsi su oznake na koje je moguće skočiti naredbom `goto`, ili, kako se u assemblerskom jeziku zove ta naredba, `jump`). 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, 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`, te globalna varijabla `PC`, *program counter*, koja pokazuje na naredbu u memoriji koja se trenutno izvršava) 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`. Naknadno je dodano da za svaku naredbu provjerava je li na njoj breakpoint. Potprogram `simulator.js` ima 690 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 *kandži*. *Kandži* 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 (što možda i nije bila dobra ideja, jer poruke o pogreškama tipa *unexpected token* zbog toga mogu biti nejasne), u asembleru za PicoBlaze to radi potprogram `TreeNode.js`.

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. Ikone za primjer *Hexadecimal Counter* te ikona koja predstavlja točku prekida (*breakpoint*) isto su uređivane u GIMP-u, ali spremljene su u PNG formatu, jer ih PNG format bolje sažima nego GIF format. 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 šest primjera programa za PicoBlaze. Oni se nalaze, kao i njihov popis u JSON formatu, na autorovom GitHub profilu, te ih potprogram `PicoBlaze.html` dohvaća koristeći JavaScriptinu naredbu `fetch`. Prvi se primjer 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. To je osobito bilo važno za računala iz 1940-ih, na kojima implementiranje protokola sinkronizacije (Manchestersko kodiranje...) nije bilo jednostavno ili čak ni moguće. Program *Gray Code* 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. Naknadno su dodani primjeri *Binary to Decimal* i *Hexadecimal Counter*. *Binary to Decimal* postavljen je kao prvi primjer s lijeva. On čita 8-bitni binarni broj unesen pomoću onih SVG-ovskih prekidača, pretvara ga u decimalni broj i rezultat prikazuje na sedam-segmentnim pokaznicima, te se vrti u beskonačnoj petlji. Pretvaranje binarnog broja u decimalni radi se algoritmom za pretvaranje binarnog broja u BCD, uz izmjenu da pazi nalazi se broj između 0 i 99, između 100 i 199 ili je više od 200 (8-bitni binarni brojevi mogu biti najviše 255). Uz to on zapošljava i LED-ice tako što na njima prikazuje broj pretvoren u Grayev kod. *Hexadecimal Counter* besmislen je program koji pali i gasi LED-ice te prikazuje

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

heksadecimalne brojeve na 7-segmentnim pokaznicima. Nakon svih tih primjera dodan je i primjer *Decimal to Binary*, koji pretvara dekadske brojeve u binarne, a dekadski brojevi se u njega upisuju koristeći UART, a isto tako se iz UART-a čitaju binarni brojevi koji su rezultati.

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. Uz to, na danas prosječnom računalu taj moj simulator može izvršiti oko 20 instrukcija po sekundi (u fast-forwardu u Firefoxu, pregledniku koji najbrže izvršava JavaScript), daleko manje od 75'000'000 operacija u sekundi koji bi bili potrebni za realistični tajming. Jedan čovjek na internetskom forumu tvrdi da ga vjerojatno najviše usporava to što se simulacijska dretva prekine kad se izvrši jedna instrukcija, a ponovno postavlja kad se treba izvršiti nova, a na postavljanje i uništavanje JavaScriptine dretve troše se mnogi računalni resursi[5]. Dakle, da bismo postigli realističan tajming, morali bismo posve preurediti simulator, a vjerojatno i napisati ga u drugom programskom jeziku (a puno sreće da ga onda pokrenete u internetskom pregledniku). Pokušao sam ga ubrzati tako što sam manipulacije stringovima zamijenio bitovnim operacijama gdje je to jednostavno za napraviti, ali to nije urodilo plodom. Također, simulacija UART-a poprilično je nerealistična, a nije očito kako bismo u programskom jeziku JavaScript napravili dobar simulator terminala. 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 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.

5 https://www.reddit.com/r/asm/comments/jyfrxy/how_to_implement_breakpoints_in_a_simulator/gd5ysu7/?utm_source=reddit&utm_medium=web2x&context=3