Univerza v Ljubljani

Fakulteta za elektrotehniko

Kristjan Saksida

LOGIČNI ANALIZATOR ZA VODILO CAN S SPLETNIM VMESNIKOM

Magistrsko delo

Mentor: izr. prof. dr. Andrej Trost

Ljubljana, 2015

Zahvala

Zahvaljujem se mentorju izr. prof. dr. Andreju Trostu za neizmerno pomoč in strokovne nasvete pri ustvarjanju magistrskega dela ter delu na drugih študijskih projektih.

Prav tako se za vso podporo, vzpodbude in odrekanja zahvaljujem ožji družini.

Vsebina

Seznam slik

Seznam tabel

Seznam uporabljenih simbolov

ABS Sistem proti blokiranju koles med zaviranjem (ang. Anti-lock Braking System)

APSOC Programirljiv SOC (ang. All-Programmable SOC)

APU Namenska procesna enota (ang. Application Processing Unit)

ASIC Namensko integrirano vezje (ang. Application Specific Integrated Circuit)

AXI Napreden razširljiv vmesnik (ang. Advanced eXtensible Interface)

BRAM Blokovni RAM

BSP Skupek programske opreme za podporo neki strojni opremi (ang. Board Support Package)

CAN Podatkovno vodilo (ang. Controller Area Network)

DHCP Omrežni protokol za dinamično dodelitev IP naslovov (ang. Dynamic Host Configuration Protocol)

DSP Digitalni signalni procesor

EMC Elektromagnetna kompatibilnost (ang. Electromagnetic Compatibility)

EMIO Razširjen MIO (ang. Extended MIO)

EOBD Standard za avto diagnostiko (ang. European On Board Diagnosis)

FIFO Vrsta pomnilnika (ang. First In First Out)

FPGA Programirljiva logična vrata (ang. Field Programmable Gate Array)

GND Ničelni potencial

GPL Licenca za prosto programje (ang. General Public License)

HTML Označevalni jezik za izdelavo spletnih strani (ang. Hyper Text Markup Language)

HTTP Komunikacijski protokol (ang. Hyper Text Transfer Protocol)

I/O Vhod/Izhod (ang. Input/Output)

IP Internetni protokol

ISO Mednarodna organizacija za standardizacijo (ang. International Organization for Standardization)

LIN Lokalno povezano omrežje (ang. Local Interconnect Network)

LWIP Sklad za TCP/IP protokol (ang. Light Weight IP)

MFS Datotečni sistem (ang. Memory File System)

MIL Lučka napake motornega sklopa (ang. Malfunction Indicator Lamp)

MIO Multipleksiran vhod/izhod (ang. Multiplexed Input/Output)

NZR Način kodiranja binarnih signalov (ang. Non-return To Zero)

OSI Konceptualni model omrežja (ang. Open Systems Interconnection)

PHP Programski jezik za spletne vsebine (ang. Hypertext Preprocessor)

PL Programirljiva logika

PS Procesorski sistem

R Upornost

RAM Bralno-pisalni pomnilnik (ang. Random Access Memory)

ROM Bralni pomnilnik (ang. Read Only Memory)

RTOS Operacijski sistem v realnem času (ang. Real-Time Operating System)

SDK Programsko razvojno okolje (ang. Software Development Kit)

SOC Sistem na integriranem vezju (ang. System On a Chip)

SRL Logični pomik v desno (ang. Shift Right Logic)

TCP Protokol za komunikacijo po omrežju (ang. Transmission Control Protocol)

UDP Protokol za komunikacijo po omrežju (ang. User Datagram Protocol)

USB Univerzalno serijsko vodilo (ang. Universal Serial Bus)

VHDL Jezik za opis izredno hitrih digitalnih vezij (ang. Very high-speed integrated circuit Hardware Description Language)

XAML Označevalni jezik za izdelavo Microsoftovih aplikacij (ang. Extensible Application Markup Language)

µP Mikroprocesor

Povzetek

Na trgu lahko najdemo več različic logičnih analizatorjev, ki pa večinoma nimajo možnosti direktnega priklopa na omrežje ter ne omogočajo neposrednega internetnega dostopa do vzorčenih podatkov. Potreba po takšnem dostopu do podatkov se pojavi npr. v avtomobilski industriji, kjer se izvajajo vzorčenja na testnih stezah. Magistrsko delo opisuje postavitev koncepta logičnega analizatorja s spletnim vmesnikom, ki je prilagojen za CAN vodilo.

Koncept temelji na sistemu na čipu Xilinx Zynq, kjer je vzorčevalni del narejen v programirljivi logiki FPGA. Vzorci se shranjujejo v pomnilnik BRAM, od koder so s TCP-protokolom poslani do uporabnika. To funkcijo opravlja operacijski sistem v realnem času FreeRTOS, ki hkrati poganja spletni strežnik. Spletni vmesnik sloni na Microsoftovem vtičniku Silverlight, ki podpira TCP-protokol.

S tako zgrajenim logičnim analizatorjem omogočimo uporabniku prijazno in enostavno upravljanje na daljavo iz samega brskalnika in to brez dodatne programske opreme.

Ključne besede: CAN-vodilo, logični analizator, sistem na čipu, FPGA, Xilinx Zynq, FreeRTOS, Microsoft Silverlight, spletni vmesnik, TCP-protokol

Abstract

Several versions of logic analyzers can be found on the market but mostly they do not have the possibility of direct connection to the network and do not provide direct internet access to the sampled data. The need for such access to data appears for example in the automotive industry where samplings are carried out on test tracks. The master's thesis describes a concept of a logical analyzer with a web interface which is adapted to the CAN bus.

The concept is based on a system on Xilinx Zynq chip where the sampling part is made in the FPGA programmable logic. Samples are stored in the BRAM memory from where they are sent to the user via TCP protocol. This function is performed in real time by the FreeRTOS operating system which simultaneously drives the web server. Web interface is based on the Microsoft Silverlight plug-in which supports TCP protocol.

With such construction of the logic analyzer we provide user friendly and easy remote management from a browser without any additional software.

Keywords: CAN bus, logic analyzer, system-on-chip, FPGA, Xilinx Zynq, FreeRTOS, Microsoft Silverlight, web interface, TCP protocol

1 Uvod

Pri razvoju oziroma analizi digitalnih vezij pogosto uporabljamo logični analizator kot pripomoček za razhroščevanje in vrednotenje celotnega sistema ali podsklopov le-tega. Logični ali tudi digitalni analizator je naprava, ki je v preteklosti zrastla iz osciloskopa, predvsem zaradi potreb, ki jih ta ni mogel zadostiti. V mislih moramo imeti predvsem število kanalov osciloskopa in več-bitnost vodil ter registrov, ki bi jih želeli opazovati. Tu nam je odveč visoka resolucija napetostnega nivoja, pojavi pa se nezmožnost hkratnega prikaza daljših večbitnih širin. Kljub temu sta v svoji osnovi napravi zelo podobni, zato najdemo na trgu logične analizatorje kot drage dodatke osciloskopom višjega cenovnega razreda. Na drugi strani pa s pojavom vse cenejših, a dovolj zmogljivih mikroprocesorjev, danes trg ponuja za majhno ceno minimalistične analizatorje z zadovoljivim številom vhodov in frekvenco vzorčenja. Če naredimo površno delitev logičnih analizatorjev, ki so na voljo, dobimo v grobem tri podzvrsti:

- Modularni, sestavljeni so iz glavne enote, lahko je to tudi osciloskop, ki skrbi za prikaz in upravljanje analize ter omogoča priklop vzorčevalne strojne opreme. Tu je praviloma na voljo možnost hkratnega priklopa več različnih vzorčevalnih modulov. S tem lahko zajamemo vodila ali registre velikih širin oziroma veliko število le-teh. To so naprave višjega cenovnega razreda [1].
- Prenosni so po navadi v srednjem cenovnem razredu, vse komponente so v eni napravi. Večinoma so to splošno namenski. Če bi želeli vzorčiti vodilo drugih specifikacij, ti v primerjavi z modularnimi ne podpirajo oziroma ne omogočajo zamenjave vzorčevalnega modula [2].
- Minimalistični, opisuje jih najnižja zmogljivost, vzorčevalna elektronika prek USB (ang. Universal Serial Bus) ali Ethernet

povezave pošilja podatke na osebni računalnik, kjer so prikazani v namenski programski opremi [3].

Če izvzamemo oddaljen dostop do osebnega računalnika ali osciloskopa, nobena od podzvrsti ne omogoča direktnega internetnega dostopa do podatkov, kaj šele neposrednega oddaljenega upravljanja z napravo. Tu se pojavi želja po umestitvi digitalnega analizatorja kot samostojne naprave v omrežje, istočasno pa potreba po spletnem vmesniku, ki bi uporabniku omogočal interakcijo z napravo brez dodatne namenske programske opreme preko tabličnega, osebnega računalnika ali pametnega telefona. To daje analizatorju širok spekter uporabe na terenu, v težkih in nevarnih pogojih za človeka, šolstvu ter industriji.

Pri spremljanju avtomobilske industrije z vidika razvoja različnih elektronskih komponent, opazimo trend čim hitrejših in raznolikih testiranj le-teh. Tu so mišljena testiranja v samem laboratoriju ali na testni stezi oziroma terenu. V avtomobilih in večini ostalih vozil današnjega časa so posamezne naprave (motorni računalnik, ABS (ang. Anti-lock Braking System) modul, klimatska naprava, zabavna elektronika itd.) povezani med seboj preko skupnega vodila ali v kombinaciji z več vodili. Najbolj pogosto se tu srečamo s CAN (ang. Controller Area Network) vodilom, ki je poleg samostojne funkcije velikokrat tudi hrbtenica cenejšemu LIN (ang. Local Interconnect Network) vodilu. Predvsem v avtomobilih višjega cenovnega razreda, kjer se želi kupcu ponuditi udobje, ki zahteva višje prenosne hitrosti (Drive-by-wire, aktivno vzmetenje, adaptivni tempomat itd.), ti dve vodili nadomešča FlexRay. Trend razvoja kaže na to, da bo zaradi cene in evropske direktive o enakem standardu za osnovno avto diagnostiko EOBD (ang. European On Board Diagnosis) pri vseh avtomobilskih proizvajalcih CAN ostal prisoten kot temeljni signalni protokol še kar nekaj časa. Podobne direktive imajo tudi ZDA in nekatere druge države [4, 5, 6].

Če želimo vzorčiti CAN-vodilo z logičnim analizatorjem med samo vožnjo, brez da bi bili prisotni v avtomobilu, moramo uporabiti kombinacijo vzorčevalnih modulov in naprave, ki ima oddaljen dostop. Druga možnost je uporaba analizatorja, ki ima z namensko programsko opremo brezžično Ethernet povezavo. Zavedati se moramo, da smo v obeh primerih omejeni s pasovno širino povezave, najbolj seveda tam, kjer je slabo omrežje. To pomeni, da potrebujemo napravo, ki je optimizirana za vzorčenje želenega vodila in tako zaseda dovolj majhno pasovno širino. Praviloma so to dražji primerki analizatorjev.

Vsa našteta dejstva nas navdihujejo za razvoj novega koncepta logičnega analizatorja, ki bi bil kot samostojna naprava povezan v omrežje. Želimo si zadovoljive frekvence vzorčenja, časovno označbo vzorčenih podatkov in spletni vmesnik z osnovnimi funkcijami. Postaviti ga moramo na taki platformi, ki bo omogočala dovolj zmogljivosti in prilagodljivosti za analiziranje različnih elektronskih sistemov in vodil. Z načrtovanjem logičnega analizatorja, prilagojenega za CAN-vodilo, ki ima zgoraj naštete lastnosti, se srečamo v naslednjih poglavjih.

2 Analizator vodila CAN

2.1 CAN-vodilo

CAN-vodilo je bilo razvito in prvič predstavljeno leta 1983 s strani podjetja Robert Bosch GmbH. V omrežju z več nadrejenimi enotami predvideva hitrosti prenosa od 125 kbit/s do 1 Mbit/s. Z razliko od vodil in omrežij, kot so USB ali Ethernet, CAN ne zahteva centralnega arbitra za usmerjanje podatkovnega prometa. Tudi sama sporočila so krajša, po navadi gre za pošiljanje temperature, hitrosti, obratov itd. Podatki so razposlani vsakemu elementu omrežja (ang. broadcast), kar zagotavlja podatkovno konsistenco za vsak element v omrežju [7].

Obravnavano vodilo je od ISO (ang. International Organization for Standardization) definirana serijska komunikacija, ki je bila v prvi vrsti razvita za avtomobilsko industrijo, da bi zamenjala kompleksne povezave z dvo-žičnim vodilom. To je tudi pomenilo zasnovo odporno na veliko električno interferenco in zmožnost zaznave ter samo odprave napak pri prenosu. Te lastnosti so pripeljale do širše uporabe vodila tudi na drugih področjih avtomatizacije, medicine in industrije. Komunikacijski protokol ISO 11898-X: 2003 sloni na slojih Open Systems Interconnection (OSI) modela in opisuje, kako poteka prenos podatkov med napravami v omrežju [8, 9, 10, 11, 12]:

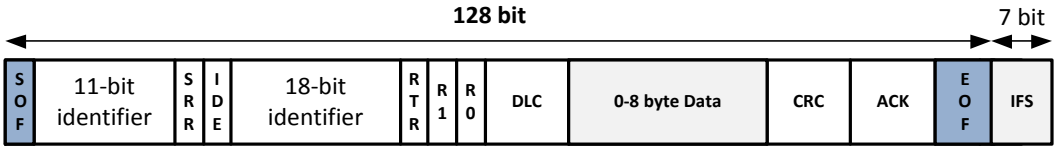
- ISO 11898-1:2003 – *podatkovni sloj in fizična signalizacija*
- ISO 11898-2:2003 – *fizični sloj (visoka hitrost)*
- ISO 11898-3:2006 – *fizični sloj (nizka hitrost, odpornost na napake)*
- ISO 11898-4:2004 – *časovno prožen CAN*
- ISO 11898-5:2007 – *nizko energijski način*

Dejanska komunikacija med napravami je definirana v fizičnem sloju modela. Sam standard pa definira najnižja dva sloja od sedmih, kolikor jih vsebuje ISO/OSI

model. CAN-vodilo je lahko konfigurirano za delovanje z dvema različnima tipoma sporočilnih okvirjev. Prvega najdemo pod opisom CAN 2.0A in vsebuje 11-bitni identifikator. Drugi je pod oznako CAN 2.0B in ima dodatni 18-bitni identifikatorski dodatek (skupaj 29 bitov). Drugih razlik med njima ni. CAN ima 4 tipe okvirjev:

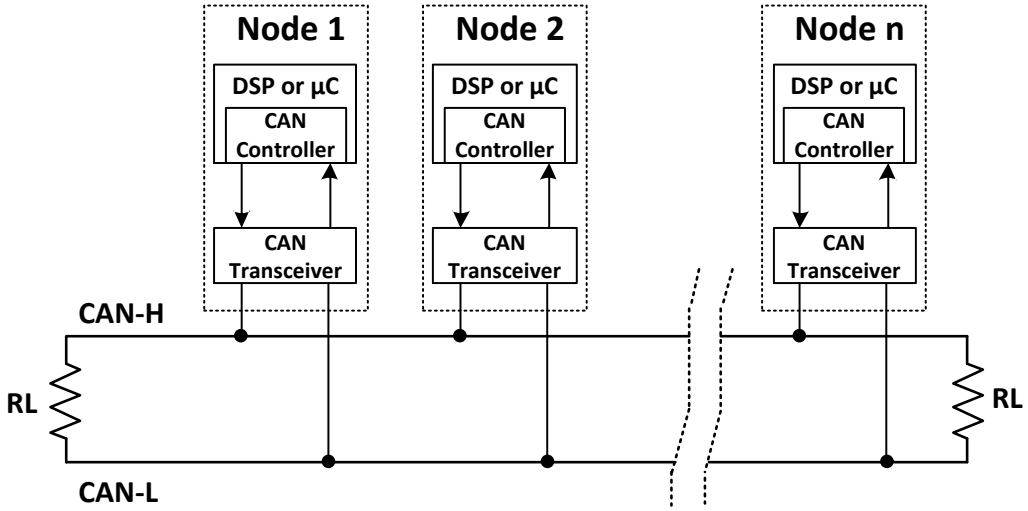
- podatkovni okvir – vsebuje podatke za prenos,
- oddaljeni okvir – vsebuje zahtevo za prenos določenega podatka,
- okvir napake – vsebuje sporočilo o napaki,
- okvir preobremenitve – zapolnjuje čas med podatkovnim in oddaljenim okvirjem.

Za nas je pomemben podatkovni okvir z daljšim 29-bitnim identifikatorjem (CAN 2.0B), saj bi radi imeli predstavo, kolikšno je število bitov, ki jih bomo morali vzorčiti z našim analizatorjem. Okvir je prikazan na sliki 2.1. Identifikator služi kot določitelj prioritete, če bi dve ali več naprav hkrati začele uporabiti vodilo. Podrobnejša analiza okvirja in samega protokola med napravami za naše potrebe ni potrebna. Pričakujemo maksimalno 128 bitov dolg okvir, z največjo hitrostjo 1 Mbit/s. Na tem mestu moramo upoštevati možnost t. i. vrivanja bitov (ang. bit stuffing). Ker je uporabljeno NZR (ang. Non-return To Zero) kodiranje, mora protokol zagotavljati sinhronizacijo z vrivanjem bita, ki je različen od šestih enakih predhodnikov. To pomeni ustrezno daljši okvir oziroma več bitov za vzorčenje [7, stran 37].



Slika 2.1: CAN 2.0B okvir brez vrivanja bitov [13]

Fizični nivo, opisan v ISO 11898-2:2003, predvideva uporabo dveh paralelnih vodnikov z nominalno impedanco R_L=120 Ω, za doseganje EMC (ang. Electromagnetic Compatibility) pa prepleten oklopljen par, čeprav sam standard ne zahteva uporabe oklopa. Dolžina je za doseganje 1 Mbit/s omejena na 40 m, s kompromisom nižje hitrosti pa lahko uporabimo daljše ožičenje. Na sliki 2.2 je prikazana groba shema vodila s priključenimi napravami oziroma celoten sistem.



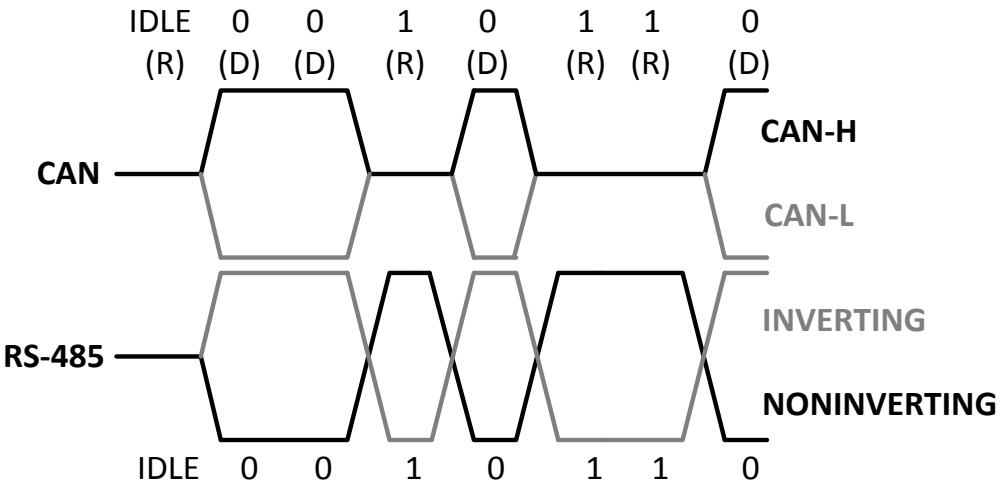
Slika 2.2: CAN-vodilo z napravami [13]

Vidimo, da vsaka priključena naprava vsebuje DSP (digitalni signalni procesor) ali µP (mikroprocesor), ki že ima vgrajen CAN-kontroler ali pa je nanj povezan. Kontroler je povezan na oddajno-sprejemno enoto, ki je fizično priklopljena na vodilo oziroma diferencialni par CAN-H (High) in CAN-L (Low). Pri večini najbolj znanih diferencialnih prenosih podatkov (npr. RS-485) je logična ena prenesena kot visok napetostni nivo na neinvertirajočem vodniku in logična ničla kot nizek nivo na invertirajočem vodniku. Sprejemnik zazna razliko napetosti med obema vodnikoma in določi logično '1' ali '0'. Ko je sprejemno-oddajna enota na vodilu v stanju visoke impedance pri vseh napravah priklopljenih na vodilo, pomeni, da je vodilo v mirujočem stanju. Standard za CAN določa samo dve stanji, recesivno (visoka impedanca) in dominantno (vodnik CAN-H je na visokem nivoju, CAN-L pa na nizkem), pragovne napetosti so prikazane v tabeli 2.1 [13]:

Logika	RS-485 nivoji	CAN stanje	CAN nivoji
1	A – B ≥ +200 mv	Recesivno	CAN-H – CAN-L ≤ 0.5 V
0	A – B ≤ –200 mv	Dominantno	CAN-H – CAN-L ≥ 0.9 V

Tabela 2.1: Primerjava med CAN in RS-485 napetostnimi nivoji [13]

Pomembno je dejstvo, da je logična '1' prenesena z recesivnim stanjem, logična '0' pa z dominantnim. Opravka imamo torej z invertirano logiko. Za boljše razumevanje si na sliki 2.3 oglejmo primerjavo z RS-485.



Slika 2.3: Grafična primerjava med CAN in RS-485 napetostnimi nivoji [13]

Za razvoj analizatorja potrebujemo fizični dostop do CAN-vodila. Ena izmed možnosti je uporaba vozila, vendar je tak način razvoja za naš in večino drugih primerov nekoliko nepraktičen. Odločimo se za emulator ECUsim^{TM} 2000 podjetja OBD Solutions.

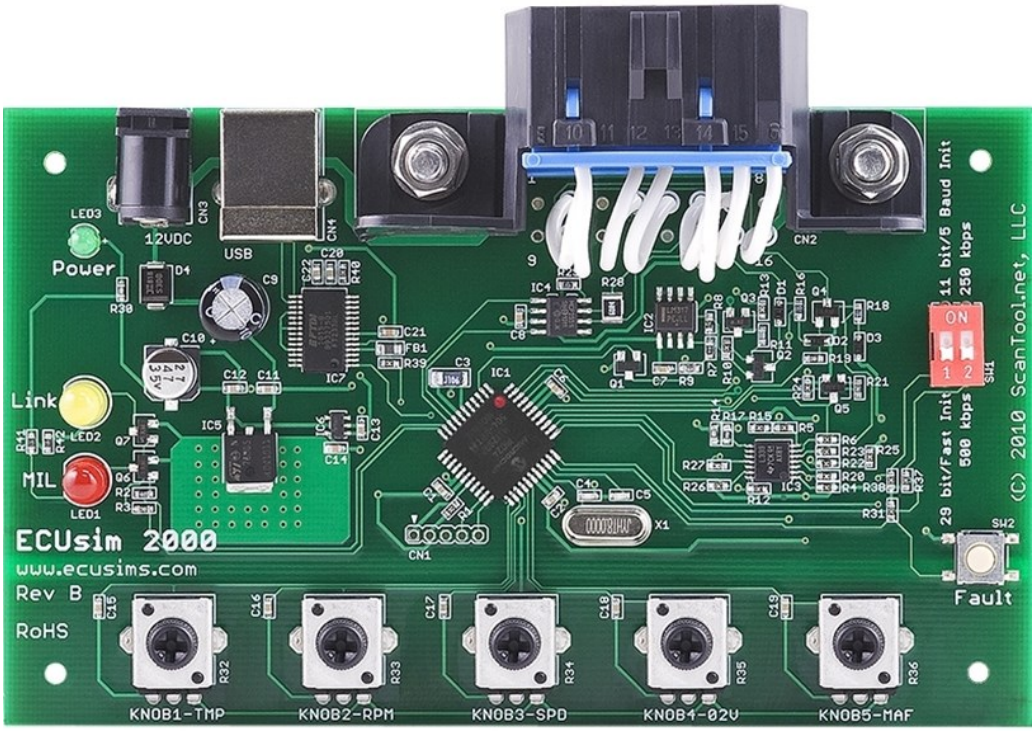
2.2 OBD-II ECU emulator

ECUsim^{TM} 2000 (slika 2.4) je minimalističen večprotokolni namizni emulator, ki ga lahko uporabljamo za testiranje in razvoj OBD-II aplikacij [14]. Podpira glavnino signalnih protokolov za OBD-II in EOBD-II ter drugih standardov za avto diagnostiko [15]:

- SAE J1850 PWM
- SAE J1850 VPW
- ISO 9141-2
- ISO 14230-4 (KWP 2000)
- **ISO 15765-4 (CAN 250/500kb/s, 11/29 bit)**

Za nas je pomemben zadnji izmed standardov, ki določa zahteve za CAN-vodilo, na katerem se ena ali več priključenih naprav uporablja za OBD [16]. Žal je v tem primeru hitrost omejena na 500 kbit/s oziroma na polovico največje, imamo pa možnost emulirati okvir z 29-bitnim identifikatorjem. Oba parametra nastavimo na preklopnem stikalu. Dotična verzija emulatorja emulira motorni računalnik in nam daje možnost, da s petimi potenciometri spreminjamo vrednosti hladilne tekočine,

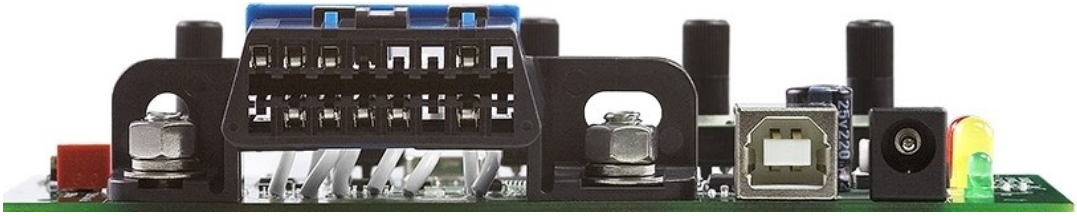
obratov motorja, hitrost vozila, napetost kisikovega senzorja in vrednosti senzorja za pretok zraka. Napake MIL (ang. Malfunction Indicator Lamp) sprožimo ročno s pritiskom na gumb.



Slika 2.4: Namizni emulator ECUsim 2000 [14]

Če si ogledamo razporeditev pinov (slika 2.5) za konektor SAE J1962, ki se pritičejo razvoja analizatorja, vidimo, da standard predvideva:

- **CAN-High** **pin 6**
- **CAN-Low** **pin 14**
- Masa podvozja pin 4
- Signalna masa pin 5
- Baterijska napetost pin 16



Slika 2.5: OBD-II priključek emulatorja [14]

Emulator kot samostojna enota ni dovolj za generiranje prometa po CAN-vodilu, saj sam od sebe ne pošilja podatkov. Iz že opisanega standarda, ki govori o okvirjih, vidimo, da potrebujemo napravo, ki zmore poslati zahtevo za določen podatek. V ta namen lahko uporabimo preprost OBD-diagnostični vmesnik Vgate™ ELM327, ki je prikazan na sliki 2.6. Na vmesnik se povežemo prek osebnega računalnika ali pametnega telefona z eno izmed univerzalnih aplikacij. Promet generiramo tako, da v aplikaciji izberemo prikaz enega izmed senzorjev, ki so na voljo za emuliranje.

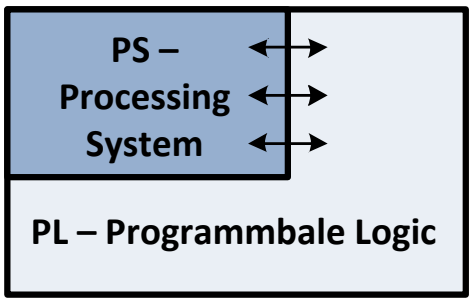
Slika 2.6: Vgate ELM327 vmesnik

Za lažji fizični dostop do pinov CAN-High in CAN-Low uporabimo odcepni adapter iz OBD-II priključka na DB-9.

2.3 Strojna platforma analizatorja

Postavili smo si cilj poiskati dovolj zmogljivo in prilagodljivo platformo, na kateri bomo poizkušali postaviti željen logični analizator. Ob raziskovanju novejših trendov razvoja elektronskih sistemov in ponudbe na trgu opazimo vse več novih produktov in sistemov, ki temeljijo na SOC (ang. System On Chip). Koncept je na trgu prisoten že kar nekaj časa. Pomeni zmožnost implementacije celotnega sistema na en sam čip, v zameno za več različnih čipov z različnimi lastnostmi in funkcijami. Večinoma se srečujemo z ASIC (ang. Application Specific Integrated Circuit) SOC. Ti so namenjeni dotični uporabi predvsem v elektronskih prenosnih napravah. Njihova pomanjkljivost se odraža v dolgem načrtovanju, dragi poti do proizvodnje, predvsem pa v slabi fleksibilnosti. Tako niso primerni za razvoj produkta, ki je razvit kot hiter odgovor na potrebo trga. Za tak namen imamo v zadnjih nekaj letih na voljo

APSOC (ang. All-Programmable SOC). To pomeni sistem na čipu, ki načrtovalcu daje določeno možnost prilagajanja za konkretno aplikacijo. To zmožnost daje FPGA (ang. Field Programmable Gate Array) programirljivo polje vrat, ki je lahko implementirano na isti rezini skupaj z mikroprocesorjem. Tako kombinacijo (slika 2.7) uporabimo tudi in predvsem takrat, ko želimo procesorsko jedro razbremeniti paralelnih ter drugih težjih nalog v realnem času. Poleg podjetja Xilinx, ki ponuja čipe družine *Zynq-7000*, najdemo na trgu produkte *SmartFusion* in *SmartFusion2* družbe MicroSemi [17] ter Alterine [18] *Stratix-10*, *Arria-10*, *Arria-V* ter *Cyclone-V*. Pri večini najdemo programirljivo logiko v povezavi z enim izmed ARM-dvojedrnih procesorskih sistemov [19, stran 1-9].

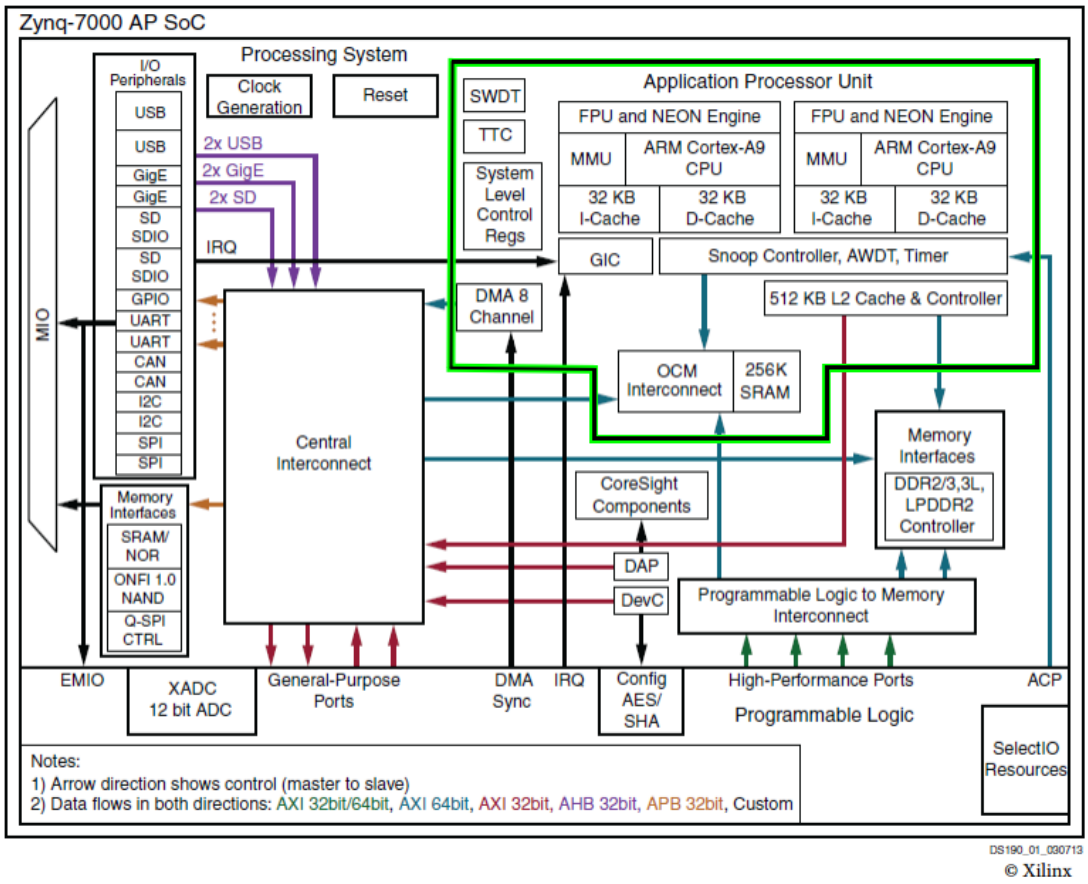


Slika 2.7: Shema SOC

2.3.1 Zynq SOC

Na voljo imamo razvojno ploščico *MicroZed* s SOC *Zynq-7010*, zato si pobližje poglejmo glavne lastnosti tega čipa. Omenjene bodo le najpomembnejše lastnosti. Vsi člani družine *Zynq-7000* (*Z-7010, Z-7015, Z-7020, Z-7030, Z-7035, Z-7045, Z-7100*) imajo enako glavno arhitekturo. Kot pri ostalih tovrstnih SOC jo ločimo na PL (programirljiva logika) in PS (procesorski sistem).

Bistvo PS je dvojedrni *ARM Cortex-A9* procesor, ki s svojo periferijo tvori APU (ang. Application Processing Unit). V *Z-7010* tiktaka pri 866MHz. Komunikacija z zunanjostjo je dosežena preko MIO (ang. Multiplexed Input/Output), ki zagotavlja 54 fleksibilnih priključkov. Če jih potrebujemo več, lahko do zunanjosti pridemo tudi preko EMIO (ang. Extended MIO), ki je posredna pot preko PL. Na sliki 2.8 vidimo, da imamo na razpolago I/O (ang. Input/Output) s standardnimi komunikacijskimi vmesniki USB(x2), SPI(x2), I2C(x2), UART(x2), CAN(x2), SD(x2), GigE(x2), GPIO(4x32bit) [19, stran 15-22].



Slika 2.8: Zynq 7000 AP SOC [19, stran 17]

PL-del sloni (odvisno od verzije) na *Artix-7 (Z-7010)* ali *Kintex-7* FPGA družini. Iz istih serij FPGA (*Xilinx 7*) najdemo tudi komponento blokovni RAM (BRAM). Ta nam omogoča implementacije ROM (ang. Read Only Memory), RAM (ang. Random Access Memory) in FIFO (ang. First In First Out) pomnilnikov.

AXI-standard

AXI (ang. Advanced eXtensible Interface), trenutna verzija AXI4, je del ARM AMBA 3.0 odprtega standarda, ki je bil razvit s strani podjetja ARM za uporabo znotraj mikroprocesorjev. Med razvojem pa je to postal uveljavljen standard za komunikacijo znotraj čipov, še posebej med komponentami SOC. Tu imamo v mislih povezave med več procesorji ali bloki v programirljivi logiki [19, stran 31-33].

Na sliki X opazimo bloke **Interconnect**. V svojem bistvu so to stikala v PS-delu čipa Zynq, ki upravljajo in razporejajo promet med AXI-vmesniki. Če želimo npr. iz enote APU dostopati do komunikacijskih vmesnikov ali PL-dela, moramo

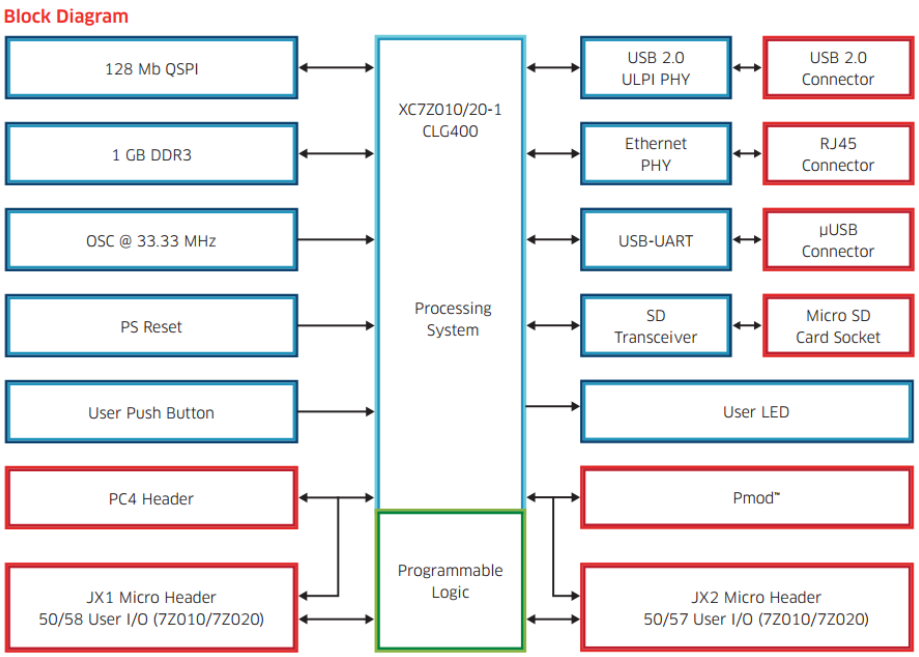
zgraditi povezavo čez Central Interconnect, ki je kot nekakšno glavno križišče povezav znotraj čipa.

2.3.4 MicroZed

MicroZed je nizkocenovna razvojna ploščica, bazirana na družini *Zynq-7000 SoC* in bo uporabljena kot osnova za razvoj logičnega analizatorja. Dotična verzija vsebuje *Z-7010* skupaj s pripadajočo periferijo, ki je prikazana na spodnjem blokovnem diagramu. Ploščica nima JTAG-programatorja, zato moramo pred programiranjem programirljive logike zagotoviti zunanji programator, npr. *Digilent JTAG HS2* [20].

MicroZed zagotavlja 100 I/O povezav na PL-stran čipa Zynq, vendar do njih lahko dostopamo šele v kombinaciji z eno izmed dodatnih razširitvenih kartic, ki ima dva MicroHeader priključka (JX1, JX2). Na voljo imamo *MicroZed Breakout Carrier Card* podjetja Avnet. Razširitveno kartico nujno potrebujemo, saj želimo pripeljati signal za vzorčevanje na PL-del čipa Zynq. Blokovni diagram ploščice je prikazan na sliki 2.9.

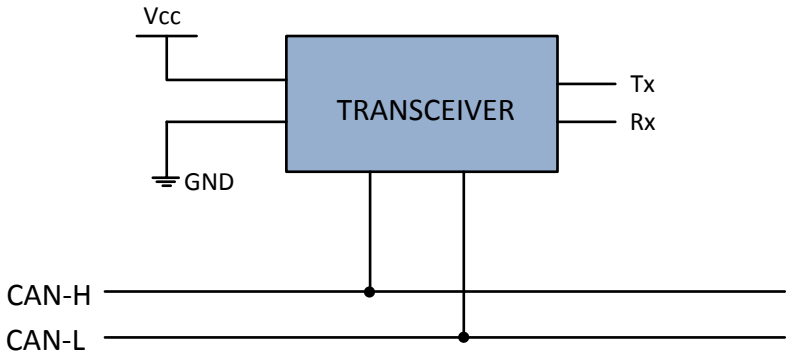
Razvojna ploščica ravno tako zagotavlja 10/100/1000 Ethernet vmesnik in tako daje dodatno podporo za umestitev končnega produkta v omrežje.



Slika 2.9: MicroZed blokovni diagram [20]

2.4 Priklop na vodilo

Za logično analizo CAN-vodila oziroma CAN-H in CAN-L-vodnikov potrebujemo najprej CAN sprejemno in oddajno enoto (ang. transceiver), ki bo diferencialno napetost med vodnikoma prilagodil na napetostni nivo ploščice (slika 2.10). Tak logični signal lahko peljemo na I/O v PL-del čipa Zynq. V tem primeru potrebujemo samo en vhod, če bi vzorčili paralelno vodilo, bi pač uporabili n-vhodov. MicroHeader konektor zagotavlja do 100 I/O v PL. *MicroZed* je skupaj z razširitveno ploščico poleg 2.2 V in 1.8 V mogoče nastaviti na napetost 3.3 V. To nam omogoča nekaj dodatne prilagodljivosti. Glede na to, da ima Zynq dva komunikacijska vmesnika za CAN, bi lahko na enega izmed njiju priklopili transceiver in s tem postavili napravo kot samostojno enoto v CAN-omrežje, vendar to ni predmet te naloge.



Slika 2.10: Priklop sprejemno-oddajne enote na vodilo

2.4.1 Transceiver

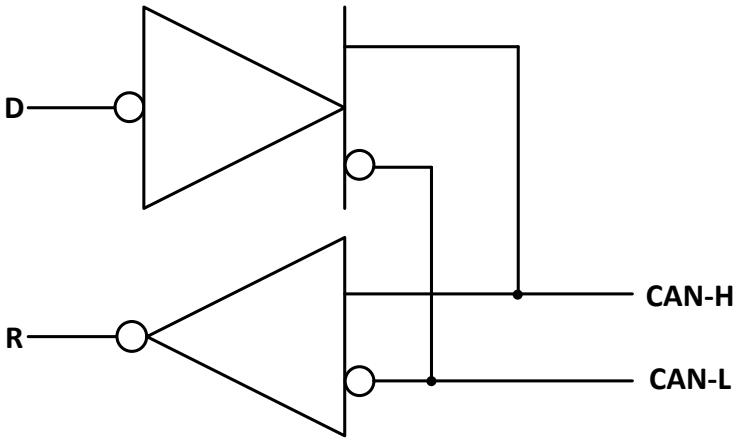
Transceiver je vmesni člen med nekimi napetostnimi nivoji, ki vladajo v nekem sistemu ali v našem primeru na vodilu in nivojih na katerih operira µC. Z osciloskopom analiziramo napetosti med CAN-H in CAN-L. Ugotovimo, da potrebujemo transformacijo 5 V diferencialne napetosti na enega izmed tistih nivojev, ki jih ponuja ploščica [21].

Za ta namen uporabimo *Texas Instruments SN65HVD232Q-Q1 3.3V CAN Transceiver*. Omogoča sprejem ali oddajanje signala do hitrosti 1 Mbit/s in ustreza ostalim zahtevam ISO 11898 standarda. Odvisnosti izhoda transceiverja od razmerij na CAN-vodilu si oglejmo v spodnji tabeli 2.2. Na izhodu bo nizki nivo ob pogoju $V_{ID} \geq 0.9$ V, visok nivo je ob $V_{ID} \leq 0.5$ V in odprtih sponkah, med pogojem $0.5\ V \leq V_{ID} \leq 0.9$ V je nedefinirano stanje.

Diferencialna vhoda	Izhod R
$V_{ID} \geq 0.9$ V	L (Nizek)
$0.5 \text{ V} < V_{ID} < 0.9$ V	Nedefiniran
$V_{ID} \leq 0.5$ V	H (Visok)
Odprte sponke	H (Visok)

Tabela 2.2: Sprejemni del transceiverja [21]

Oddajnega dela ne uporabljamo. Ker so vhodi in izhodi čipa vezani skupaj (spomnimo, da velja to za vse čipe, priključene na CAN-vodilo, vidno je na spodnji sliki 2.11), moramo biti pozorni na stanje ob odprtih sponkah na vhodu v oddajni del. Stanje je recesivno in vhoda ni potrebno nikamor priklapljati.



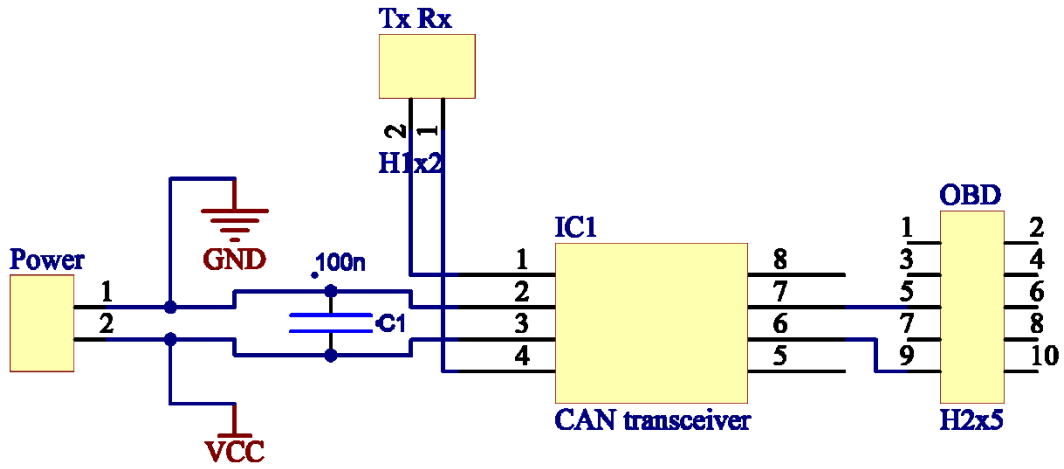
Slika 2.11: Logični diagram SN65HVD232Q (pozitivna logika) [21]

2.4.2 Miniaturno vezje

Izbrani transceiver se nahaja v SOIC-ohišju, ki je SMD (ang. Surface Mounted Device). Za lažjo uporabo in priklop odločimo narediti univerzalno miniaturno vezje z zahtevami:

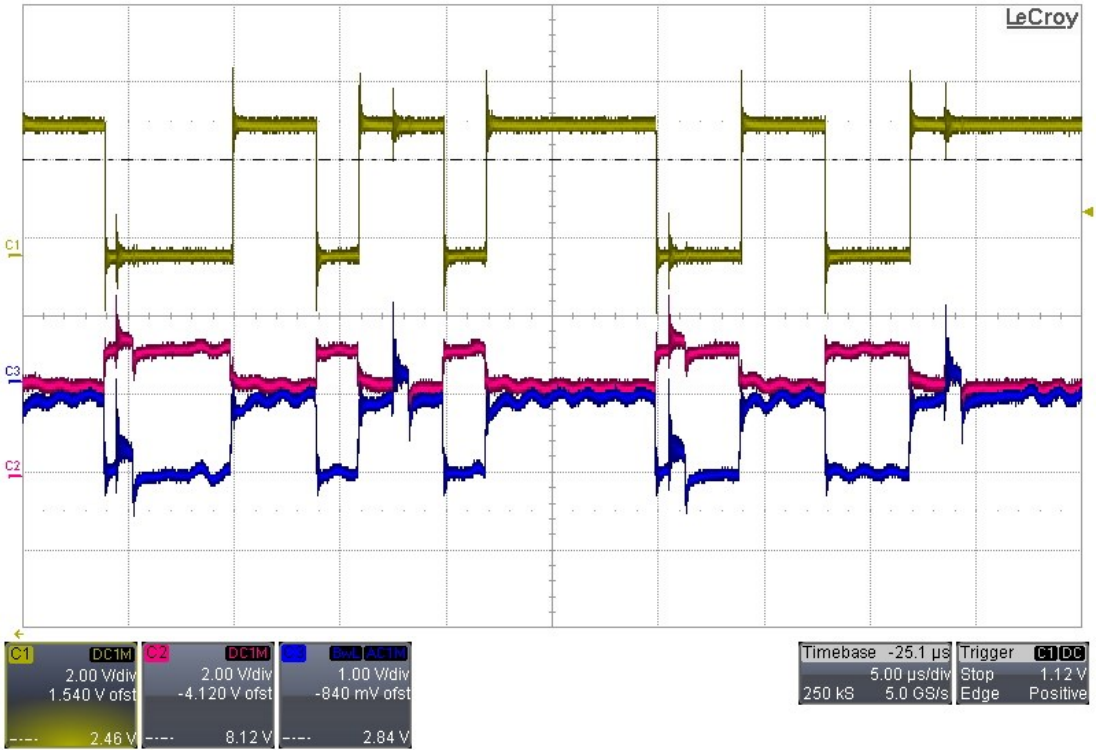
- vsebuje čip SN65HVD232Q,
- DB-9 priključek na CAN-H in CAN-L,
- priključek za izhod R, vhod D,
- priključek za V_{CC} in GND,
- blokirni kondenzator 100 nF.

Vezje oziroma prikazana shema na sliki 2.12 je bila načrtovana v okolju Altium Designer.



Slika 2.12: Shema miniaturnega vezja

Preden priklopimo vezje na razvojno ploščico, se prepričajmo o pravilnem delovanju. Spodnja slika 2.13 z osciloskopa kaže signale CAH-H (rdeč) in CAN-L (moder) in izhod R (rumen) iz oddajno-sprejemne enote.

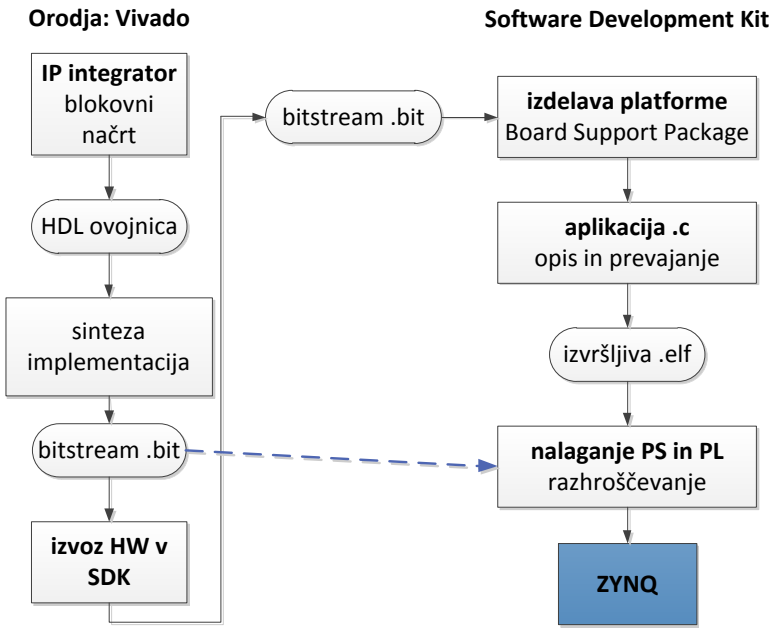


Slika 2.13: Zajem CAN-H, CAN-L in R z osciloskopom

3 Gradniki strojne platforme Zynq

Pri običajnem µP je strojna platforma že definirana. Proizvajalec določi lastnosti in periferijo procesorja ob samem načrtovanju čipa. Za uporabo takega, že definiranega µP, je potrebno zgolj nastaviti ciljno platformo v SDK (ang. Software Development Kit). Družina *Zynq-7000* je drugačna, saj prinaša številne gradnike za izgradnjo platforme po meri inženirja. To omogoča večjo prilagodljivost in hkrati zahteva intenzivnejšo pripravo, preden lahko sploh začnemo razvijati samo aplikacijo v SDK.

Za družine *čipov Ultrascale*, *Virtex-7*, *Kintex-7*, *Artix-7* in *Zynq-7000* Xilinx [22] zagotavlja orodje Vivado Design Suite, ki je nekakšen naslednik ISE Design Suite. Prav tako ponuja orodje Xilinx SDK, ki sloni na Eclipse platformi. Na sliki 3.1 je prikazan celoten postopek razvoja neke rešitve, ki bazira na čipu Zynq:



Slika 3.1: Potek razvoja na čipu Zynq [23]

Začnemo v okolju Vivado, kjer z IP-integratorjem naredimo blokovni načrt, ta pa namesto nas opiše gradnike v jeziku (HDL-ovojnica). Tu lahko izbiramo med že narejenimi gradniki (brezplačni in plačljivi) ali pa ustvarimo gradnik po meri. IP-komponenta ali gradnik predstavlja neko zaključeno funkcionalno celoto, npr. blokovni RAM, Zynq Processing System, AXI Interconnect ipd. Grafično konstruiranje takega sistema nam je v veliko pomoč, saj bi bilo drugače načrtovanje oziroma povezovanje gradnikov precej kompleksen postopek. Ko smo zadovoljni s specifikacijo lahko poženemo sintezo in implementacijo vezja. S tem imamo pripravljen *bitstream.bit* za programiranje PL-dela, narejeno specifikacijo, pa izvozimo v SDK [23].

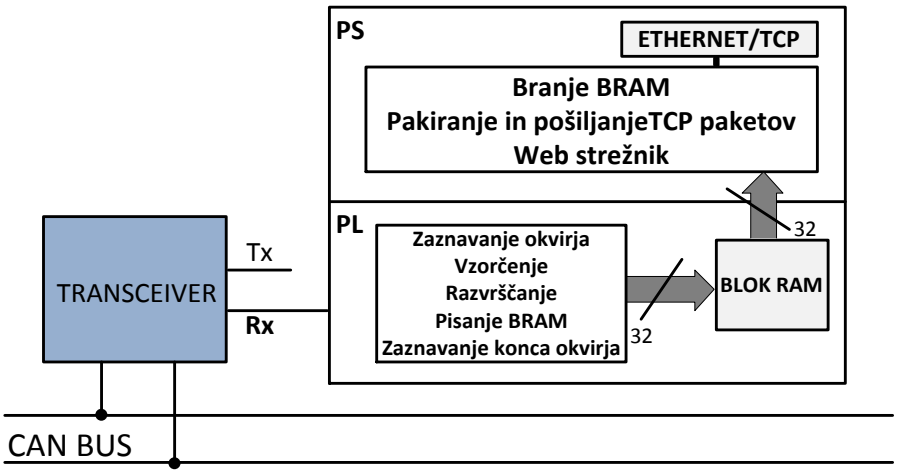
Šele po izvozu v SDK se pripravi knjižnica in izdela BSP (ang. Board Support Package). Od tu naprej lahko razvijamo aplikacijo v jeziku *C*. Za preizkus delovanja se sprogramira PL-del in požene izvršljivo *.elf* datoteko. Če bi želeli uporabljati samo PS-del, torej programirati samo v jeziku C, lahko začnemo delo kar v okolju SDK, in sicer tako, da izberemo že generičen BSP uporabljene ploščice. Prav tako nam ni potrebno uporabljati okolja SDK, če se želimo ukvarjati samo s programirljivo logiko.

FPGA bomo programirali v VHDL (ang. Very high-speed integrated circuit Hardware Description Language), ki je v svojem bistvu jezik za opis izredno hitrih integriranih vezij [24].

3.1 Delitev na strojno in programsko opremo

Preden začnemo s programiranjem PL-dela, postavimo grob koncept delovanja celotnega analizatorja. Pri tem moramo upoštevati odnos med PL in PS-delom, kot je prikazano na sliki 3.2.

Narediti bomo morali svoj IP-gradnik za vzorčenje signala. Glede na to, da pričakujemo največjo hitrost 1 Mbit/s in bi radi imeli vsaj štiri vzorce vsake vrednosti bita, moramo zajemati vhod s frekvenco 5 MHz. Ker ne želimo zasedati pasovne širine Ethernet povezave, ko na vodilu ni prometa, moramo vgraditi zaznavo začetka okvirja. Prav tako želimo zaznati konec. Med vzorčenjem moramo vzorcem dodajati časovno značko in jo skupaj s podatki vpisati v blok RAM. Urediti moramo signalizacijo do PS-dela, saj moramo sporočiti, kdaj je pomnilnik poln in je pripravljen na branje.



Slika 3.2: Blokovna shema analizatorja

V PS-delu podatek iz blok RAM-a preberemo, ustrezno zapakiramo in pošljemo prek Ethernet povezave s TCP (ang. Transmission Control Protocol) protokolom. Ko končamo, signaliziramo PL-delu, da je pomnilnik prost za pisanje. Da bo uporabnik lahko dobil dostop do vzorčenih podatkov, moramo vzpostaviti HTTP (ang. Hyper Text Transfer Protocol) strežnik.

3.2 Blokovni RAM

Kot je bilo že omenjeno, imamo v FPGA-ju na voljo že pripravljene strukture blok RAM (BRAM). Za njihovo uporabo v projektu poiščemo IP-gradnik z imenom Block Memory Generator. Ker želimo dostop iz PS in PL-dela, ga nastavimo kot *True Dual Port RAM*. Dostop iz PS mu zagotovimo prek AXI-vodila, za ta namen moramo v lastnostih izbrati način *BRAM controller*. V lastnostih opazimo še, da sta oba priključka (port A in port B) široka fiksnih 32 bitov, kar pomeni pisanje 32 vzorcev vodila naenkrat.

3.3 Namenski IP-gradnik za vzorčenje vodila

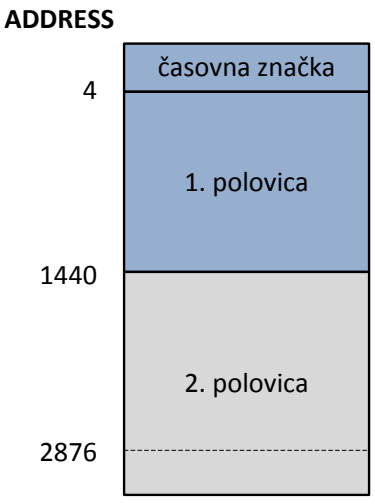
Za samo programiranje IP-gradnika, ki bo vzorčil vodilo, raje uporabimo Xilinx ISE Design Suite 14.7. Čeprav je to starejša različica Xilinxovega razvojnega okolja, nam za razliko od novejše (Vivado) daje možnost samodejnega kreiranja testne strukture za simulacijo vezja, s tem si nekoliko olajšamo delo.

Nov IP definiramo kot sinhroni proces, odvisen od vzorčevalne frekvence 5 MHz. Uro *clk* bomo pripeljati iz PS-dela in jo bomo nastavili pri poznejši

konfiguraciji. Določiti moramo bitni vhod za zajemanje *data_in*, vhodne *flags_in* in izhodne *flags_out* zastavice ter signal za resetiranje blokovnega RAM *rstb*. Pomembno je, da si ogledamo, kakšne zahteve glede priključkov ima že narejena komponenta blok RAM, saj je potrebno definirati ekvivalentne priključke za pravilen priklop nanj.

Ker pišemo v blok RAM 32 bitov naenkrat (4 x 8 bit), bo potrebno definirati medpomnilnik *cache* iste dolžine, iz katerega bomo ob napolnitvi paralelno prenesli vseh 32 bitov hkrati. Ko so prisotni pogoji za vzorčenje, moramo zagotoviti, da se bo ob vsakem urinem ciklu vzorec prepisal v medpomnilnik, drugače bo prišlo do izgube podatkov. Prav tako med vsako spremembo ure preverjamo, ali je še vedno omogočeno vzorčenje.

Namesto dveh bomo uporabili en sam gradnik blok RAM in ga glede na shranjevanje in branje podatkov razdelili na dve polovici (slika 3.3). Delitev potrebujemo zaradi rezervacije dostopa ob branju podatkov samo za PS-del. Izogniti se moramo situaciji hkratnega branja in pisanja. Zagotoviti moramo, da bo PL-del po končanem pisanju do naslova 1440 začel pisati drugo polovico. Prva pa bo na voljo za branje PS-delu. Ko PL-del konča pisanje pri 2880, lahko PS-del začne brati drugo polovico, PL pa se vrne s pisanjem na začetek. Velikost pomnilnika določimo glede na zahteve za prenos s TCP-protokolom, kjer uporabljamo paket velikosti 1440 bajtov. Is skice vidimo, da časovna značka pripada celotnemu pomnilniku oziroma dvema paketoma (2 x 1440). Dejanski čas posameznega odčitka bomo računali naknadno.



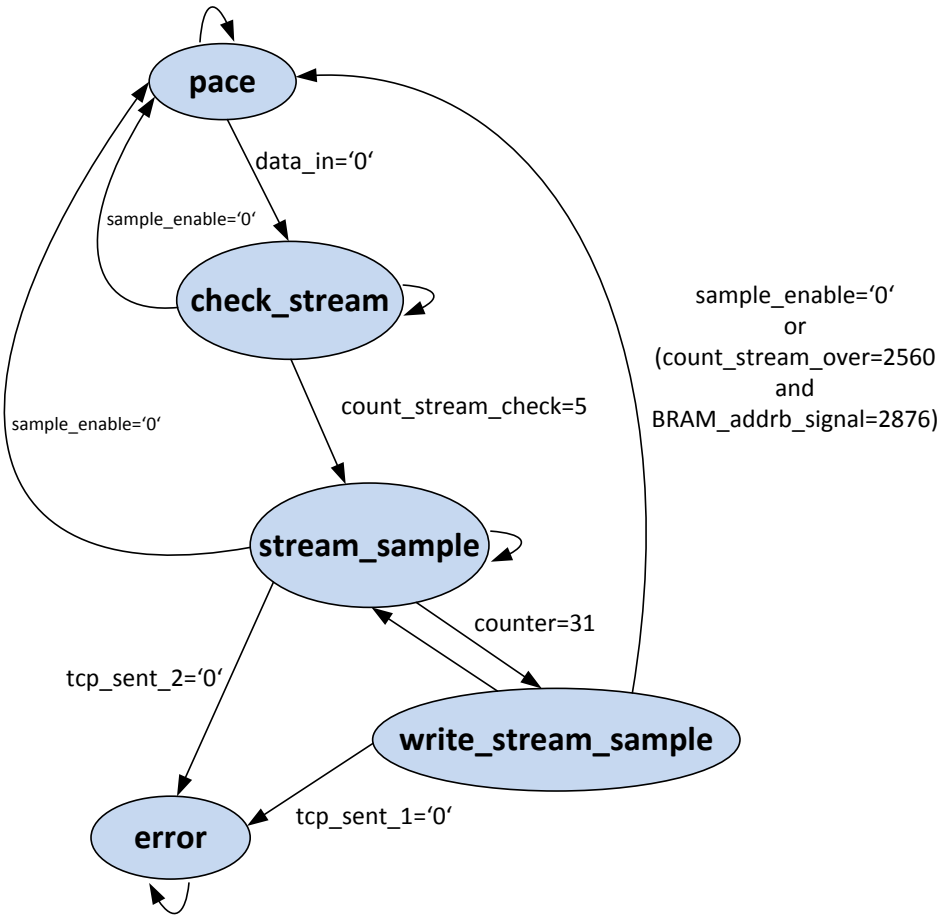
Slika 3.3: Struktura pomnilnika BRAM

Če želimo podatkom dodati časovno značko, moramo vsak urin cikel prešteti. Štetje začnemo, ko je vzorčenje omogočeno in je prišel prvi podatek. Pozorni moramo biti, da ne definiramo premajhne širine števca, saj bi drugače ob predolgem času vzorčenja lahko velikost števca presegli. 32-bitna širina bo zadoščala za nekaj več kot 14 minut vzorčenja. Koda je prikazana spodaj:

```
if sample_enable = '1' and stream_on='1' then--napaka 1 clk
   count_time<=count_time + "1";
else --povečujem števec za 1, kar v realnosti pomeni 0,2us
   count_time<=(others=>'0');
end if;
```

Ostale funkcionalnosti našega IP-gradnika so vgrajene v sekvenčnem avtomatu s petimi stanji, ki je prikazan na sliki 3.4.

3.3.1 Diagram prehajanja stanj



Slika 3.4: Diagram prehajanja stanj

Stanje *peace*

Zaznavanje dogajanja na vodilu implementiramo v stanje *peace*, ki je hkrati začetno stanje avtomata. Če je omogočeno vzorčenje, čakamo na pojav logične '0', ki jo shranimo v medpomnilnik. Povečamo števec medpomnilnika *counter* in števec za zaznavo glitcha *count_stream_check*. Sledi stanje *check_stream*.

```
when peace =>
      BRAM_addrb_signal<=(others=>'0');
      BRAM_web<=(others=>'0');
      if sample_enable = '1' then
            if data_in = '0' then --na vhodu se pojavi '0'
                  stream_on<='1'; --zaženemo števec časa
                  cache(counter)<=data_in; --vpis v buffer
                  counter<=counter + 1;
                  count_stream_check<=count_stream_check + "1";
                  state<=check_stream; --gremo v naslednje stanje
            end if;
      end if;
```

Stanje *check_stream*

Stanje *check_stream* vpeljemo, ker se želimo prepričati, da ni po dolgem stanju mirovanja na vodilu prišlo do morebitne motnje. Ker vzorčimo s petkrat višjo hitrostjo, bomo v najslabšem primeru imeli štiri vzorce istega bita, zato čakamo števec *count_stream_check*, da doseže to vrednost. Če se v tem času pojavi na vhodu logična '1', ponastavimo števca in se vrnemo v stanje *peace*, drugače gremo v *stream_sample*.

```
when check_stream =>
      if sample_enable = '1' and data_in = '0' then
            count_stream_check<=count_stream_check + "1";
            if count_stream_check < "100" then --štejemo do 4 za
                  cache(counter)<=data_in;     --izločitev glitch-a
                  counter<=counter+1;
                  else  --vsakič vzorčimo in povečujemo števec
                        cache(counter)<=data_in;
                        counter<=counter + 1;
                        state<=stream_sample;
                        --ni glitch, gremo v novo stanje
                        count_stream_check<="00000";
                        --ponastavimo števec za glitch
                  end if;
            elsif sample_enable = '0' or data_in = '1' then
                  counter<=0;
                  count_stream_check<="00000";
                  state<=peace;--vrnitev v peace
            end if;
```

Stanje *stream_sample*

V stanju *stream_sample* začnemo preverjati, ali ni morda na vodilu prišlo do mirovnega stanja. To storimo s povečevanjem števca *count_stream_over* vsakič, ko je na vhodu logična '1'. Takoj ko zaznamo logično '0', števec ponastavimo. Za določitev vrednosti števca, ko na vodilu ni prometa, je potrebno poznati najpočasnejši še vzorčeni signal. To nas postavlja v položaj, kjer moramo najti kompromis med najpočasnejšim signalom in verjetnostjo, da bomo ob pretežni neaktivnosti vodila privarčevali na uporabi pasovne širine pri Ethernet povezavi. Za naš primer analizatorja se ravnamo po signalnem standardu za OBD ISO 15765-4, ki določa minimalnih 250 kbit/s. S tem dokončno določimo analizatorju območje vzorčenja signalov od 250 kbit/s do 1 Mbit/s. Največjo vrednost števca *count_stream_over* določimo z upoštevanjem vseh mejnih možnosti:

- 128 – Maksimalno število bitov v okvirju CAN 2.0B
- 5 MHz – Frekvenca vzorčenja
- 250 kbit/s – Hitrost najpočasnejšega še vzorčenega signala

Število bitov v CAN-okvirju pomnožimo z razmerjem med frekvenco vzorčenja in hitrostjo najpočasnejšega signala. Vrednost števca bo *2560*, kar pomeni ob popolni sinhronizaciji vzorčevalne ure s signalom največje število vzorcev enega okvirja. V praksi se seveda takšna mejna kombinacija ne more zgoditi, a je rezerva dobrodošla, saj moramo upoštevati še morebitno vrivanje bitov.

```
when stream_sample =>
    if sample_enable = '1' then
        if data_in = '1' then
    if count_stream_over < "101000000000" then
            --128*5(MHz)/250kbit/s=2560
                count_stream_over<=count_stream_over + "1";
    end if;
        else
            count_stream_over<="000000000000";
        end if;
        ------------polnenje bufferja 32bit-----------
        if counter < 31 then
            BRAM_web<=(others=>'0');
            -- ponastavimo še enkrat, če pridemo iz stanja
            -- write_stream_sample ali pisanja timestamp
            cache(counter)<=data_in;
            counter<=counter + 1;
            if counter = 25 and BRAM_addrb_signal = 0 then
                BRAM_doutb<=std_logic_vector(count_time - "11000");
                --pravilni timestamp je 24 urinih ciklov nazaj
                --upoštevamo napako 1 clk kasnejšega začetka štetja
                BRAM_web<=(others=>'1');
```

```
            end if;
            if counter = 26 and BRAM_addrb_signal = 0 then
                BRAM_web<=(others=>'0');
            end if;
        end if;
        if counter=31 then--buffer za vpis je poln
            BRAM_doutb(30 downto 0)<=cache (30 downto 0);
            BRAM_doutb(31)<=data_in;
            counter<=0;
            BRAM_addrb_signal<=BRAM_addrb_signal + 4;
            --povečamo naslov BRAM
            BRAM_web<=(others=>'1');
            if BRAM_addrb_signal = 1436 then --poln 1. del BRAM
                  if tcp_sent_2 = '1' then--preverimo flag iz PS
                        flags_out(31 downto 28)<="0001";--postavimo
                        state<=write_stream_sample;     --flag za PS
                  else
                        state<=error;--prenos prepočasen
                  end if;            --gremo v stanje error
            else
                state<=write_stream_sample; --skok v novo stanje
            end if;
        end if;
    elsif sample_enable = '0' then--konec vzorčenja
        count_stream_over<="000000000000";
        state<=peace;
        counter<=0;
    end if;
```

Časovno značko vpisujemo, ko je *BRAM_addrb_signal=0000*. To se zgodi med prvo polnitvijo medpomnilnika *cache*. Točen čas vpisa ni pomemben. Izberemo si npr. stanje števca *counter=25*. Paziti moramo samo na pravilen izračun časovne značke, ki je enaka trenutni vrednosti *count_time* zmanjšani za *24*.

Vpisovanje v pomnilnik se na željen naslov *BRAM_addrb_signal* zgodi ob vrednosti 4-bitnega signala *BRAM_web<=(others=>'1')*, ki ga v naslednjem urinem ciklu spet postavimo na stanje *0000*. Z vpisovanjem 32 bitov naenkrat v fizičnem smislu napolnimo 4 x 8 bitov oziroma štiri naslove z 1 bajtom, zato je potrebno *BRAM_addrb_signal* povečevati za vrednost 4.

Prva polovica pomnilnika je polna ob vrednosti *BRAM_addrb_signal = 1436*. Takrat preverimo zastavico iz PS-dela (če je druga polovica prosta za pisanje) in postavimo zastavico za branje prve polovice, drugače gremo v stanje *error*. Enak postopek sledi ob *BRAM_addrb_signal = 1876*.

Stanje *write_stream_sample*

Glavna funkcija tega stanja je preverjanje, če se nič več ne dogaja na liniji. Četudi *count_stream_over* doseže vrednost 2560, moramo počakati s prenehanjem vzorčenja, vse dokler ne napolnimo celoten blokovni RAM (vseh 2880 naslovov). Razlog tiči v razvrščanju podatkov na drugi strani Ethernet povezave, kar bomo opisali v poznejšem poglavju. Če v tem času pride na vhod logična *'0'*, ponastavimo *count_stream_over* in vzorčimo naprej po istem postopku.

Podobno kot v *stream_sample* se tudi tu upravlja z zastavicami v odnosu s PS-delom. Ko je *BRAM_addrb_signal = 2876*, ga ponastavimo in postavimo dovoljenje za branje druge polovice blokovnega RAM. Če zaradi praznega vodila prekinemo z vzorčenjem, se namesto v *stream_sample* vrnemo v stanje *peace*.

```
when write_stream_sample =>
        if sample_enable = '1' then
            if count_stream_over < "101000000000" then
                if data_in = '1' then--števec ni še preštel
                    count_stream_over<=count_stream_over + "1";
                else
                    count_stream_over<="000000000000";
                end if;
                cache(0)<=data_in;
                state<=stream_sample;
                counter<=counter + 1;
                if BRAM_addrb_signal = 2876 then--cel BRAM napolnjen
                    BRAM_addrb_signal<=(others=>'0');
                    if tcp_sent_1 = '1' then
                        flags_out(31 downto 28)<="0010";
                        state<=stream_sample;
                --se vrnemo nazaj v stream_sample
                    else
                        state<=error;
                    end if;
                end if;

            elsif count_stream_over = "101000000000" then
            --števec je preštel, vendar napolniti moramo cel BRAM
                if BRAM_addrb_signal = 2876 then --cel BRAM poln
                    if data_in = '1' then
                        count_stream_over<=(others=>'0');
                        if tcp_sent_1 = '1' then
                            flags_out(31 downto 28)<="0010";
                            BRAM_addrb_signal<=(others=>'0');
                            state<=peace;
                        else
                            state<=error;
                        end if;
                    else
                        counter<=counter + 1;
                        cache(0)<=data_in;
```

```
                        if tcp_sent_1 = '1' then
                            flags_out(31 downto 28)<="0010";
                            state<=stream_sample;
                            --gremo v prejšnje stanje
                            BRAM_addrb_signal<=(others=>'0');
                        else
                            state<=error;
                        end if;
                    end if;
                else--ni še napolnjen BRAM, še vedno vzorčimo
                    counter<=counter + 1;
                    cache(0)<=data_in;
                    state<=stream_sample;
                end if;
            end if;
    elsif sample_enable = '0' then
            count_stream_over<=(others =>'0');
            state<=peace;
            counter<=0;
    end if;
```

Stanje *error*

V stanju *error* se dokončno ujamemo, če pride do prepočasnega prenosa, takrat postavimo zastavico napake.

```
    when error =>
        flags_out(31 downto 28)<="0100";
```

3.4 Testna struktura

Preden pričnemo z razvojem aplikacije v jeziku *C*, se je s simulacijo dobro prepričati o pravilnem delovanju našega gradnika. Okolje ISE omogoča delno samodejno kreiranje testne strukture. Za naš primer še dodajmo nekoliko poenostavljeno definicijo blokovnega RAM ter sam proces z opisom njegovega obnašanja:

```
--BRAM definition
type RAM is array (0 to 719) of STD_LOGIC_VECTOR (31 downto 0);
signal BRAM: RAM;
signal addrb_convert: integer range 0 to 719;
```

V poenostavljeni verziji pomnilnika pišemo 32 bitov na isti naslov, *BRAM_addrb* pa se povečuje za vrednost *4*. Z logičnim pomikanjem za dve mesti v desno lahko dobimo za faktor *4* zmanjšano vrednost naslova. SRL (ang. Shift Right Logic) ne

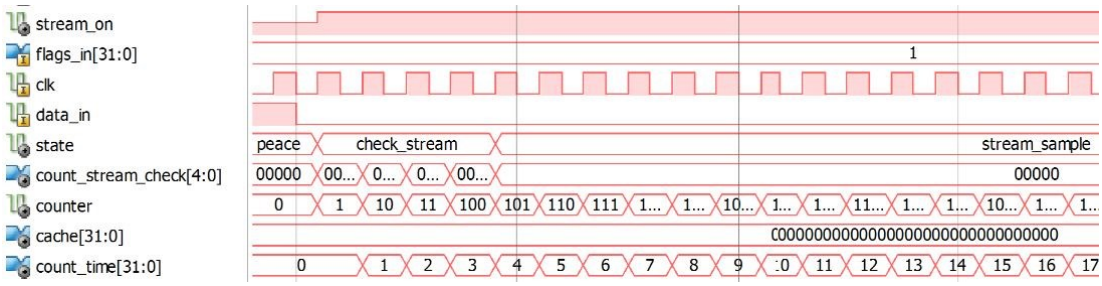
deluje nad podatkovnim tipom *std_logic_vector*, zato je potrebna predhodna pretvorba v tip *unsigned*.

```
--BRAM simulation
process (clk)
    begin
         if rising_edge (clk) then
            if (BRAM_enb = '1') then
                if (BRAM_web = "1111") then--shift desno 2 mesti
             addrb_convert<=to_integer((unsigned(BRAM_addrb))srl 2);
                        BRAM(addrb_convert) <= BRAM_doutb;
                end if;
            end if;
        end if;
    end process;
```

Za simulacijo vhoda v analizator z na primer 250 kbit/s moramo spreminjati vhodni signal v analizator 20-krat počasneje od ure sistema oziroma vzorčenja (`wait for clk_period*20`). Vhod spreminjamo tako, da simuliramo dejanski CAN-okvir.

3.4.1 Simulacija

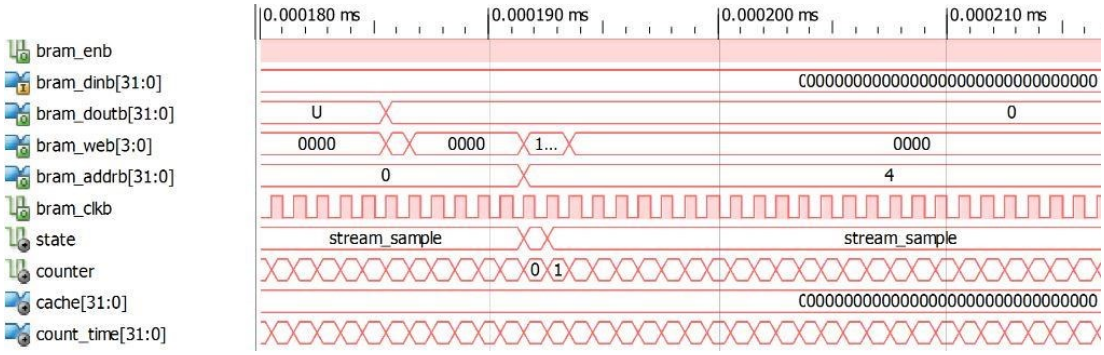
Preden poženemo simulacijo, še ročno nastavimo zastavici, ki prihajata iz PS-dela (*tcp_sent_1* in *tcp_sent_2*), na logično *'1'*. Drugače bo PL-del skočil v stanje *error*. Najprej si oglejmo dogajanje ob pojavu logične ničle na vhodu. Iz stanja *peace* pravilno skočimo v *check_stream*, kjer ostanemo 4 urine cikle oziroma dokler *count_stream_check* ne doseže vrednosti *4*. Takrat števec ponastavimo in skočimo v stanje *stream_sample*. Obnašanje je prikazano na sliki 3.5.



Slika 3.5: Simulacija 1

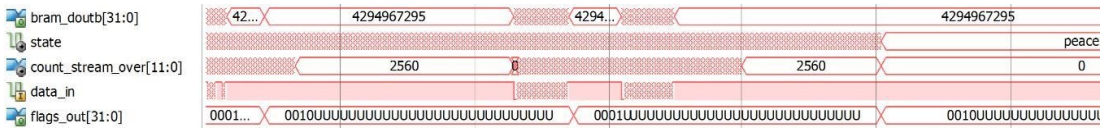
Slika 3.6 prikazuje vpis prve časovne značke. Vpisujemo, ko je vrednost števca medpomnilnika *25*. Signal *bram_web* se za urin cikel postavi na vrednost *"1111"*, kar pomeni, da pride do vpisa časovne značke, zmanjšane za vrednost *24*. Na simulaciji

že vidimo naslednji vpis po ponastavitvi števca medpomnilnika, ki traja dva urina cikla.



Slika 3.6: Simulacija 2

Zadnji izsek simulacije na sliki 3.7 prikazuje spreminjanje signalnih zastavic *flags_out* po končanem pisanju polovice blokovnega RAM. Opazujemo lahko signal *count_stream_over*, ki se postavi na največjo dovoljeno vrednost *2560*, kljub temu pa se vzorčenje nadaljuje, dokler ni poln celoten pomnilnik. Ker v tem času vhod *data_in* nikoli ne doseže logične ničle, gremo v stanje *peace*.

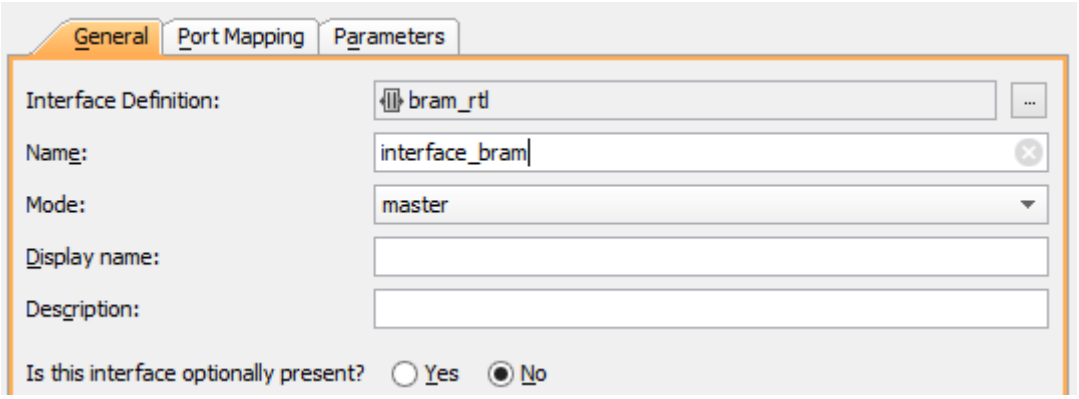


Slika 3.7: Simulacija 3

3.5 Nova IP-komponenta

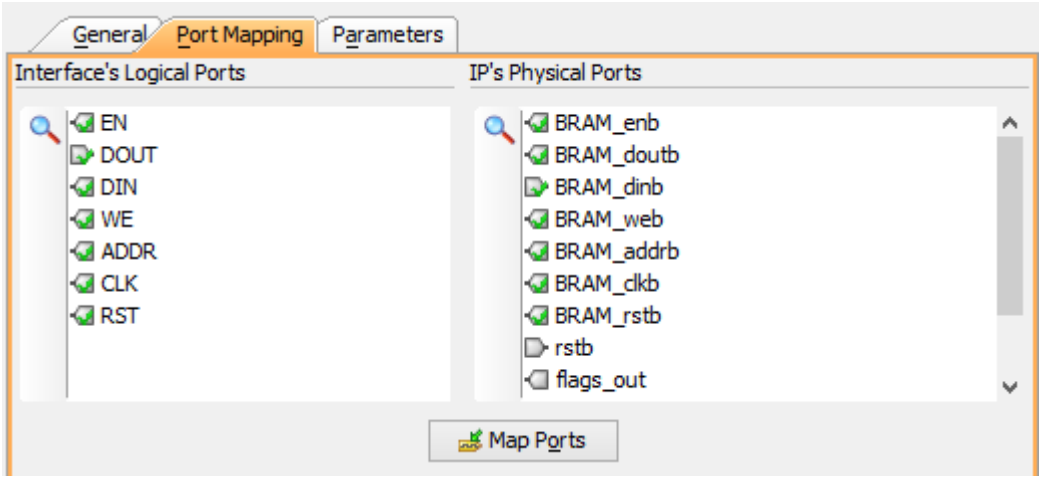
Pripravljeno imamo torej novo entiteto ***Analyzer*** in jo želimo v okolju Vivado vnesti v nov blokovni načrt, ki ga poimenujemo ***AnalyzingSystem***. Preden lahko to izvedemo, jo moramo zapakirati v nov gradnik oziroma IP-komponento. To storimo s čarovnikom Package IP, kjer novi komponenti določimo ime, verzijo, knjižnico ipd.

Omenili smo že, da so priključki na blokovni RAM točno določeni. Čarovnik nam pod zavihkom Ports and Interfaces omogoča (slika 3.8), da omenjene namenske priključke združimo v skupen vmesnik oziroma vodilo, ki ga bo čarovnik za samodejno povezovanje med komponentami blokovnega načrta zaznal in priključil namesto nas. Definicijo priključkov najdemo pod imenom *bram_rtl*, določimo še ime, npr. *interface_bram* ter način *master*.



Slika 3.8: Zavihek General

Pod zavihkom Port Mapping (slika 3.9) povežemo logične priključke vmesnika s fizičnimi naše IP-komponente.



Slika 3.9: Zavihek Port Mapping

3.6 Blokovni načrt

Naš blokovni načrt že vsebuje gradnik *Block Memory Generator*. Če želimo gradnik priklopiti na PS-del prek AXI-vmesnika, mu moramo dodati komponento *AXI BRAM Controller*, kjer je dovolj samo vmesnik *BRAM_PORT_A*. Čarovnik Run Connection Automation samodejno poveže komponenti med seboj.

Knjižnica vsebuje naš na novo narejeni gradnik z imenom ***Analyzer_v1_0***, ki ga lahko dodamo v načrt. Vodili signalnih zastavic (*flags_out* in *flags_in*) povežemo na PS prek vmesnika *AXI GPIO*. *Interface_bram* je samodejno povezan na preostali priključek blokovnega RAM.

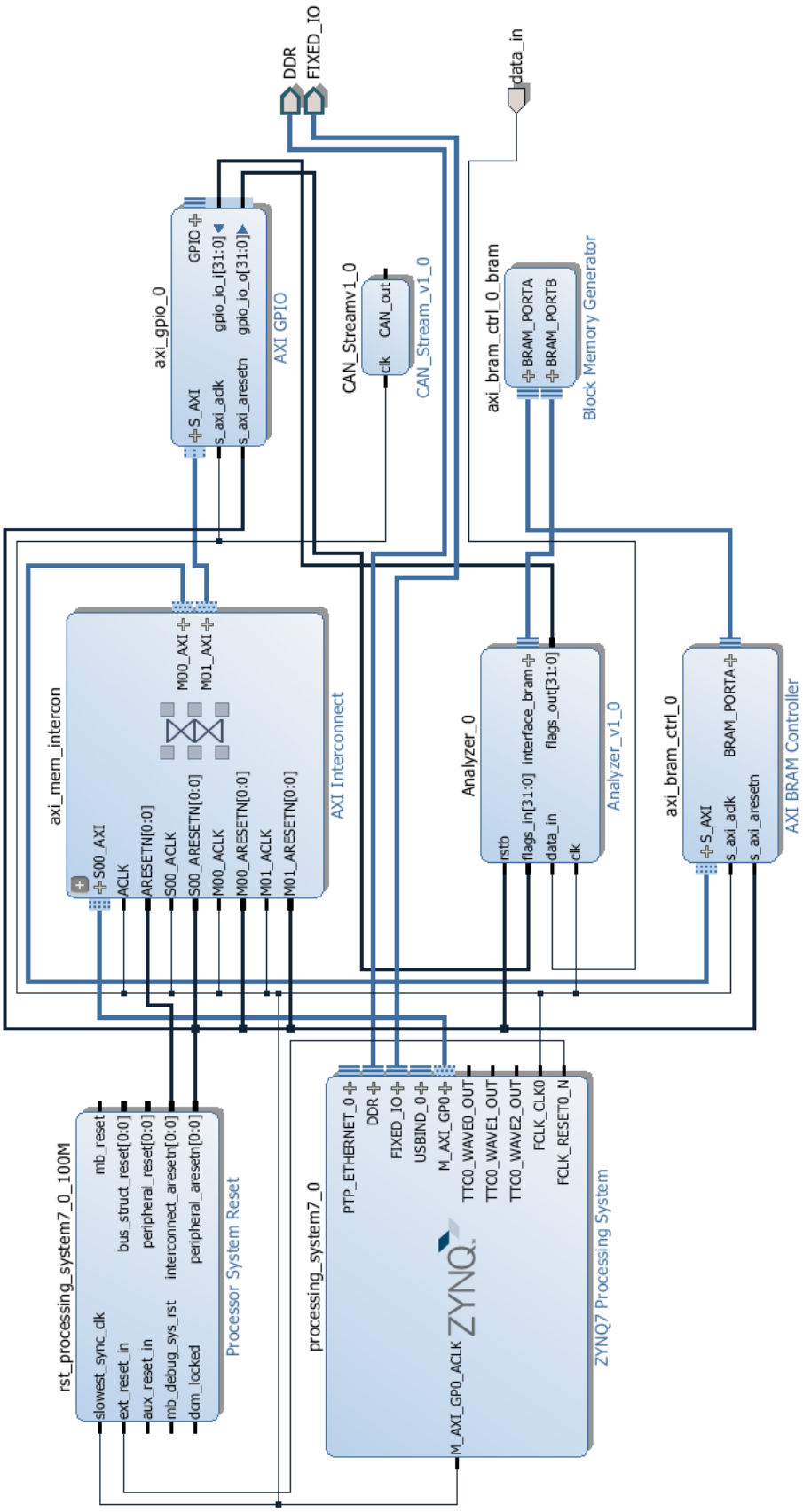
Sam procesorski sistem je zapakiran v bloku *ZYNQ7 Processing System*. V obsežnih lastnostih bloka moramo nastaviti oziroma omogočiti funkcije, ki so potrebne za uspešno povezovanje z že vnesenimi bloki v načrtu:

- za povezavo z *AXI GPIO* oziroma signalnimi zastavicami pod zavihkom PS_PL Configuration/ GP Master AXI Interface omogočimo *M AXI GP0 interface*;
- za povezavo Ethernet mora biti pod MIO Configuration/ I/O Peripherals omogočen *ENET 0*;
- kot je bilo že omenjeno, pripeljemo urin signal v PL iz PS-dela. Pod Clock Configuration/ PL Fabric Clocks omogočimo *FCL_CLK0* in nastavimo želeno frekvenco na 5 MHz.

Ostale lastnosti za naš primer ni potrebno izpostavljati. Okolje nam ponudi opcijo RunBlockAutomation, ki nam samodejno doda manjkajoče komponente in povezave. Dodana sta bloka *AXI Interconnect* za usmerjanje AXI-povezav ter *Processor System Reset*, ki upravlja s signali za ponovni zagon različnih komponent v blokovnem načrtu.

Za dostop do izhoda naše sprejemno-oddajne enote na CAN vodilu ustvarimo vhod ***data_in***, ki ga povežemo na *Analyzer_v1_0*.

Lažje, predvsem pa hitrejše razhroščevanje si omogočimo z novim gradnikom *CAN_Stream_v1_0*. Komponenta simulira promet po CAN-vodilu in jo lahko priključimo na mesto vhoda *data_in*. Celoten blokovni načrt je prikazan na sliki 3.10.



Slika 3.10: Blokovni načrt

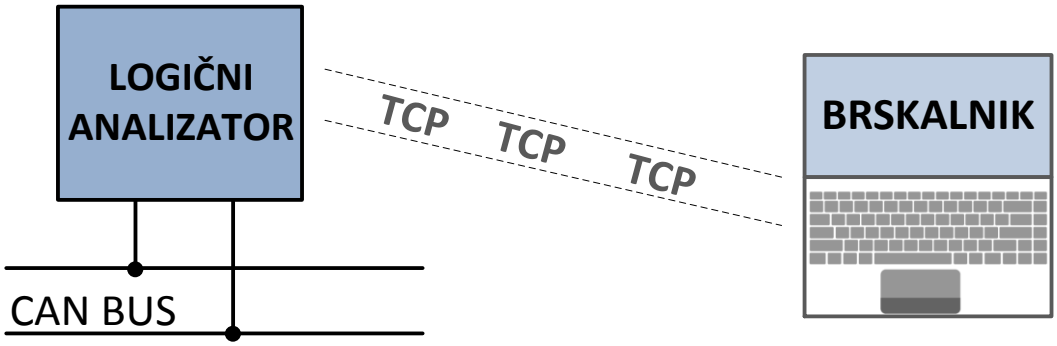
4 Programska oprema analizatorja CAN

Strojna platforma je pripravljena in jo po končani sintezi ter implementaciji skupaj z *bitstream.bit* datoteko lahko izvozimo v Xilinx SDK. Na tem mestu se vprašamo, kako začeti z zasnovo aplikacije, ki bo podatke iz blokovnega RAM ustrezno zapakirala in poslala preko Ethernet povezave. Odločimo se za uporabo operacijskega sistema *FreeRTOS* z že vgrajenim LWIP (Light Weight IP) TCP/IP skladom. Razlog je velika razširjenost ter prilagodljivost celotnega sistema. Na voljo imamo sicer neuradno, a že prilagojeno verzijo za Zynq in Vivado [25].

FreeRTOS (ang. Real-Time Operating System) je eden izmed popularnejših operacijskih sistemov za vgrajene naprave s podporo za več kot 30 arhitektur. Na trgu je pod GPL (ang. General Public License) licenco s posebno izjemo, ki pomeni, da je jedro odprtokodno, aplikacija okrog njega pa je lahko zaprta. Prav to je eden izmed razlogov za veliko razširjenost tudi v sistemih, namenjenih za komercialno rabo.

Vgrajen LWIP-sklad omogoča zaradi majhne zahteve po sistemskih virih uporabo polnega TCP-protokola na vgrajenih napravah s šibko zmogljivostjo. Pogled na ISO/OSI model kaže široko podporo protokolov. Za naše potrebe omenimo še IP (ang. Internet Protocol) ter denimo DHCP (ang. Dynamic Host Configuration Protocol). LWIP podpira tudi manj sistemsko zahteven UDP (ang. User Datagram Protocol), vendar ta ne zagotavlja prenosa podatkov v pravilnem vrstnem redu, kakor tudi ne možnosti za ponoven prenos ob napaki.

Na tej točki moramo vsaj približno vedeti, kakšna tehnologija bo uporabljena na drugi strani Ethernet povezave (slika 4.1). Transportni sloj se bo torej ravnal po TCP-protokolu, kar pomeni vzpostavitev seje z oddaljenim računalnikom oziroma brskalnikom. Vzpostavitev TCP-povezave deluje po principu strežnik-klient (klientov je lahko več), vsak pa ima svojo vtičnico (ang. socket).



Slika 4.1: Groba shema sistema

Zaradi varnosti se TCP-seja med brskalnikom in neko drugo instanco na omrežju redko vzpostavlja. Temu primerno je tudi malo tehnologij, ki takšno vzpostavitev na strani brskalnika omogočajo. HTML5 (ang. Hyper Text Markup Language) skupaj s tehnologijo JavaScript omogoča WebSocket, ki je v svojem bistvu TCP z nekaj daljšo glavo ter dodatno varnostjo. Vendar ta spremenjen protokol ni več tokovni (ang. streaming), prav tako LWIP ni več primeren za uporabo, saj tega protokola ne podpira. Rešitev najdemo v Microsoftovem vtičniku Silverlight, na katerem bo baziral uporabniški vmesnik analizatorja.

4.1 FreeRTOS

Za uvoz operacijskega sistema (OS) v SDK k strojni platformi v čarovniku izberemo FreeRTOS lwIP Socket I/O Apps. V samo drobovje operacijskega sistema se za naše potrebe ne bo nujno poglabljati. Dotična verzija FreeRTOS je *preemptive* z že vgrajenimi opravili, ki jih lahko v datoteki *config_apps.h* preprosto vključimo ali izključimo iz sistema [26]:

```
#ifndef __CONFIG_APPS_H_
#define __CONFIG_APPS_H_

#define THREAD_STACKSIZE 2048 // Empirically chosen

#define INCLUDE_TXPERF_CLIENT  1
#define INCLUDE_ECHO_SERVER    0
#define INCLUDE_WEB_SERVER     1
#define INCLUDE_TFTP_SERVER    0
#define INCLUDE_RXPERF_SERVER  0
#define INCLUDE_UTXPERF_CLIENT 0
#define INCLUDE_URXPERF_SERVER 0

#endif
```

Za naše potrebe pustimo omogočeno nit, ki skrbi za spletni strežnik ter TCP-povezavo (TXPERF klient). Opravila bi lahko ustvarili sami, vendar je najlažje, če v tem primeru predelamo obstoječe. V datoteki *dispatch.c* je definirana funkcija *launch_app_threads()*, ki zažene izbrane niti:

```
/* start webserver thread */
    if (INCLUDE_WEB_SERVER)
        sys_thread_new("httpd", web_application_thread, 0,
            THREAD_STACKSIZE,
            DEFAULT_THREAD_PRIO);
/* start tx perf client thread */
    if (INCLUDE_TXPERF_CLIENT)
        sys_thread_new("txperfd", tx_application_thread, 0,
            THREAD_STACKSIZE,
            DEFAULT_THREAD_PRIO);
```

Obe niti požene z isto prioriteto ter z enako velikim skladom. Delovanje funkcije *web_application_thread()* lahko pustimo na obrobju, posvetimo se najprej samemu TCP prenosu podatkov.

4.1.1 TXPERF.C

Prvotna funkcija *tx_application_thread()*, ki je definirana v *txperf.c*, je namenjena za testiranje TCP-povezave, in sicer oddajanja podatkov v smeri do Iperf TCP-strežnika. Zaradi varnostne politike vtičnika Silverlight lahko brskalnik vzpostavi TCP-povezavo samo kot klient, zato moramo funkcijo predelati tako, da bo naš analizator postavljen v omrežje kot strežnik. V veliko pomoč nam je datoteka *rxperf.c*, kjer najdemo konfiguracijo TCP-strežnika, ki čaka na podatke iz strani Iperf klienta.

Metoda, ki jo pri programiranju Silveriglht aplikacije (jezik *C#*) uporabimo za sprejem podatkov po protokolu TCP, uporablja podatkovni tip *byte*. To od nas zahteva razdelitev prebranega 32-bitnega podatka iz pomnilnika BRAM na 4 dele in pisanje vsakega dela posebej v ustrezno globlji medpomnilnik (`send_buf[SEND_BUFSIZE]`). Za namen branja obeh polovic pomnilnika ustvarimo funkciji *transfer_half1()* in *transfer_half2()*. Razlikujeta se v začetnih naslovih za branje pomnilnika BRAM. Vsaka ima svoj delovni medpomnilnik. Primer je prikazan spodaj:

```
int transfer_half1()
{//beremo 1. polovico BRAM
	for (i = 0; i < 360; i++)
	{
		number=Xil_In32(addr);

		ab = number & 0xff;
		cd = (number>>8) & 0xff;
		ef = (number>>16) & 0xff;
		gh = (number>>24) & 0xff;
		//pomikanje in maskiranje 32bit
		send_buf[i*4]=ab;
		send_buf[i*4+1]=cd;
		send_buf[i*4+2]=ef;
		send_buf[i*4+3]=gh;

		addr=addr+0x00000004;//povečamo naslov
	}
	Xil_Out32(0x41200000,0x00000003);//postavimo zastavico PL delu
	y=lwip_write(new_sd, send_buf, SEND_BUFSIZE);//pošljemo podatke
	return 1;//ko končamo, vrnemo vrednost 1
```

Pri programiranju glavne funkcije najprej ustvarimo *socket* ter določimo parametre povezave. Silverlight ima zelo ozek pas dovoljenih vrat za TCP-povezavo, izberemo npr. `address.sin_port=htons(4502)`. Po vzpostavitvi povezave čakamo v neskončni zanki na prvo zastavico iz PL-dela (1. del BRAM je poln). Pojav zastavice pomeni nastavitev naslova za začetek branja pomnilnika ter klic funkcije za prenos podatkov. Do signalnih zastavic, priključenih na AXI GPIO ter BRAM, dostopamo prek naslovov, ki jih najdemo v okolju Vivado pod zavihkom Addres Editor. Ko je pošiljanje končano, čakamo na pojav druge zastavice in postopek ponovimo za ostalo polovico pomnilnika. Če pride do napake, jo javimo na terminal, tako kot tudi ostala sporočila, ki so nam v pomoč predvsem pri razhroščevanju.

```
void tx_application_thread()
{	//ustvarimo socket
	if ((sock = lwip_socket(AF_INET, SOCK_STREAM, 0)) < 0) {
		xil_printf("error creating socket\r\n");
	#ifdef OS_IS_FREERTOS
		vTaskDelete(NULL);
	#endif
		return;
	}
	address.sin_family = AF_INET;
	address.sin_port = htons(4502);
	address.sin_addr.s_addr = INADDR_ANY;
	print("Waiting for client to connect...\n\r");
	if (lwip_bind(sock,(struct sockaddr *)&address, sizeof(address))< 0)
	{#ifdef OS_IS_FREERTOS
		vTaskDelete(NULL);
```

```
#endif
	return;}
//čakamo na povezavo
lwip_listen(sock, 0);

size = sizeof(remote);
new_sd = lwip_accept(sock, (struct sockaddr *)&remote,
(socklen_t*)&size);
//povezava vzpostavljena
print("Connected\n\r");

Xil_Out32(0x41200000,0x00000005);
print("Waiting to sample...\n\r");
while(Xil_In32(0x41200000)<0x01000000)
{}//čakamo na prvo zastavico
print("Sampling\n\r");

while(1)
{	PS_Flag=Xil_In32(0x41200000);
	if (PS_Flag==0x10000000)
	{//nastavitev naslova na branje 1. polovice
		addr=0x40000000;
		 if (transfer_half1()==1)
		 {//TCP paket poslan
			while (1)
			{
				PS_Flag=Xil_In32(0x41200000);
				if (PS_Flag==0x20000000)
				{
					if (transfer_half2()==1)
					{//TCP 2 paket poslan
						break;}
				}
				else if (PS_Flag==0x40000000)
				{
					 print("error");}
			}
		 }
	 else if (PS_Flag==0x40000000)
	 {	print("error");}}}
```

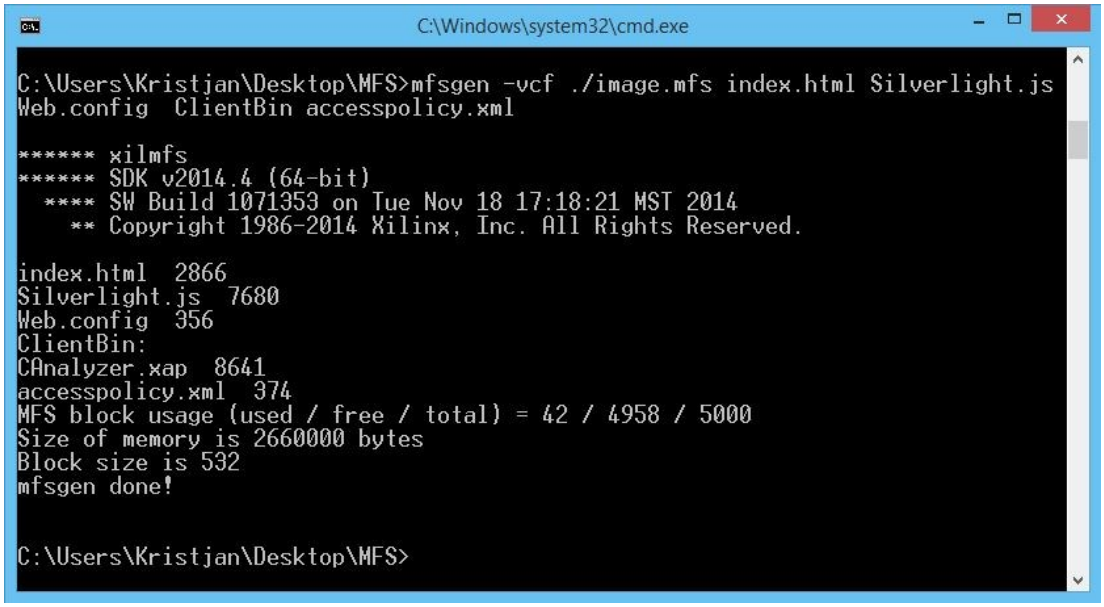
4.2 Datotečni sistem

Ker je naš analizator hkrati tudi spletni strežnik, se moramo dotakniti uporabe datotečnega sistema. FreeRTOS uporablja Xilinx Memory File System (MFS) [27], ki ga lahko tudi sami uporabimo pri razvoju morebitne samostojne aplikacije. Uvoziti moramo knjižnico *xilmfs*, kjer so nam na voljo različne funkcije za upravljanje z datotekami in mapami. Čeprav do te točke besedila še ni bilo razvidno, kakšen spletni vmesnik ponuja naš analizator, pokažimo, kako se pripravi potrebne datoteke za prenos na ploščico. Skupaj z MFS-knjižnico nam je na voljo orodje *mfsgen*, ki nam omogoča kreiranje MFS-spominske slike (ang. memory image) na

osebnem računalniku. Orodje zaženemo iz lupine v okolju SDK v meniju XilinxTools/LaunchShell. Potrebujemo datoteke:

- Index.html
- Silverlight.js
- Web.config
- accesspolicy.xml
- CAnalyzer.xap

Uporabimo (slika 4.2) ukaz s parametri -vcf:
mfsgen -vcf ./image.mfs index.html Silverlight.js Web.config ClientBin accesspolicy.xml



Slika 4.2: Uporaba orodja mfsgen

Ustvarjeno sliko *image.mfs* moramo glede na konfiguracijo v FreeRTOS namestiti na naslov *0x7200000*. To storimo tako, da odpremo nastavitve Run Configurations, kjer pod *Data Files to download before launch* izberemo željen *image.mfs* ter določimo naslov.

4.2.1 Omejitev dolžine imena

V enem izmed naslednjih poglavjih se bomo srečali z varnostno politiko vtičnika Silverlight. Ta zahteva pred vzpostavitvijo TCP-povezave od strežnika datoteko *clientaccesspolicy.xml*, ki vsebuje dovoljenje za sejo. Uporabljen datotečni

sistem ima omejitev glede najdaljše dolžine imena datoteke ali mape postavljeno na 21 znakov, kar pomeni, da zahtevane datoteke ne moremo zapisati v MFS. Problem zaobidemo z majhno predelavo spletnega strežnika.

Ideja je, da ko strežnik sprejme HTTP GET-zahtevo za *clientaccesspolicy.xml*, vrne klientu vsebino druge datoteke, ki ima samo ustrezno krajše ime. V datoteki *http_response.c* najdemo funkcijo *do_http_get(int sd, char *req, int rlen),* ki upravlja z HTTP GET-zahtevki. Funkcijo preuredimo tako, da zahtevano ime datoteke primerjamo z `"clientaccesspolicy.xml"`. Če sta vrednosti enaki, preprosto zamenjamo ime z `strcpy(filename, "accesspolicy.xml")`. Funkcija bo tako vrnila vsebino datoteke, ki smo jo lahko zapisali v *image.mfs*. Dodana koda je prikazana spodaj:

```
int compare;
/* determine file name */
extract_file_name(filename, req, rlen, MAX_FILENAME);

//primerjamo zahtevano datoteko z "clientaccesspolicy.xml"
compare=strcmp(filename, "clientaccesspolicy.xml");

if(compare==0)//imeni sta enaki
{//zamenjamo ime z datoteko v MFS
strcpy(filename, "accesspolicy.xml");
}
```

5 Microsoft Silverlight

Microsoft Silverlight danes postaja vse bolj opuščen vtičnik oziroma aplikacijski okvir (ang. application framework) z dokončnim prenehanjem podpore oktobra 2021. Z razliko od podobne tehnologije Adobe Flash vtičnika Silverlight ni mogoče uradno namestiti v brskalnike na Linux sistemih, prav tako ni podpore za operacijske sisteme na mobilnih napravah. Čeprav je tehnologija v prvi vrsti zaradi zelo dobre podpore za video aplikacije dosegla kar precejšnjo razširjenost in uporabo, jo vse bolj dozorel standard HTML 5 vztrajno izpodriva [28].

Kljub zgoraj naštetim dejstvom pa bomo za naš primer analizatorja uporabili zadnjo verzijo Silverlight 5. Glavni razlog je v prejšnjih poglavjih že omenjena podpora za TCP-protokol. Programiramo v objektnem jeziku *C#* v okolju Visual Studio, kjer ustvarimo nov Silverlight projekt.

5.1 Varnostna politika

Silverlight aplikacija teče znotraj t. i. sandboxa oziroma peskovnika uporabnikovega brskalnika. Ta izraz se uporablja za območje izvajanja kode, kjer je zaradi varnostnih razlogov dostop do datotečnega sistema ter ostalih sistemskih virov omejen ali celo onemogočen. Če želimo uporabljati TCP-vtičnice, moramo upoštevati sledeče omejitve oziroma zahteve [29, stran 892]:

- povezava je lahko vzpostavljena samo med vrati 4502-4534,
- strežnik z dovoljenjem za povezavo mora biti dostopen na vratih 943 in to na istem strežniku, kjer želimo vzpostaviti TCP-povezavo,
- dovoljenje, pridobljeno iz strežnika na vratih 943, se mora ujemati z zahtevami aplikacije, ki po njem sprašuje.

V *cilentaccesspolicy.xml* datoteki, ki jo zahteva metoda iz razreda *socket* za *cross-domain* dostop, definiramo parametre za povezavo na način, kot je prikazan spodaj [30, stran 895]:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from>
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <socket-resource port="4502-4534" protocol="tcp" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

- V elementu domain z vrednostjo "*" dovolimo povezavo vsem morebitnim Silverlight aplikacijam.
- Element grant-to vsebuje poti oziroma vrata, do katerih lahko dostopajo aplikacije, določene v domain, pod socket-resource je določen TCP-protokol.

Za potrebe razvoja lahko ustvarimo Policy Server [29, stran 896], strežnik ki nam bo serviral *cilentaccesspolicy.xml* datoteko. Knjižnica *.NET Framework*, na kateri bazira tudi Silverlight, za ta namen že vsebuje razred *PolicyServer*. Lahko pa datoteko naložimo kar v *image.mfs* in tako pri razvoju dostopamo kar do končnega mesta datoteke.

5.2 TCP-povezava

TCP.cs datoteko ustvarimo za namen vzpostavitve same TCP-komunikacije. Datoteki oziroma projektu ne pozabimo dodati knjižnice *System.Net.Sockets*. Deklariramo nov razred *SocketConnection*, v katerem bodo vse metode, ki se nanašajo na samo povezavo. Samih metod razreda *socket* in tudi tistih na novo ustvarjenih na tem mestu ni smiselno podrobno opisovati, saj skupaj tvorijo znan način za vzpostavitev TCP-komunikacije. Omenimo pa najpomembnejše nastavitve in dejstva, na katere moramo biti pozorni.

V na novo deklarirani metodi *SocketConnection* ustvarimo nov vtičnik s parametri:

```
socket=newSocket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
```

- prvi pomeni protokol IPv4, za IPv6 bi uporabili vrednost *InterNetworkV6*,
- Silverlight podpira samo *Stream* in *Unknown* tipa vtičnikov,
- kot tip protokola izberemo *TCP*, možnost imamo še *Unknown* in *Unspecified*.

Ko je vtičnik ustvarjen, mu moramo določiti argumente za samo povezavo. Izberemo IP razvojne ploščice *192.168.1.10* ter vrata *4502*, ki smo ju predhodno že določili pri konfiguraciji PS-dela. V odvisnosti od nahajanja *clientaccesspolicy.xml* datoteke nastavimo protokol dostopa. V našem primeru se nahaja na spletnem strežniku, zato nastavimo parameter na http.

```
SocketAsyncEventArgs args = new SocketAsyncEventArgs()
{
      RemoteEndPoint = new DnsEndPoint("192.168.1.10", 4502),
      SocketClientAccessPolicyProtocol = SocketClientAccessPolicyProtocol.Http
};
```

Zaradi razvrščanja podatkov je najbolje, da deklariramo medpomnilnik iste globine kot na PS-strani (1440), ki bo ob spremembi prožil dogodek. Spomnimo, širina je že določena z metodo *SetBuffer*, ki prebrane podatke iz vtičnika posreduje v medpomnilnik z 8-bitno širino.

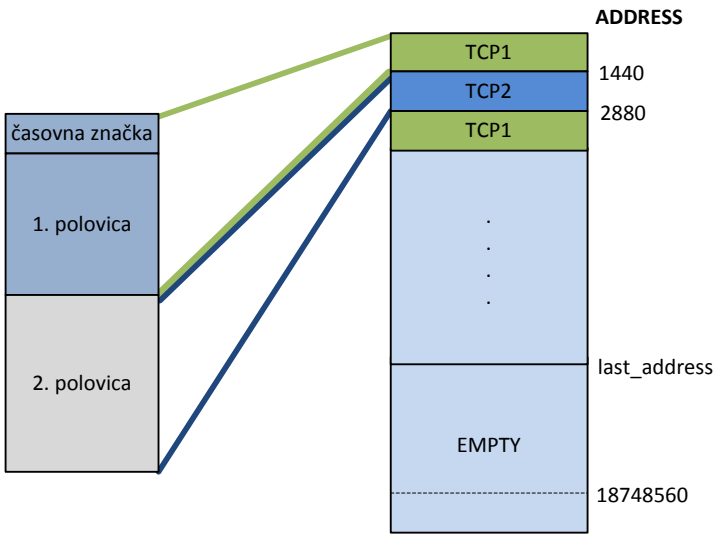
5.3 Razvrščanje podatkov in pisanje v pomnilnik

V korenskem razredu *MainPage* moramo urediti shranjevanje podatkov v končni pomnilnik ter postaviti grafični vmesnik za njihov prikaz. Končni pomnilnik deklariramo velikosti `datamemory = new byte[18750000]`, kar zadošča za slabe pol minute vzorčenja pri 5 MHz brez prekinitve ob neaktivnosti na liniji upoštevajoč prostor, kjer je zapisana časovna značka.

Ob prihodu posameznega TCP-paketa (velikost 1440) se proži dogodek in s tem požene metoda, ki pakete v prihajajočem vrstnem redu piše na naslov *address* v končni pomnilnik. *Address* se povečuje ob vsakem vpisu, dokler pomnilnik ni poln in je vzorčenja konec. Metoda ob prvem prihodu celotnega okvirja podatkov s časovno značko pokliče drugo metodo za risanje. Prav tako ob istem dogodku povečujemo števec, ki nosi vrednost o zadnjem naslovu v pomnilniku s podatki.

```
void OnPointReceived(object sender, DataEventArgs e)
        {
            if (address < 18748560)
            {
                Buffer.BlockCopy(e.data, 0, datamemory, address, 1440);
                address = address + 1440;
                if (parity_flag == 1)
                {
                    last_address = last_address + 2880;
                    parity_flag = 0;
                }
                else
                { //celoten okvir s časovno značko pride v dveh TCP paketih
                    parity_flag = 1;
                }
            }
            else
            {
                this.VisualConsole.Text = "Full Memory, Sampling is over";
            }
            We++;
            if (We == 3)    //šele v drugo pokličemo metodo za risanje, ker
            {               //rišemo 2x 1440
                Dispatcher.BeginInvoke(new Action(() =>
                {
                    DrawSignal();
                    this.VisualConsole.Text = "Sampling at 5MHz...";
                }));}}
```

Na spodnji sliki 5.1 vidimo razvrščanje paketov v pomnilnik brskalnika na uporabnikovi strani. Paketi se vpisujejo zapovrstjo na večkratnik naslova 1440. Takšno vpisovanje potrebujemo zaradi izrisa, saj moramo natančno vedeti, kje v pomnilniku se nahaja časovna značka in do kje so podatki, ki se nanašajo nanjo.



Slika 5.1: Razvrščanje paketov v pomnilnik brskalnika

5.4 Grafični vmesnik

Za prikaz koncepta našega analizatorja lahko razvoj zahtevnejšega grafičnega vmesnika v smislu raznolike manipulacije s podatki pustimo ob strani. Želimo si enostaven prikaz vzorčenih podatkov oziroma signala na časovni osi, zagon in prekinitev vzorčenja ter možnost enostavne meritve časovne razlike med dvema označenima vzorcema.

Postavitev elementov v Silverlightu opišemo v XAML (ang. Extensible Application Markup Language) datoteki *MainPage.xml*. V korenski element *UserControl* dodamo platno *Canvas*, v katerem lahko vstavljenim elementom določimo absolutno pozicijo. Platno se uporablja tudi za risanje, zato dodamo še en element te vrste, ki ga ugnezdimo v element *ScrollViewer*, saj bo prikazan signal širši od okna brskalnika [29, stran 65]:

```
<UserControl x:Class="CAnalyzer.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

    <Canvas x:Name="LayoutRoot" Height="600" Width="1438"
    RenderTransformOrigin="0.507,0.622">
        <ScrollViewer HorizontalScrollBarVisibility="Visible"
                      VerticalScrollBarVisibility="Hidden" x:Name="sve"
                      Height="200" Width="1400" Canvas.Top="50">
            <Canvas
                x:Name="DrawingCanvas"
                Width="115050"
                Height="180"
                MouseLeftButtonUp="CanvasLeftClick"
                Cursor="Hand">
            </Canvas>
        </ScrollViewer>
        <TextBox VerticalAlignment="Top"  Name="dtconsole" FontSize="14"
                 Text="..." Width="160" Height="30" TextWrapping="Wrap"
                 Canvas.Top="420" Canvas.Left="965"/>

        <ScrollViewer Height="120" HorizontalAlignment="Left"
                      Canvas.Top="330" Canvas.Left="300" Width="430"
                      Name="scrollViewer1" VerticalAlignment="Top" >
            <TextBox VerticalAlignment="Top"
                     Name="VisualConsole" FontSize="14"
                     Text="..." Width="430" Height="120"/>
        </ScrollViewer>

        <Button ClickMode="Press" x:Name="btnTest"  Height="30"
                Width="70" Click="return_to_0" Content="Go to first"
                Canvas.Top="270" Canvas.Left="225">

        </Button>
```

Za prikaz časovne razlike med vzorcema ter sporočila o delovanju analizatorja ustvarimo dva *TextBox* elementa. Vanju bomo iz programske kode pisali s spremembo atributa *Text*. Vmesniku dodamo še 12 gumbov, vsakemu pa pod atributom *Click* določimo ime metode, ki se bo izvedla s proženjem dogodka oziroma klika, primer enega izmed gumbov najdemo v zgornji kodi.

5.4.1 Risanje signala

V razredu *MainPage* ustvarimo metodo *DrawSignal*, ki se bo izvedla po prejemu prvih dveh TCP paketov, torej ko bo v končni pomnilnik vpisanih 2880 bajtov ali s pritiskom na gumb. Za risanje bomo uporabili črte oziroma elemente *Line*, ki za postavitev na platno potrebujejo dve koordinati oziroma točki. Deklariramo ju na začetku izvajanja metode poleg ostalih atributov, kot je npr. barva.

Risanje signala postavimo v zanko, kjer se sprehodimo po celotnem okvirju oziroma podatkih dveh TCP-paketov. To je najmanjša enota, ki je lahko poslana iz PL oziroma PS-dela in hkrati ni preveč velika za zahtevnost izrisa v vtičniku. Veliko število vzorcev za izris zelo upočasni odzivnost brskalnika. Preskočimo prve štiri naslove in prvih 8 bitov shranimo v delovno spremenljivko, nad katero delamo *and* (&) operacijo z masko. Vrednost maske vsake iteracije spreminjamo s pomikanjem v desno. Tako dobimo binarne vrednosti posameznega vzorca. Za izris moramo vsakič poznati vrednost prejšnjega vzorca, saj je od tega odvisno, kako bomo sestavili signal. Če je nova vrednost enaka, se bo naslednja črta nadaljevala v isti liniji, kar pomeni spremembo samo koordinat X. Če je prišlo do spremembe, najprej narišemo pokončno črto in zatem še vodoravni del z ustrezno vrednostjo Y-koordinate.

```
public void DrawSignal()
{
int points_counter = 0;
int r = 0;
int mask = 1;
int result;
byte work_cache = 0x00;
//točke deklaracija
System.Windows.Point OldPoint = new System.Windows.Point(points_counter, 110);
System.Windows.Point NewPoint = new System.Windows.Point(points_counter, 110);
System.Windows.Point TextPoint = new System.Windows.Point(0, 140);
//barve deklaracija
SolidColorBrush redBrush = new SolidColorBrush();
redBrush.Color = Colors.Red;
SolidColorBrush grayBrush = new SolidColorBrush();
grayBrush.Color = Colors.Gray;
SolidColorBrush blackBrush = new SolidColorBrush();
blackBrush.Color = Colors.Black;
```

```
//vsakič resetiramo canvas, scrollbar pozicijo, timer_display, timer
this.DrawingCanvas.Children.Clear();
this.sve.ScrollToHorizontalOffset(0);
timer_display = 0;
timer = 0;
for (int q = 0; q < 4; q++)
{//iz prvih 32bitov okvirja, izračunamo naslov za prikaz
timer_display += (datamemory[displayaddress + q]) << (8 * (q));
}
//časovno značko okvirja zapišemo v novo spremenljivko
timer_display_global = timer_display;
//prikazujemo številni časovni odčitek vsakih 50 bitov
timer_display = timer_display / 10;

for (int i = 4; i < 2880; i++)//preskočimo naslov s časovno značko
{
work_cache = datamemory[displayaddress + i];
mask = 1;//ob vsaki iteraciji ponastavimo masko na 1

for (int y = 0; y < 8; y++)
{//na vsakem naslovu naredimo & z masko nad vsakim bitom
timer++;//vsakič povečamo timer
result = work_cache & mask;

if (result > 0)
    if (r > 0)
    {// visok nivo -> visok nivo
        points_counter = points_counter + 5;//conter + 5pixel
        NewPoint.X = points_counter;
        NewPoint.Y = 50;
        var line = new Line { Fill = new SolidColorBrush(Colors.Red) };
        line.X1 = OldPoint.X;
        line.Y1 = OldPoint.Y;
        line.X2 = NewPoint.X;
        line.Y2 = NewPoint.Y;

        // določitev širine in barve
        line.StrokeThickness = 2;//določanje debeline
        line.Stroke = redBrush;
        DrawingCanvas.Children.Add(line);//dodajanje črte v canvas
        OldPoint = NewPoint;
    }
    else
    {//nizek nivo -> visok nivo
        NewPoint.X = points_counter;
        NewPoint.Y = 50;
        var line = new Line { Fill = new SolidColorBrush(Colors.Red) };
        line.X1 = OldPoint.X;
        line.Y1 = OldPoint.Y;
        line.X2 = NewPoint.X;
        line.Y2 = NewPoint.Y;

        line.StrokeThickness = 2;
        line.Stroke = redBrush;
        DrawingCanvas.Children.Add(line);
        OldPoint = NewPoint;
        //v tem primeru rišemo dve črti, vodoravno in navpično
        points_counter = points_counter + 5;
        NewPoint.X = points_counter;
        var line2 = new Line { Fill = new SolidColorBrush(Colors.Red) };
```

```
        line2.X1 = OldPoint.X;
        line2.Y1 = OldPoint.Y;
        line2.X2 = NewPoint.X;
        line2.Y2 = NewPoint.Y;
        //deklarirati moramo novo line2
        line2.StrokeThickness = 2;
        line2.Stroke = redBrush;
        DrawingCanvas.Children.Add(line2);
        OldPoint = NewPoint;
    }
else
    if (r > 0)
         { ...
```

Za prikaz časa na X-osi je potrebno pridobiti časovno značko s prvih štirih naslovov želenega okvirja. Vrednosti z naslovov z nekaj manipulacije združimo in shranimo v globalni spremenljivki *timer_display_global* ter *timer_display*. Sicer nam ni težko določiti posamezne vrednosti vsakega vzorca, saj so shranjeni zapovrstjo. Tak prikaz bi bil zelo razpotegnjen, hkrati pa ne bi imel velikega pomena, saj bi videli preozek del signala. Deljenje spremenljivke *timer_display* z vrednostjo *10* daje 10 µs natančnost oziroma prikaz značke vsakih 50 vzorcev. V ta namen pri določitvi vrednosti vzorca povečujemo števec *timer* in ob doseženi vrednosti *50* izrišemo časovno značko.

```
...r = result;//shraniti moramo predhodni rezltat
mask = mask << 1;//pomik maske

if (timer == 50) //narišemo timeline in grid za vsakih 10us
{
    timer = 0;//resetiramo timer
    timer_display += 10; //povecamo za 10us

    var textb = new TextBlock { Text = "no time" };
    textb.Text = Convert.ToString(timer_display) + " µs";
    TextPoint.X = NewPoint.X;
    textb.SetValue(Canvas.LeftProperty, TextPoint.X);
    textb.SetValue(Canvas.TopProperty, TextPoint.Y);
    DrawingCanvas.Children.Add(textb);

    var line = new Line { Fill = new SolidColorBrush(Colors.Black) };
    line.X1 = NewPoint.X;
    line.Y1 = 20;
    line.X2 = NewPoint.X;
    line.Y2 = 130;
    line.StrokeThickness = 1;
    line.Stroke = blackBrush;

    DrawingCanvas.Children.Add(line);
}
else
{//drugače rišemo krajše pokončne črtice, za vsakih 0,2us
    var line = new Line { Fill = new SolidColorBrush(Colors.Gray) };
```

```
    line.X1 = NewPoint.X;
    line.Y1 = 115; //krajša črta, večji y
    line.X2 = NewPoint.X;
    line.Y2 = 130;
    line.StrokeThickness = 1;
    line.Stroke = grayBrush;

    DrawingCanvas.Children.Add(line);
}
```

5.4.2 Premik do naslednje logične vrednosti '0'

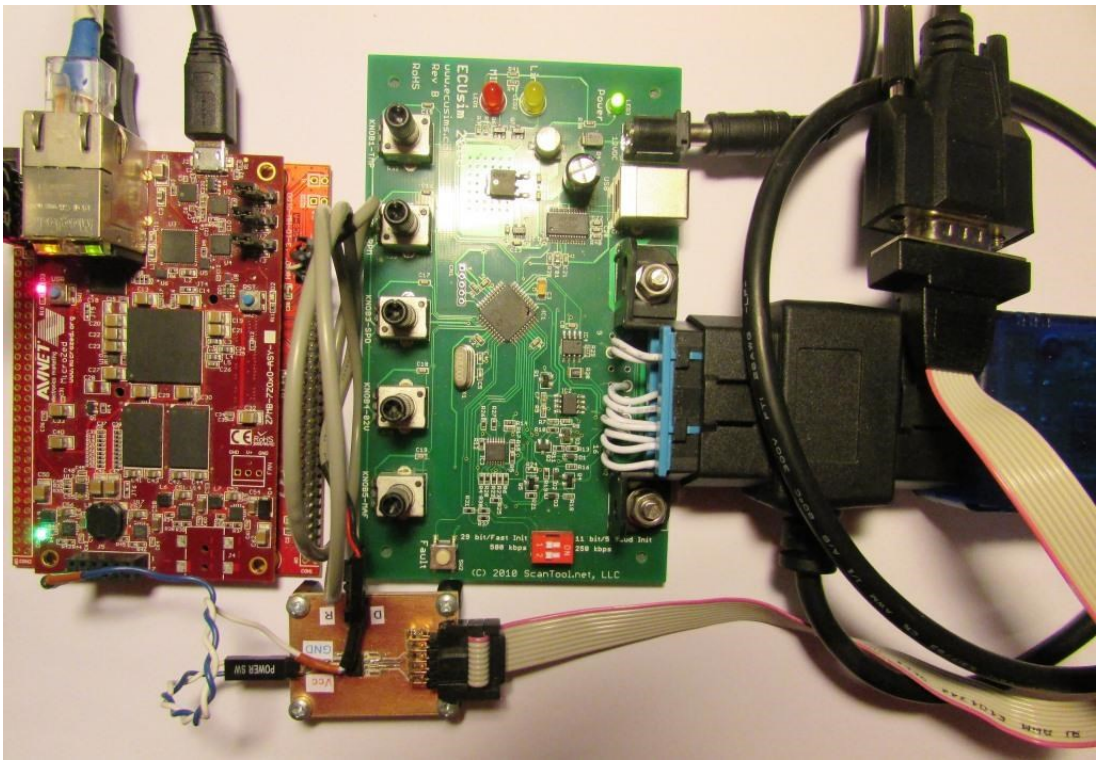
Z vgrajenim gumbom za premik znotraj izrisanega okvirja do naslednjega vzorca z vrednostjo *'0'* uporabniku omogočimo lažje iskanje začetka nekega CAN-okvirja ter hitrejše iskanje uporabnih podatkov na platnu. V metodi *edge_right* najprej prek pozicije drsnika izračunamo, kateri del okvirja uporabnik gleda. To pomeni določitev naslova v pomnilniku, od koder začnemo iskati naslednjo ničelno vrednost. Naslov, na katerem se najdena vrednost nahaja, pretvorimo v novo pozicijo drsnika.

5.4.3 Meritev časa med vzorcema

Za meritev časa pripravimo dva gumba, s katerima lahko uporabnik omogoči dva kazalca (ang. cursors). Ob kliku na platno se kazalec izriše in iz njegove koordinate se izračuna ekvivalenta časovna značka vzorca. V kombinaciji z drugim kazalcem se njuni vrednosti odštejeta in prikaže se absolutna vrednost razlike.

6 Testiranje in sklepna ugotovitev

Naš prototip logičnega analizatorja (slika 6.1) je na tem mestu pripravljen za prva testiranja in preizkus uporabe.

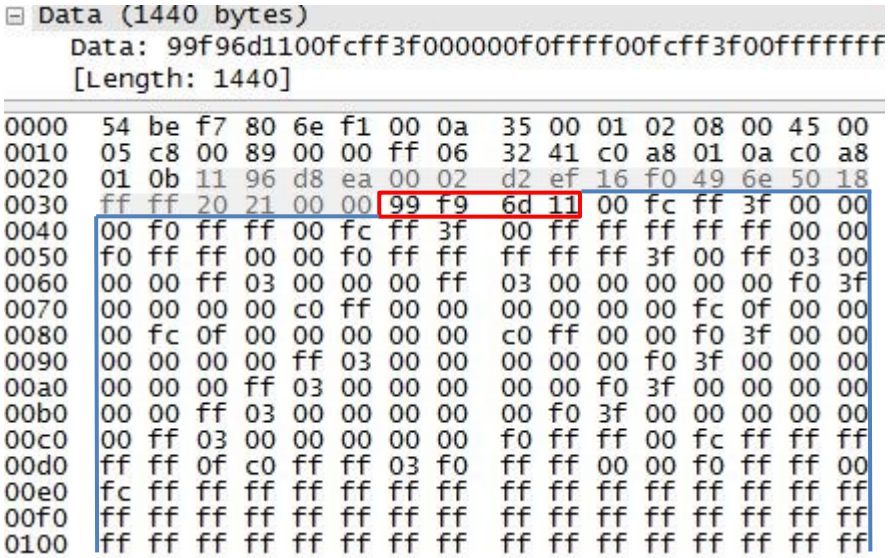


Slika 6.1: Fotografija sistema

V primeru, da smo pustili DHCP izklopljen, moramo za priklop na osebni računalnik preko Ethernet povezave v nastavitvah mrežne kartice nastaviti fiksne vrednosti IP. Preden logični analizator dejansko preizkusimo na CAN-emulatorju bi se bilo smiselno prepričati o pravilnem pošiljanju podatkov preko TCP.

6.1 Paketna analiza

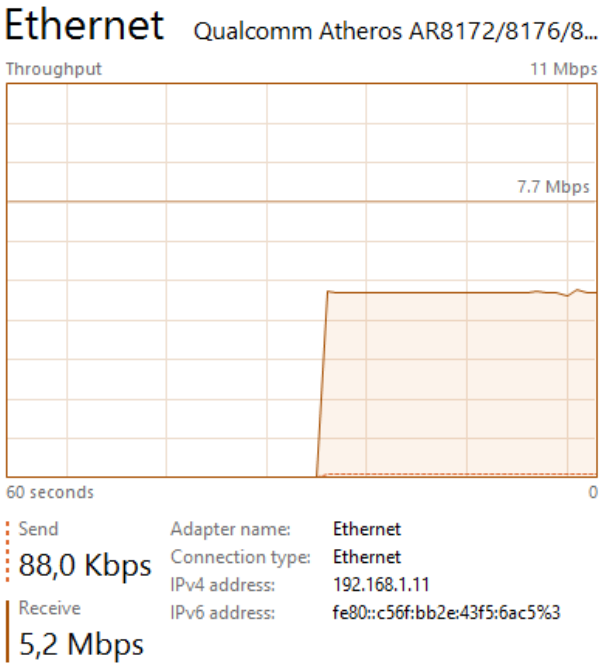
Same TCP-pakete lahko opazujemo z omrežnim analizatorjem WireShark, ki nam že sam izlušči podatke iz paketa, na nas je, da ločimo časovno značko (rdeč okvir na sliki 6.2) od ostalih podatkov.



Slika 6.2: Paketna analiza v programu WireShark

6.2 Največja frekvenca vzorčenja

Največja frekvenca vzorčenja našega analizatorja, pri kateri bo zmogel vzorčene podatke še vedno sproti poslati na drugo stran Ethernet povezave, je okrog 12 MHz. Največje ozko grlo za to omejitev je zamudno prepisovanje podatkov iz pomnilnika BRAM v TCP-medpomnilnik, ki je, kot je bilo že omenjeno, širok samo 8 bitov. Tu je možno optimizacija tudi v zvezi z balansiranjem prioritet med spletnim in TCP-strežnikom. Slika 6.3 prikazuje graf prenosa podatkov za vzorčenje pri 5 MHz.



Slika 6.3: Prenos podatkov pri vzorčenju 5 MHz

6.3 Prikaz delovanja

Na sliki 6.4 je prikazan uporabniški vmesnik z vzorčenim digitalnim signalom. Za izračun časovne razlike sta uporabljena kazalca, sama vrednost pa je izpisana v manjšem izmed oken. Kazalca je možno tudi onemogočiti. Poleg zagona in zaustavitve imamo z gumbi možnost premika po končnem pomnilniku naprej in nazaj, bodisi za mesto ali pet. Prav tako se lahko neposredno premaknemo na zadnji ali začetni naslov. Informacije o delovanju so izpisane v večjem oknu.

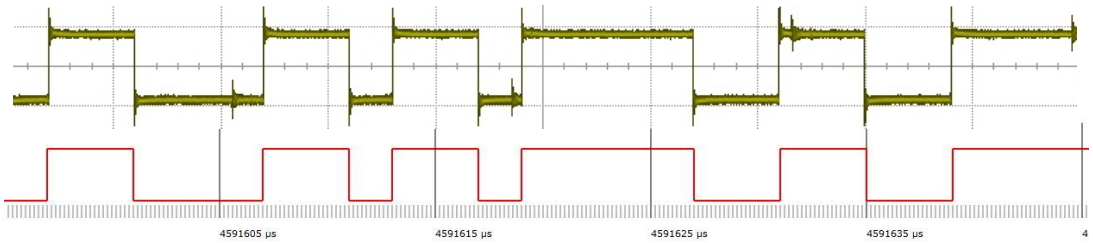
Slika 6.4: Spletni vmesnik

6.4 Sklepna ugotovitev

Načinov za nadgradnjo naprave je kar nekaj. Predvsem je veliko neizkoriščenega prostora pri paralelnem zajemu širših vodil ali signalov, saj nam na tem mestu programirljiva logika FPGA omogoča veliko zmogljivost in prilagodljivost, skupaj s PS-delom pa zagotavlja hitro paralelno procesiranje vzorcev.

Možna bi bila uporaba enega izmed operacijskih sistemov Linux. Tu bi npr. s strežnikom PHP (ang. Hypertext Preprocessor) in drugimi sorodnimi tehnologijami izboljšali uporabniško izkušnjo.

Prikazan razvoj koncepta logičnega analizatorja predstavlja samo enega izmed načinov, kako osnovati napravo za vzorčenje nekega digitalnega signala, ki je kot samostojen člen povezan v omrežje. Delovanje naprave daje pričakovane in zadovoljive rezultate. Lahko ga ovrednotimo s primerjavo vzorcev istega okvirja, ki smo jih zajeli z osciloskopom. Primerjavo prikazuje slika 6.5. Na tem mestu razvoja je naslednji korak implementacija dekodiranja CAN-okvirja.



Slika 6.5: Primerjava med vzorci analizatorja in zajemom z osciloskopa

Literatura

[1] Tektronix, "TLA7000 Series Data Sheet [Online]." Dosegljivo: http://www.tek.com/sites/tek.com/files/media/media/resources/TLA7000-Logic-Analyzer-Datasheet-18_0.pdf. Dostopano: [20. 1. 2015].

[2] Agilent, "Agilent 16800 Series Portable Logic Analyzers [Online]." Dosegljivo: http://cp.literature.agilent.com/litweb/pdf/5989-5063EN.pdf. Dostopano: [20. 1. 2015].

[3] *Logic analyzer* [Online]. Dosegljivo: https://en.wikipedia.org/?title=Logic_analyzer. Dostopano: [20. 1. 2015].

[4] Freescale Semiconductor, "In-Vehicle Networking [Online]." Dosegljivo: http://cache.freescale.com/files/microcontrollers/doc/brochure/BRINVEHICLENET.pdf. Dostopano: [21. 1. 2015].

[5] Volkswagen VK-36 Service Training, "European On-Board Diagnosis for Diesel Engines [Online]." Dosegljivo: http://www.volkspage.net/technik/ssp/ssp/SSP_31 5.PDF. Dostopano: [21. 1. 2015].

[6] "Direktiva 98/69/ES Evropskega parlamenta in Sveta z dne 13. oktobra 1998 o ukrepih proti onesnaževanju zraka z emisijami iz motornih vozil in o spremembah Direktive Sveta 70/220/EGS [Online]." Dosegljivo: http://eur-lex.europa.eu/legalcontent/SL/TXT/HTML/?uri=CELEX:31998L0069&from=EN. Dostopano: [15. 5. 2015].

[7] D. Paret, "Multiplexed Networks for Embedded Systems." Chichester: John Wiley & Sons, 2007.

[8] Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signalling, ISO 11898-1, 2003.

[9] Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit, ISO 11898-2, 2003.

[10] Road vehicles - Controller area network (CAN) - Part 3: Low-speed, fault-tolerant, medium-dependent interface, ISO 11898-3, 2006.

[11] Road vehicles - Controller area network (CAN) - Part 4: Time-triggered communication, ISO 11898-4, 2004.

[12] Road vehicles - Controller area network (CAN) - Part 5: High-speed medium access unit with low-power mode, ISO 11898-5, 2007.

[13] S. Corrigan, "Introduction to the Controller Area Network (CAN) [Online]." Texas Instruments: Dallas, Texas, SLOA101A, 2008. Dosegljivo: http://www .ti.com/lit/an/sloa101a/sloa101a.pdf. Dostopano: [15. 1. 2015].

[14] OBD Solutions, "ECUsim 2000 User Guide." OBD Solutions, 2013.

[15] *OnBoardDiagnostics*. *OBD-II NETWORK STANDARDS* [Online]. Dosegljivo: http://www.onboarddiagnostics.com/page03.htm. Dostopano: [15. 1. 2015].

[16] Road vehicles - Diagnostic communication over Controller Area Network (DoCAN) - Part 4: Requirements for emissions-related systems, ISO 15765-4, 2011.

[17] *Microsemi SoC FPGAs* [Online]. Dosegljivo: http://www.microsemi.com/ products/fpga-soc/soc-fpgas. Dostopano: [21. 5. 2015].

[18] *Altera SoCs Portfolio* [Online]. Dosegljivo: https://www.altera.com/products/ soc/portfolio .html. Dostopano: [21. 5. 2015].

[19] L. H. Crockett et al., "The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC." Strathclyde Academic Media, 2014.

[20] *ZedBoard, Microzed* [Online]. Dosegljivo: http://zedboard.org/product/ microzed. Dostopano: [21. 2. 2015].

[21] Texas Instruments, "3.3-V CAN TRANSCEIVERS [Online]." Dosegljivo: http:// www.farnell.com/datasheets/1864416.pdf. Dostopano: [15. 1. 2015].

[22] *Xilinx* [Online]. Dosegljivo: http://www.xilinx.com/. Dostopano: [21. 2. 2015].

[23] A. Trost, "Uvodna delavnica Xilinx Zynq in Vivado." Ljubljana: 2013.

[24] A. Trost, "Načrtovanje digitalnih vezij v jeziku VHDL." Ljubljana: Založba FE in FRI, 2011.

[25] *Xilinx Zynq FreeRTOS and lwIP demo (XAPP1026) Vivado 2014.2* [Online]. Dosegljivo: http://interactive.freertos.org/entries/31659559-Xilinx-ZynqFree RT OS-and-lwIP-demo-XAPP1026-Vivado-2014-2. [Dostopano: 15. 3. 2015].

[26] R. Barry, "Using the FreeRTOS real time kernel." 2009.

[27] Xilinx, "Xilinx Memory File System (MFS) [Online]." Dosegljivo: http://www.xilinx.com/ise/embedded/edk82i_docs/sa_xilmfs_v1_00_a.pdf. Dostopano: [15. 1. 2015].

[28] *Microsoft Silverlight* [Online]. Dosegljivo: https://www.microsoft.com/silverlight. [Dostopano: 19. 3. 2015].

[29] M. MacDonald, "Pro Silverlight 5 in C#." New York: Springer Science, 2012.