UNIVERZA V LJUBLJANI **Fakulteta za elektrotehniko**
UNIVERSITY OF LJUBLJANA **Faculty of Electrical Engineering**
Tržaška 25
1001 Ljubljana, p.p. 2999
Slovenia
Tel.: +386 61 17 68 411
+386 61 12 64 620
Fax: +386 61 12 64 630

Št.: M-830/2000
Datum: 4. 05. 2000

Uroš B I Z J A K, univ. dipl. inž. el.

LJUBLJANA

Fakulteta za elektrotehniko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **INTEGRIRANI NADZORNI IN KRMILNI MIKROSISTEM**

Tematika naloge:
Načrtajte nadzorni in krmilni integrirani mikrosistem za nadzor zajema in obdelave podatkov v realnem času. Analizirajte sistemske procese in na osnovi analize izdelajte sistem s komponentami izvedenimi s strojno in programsko opremo. Izvedba sistema naj bo optimizirana na velikost končne geometrije integriranega vezja.

Mentor:

prof. dr. Baldomir ZAJC

Fakulteta za elektrotehniko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Tadej BAJD

UNIVERZA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

Uroš Bizjak

Integrirani nadzorni in krmilni mikrosistem

Magistrska naloga

Ljubljana, junij 2000

Povzetek

Namen magistrske naloge je izdelava nadzornega in krmilnega vezja vključenega mikrosistema. Sistem bo izveden v hibridni softversko-hardverski arhitekturi, sestavljen pa bo iz programabilnega mikroprocesorskega jedra in optimiziranih hardverskih podsklopov, s katerimi razbremenimo delovanje mikroprocesorja.

V prvem delu naloge je predstavljen sistemski model, sledi pa analiza sistemskih procesov s pomočjo data-flow grafov. Glede na omejitve, ki jih postavlja delovanje procesov v realnem času, funkcionalnost sistema razdelimo na softverske in hardverske komponente.

Rezultati sistemske analize in simulacij procesov nam služijo kot vodilo pri načrtovanju posameznih komponent; izvedene optimizirane hardverske podsklope z nizkonivojskimi V/I gonilnimi rutinami povežemo v funkcionalno celoto. Želimo, da je celoten sistem izveden na kar najmanjši končni geometriji, delovanje V/I rutin pa naj kar najmanj obremenjuje integrirano mikrojedro.

Funkcionalnost celotnega sistema je združena na enem integriranem vezju, hardverski del je načrtanim z uporabo načrtovalskega paketa Cadence/Verilog, softverske rutine pa so bile razvite s pomočjo GNU Compiler Collection paketa in prevedene z razvojnim sistemom winuSim za MTC-8308 mikrojedro.

Abstract

The purpose of masters thesis is a design of supervisory and control chip for an embedded system. The chip consists of programmable microprocessor core and optimized hardware subsystems to lower on-chip microprocessor utilization.

System model of chip is presented in first section, followed by process analysis using data-flow graphs. Regarding real-time process constraints, functionality of system is divided into co-designed hardware and software part.

Results of system analysis and process simulation will be our guide to system components design. Optimized hardware blocks, designed to be driven by lightweight low-level software I/O routines, should achieve our goal of system, realized on smallest layout possible.

Integrated circuit, on which functionality of the system is realized, is designed with Cadence/Verilog design environment. Software routines were designed using GNU Compiler Collection package and compiled with winuSim development tool for MTC-8308 microcore.

Kazalo

Kazalo slik

Kazalo tabel

1. Uvod

Zadnja leta smo priče precejšnjemu povečanju uporabe digitalnih sistemov na različnih področjih. V sistemih za procesno vodenje, merjenje in nadziranje električnih veličin, komunikacijskih sistemih in napravah za široko potrošnjo, se elektronski in velikokrat tudi elektro-mehanski sklopi nadomeščajo s programabilnimi podsklopi, sestavljenimi iz mikroprocesorja in potrebne periferije. Programabilni podsklop je navadno vključen v večje - in mnogokrat ne-elektronsko okolje, zato takšne sisteme opisujemo kot vključene, "embedded" sisteme.

Takšni sistemi so prikrojeni posebnim zahtevam aplikacije, zato sta porabljeni čas in stroški za načrtovanje sistema precej višji v primerjavi s stroški za razvoj istega sistema s splošnonamenskimi komponentami. Vendar so stroški izdelave vključenega sistema manjši, zato velike serije upravičijo višji začetni vložek. Vključeni sistemi so navadno načrtani za delovanje v realnem času (real-time embedded systems), ovire, ki jih postavlja tak način delovanja, pa so velikokrat rešljive le z uporabo specialnih integriranih podsklopov, ter posebnih programov za splošno-namenski mikroprocesor.

Razvoj elektronskih komponent je v veliki meri omogočil povečanje uporabe programabilnih elementov. Mikroprocesorji in ostali programabilni elementi vsakih 18 mesecev podvojijo svoje zmogljivosti, s hitrim tempom pa se veča tudi velikost razpoložljivega pomnilnika. Hkrati se razvijajo tudi programska orodja; namesto mučnega kodiranja algoritmov v zbirnem jeziku, lahko splošno-namenski procesor vključenega sistema programiramo z uporabo višjih programskih jezikov. Čeprav so kritični deli programske opreme še vedno napisani v zbirniku, pa so programi, napisani z uporabo višjih programskih jezikov, precej bolj zanesljivi.

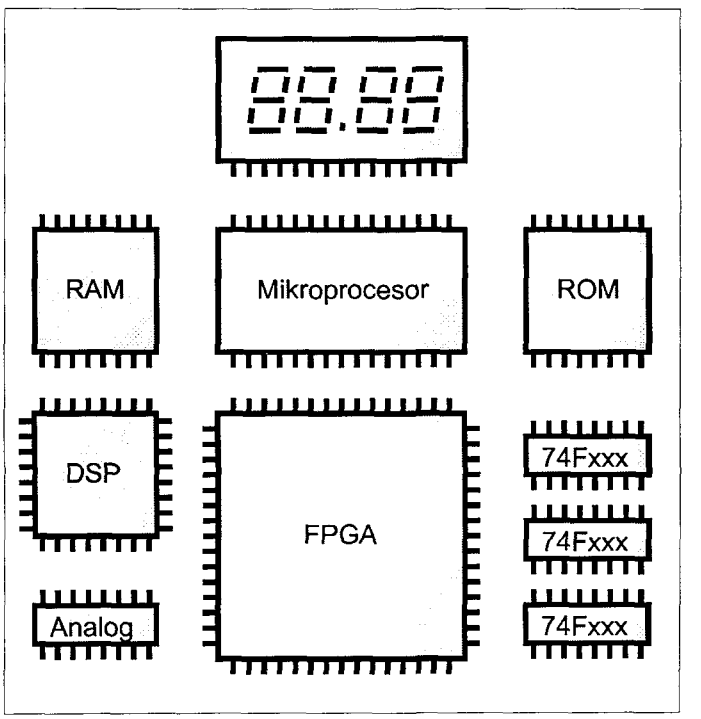
Razvita so bila orodja za sintezo sistema iz opisa njegovega delovanja (VHDL, verilog), posebej uspešno pa so se izkazala pri načrtovanju integriranih vezij. Kot alternativo sintezi iz funkcionalnega opisa lahko uporabimo direktno sintezo integriranega vezja iz programskega prototipa.

Z razvojem na področju procesiranja silicijeve rezine je postalo možno, da vse komponente vključenega sistema združimo v eno samo integrirano vezje. Takšno integrirano vezje predstavlja zaključen sistem v malem, vsebuje pa tako procesor, kot več tipov pomnilnika, vhodno-izhodne enote, uporabniško narejene digitalne in analogne integrirane podsklope, ne nazadnje pa vsebuje tudi kompletno sistemsko programsko opremo.

2. Tradicionalni vključeni sistemi

Vključeni sistemi so sestavljeni iz množice komponent, zgrajenih okrog ene ali več programabilnih komponent (mikroprocesor, digitalni signalni procesor), vsaka programabilna komponenta pa za delovanje potrebuje enega ali več različnih tipov pomnilnika - običajno statični RAM pomnilnik, za večje količine tudi dinamični RAM z ustreznim krmilnim vezjem.

Programska oprema vključenega sistema je vpisana v različne tipe bralnih pomnilnikov, za velike serije izbiramo med mask-programmable ROM pomnilnikom, za prototipne ali manjše serije pa izberemo EPROM, oziroma EEPROM pomnilnik. EEPROM pomnilnik uporabljamo predvsem tam, kjer moramo programsko opremo prilagajati potrebam okolja, največkrat pa tudi pri razvoju in razhroščevanju programske opreme.



Slika 1: Tradicionalni vključeni sistem

Posebne funkcije vključenega sistema lahko izvedemo s pomočjo programabilnih logičnih vezij (PLA, PAL), večje podsisteme pa izvedemo s FPGA vezji. Takšna vezja pa zahtevajo zagonske PROM pomnilnike, v katere vpišemo binarno kodo z opisom strukture vezja, lahko pa binarno kodo s pomočjo mikroprocesorja naložimo ob zagonu sistema.

Vključeni sistemi komunicirajo s svojo okolico preko analognih ali digitalnih pretvornikov:

- z analogno-digitalnimi pretvorniki z ustreznimi senzorji in prilagojevalniki signalov
- preko serijskih in paralelnih vodil z različnimi protokoli

Stanje vkjlučenega sistema lahko nadziramo na prikazovalniku, običajno je to ena izmed izvedb LCD prikazovalnika. Za zmanjšanje števila priključnih pinov gonilnega vezja sodobni matrični LCD prikazovalniki multipleksirajo signale po principu vrstica/stolpec. Takšen način pa zahteva, da gonilno vezje generira analogne signale takšne oblike, kot zahteva izbrana shema multipleksiranja.

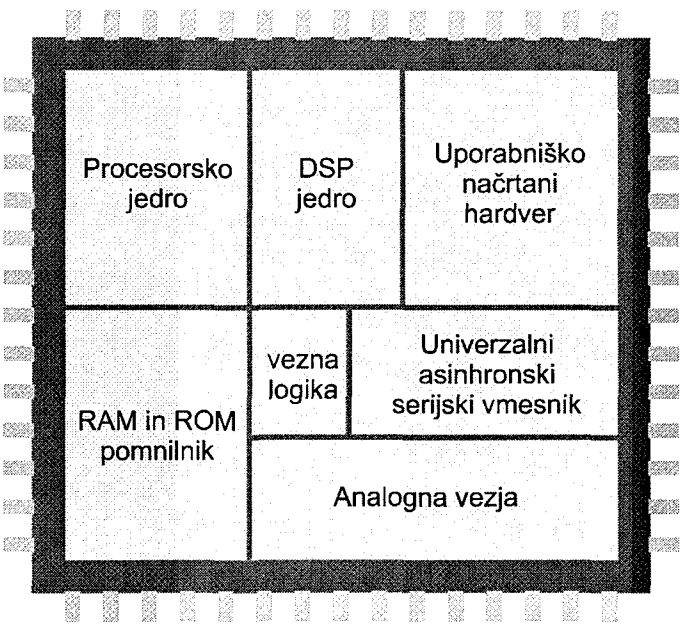
Vključeni sistemi delujejo brez človeškega nadzornika, zato mikroprocesor za zanesljivo delovanje potrebuje nadzorna vezja: watchdog vezje, generator reset signala, ter običajno še katero od izvedb RAM pomnilnika, ki ohrani vsebino po prekinitvi napajanja, v katerega mikroprocesor ob prekinitvi napajanja shrani ključne podatke.

Posamezni podsklopi so povezani s pomočjo vezne ("glue") logike, sestavljene iz programabilnih logičnih vezij, ali pa kar iz diskretnih vezji nizke integracije (npr. vezja serije 74xxx).

Množica vezij, iz katerih je sestavljen sodoben vključen sistem pa se ne sklada več s prostorskimi omejitvami, majhno porabo energije in predvsem s ceno takšnega sistema.

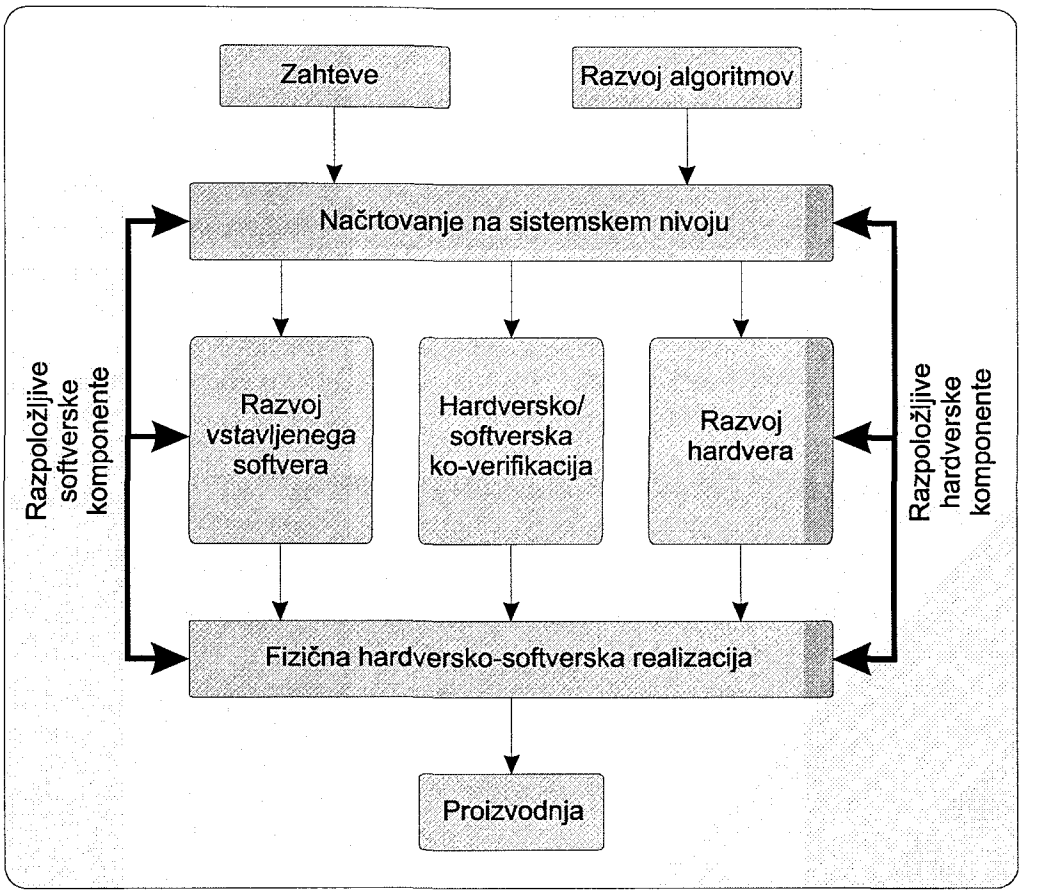
3. Sistem-on-a-chip (SoC) vključeni sistemi

Z napredkom v tehnologiji izdelave integriranih vezij in razvojem orodij za avtomatizacijo načrtovanja integriranih vezij je postalo število razpoložljivih tranzistorjev na posameznem integriranem vezju tako veliko, da presega zahteve še tako kompleksnih elektronskih podsklopov. Brez takšnih omejitev pa se je pojavila možnost, da funkcionalnost celotnega sistema združimo v enem samem integriranem vezju (System on a Chip, SoC).



Slika 2: SoC sistem

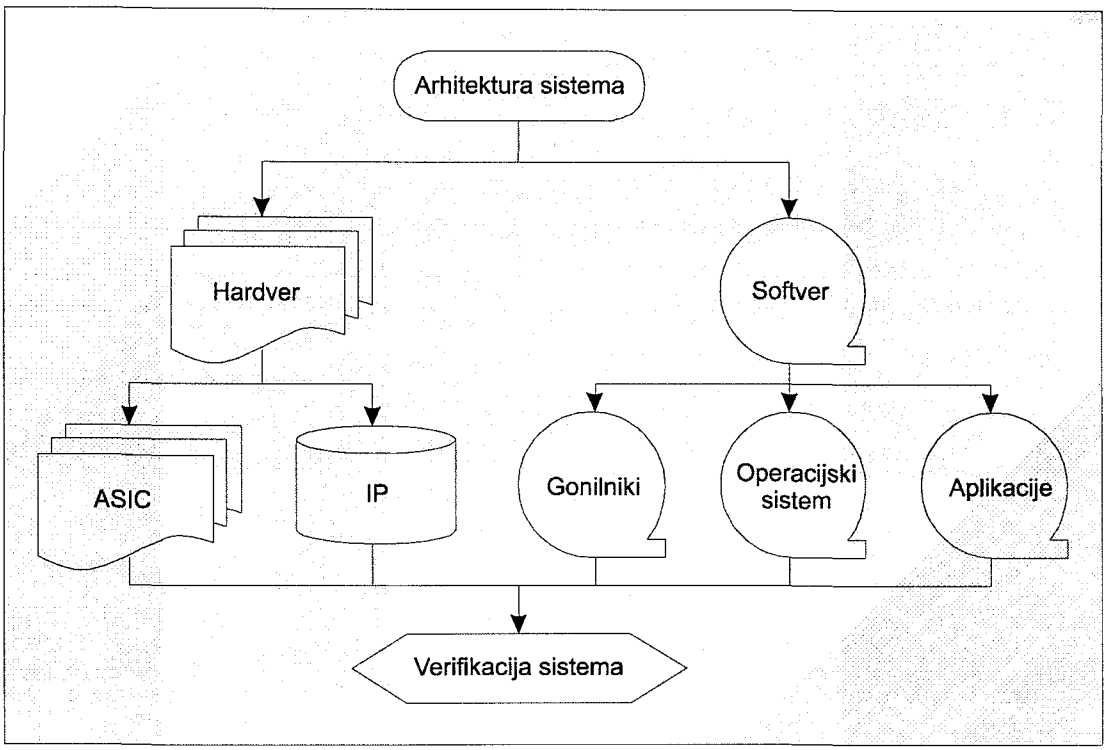
Integrirana vezja, načrtana po SoC principu so sestavljena iz kombinacije vnaprej narejenih megacelic z določeno funkcionalnostjo (tudi IP - "Intellectual Property" blok), npr. mikroprocesorji, DSP, pomnilniki in ostali bloki z regularno strukturo. Z vključenim programabilnim elementom postane programska oprema del integriranega vezja, s tem pa pomemben del načrtovalskega procesa. Hkrati kombinacija kompleksnih IP blokov in vključene programske opreme zahteva, da se načrtovalci osredotočijo predvsem na vrednotenje, preverjanje in medsebojno integracijo več IP blokov in softverskih komponent.



Slika 3: Načrtovanje sistemske arhitekture

Načrtovanje SoC zahteva podrobno analizo in poznavanje funkcije sistema, hkratno načrtovanje programske in strojne opreme z uporabo običajnih orodij (HDL prevajalnik, C prevajalnik, itd...), ter simulacijo in verifikacijo strojne in programske opreme z logičnimi simulatorji in programskimi razhroščevalniki na vseh nivojih načrtovanja.

Funkcionalnost sistema porazdelimo med hardver, sestavljen iz različnih IP blokov, softver, izveden s softverskimi rutinami, nekaj vezne logike in različne uporabniško načrtane analogne in/ali digitalne bloke. Osnovni problem načrtovanja SoC vezja je, kako razdeliti funkcionalnost sistema, kateri bloki in kako bodo uporabljeni, ter na kakšen način bodo medsebojno povezani.



Slika 4: Načrtovanje SoC sistema

Specifikacije iz procesa načrtovanja sistema bodo vodilo načrtovalcem posameznih hardverskih in softverskih podsklopov, hkrati pa je treba že v fazi načrtovanja sistema predvideti načine simulacije in verifikacije delovanja celotnega sistema.

Proizvajalci nudijo knjižnice IP blokov predvsem s področja žičnih (ISDN, HDLC) in brezžičnih (GSM) telekomunikacij, kar je posledica standardiziranih povezovalnih protokolov. Vendar se veča tudi število knjižnic za druga področja uporabe - za elektroniko široke potrošnje ter različne računalniške in multimedijske aplikacije.

4. Namen naloge

Namen naloge je izdelava digitalnega nadzorno-krmilnega vezja. Vezje naj bo izvedeno kot hibridni softversko-hardverski sistem, sestavljen iz programabilnega mikroprocesorskega jedra, ter optimiziranih hardverskih podsklopov za določene naloge, s katerimi razbremenimo mikroprocesor.

Sistem je v grobem sestavljen iz:

- mikroprocesorskega jedra
- pomnilnikov (RAM in ROM) ustrezne velikosti
- logičnih podsklopov, ki opravljajo določeno funkcijo
- vhodno - izhodnih vezij

V vezju bo mikroprocesorsko jedro predstavljalo sklop, s katerim bomo dosegli optimalno delovanje celotnega sistema. Naš cilj ni sistem, ki bo v kar največji meri izkoristil funkcionalnost in zmogljivost mikroprocesorja, pač pa bomo z vgrajenim mikroprocesorjem zmanjšali velikost, kompleksnost in s tem ceno integriranega vezja.

5. Zasnova sistema

Nadzorno-krmilno vezje naj opravlja naslednje funkcije:

- Izvajanje uporabniškega programa za upravljanje sistema in obdelavo vhodnih in izhodnih podatkov
- Registracija impulzov merilnega senzorja
- Ura realnega časa
- Oddaja in sprejem podatkov (komunikacija) od/do nadrejenega nadzornega sistema
- Prikaz merjenih vrednosti na multipleksnem LCD prikazovalniku
- Nadzor napajanja in možnost shranjevanja podatkov v primeru izpada napajalne napetosti

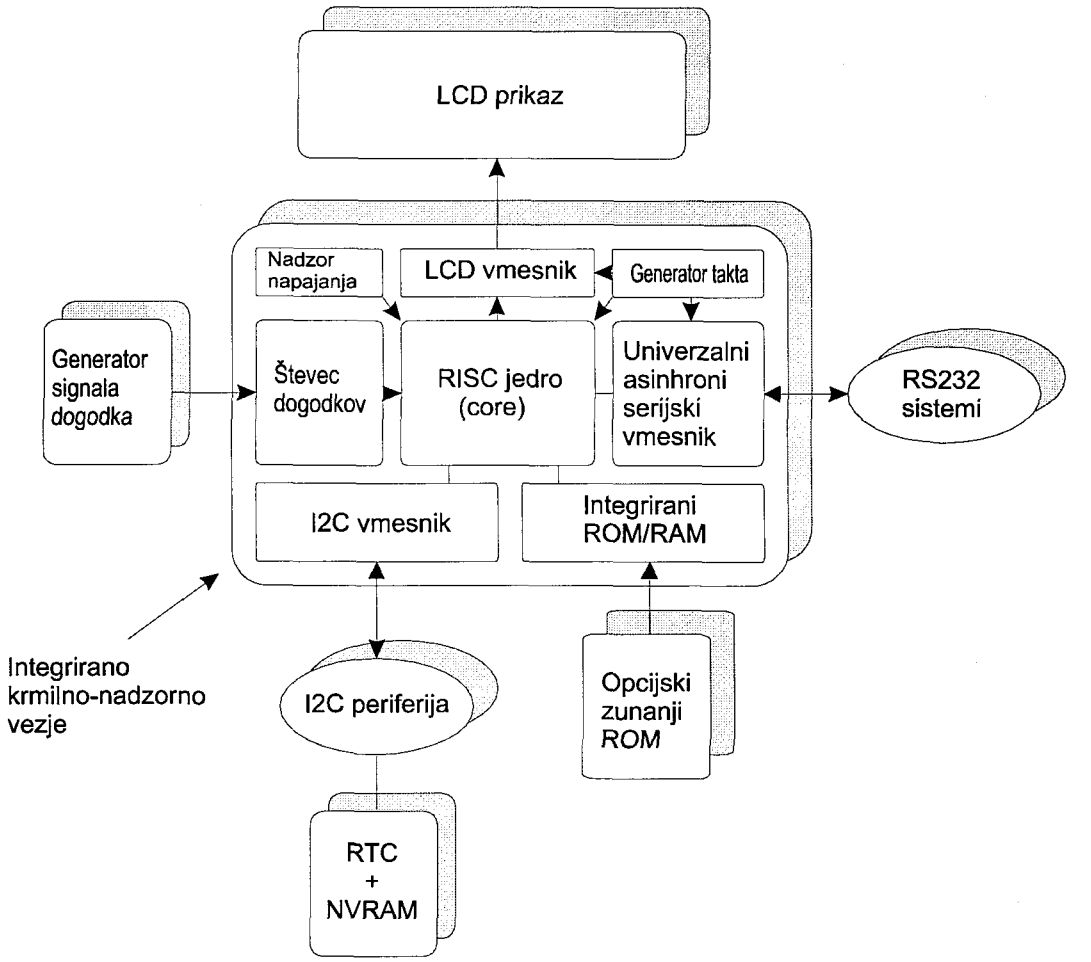
V vezju naj se procesi izvajajo hkrati, funkcije sistema (registracija, ura, komunikacija, ...) pa naj imajo čim manj vpliva na hkratno izvajanje uporabniškega programa. Zaželjeno je tudi, da komunikacija proti nadrejenemu sistemu poteka asinhronsko, saj ne vemo, kdaj bo nadrejeni sistem komuniciral z našim vezjem.

6. Modeliranje sistema

Model predstavlja abstrakcijo funkcionalnosti sistema, s katero lahko analiziramo različne izvedbe sistema. Z modeli zajamemo pomembne medsebojne vplive posameznih komponent celotnega sistema. Model vključenega sistema mora prikazati:

- zgradbo sistema iz različnih komponent, ki lahko delujejo z različnimi hitrostmi izvajanja
- medsebojne vplive posameznih komponent in njihovo sinhronizacijo pri komunikaciji
- časovne omejitve pri izvajanju posameznih operacij

7. Sistemski model



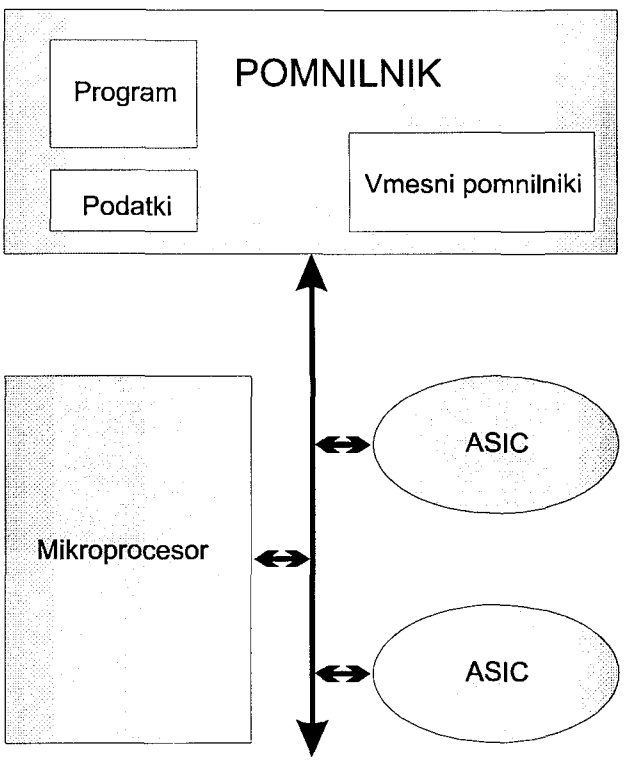
Slika 5: Blokovna shema sistema

S sistemskim modelom predstavimo zgradbo in funkcionalnost sistema, s katerim bomo izvedli točno določeno funkcijo. Sistem, predstavljen na Slika 5, bo za opravljanje svoje funkcije sestavljen iz naslednjh blokov:

- Programabilno jedro, ki ga tvorijo integrirano mikrojedro ("microcore") RISC procesorja, z ustrezno količino programskega (ROM) in podatkovnega (RAM) pomnilnika, potrebnega za delovanje programa za krmiljenje in nadzor okolice, ter delovanje programskega sklopa za obdelavo podatkov
- Števec dogodkov, ki se poveča ob vsakem signalu dogodka
- Gonilnik multipleksnega LCD prikaza za nadzor stanja sistema
- Standardni serijski komunikacijski vmesnik (UART) za komunikacijo navzgor proti nadrejenim sistemom

- I_2C sistemsko vodilo za komunikacijo med zunanjimi elementi sistema (ura realnega časa z generatorjem prekinitvenih impulzov, pomnilnik brez izgube vsebine ob prekinitvi napajanja)
- Blok za nadzor napajanja

Sistem naj bo izveden kot optimalna kombinacija softverskega in hardverskega dela, potrebnega za zanesljivo opravljanje željenih naloge. Za ciljno arhitekturo izvedbe izberemo arhitekturo na Slika 6, kjer programabilni komponenti (procesorju) pri izvajanu nalog pomagajo uporabniško načrtani hardverski sklopi za opravljajnje posebnih opravil.



Slika 6: Ciljna arhitektura

Pri načrtovanju našega sistema privzemimo, da imamo na voljo le eno programabilno komponento. Sistem, sestavljen iz več procesorjev zahteva dodatno sinhronizacijo njihovega delovanja, prav tako pa se zaradi medprocesorske komunikacije poveča promet in s tem izkoriščenost vodila.

Sistem naj ima samo en nivo pomnilnika, do katerega lahko dostopa izključno procesor. Komunikacija do ostalih hardverskih sklopov naj poteka po mikroprocesorjevem naslovnem in podatkovnem vodilu, hardverski podsklopi pa naj komunicirajo s procesorjem po načelu deljenega pomnilnika ("shared memory"), definiranega v podatkovnem naslovnem prostoru mikroprocesorja (MMIO, memory mapped I/O).

Upravljanje z vodilom naj bo v pristojnosti mikroprocesorja, kajti vsak mikroprocesor vsebuje optimiziran sklop za nadzor vodila. Izvedba takšnih sklopov v uporabniško načrtanih hardverskih komponentah bi sistem po nepotrebnem zapletla in precej povečala končno geometrijo vezja.

Mikroprocesor naj vsebuje dovolj prekinitvenih linij, kar bo omogočilo lažjo signalizacijo dogodka med posameznimi sklopi sistema, vsi hardverski sklopi pa naj imajo dobro definirano začetno, RESET stanje.

Mikroprocesor bo v našem sistemu opravljal le vlogo dodatnega vira, s katerim bomo dosegli željeno funkcionalnost in hkrati zmanjšali končno geometrijo vezja.

8. Procesni model

Za razliko od sistemskega modela, s katerim ponazorimo zgradbo sistema, se pri procesnem modelu osredotočimo na lastnosti procesov, ki potekajo v sistemu. S procesnim modelom lahko prikažemo potek posameznega procesa v obliki algoritma ali kot opis funkcije, ki jo proces opravlja. Modeliramo tudi reakcijo procesa na signal iz okolice.

Iz analize hkratnega poteka izvajanja procesov in njihove sinhronizacije lahko ocenimo količino virov, ki jih zahteva hkratnost izvajanja, število virov pa pomeni, s kakšnim številom hardverskih podsklopov bomo lahko zadostili vsem časovnim omejitvam izvajanja procesov, kot jih narekujejo specifikacije.

Procesni model sistema vsebuje tudi nabor vrat, preko katerih sistem komunicira s svojo okolico. Okolica vpliva na sedanje in prihodnje delovanje sistema, reaktivna narava sistema pa je modelirana z odzivom na signale iz vrat. Vrata lahko modelirajo pomnilniško lokacijo, drug sistem, ali podsklop sistema.

Delovanje procesov je omejeno z različnimi omejitvami, ki jih lahko definiramo na različnih nivojih abstrakcije sistema. Običajno predpišemo minimalno in maksimalno mejo zaksnitve pri delovanju dveh zaporednih operacij ali minimalno in maksimalno mejo hitrosti izvajanja ene operacije v zanki.

9. Željena funkcionalnost sistema

Funkcionalnost našega sistema lahko razdelimo na hkratno delujoče procese, potekajoče v realnem času:

- Štetje dogodkov
- Možnost preklapljanja med izvajanjem 8 neodvisnih softverskih opravil za kontrolo delovanja sistema in obdelavo prejetih podatkov.
- Vzdrževanje ure realnega časa
- Komunikacija po asinhronskem serijskem vmesniku
- Osveževanje LCD prikaza
- Nadzor napajanja in reagiranje na prekinitev napajanja

Procesi v integriranem vezju naj z okolico komunicirajo preko naslednjih vrat:

- Dogodek naj se registrira, ko se na vhodu spremeni logično stanje iz logične "0" v logično "1"
- Preklop softverskih opravil naj se zgodi ob prehodu krmilnega signala zunanjega časovnega vezja iz logične "0" v logično "1"
- Prav tako naj se ob istem signalu osveži ura realnega časa
- Komunikacija po asinhronskem serijskem vmesniku naj poteka po RS232 priporočilu z 1 start bitom, 1 stop bitom in 8 podatkovnimi biti, brez paritetnega bita
- Osveževanje LCD prikaza naj poteka neprekinjeno, z analognimi signali, primernimi za osveževanje multipleksiranega 4/1 sedem-segmentnega LCD prikaza

Določiti moramo še omejitve, katere bodo vplivale na delovanje posameznih procesov:

- Zaradi navezanosti obdelave podatkov na točno uro realnega časa, mora vezje vsebovati sklop za natančno določanje časovnih intervalov
- Maksimalna frekvenca dogodkov je ~100/s
- Maksimalna hitrost komunikacije po serijskem vodilu je 9600bit/s (= 960 znakov/s), polni dupleksni način
- Konstantno osveževanje LCD prikaza s frekvenco najmanj 100Hz
- Čim hitrejši odziv na signal prekinitve napajanja

Funkcionalnost procesov lahko modeliramo na več načinov. Sistem lahko opišemo s katerim od proceduralnih jezikov za opis integriranih vezij (npr. verilog, VHDL), vendar t. im. "data-flow graph" modeli bolje prikažejo hkratno delujoče procese. Načrtovalec je pri uporabi HDL jezika prisiljen nekatere hkrati delujoče procese modelirati kot serijo zaporednih operacij, oziroma spremeniti naravno obnašanje modelov[1].

10. "Data-flow graf" (DFG) model procesov

"Data-flow" grafi (DFG) omogočajo podatkovno gnani ("data-driven") opis, s katerim naravneje modeliramo aktiviranje posameznih hkratno delujočih procesov. Aktiviranje procesa povzroči prisotnost podatka na vhodnih vratih procesa, na ta način pa lahko z DFG zadovoljivo modeliramo tako softverske, kot hardverske procese. Vendar moramo pri modeliranju softverskih procesov z DFG paziti, da ne zmanjšamo zrnatosti softverskega DFG opisa do nivoja strojnega procesorskega ukaza, saj takšni opisi postanejo prekomplicirani za DFG analizo.

DFG je polarni aciklični graf $G = (V, E, \chi)$, sestavljen iz operacij $V = \{v_0 , v_1 \ldots v_N\}$, kjer sta v_0 in v_N operaciji izvora in ponora. Robna množica $E = \{(v_i , v_j)\}$ pa predstavlja odvisnosti med dvema operacijami. Funkcija χ predstavlja Boolovo operacijo za vsako posamezni robno odvisnost.

V Tabela 1 so zbrane operacije, uporabljane v DFG modelu. Z njimi lahko predstavimo tako softverske, kot tudi hardverske operacije, ki potekajo v kombiniranem hardversko-softverskem sistemu.

Operacija	Pomen
no-op	Brez operacije
cond	Pogojna vejitev
join	Pogojna združitev
op-logic	logična operacija
op-arithmetic	Aritmetična operacija
op-relational	Primerjalna operacija
op-io	V/I operacija
wait	Čakaj na signal
link	Hierarhična operacija

Tabela 1: Operacije DFG modela

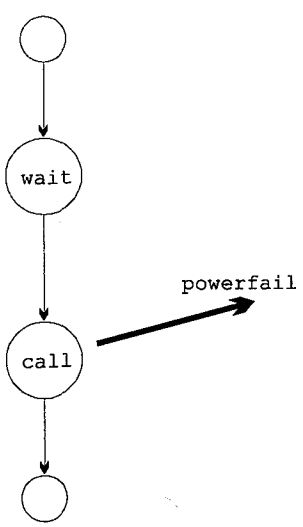
[1] Takšen primer je model prekinitvenega vezja. Posamezen bit v registru aktivnih prekinitev se postavi ob prekinitvi in zbriše ob koncu servisne rutine, vendar HDL jeziki ob vpisu nove prekinitve v prekinitveni register prejšnjo, mogoče še aktivno, izbrišejo.

11. Procesi in omejitve v procesih sistema

11.1. Nadzor napajanja

Nadzor napajaja lahko preprosto opišemo z uporabo operacije `wait`, s katero ustavimo izvajanje procesa, dokler željeni pogoj ni izpolnjen. Proces ob izpolnjenem pogoju pokliče `power-fail` proceduro, kjer se sistem ustavi, zato ni omejitev v hitrosti ponavljanja procesa. Omejitev v procesu je le čas med izvajanjem `wait` operacije in `call` operacije.

```
process savedata (pf-trig)
     // deklaracija vrat preko katerih
     // komunicira proces
     in port pf-trig;
{

     // cakaj na signal
     wait (posedge pf-trig);

     // skoci na power-fail proceduro
     call (power-fail);
}
```



Slika 7: Proces nadzora napajanja

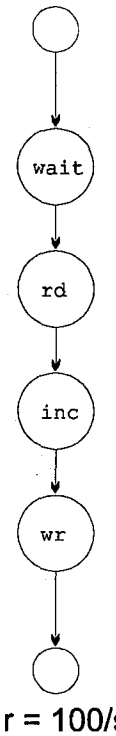
11.2. Štetje dogodkov

Izvajanje procesa tudi tukaj ustavimo z `wait` funkcijo, ki blokira izvajanje procesa. *Zahteva po minimalno 100 registriranih dogodkih na* sekundo *(r_i = 100s-1)* pa nam omeji spodnjo mejo hitrosti ponavljanja procesa, oziroma maksimalni čas med dvema ponovitvama aritmetične operacije seštevanja *(v)*:

$$\max_k\left(t_k(v_i)-t_{k-1}(v_i)\right)\le\frac{\tau}{r_i} \qquad (11.2.1)$$

`Wait` operacija je nedeterministična operacija, zato jo za izračun maksimalnega dovoljenega časa med operacijama seštevanja izločimo, oz. ji določimo čas izvajanja *t(v)=0*. Maksimalna hitrost ponavljanja procesa navzgor ni omejena, saj `wait` operacija poskrbi za sinhronizacijo s signalom dogodka.

```
process stevec (dogodek, st_dogodkov)
      // deklaracija vrat preko katerih
      // komunicira proces
      in port dogodek;
      mem port st_dogodkov;
{
      // lokalne spremenljivke procesa
      int temp;

      // cakaj na signal
      wait (posedge signal);

      // ob signalu povecaj stevec dogodkov
      temp = read (st_dogodkov);
      temp = temp + 1;
      write (st_dogodkov) = temp;
}
```



Slika 8: Proces štetja dogodkov

11.3. Razdeljevalnik opravil (scheduler) in števec ure realnega časa

V enem procesu bomo izvršili dve medsebojno nedvisni operaciji: izbiro trenutnega softverskega opravila (task) in osveževanje števca ure realnega časa. Osveževanje ure in hkrati preklop softverskega opravila naj se aktivira ob signalu zunanje časovne reference (1024 impulzov/s).

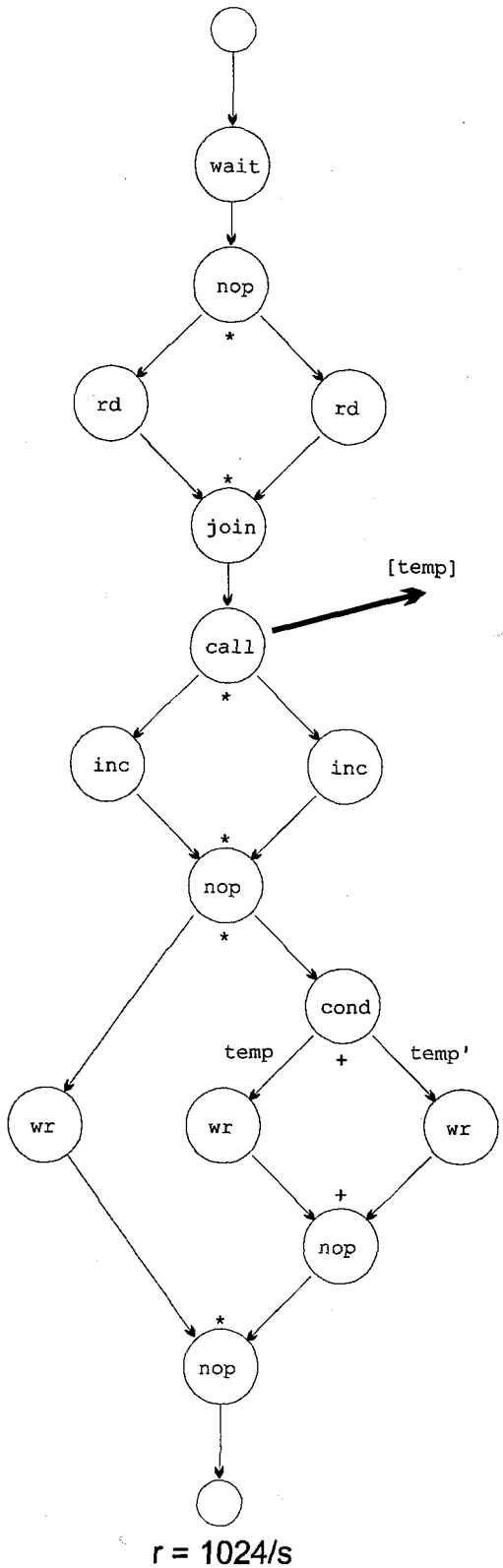
(Hkratnost izvajanja operacij je označena z oklepajema < in >.)

Enako kot prej nam zahteva po obdelavi 1024 signalov na sekundo *(r_i =1024s-1)* omeji spodnjo mejo hitrosti ponavljanja procesa. Celoten proces, vključno s klicano rutino softverskega opravila se mora končati v 1/1024 sekunde. V nasprotnem primeru pride do napake zaradi neželjene prekinitve izvajanje posameznega softverskega opravila.

Razdeljevalnik opravil ciklira med osmimi softverskimi opravili, zato vsako opravilo pride na vrsto v 8/1024 = 7.8ms. Hitrost izvajanja navzgor ni omejena, saj za sinhronizacijo z zunanjo časovno referenco poskrbi `wait` operacija.

```
process scheduler (signal, scheduled, RTC)
      // deklaracija vrat preko katerih
      // komunicira proces
      in port signal;
      mem port scheduled;
      mem port RTC;
{
      // lokalne spremenljivke procesa
      int  temp;
      int  ticks;

      // cakaj na signal
      wait (posedge signal);

      <
      // Stevec 0 -> 7 (Round-Robin) izbiralnika
      // aktivnega opravila
      temp = read (scheduled);

      // preberi uro realnega casa
      ticks = read (RTC);
      >

      // klic podrejenega procesa
      call (temp);
```

```
      <
      // krozno povecaj stevec izbiralnika
      temp = temp + 1;

      // povecaj stevec ure realnega casa
      ticks = ticks + 1;
      >

      <
      // shrani indeks aktivnega opravila
      if (temp >= 8)
            write (scheduled) = 0;
      else
            write (scheduled) = temp;

      // shrani stevec ure
      write (RTC) = ticks;
      >
}
```



r = 1024/s

Slika 9: Proces razdeljevalnika opravil

11.4. Sprejem podatkov po asinhronskem serijskem vmesniku

Asinhronski prenos po serijskem vmesniku se lahko zgodi ob kateremkoli času, zato mora biti vezje vedno pripravljeno na sprejem in delno obdelavo sprejetega podatka. 8-bitni podatek je uokvirjen med start in stop bit, pretok bitov znotraj okvirja pa je lokalno sinhronski z 9600bit/s

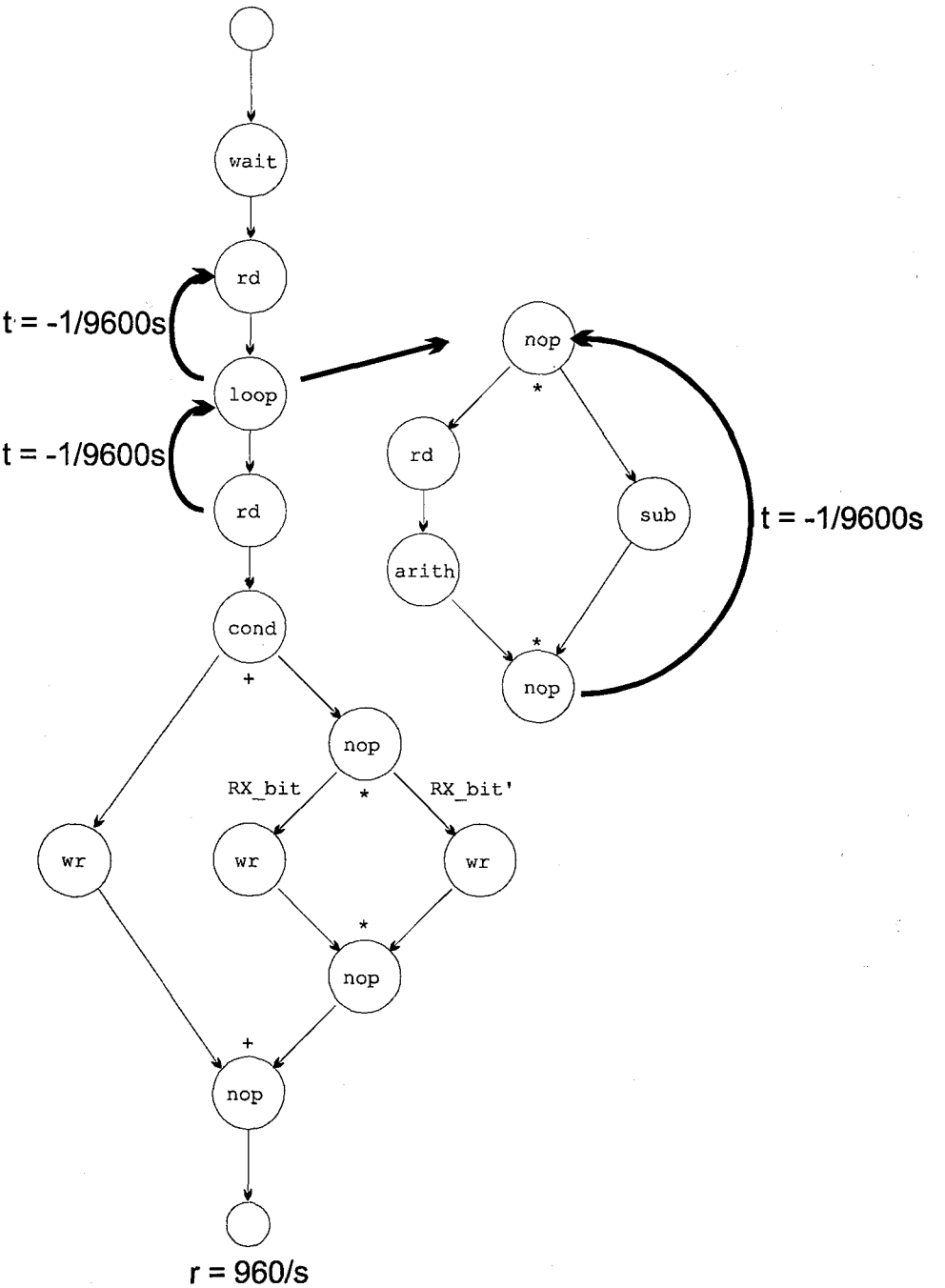
Proces sprejema mora pravilno sprejeti lokalno sinhroni start/stop okvir z 8-bitnim podatkom, ter ga shraniti, preden je sprejeti podatek pretečen (overflow) z naslednjim sprejetim okvirjem. Okvirji si lahko sledijo eden za drugim z maksimalno frekvenco 960 okvirjev/s.

Lokalno sinhronski prenos obdelamo z zanko v procesu, ki z 9600 ponavljanji na sekundo sinhrono bere RX vrata. Zanka se aktivira takoj, ko `wait` operacija zazna prisotnost startnega bita na vhodu. Maksimalna in minimalna hitrost ponavljanja sinhronske zanke pa sta odvisni od toleranc serijskega prenosa.

Hitrost ponavljanje celotnega procesa je omejena s frekvenco okvirjev. Minimalna hitrost ponavljanja procesa je 960 /s, za sinhronizacijo z začetkom okvirja poskrbi operacija `wait`. Maksimalni čas Δt_Z med dvema zagonoma zanke za zanesljivo branje serijskega prenosa ne sme biti daljši kot 1.04 ms.

```
#define START_BIT    0
#define STOP_BIT     1

process serial_RX (RX, RX_data, RX_framerr)
      // deklaracija vrat preko katerih
      // komunicira proces
      in port RX;
      mem port RX_data;
      mem port RX_framerr;

{
      // lokalne spremenljivke procesa
      boolean   RX_bit;
      int  RX_temp [8];
      int  i;

      // cakaj na detekcijo startnega bita
      wait (RX == START_BIT)
```

```
label1:
     RX_bit = read (RX);

     // nadaljuj v zanki:
     //  8 podatkovnih bitov
     for (i = 0; i < 8, i++)
     {
label2:
          RX_bit = read (RX);
          // premakni RX_temp v levo in dodaj RX_bit
          RX_temp = (RX_temp << 1) & RX_bit;
     }
     // testiraj stop bit
label3:
     RX_bit = read (RX);
     if (RX_bit == STOP_BIT)
     {
          // ce je okvir podatka v redu,
          // shrani prejeti podatek v pomnilnik
          <
          write (RX_framerr) = 0;
          write (RX_data) = temp;
          >
     }
     else
          // signaliziraj napako okvirja
          write (RX_framerr) = 1;
}
```



Slika 10: Proces sprejema po UART vmesniku

11.5. Osveževanje LCD prikaza

Prikaz na LCD prikazovalniku naj se osveži vsaj 100x na sekundo, s čimer preprečimo utripanje prikazovalnika. Proces osveževanja LCD prikazovalnika lahko izvedemo tudi kot proces sekvenčnega osveževanja posameznih vrstic. Na ta način z enim klicem procesa osvežimo samo eno vrstico, zato moramo za željeno frekvenco osveževanja proces klicati

$$r_i = n \cdot f_{FR} \qquad (11.5.1)$$

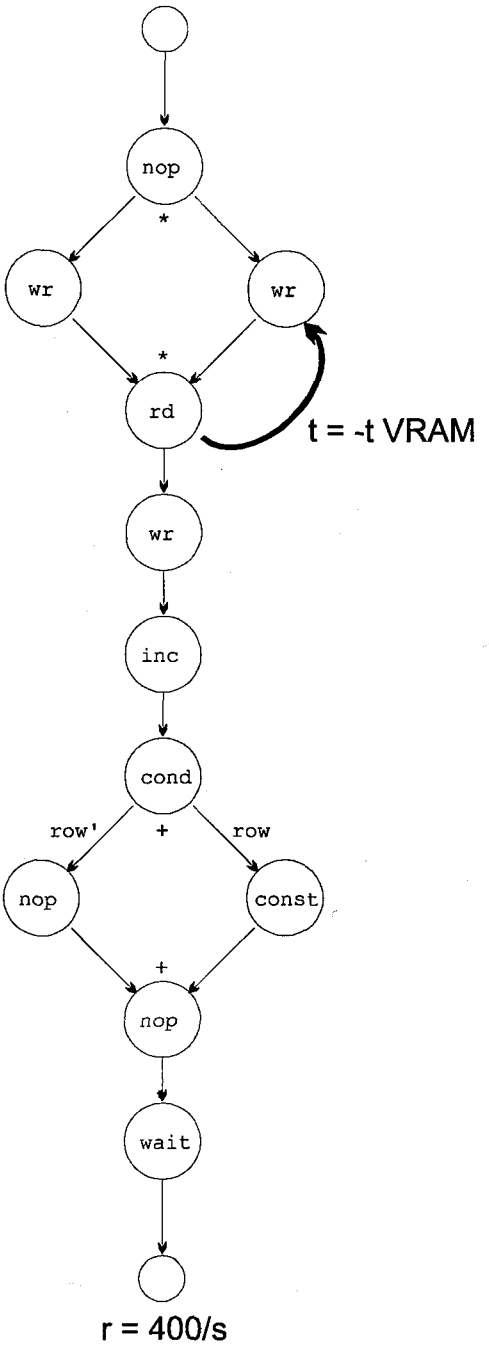
kjer n pomeni število vrstic, f_{FR} pa frekvenco osveževanja prikaza.

Z osveževanjem po vrsticah moramo proces izvesti najmanj 400x v sekundi, kar postavi minimalni čas izvajanja procesa $t(G)$ = 2.5ms. Hitrost izvajanja osveževanja omejimo z operacijo `wait` v zadnji vrstici procesa.

Video pomnilnik ima določen dostopni čas, zato imamo med naslavljanjem in branjem video pomnilnika omejitev izvajanja zaporednih operacij naslavljanja pomnilnika in branja. Omejitev označimo z negativnim robom v DFG, odvisna pa je od izbranega tipa pomnilnika.

```
// osvezevanje 100Hz
#define REFRESH_TIME_MS  10
#define REFRESH_ROW (REFRESH_TIME_MS / 4)

process LCD_refresh (LCD_row, LCD_data,
                     VRAM_addr, VRAM_data)
     // deklaracija vrat preko katerih
     // komunicira proces
     out port LCD_row    [2];
     out port LCD_data   [26];
     out port VRAM_addr  [2];
     in port VRAM_data   [26];
{
     // lokalne spremenljivke procesa
     int  row [2];
     int  row_data [26];

     // postavi VRAM_addr in LCD_row na trenutno vrstico
     // VRAM_addr in LCD_row sta neodvisna, zato se lahko
     // ukaza izvajata hkrati
     <
     write (VRAM_addr) = row;
     write (LCD_row) = row;
     >
```

```
      // preberi trenutno vrstico iz VRAM
      // in jo poslji na LCD_data
      row_data = read (VRAM_data);

label1:
      write (LCD_data) = row_data;

      // krozno povecaj stevec vrstic
      row = row + 1;
      if (row >= 4)
            row = 0;

      // cakaj na osvezevanje naslednje vrstice
      // cas osvezevanje ene vrstice je
      // 1/4 casa osvezevanja celotnega prikaza
      wait (REFRESH_ROW);
}
```



Slika 11: Proces osveževanja LCD prikaza

12. Opis in simulacija procesov s HDL jezikom

Paralelne procese, ki se izvajajo v sistemu lahko simuliramo v enem od proceduralnih jezikov za opis vezij. Na Slika 12 je prikazana simulacija vseh petih paralelnih procesov našega sistema, skupaj z njihovimi notranjimi spremenljivkami stanj.

Z verilogom, katerega smo izbrali za opis procesov, na eleganten način opišemo časovne omejitve: čakanje na signal prekinitve izvedemo z ukazom `@(pogoj)`, s katerim nadomestimo ukaz `wait`, zakasnitve posameznih procesov pa opišemo z `#(zakasnitev)` ukazom.

Opis simulacije paralelnih procesov v verilogu, skupaj s testno datoteko vzbujalnih signalov se nahaja v prilogi 26.1 na strani 97.

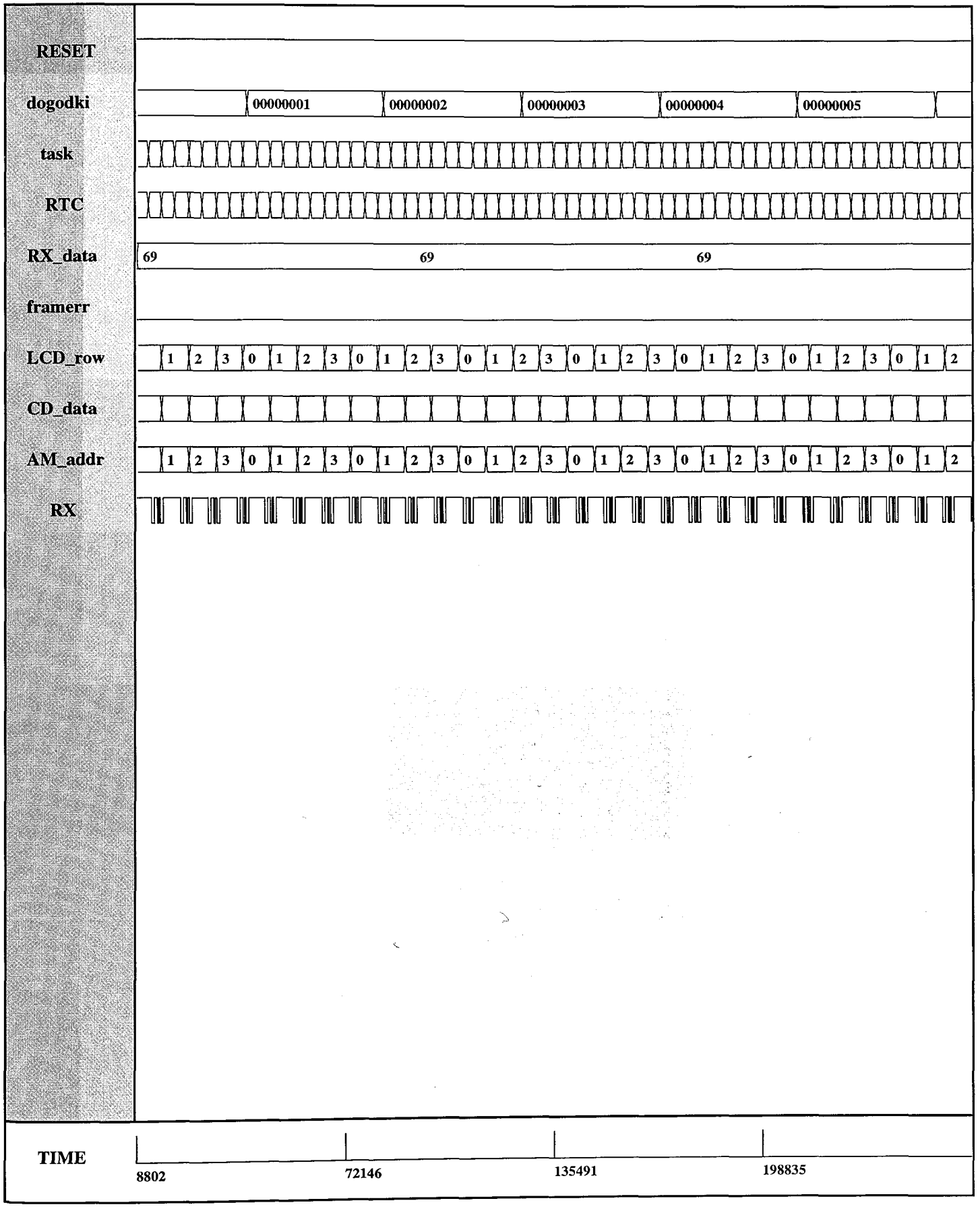
Izvedba celotnega sistema s paralelnimi procesi je precej potratna, saj za vsak vzporedni proces potrebujemo svoj vir. Radi pa bi, da se procesi izvajajo sekvenčno na čimmanjšem številu virov. Eden izmed virov je mikroprocesor kot univerzalna programabilna komponenta, podprta z uporabniško definiranimi podsklopi, s katerimi zadostimo časovnim omejitvam izvajanja posameznih procesov.

(na naslednjih straneh)

Slika 12: Simulacija paralelenih procesov

a) simulacija paralelnih procesov
b) detajl simulacije paralelnih procesov

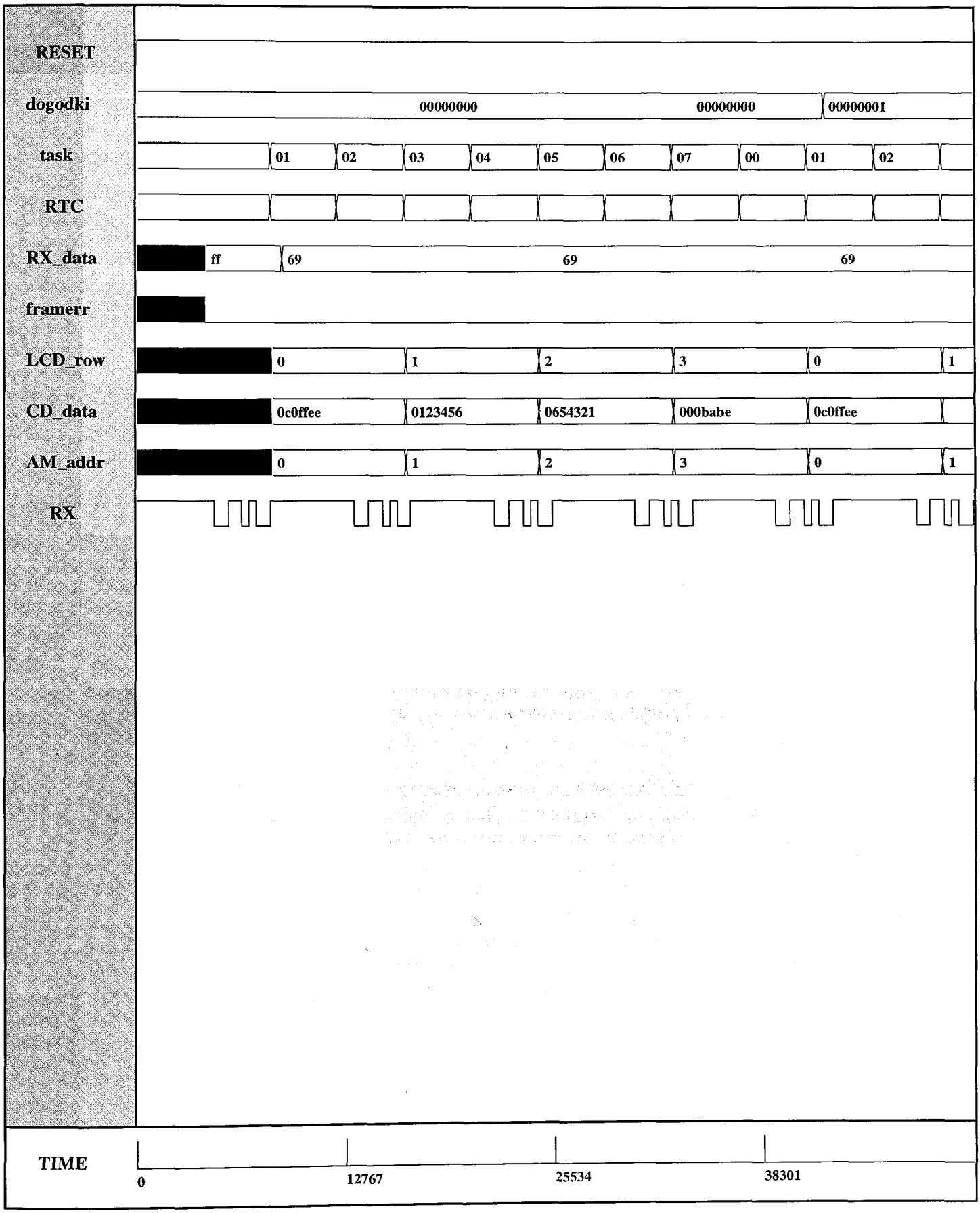
Header: paralelni procesi	
User:	
Date: Mar 13, 2000 14:31:31	Time Scale From: 8802 To: 262180 Page: 1 of 1



Header: paralelni procesi, detajl

User:

Date: Mar 13, 2000 14:33:23 Time Scale From: 0 To: 51068 Page: 1 of 1



13. Homogeni in heterogeni sistemi

Sistem, opisan z zbirko procesov, izvedemo s pravilno izbranimi komponentami, s katerim zadostimo omejitvam hitrosti izvajanja in omejitvam frekvence ponavljanja operacije. Sistem lahko izvedemo z enim od dveh pristopov:

- homogeni sistemi:

Funkcionalnost homogenega sistema je realizirana z minimalnim številom komponent, ki so lahko izvedene v celoti v hardveru, ali pa v celoti kot softverske rutine. Pri tem so hardverske izvedbe omejene z velikostjo končne geometrije (layout) posameznih komponent, medtem ko so softverske izvedbe omejene z izkoriščenostjo procesorja, izkoriščenostjo vodil in časom izvajanja posameznega opravila.

- heterogeni sistemi

Heterogeni sistem sestavljajo hardverske in softverske komponente, naloga načrtovalca pa je, da ob upoštevanju vseh zahtev, funkcionalnost sistema optimalno porazdeli med softverski in hardverski del.

Eden izmed pristopov je, da vključeni mikroprocesor obravnavamo kot splošno sistemsko sredstvo, ki lahko izvaja različne tipe softversko izvedenih opravil, medtem ko hardverske komponente lahko opravljajo le eno samo specializirano operacijo. Zaradi različnega načina izvajanja osnovnih operacij (zaporedno izvajanje ukazov v softveru proti vzporednem izvajanje logičnih funkcij v hardveru) je izvajanje opravil v hardveru precej hitrejše. Ker lahko hardver optimiziramo za opravljanje željenega opravila, v hardveru izvedemo funkcije, ki morajo zadostiti določenim časovnim omejitvam - največkrat pri komunikaciji z zunanjim svetom.

Pomembna razlika je v načinu izvajanja softverskih in hardverskih komponent. Mikroprocesor izvaja opravilo iz zaporedja vnaprej definiranih ukazov ("instruction-driven"), ter glede na podatke na vhodu veji program. Hardverska komponenta pa reagira, ko se podatek pojavi na vhodu ("data-driven") in izvede svoje opravilo na sprejetem podatku.

Mikroprocesorske prekinitve s pripadajočimi ko-rutinami so hibrid med ukazno in podatkovno gnanimi komponentami. Prekinitev se aktivira glede na prisotnost podatka na na vhodu, mikroprocesor pa izvede željeno opravilo iz zaporedja ukazov. Zaradi možnosti, da mikroprocesor maskira nekatere prekinitve, ter zaradi prioritete reagiranja na prekinitve, so prekinitve zelo močno orodje za izvedbo željene funkcionalnosti sistema.

Komponente hibridnega sistema delujejo pri različnih hitrostih izvajanja, zato se morajo pri medsebojni komunikaciji sinhronizirati. Mehanizmi sinhronizacije pa dodajo večjo kompleksnost takšni medsebojni komunikaciji.

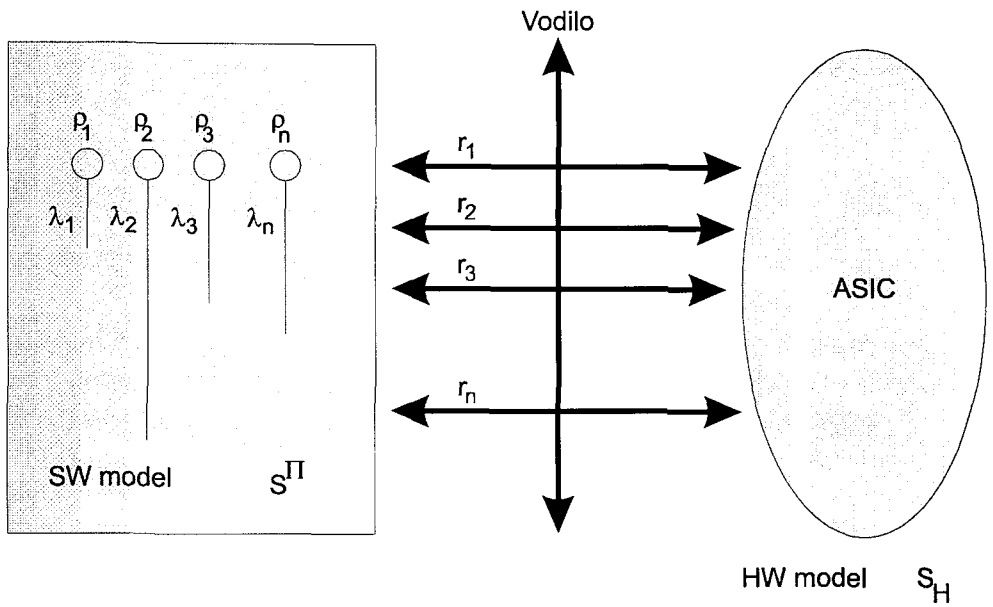
14. Delitev sistema na hardverski in softverski del

Delitev sistema na hardverski in softverski del mora zadovoljiti specifikacijam sistema, omejeni pa smo tudi z omejitvami pri načrtovanju in omejitvami pri hitrosti reagiranja na podatke iz zunanjega sveta.

Softverski del - program, razdeljen na posamezne vzporedno delujoče dele programa - programske niti karakteriziramo z:

- latenco posamezne programske niti, λ_l predstavlja gornjo mejo zakasnitve izvajanja programske niti
- reakcijska hitrost programske niti, ρ_l je definirana kot frekvenca izvajanja klicev posamezne niti. Spodnja meja frekvence izvajanja klicev programske niti je postavljena glede na omejitev frekvence izvajanja operacij v posamezni niti
- Velikost programa S^{Π} predstavlja izvedbo programa, skupaj s podatki. Program in podatki zahtevajo različne tipe pomnilnika, izvedba pomnilnika na istem vezju pa poveča končno geometrijo vezja.

Hardverski del karakteriziramo z njegovo velikostjo (S_H), navadno v številu ekvivalentnih NAND vrat. Zgodi se, da je izvedba v hardveru manjša po porabljenem prostoru in hitrejša od ekvivalentne softverske izvedbe (npr. izvedba kodirne tabele s programsko tabelo v ROM pomnilniku, ali izvedba iste tabele z optimizirano kombinacijsko logiko).



Slika 13: Delitev elementov na HW inSW

Pri hibridni hardversko-softverski izvedbi se pojavita še dve omejitvi:

- izkoriščenost procesorja (*P*), ki jo definiramo kot:

$$P = \sum_{i=1}^{n} \lambda_i \cdot \rho_i \qquad (14.1)$$

- Izkoriščenost vodila (*B*), ki je merilo vse komunikacije, potekajoče med procesorjem (oziroma programskim delom) in hardverskim delom integriranega vezja. Prenos *m* spremenljivk, ki se pojavljajo s frekvenco r_j, izkoristi vodilo

$$B = \sum_{j=1}^{m} r_j \qquad (14.2)$$

Posamezna opravila razdelimo na softverski (Φ_S) in hardverski (Φ_H) del z minimiziranjem funkcije:

$$f(\omega) = a_1 \cdot S_H(\Phi_H) - a_2 \cdot S^{\Pi}(\Phi_S) + b \cdot B - c \cdot P + d \cdot |m| \qquad (14.3)$$

Funkcija definira ceno hibridne softversko-hardverske izvedbe, pozitivne konstante a_1, a_2, *b*, *c* in *d* pa predstavljajo kompromise med velikostjo hardverskega in softverskega dela izvedbe, izkoriščenostjo procesorja in vodila, ter komunikacijski balast pri prenosu m spremenljivk.

Očitno je, da bo cena sistema najmanjša, ko čimveč opravil izvedemo v softveru s čimvečjo izkoriščenostjo procesorja. Cena izvedbe opravil v hardveru pa poleg večje končne geometrije pomeni tudi večjo izkoriščenost vodila, ter večji komunikacijski balast.

15. Potrebe procesov po virih

Vsak vzporedno delujoč proces potrebuje vir za svoje delovanje. Tako proces nadzora napajanja potrebuje ustrezni avtomat prehajanja stanj, ki bo ob prekinitvi napajanja aktiviral podsklop, s katerim se bo ustavilo delovanje integriranega vezja in izvedel prenos podatkov iz notranjih registrov v zunanje NVRAM vezje.

Tako lahko proces štetja dogodkov izvedemo v hardveru z binarnim števcem ustrezne dolžine, sestavljenim iz flip-flop vezij in nekaj logike. Izvedba ure realnega časa kot vir potrebuje svoj števec urinih signalov, izvedba procesa serijske kominikacije potrebuje svoj podsklop registrov, avtomatov prehajanja stanj in časovnih vezij za sinhronizacijo z zunanjim tokom podatkov; osveževanje LCD prikaza pa potrebuje svoj podsklop z video-pomnilnikom, generatorjem osveževalnega signala z ustreznim analognimi vezjem za generiranje signala pravilne oblike.

Z izvedbo vsakega procesa na svojem viru zagotovimo popolnoma nemoteno delovanje uporabniškega programa na vgrajenem mikroprocesorju. Prav tako kakor mikroprocesor, sta tudi RAM in ROM pomnilnik v celoti na voljo uporabniškemu programu.

Z izvedbo homogenega softverskega sistema si procesi med seboj delijo en sam univerzalni vir - mikroprocesor. Vendar ima takšna izvedba težave s sinhronizacijo z zunanjimi dogodki. Operacija čakanja na dogodek je softversko izvedena s t.im. "busy loop" zankami, s tem pa procesi, ki čakajo na dogodek s preverjanjem vhodnih vrat, obremenjujejo mikroprocesor in tako zmanjšujejo njegovo razpoložljivost za izvajanje uporabniškega programa. Takšno preverjanje vhodnih vrat tudi močneje obremeni notranje vodilo med mikroprocesorjem in vhodno / izhodnimi napravami, še posebej v izvedbah, kjer je podatkovno vodilo vhodno / izhodnih naprav skupno s podatkovnim vodilom pomnilnika.

Zaradi narave softverske izvedbe ukaza `wait` so tudi sinhronizacijski ukazi, s katerimi zadostimo minimalni / maksimalni omejitvi frekvence ponavljanja operacij (npr. v serijskem vmesniku in gonilniku LCD prikaza), prav tako izvedeni s pomočjo softverskih "busy loop" zank z določenim številom ponavljanj, katere spet precej obremenijo procesor.

Deljenje enega samega vira med več procesov v homogeni softverski izvedbi tudi poveča čas reagiranja procesa na aktivacijski signal, saj je frekvenca ponavljanja procesov omejena s hitrostjo razdeljevalnika softverskih opravil - "schedulerja", omeji pa tudi hitrost ponavljanja posameznega softverskega opravila. Na takšen način bi softversko krmiljenje serijskega vmesnika (po metodi "bit-banging") z 9600 bit/s zahtevalo aktiviranje pripadajočega opravila najmanj 9600x v sekundi. Podobno je za osveževanje LCD prikazovalnika po vrsticah zahtevano ponavljanje najmanj 400x v sekundi.

Procese, ki zahtevajo sinhrono delovanje, oziroma sinhronizacijo z zunanjim tokom podatkov, najbolje izvedemo v hardveru. Tako izvedeni procesi niso vezani na hitrost izvajanja mikroprocesorja, lahko potekajo pri poljubni hitrosti, poskrbeti moramo le za sinhronizacijo z mikroprocesorjem pri prenosu podatkov.

16. Izvedba procesov

16.1. Nadzor napajanja in komunikacija po I2C vodilu

Ob signalu prekinitve napajanja naj sistem čim prej shrani notranje stanje sistema v zunanji pomnilnik z rezervnim napajanjem. Pred tem naj konča trenutno delujoče softversko opravilo.

Signal prekinitve napajanja generira analogno Schmittovo preklopno vezje, mikroprocesor pa po prejetem signalu najprej konča trenutno softversko opravilo, nato pa aktivira prenos podatkov po I2C vodilu.

Ker se ob prekinitvi napajanja ustavi obdelava vseh procesov, lahko izvedemo komunikacijo po I2C vodilu kot softverski proces, s pomočjo "bit-banging" rutin, s katerimi direktno kontroliramo stanje na izhodih integriranega vezja.

Omejitev zakasnitve med signalom in aktiviranjem ustreznega softverskega procesa lahko zmanjšamo na minimum z uporabo "level 0" prekinitve mikroprocesorja. Ker moramo pred komunikacijo po I2C vodilu končati softversko opravilo Softversko opravilo, ki traja maksimalno 1ms, se bo komunikacija po I2C vodilu začela najkasneje po 1ms od prejetega signala prekinitve napajanja.

Pri maksimalni frekvenci I2C vodila 100kHz, traja prenos 100bitov približno 1ms, torej mora napajalnik po izpadu napajalne napetosti zagotoviti stabilno napetost še za najmanj 2ms. Mikroprocesor se po opravljenem prenosu podatkov ustavi v prazni zanki, kjer počaka na RESET signal, zato dolžina izvajanja procesa sama po sebi ni omejena.

16.2. Štetje dogodkov

Upoštevamo, da je proces štetja dogodkov "lahek" proces, saj predstavlja le proceduro povečanja 32bitne spremenljivke, poleg tega pa je frekvenca ponavljanja procesa štetja dogodkov relativno majhna (maksimalno 100x v sekundi).

Izvedba števca v hardveru zahteva 32 D-FF flip-flop celic (vsaka predstavlja 6 ekvivalentnih NAND vrat) z ustreznim logičnim vezjem, poleg tega pa še dekodersko vezje za dekodiranje V/I naslova. Vrednost registra števca mora biti dostopna softverskim opravilom, ki jih izvaja mikroprocesor. Ker pa je vodilo, s katerim komuniciramo z vhodno / izhodnimi registri preiferije le 8 bitno, zahteva hardverska izvedba sinhronizacijo med procesom pisanja in procesom branja iz registra števca: med zaporednim branjem štirih 8-bitnih lokacij, s katerimi izvedemo 32bitni register števca, lahko signal dogodka aktivira štetje, kar ima za posledico napačno prebrano vrednost iz registra.

Števec zato izvedemo kot softversko komponento. Maksimalna frekvenca ponavljanja softverskega opravila je 125/s, zato bi lahko proces štetja izvedli s softverskim opravilom. Ker pa je signal dogodka nedeterminističen, bi softversko opravilo večino časa v zanki čakalo na signal, zato softversko izvedbo števca izvedemo kot prekinitveno rutino.

Slaba stran izvedbe s prekinitvijo pa je, da moramo času izvajanja rutine števca dodati tudi čas zakasnitve detekcije prekinitve in čas izvajanja glavne prekinitvene rutine.

16.3. Razdeljevalnik opravil in števec ure realnega časa

Razdeljevalnik je že sam po sebi vezan na softverska opravila, med katerimi preklaplja, zato izvedemo celoten proces v softverski izvedbi. Ker so softverska opravila vezana na realni čas, tudi hitrost njihovega preklopa vežemo na zunanjo časovno referenco; na ta način tudi sinhroniziramo delovanje celotnega sistema. Razdeljevalnik opravil ciklira med osmimi opravili, vsakemu opravilu pa odmeri 1ms dolgo časovno rezino. Opravila si bodo tako sledila vsako milisekundo, vsako opravilo pa se bo zagotovo ponovilo s frekvenco 125Hz, oziroma na vsakih 8ms.

Zunanjo časovno referenco priključimo na prekinitveni vhod, s katerim aktiviramo prekinitveno rutino razdeljevalnika opravil. Ta poskrbi za preklop in pravilno izvajanje softverskega opravila, hkrati pa vzdržuje števec ure realnega časa. Ura realnega časa je tako vedno na voljo vsem procesom in ne potrebuje bralno-pisalne sinhronizacije.

Proces razdeljevalnika opravil vsakemu posameznemu softverskemu opravilu doda čas, ki ga porabi za preklapljanje med opravili, ter za osnovno preverjanje pravilnosti delovanja opravil.

16.4. Asinhroni serijski vmesnik (UART)

Želimo, da bi integrirano vezje s svojo okolico komuniciralo z univerzalnim asinhronim vmesnikom, po RS232 specifikaciji s hitrostjo 9600bit/s. RS232 protokol predvideva, da se 8bitni podatek (zadnji bit podatka je lahko paritetni) - znak - vstavi v okvir, ki se začne s START bitom in konča s STOP bitom. Izvedba RS232 9600bit/s protokola v softveru z direktnim krmiljenjem ustreznih vhodno-izhodnih vrat vezja ("bit-banging") zahteva operacijo branja ali pisanja na vrata s frekvenco najmanj 9600Hz. To pa v našem primeru ni mogoče, saj se posamezno softversko opravilo začne le vsakih 8ms. Hkrati bi zaradi naključne narave komunikacije takšen proces večino časa čakal v zanki na START bit, kar pa bi po nepotrebnem obremenjevalo procesor.

Poleg tega je prenos bitov znotraj podatkovnega okvira sinhron, vsako preveliko odstopanje med taktom sprejemnika in taktom oddajnika pa lahko povzroči napako okvirja. Proces branja RS232 vrat zahteva ustrezno časovno referenco, s katero se uskladi serijski prenos med oddajnikom in sprejemnikom.

Dvožična komunikacija po RS232 vmesniku predvideva, da je sprejemnik vedno pripravljen na sprejem okvirja. Takšna specifikacija pa predvideva neblokirno pisanje in blokirno branje - sprejemnik čaka, kdaj se bo začel prenos podatka. Zato proces komunikacije razdelimo na sprejemni proces, izveden v hardveru, ki sprejme okvir z 8-bitnim podatkom in detektira možne napake okvirjanja; ter proces obdelave sprejetega podatka, ki se izvaja kot asinhronski proces, izveden kot softversko opravilo.

Hardverski del procesa naj samostojno zazna začetek prenosa okvirja, sprejme znak in po končanem prenosu okvirja s prekinitvijo pokliče prekinitveno rutino, ta pa bo sprejeti znak prestavila iz registra sprejemnika v pomnilnik. Prekinitvena rutina se mora izvesti vsaj 960x v sekundi da zagotovi nemoteno komunikacijo brez pretekanja registra sprejemnika.

Pojavi pa se problem, ker se softverska opravilo lahko ponavlja s frekvenco največ 128Hz, medtem ko lahko okvirji s podatki po RS232 prihajajo s frekvenco 960 znakov / s. Zato za začasno shranjevanje sprejetih znakov uporabimo t.im. FIFO izravnalnik ("first in, first out" - prvi noter, prvi ven), globine:

$$N_{FIFO} = \frac{960}{128} = 7.5 \Rightarrow 8 bitov \qquad (16.4.1)$$

Da podatki, shranjeni v FIFO izravnalniku končne dolžine 8 bitov ne pretečejo ostalih, naj pripadajoče softversko opravilo izravnalnik ob vsakem klicu popolnoma izprazne - obdelati mora vse sprejete znake.

Tudi oddajni del serijskega vmesnika izvedemo v hardveru, saj bi zaradi delno sinhronskega delovanja oddajna rutina pri oddaji preveč časa porabila za izvajanje sinhronizacijskih čakalnih zank. Časovno referenco 9600Hz potrebujemo že v sprejemnem dela vmesnika, tudi dekoder naslova si lahko oddajni del "izposodi" pri sprejemnem delu. Oddajni del, izveden v hardveru predstavlja le premikalni register in manjši avtomat prehajanja stanj, z njima pa lahko brez problemov dosežemo delovanje v polnem dvosmernem, "duplex" načinu.

Zaradi neblokirnega oddajnega protokola predvidevamo, da je zunanji sprejemnik vedno pripravljen na sprejem našega okvirja. To pomeni, da lahko softversko opravilo direktno vpisuje v oddajni register, saj ga bo asinhronski vmesnik poslal naprej ne glede na pripravljenost zunanjega sprejemnika da sprejme novi podatek. Pred vpisovanjem se mora softversko opravilo le prepričati, če je oddajni register prazen, saj bi v nasprotnem primeru prišlo do pretekanja vsebine registra.

16.5. Osveževanje LCD prikazovalnika

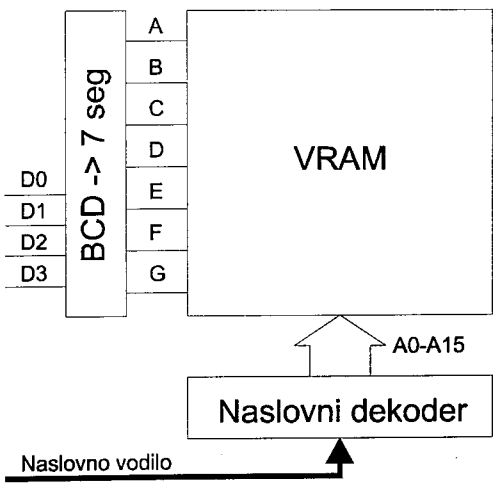
Matrični LCD prikazovalnik formata 4 x 26 segmentov zahteva za pravilno delovanje izmenične pravokotne signale točno določenih amplitud. S kombinacijo analognih signalov osveževalno vezje generira "vklopljeno" ali "izklopljeno" stanje posameznih segmentov. Da preprečimo utripanje, pa moramo celoten LCD prikaz neprekinjeno osveževati z dovolj visoko frekvenco - navadno več kot 100Hz. Tako se mora signal vrstice za 4-vrstični multipleksirani LCD zaslon spremeniti najmanj 400x v sekundi. Tudi tu nimamo na voljo softverskega opravila, ki bi se ponavljalo s tolikšno frekvenco.

Analogni signali, s katerimi poganjamo LCD prikaz, morajo biti točno določene oblike, s 50% razmerjem signal-ničla, da preprečimo enosmerno komponento, ki bi s časom poslabša delovanje prikaza.

Za neprekinjeno osveževanje vsebine LCD prikazovalnika potrebuje osveževalno vezje neposreden dostop do vsebine pomnilnika (DMA). Ker je pomnilnik nestandardne širine (26 bitov), se raje odločimo za lastno izvedbo video-pomnilnika velikosti 4 x 26 bitov. Žal so posamezni priključki na LCD prikazu multipleksirani popolnoma brez vsakega pravila; segmenti posameznih cifer so razpršeni dobesedno po vsem video pomnilniku. Takšna razpršenost pa za softversko izvedbo vpisovanja v video pomnilnik zahteva kompleksne razpršilne ("scatter") tabele, pri hardverski izvedbi video pomnilnika pa razpršilna tabela pomeni le, kateri bit (oziroma D flip-flop) naslovimo in kateri podatek vpišemo vanj.

Še vedno pa potrebujemo pretvornik iz 4-bitnega binarnega zapisa v 7-segmentnega. Izvedba s softversko ROM tabelo zahteva 16 besed v ROM pomnilniku, od vsake besede pa potrebujemo 7 bitov. V primeru ROM pomnilnika z besedo širine 16 bitov, bi bila takšnaizvedba nesmotrna. Tabelo lahko pri hardverski izvedbi realiziramo z optimizirano kombinacijsko logiko, kar skupaj z uporabniško načrtanim video-RAM dekodirnikom prinese precej prihranjenega prostora v pomnilniku.

Gonilnik LCD prikaza zato izvedemo kot popolnoma neodvisen proces, samostojno vezje, ki deluje povsem neodvisno od ostalega vezja. Navzven je vezje vidno le kot 16 izhodnih lokacij v katere vpisujemo 4-bitne binarne vrednosti, oziroma bitno-mapirane posebne znake na prikazu. Vse ostalo, vključno s pretvornikom iz 4-bitne dvojiške kode v 7-segmentno kodo in razpršilno tabelo, pa bo popolnoma nevidno za uporabniške programe. Tudi osveževanje bo potekalo samostojno, brez kakršnegakoli vpliva na izvajanje uporabniškega programa.



Slika 14: Dekoderji VRAM pomnilnika

17. Izvedba procesov s prekinitveno rutino

Prekinitve omogočajo signaliziranje dogodka mikroprocesorju. Ta ob signalu prekinitvi prekine trenutno aktivno softversko opravilo in skoči na podprogram, ki servisira nastalo prekinitev ("interrupt service routine").

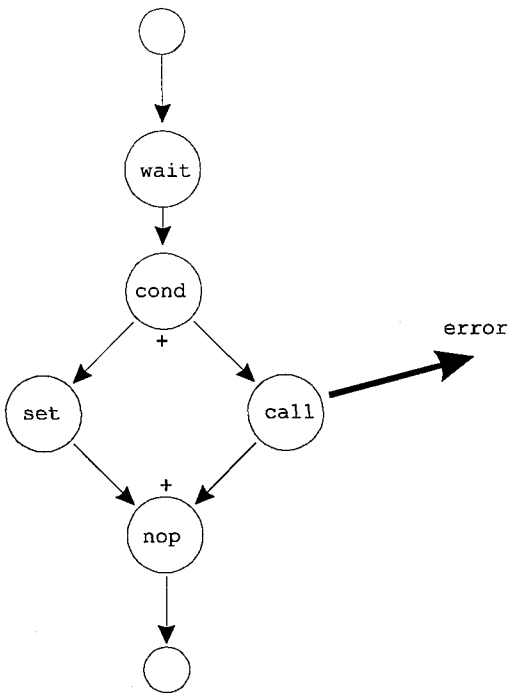
Izvedba procesa s prekinitvijo doda času izvajanja procesa t_P še čas odziva na prekinitev (t_i) in čas servisne rutine (t_s), s katero določimo indeks prekinitve in pokličemo ustrezno prekinitveno rutino.

$$\eta_{int\,r}(v) = t_i + t_s \qquad (17.1)$$

Ker so lokacije podatkov statično postavljene v pomnilnik, so prekinitvene rutine ne-rekurzivne, prav tako dva procesa ne moreta hkrati vstopiti v določeno rutino. Omejitvi zahtevata, da prekinitvene rutine in glavni program ne smejo imeti skupnih klicanih podprogramov, pomeni pa tudi, da v glavno prekinitveno rutino ne moramo vstopiti, če se prejšnji vstop vanjo še ni končal.

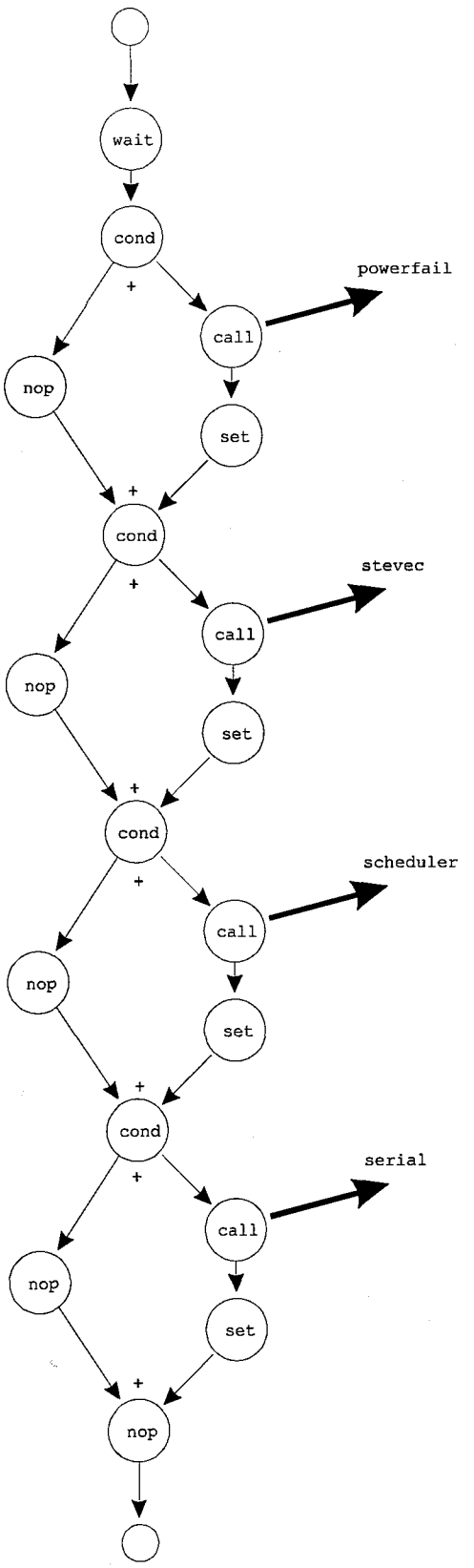
Temu pogoju zadostimo z blokado vseh prekinitev, dokler glavna prekinitvena rutina ni končana. Ker pa se prekinitev lahko pojavi tudi med izvajanjem glavne prekinitvene rutine, ko morajo biti prekinitve blokirane - vendar ne smemo možne aktivirane prekinitve preprosto ignorirati - zahteva servisiranje prekinitev samo z eno samo prekinitveno rutino posebno obdelavo prekinitvenih signalov. Prekinitev signaliziramo z zastavico, glavna rutina pa izvede zakasnjen klic ustrezne prekinitvene rutine. Seveda pa moramo tudi z zakasnjenim klicem zadostiti časovnim zahtevam izvajanih procesov.

Na Slika 15 je prikazan proces postavljanja zastavice posamezne prekinitve. Proces čaka na signal prekinitve, v primeru, če prejšnja prekinitev še ni bila servisirana pokliče rutino napake, drugače pa postavi zastavico prekinitve.



Slika 15: Postavljanje zastavice prekinitve

Glavna prekinitvena rutina čaka na spremembo katere od zastavic in glede na postavljeno zastavico kliče ustrezno prekinitveno rutino.



Slika 16: Glavna prekinitvena rutina

Izvedba sistema s prekinitvami vnese nedeterminiranost v čas izvajanje softverskih opravil. Izvajanje softverskega opravila, ki mora biti končano v času t_S , lahko do naslednjega preklopa opravila prekini ena ali več prekinitev z odzivom (η_{intr}), ki aktivirajo pripadajoči proces s časom izvajanja t_p. S tem se skrajša čas, ki je na voljo softverskemo uporavilu:

$$t_S' = t_S - \sum_N \left(\eta_{\text{int}\,r} + t_P\right) \qquad (17.2)$$

Želimo, da izvajanje softverskih opravil prekine čim manj prekinitev, čas izvajanja servisnega procesa pa naj bo čim krajši. Tiste procese, ki servisirajo veliko prekinitev z visoko frekvenco pojavljanja, je bolje vsaj delno izvesti v hardverski izvedbi. Vsaka prekinitev bo pravočasno obdelana, če bo čas med dvema zaporednima signaloma iste prekinitve krajši od vsote izvajalnih časov vseh prekinitvenih rutin:

$$t_{K2} - t_{K1} \geq \sum_N \left(\eta_{\text{int}\,r} + t_P\right) \qquad (17.3)$$

18. Sinhronizacija posameznih procesov

Sinhronizacija posameznih procesov predstavlja mehanizem za začasno uskladitev delovanja posameznih vzporedno delujočih sistemskih komponent pri medsebojni komunikaciji. Medtem, ko so procesi, realizirani v softveru, medsebojno sinhroni na nivoju mikroprocesorskih strojnih ukazov, zahteva prenos podatkov med softverskimi in hardverskimi komponentami, delujočih pri različnih hitrostih, eksplicitno sinhronizacijo. Poleg tega softverske komponente delujejo le navidezno hkrati - mikroprocesor izvaja dele softverskih procesov enega za drugim.

Sinhronizacijo med softverskimi in hardverskimi komponentami dosežemo na dva načina:

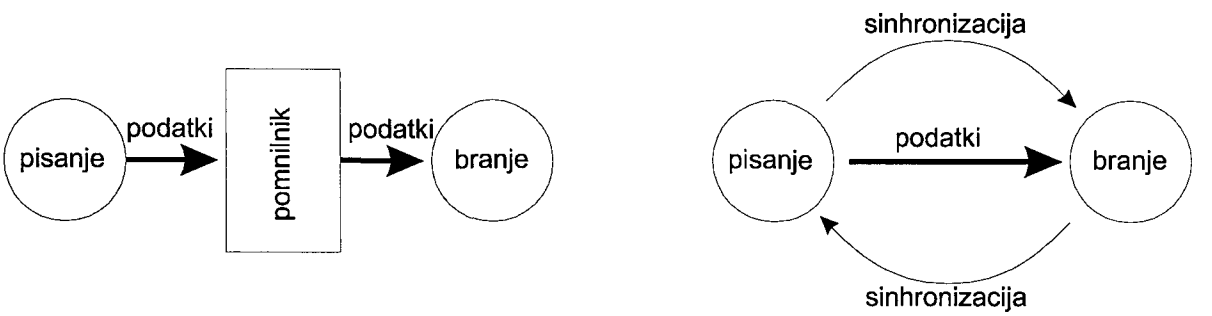
- polling: softversko opravilo preverja status hardverske komponente in glede na status opravi prenos podatka. Takšen način je primeren pri statičnih razdeljevalnikih softverskih opravil, saj softversko opravilo preverja status v točno določenih časovnih intervalih. Takšen način je primeren tudi za komunikacijsko omejene sisteme, kjer procesor večino časa čaka na V/I operacijo.

- dynamic scheduling: Dinamično razdeljevanje opravil temelji na prisotnosti podatka na vhodu. Če je podatek prisoten, procesor postavi ustrezno softversko opravilo v opravilno FIFO čakalno vrsto. S takšnim pristopom lahko ob minimalni porabi sistemskih sredstev obdelamo podatke, ki se z različnimi frekvencami naključno-nedeterminirano pojavljajo na vhodu.

Čakalno vrsto opravil lahko elegantno izvedemo s prekinitvami. Prekinitve z "zastavicami" signalizirajo glavni prekinitveni rutini, kdaj prekinitev čaka na obdelavo. Z blokado prekinitev zagotovimo, da se softverska opravila med seboj ne prekinjajo, prekinitvena rutina pa glede na postavljene zastavice sekvenčno izvaja ustrezna softverska opravila.

19. Podatkovni vmesnik med HW in SW komponentami

Podatke med hardverskimi in softverskimi komponentami lahko prenašamo v skupnem pomnilniku ("shared memory") ali kot prenos sporočil ("message passing"). Pri komunikaciji s skupnim pomnilnikom operacija pošiljanja spremeni vsebino spremenljivke na določeni pomnilniški lokaciji, operacija sprejema pa to spremenljivko prebere. Za operaciji ni nujno, da sta sinhronizirani, medtem ko pri pošiljanju sporočil velja, da se morata operaciji branja in pošiljanja predhodno sinhronizirati. Operaciji branja in pošiljanja pri prenosu sporočil se izvršita hkrati.



Deljeni pomnilnik

Pošiljanje sporočil

Slika 17: Podatkovni vmesnik med HW in SW

Zaporedje operacij pošiljanja in sprejema pri prenosu kompleksnejših sporočil poteka po določenem komunikacijskem protokolu. Takšen protokol je lahko blokirni ("blocking") ali neblokirni ("non-blocking"), blokira pa lahko operacijo pošiljanja, operacijo sprejema, ali pa obe. Blokirni komunikacijski protokol predstavlja zaporedje preprostejših operacij s komunikacijskimi vrati, potrebno sinhronizacijo pa dosežemo z dodatnim kontrolnim signalom.

Neblokirna operacija oddajanja zahteva, da operacija sprejemanja sprejemnika poteka z isto hitrostjo izvajanja. V primeru, ko sprejemnik prejete informacije ne zmore dovolj hitro obdelati, uporabimo posebno strukturo - FIFO vmesni pomnilnik (oziroma podatkovno čakalno vrsto). Pri sprejemu znakov po serijskem vmesniku se podatki pojavljajo z maksimalno hitrostjo 960 znakov/s; softversko opravilo, ki prejete znake obdela pa se ponovi le 125x v sekundi. Med tem časom nove podatke shranimo v softversko izveden FIFO pomnilnik.

20. Simulacija procesov mikroprocesorja

S simulacijo preverimo delovanje skupka procesov, ki tečejo navidezno hkrati na mikroprocesorju. Za razliko od simulacije paralelnih procesov, kjer je vsak proces tekel na svojem viru, tu več procesov deli en vir - mikroprocesor, zato moramo uskladiti njihovo medsebojno delovanje. Poleg tega so procesi dinamično aktivirani - odvisno od prisotnosti podatka na vhodnih vratih procesa. Zanima nas, če bo mikroprocesor pri maksimalni obremenitvi še zmogel v realnem času servisirati vse procese.

Hkrati lahko s simulacijo opazujemo notranja stanja procesov, postavljanje in podiranje zastavic prekinitev, spremljamo lahko potek blokade prekinitev, verilog pa nam s svojimi sistemskimi rutinami omogoča tudi tvorjenje in nadzor stanja FIFO vmesnika.

Vsako prekinitev lahko opremimo s pastjo, ki se sproži, ko prekinitev ni bila pravočasno servisirana. Ker mora naš sistem pravočasno servisirati vse prekinitve, sprožena past pomeni, da je med delovanjem sistema prišlo do napake, oziroma se softversko opravilo ni končalo pravočasno. V takšnem primeru naj mikroprocesor izvede ponovni zagon celotnega sistema.

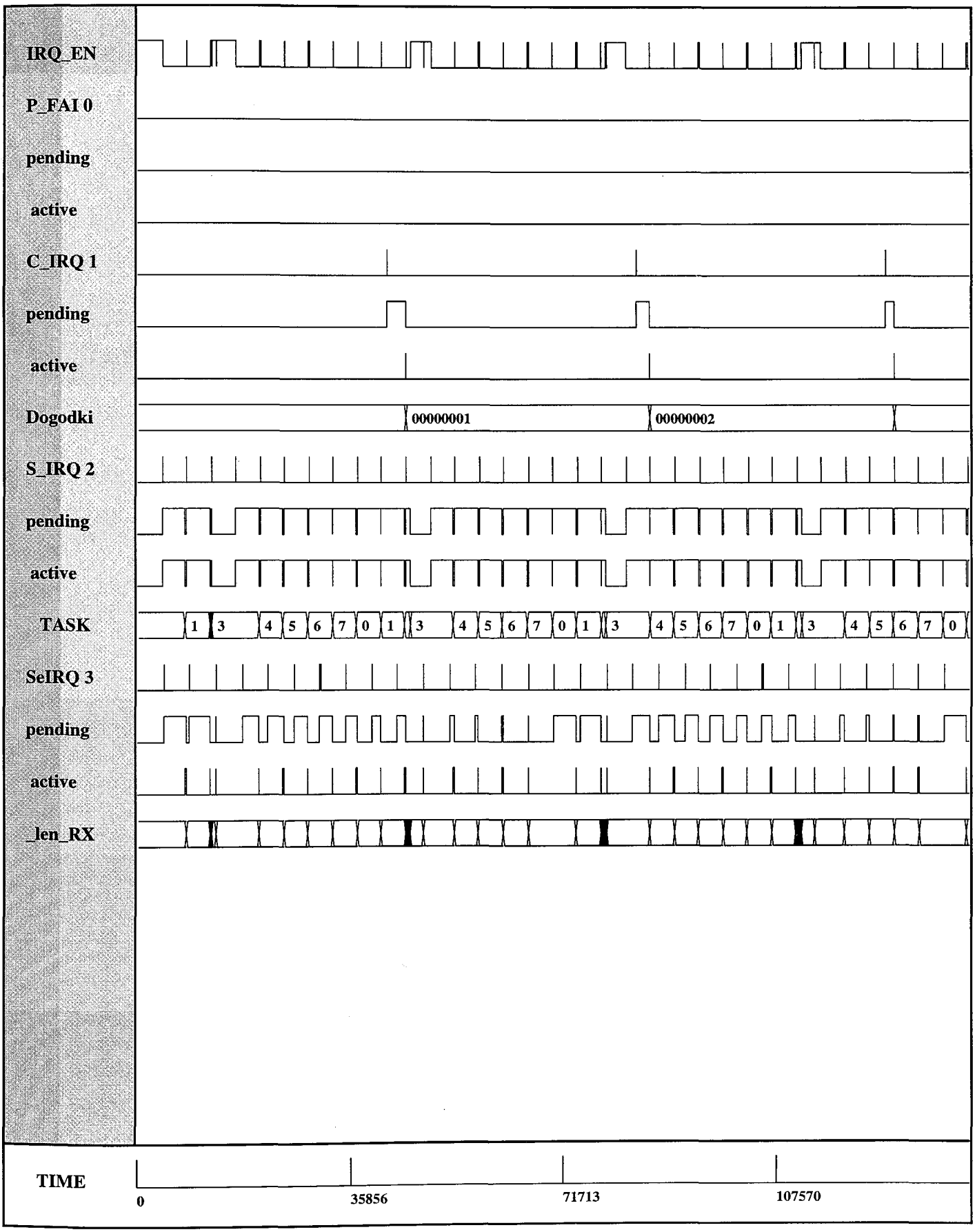
Rezultat simulacija procesov v mikroprocesorju je na Slika 18, HDL opis procesov, skupaj s testnim programom pa v prilogi 2 na strani 103.

(na naslednjih straneh)

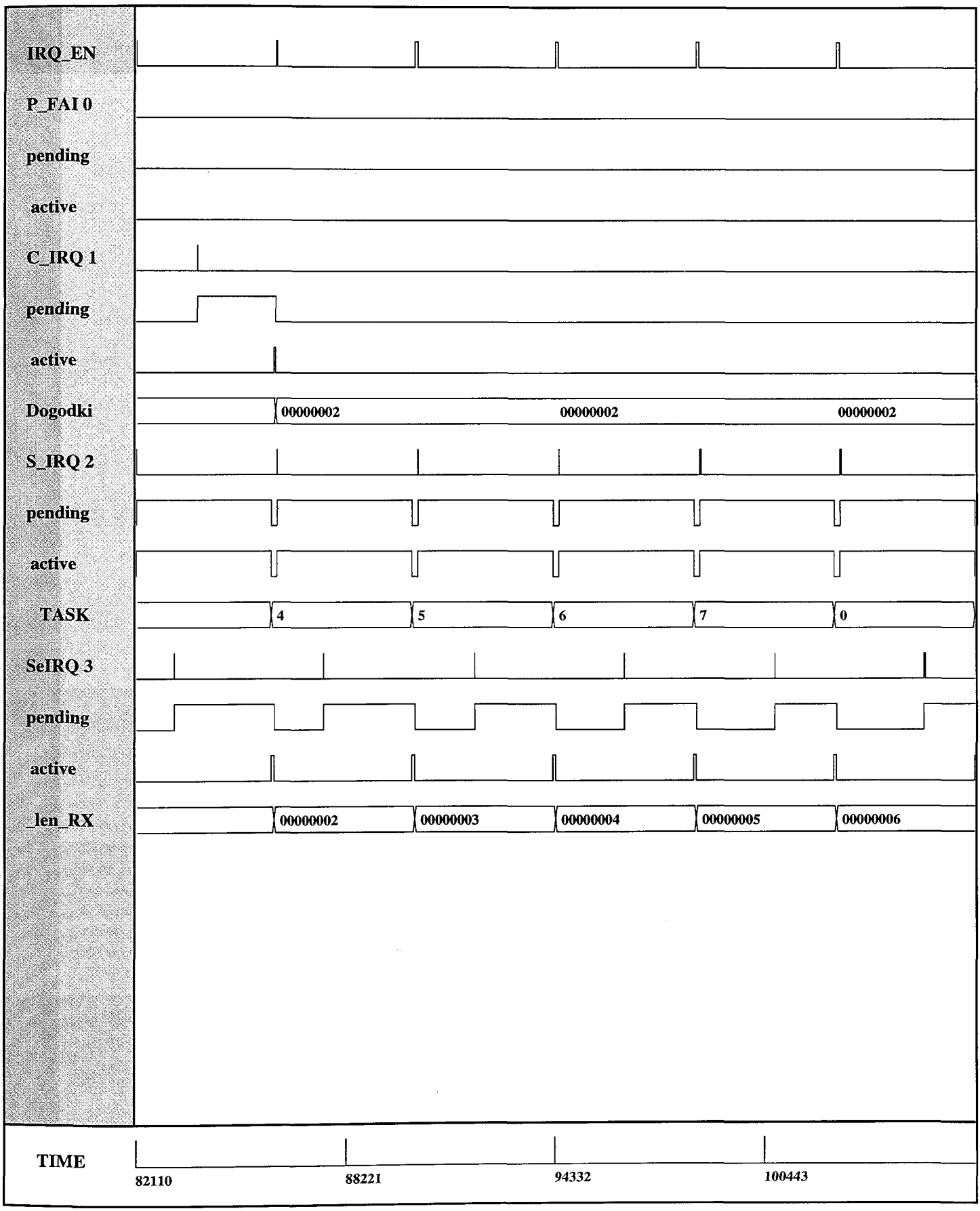
Slika 18: Simulacija procesov v mikroprocesorju

a) simulacija procesov v mikroprocesorju
b) detajl simulacije procesov v mikroprocesorju
c) detajl procesa sprejema po serijskem vmesniku

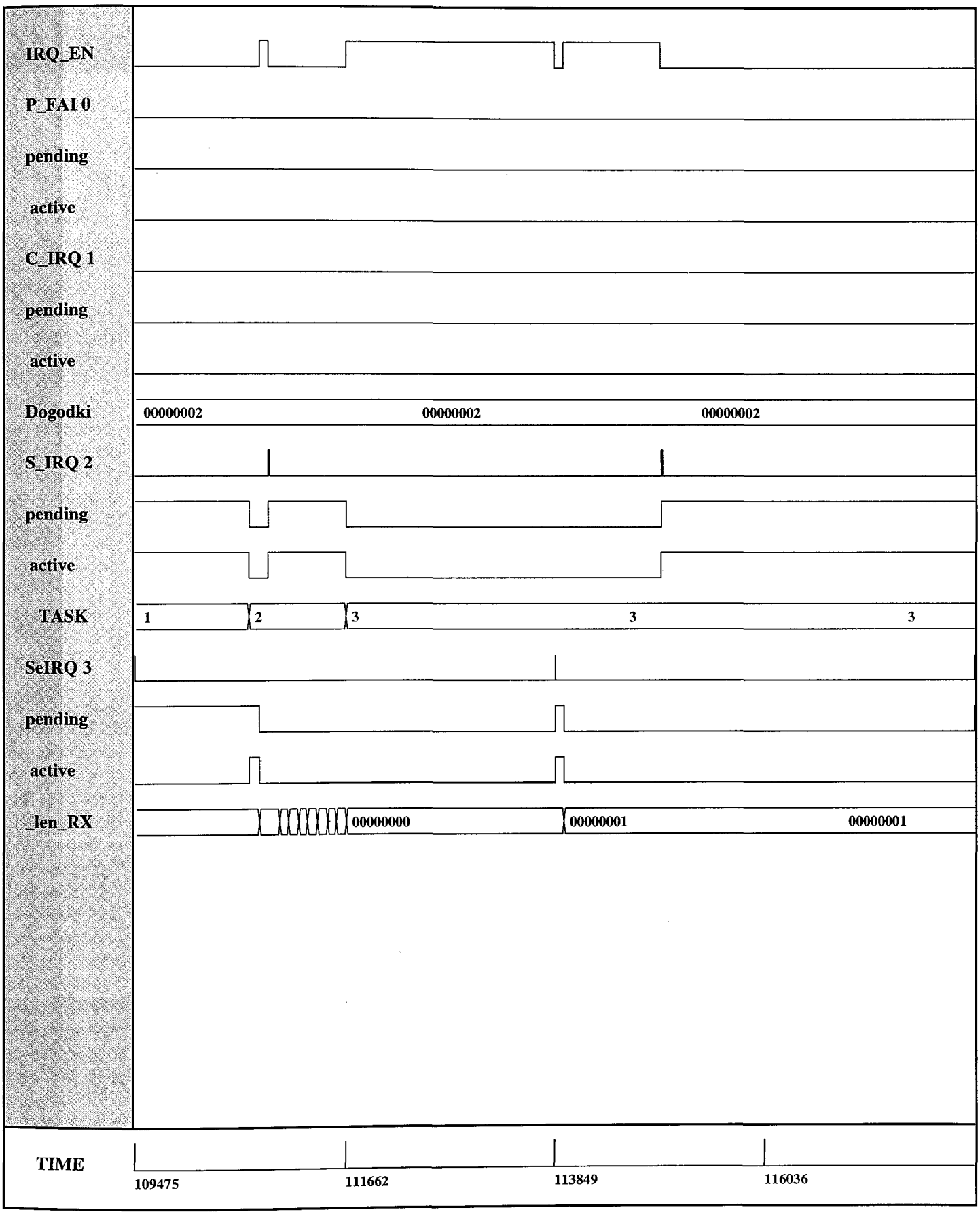
Header: simulacija sistema		
User:		
Date: Mar 13, 2000 14:23:33	Time Scale From: 0 To: 140000	Page: 1 of 1



Header: detajl simulacije sistema	
User:	
Date: Mar 13, 2000 14:20:02	Time Scale From: 82110 To: 106555 Page: 1 of 1



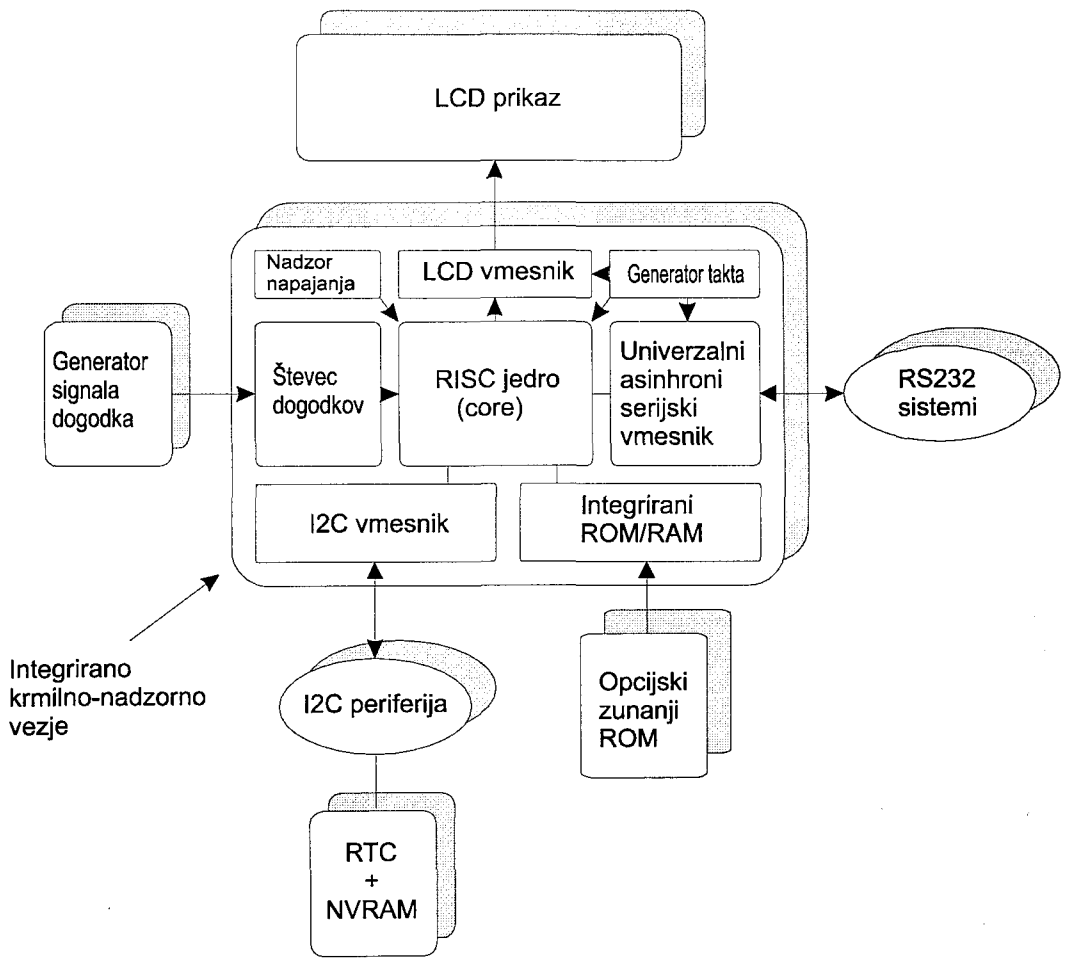
Header: praznenje RX pomnilnika		
User:		
Date: Mar 13, 2000 14:27:02	Time Scale From: 109475 To: 118223	Page: 1 of 1



21. Izvedba hardverskih sklopov

Integrirano vezje sestavljajo naslednji hardverski sklopi:

- mikroprocesorsko jedro z ustreznim ROM in RAM pomnilnikom
- sklop serijskega vmesnika (z nizkonivojskimi gonilnimi rutinami)
- sklop LCD gonilnika
- vezje nadzora napajanja z reset generatorjem
- I_2C V/I vmesnik (z nizkonivojskimi gonilnimi rutinami)

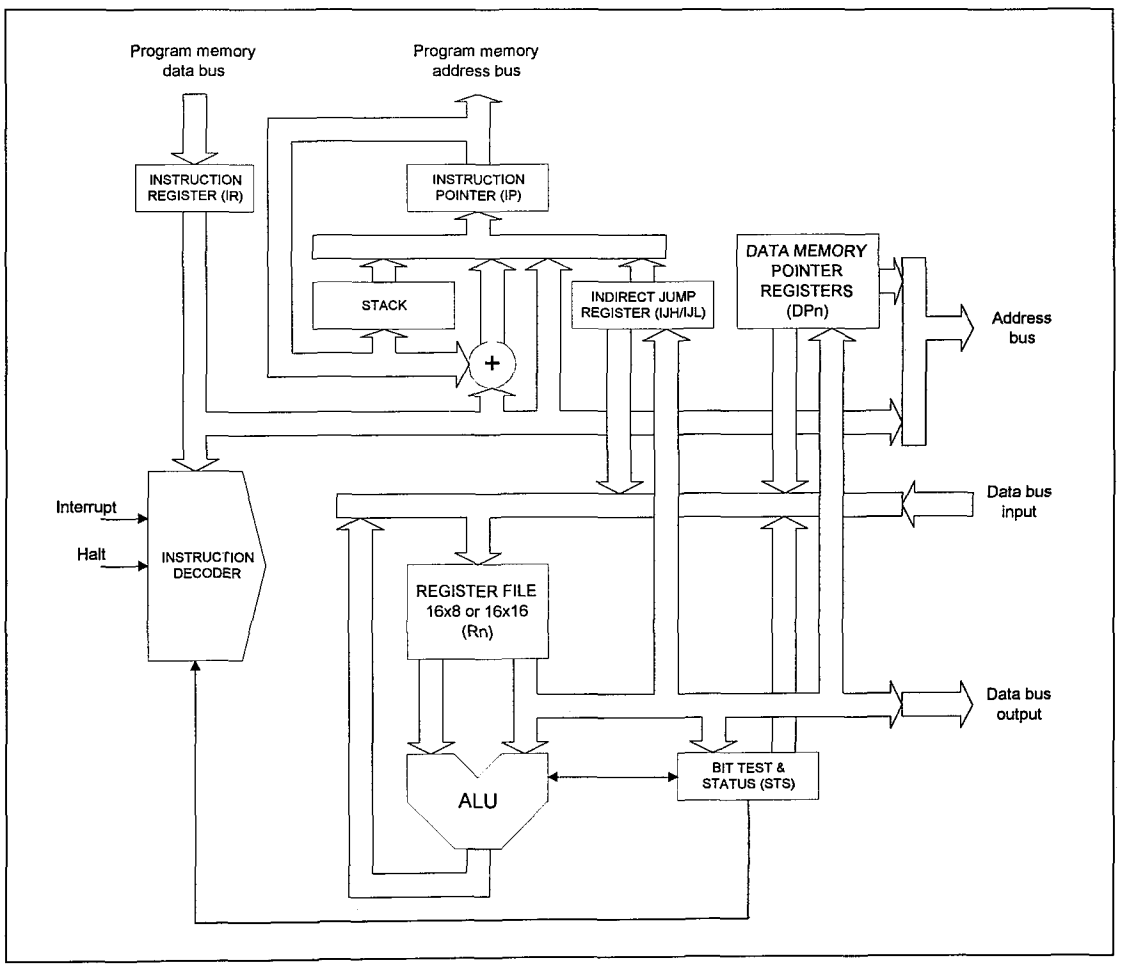


Slika 19: Blokovna shema sistema

21.1. Mikroprocesorsko jedro integriranega vezja

Za programabilno jedro izberemo ustrezen IP blok, kateri mora zadostiti potrebam programske opreme. Proizvajalci nam ponujajo široko paleto mikroprocesorskih jeder, od 4-bitnih mikrokontrolerjev, pa vse do 32 bitnih RISC mikroprocesorjev tipa ARM ali SPARC. Mikroprocesorsko jedro izberemo glede na kompleksnost programske opreme, potrebe naše aplikacije po pomnilniku, željene frekvence delovanja, ne nazadnje pa tudi glede porabe energije, velikosti končne geometrije, kvalitete razvojnih orodij in uporabniške podpore.

Mikroprocesorska jedra so na voljo v obliki izvorne kode sintesizabilnega funkcionalnega RTL (Register Transfer Logic) modela, katerega lahko s HDL prevajalnikom prevedemo v ciljno arhitekturo standardnih celic določenega proizvajalca. Takšen model, tudi "soft-core" model opiše funkcionalnost jedra, medtem ko je časovni potek signalov odvisen od ciljne arhitekture.



Slika 20: Shema mikroprocesorskega jedra

Za programabilno jedro integriranega sistema izberemo 8 bitni mikrokontroler uRISC MTC-8308 proizvajalca Nordic VLSI. MTC-8308 je načrtan posebej za zamenjavo kompleksne digitalne kontrolne in procesne logike v uporabniško načrtanem integriranem vezju s programabilnim elementom. Takšen pristop omogoča opis funkcionalnosti integriranega vezja na višjem nivoju - kot programski opis funkcionalnosti, shranjen v pomnilniku integriranega vezja, namesto opisa funkcionalnosti na nivoju sheme povezav posameznih celic integriranega vezja, oz. na nivoju sintesizabilnih opisov. Takšen pristop omogoča krajše načrtovalske cikle, saj se načrtovalec lahko osredotoči na pisanje in preverjanje programa, namesto na načrtovanje digitalne logike.

MTC-8308 je mikrojedro z reduciranim naborom ukazov. Glavni razredi ukazov predstavljajo dostop do pomnilnika (direktni in indirektni), aritmetični in logični ukazi, brezpogojni skok, pogojni preskok ukaza, večina ukazov pa se izvrši v enem urinem taktu.

Mikrojedro ima posebno podatkovno in naslovno vodilo za dostop do ROM pomnilnika s programom, posebno podatkovno vodilo do RAM pomnilnika, naslovno vodilo RAM pomnilnika pa je deljeno z naslovnim vodilom V/I enot. 12 bitno programsko naslovno vodilo omogoča naslavljanje 4k besed programskega pomnilnika širine 16bitov. Maksimalna velikost RAM pomnilnika je 192 bytov, naslovni prostor pa si deli s 64 V/I lokacijami. Na vrhu RAM naslovnega prostora je 5 specialnih registrov: statusni register (STATREG), prekinitveni register (INTREG), register indirektnega podatkovnega naslova (SMAR) in indirektni register skoka (JREG). V Tabela 2 je predstavljena pomnilniška karta ("memory map") mikrojedra.

Naslov	Funkcija
0xFF 0xFE 0xFD 0xFC 0xFB	JREGH (4 biti) JREGL SMAR INTREG STATREG
0xFA ... 0xC0	V/I
0xBF ... 0x00	RAM

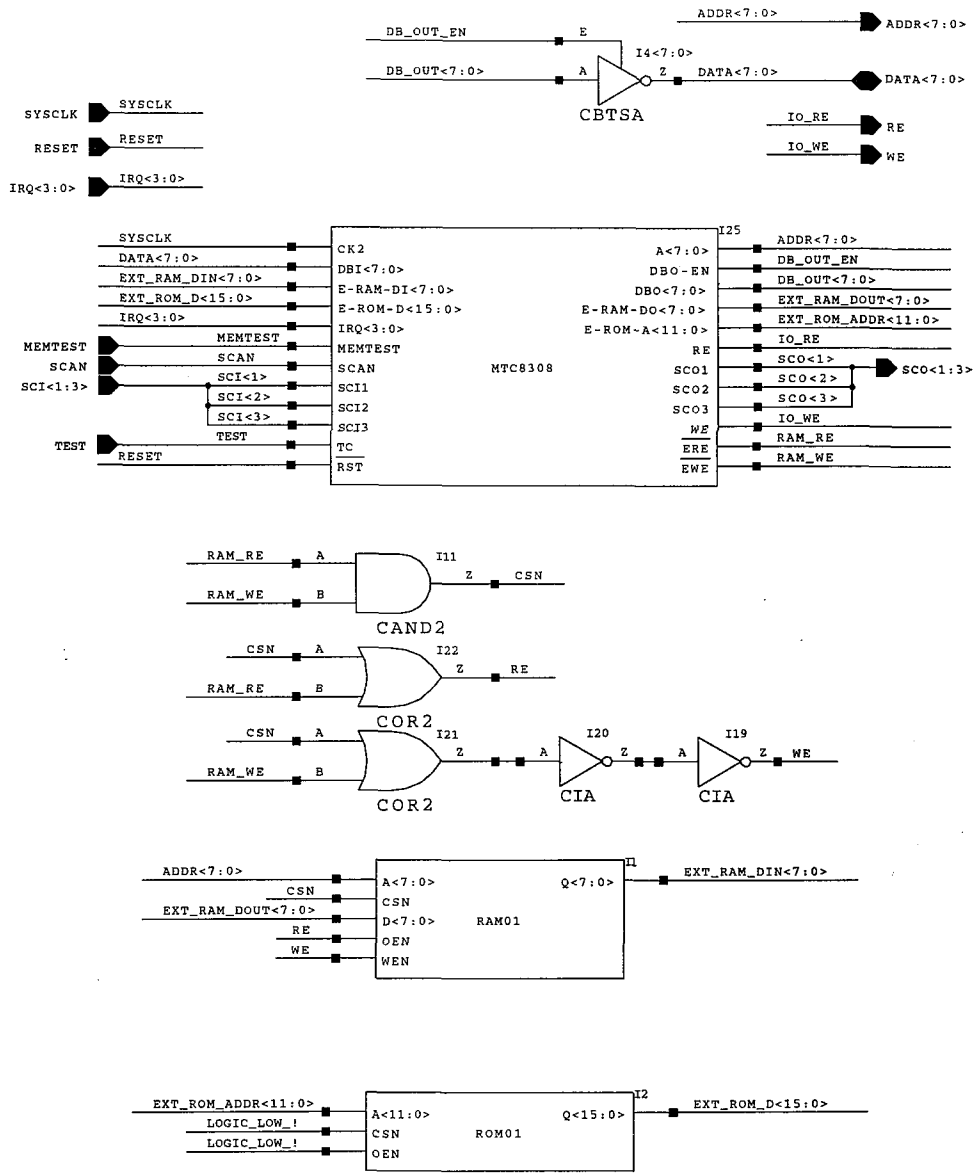
Tabela 2: Pomnilniška karta mikroprocesorja

Pomnilniki spadajo med regularne strukture, katere generiramo z različnimi generatorji - "silicon compilers". Takšen generator sestavi optimizirano geometrijo velike gostote iz knjižnice osnovnih celic ("leaf cells"). Takšna generirana končna geometrija je optimizirana v North-South in East-West smereh, kar pomeni optimalno izrabo prostora na integriranem vezju.

Regularne strukture, med katere poleg ROM in RAM pomnilnikov štejemo tudi množilnike, FIFO strukture, itd, generiramo za točno določeno ciljno tehnologijo, zato običajno generator glede na željene parametre generira tudi dokument s točnim časovnim potekom signalov bralnega ali pisalnega cikla. Hkrati generator karakterizira celico glede na višino in širino, standardno obremenitev celice na vhodu in izhodu, porabo in občutljivost izhodov na obremenitev (priloga 26.3 in 26.4).

Poleg optimizirane končne geometrije nam generator generira tudi funkcionalni opis strukture, s katerim lahko simuliramo obnašanje regularne strukture v povezavi z ostalimi elementi. Ker poznamo končno tehnologijo, poznavanje parametrov tehnologije omogoča nadzor nad potekom posameznih signalov znotraj regularne strukture. Zato je v funkcionalni opis vstavljenih več "pasti", ki se sprožijo ob kršitvi kateregakoli časovnega parametra. Na takšen način lahko načrtamo vezno logiko, ki bo omogočila optimalno medsebojno delovanje posameznih blokov.

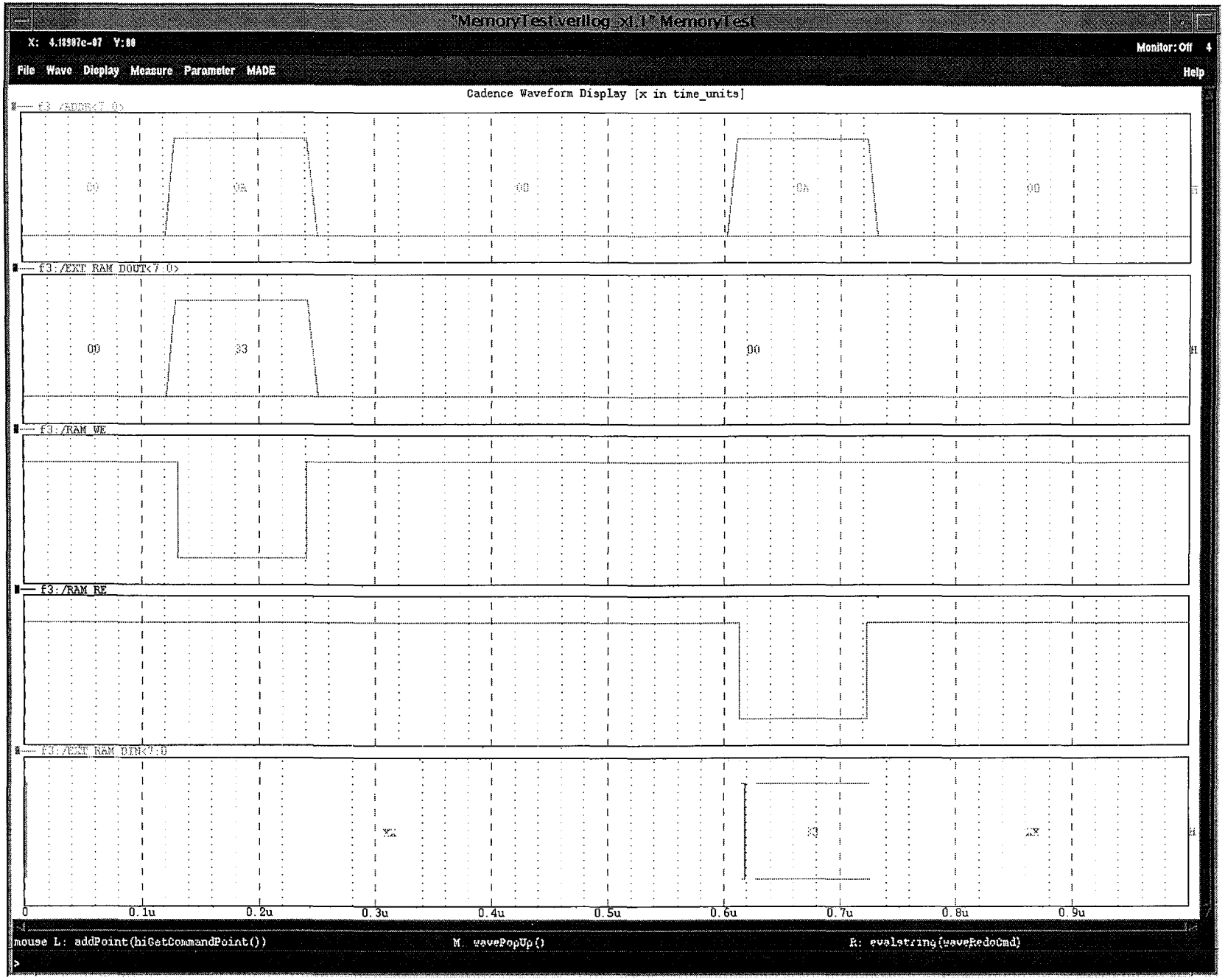
Na Slika 21 je programabilno jedro integriranega vezja, sestavljeno iz mikrojedra, RAM pomnilnika velikosti 256 bytov in ROM pomnilnika velikosti 4k 16 bitnih besed. Pravilnost delovanja vezne logike in medsebojnih povezav preverimo z vpisom in branjem poljubne vrednosti v RAM pomnilnik ter branjem programskega ukaza iz ROM.



Slika 21: Shema programabilnega jedra

(na naslednji strani)

Slika 22: Simulacija bralno-pisalnega cikla


"MemoryTest.verilog_xl.1" MemoryTest
X: 4.18987e-07 Y: 80
Monitor: Off 4
File Wave Display Measure Parameter MADE
Help
Cadence Waveform Display [x in time_units]
f3:/ADDR<7:0>
00
0E
00
0A
00
f3:/EXT_RAM_DOUT<7:0>
00
33
00
f3:/RAM_WE
f3:/RAM_RE
f3:/EXT_RAM_DIN<7:0>
XX
33
XX
0
0.1u
0.2u
0.3u
0.4u
0.5u
0.6u
0.7u
0.8u
0.9u
mouse L: addPoint(hiGetCommandPoint())
M: wavePopUp()
R: evalstring(waveRedoCmd)
>

Posamezne I/O bloke aktiviramo s pomočjo dekoderja naslovov. Ker si I/O bloki delijo naslovni prostor z RAM pomnilnikom, potrebujemo dekodiranje popolnega naslovnega prostora.

Poseben problem naslovnega dekoderja predstavlja sinhronizacija dekodiranega naslova na procesorjev bralni, oziroma pisalni cikel. Ker se signal dekodiranega naslova uporablja za aktiviranje zapahov naslovnega in podatkovnega vodila posameznih blokov, mora biti signal dekodiranega naslova povsem brez t. im. "glitchev", nezaželenih kratkih impulzov, ki so posledica različnih notranjih časovnih zakasnitev logičnih vezij.

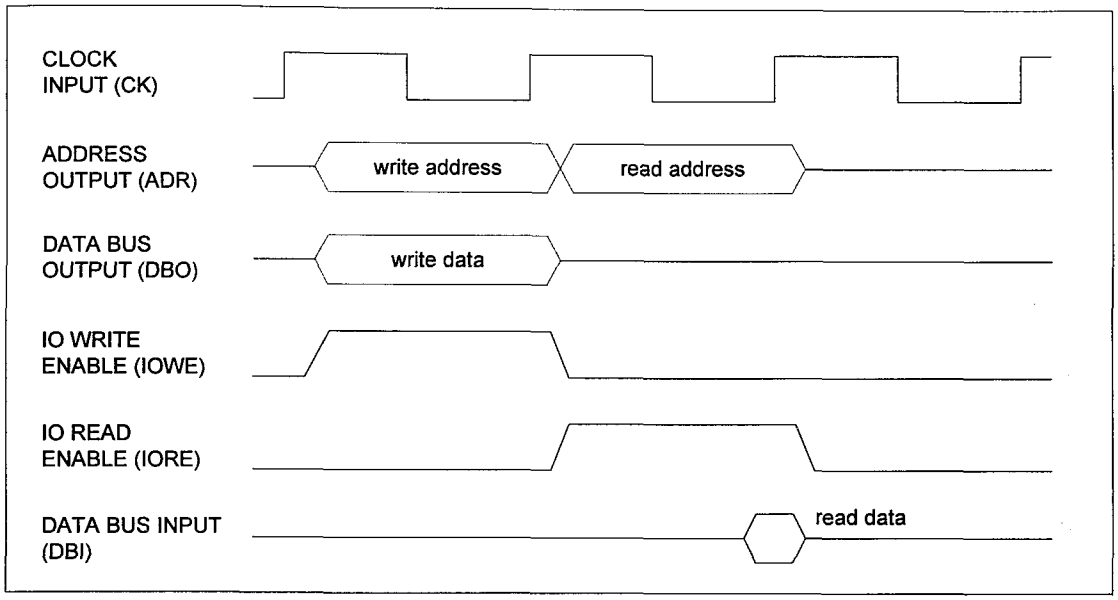
Problem "glitchev" rešimo z enonivojsko logično strukturo z večvhodnimi NOR vrati. NOR vrata so po DeMorganovem teoremu z invertiranimi vhodnimi signali funkcijsko ekvivalentna AND vratom, vendar so brez izhodnega CMOS inverterja in zato hitrejša.

Naslov	Funkcija
0xD0 - 0xDF	LCD prikaz, VRAM
0xC8 - 0xCF	I2C vodilo
0xC0 - 0xC7	UART

Tabela 3: Pomnilniška karta sistema

Pri vpisu v željeni I/O blok mikroprocesor stabilizira podatke na podatkovnem in naslovnem vodilu po določeni zakasnitvi za rastočim signalom ure, hkrati pa pisalni tip I/O dostopa z "IO WRITE ENABLE" signalom. Podatki na vodilih so zagotovo stabilni med negativno polperiodo sistemske ure, zato za zanesljiv prenos podatkov v IO blok signal dekodiranega naslova sinhroniziramo z negativno polperiodo sistemske ure.

Mikroprocesor bere podatke ob koncu bralnega cikla, zato sinhronizacija dekodiranega naslova med bralnim ciklom ni potrebna.

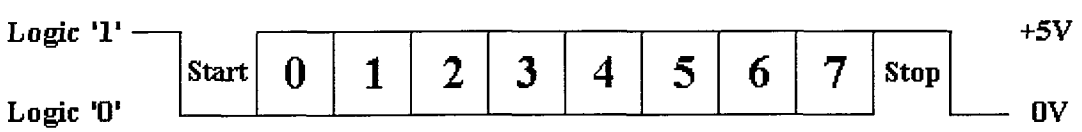


Slika 23: Dostop do I/O vrat

21.2. Sklop serijskega vmesnika

Serijsko komunikacijo realiziramo z univerzalnim asinhronim oddajno-sprejemnim vezjem (UART). Takšno vezje lahko hkrati komunicira po enem sprejemnem in enem oddajnem kanalu: lahko po dveh ločenih žicah (+skupna masa) ali pa optično po dveh LED<->opto-tranzistor parih.

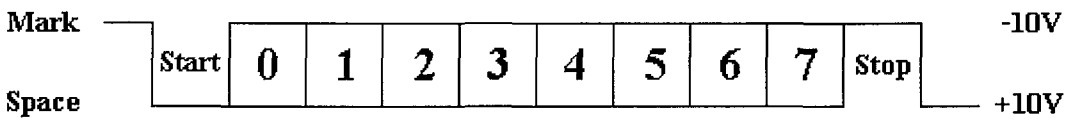
Asinhrona komunikacija pomeni, da je signal s podatki poslan brez signala ure, zato je vsak začetek okvirja s podatki sinhroniziran s start bitom, takt notranje časovna reference na sprejemni strani pa mora biti za uspešen prenos podatkov enak oddajnemu taktu.



Gornji diagram prikazuje signal na priključkih UART vezja v običajnem 8N1 formatu: 8 podatkovnih bitov, brez paritete in 1 stop bit. Neaktivna linija je postavljena v logični 1 ("Mark state"). Prenos se prične s startnim bitom (logična 0, "Space state"), po vrsti pa mu eden za drugim sledijo posamezni biti okvirja, od najmanjšega (LSB) do največjega (MSB). Prenos se zaključi s stop bitom, oziroma z logično 1.

Diagram kaže, da se prenos takoj za stop bitom nadaljuje z novim okvirjem, torej je bit, ki sledi stop bitu, startni bit naslednjega podatka. V nasprotnem primeru, ko se tok podatkov ustavi, pa mora linija ostati v logični 1 - "Idle state". Stanje na liniji, ko linija ostane v logični "0" za dalj, kot je čas enega okvirja, sprejemnik razpozna kot "Break" signal. Break signal lahko razpoznamo kot napako okvirja pri sprejeti vrednosti 00000000_2.

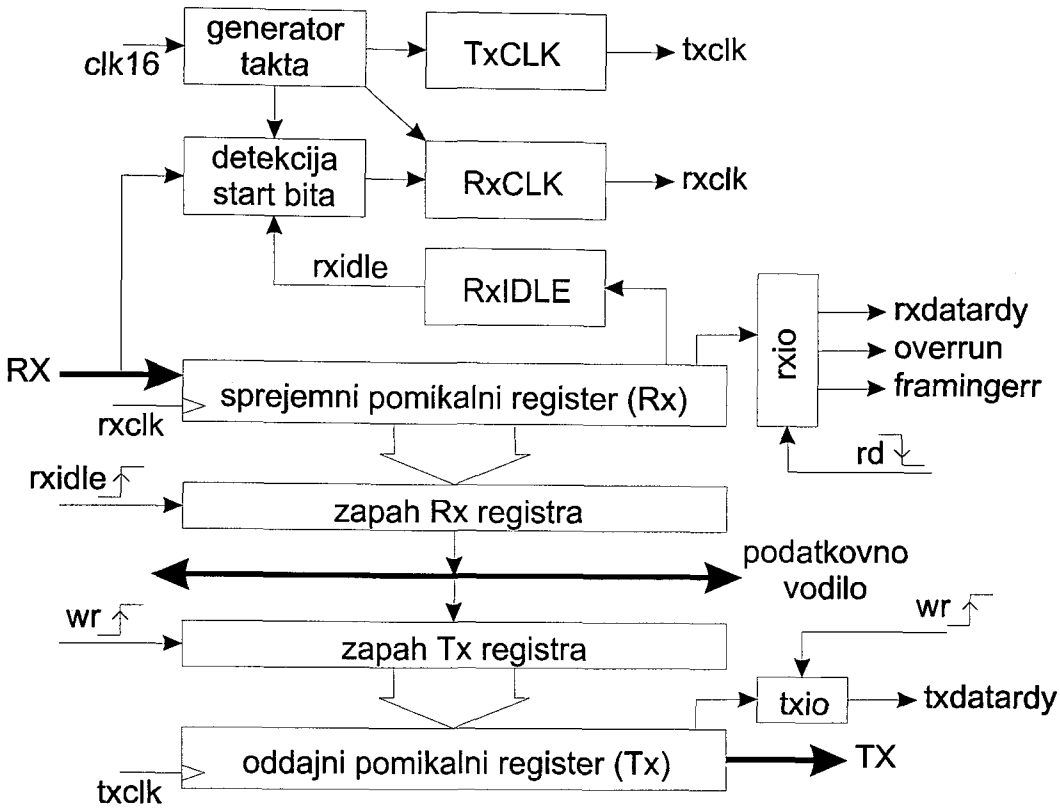
Zgornji diagram velja samo za signale direktno na izhodih UART vezja. RS-232 predpis pa zahteva za "Space" (logična 0) vrednost med +3 in +25V ter za "Mark" vrednost med -3 in -25V. Vsaka vrednost med -3 in +3 je nedefinirana, torej moramo signal pretvoriti z RS232 pretvornikom nivojev.



Za serijsko komunikacijo po optičnem vmesniku podobno velja, da "Idle" in "Mark" stanje pomenita ugasnjemo LED diodo, "Space" pa prižgano.

Izvedba UART vezja (Slika 24) je zgrajena okrog sprejemnega in oddajnega pomikalnega registra. Izvedba vsebuje še enonivojski izravnalni pomnilnik ("buffer"), generatorjem takta, detekcijo startnega bita, generator prekinitev, ter vezje za detekcijo napak na sprejetem podatku. Ker UART vezje deluje z nižjim urnim taktom kot mikroprocesorsko jedro, moramo vezje priključiti na sistemska vodila preko zapahov in gonilnikov vodila ustreznih bitnih širin. Zapahe in gonilnike krmilimo z dekodiranim signalom iz naslovnega dekoderja.

Celotna shema UART vmesnika je v prilogi 26.5.



Slika 24: Shema UART vmesnika

21.2.1. Detekcija startnega bita in generator sprejemnega takta

V bloku **clkgen_rx** združimo detekcijo startnega bita in generator sprejemnega takta. Želimo, da se prenos aktivira šele takrat, ko je nivo startnega bita prisoten dovolj časa, s čemer zmanjšamo občutljivost vezja na zunanje motnje, ki so lahko pristone na liniji. Sprejemno linijo vzorčimo s 16x frekvenco takta serijskega prenosa, vezje pa naj detektira startni bit po 8 zaporednih vzorcih.

Predvidevamo, da je sprejemno vezje neaktivno, kar vezje signalizira z nizkim nivojem rxidle signala. S pomočjo D-FlipFlop elementa detektiramo prehod sprejemne linije iz logične "0" v "1", v kombinaciji z rxidle signalom pa postavimo izhod JK-FlipFlop[2] vezja (wait_startb) v visok logični nivo.

[2] JK-FlipFlop vezje se obnaša kot boolova spremenljivka. Izhod vezja se postavi v logično "1" samo pri prehodu J vhoda iz "0" v "1" , logično "0" na izhodu pa dobimo pri prehodu K vhoda iz logične "0" v logično "1" postavi v logično 1. Na ta način lahko izvedemo pogoje tipa:
IF (pogoj_1) then (izhod) = TRUE;
IF (pogoj_2) then (izhod) = FALSE;

Vezje detekcije s tem preklopi v stanje čakanja na startni bit, kar signalizira s signalom wait_startb. Ta signal sprosti štetje 4-bitnega števca s paralelnim vpisom, na katerega izhod je vezana ura sprejemnega takta rxclk. Števec nadaljuje s štetjem, dokler je wait_startb signal v logični "0". Ker je vsak sprejeti bit 16x vzorčen, rxclk aktiviramo po osmih vzorcih. S tem detektiramo bit na njegovi sredini ter tako zmanjšamo vpliv razlik med frekvenco oddajne iin sprejemne ure.

Če v wait_startb stanju sprejemna linija preklopi v logično "1", se v števec znova naloži začetna vrednost. Tako števec resetiramo v začeno stanje in proces detekcije se ponovi.

Na MSB izhod števca je vezana ura sprejemnega takta rxclk, ta pa s prehodom iz logične "0" v logično "1" povzroči premik sprejemnega registra in s tem prehod rxidle signala iz logične "1" v logično "0". Logika v detekciji startnega bita poskrbi, da števec šteje, dokler se prenos okvirja ne konča - takrat rxidle signal preide v logično "1".

21.2.2. Sprejemni pomikalni register

Sprejemni pomikalni register je izveden v bloku **rxshift**. Načrtovalski program nam nudi možnost, da ponavljajoče vezave narišemo v "vektorski obliki", v tem primeru ena povezava, opisana v obliki <net1, netN> predstavlja skupek N povezav, ena komponenta v takšnem opisu pa predstavlja N komponent.

Sprejemni premikalni register naj ima v neaktivnem stanju vrednost 11111111_2, na desni strani pa ga zaključimo z rxstop registrom, ki naj ima v neaktivnem stanju vrednost "0". Z vrednostjo v tem registru bomo v bloku **rxidle_gen** detektirali konec prenosa, saj bo pri premikanju vsebine registra v desno to prva vrednost z logično "0" v zadnjem bitu serijskega sprejemnega registra. Po prenosu celotnega okvirja bo v serijskem sprejemnem registru vrednost sprejete besede, v registru rxstop pa vrednost stop bita serijskega prenosa, zato ga bomo uporabili za detekcijo napake okvirja.

Naloga **rxidle_gen** bloka je, da ob prvem prehodu signala rxclk iz logične "0" v logično "1" dvigne signal rxidle na logično "1", ter po prenosu vseh 10 bitov okvirja spusti signal nazaj na logično "0".

21.2.3. Vezje za detekcijo napak na sprejetem podatku

Prehod rxidle signala iz logične "0" v logično "1" ob koncu prenosa uporabimo tudi za osvežitev statusnih signalov sprejetega okvirja v bloku **rxio**. Prehod detektiramo z D-FlipFlopom, postavimo zastavico prejetega podatka rxdatardy, glede na vrednost prejetega stop bita postavimo zastavico okvirja framingerr, glede na prejšnjo vrednost rxdatardy pa zastavico preteka overrun. Hkrati prenesemo vrednost rxreg v enonivojski izravnalni pomnilnik rxhold, ter tako sprostimo rxreg za sprejem naslednjega okvirja. Zastavice zbrišemo s tem, da preberemo vrednost registra rxhold, kar signaliziramo s prehodom read signala iz logične "0" v logično "1".

21.2.4. Generator oddajnega takta

Signal ure sprejemnega takta txclk v bloku **clkgen_tx** izvedemo s 3-bitnim binarnim števcem. Števec šteje navzgor in vsakokrat, ko se števec prelije (oziroma skoči v vrednost 000_2) preklopimo txclk iz logične "0" v logično "1" in obratno. Na ta način delimo sistemsko uro UART vezja s 16.

21.2.5. Oddajni pomikalni register

Oddajni pomikalni register UART vmesnika **txshift** izvedemo podobno, kot sprejemni pomikalni register. Tudi tukaj uporabimo vektorski opis registra, konec prenosa pa detektiramo z dodatnim registrom txtag z vrednostjo "0" za MSB bitom oddajne besede.

V neaktivnem stanju je signal _txdatardy v logični "0", ob vpisu podatka v izravnalni register pa _txdatardy preide v logično "1". Če je prejšnji okvir že zapustil oddajni premikalni register (signal txdone je v logični "1"), signal init preide v logično "1". S tem signalom preko multiplekserja v txreg paralelno vpišemo vrednost izravnalnega registra txhold, v txtag register vpišemo logično "1", v start register pa logično "0", ki pomeni začetek prenosa. Signal txdone zaradi vpisa logične "1" v txtag register preide v logično "0".

Pri premikanju vrednosti pomikalnega registra v desno se v txtag register vpisuje vrednost logične "0", dokler vrednost "1" iz init cikla txtag registra ne doseže registra start. Signal txdone, ki ga generira NOR vezje takrat preide v logično "0" in s tem detektira konec prenosa.

Poskrbeti moramo, da je izhodni signal neaktivnega oddajnega dela UART vmesnika vedno v "Idle" stanju, saj bi v nasprotnem primeru sprejemnik na sprejemni strani napačno razpoznal začetek okvirja, ali celo "Break" signal.

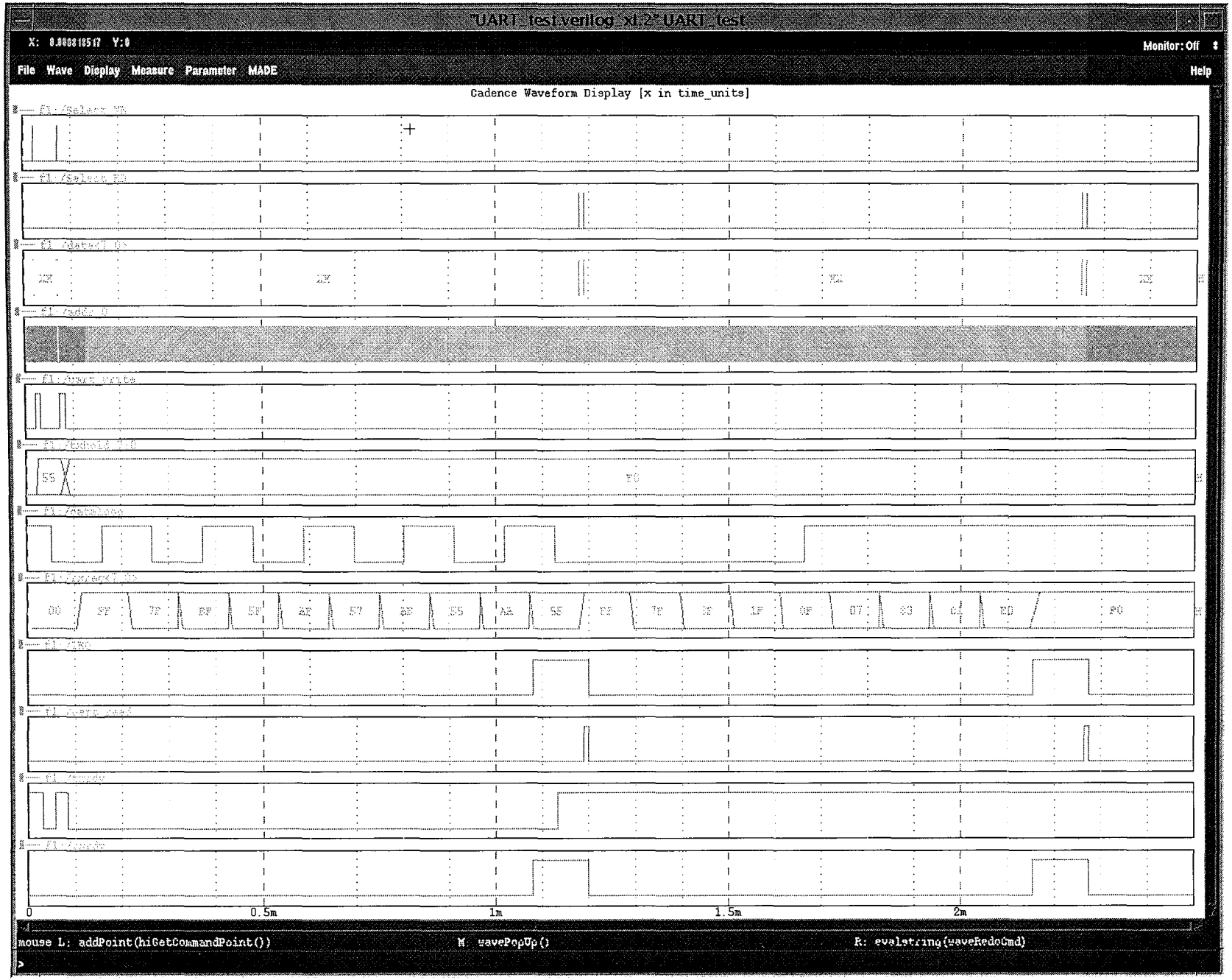
21.2.6. Detekcija razpoložljivosti oddajnega registra

Z blokom **txio** krmilimo invertirani signal razpoložljivosti oddajnega registra _txdatardy. Postavimo ga ob prehodu signala mikroprocesorskega vpisa write iz logične "1" v logično "0". Ker izravnalni register vpišemo ob prehodu signala write iz logične "0" v logično "1", lahko na ta način začnemo prenos okvirja preko serijske inije takoj, ko mikroprocesor vpiše podatek v UART vezje.

Signal _txdatardy postavimo v logično "0", ko premikalni register premakne celotni vpisani podatek.

(na naslednji strani)

Slika 25: Simulacija sprejemnega in oddajnega cikla UART vmesnika


"UART_test.verilog_xl 2" UART_test
X: 0.000818517 Y:0
Monitor: Off
File Wave Display Measure Parameter MADE
Help
Cadence Waveform Display [x in time_units]
0
0.5m
1m
1.5m
2m
mouse L: addPoint(hiGetCommandPoint())
M: wavePopUp()
R: evalstring(waveRedoCmd)

21.2.7. Priključitev UART vmesnika na vodila jedra

Ker ima UART vmesnik počasnejši sistemski takt (16*9600Hz) kot mikroprocesorsko vezje (4MHz), moramo pri prenosu podatkov začasno sinhronizirati njuno medsebojno delovanje. Prav tako moramo poskrbeti, da UART vmesnik zaseže vodilo samo takrat, ko je naslovljen za branje, povezati pa moramo tudi prekinitveno linijo.

Krmilne signale UART vmesnika (uart_read in uart_write) zato zaklenemo ob koncu dekodiranega bralnega ali pisalnega cikla. Ob pisalnem ciklu tudi zaklenemo tudi podatek v txhold registru., medtem ko statusne zastavice zbrišemo, ko se konča ustrezni cikel UART vmesnika.

Status UART vezja preberemo v statusnem registru na naslovu 0xC0. Ob dekodiranju naslova podatkovnega ali statusnega registra preko tristanjskega gonilnika vodila postavimo podatek iz ustreznega registra na vodilo. UART vezje dekodira naslednjo V/I naslovno tabelo:

Naslov	Funkcija
0xC0, R	Statusni byte
0xC1, R/W	Podatkovni byte

Tabela 4: Pomnilniška karta UART vmesnika

Posamezni biti statusnega byta pomenijo naslednje:

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	TxRdy	RxRdy	Overrun	Framingerr

Prekinitveni vhod mikroprocesorja reagira na prehod linije iz logične "0" v logično "1" (rising edge triggered). Zato lahko prekinitev prožimo kar s rxrdy linijo.

21.2.8. Rutini branja in pisanja

Pred branjem in pisanjem moramo preveriti ustrezne statusne bite v statusni besedi, da preprečimo ponovljeno branje sprejemnega registra, ali pretekanje oddajnega registra. Rutino beri_UART() lahko izvedemo kot prekinitveno rutino, saj nam vezje signalizira novo vrednost s prekinitvijo.

Rutini sta blokirnega tipa, saj čakata, dokler UART ne sprejme podatka, ali dokler oddajni kanal ni prost.

```
#define UART_status_addr 0xC0
#define UART_data_addr   0xC1
#define Overrun      0x02
#define Framingerr   0x01
#pragma char UART_status = UART_status_addr
#pragma char UART_data = UART_data_addr
#pragma bit TxRdy = UART_status.4
#pragma bit RxRdy = UART_status.3

char beri_UART(void) {
     char temp;

     // cakaj na RxRdy
     while (RxRdy == 0);

     temp = UART_status;
     if (temp & (Overrun | Framingerr)) {
          return temp; //vrni kodo napake!
     } else {
          temp = UART_data;

          // cakaj na potrditev branja
          while (RxRdy != 0);

          return temp;
     }
     return 0;
}
```

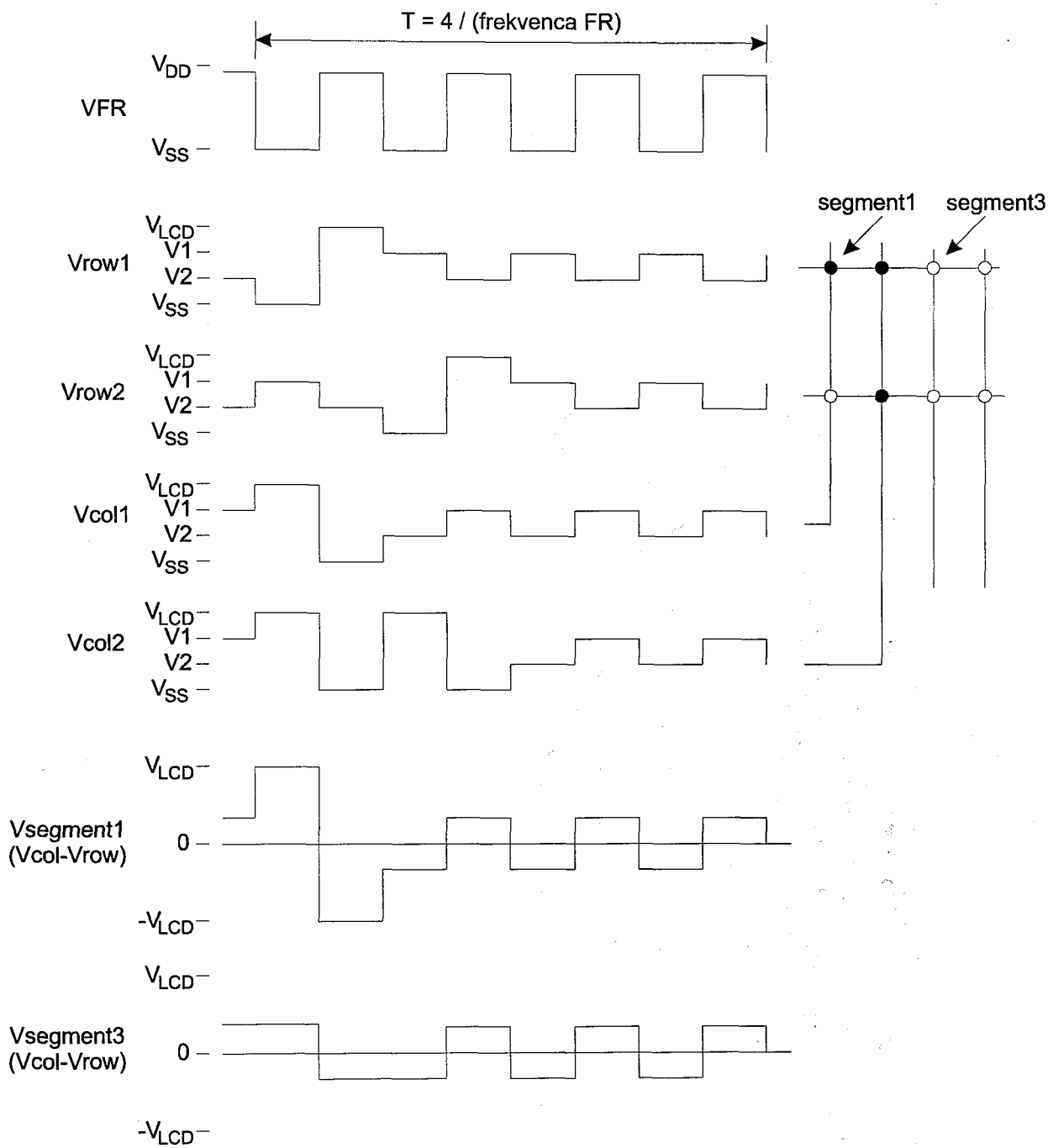
```
void pisi_UART(char TxData) {

      // cakaj na TxRdy
      while (TxRdy != 0);

      UART_data = TxData;

      // cakaj na potrditev pisanja
      while (TxRdy == 0);
}
```

21.3. Sklop LCD gonilnika

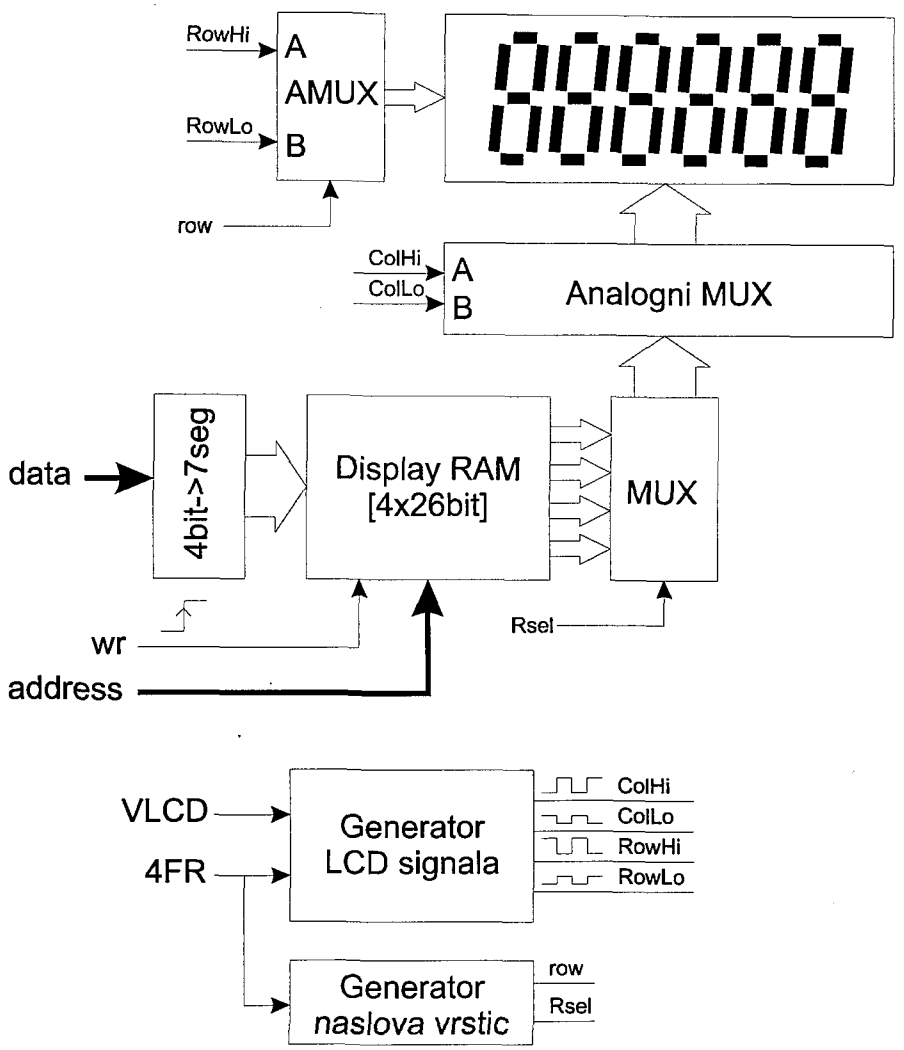
Aplikacija zahteva, da vezje med delovanjem prikazuje trenutne vrednosti izmerjenih veličin na prikazovalniku iz tekočih kristalov (LCD). Uporabimo prikazovalnik, načrtan po specifikacijah naročnika, zaradi velikega števila LCD segmentov pa je takšen prikazovalnik multipleksiran - segmenti so organizirani v 2, 4 ali 8 skupin, posamezni segmenti pa si delijo skupno zadnjo elektrodo (backplane).

Naslavljanje posameznega segmenta multipleksiranih LCD prikazovalnikih poteka s kombinacijo vrstice (row) in stolpca (column), vezje MOS stikal pa kombinira ustrezne izmenične signale iz generatorja LCD signala.

Generator signala generira izmenične pravokotne signale z napetostnimi nivoji, ki ustrezajo izbrani shemi multipleksiranja vrstic. Potek signala osveževanje štirivrstičnega LCD prikaza s štirikratnim multipleksiranjem prikazuje Slika 27.



Slika 26: Potek signalov LCD prikazovalnika



Slika 27: Sklop LCD gonilnika

Sklop LCD gonilnika vsebuje video pomnilnik, kjer posamezni postavljen bit ustreza prižganemu segmentu na prikazovalniku. Zaradi tehnologije izdelave LCD prikaza so segmenti multipleksirani brez vsakršnega pravila, zato mora načrtovalec znake, ki jih želi prikazati na LCD prikazu, predhodno ustrezno prekodirati glede na prikazovalnikovo multipleksirno tabelo. V izvedbi sistema s komercialnim mikrokontrolerjem je tabela običajno izvedena s programsko prekodirno rutino in tabelo v ROM pomnilniku, pri uporabniško načrtanem vezju pa prekodirno tabelo izvedemo s pravilnim naslavljanjem posameznih bitov video RAM pomnilnika.

Prikazovanje numeričnih simbolov zahteva prekodiranje podatka iz binarno kodirane decimalne vrednosti (BCD) v 7-segmentno kodo. Tudi to tabelo lahko izvedemo s programsko rutino s prekodirno tabelo, lahko pa prekodiranje izvedemo z optimizirano BCD -> 7-segmentno tabelo iz kombinacijske logike.

21.3.1. Generator izmeničnega LCD signala

Izbrana shema štirikratnega multipleksiranja zahteva štiri različne napetosti:

V_{LCD}	
V1	2 V_{LCD} / 3
V2	V_{LCD} / 3
V_{SS}	

Tabela 5: Napetosti 4/1 multipleksa

Napetosti generiramo z uporovnim vezjem v bloku **waveform_gen**. Vrednosti uporov izberemo glede na velikost, oziroma kapacitivnost LCD segmentov. Ker je naše vezje predvideno za manjše LCD prikazovalnike, lahko načrtamo notranje upore z vrednostjo okrog 20kΩ. Z večjo vrednostjo uporov zmanjšamo lastno porabo vezja, z manjšo pa zaradi manjše RC vrednosti izboljšamo odzivnost LCD prikaza.

Hitrost osveževanja prikazovalnika določimo s FR (frame rate) vhodom. Frekvenca signala na FR vhodu mora biti 4x frekvenca osveževanja, da pa preprečimo enosmerno komponento na LCD segmentih, mora imeti signal 50% delovno razmerje signal / ničla.

Vezje sekalnika signala (**chopper**) s frekvenco FR izmenoma preklaplja med vhodoma A in B. Rezultat sekanja je izmenični pravokotni signal frekvence FR in amplitudo med V_A in V_B.

	Amplituda	Faza
VRowHi	visoka, V_{LCD} - V_{SS}	0°
VRowLo	nizka, V_1 - V_2	180°
VColHi	visoka, V_{LCD} - V_{SS}	180°
VColLo	nizka, V_1 - V_2	0°

Tabela 6: Lastnosti signalov LCD prikazovalnika

Signali z lastnostmi, navedenimi v Tabela 6, bodo prižgali segment le pri kombinaciji VRowHi in VColHi, pri vseh drugih kombinacijah bo napetost na segmentu V_{LCD} / 3, kar pa ni dovolj za počrnitev segmenta.

21.3.2. Generator naslova vrstic

S signalom FR kontroliramo frekvenco spreminjanja signala vrstic in s tem hitrost osveževanja prikazovalnika. Naslov vrstice in signal izbrane vrstice generiramo v **row_gen** vezju.

Rsel<1:0>	ROW<1:4>
00	1000
01	0100
10	0010
11	0001
00	1000
01	0100
...	...

Tabela 7: Naslov vrstice

S signalom Rsel krmilimo multiplekserje v video-RAM pomnilniku, s signalom izbrane vrstice ROW pa lahko direktno krmilimo izhodna stikala analognega multiplekserja vrstic.

21.3.3. Video RAM pomnilnik

Video RAM pomnilnik (**display_ram**) je orgraniziran kot matrika 4vrstice x 26stolpcev D-FlipFlop celic, vsaka celica pa ustreza enemu segmentu na LCD prikazu, vrstice pa ustrezajo posamezni skupini multipleksiranih segmentov.

Izvedba video pomnilnika z D-FF celicami ni regularna struktura, saj za izvedbo prekodirne tabele potrebujemo naslavljanje posameznega bita v video pomnilniku. Poleg tega izhode posamezne pomnilne celice multipleksiramo s štirivhodnim multiplekserjem, ki glede na naslov vrstice izbere ustrezno vrstico video-RAM pomnilnika, izhod multiplekserja pa krmili stikala, s katerimi izberemo pravokotni izmenični signal ustrezne amplitude in faze.

S takšno izvedbo poteka osveževanje LCD prikaza povsem nemoteno, saj so vhodi in izhodi takšnega pomnilnega vezja neodvisni. Sinhronizacija branja in pisanja iz video pomnilnika ni potrebna.

LCD prikaz je sestavljen iz 10 7-segmentnih polj, ter iz množice simbolov, s katerimi ponazorimo stanje delovanja sistema. Zato dostop do video pomnilnika s strani mikroprocesorja organiziramo v obliki 16 izhodnih naslovov, ki ustrezajo 10 poljem, v preostalih 6 naslovih pa kot posamezne bite v bitni karti izvedemo prižiganje in ugašanje simbolov:

Naslov	LCD naslov	Funkcija
0xD0	SA1	polje 1
0xD1	SA2	polje 2
...	...	...
0xD8	SA9	polje 9
0xD9	SA10	polje 10
0xDA	EA1	simboli 1
...	...	...
0xDE	EA5	simboli 5
0xDF	XA	*prazno*

Tabela 8: Naslovi LCD segmentov

Ker dekodirnik naslovov krmili D-FF, tudi tukaj "glitchi" niso zaželjeni, zato naslove dekodiramo z enonivojskim vezjem iz NOR vrat.

Sestavni del bloka je prekodirno vezje, ki iz BCD kode generira ustrezno 7-segmentno kodo. Takšno optimizirano vezje omogoči, da program na željeno lokacijo vpiše 4-bitno binarno kodo decimalne cifre (BCD), preostalih 6 kombinacij pa lahko uporabimo za prikaz preprostih alfanumeričnih znakov za besedice `Error`, `Addr`, ...

Tabelo bomo generirali s pomočjo HDL optimizatorja. Z vhodno datoteko opišemo izhode vezja kot funkcijo vhodov:

```
ENTITY BCD_to_7seg IS
PORT (
        X       : IN BIT_VECTOR (3 DOWNTO 0);
        A       : OUT BIT;
        B       : OUT BIT;
        C       : OUT BIT;
        D       : OUT BIT;
        E       : OUT BIT;
        F       : OUT BIT;
        G       : OUT BIT
        );
END BCD_to_7seg;

ARCHITECTURE beh OF BCD_to_7seg IS

SIGNAL W0 : BIT;
SIGNAL W1 : BIT;
SIGNAL W2 : BIT;
SIGNAL W3 : BIT;
SIGNAL W4 : BIT;
SIGNAL W5 : BIT;
SIGNAL W6 : BIT;
SIGNAL W7 : BIT;
SIGNAL W8 : BIT;
SIGNAL W9 : BIT;
SIGNAL W10: BIT;
SIGNAL W11: BIT;
SIGNAL W12: BIT;
SIGNAL W13: BIT;
SIGNAL W14: BIT;
SIGNAL W15: BIT;

BEGIN

W0  <= NOT X(3) AND NOT X(2) AND NOT X(1) AND NOT X(0);
W1  <= NOT X(3) AND NOT X(2) AND NOT X(1) AND     X(0);
W2  <= NOT X(3) AND NOT X(2) AND     X(1) AND NOT X(0);
W3  <= NOT X(3) AND NOT X(2) AND     X(1) AND     X(0);
W4  <= NOT X(3) AND     X(2) AND NOT X(1) AND NOT X(0);
W5  <= NOT X(3) AND     X(2) AND NOT X(1) AND     X(0);
W6  <= NOT X(3) AND     X(2) AND     X(1) AND NOT X(0);
W7  <= NOT X(3) AND     X(2) AND     X(1) AND     X(0);
W8  <=     X(3) AND NOT X(2) AND NOT X(1) AND NOT X(0);
W9  <=     X(3) AND NOT X(2) AND NOT X(1) AND     X(0);
W10 <=     X(3) AND NOT X(2) AND     X(1) AND NOT X(0);
W11 <=     X(3) AND NOT X(2) AND     X(1) AND     X(0);
W12 <=     X(3) AND     X(2) AND NOT X(1) AND NOT X(0);
W13 <=     X(3) AND     X(2) AND NOT X(1) AND     X(0);
W14 <=     X(3) AND     X(2) AND     X(1) AND NOT X(0);
W15 <=     X(3) AND     X(2) AND     X(1) AND     X(0);

A <= W0 OR W2 OR W3 OR W5 OR W7 OR W8 OR W9 OR W10 OR W13;
B <= W0 OR W1 OR W2 OR W3 OR W4 OR W7 OR W8 OR W9  OR W13 OR W14;
C <= W0 OR W1 OR W3 OR W4 OR W5 OR W6 OR W7 OR W8 OR W9 OR W12 OR W13 OR W14;
D <= W0 OR W2 OR W3 OR W5 OR W6 OR W8 OR W10 OR W12 OR W14;
E <= W0 OR W2 OR W6 OR W8 OR W10 OR W11 OR W12 OR W13 OR W14;
F <= W0 OR W4 OR W5 OR W6 OR W8 OR W9 OR W10 OR W13;
G <= W2 OR W3 OR W4 OR W5 OR W6 OR W8 OR W9 OR W10 OR W11 OR W12 OR W13 OR W14;

END;
```

Optimizator s pomočjo vgrajenih pravil generira funkcionalni opis minimalnega logičnega vezja:

```
-- VHDL data flow description generated from `BCD_to_7seg`
--              date : Wed Nov  3 12:15:53 1999

-- Entity Declaration

ENTITY test IS
  PORT (
  x : in bit_vector(3 DOWNTO 0) ;     -- x
  a : out BIT; -- a
  b : out BIT; -- b
  c : out BIT; -- c
  d : out BIT; -- d
  e : out BIT; -- e
  f : out BIT; -- f
  g : out BIT  -- g
  );
END test;

-- Architecture Declaration

ARCHITECTURE VBE OF test IS
  SIGNAL aux29 : BIT;          -- aux29
  SIGNAL aux31 : BIT;          -- aux31

BEGIN
  aux31 <= (not (x (3)) and x (0));
  aux29 <= (x (2) xor x (1));

g <= ((not (x (0)) and x (1)) or (not (x (1)) and x (3)) or aux29);

f <= ((not (x (1)) and x (0) and (x (3) or x (2))) or (not (x (0))
       and ((not (x (1)) and not (x (2))) or (x (3) xor x (2)))));

e <= ((not (x (0)) and (not (x (2)) or x (1))) or (x (3) and aux29));

d <= ((aux31 and aux29) or (not (x (0)) and (x (3) or not (x (2)) or x (1))));

c <= (not (x (1)) or (not (x (0)) and x (2)) or aux31);

b <= ((x (3) and (x (1) xor not ((not (x (0)) and x (2))))) or (not
       (x (3)) and (not (x (2)) or not ((x (1) xor x (0))))));

a <= ((not (x (2)) and not (x (0))) or (x (0) and not (x (1)) and
       x (3)) or (not ((not (x (1)) and not (x (2)))) and aux31));

END;
```

Iz takšnega optimiziranega funkcionalnega opisa s HDL prevajalnikom generiramo spisek povezav med naborom celic določene tehnologije. Shema optimiziranega vezja se nahaja v prilogi 26.6.

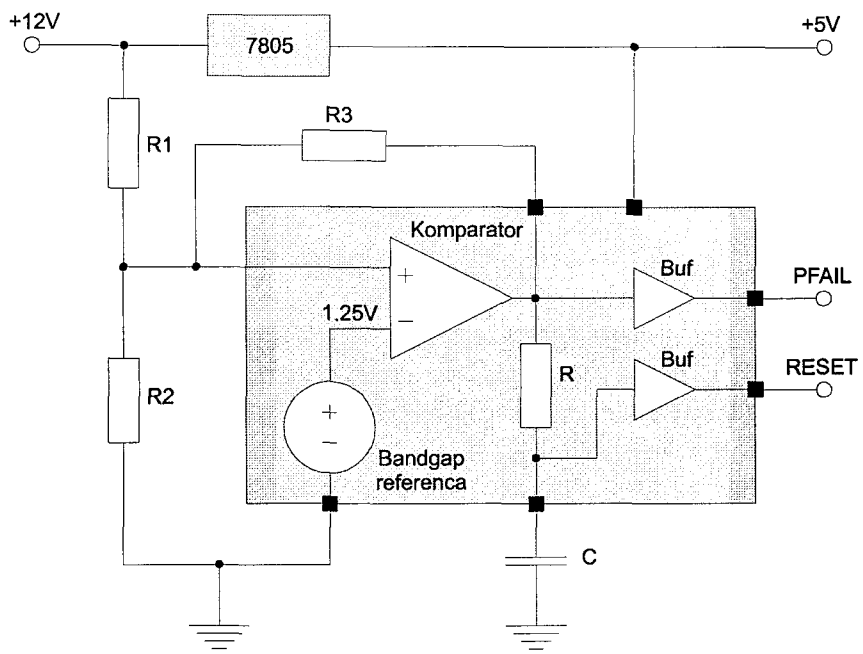
21.3.4. Analogna stikala

Med izhodnima napetostima (VcolHi, VcolLo, oziroma med VrowHi in VrowLo) preklapljamo s t.im. MOS transmisijskimi stikali. Z vzporedno vezavo p in n-MOS tranzistorja poskrbimo, da ima vključeno stikalo relativno nizko prehodno upornost po celotnem napetostnem območju, tudi tukaj mora biti upornost stikala (driver impedance) dovolj majhna, da lahko dovolj hitro preklopi segment z določeno kapacitivnostjo.

21.4. Vezje nadzora napajanja z reset generatorjem

Vezje nadzora napajanja nadzira neregulirano napetost 9-12V pred linearnim napetostnim regulatorjem, signal nezadostne napajalne napetosti pa naj generira pri neregulirani napetosti približno 7.5V.

Hkrati želimo, da vezje generira RESET signal, s katerim zadrži delovanje mikroprocesorja in periferije, dokler se napajanje ne stabilizira.



Slika 28: Vezje nadzora napajalne napetosti

Vezje je sestavljeno iz bandgap generatorja referenčne napetosti z izhodno napetostjo okrog 1.25V, referenčno napetost pa primerjamo z neregulirano napetostjo, deljeno z uporovnim delilnikom R_1, R_2.

Za manjšo občutljivost na šum napajalne linije dodamo med vhod in izhod komparatorja upor R_3, s tem pa komparatorju dodamo histerezo. Napetost preklopa izhoda komparatorja iz nizkega v visoko stanje (V_H), preklopa iz visokega v nizko (V_L) ter histerezo komparatorja V_H - V_L izračunamo:

$$V_H = V_{BG}\left(1+\frac{R_1}{R_2}+\frac{R_1}{R_3}\right)$$

$$V_L = V_{BG}\left(1+\frac{R_1}{R_2}-\frac{R_1\left(V_{DD}-V_{BG}\right)}{V_{BG}R_3}\right) \qquad (21.4.1)$$

$$V_H - V_L = V_{DD}\left(\frac{R_1}{R_2}\right)$$

Pri referenčni napetosti V_{BG} = 1.25V, V_{DD} = 5V, V_L = 7.5V in histerezi V_H - V_L = 1V, po formuli 21.4.1 izračunamo upore:

R_1 = 50k
R_2 = 10k
R_3 = 300k

Vezje naj drži mikroprocesor in periferijo v reset stanju še 2ms za tem, ko napajalna napetost preseže V_H, ter postavi v reset 2ms za tem, ko napajalna napetost pade pod V_L. Na ta način poskrbimo, da mikroprocesor začne delovati pri popolnoma stabilni napetosti, po signalu nezadostne napetosti pa mikroprocesor deluje še vsaj 2ms. V času med signalom nezadostne napetosti in aktiviranjem reset signala naj mikroprocesor kontrolirano zaključi delovanje in shrani kritične podatke v baterijsko napajani pomnilnik.

Zakasnilno vezje izvedemo z RC vezjem, vezje pa preklopi, ko napetost na vhodu inverterja preide 50% napajalne napetosti.

Iz formule:

$$V_{out} = V_{in}\left(1-e^{-\frac{t}{RC}}\right) \qquad (21.4.2)$$

za V_{out} = 50% V_{in} in t = 2ms, izračunamo:

R = 250kΩ in C = 10nF

Celotna shema vezja je v prilogi 26.7.

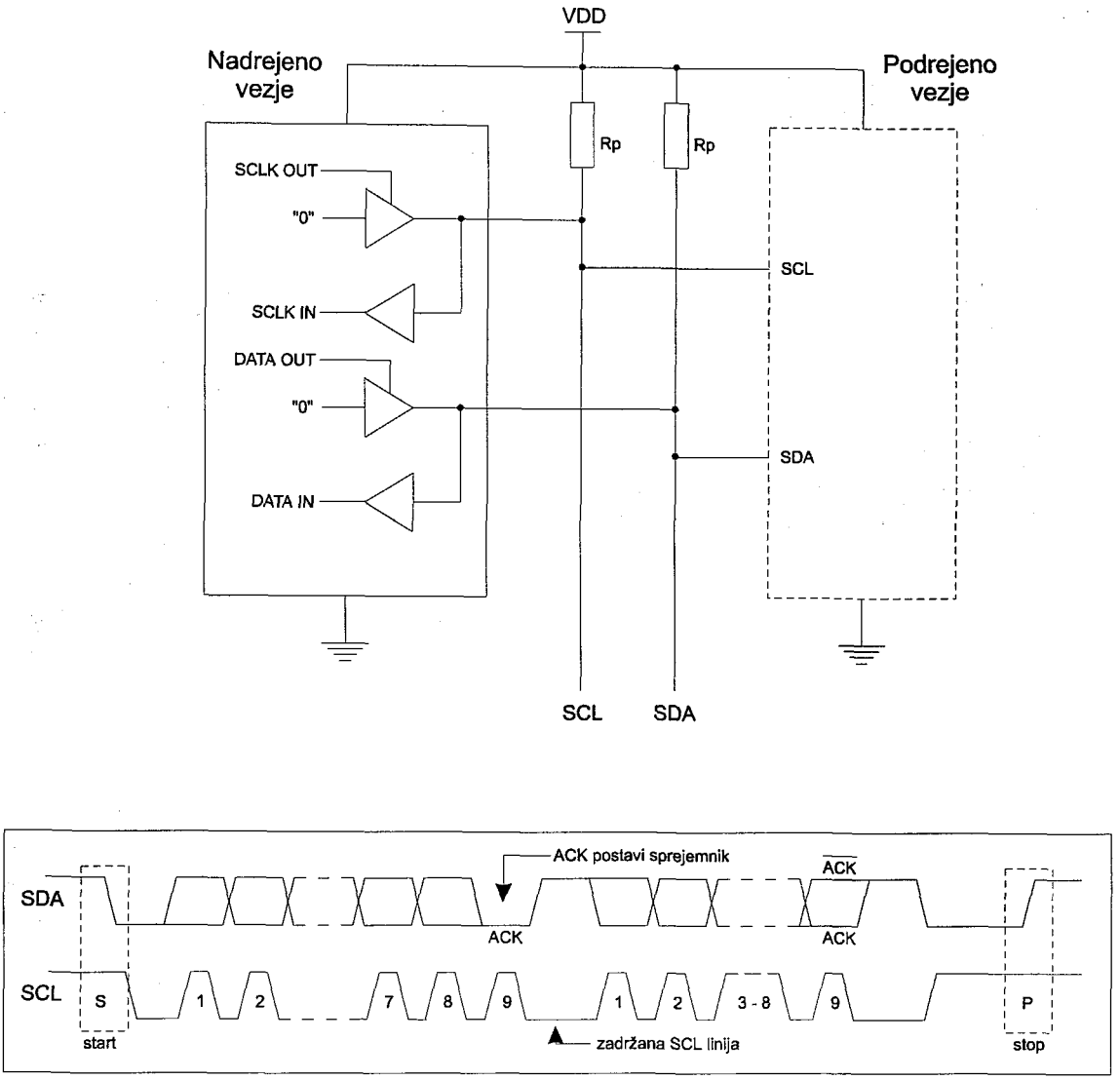
21.5. I_2C V/I vmesnik

I_2C vmesnik je dvožični sinhroni serijski vmesnik, sestavljen iz ene podatkovne (SDA) in linije ure (SCL). Zaradi "open drain" izhodov lahko povezuje vse vrste vezij (CMOS, bipolarne, itd...). Protokol I^2C vodila je bil načrtan kot preprost in poceni način za komunikacijo med integriranimi vezji v telekomunikacijski, industrijski, ter elektroniki za široko potrošnjo. Danes je I^2C vodilo izvedeno v velikem številu mikrokontrolerjev in perifernih vezij, ter predstavlja dobro izbiro za medsebojne povezave nizkih hitrosti.

Na I_2C vmesniku je lahko več nadrejenih vezij (master), z vodilom pa *lahko upravlja eno, ali več takšnih vezij. Vendar je navadno na vodilu le eno* nadrejeno vezje, kar poenostavi upravljanje. V takšnem primeru ni potrebno reševati konfliktov pri sinhronizaciji dostopa do vodila in medsebojne komunikacije več nadrejenih vezij.

SCL linijo lahko krmili le nadrejeno vezje, medtem ko podrejeno (slave) lahko le pošilja podatke po SDA liniji, vendar le, ko je izbrano s strani nadrejenega vezja.

Prenos podatkov aktivira nadrejeno vezje. Prehod SDA linije iz visokega v nizko stanje pri visokem stanju na SCL liniji predstavlja START pogoj, temu pa sledita 7-bitni naslov podrejenega vezja in bit smeri prenosa podatkov (nadrejeno vezje -> podrejeno, ali obratno). Podrejeno vezje pošlje potrditev tako, da drži SDA linijo v nizkem stanju en urni cikel. Brez potrditve nadrejeno vezje prekine prenos, v nasprotnem primeru pa glede na bit smeri steče prenos 8-bitnega podatka.



Slika 30: Shema in potek prenosa po I2C vodilu

Sprejemno vezje potrdi vsak sprejem 8-bitnega podatka, prenos več bytov pa traja, dokler nadrejeno vezje ne izvede STOP ali ponovljenega START ukaza. STOP ukaz je definiran kot prehod SDA linije iz nizkega v visoko stanje, ko je SCL linija v visokem stanju. Podrejeno vezje lahko prekine branje nadrejenega vezja tako, da ne potrdi zadnjega prebranega podatka. Nadrejeno vezje za tem aktivira STOP ali ponovljeni START.

Podrejeno vezje lahko zadrži prenos podatkov nadrejenega vezja s tem, da drži SCL linijo v nizkem stanju, to pa postavi nadrejeno vezje v čakalno stanje.

Posledica izvedbe START in STOP ukaza je, da se podatki na SDA liniji lahko spreminjajo samo takrat, ko je SCL linija v nizkem stanju (drugače bi podrejeno vezje spremebo razumelo kot START ali STOP ukaz).

Obe liniji sta dvosmerni, zato morajo biti izhodi izvedeni z odprtim ponorom (open-drain) v CMOS vezjih, ali odprtim kolektorjem v bipolarnih / BiCMOS vezjih. Vsaka linija mora biti priključena na napajalno napetost preko pull-up upora. Na ta način je nezasežena linija v visokem nivoju, ko pa ena od naprav zaseže linijo, linija preide v nizki nivo.

Ker je I^2C vodilo sinhrono, duty-cycle in perioda takta na SCL liniji nista kritična. Tako tudi ni potrebno, da so zakasnitve softverskih rutin natančno določene. Paziti je treba le, da frekvenca signalov na linijah ne preseže 100kHz, kolikor še dopušča I^2C standard.

Za softversko izvedbo I^2C vodila potrebujemo dva kombinirana vhodno / izhodna pina odprtega ponora, priključena na SDA in SCL linijo. Občutljivost na motnje izboljšamo z uporabo Schmittovih preklopnih vezij v vhodnem delu vezja. Naslov vhodov in izhodov pina dekodiramo na naslovu 0xC8 naslovnega vodila mikroprocesorja, v D-FlipFlop pomnilnih celicah pa zaklenemo vpisano željeno stanje SCL in SDA linij.

Naslov	Funkcija
0xC8 - 0xCF	I2C vodilo

Tabela 9: Pomnilniška karta I2C vmesnika

Biti vpisanega ali prebranega byta pa pomenijo:

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	0	SDA	SCL

Softverska izvedba I^2C vmesnika temelji na nizkonivojski rutini i2c_set(), s katero postavljamo stanja na SDA in SCL liniji. Paziti moramo, da frekvenca signalov ne preseže 100kHz, zato v funkcijo vstavimo zakasnilno zanko.

I^2C protokol izvedemo z zaporedjem funkcij srednjega nivoja i2c_reset(), i2c_repstart(), i2c_start(), i2c_stop(), i2c_one(), i2c_zero() in i2c_ack(). Iz teh funkcij sestavimo funkcije višjega nivoja i2c_readbyte() in i2c_sendbyte.

S funkcijama i2c_read() in i2c_write() lahko testiramo delovanje I^2C komunikacije. Ob zagonskem testu vpišemo poljubno vrednost v prazno lokacijo RAM pomnilnika, z naslednjim ukazom pa jo preberemo.

```
/*
 * Frekvenca sistemske ure:
 * SYSCLK = 4194304Hz
 *
 * Cas thigh i2c takta (standard mode):
 * tclkhi >= 5us
 *
 * stevilo SYSCLK i2c_set ukaza
 * n =  tclkhi * 4194304 = 21 SYSCLK
 */

/*
 * I2C zakasnilna zanka, 10 SYSCLK
 */
#define I2C_DELAY_LOOP  1

/*
 * naslov i2c vmesnika
 */
#define i2c_addr  0xC8
#pragma char i2c_state = i2c_addr


/*
 * Podakovna linija I2C vmesnika
 */
#pragma bit i2c_data = i2c_addr.1

/*
 * Funkcija postavi SCL in SDA liniji
 * Porabljenih iklov: 12 SYSCLK + 10 SYSCLK * I2C_DELAY_LOOP
 */
void i2c_set(char scl, char sda)
{
      char delay;

      i2c_state = (sda << 1) | scl;
      delay = I2C_DELAY_LOOP;
      while (delay)
            delay--;
}

/*
 * Inicializacija I2C vodila v neaktivno stanje
 *  SCL: 1
 *  SDA: 1
 */
void i2c_reset( void)
{
      i2c_set(1, 1);
}

/*
 * Ponovljeni start, za ACK ciklom
 * SCL: 0 -> 1 -> 1 -> 0
 * SDA: 1 -> 1 -> 0 -> 0
 */
void i2c_repstart ( void)
{
      i2c_set(0, 1); i2c_set(1, 1); i2c_set(1, 0); i2c_set(0, 0);
}
```

```
/*
 * Zacetek prenosa
 *  SCL: 1
 *  SDA: 1 -> 0
 *
 */
void i2c_start( void)
{
      i2c_set(1, 1); i2c_set(1, 0); i2c_set(0, 0);
}

/*
 * Konec prenosa
 *  SCL: 1
 *  SDA: 0 -> 1
 *
 */
void i2c_stop( void)
{
      i2c_set(0, 0); i2c_set(1, 0); i2c_set(1, 1);
}

/*
 * Poslji "1"
 *  SCL: 0 -> 1 -> 0
 *  SDA: 1
 */
void i2c_one( void)
{
      i2c_set(0, 1); i2c_set(1, 1); i2c_set(0, 1);
}

/*
 * Poslji "0"
 *  SCL: 0 -> 1 -> 0
 *  SDA: 0
 */
void i2c_zero( void)
{
      i2c_set(0, 0); i2c_set(1, 0); i2c_set(0, 0);
}

/*
 * Preberi potrditev
 *  SCL: 0 -> 1 -> 0
 *  SDA: 1
 *
 * Vrne stanje SDA
 * (SDA postavi sprejemnik)
 */
char i2c_ack( void)
{
      char ack;

      i2c_set(0, 1); i2c_set(1, 1);
      /* preberi vrednost SDA, 0'b000000?0 */
      ack = (i2c_data >> 1) & 0x01;
      i2c_set(0, 1);
      return ack;
}
```

```
char i2c_sendbyte( char data)
{
	char i, ack;

	for (i = 7; i >= 0; i--) {
		if (data & 0x80)
			i2c_one();
		else
			i2c_zero();
		data <<= 1;
	}
	ack = i2c_ack();
	return ack;
}

char i2c_readbyte( char last)
{
	char i, data;

	data = 0;
	for (i = 7; i >= 0; i--) {
		data <<= 1;
		i2c_set(0, 1);
		i2c_set(1, 1);
		if (i2c_data)
			data |= 1;
		i2c_set(0, 1);
	}
	/* Poslji potrditev ACK ali NACK ob zadnjem bytu prenosa */
	if (last)
		i2c_one();
	else
		i2c_zero();
	return data;
}

/* ---------------------------------------------------------------*/

/*
 * TESTNA funkcija branja iz I2C naslova
 *  naslov: 7bit naslov podrejenega vezja
 */		(Funkcija doda LSB "1")
char i2c_read(char addr)
{
	char ret;

	i2c_start();
	i2c_sendbyte((addr << 1) | 1);
	ret = i2c_readbyte(1);
	i2c_stop();
	return ret;
}
```

```
/*
 * TESTNA funkcija pisanja na I2C naslov
 *  naslov: 4b: naslov podrejenega vezja +
 *          3b: naslov strani
 *          (Funkcija doda LSB "0")
 *  data1: naslov v posamezni podatkovni strani
 *  data2: vpisani podatek
 */
char i2c_write( char addr, char data1, char data2)
{
      char ack;

      i2c_start();
      i2c_sendbyte(addr << 1);
      ack = i2c_sendbyte(data1);
      ack = i2c_sendbyte(data2);
      i2c_stop();
      return ack;
}

void main( void)
{
      i2c_reset();

      // FM24C16 Slave ID = 1010
      // FM24C16 Page Select = 000
      // slave address: 01010000 = 0x50
      // word address: 0x07

      // poslji podatek 0xFA
      i2c_write (0x50, 0x07, 0xFA);

      // preberi podatek s trenutnega naslova
      if (i2c_read (0x50) != 0xFA) {
            // NAPAKA I2C komunikacije
      }
}
```

22. Izvedba softverskih rutin

Delovanje integriranega vezja poleg nizkonivojskih rutin, s katerimi dostopamo do hardverskih blokov podpirajo naslednje rutine:

- 32-bitni števec dogodkov, prožen s prekinitvijo
- Razdeljevalnik softverskih opravil z uro realnega časa
- FIFO vmesni pomnilnik serijskega UART vmesnika
- glavna prekinitvena rutina

22.1. Števec dogodkov

Globalna spremenljivka 32bitnega števca dogodkov je sestavljena iz štirih osembitnih spremenljivk. Povečujemo le LSB byte, višje byte pa povečamo le ob postavljeni zastavici prenosa (carry flag, CFLAG) povečanja nižjega byta.

Rutino counter() kličemo iz glavne prekinitvene rutine, takrat so prekinitve blokirane. Branje spremenljivke števca poteka iz softverskih opravil, med izvajanjem opravil pa so prekinitve prav tako blokirane. Čeprav spremenljivke števca spreminjamo iz druge programske niti, pa nam zaradi blokade prekinitev pri operacijah s spremenljivkami števca, spremenljivk ni treba deklarirati kot spremenljive (volatile).

```
// Globalne spremenljivke
char count_3, count_2, count_1, count_0;

void counter( void)
{
      // povecaj 32bitni stevec dogodkov
      count_0++;
      if (CFLAG)
            count_1++;
      if (CFLAG)
            count_2++;
      if (CFLAG)
            count_3++;
      return;
}

/* TESTNA rutina */
void main (void)
{
      count_0 = 0;
      count_1 = 0;
      count_2 = 0;
      count_3 = 0;

      while (count_2 == 0)
            counter();
}
```

22.2. Razdeljevalnik softverskih opravil z uro realnega časa

Želimo, da bi bil razdeljevalnik opravil kar najbolj preprost, saj se procesorski cikli razdeljevalnika odštevajo od 4000 procesorskih ciklov, kolikor jih ima na voljo vsako softverskega opravila.

Čeprav je izvedba z indirektnim skokom (IJMP) in tabelo skokov precej pripravnejša, nam arhitektura MTC8308 mikroprocesorja tega ne dopušča, saj indirektni skok pobriše naslov povratka iz prekinitve. Zato izvedemo tabelo klicanih podprogramov z zaporedjem if() stavkov.

Razdeljevalnik opravil aktiviramo z zunanjo časovno referenco vsako milisekundo, zato vanj vdelamo števec milisekund, katerega lahko softverska opravila uporabijo kot svojo časovno referenco. Delovanje števca milisekund poteka enako kot delovanje števca dogodkov, tudi tu so prekinitve blokirane ob vsakem dostopu do spremenljivke.

```
// Globalne spremenljivke
char scheduled;
char ticks_3, ticks_2, ticks_1, ticks_0;

// Softverske rutine razdeljevalnika opravil
// Izvedba z IJMP (indirektni skok) tukaj ni
// mozna, ker IJMP pobrise naslov povratka, ce
// je klican iz subrutine.

void void_task_0( void) { /* empty */ }

void void_task_1( void) { /* empty */ }

void void_task_2( void) { /* empty */ }

void void_task_3( void) { /* empty */ }

void void_task_4( void) { /* empty */ }

void void_task_5( void) { /* empty */ }

void void_task_6( void) { /* empty */ }

void void_task_7( void) { /* empty */ }
```

```
/*
 * Razdeljevalnik softverskih opravil
 */
void scheduler (void)
{

      // Tabela razdeljevalnika opravil
      if (scheduled == 0) void_task_0();
      else if (scheduled == 1) void_task_1();
      else if (scheduled == 2) void_task_2();
      else if (scheduled == 3) void_task_3();
      else if (scheduled == 4) void_task_4();
      else if (scheduled == 5) void_task_5();
      else if (scheduled == 6) void_task_6();
      else if (scheduled == 7) void_task_7();

      // krozno povecaj indeks softverskega opravila
      if (++scheduled >= 8)
            scheduled = 0;

      // povecaj 32bitni stevec milisekund
      ticks_0++;
      if (CFLAG)
            ticks_1++;
      if (CFLAG)
            ticks_2++;
      if (CFLAG)
            ticks_3++;
      return;
}

/* TESTNA rutina */
void main (void)
{
      ticks_0 = 0;
      ticks_1 = 0;
      ticks_2 = 0;
      ticks_3 = 0;

      while (ticks_2 == 0)
            scheduler();
}
```

22.3. FIFO vmesni pomnilnik serijskega UART vmesnika

Prvi noter, prvi ven (FIFO) pomnilnik potrebujemo, kadar proces pisanja poteka hitreje od procesa branja. UART vmesnik sprejme znak 960x v sekundi, medtem ko softversko opravilo, ki sprejete podatke obdela, aktiviramo vsakih 8ms, oziroma 125x v sekundi. Prejete podatke med dvema obdelavama shranimo v FIFO strukturo.

FIFO pomnilnik UART vezja izvedemo kot krožno FIFO strukturo, v katero kažeta dva kazalca naslova: kazalec pozicije pisanja in kazalec pozicije branja. Kazalca prestavljata indeks v FIFO pomnilnik, krožni dostop pa izvedemo z maskiranjem spodnjih N bitov kazalca.

Podprograma vpisa in branja vzdržujeta dve statusni spremenljivki, ki signalizirata, kdaj je pomnilnik poln, oziroma prazen. Pred klicem podprogramov moramo preveriti stanje FIFO pomnilnika, kajti branje iz praznega, oziroma pisanje v polni FIFO vrne nedefinirane rezultate.

```
// Globalni FIFO (first in, first out)
// vmesni pomnilnik dolzine 8 bytov, indeksi [0 - 7]

#define FIFO_LEN  8
#define MAX_FIFO_IDX    (FIFO_LEN - 1)

// Globalni FIFO pomnilnik
char FIFO [FIFO_LEN];

// pozicija vpisa in pozicija branja
char writeptr, readptr;

// statusne spremenljivke
bit fifo_empty, fifo_full;

// funkcija pisanja v FIFO pomnilnik, non-reentrant zaradi
// globalnih spremenljivk. Ena funkcija za VSAKO FIFO strukturo
void fifo_write (char data)
{
      writeptr++;
      writeptr &= MAX_FIFO_IDX;
      if (writeptr == readptr)
            fifo_full = 1;
      FIFO [writeptr] = data;
      fifo_empty = 0;
}
// funkcija branja iz FIFO pomnilnika, non-reentrant zaradi
// globalnih spremenljivk. Ena funkcija za VSAKO FIFO strukturo
char fifo_read (void)
{
      readptr++;
      readptr &= MAX_FIFO_IDX;
      if (readptr == writeptr)
            fifo_empty = 1;
      fifo_full = 0;
      return (FIFO [readptr]);
}
```

```
/* ---------------------------------- */
// Inicializacija in TESTNA procedura,
// preizkusi vse FIFO lokacije in funkcije

void main(void)
{
      char x;

      writeptr = (-1 & MAX_FIFO_IDX);
      readptr = (-1 & MAX_FIFO_IDX);

      fifo_empty = 1;
      fifo_full = 0;

      fifo_write ('A'); /* zbrise fifo_empty */
      fifo_write ('B');
      fifo_write ('C');
      fifo_write ('D');
      fifo_write ('E');
      fifo_write ('F');
      fifo_write ('G');
      fifo_write ('H'); /* postavi fifo_full */

      x = fifo_read(); /* zbrise fifo_full */

      fifo_write ('I'); /* postavi fifo_full */

      x = fifo_read();
      x = fifo_read();
      x = fifo_read();
      x = fifo_read();
      x = fifo_read();
      x = fifo_read();
      x = fifo_read();

      // Mesane kombinacije
      fifo_write ('J');
      fifo_write ('K');
      fifo_write ('L');
      fifo_write ('M');

      x = fifo_read();
      x = fifo_read();

      fifo_write ('N');
      fifo_write ('O');
      fifo_write ('P');
      fifo_write ('Q');

      x = fifo_read();
      x = fifo_read();
      x = fifo_read();
      x = fifo_read();

      fifo_write ('R');

      x = fifo_read();
      x = fifo_read();
      x = fifo_read();
      x = fifo_read(); /* postavi fifo_empty */
}
```

22.4. Glavna prekinitvena rutina

Glavna prekinitvena rutina dekodira indeks aktivirane prekinitve glede na postavljeno zastavico aktivirane prekinitve. Zastavica prekinitve se postavi ob rastočem robu prekinitvenega signala na ustrezni prekinitveni liniji, neodvisno od stanja maskirnih bitov posamezne prekinitve. Rutina glede na indeks prekinitve pokliče ustrezno prekinitveno rutino.

Jedro MTC-8308 lahko obdeluje prekinitve dveh nivojev: nivoja 0, katerega generira prekinitev INT0, ter nivoja 1, ki ga generirajo prekinitve INT1-INT3. Če je prekinitev omogočena, se mikroprocesorsko jedro odzove tako, da shrani trenutno veljavni naslov ukaza in izvede skok na naslov 0x01 (za prekinitve nivoja 0), oziroma na naslov 0x02 (za prekinitve nivoja 1). Ob skoku mikroprocesor onemogoči vse prekinitve, prekinitve programsko omogočimo na koncu prekinitvene rutine.

Ob klicu glavne prekinitvene rutine moramo shraniti statusni register, povrnemo pa ga ob izhodu iz rutine.

Pojavi se problem, kako obdelati prekinitve, ki so se pojavile med blokado prekinitev. Aktivirano prekinitev označuje postavljena zastavica, zato pred izhodom iz glavne prekinitvene rutine ponovno preverimo status prekinitvenih zastavic. Ob katerikoli postavljeni zastavici še enkrat skočimo na začetek obdelave prekinitev obeh nivojev, v nasprotnem primeru pa vrnemo statusni register, omogočimo prekinitve in zapustimo glavno prekinitveno rutino. Na ta način lahko pravočasno obdelamo vse čakajoče prekinitve - pod pogojem, da je vsota izvajalnih časov vseh prekinitvenih podprogramov, skupaj z izvajalnim časom glavne prekinitvene rutine, krajša kot najkrajši čas med dvema zaporednima prekinitvama istega indeksa.

```
#pragma bit IRQ0 = INTREG.0
#pragma bit IRQ1 = INTREG.1
#pragma bit IRQ2 = INTREG.2
#pragma bit IRQ3 = INTREG.3

// Zgornji stirje biti INTREG so maskirni biti prekinitev
#define IRQMASK 0xF0

// "Optimizacija" zbrise level0 in level1
// prekintvena vektorja, zato
#pragma optimize 0
```

```
// Prekinitvena vektorja si sledita
// na naslovu 0x01 in 0x02
#pragma origin = 0x01

interrupt level0_1 ( void)
{
      char flags;

      goto level0;
      goto level1;

// Prekinitev nivoja 0 (INT0)
level0:

      // Ob prekinitvi MTC-8308 onemogoci vse prekinitve,
      // zato naslednji ukaz ni potreben
      // INTREG &= ~IRQMASK;

      // shrani statusni register
      flags = STATREG;

level0_again:
      powerfail();

      // Pobrisi zastavico prekinitve
      IRQ0 = 0;

      // Izhod iz prekinitve
      goto eofirq;

// Prekinitev nivoja 1 (INT1, INT2, INT3)
level1:

      // Ob prekinitvi MTC-8308 onemogoci vse prekinitve,
      // zato naslednji ukaz ni potreben
      // INTREG &= ~IRQMASK;

      // shrani statusni register
      flags = STATREG;

level1_again:
      if (IRQ1) {
            stevec();

            // pobrisi zastavico prekinitve
            IRQ1 = 0;
      }

      // servisiraj naslednjo prekinitev
      // zastavica se lahko se pojavi tudi
      // med ostalimi prekinitvami!
      if (IRQ2) {
            scheduler();

            // pobrisi zastavico prekinitve
            IRQ2 = 0;
      }
```

```
        // itd...
        if (IRQ3) {
              beri_UART();

              // pobrisi zastavico prekinitve
              IRQ3 = 0;
        }

// Vstopna tocka izhoda iz prekinitvene rutine
eofirq:

        // Je bila med blokiranimi prekinitvami kaksna
        // zastavica na novo postavljena?
        // (Zgornji 4 biti so se vedno 0000.)

        // Poglej level0
        if (IRQ0)
              goto level0_again;

        // Poglej level1 (level0 je obdelana).
        if (INTREG)
              goto level1_again;

        // drugace vrni shranjeni statusni register
        STATREG = flags;

        // Ni aktivnih prekinitev, omogoci prekinitve in adijo.
        INTREG = IRQMASK;
}


//Po uspesnem power-on self-testu
//aktiviraj prekinitve
void main( void)
{
        //setup periferije, testi, itd...

        // Omogoci prekinitve
        INTREG = IRQMASK;
}
```

23. Simulacije in emulacije

Sodobna načrtovalska orodja omogočajo verno simulacijo delovanja sklopov integriranega vezja. S simulacijo funkcionalnosti posameznega sklopa lahko predvidimo njegov odziv na zunanje vzbujanje, preverimo delovanje sklopa, povezanega z ostalimi sklopi integriranega vezja v večji sistem, lahko predvidimo lahko tudi porabo moči med delovanjem sklopa in posledično trajanje delovanja z baterijskim napajanjem.

Celotno vezje, skupaj s programabilnim jedrom, ROM in RAM pomnilnikom, lahko simuliramo s funkcionalnim simulatorjem (verilog ali VHDL) tako, da v model ROM pomnilnika naložimo datoteko s prevedeno programsko kodo ("personality file"). Takšna simulacija pa ni preveč praktična, saj traja preveč časa, navadno ne omogoča statične časovne analize signalov (ta je odvisna od tehnologije, v kateri bo izvedeno integrirano vezje, ter končne geometrije vezja), zato načrtovalci raje uporabljajo emulacijo mikroprocesorskega jedra.

Proizvajalci ponujajo t.im. ICE emulatorje (In-Circuit Emulator), z razvojno različico mikroprocesorskega jedra. Takšno jedro ima možnost spremljanja notranjih registrov, omogoča izvajanje programa ukaz za ukazom ("single step instruction"), ter omogoča nadzor sklada in pomnilniških lokacij. Uporabniško načrtane digitalne bloke izvedemo s FPGA vezjem, katerega priključimo na zunanja mikroprocesorska vodila emulatorja. Tako lahko preverimo medsebojno delovanje uporabniško načrtanega hardvera (izvedenega s FPGA) in uporabniškega programa, shranjenega v programabilni ROM ICE emulatorja. Z ustreznim "vzbujalnim" vezjem emuliramo signale okolice, odziv emuliranega vezja pa preverimo z analizo stanj ICE emulatorja.

Analogno vezje (MOS prehodna stikala, generatorje analognega signala) izvedemo z diskretnimi elementi - kar pri velikem številu LCD stikal spet ni preveč praktično, navadno pa lahko kar zaupamo simulaciji analognega vezja.

Načrtovanje končne geometrije vezja predstavlja optimalno razporeditev ("floorplaning") vnaprej načrtanih blokov končnih ROM, RAM in ostalih regularnih struktur, ter izvedba in razporeditev uporabniško načrtanih blokov s tehnologijo standardnih celic ali optimiziranih datapath celic.

24. Sklep

Posamezni sklopi integriranega vezja so bili simulirani z verilog simulatorjem, rezultati simulacij pa ustrezajo željeni funkcionalnosti. Posamezne bloke smo izvedli z optimiziranim arhitekturnim opisom na nivoju standardne celice, regularne strukture ROM in RAM sklopov pa smo generirali z generatorji končne geometrije.

Nizkonivojska programska oprema je napisana v programskem jeziku C in prevedena z optimizirajočim prevajalnikom. Procesni model programske opreme je preizkušen s pomočjo verilog simulatorja, izvedene programske rutine pa so preizkušene s simulatorjem mikroprocesorskega jedra Winusim. Tudi tu so rezultati potrdili predvidevanja.

Za popolno izvedbo predstavljenega integriranega vezja je potrebno programski opremi dodati še uporabniški program, ki bo nizkonivojske rutine povezal v funkcionalno celoto in iz predstavljene infrastrukture naredil uporaben izdelek z možnostjo prilagajanja potrebam trga.

25. Zahvala

Zahvaljujem se mentorju prof. dr. Baldomirju Zajcu in kolegom v Laboratoriju za načrtovanje integriranih vezij za izkazano pomoč pri izdelavi magistrske naloge.

Zahvaljujem se tudi podjetju ISKRAEMECO d.d. za možnost uporabe razvojne opreme, ter financiranje mojega magistrskega študija. Ne nazadnje se toplo zahvaljujem tudi sodelavcem Bazičnega razvoja za koristne strokovne nasvete, ter njihovo pomoč pri študiju.

26. Priloge

26.1. Izvorna koda simulacije paralelnih procesov

file: sistem_parallel.v

```
`define CLK_FREQ  4194304

`define TASK_DELAY      4000                // v CLK enotah

`define START_BIT 0
`define STOP_BIT 1
`define RX_delay  (`CLK_FREQ / 9600)       // v CLK enotah

`define LCD_REFRESH     (`CLK_FREQ / 128) // v CLK enotah


module CPUProcesi (Reset, powerfail, dogodek, sched, RX,
                VRAM_addr, VRAM_data);

  input     Reset;       // Power On Reset
  input     powerfail;   // signal prekinitve napajanja
  input     dogodek;     // signal dogodka
  input     sched;       // signal schedulerja
  input     RX;          // vhod serijskega vmesnika

  output    [1:0]        VRAM_addr;  // naslov VRAM
  input     [25:0]       VRAM_data;  // podatki iz VRAM


  // pomnilniske lokacije
  reg [31:0]       st_dogodkov;       // stevec dogodkov

  reg [7:0]        scheduled;  // trenutno opravilo schedulerja
  reg [31:0]       RTC;        // ticki ure realnega casa

  reg [7:0]        RX_data;    // RX premikalni register
  reg              RX_framerr; // signal napake okvirjanja
                               // serjskega vmesnika
  reg [1:0]        LCD_row;    // LCD vrstica
  reg [25:0]       LCD_data;   // LCD stolpec
  reg [1:0]        VRAM_addr;  // naslov VRAM vrstice

  // notranje spremenljivke procesov
  integer stevec_temp;

  integer sched_temp;
  integer sched_ticks;
  integer TASK_ACTIVE;  // zastavica aktivnega opravila

  integer RX_bit;
  integer RX_temp;      // RX pomikalni register
```

```
  integer RX_i;

  integer LCD_row_temp;
  integer LCD_row_data;


// Proces powerfail poklice power-fail proceduro ob
// prekinitvi napajanja. Proces naj izpise opozorilo
// in ustavi simulacija

// proces_powerfail
  always @(posedge powerfail) // izvedi ob prehodu iz 0 -> 1
    begin
      $display("NMI - Power Failure!");
      $stop;
    end


// Proces stevec povecuje stevec ob vsakem dogodku

// proces_stevec
  always @(posedge dogodek)    // izvedi ob prehodu iz 0 -> 1
    begin
      stevec_temp = st_dogodkov;
      stevec_temp = stevec_temp + 1;
      st_dogodkov = stevec_temp;
    end


// Proces RT schedulerja. Scheduler klice RT podprograme
// krozno  po vrstnem redu. Program se mora izvrsiti do naslednje
// prekinitve schedulerja, drugace pride do overrun napake

  task Sched_TaskIdle;         // Prazno opravilo schedulerja
    begin
      #(`TASK_DELAY);          // Ne delaj nic
    end
  endtask

// proces_sched
  always @(posedge sched)      // ob prekinitvi RTC
    begin
      if (TASK_ACTIVE)
      begin
        $display ("NAPAKA! Prejsnje RT opravilo ni koncano!\n");
        $display ("Trenutno opravilo: %d\n", sched_temp);
      end

      sched_temp = scheduled;
      sched_ticks = RTC;

      TASK_ACTIVE = 1;  // Postavi zastavico aktivnega opravila
      case (sched_temp)        // Klic opravila
      0:  Sched_TaskIdle;
      1:  Sched_TaskIdle;
      2:  Sched_TaskIdle;
      3:  Sched_TaskIdle;
```

```
      4:  Sched_TaskIdle;
      5:  Sched_TaskIdle;
      6:  Sched_TaskIdle;
      7:  Sched_TaskIdle;
      endcase
      TASK_ACTIVE = 0;          // Opravilo se je koncalo pravocasno

      sched_temp = sched_temp + 1;      // Naslednje opravilo
      sched_ticks = sched_ticks + 1;  // povecaj uro realnega casa

      if (sched_temp >= 8)
      scheduled = 0;
      else
        scheduled = sched_temp;

      RTC = sched_ticks;
    end


// Proces sprejema podatkov preko serijskega vmesnika

// proces_RX
  always @(RX == `START_BIT)  // startaj ob nivoju zacetnega bita
    begin
      #(3 * `RX_delay / 2);   //  RX_delay start bita in polovica
                              //  naslednjega bita
      for (RX_i = 0; RX_i < 8; RX_i = RX_i + 1)
      begin
          RX_bit = RX;        // preberi RX
        RX_temp = (RX_temp << 1) | RX_bit;      // premakni v levo
                              // in dodaj RX_bit
        #(`RX_delay);         // cakaj RX_delay
      end

      RX_bit = RX;                    // preberi stop bit
      if (RX_bit == `STOP_BIT)        // ce je STOP_BIT
      begin
        RX_framerr = 0;       // okvir je pravilen
        RX_data = RX_temp;    // shrani prejeti byte
      end
      else
      RX_framerr = 1;         // signaliziraj napako okvirja
    end


// Proces osvezevanja LCD prikazovalnika

//proces_LCD
  always #(`LCD_REFRESH / 4)  // obnovitev posamezne vrstice
    begin
      VRAM_addr = LCD_row_temp;     // postavi VRAM_addr in LCD_row
      LCD_row     = LCD_row_temp;   // na vrednost trenutne vrstice

      #10               // Min. zakasnitev med dvema operacijama
                        // VRAM dostopni cas

      LCD_row_data = VRAM_data;     // preberi vrstico iz VRAM
```

```
      LCD_data = LCD_row_data;       // poslji vrstico na display

      LCD_row_temp = LCD_row_temp + 1;     // naslednja vrstica

      if (LCD_row_temp >= 4)  // krozno povecaj stevec vrstic
      LCD_row_temp = 0;

    end


//------
// RESET

  always @(Reset == 0)
    begin
      $display ("RESET");

      // Postavi notranja stanja na 0
      stevec_temp = 0;
      sched_temp = 0;
      sched_ticks = 0;

      TASK_ACTIVE = 0;
      RX_bit = 0;
      RX_temp = 'hFF;
      RX_i = 0;

      LCD_row_temp = 0;

      // Postavi pomnilniske lokacije na 0
      st_dogodkov = 0;
      scheduled = 0;
      RTC = 0;

      // Cakaj na konec Reset signala
      wait (Reset == 1);
  end


// inicializacija verilogovih sistemskih opravil
  initial
    begin

      $gr_waves ("RESET", Reset,
             "dogodki", st_dogodkov,
             "task", scheduled, "RTC", RTC,
             "RX_data", RX_data, "RX_framerr", RX_framerr,
             "LCD_row", LCD_row, "LCD_data", LCD_data,
             "VRAM_addr", VRAM_addr,
             "RX", RX);
    end
endmodule
```

file: sistem_parallel_test.v

```
`define CLK_FREQ  4194304

`define SCHED_T          (`CLK_FREQ / 1024) // v CLK enotah
`define DOGODEK_T        (`CLK_FREQ / 100) // v CLK enotah
`define RX_T             (`CLK_FREQ / 900) // v CLK enotah
`define POWERFAIL_T      300000            // v CLK enotah

`define RX_delay  (`CLK_FREQ / 9600)       // v CLK enotah

`define START_BIT 0
`define STOP_BIT 1


module CPUProcesi_test;
  reg       Reset;
  reg       powerfail;
  reg       dogodek;
  reg       sched;
  reg       RX;

  wire      [1:0]       VRAM_addr;
  wire      [25:0]      VRAM_data;


  // Video RAM registri
  reg       [25:0]      VRAM [0:3];


  CPUProcesi Procesor1 (
      Reset,
      powerfail,
      dogodek,
      sched,
      RX,
      VRAM_addr,
      VRAM_data);

  // Video RAM
  assign VRAM_data = VRAM[VRAM_addr];

// glavni testni program
initial
  begin
      Reset = 0;          // Power on Reset
      powerfail = 0;
      dogodek = 0;
      sched = 0;
      RX = 1;                    // RS232 mirovno stanje

      VRAM[0] = 'hC0FFEE;
      VRAM[1] = 'h123456;
      VRAM[2] = 'h654321;
      VRAM[3] = 'h00BABE;
```

```
    #10       Reset = 1;
    #(`POWERFAIL_T)
       powerfail = 1;           // koncaj s powerfail
  end

// Prekinitve schedulerja
always #(`SCHED_T)
  begin
       sched = 1;
    #10       sched = 0;
  end

// Dogodki
always #(`DOGODEK_T)
  begin
       dogodek = 1;
    #10       dogodek = 0;
  end

// Serijski prenos
always #(`RX_T)
  begin
       RX = `START_BIT;
    #(`RX_delay)
       RX = 'b0;
    #(`RX_delay)
       RX = 'b1;
    #(`RX_delay)
       RX = 'b1;
    #(`RX_delay)
       RX = 'b0;
    #(`RX_delay)
       RX = 'b1;
    #(`RX_delay)
       RX = 'b0;
    #(`RX_delay)
       RX = 'b0;
    #(`RX_delay)
       RX = 'b1;
    #(`RX_delay)
       RX = `STOP_BIT;
  end

endmodule
```

26.2. Izvorna koda simulacije sistema s prekinitvami

file: sistem_irq.v

```
// Globalne definicije

`define CLK_FREQ  4194304
`define CNTUP_DELAY     50    // zakasnitev stevca dogodkov v CLK
`define SCHED_DELAY     15    // overhead razdeljevalnika v CLK
`define TASK_DELAY      3900  // dolzina softverskega opravila v CLK
`define FIFO_DELAY      100   // zakasnitev FIFO rutine serijskega
                              // vmesnika

`define serial_RX 1     // identifikacija cakalne vrste
`define FIFO 1          // FIFO cakalna vrsta
`define FIFO_LEN_RX 16  // Dolzina FIFO vrste serijskega vmesnika


module CPUPrekinitve (Reset, PowerFAIL_NMI, CountUp_IRQ,
                      Scheduler_IRQ, Serial_IRQ );
  input Reset;          // Power on Reset signal
  input PowerFAIL_NMI;  // Prekinitev nadzora napajanja
  input CountUp_IRQ;    // Prekinitev stevca dogodkov
  input Scheduler_IRQ;  // Prekinitev razdeljevalnika opravil
  input Serial_IRQ;     // Prekinitev serijskega vmesnika

  reg IRQ_Enable_Flag;  // Zastavica omogocenih prekinitev

  reg pending_PowerFAIL_NMI;  // Zastavice prekinitev v teku
  reg pending_CountUp_IRQ;
  reg pending_Scheduler_IRQ;
  reg pending_Serial_IRQ;

  reg active_PowerFAIL; // Zastavice aktivnih procesov
  reg active_CountUp;   // Za prikaz procesov v oknu
  reg active_Scheduler;
  reg active_Serial;

  reg [31:0] COUNTER;   // Stevec dogodkov

  reg        TASK_ACTIVE;// Zastavica aktivnega RT podprograma
  reg [2:0] SCHED_TASK; // Indeks trenutno aktivnega RT podprograma

  integer   line_len_RX;// Dolzina sprejemne FIFO vrste s. vmesnika
  integer   status_RX;  // Status sprejemne FIFO vrste s. vmesnika

  integer temp;

initial
begin

$gr_waves ("IRQ_EN ", IRQ_Enable_Flag, "P_FAI 0", PowerFAIL_NMI,
"pending", pending_PowerFAIL_NMI, "active", active_PowerFAIL, "C_IRQ
1", CountUp_IRQ, "pending", pending_CountUp_IRQ, "active",
active_CountUp, "Dogodki", COUNTER, "S_IRQ 2", Scheduler_IRQ,
"pending", pending_Scheduler_IRQ, "active", active_Scheduler, "TASK",
```

```
SCHED_TASK, "SeIRQ 3", Serial_IRQ, "pending", pending_Serial_IRQ,
"active", active_Serial, "line_len_RX", line_len_RX);


  // Zacetno stanje zastavic

  pending_PowerFAIL_NMI = 0;  // Zastavice prekinitev v teku
  pending_CountUp_IRQ = 0;
  pending_Scheduler_IRQ = 0;
  pending_Serial_IRQ = 0;

  active_PowerFAIL = 0; // Zastavice aktivnih procesov
  active_CountUp = 0;
  active_Scheduler = 0;
  active_Serial = 0;

  IRQ_Enable_Flag = 1;  // Prekinitve omogocene

  // inicializacija sprejemne (RX) cakalne vrste ser. vmesnika
  $q_initialize(`serial_RX, `FIFO, `FIFO_LEN_RX, status_RX);
  if (status_RX != 0)
  begin
    $display ("NAPAKA pri tvorjenju RX cakalne vrste!");
    $finish;
  end

  // Inicializacija line_len_RX
  $q_exam(`serial_RX, 1, line_len_RX, status_RX);

end

//------
// RESET
  always @(Reset == 0)        // Pocakaj, da mine Power on Reset
    begin
      $display ("RESET");

      TASK_ACTIVE = 0;
      SCHED_TASK = 0;
      COUNTER = 0;

      // Cakaj na konec Reset signala
      wait (Reset == 1);
  end


// Zanke IRQ zastavic in pasti zgresenih prekinitev

always @(posedge PowerFAIL_NMI)
  begin
    if (pending_PowerFAIL_NMI)      //
      $display ("PowerFAIL_NMI prekinitev ni bila servisirana!");
    pending_PowerFAIL_NMI = 1;
  end

always @(posedge CountUp_IRQ)
```

```
  begin
    if (pending_CountUp_IRQ)  //
      $display ("CountUp_IRQ prekinitev ni bila servisirana!");
    pending_CountUp_IRQ = 1;
  end

always @(posedge Scheduler_IRQ)
  begin
    if (pending_Scheduler_IRQ)      //
      $display ("Scheduler_IRQ prekinitev ni bila servisirana!");
    pending_Scheduler_IRQ = 1;
  end

always @(posedge Serial_IRQ)
  begin
    if (pending_Serial_IRQ)   //
      $display ("Serial_IRQ prekinitev ni bila servisirana!");
    pending_Serial_IRQ = 1;
  end

// Glavna "prekinitvena zanka" mikroprocesorja
always wait(IRQ_Enable_Flag && (pending_PowerFAIL_NMI ||
                        pending_Scheduler_IRQ ||
                        pending_CountUp_IRQ ||
                        pending_Serial_IRQ))
  begin
    IRQ_Enable_Flag = 0;      // Rutina onemogoci vse prekinitve
    if (pending_PowerFAIL_NMI)
    begin
      active_PowerFAIL = 1;   // Postavi zastavico
      NMI_PowerFailure;       // Izvedi proces -> konec
    end
    if (pending_CountUp_IRQ)
    begin
      active_CountUp = 1;     // Postavi zastavico
      IRQ_CountUp;            // Izvedi proces
      active_CountUp = 0;     // Podri zastavico
      pending_CountUp_IRQ = 0;      // Opravljena prekinitvena rutina
    end
    if (pending_Scheduler_IRQ)
    begin
      active_Scheduler = 1;   // Postavi zastavico
      IRQ_Scheduler;          // Izvedi proces
      active_Scheduler = 0;   // Podri zastavico
      pending_Scheduler_IRQ = 0;    // Opravljena prekinitvena rutina
    end
    if (pending_Serial_IRQ)
    begin
      active_Serial = 1;      // Postavi zastavico
      IRQ_Serial;       // Izvedi proces
      active_Serial = 0;      // Podri zastavico
      pending_Serial_IRQ = 0; // Opravljena prekinitvena rutina
    end

    IRQ_Enable_Flag = 1;      // Omogoci prekinitve na koncu
                        // servisne rutine
  end
```

```
// Prazno opravilo RT razdeljevalnika, dolzine MAX_TASK_CLK

  task Sched_TaskIdle;
    begin
      #(`TASK_DELAY);
    end
  endtask


// Preberi byte iz FIFO serijskega vmesnika
  task Sched_SerialReadQueue;
    begin
      // Ce je dolzina cakalne vrste > 0, obdelaj vrsto
      $q_exam(`serial_RX, 1, line_len_RX, status_RX);
      for (temp = line_len_RX; temp != 0; temp = temp - 1)
      begin
      #(`FIFO_DELAY)          // Zakasnitev FIFO podprograma
      $q_remove(`serial_RX, 0, 0, status_RX);

      // Osvezi line_len_RX
      $q_exam(`serial_RX, 1, line_len_RX, status_RX);
      end
    end
  endtask

// PowerFailure je ne-realnocasovni podprogram, ki se izvede ob
// prekinitvi napajanja. Naloga podprograma je, da shrani vse
// pomembne spremenljivke v baterijski RAM

// Ob klicu PowerFailure naj se izpise opozorilo in
// ustavi simulacija

  task NMI_PowerFailure;
    begin
      $display("NMI - Power Failure!\n");
      $stop;
    end
  endtask


// Stevec

  task IRQ_CountUp;
    begin
      #(`CNTUP_DELAY)          // Zakasnitev podprograma
      COUNTER = COUNTER + 1;  // Povecaj stevec dogodkov
    end
  endtask


// Prekinitev RT razdeljevalnika opravil. Razdeljevalnik klice RT
// podprograme po staticno doloceni prioriteti. Program se mora
// izvrsiti do naslednje prekinitve razdeljevalnika, drugace
// signalizira overrun napako
```

```
  task IRQ_Scheduler;
    #(`SCHED_DELAY)      // Zakasnitev klica razdeljevalnika
    begin
      if (TASK_ACTIVE)
      begin
        $display ("NAPAKA! Prejsnje RT opravilo ni koncano!");
        $display ("Trenutno opravilo: %d\n", SCHED_TASK);
      end

      TASK_ACTIVE = 1;  // Postavi zastavico aktivnega opravila
      case (SCHED_TASK)
      0:  Sched_TaskIdle;
      1:  Sched_TaskIdle;
      2:  Sched_SerialReadQueue;
      3:  Sched_TaskIdle;
      4:  Sched_TaskIdle;
      5:  Sched_TaskIdle;
      6:  Sched_TaskIdle;
      7:  Sched_TaskIdle;
      endcase

      SCHED_TASK = SCHED_TASK + 1;  // Naslednji task
      TASK_ACTIVE = 0;              // Task se je koncal do casa
    end
  endtask

  task IRQ_Serial;
    begin
      // Dodaj element v cakalno vrsto
      #(`FIFO_DELAY)           // Zakasnitev FIFO podprograma
      $q_add (`serial_RX, 0, 0, status_RX);
      if (status_RX != 0)
      begin
        $display ("OVERRUN pri dodajanju elementa v RX!\n");
        $stop;
      end
      // Osvezi line_len_RX
      $q_exam(`serial_RX, 1, line_len_RX, status_RX);
    end
  endtask

endmodule
```

file: sistem_irq_test.v

```
`define CLK_FREQ  4194304
`define SCHED_T          (`CLK_FREQ / 1024)       // v CLK enotah
`define DOGODEK_T (`CLK_FREQ / 100) // v CLK enotah
`define RX_T             (`CLK_FREQ / 960) // v CLK enotah
`define POWERFAIL_T      300000                   // v CLK enotah

module CPUPrekinitve_test;
  reg       Reset;
  reg       powerfail;
  reg       dogodek;
  reg       sched;
  reg       serial;

  CPUPrekinitve Procesor1 (
      Reset,
      powerfail,
      dogodek,
      sched,
      serial);

// glavni testni program
initial
  begin
      Reset = 0;         // Power on Reset
      powerfail = 0;
      dogodek = 0;
      sched = 0;
      serial = 0;

   #10      Reset = 1;
   #(`POWERFAIL_T)
      powerfail = 1;          // koncaj s powerfail
  end

// Dogodki
always #(`DOGODEK_T)
  begin
      dogodek = 1;
   #10      dogodek = 0;
  end

// Prekinitve schedulerja
always #(`SCHED_T)
  begin
      sched = 1;
   #10      sched = 0;
  end

// Serijski prenos
always #(`RX_T)
  begin
      serial = 1;
   #10      serial = 0;
  end
endmodule
```

26.3. Karakterizacija ROM celice

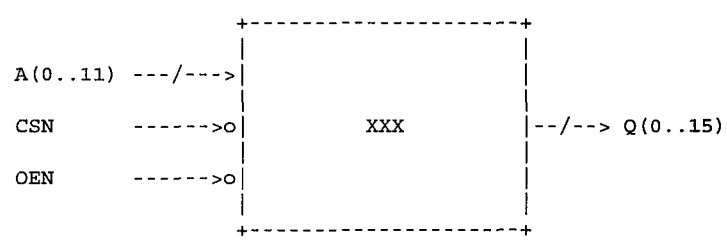
```
+------------+-----------------------------------------+--------------------+
| CB22000    |       Fully static synchronous ROM      | Release 1.0        |
| c22_romgen |                                         | January 10th, 1993 |
+------------+-----------------------------------------+--------------------+
```

SECTION 1. FEATURES

- fully static operation
- synchronous operation
- fast access time (15.0 ns worst case for 2048 x 8)
- low power (no static power consumption)
- up to 64K bits total capacity
- word width between 2 and 32 bits (limited conditions)
- word number in the range of 32 to 8192
- max density of 13000 bits/mm2 (2048 X 16 mux 16)
- supply voltage 5V+/-10%, but functional down to 3.3V+/-10%
- selectable aspect ratio (three options available)
- contact programmable to enable rapid design changes
- HCMOS4T process technology with triple metal, silicide POLY.

SECTION 2. SYMBOL

ROM 4096X16 with mux size = 16



SECTION 3. SIGNAL DESCRIPTION

n = number of bits
m = number of words

Signal name	I/O	FUNCTION
CSN	IN	Chip select (active low)
OEN	IN	Output enable (actibe low)
A0, ..., A(n-1)	IN	Address bus A0 : LSB A(n-1) : MSB
Q0, ..., A(m-1)	OUT	Data ouput Q0 : LSB Q(m-1) : MSB

SECTION 4. DIMENSIONS AVAILABLE

col = mux * bits (max 256)
row = words / mux (max 256)

PARAMETERS	MIN. LIMIT	MAX. LIMIT	STEP	VALUES
words	32	8192	words = words+2mux	4096
bits	2	32	bits = bits +1	16
mux	8	32	mux = mux*2	16
col	mux	256		
row	4	256		
capacity	64	65536		

SECTION 5. AC/DC CHARACTERISTICS

TYPICAL PROCESS, TEMPERATURE=25 degrees, VCC=5.0V, CL=0pF

SUBSECTION 5.1 TIMING TABLE

spec	symbol	min	typ	max	unit
precharge time	tpr	3.900			ns
access time	taa			10.000	ns
address setup	tas	1.000			ns
address hold	tah	0.500			ns
OEN to low impedance	tlz	0.100			ns
OEN to high impedance	thz			0.700	ns
CSN high to data high	to1			3.500	ns
data valid after CSN	toh	0.500			ns

SUBSECTION 5.2 CONSUMPTION

average consumption (ICC) in mA = 0.100 + 3.110 * f (MHz)
f = Frequency

SUBSECTION 5.3 SENSITIVITY

output load sensitivity (Ks) = 0.022 ns/SL
Ks applies to taa and to1 : sum (Ks * CL) to intrinsic

SUBSECTION 5.4 INPUT/OUTPUT LOAD

Pin Name	I/O	Load (SL)
Q[16]	OUT	1
A[12]	IN	3
CSN	IN	5
OEN	IN	2

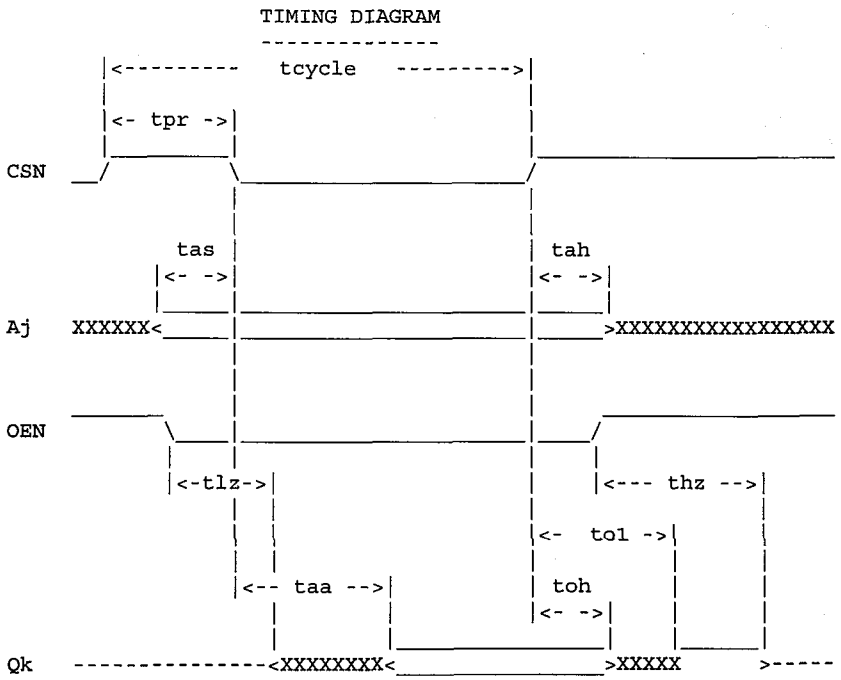
SUBSECTION 5.5 OUTPUT DRIVE

output drive = 32 SL

SUBSECTION 5.6 CELL AREA

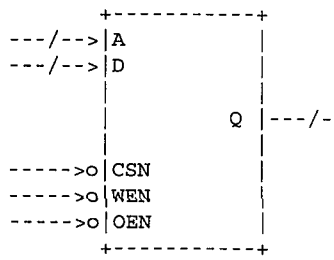
Height = 2260.500
Length = 1665.200
Area = 3764184.600 um2

SECTION 6. WAVEFORMS



26.4. Karakterizacija RAM celice

```
+------------+-----------------------------------------+------------+
|  CB22000   |          FULLY STATIC ARCHITECTURE      |  CB22000   |
|  SPRAMGEN  |               SINGLE PORT RAM           |  SPRAMGEN  |
+------------+-----------------------------------------+------------+
```



unit name = SP256X8M8
number of words = 256
number of data bits = 8
number of mux inputs = 8
mode of operation = GENERIC

AC/DC SPECS (TYPICAL PROCESS, TEMPERATURE=25 degrees, VCC=5.0V)

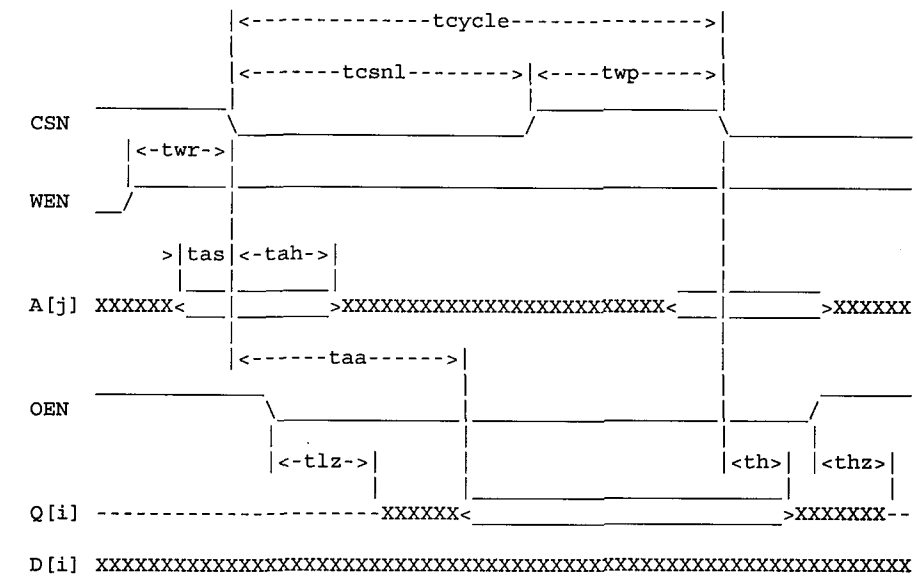
spec	symbol	min	typ	max	unit
precharge time	twp	1.40			ns
CSN low pulse width	tcsnl	1.51			ns
cycle time	tcycle	8.77			ns
access time	taa			5.43	ns
address setup	tas	1.00			ns
address hold	tah	1.51			ns
write pulse width	tw	2.29			ns
write-thru access time	tac			2.85	ns
WEN recovery to CSN	twr	5.13			ns
data setup	tds	1.33			ns
data hold	tdh	0.96			ns
OEN to low impedance	tlz	0.48			ns
OEN to high impedance	thz	0.73			ns
data valid after CSN	th	3.36			ns
WEN low after CSN	twh	1.00			ns

average consumption	ICC	0.10 + 0.37 * f (MHz)	mA
output load sensitivity	Ks	0.013	ns/SL

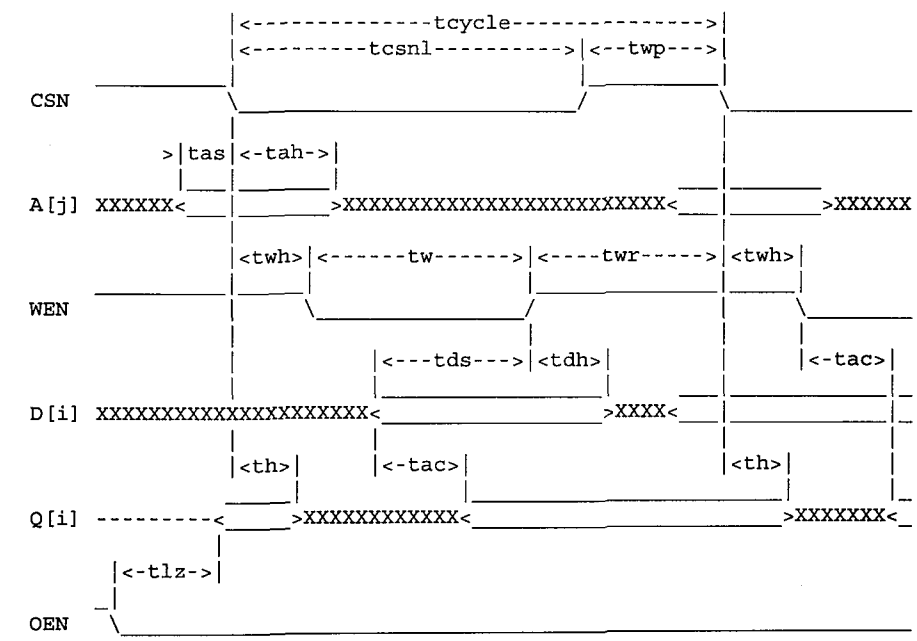
N.B.: Ks applies to taa,tac : sum (Ks*CL) to intrinsic

CELL AREA (um)		CELL LOAD (SL)		OUTPUT
L	H	INPUT	OUTPUT	DRIVE (SL)
1284	1035	17	3	96

MODE = GENERIC, READ CYCLE



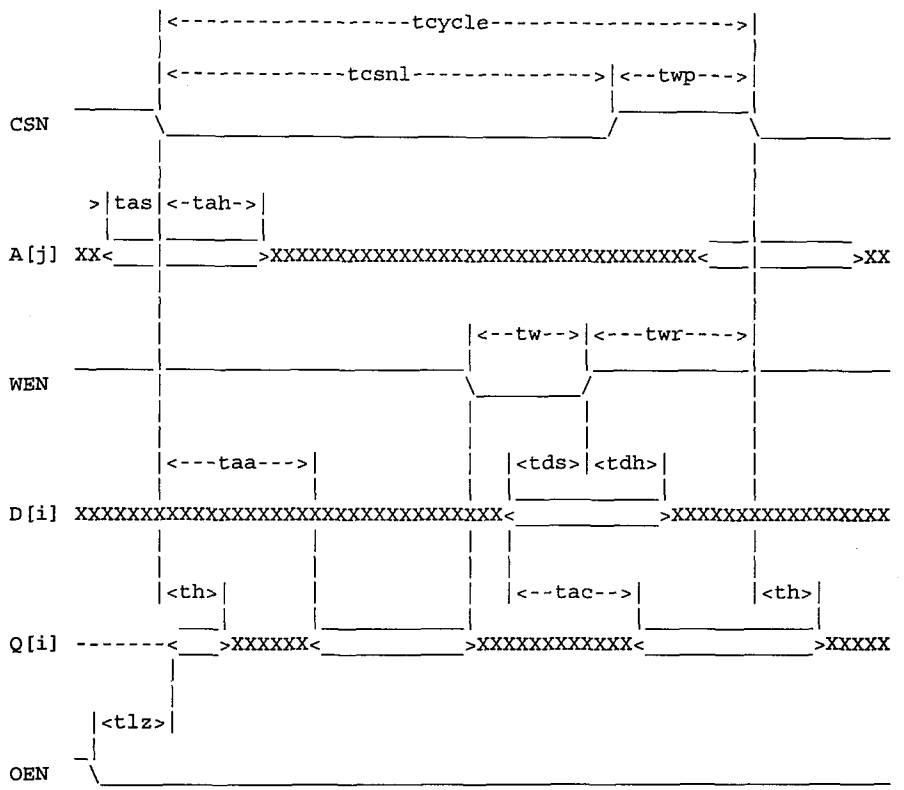
MODE = GENERIC, WRITE CYCLE



NOTE : tw full pulse width must elapse before CSN goes high,
otherwise WRITE cycle is CSN controlled.

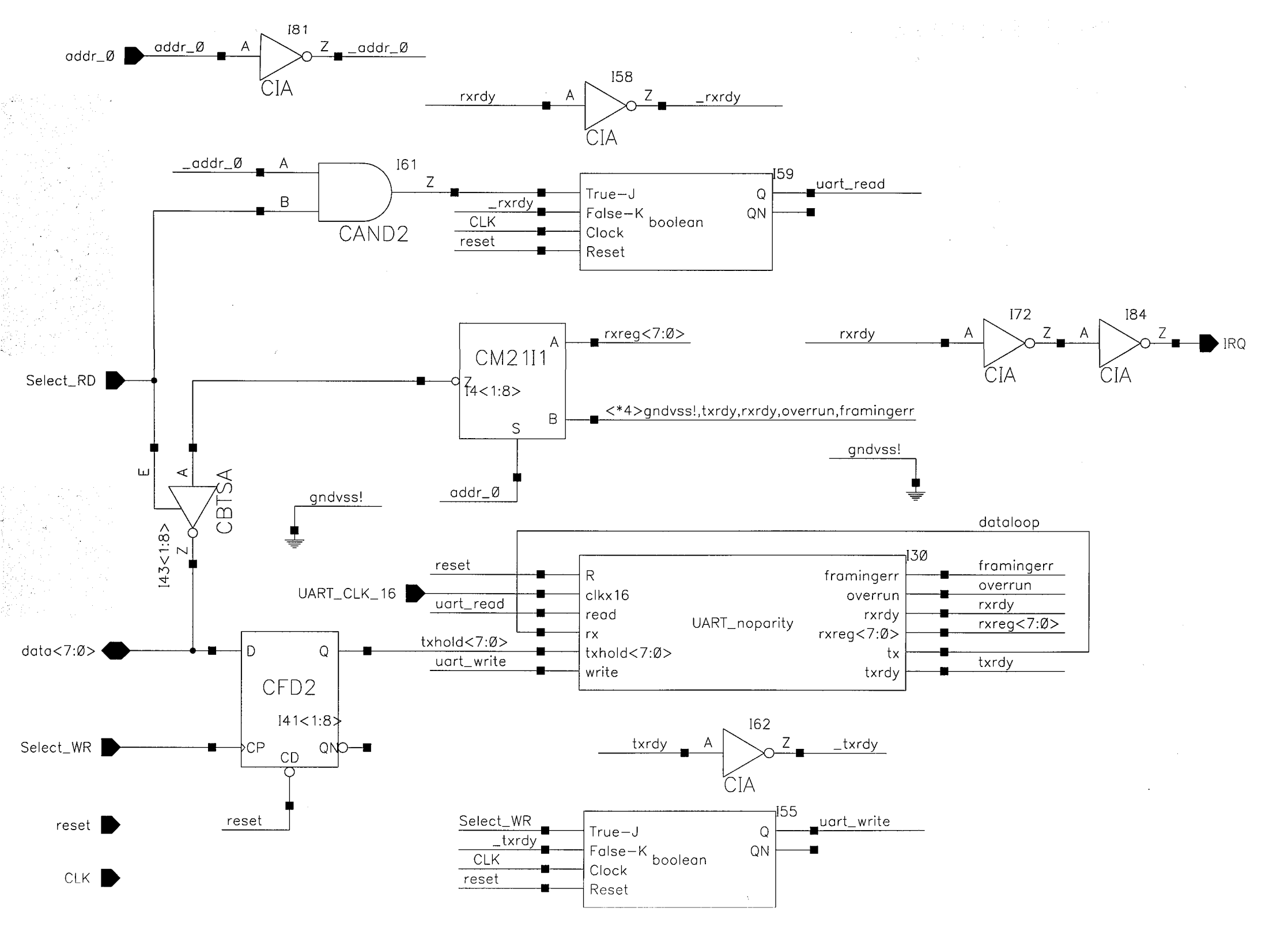
CB22000 SPRAMGEN	FULLY STATIC ARCHITECTURE SINGLE PORT RAM	CB22000 SPRAMGEN

MODE = GENERIC, READ-MODIFY-WRITE CYCLE

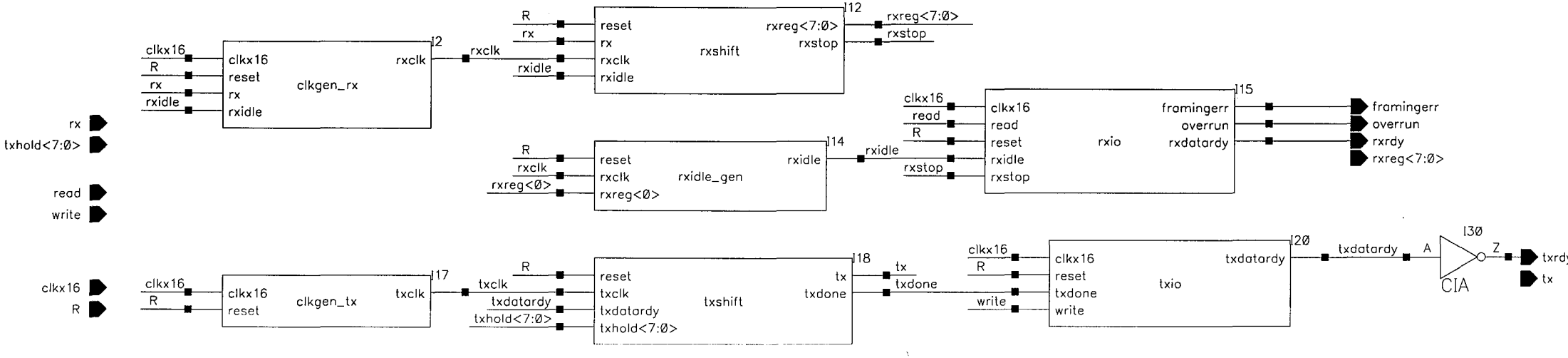


26.5. Shema UART vmensika

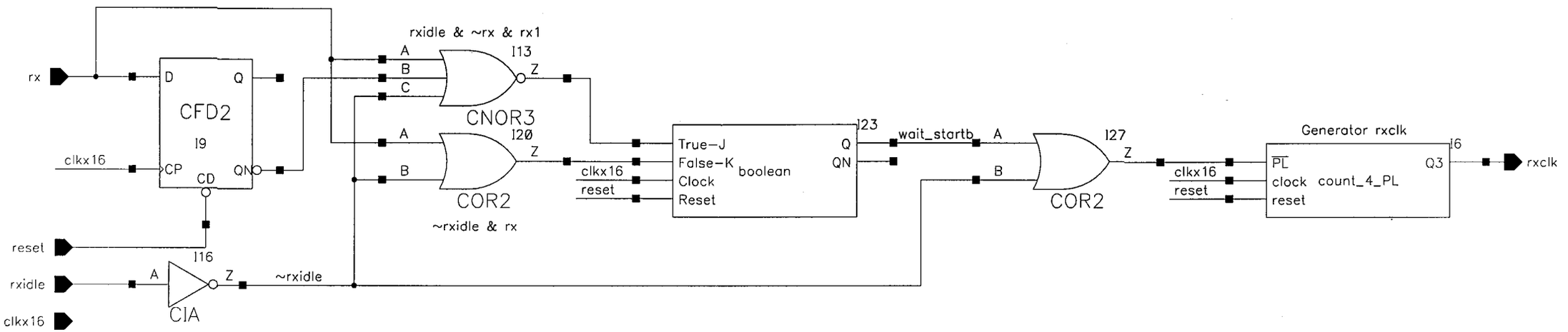
(na naslednjih straneh)


addr_Ø
addr_Ø
A
I81
Z
_addr_Ø
CIA
rxrdy
A
I58
Z
_rxrdy
CIA
_addr_Ø
A
I61
B
Z
CAND2
_rxrdy
CLK
reset
True-J
False-K
Clock
Reset
boolean
I59
Q
QN
uart_read
CM21I1
A
rxreg<7:0>
Z
I4<1:8>
S
B
<*4>gndvss!,txrdy,rxrdy,overrun,framingerr
rxrdy
A
I72
Z
A
I84
Z
IRQ
CIA
CIA
Select_RD
E
A
CBTSA
Z
I43<1:8>
gndvss!
addr_Ø
gndvss!
dataloop
reset
UART_CLK_16
uart_read
R
clkx16
read
rx
txhold<7:0>
write
UART_noparity
I30
framingerr
overrun
rxrdy
rxreg<7:0>
tx
txrdy
framingerr
overrun
rxrdy
rxreg<7:0>
txrdy
data<7:0>
D
Q
CFD2
I41<1:8>
txhold<7:0>
uart_write
Select_WR
CP
CD
QN
reset
reset
txrdy
A
I62
Z
_txrdy
CIA
CLK
Select_WR
_txrdy
CLK
reset
True-J
False-K
Clock
Reset
boolean
I55
Q
QN
uart_write

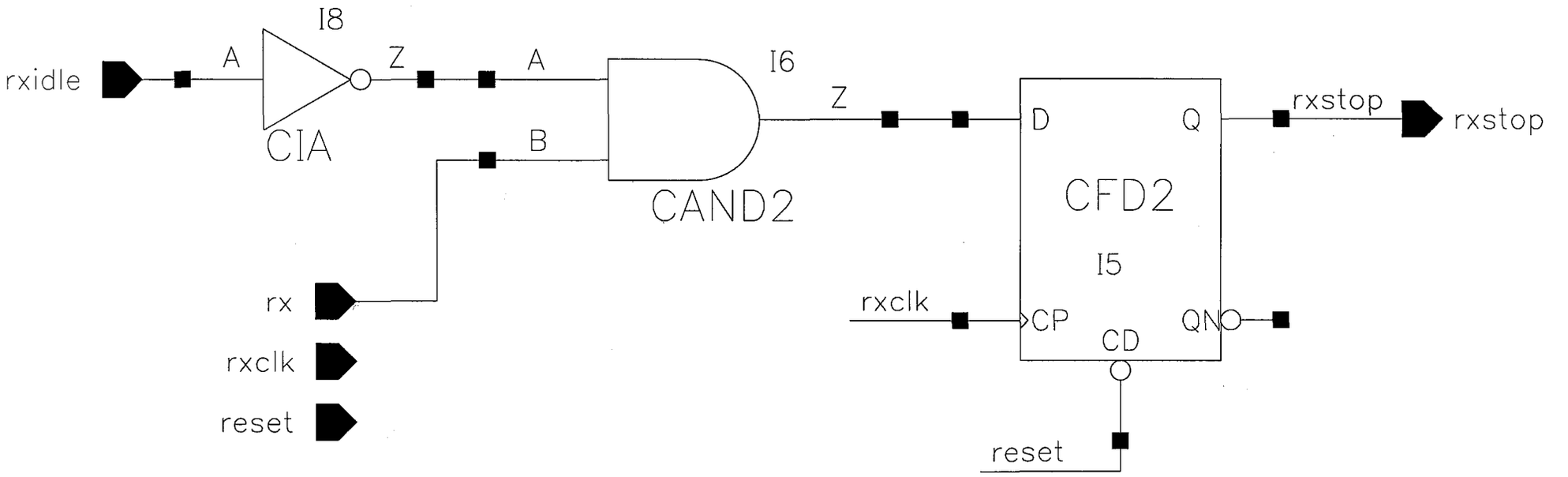
UART_noparity

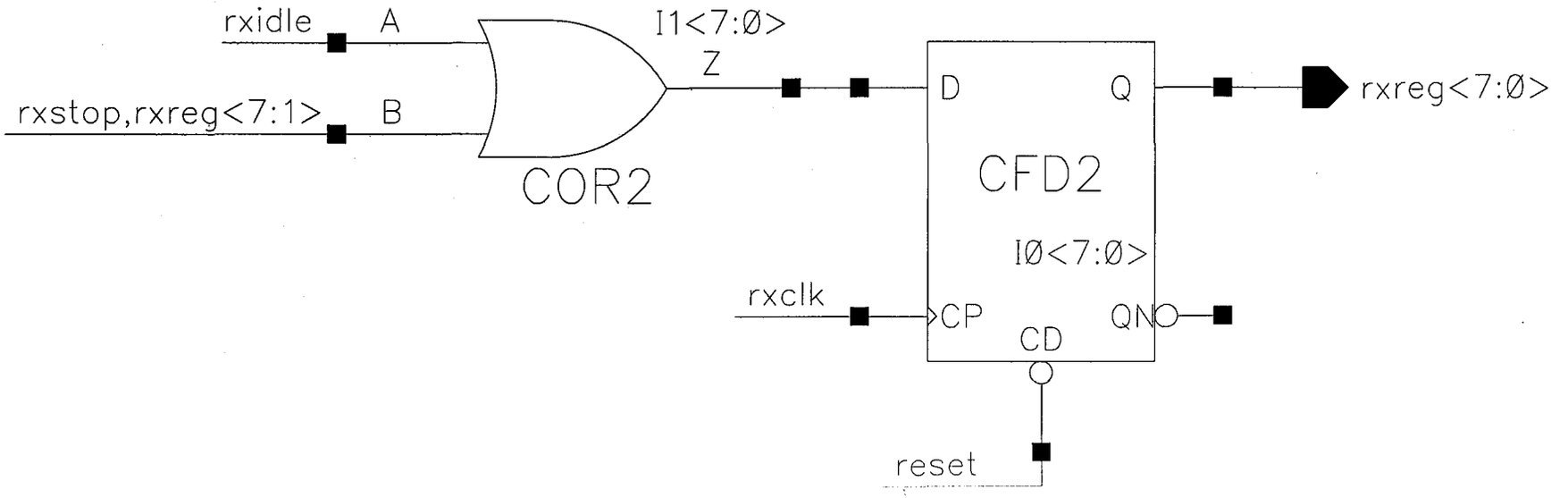


clkgen_rx

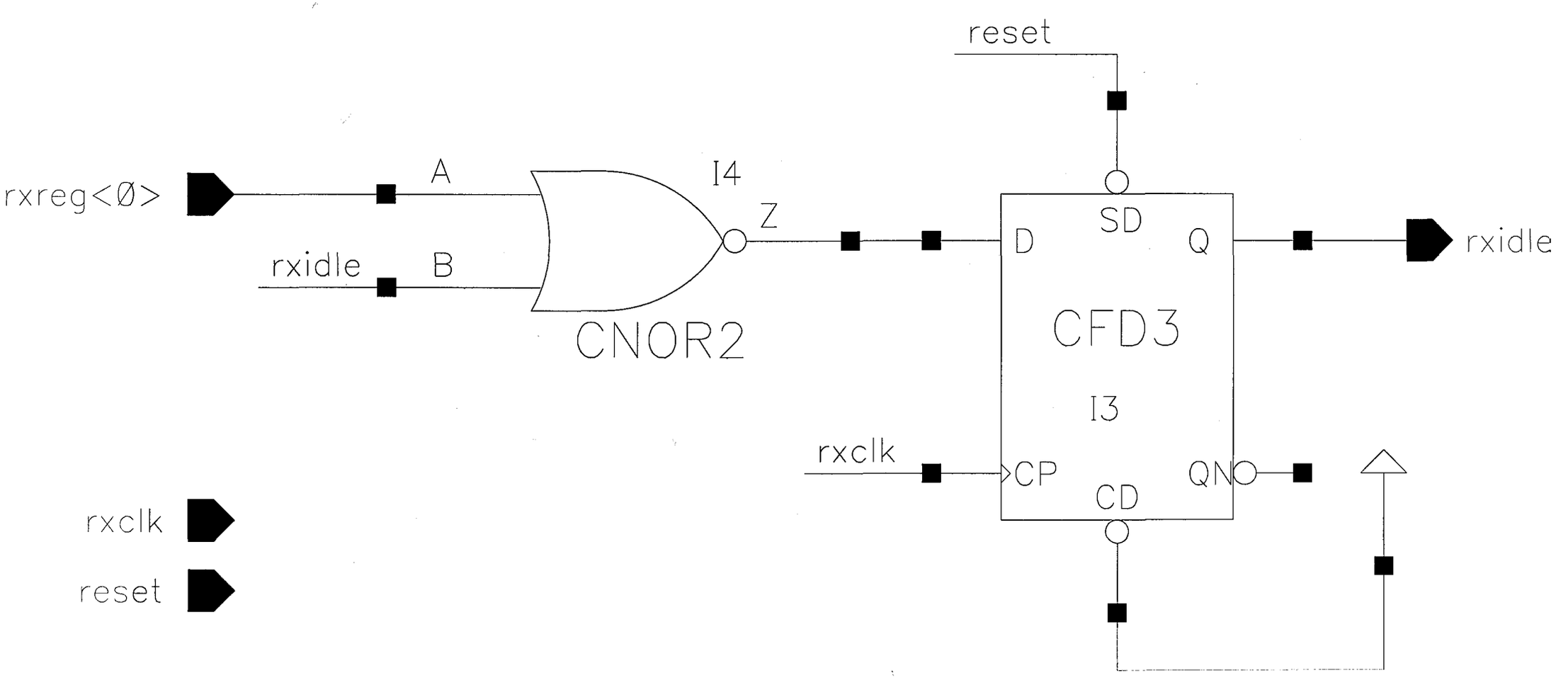

rx
clkx16
reset
rxidle
clkx16
D
Q
CFD2
I9
CP
CD
QN
I16
A
Z
CIA
~rxidle
rxidle & ~rx & rx1
A
B
C
I13
Z
CNOR3
A
B
I20
Z
COR2
~rxidle & rx
clkx16
reset
True-J
False-K
Clock
Reset
boolean
Q
QN
I23
wait_startb
A
B
I27
Z
COR2
clkx16
reset
Generator rxclk
PL
clock
reset
count_4_PL
Q3
I6
rxclk

rxshift

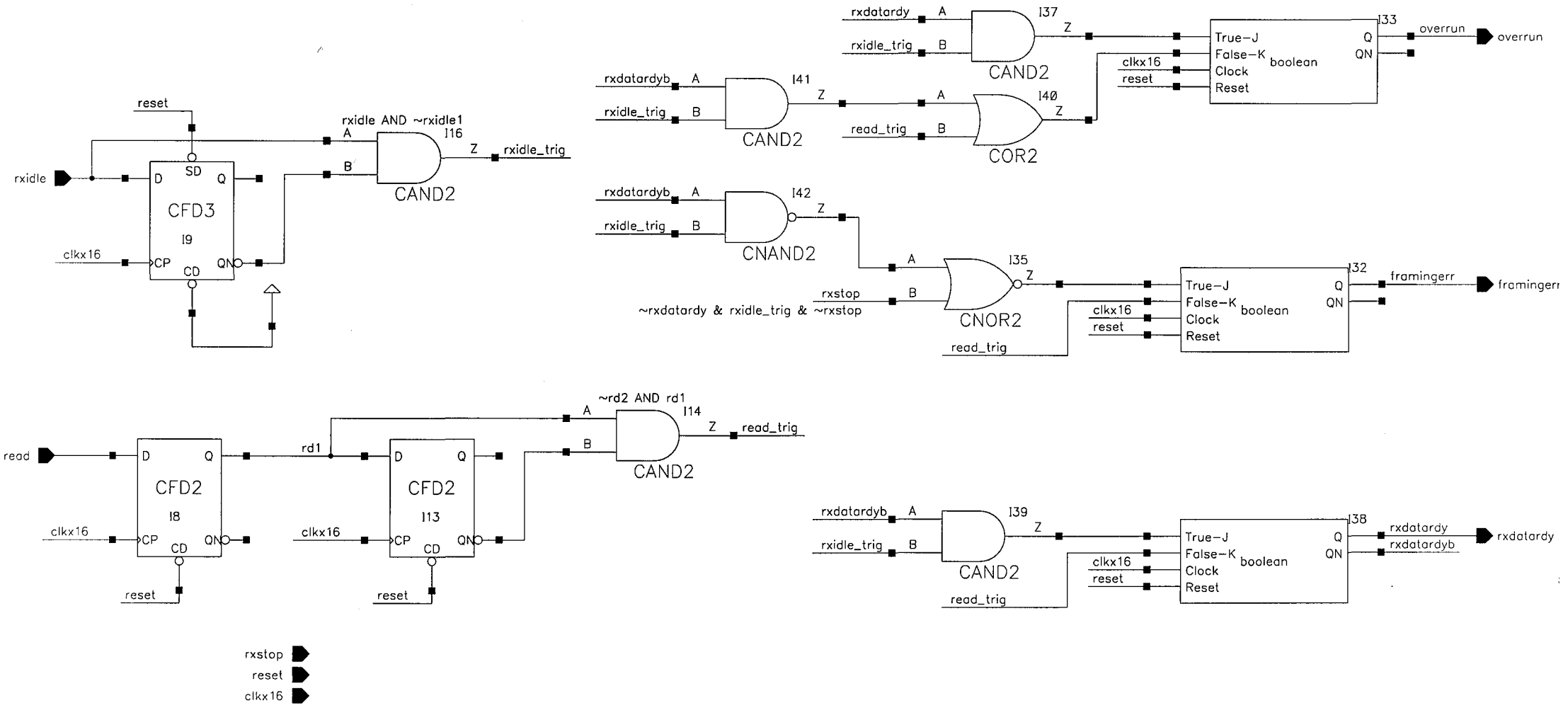

I8
rxidle
A
Z
CIA
A
B
I6
CAND2
Z
D
Q
CFD2
I5
rxstop
rxstop
rx
rxclk
reset
rxclk
CP
QN
CD
reset


rxidle
A
rxstop,rxreg<7:1>
B
COR2
I1<7:Ø>
Z
D
Q
CFD2
IØ<7:Ø>
rxreg<7:Ø>
rxclk
CP
QN
CD
reset

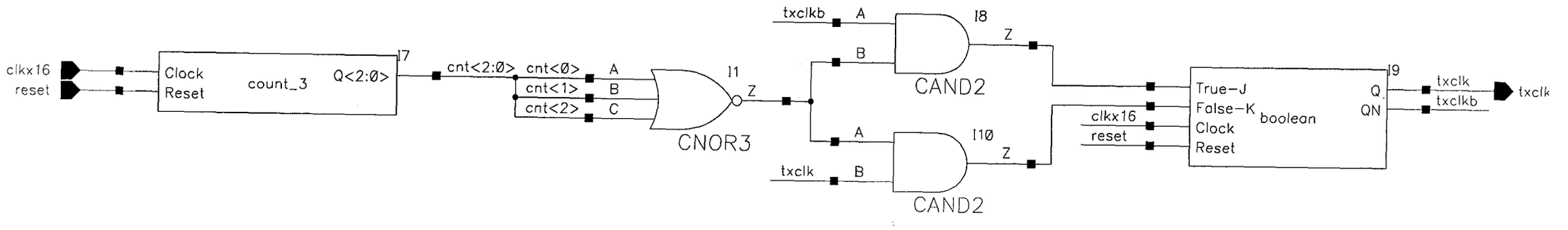
rxidle_gen


reset
rxreg<Ø>
A
I4
Z
SD
D
Q
rxidle
rxidle
B
CNOR2
CFD3
I3
rxclk
CP
QN
CD
rxclk
reset

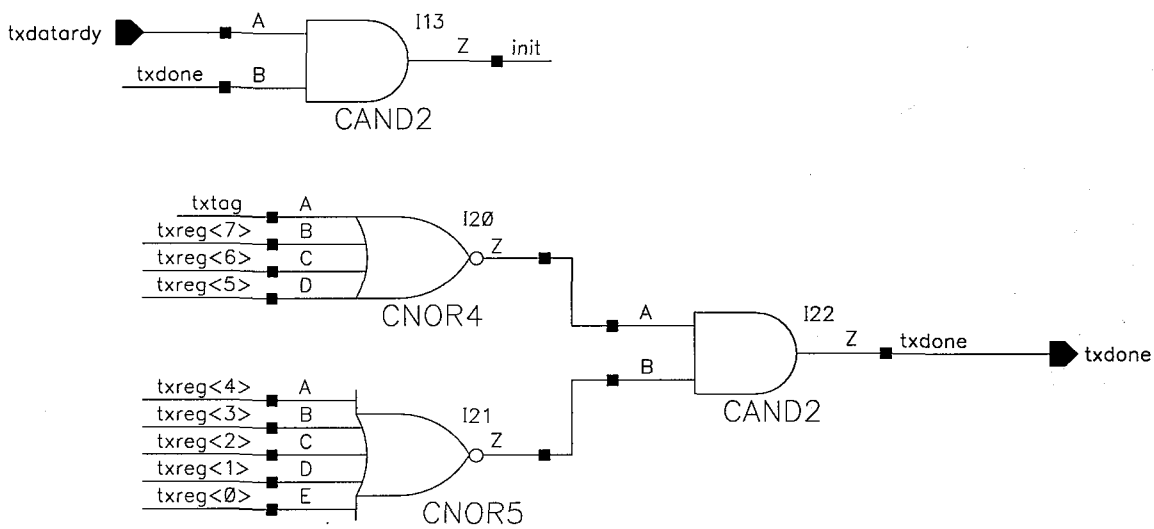
rxio

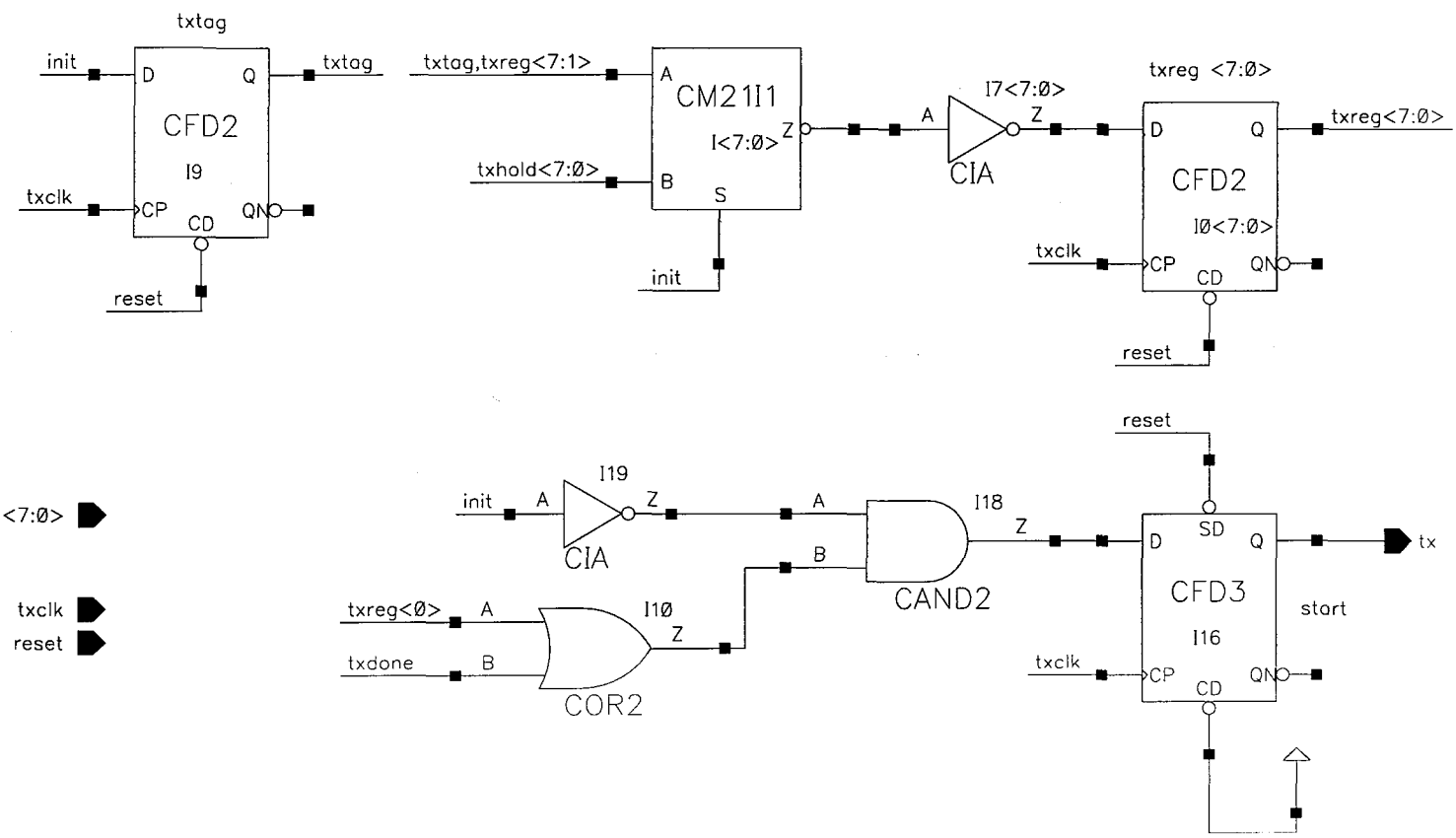

reset
rxidle AND ~rxidle1
I16
A
B
Z
rxidle_trig
rxidle
D
SD
Q
CFD3
I9
clkx16
CP
CD
QN
rxdatardy
I37
rxidle_trig
CAND2
rxdatardyb
I41
rxidle_trig
CAND2
read_trig
I40
COR2
True-J
False-K
Clock
Reset
boolean
clkx16
reset
I33
overrun
overrun
rxdatardyb
I42
rxidle_trig
CNAND2
rxstop
~rxdatardy & rxidle_trig & ~rxstop
I35
CNOR2
read_trig
I32
framingerr
framingerr
~rd2 AND rd1
I14
read_trig
read
CFD2
I8
rd1
CFD2
I13
clkx16
reset
rxdatardyb
I39
rxidle_trig
CAND2
read_trig
I38
rxdatardy
rxdatardyb
rxdatardy
rxstop
reset
clkx16

clkgen_tx

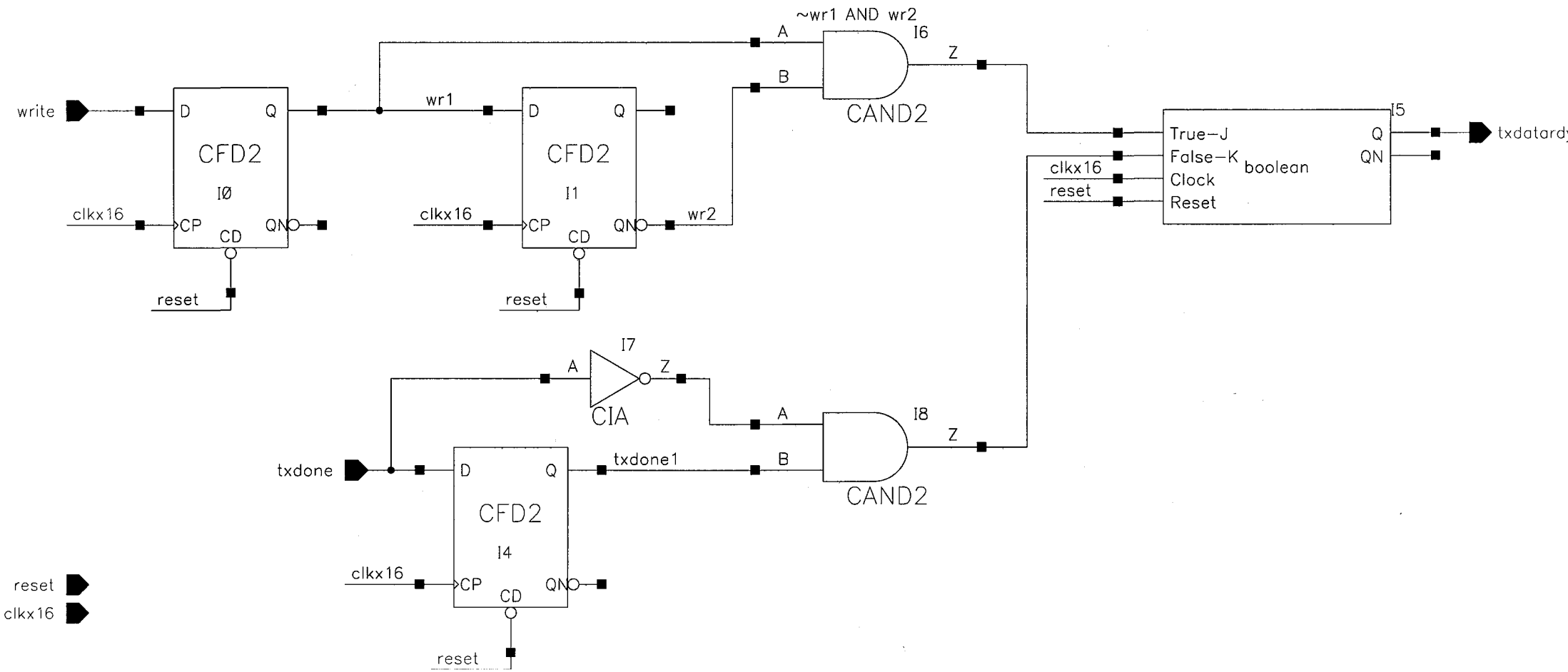

clkx16
reset
Clock
Reset
count_3
Q<2:0>
I7
cnt<2:0>
cnt<0>
cnt<1>
cnt<2>
A
B
C
I1
CNOR3
Z
txclkb
A
B
I8
CAND2
Z
A
txclk
B
I10
CAND2
Z
clkx16
reset
True-J
False-K
Clock
Reset
boolean
I9
Q
QN
txclk
txclkb
txclk

txshift


txdatardy
A
I13
Z
init
txdone
B
CAND2
txtag
txreg<7>
txreg<6>
txreg<5>
A
B
C
D
I2Ø
Z
CNOR4
A
I22
Z
txdone
txdone
B
CAND2
txreg<4>
txreg<3>
txreg<2>
txreg<1>
txreg<Ø>
A
B
C
D
E
I21
Z
CNOR5


txtag
init
D
Q
txtag
CFD2
I9
txclk
CP
CD
QN
reset
txtag,txreg<7:1>
A
CM21I1
I<7:Ø>
Z
txhold<7:Ø>
B
S
init
I7<7:Ø>
A
Z
CIA
txreg <7:Ø>
D
Q
txreg<7:Ø>
CFD2
IØ<7:Ø>
txclk
CP
CD
QN
reset
txhold<7:Ø>
txclk
reset
reset
init
A
I19
Z
CIA
A
B
I18
Z
CAND2
txreg<Ø>
txdone
A
B
I1Ø
Z
COR2
D
SD
Q
tx
CFD3
I16
start
txclk
CP
CD
QN

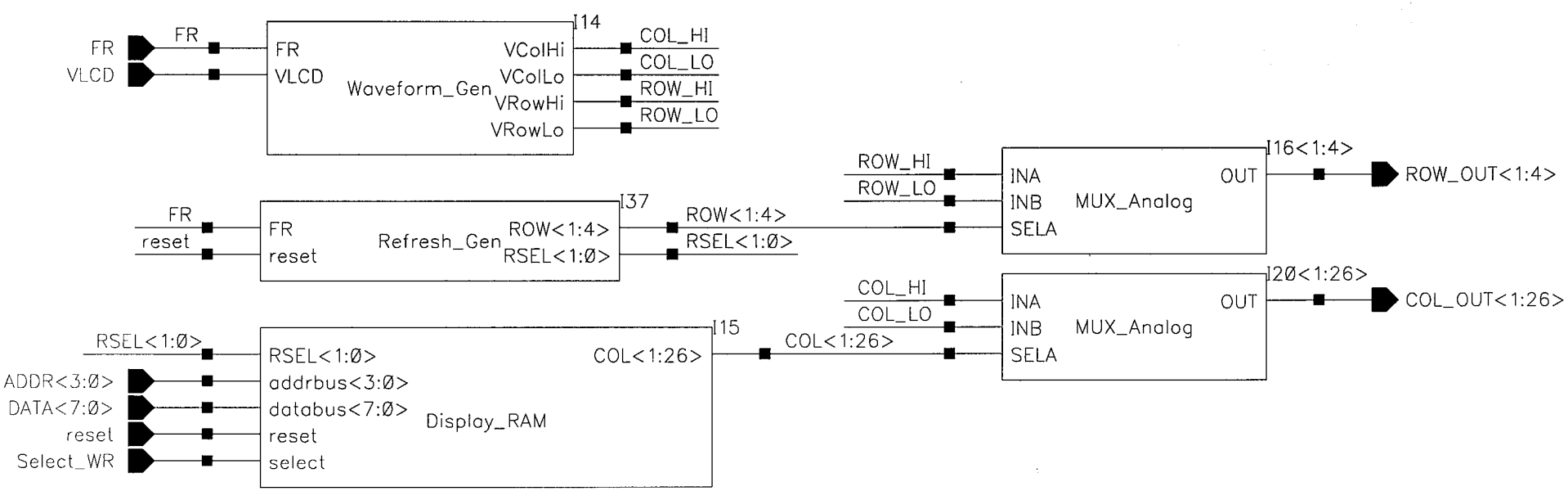
txio


~wr1 AND wr2
A
I6
Z
B
CAND2
write
D
Q
CFD2
I0
clkx16
CP
CD
QN
reset
wr1
D
Q
CFD2
I1
clkx16
CP
CD
QN
wr2
reset
True-J
False-K
Clock
Reset
boolean
I5
Q
QN
txdatardy
clkx16
reset
I7
A
Z
CIA
txdone
D
Q
txdone1
CFD2
I4
clkx16
CP
CD
QN
reset
A
I8
Z
B
CAND2
reset
clkx16

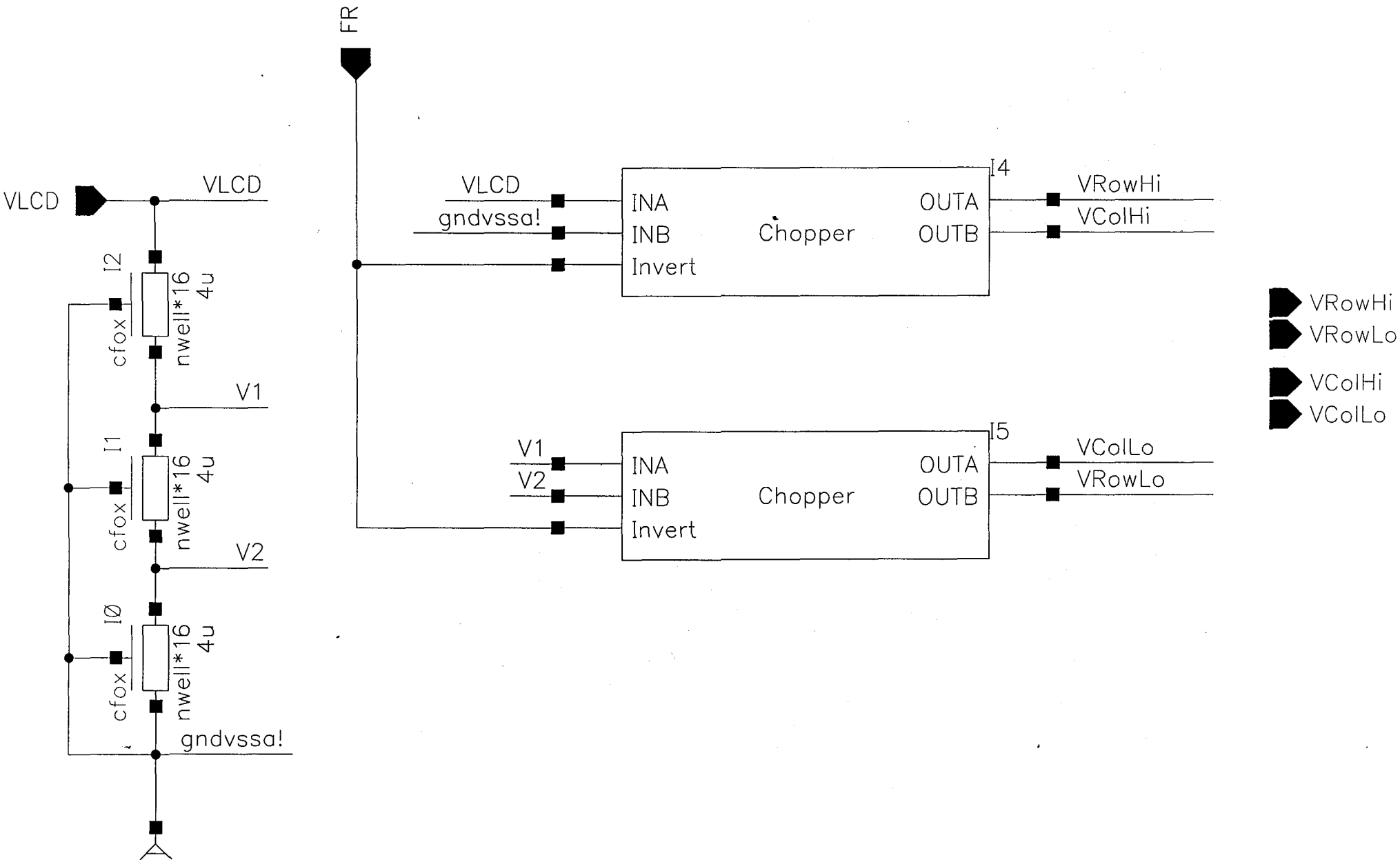
26.6. Shema LCD gonilnika

(na naslednjih straneh)

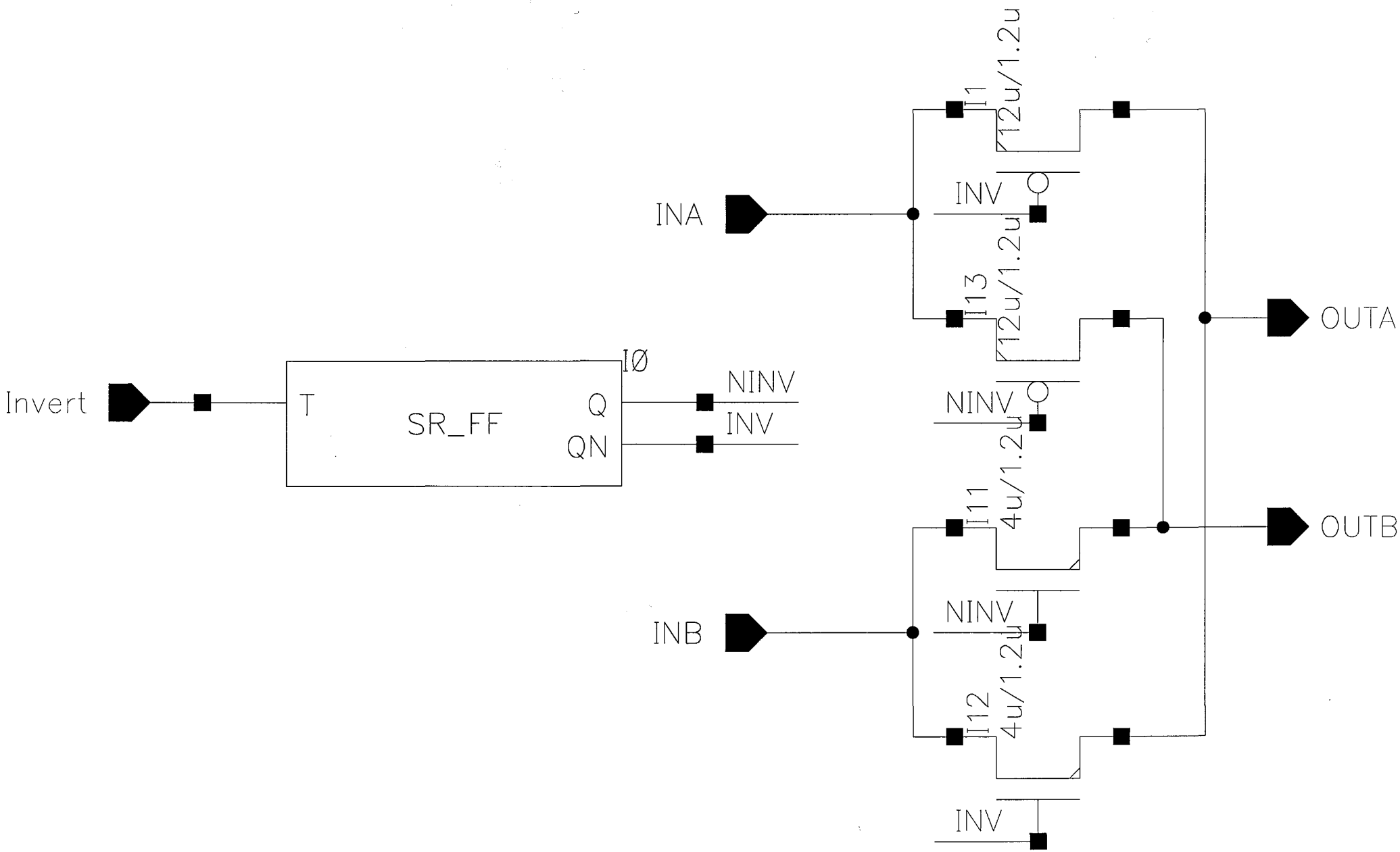
LCD_driver


FR
VLCD
FR
FR
VLCD
Waveform_Gen
I14
VColHi
VColLo
VRowHi
VRowLo
COL_HI
COL_LO
ROW_HI
ROW_LO
FR
reset
FR
reset
Refresh_Gen
I37
ROW<1:4>
RSEL<1:0>
ROW<1:4>
RSEL<1:0>
ROW_HI
ROW_LO
INA
INB
SELA
MUX_Analog
OUT
I16<1:4>
ROW_OUT<1:4>
COL_HI
COL_LO
INA
INB
SELA
MUX_Analog
OUT
I20<1:26>
COL_OUT<1:26>
RSEL<1:0>
ADDR<3:0>
DATA<7:0>
reset
Select_WR
RSEL<1:0>
addrbus<3:0>
databus<7:0>
reset
select
Display_RAM
COL<1:26>
I15
COL<1:26>

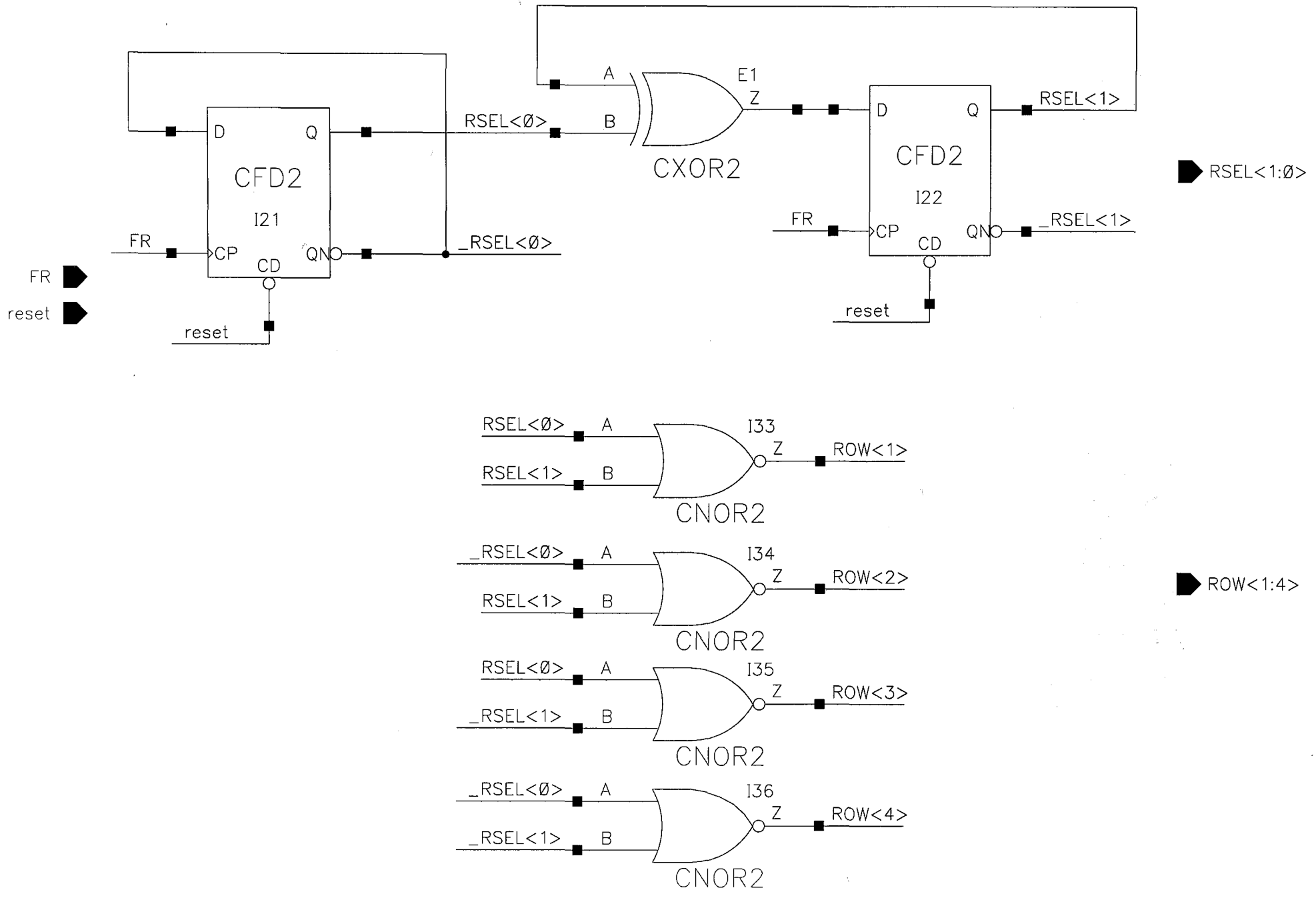
Waveform_gen



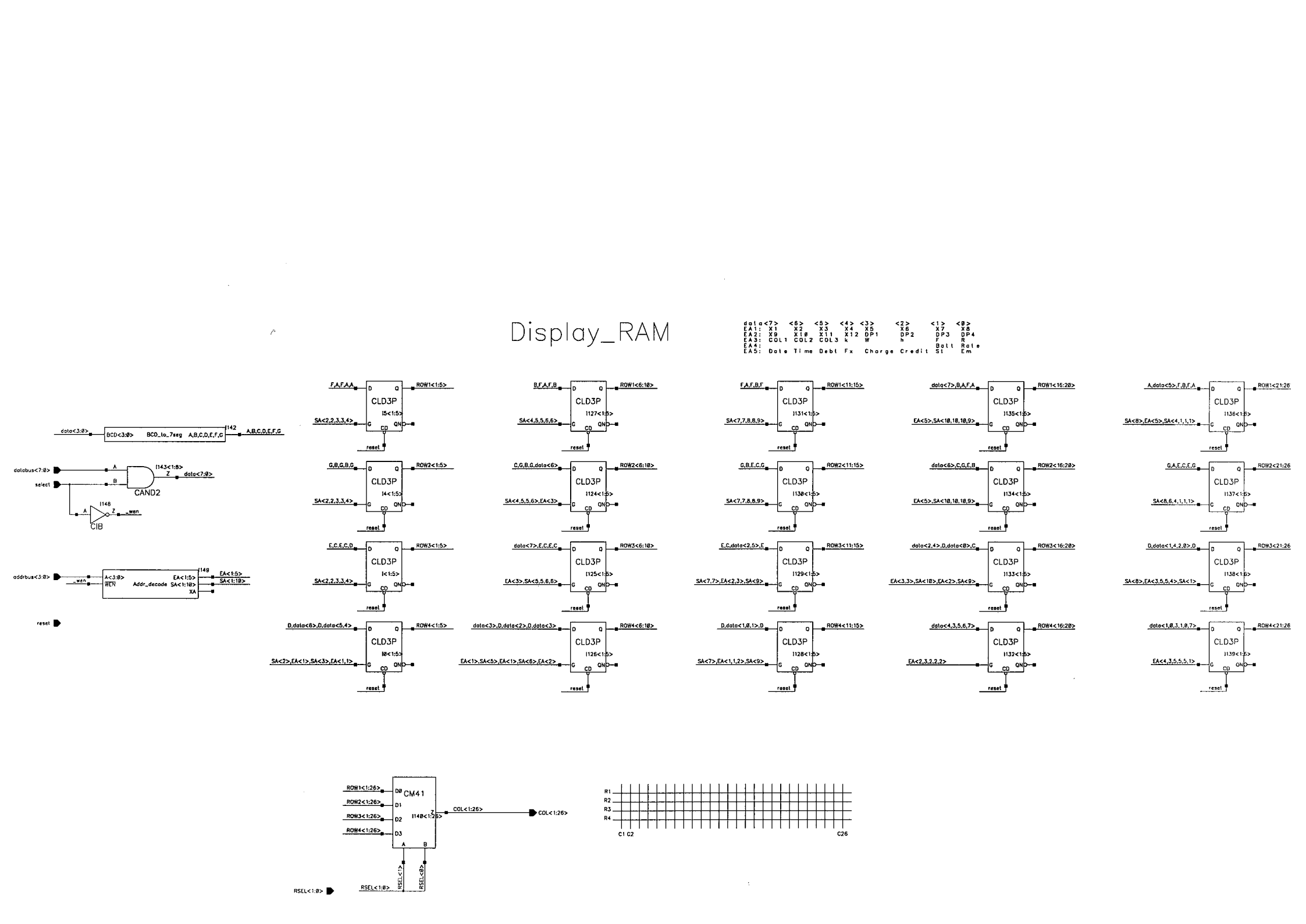
Chopper


I1
12u/1.2u
INV
INA
I13
12u/1.2u
OUTA
IØ
NINV
Invert
T
SR_FF
Q
INV
QN
NINV
I11
4u/1.2u
OUTB
NINV
INB
I12
4u/1.2u
INV

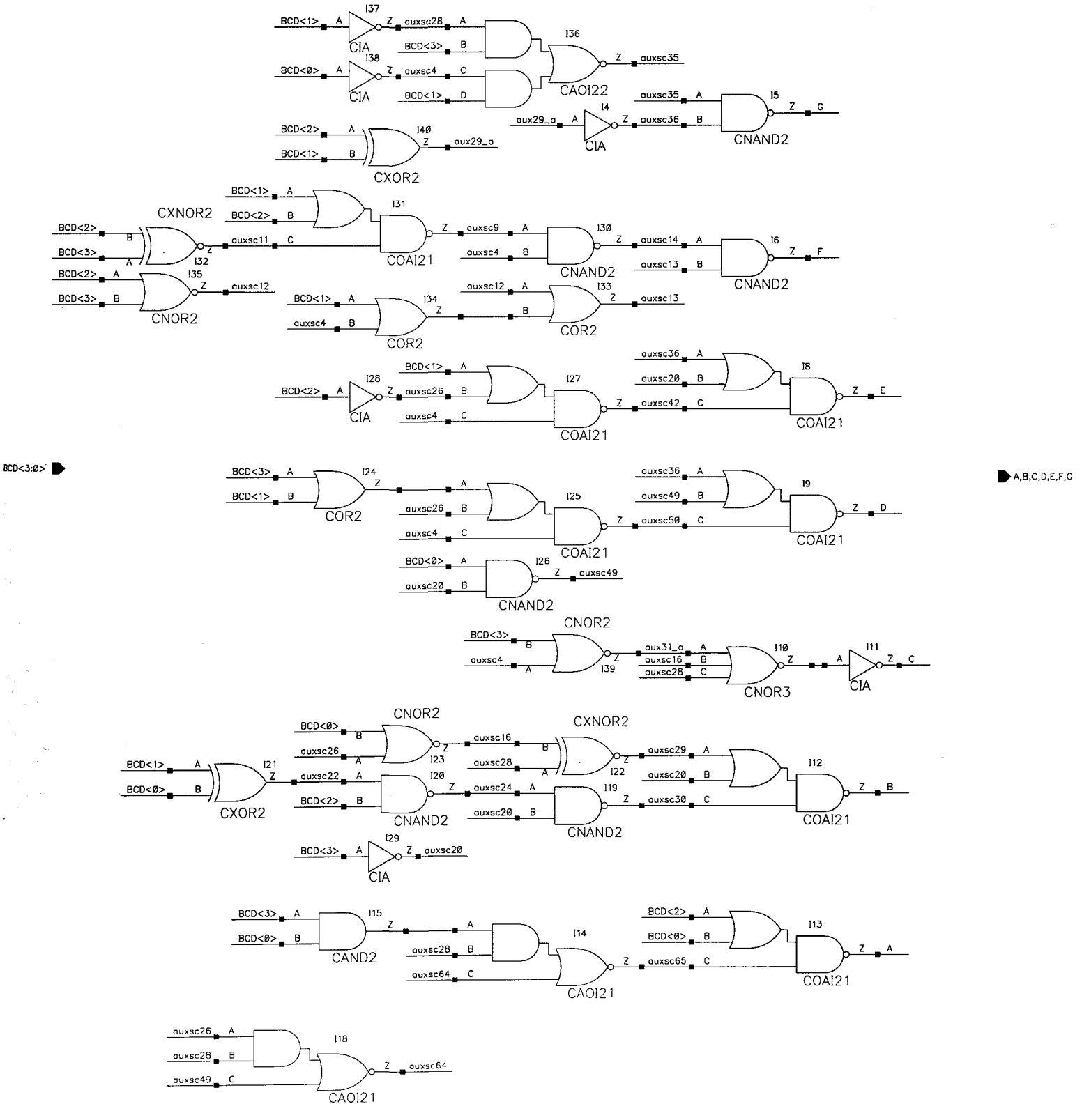
Refresh_gen


FR
reset
D
Q
CFD2
I21
FR
CP
CD
QN
reset
_RSEL<Ø>
RSEL<Ø>
A
B
E1
Z
CXOR2
D
Q
CFD2
I22
FR
CP
CD
QN
reset
RSEL<1>
_RSEL<1>
RSEL<1:Ø>
RSEL<Ø>
A
RSEL<1>
B
I33
Z
ROW<1>
CNOR2
_RSEL<Ø>
A
RSEL<1>
B
I34
Z
ROW<2>
CNOR2
RSEL<Ø>
A
_RSEL<1>
B
I35
Z
ROW<3>
CNOR2
_RSEL<Ø>
A
_RSEL<1>
B
I36
Z
ROW<4>
CNOR2
ROW<1:4>

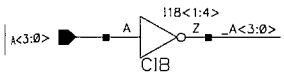
Display_RAM


data<7> <6> <5> <4> <3> <2> <1> <0>
EA1: X1 X2 X3 X4 X5 X6 X7 X8
EA2: X9 X10 X11 X12 DP1 DP2 DP3 DP4
EA3: COL1 COL2 COL3 k W h r R
EA4: Batt Rate
EA5: Date Time Debt Fx Charge Credit St Em
data<3:0>
BCD<3:0>
BCD_to_7seg
A,B,C,D,E,F,G
databus<7:0>
select
CAND2
CIB
addrbus<3:0>
Addr_decode
reset
CLD3P
CM41
RSEL<1:0>
R1
R2
R3
R4
C1 C2
C26

BCD_to_7seg


BCD<3:0>
A,B,C,D,E,F,G
I37
CIA
I38
CIA
I36
CAOI22
I5
CNAND2
I4
CIA
I40
CXOR2
CXNOR2
I31
COAI21
I32
I30
CNAND2
I6
CNAND2
I35
CNOR2
I34
COR2
I33
COR2
I28
CIA
I27
COAI21
I8
COAI21
I24
COR2
I25
COAI21
I9
COAI21
I26
CNAND2
CNOR2
I39
CNOR3
I10
I11
CIA
CNOR2
I23
CXNOR2
I22
I21
CXOR2
I20
CNAND2
I19
CNAND2
I12
COAI21
I29
CIA
I15
CAND2
I14
CAOI21
I13
COAI21
I18
CAOI21

Addr_decode



_WEN = 0

0000 SA<1> = 1

....

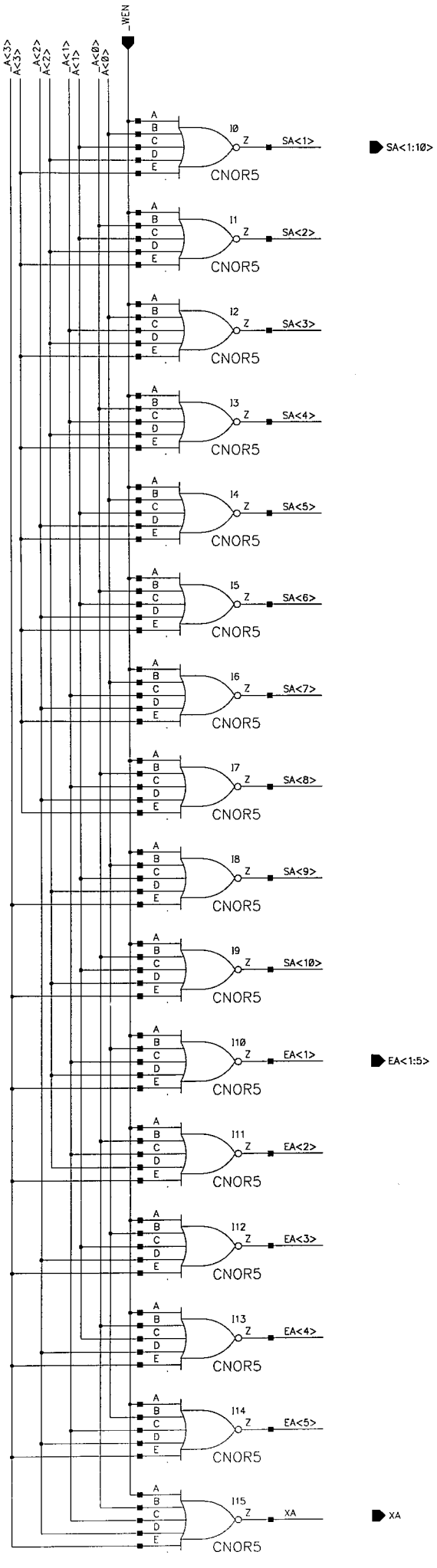
1010 SA<10> = 1

1011 EA<1> = 1

....

1110 EA<5> = 1

1111 XA = 1



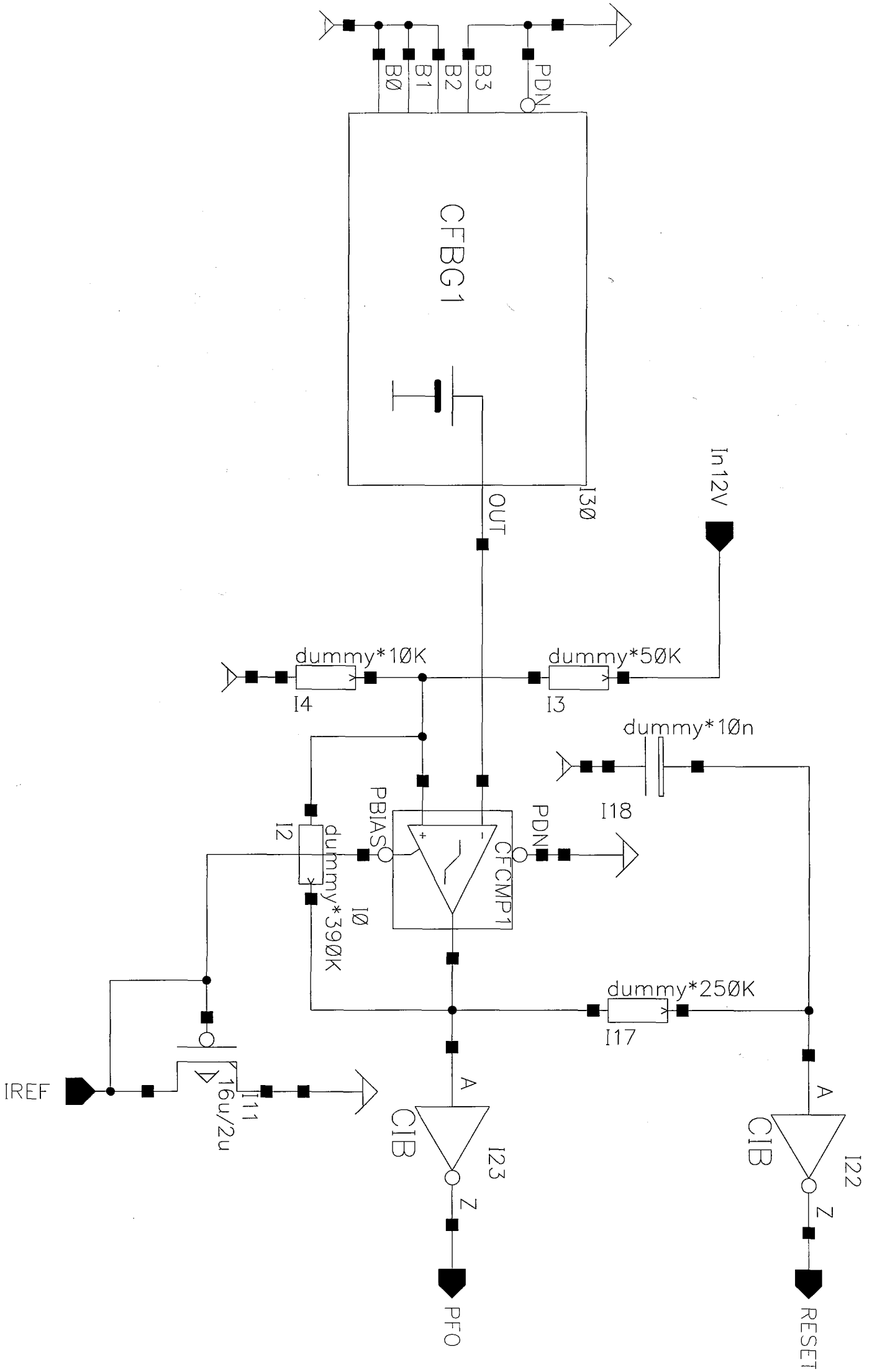
SA<1:10>

EA<1:5>

XA

26.7. Shema nadzora napajanja

(na naslednjih straneh)


B0
B1
B2
B3
PDN
CFBG1
I30
OUT
In12V
dummy*10K
I4
dummy*50K
I3
dummy*10n
I18
PBIAS
PDN
I2
dummy*390K
I0
CFCMP1
dummy*250K
I17
IREF
16u/2u
I11
A
CIB
I23
Z
PFO
A
CIB
I22
Z
RESET

27. Seznam uporabljene literature

1. R. K. Gupta: Co-Synhesis of Hardware and Software for Digital Embedded Systems, Kluwer Academic Publishers, 1995, ISBN 0-79-239613-8

2. Charles H. Roth, Jr.: Fundamentals of Logic Design, West Publishing Company, 1992, ISBN 0-314-92218-0

3. Miloš Ercegovac, Tomas Lang, Jaime H. Moreno: Introduction to Digital Systems, John Wiley and Sons, 1999, ISBN 0-471-52799-8

4. Neil H. E. Weste, Kamran Eshrigan: Principles of CMOS VLSI Design, Addison-Wesley, 1985, ISBN 0-201-08222-5

5. Masakazu Shoji: CMOS Digital Circuit Technology, Prentice-Hall International, Inc., ISBN 0-13-138843-6

6. Harry W. Fox: Master Op-Amp Applications Handbook, TAB Books, Inc., 1978, ISBN 0-8306-7856-5

7. Peter J. Ashenden: The VHDL Cookbook, University of Adelaide, 1990, http://www.ecsi.org/EARNEST/digest/VHDL_cookbook/

8. Open Verilog International Web Site, http://www.ovi.org

9. Interfacing the Serial /RS232 Port: http://www.beyondlogic.org/serial/serial.htm

10. Alcatel Microelectronics: Software Development kit MTC-8308 8bit uRISC, User's manual

11. Philips Semiconductors: The I^2C-bus and how to use it (including specifications), 1995

12. LCD driver & LCD data book, Hitachi corp, 1987

13. XILINX: The programmable Logic data book, 1998

14. Različni članki in viri na internetu, http://www.eetimes.com

28. Izjava

Izjavljam, da sem magistrsko delo samostojno izdelal pod vodstvom mentorja pdof. dr. Baldomirja Zajca. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.